Applications Handbook



Cypress Semiconductor, 3901 North First Street, San Jose, CA 95134 (408) 943–2600
Telex: 821032 CYPRESS SNJ UD, TWX: 910 997 0753, FAX: (408) 943–2741

How to Use This Book

This *Applications Handbook* is a learning tool for using Cypress devices. The application notes included here range from general product overview articles, such as "Understanding Dual-Port RAMs," to specific design examples. To summarize each application note, an abstract listing has been provided at the front of each section.

The general overviews describe product-family characteristics and explain some of the products-capabilities. These application notes appear at the beginning of this *Handbook*.

Next appear application examples that show how to use specific Cypress devices in the context of real designs. The application examples are organized by product type (e.g., PROMs or CPLDs). Within each product type examples are arranged by product number, using the product that is the article's primary focus.

Although your specific application might not appear explicitly in an application note, the design examples can still be useful to you. If the design example is similar to your application, you might be able to adapt the hardware or software to your design easily. Many of the application notes provide PLD software code for design tools from a variety of vendors, so that you can copy the code and use it as a skeleton for your own PLD designs. Even if none of the examples relate directly to your design, they can stimulate new ideas by showing features or applications that might not have occurred to you. The information can also significantly reduce the learning curve normally associated with unfamiliar ICs.

Published April 1994

© Cypress Semiconductor Corporation, 1994. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress Semiconductor Corporation product. Nor does it convey or imply any license under patent or other rights. Cypress Semiconductor does not authorize its products for use as critical components in life-support systems where a malfunction or failure of the product may reasonably be expected to result in significant injury to the user. The inclusion of Cypress Semiconductor products in life-support systems applications implies that the manufacturer assumes all risk of such use and in so doing indemnifies Cypress Semiconductor against all damages.

Contents

General Information

System Design Considerations When Using Cypress CMOS Circuits 1–1
Protection, Decoupling, and Filtering of Cypress CMOS Circuits 1–30
Power Characteristics of Cypress Products 1–34
Input/Output Characteristics of Cypress Circuits 1–49

PROM

Introduction to the State Machine PROM 2–1
Generating PROM Programming Files 2–19
Introduction to the CY7C270 Processor-Intelligent PROM 2–32
Interfacing the CY7C276 High-Speed PROM to the AT&T, AD, Motorola, and TI DSPs 2–51
Designing Custom ALUs and Multipliers with PROMs 2–59

UltraLogic/PLDs

Are Your PLDs Metastable? 3–1
State Machine Design Considerations and Methodologies 3–27
Designing with the CY7C335 and *Warp2* VHDL Compiler 3–73
The FLASH370 Family Of CPLDs and Designing with *Warp2* 3–102
Implementing a Reframe Controller for the CY7B933 HOTLink Receiver in a CY7C371 CPLD 3–121
Architectures and Technologies for FPGAs 3–137
Designing with FPGAs
An Introduction to Cypress's 380 Family of FPGAs and the *Warp3* Design Tool 3–149
CY7C380 Family Quick Power Calculator 3–172
Using Scan Mode on pASIC380 For In-Circuit Testing 3–177
Getting Started Converting .ABL Files to VHDL 3–180
Top-Down Design Methodology With VHDL
(Designing an Interrupt Controller) 3–207

Abel–HDL vs. IEEE–1076 VHDL 3–217
VHDL Techniques for Optimal Design Fitting 3–231
Describing State Machines with *Warp2* VHDL 3–236
Using Hierarchical VHDL Design 3–269

Specialty Memories

Understanding Dual-Port RAMs 4–1
Understanding Large FIFOs 4–19
Understanding Clocked FIFOs 4–29
FIFO Dipstick Using *Warp2* VHDL and the CY7C371 4–39
CY7C42X/46X FIFO Interface to the CY7B923 (HOTLink) 4–46
Interfacing the CY7B923 and CY7B933 (HOTLink) to a Clocked FIFO 4–49
Interfacing the CY7B923 and CY7B933 (HOTLink) to a Wide Data Clocked FIFO 4–57

Data Communications

HOTLink Design Considerations 5–1
HOTLink Built-In Self-Test (BIST) 5–62
Upgrade Your TAXI–275 with HOTLink 5–79
Replace Your TAXI–125 and TAXI–175 5–92
Parallel Cyclic Redundancy Check (CRC) for HOTLink 5–105
Fibre Channel Level 2 Design Considerations 5–111
Using High-Speed Serial Links to Supplement Parallel Data Buses 5–128
Using the CY7B923 as an ECL Clock Source 5–135
CY9266 HOTLink Demo Board User's Guide 5–141
HOTLink CY7B933 RDY Pin Description 5–208
HOTLink CY7B9331 OLC Receiver 5–214
Interfacing with the SST 5–221

Timing Products

Everything You Need to Know About CY7B991/CY7B992 (RoboClock) But Were Afraid to Ask 6–1
Innovative RoboClock Application 6–41
Using the CY7B991 with the 50-MHz 486 Cache Module and the 40-MHz R3000 6–53
CY7B991 and CY7B992 (RoboClock) Test Mode 6–77
Crystal Oscillator Topics 6–81
ECL Outputs 6–83

Bus Products

Features of the VIC068A VMEbus Interface Controller 7–1
Interfacing the VIC068A to the MC68020 7–6
Using VIC068A on a Board Without a Microprocessor 7–13
Connecting the Cypress VIC068/VAC068 to the TI TMS320C40: A Prototype Design 7–31
Software Considerations for the VIC64 7–69
VIC64 to Motorola 68040 Interface 7–84
Interfacing the CY7C611A with the VIC64 7–125
Using the CY7C964 with VIC 7–138

Modules

DRAM Accelerator: Set-Up for Basic Operation 8–1
DRAM Configuration and Diagnostics 8–7
DRAM Interfacing 8–13
CYM7232/7264 DRAM Accelerator: Mixing 5-Volt and 3.3-Volt DRAMs 8–32

CAM-based Products

Introduction to SmartCAM 9–1
SmartCAM Hardware Design Considerations 9–12
Pattern Matching with the SmartCAM 9–21

Sales Addresses **A–1**

Glossary **G–1**

Index **I–1**

General Information – 1



General Information Section Contents and Abstracts

System Design Considerations When Using Cypress CMOS Circuits **1–1**

This application note describes factors to consider when designing a digital system using high-performance CMOS integrated circuits. A formula is derived that enables the designer to predict when a trace on a PCB may become a transmission line. A simplified transmission line analysis is presented that eliminates the $j\omega t$ phase terms from the classical transmission line equations. Step function responses and pulse responses are tabulated for various line terminations. Various types of transmission lines and types of terminations are presented and analyzed. An analysis of an unterminated line is performed to illustrate the procedure.

Protection, Decoupling, and Filtering of Cypress CMOS Circuits **1–30**

This application note explains how to protect your CMOS circuits using an inexpensive zener diode. It also explains how to calculate the value of the decoupling capacitor for your integrated circuits and why the decoupling capacitor does not function well as a filtering capacitor. A capacitor impedance versus frequency curve is presented that shows how capacitor size is related to its series resonance frequency. The Fourier Transform of a periodic pulse is presented in order to show how high-frequency noise is generated.

Power Characteristics of Cypress Products **1–34**

A simple model is developed that can be used to predict the supply current versus frequency characteristic curve for Cypress products. Tables that contain the values of the relevant parameters for the model are presented and a sample set of calculations are made to illustrate the procedure.

Input/Output Characteristics of Cypress Circuits **1–49**

This application note presents, explains, and analyzes the Input/Output characteristics of the integrated circuits manufactured by Cypress Semiconductor. A brief history of integrated circuits, beginning with RTL Micrologic and ending with CMOS, is presented. Other topics discussed are TTL levels, CMOS levels, ESD protection, bias generator, I/O clamping diodes, latch-up, and ECL transfer characteristics.

System Design Considerations When Using Cypress CMOS Circuits

This application note describes some factors to consider when either designing new systems using Cypress high-performance CMOS integrated circuits or when using Cypress products to replace bipolar or NMOS circuits in existing systems. The two major areas of concern are device input sensitivity and transmission line effects due to impedance mismatching between the source and load.

To achieve maximum performance when using Cypress CMOS ICs, pay attention to the placement of the components on the printed circuit board (PCB); the routing of the metal traces that interconnect the components; the layout and decoupling of the power distribution system on the PCB; and perhaps most important of all, the impedance matching of some traces between the source and the loads. The latter traces must, under certain conditions, be analyzed as transmission lines. The most critical traces are those of clocks, write strobes on SRAMs and FIFOs, output enables, and chip enables.

Replacing Bipolar or NMOS ICs

Cypress CMOS ICs are designed to replace both bipolar ICs and NMOS products and to achieve equal or better performance at one-third (or less) the power of the components they replace.

When high-performance Cypress CMOS circuits replace either bipolar or NMOS circuits in existing sockets, be aware of conditions in the existing system that could cause the Cypress ICs to behave in unexpected ways. These conditions fall into two general categories: device input sensitivity and sensitivity to reflected voltages.

Input Sensitivity

High-performance products, by definition, require less energy at their inputs to change state than low- or medium-performance products.

Unlike a bipolar transistor, which is a current-sensing device, a MOS transistor is a voltage-sensing device. In fact, a MOS circuit design parameter called K' is analogous to the gm of a vacuum tube and is inversely proportional to the gate oxide thickness.

Thin gate oxides, which are required to achieve the desired performance, result in highly sensitive inputs. These inputs require very little energy at or above the device input-voltage threshold (approximately 1.5V at 25°C) to be detected. CMOS products may detect high-frequency signals to which bipolar devices may not respond.

MOS transistors also have extremely high input impedances (5 to 10 MΩ), which make the transistors' gate inputs analogous to the input of a high-gain amplifier or an RF antenna. In contrast, because bipolar ICs have input impedances of 1000Ω or less, these devices require much more energy to change state than do MOS ICs. In fact, a typical Cypress IC requires less that 10 picojoules of energy to change state. Thus, when Cypress CMOS ICs replace bipolar or NMOS ICs in existing systems, the CMOS ICs might respond to pulses of energy in the system that are not detected by the bipolar or NMOS products.

Reflected Voltages

Cypress CMOS ICs have very high input impedances and—to achieve TTL compatibility and drive capacitive loads—low output impedances. The im-

pedance mismatch due to low-impedance outputs driving high-impedance inputs might cause unwanted voltage reflections and ringing under certain conditions. This behavior could result in less-than-optimum system operation.

When the impedance mismatch is very large, a nearly equal and opposite negative pulse reflects back from the load to the source when the line's electrical length (PCB trace) is greater than

$$l = \frac{t_r}{2t_{pd}} \qquad \text{Eq. 1}$$

where t_r is the rise time of the signal at the source, and t_{pd} is the one-way propagation delay of the line per unit length.

The classical way of stating the condition for a voltage reflection to occur is that it will occur if the signal rise time is less than or equal to the round-trip (two-way) propagation delay of the line.

Input clamping diodes to ground were added to bipolar IC families (e.g., TTL, AS, LS, ALS, FAST) when the circuit designers decided that the fast rise and fall times of the outputs could cause voltage reflections. The clamping diodes to V_{CC} are inherent in the junction isolation process. For a more detailed explanation, see "Input/Output Characteristics of Cypress Products."

Historically, as circuit performance improved, the output rise and fall times of the bipolar circuits decreased to the point where voltage reflections began to occur (even for short traces) when an impedance mismatch existed between the line and the load. Most users, however, were unaware of these reflections because they were suppressed by the diodes' clamping action.

Conventional CMOS processing results in PN junction diodes, which adversely affect the ESD (electrostatic discharge) protection circuitry at each input pin and cause an increased susceptibility to latch-up. In addition, when the input pin is negative enough to forward bias the input clamping diodes, electrons are injected into the substrate. When a sufficient number of electrons are injected, the resulting current can disturb internal nodes, causing soft errors at the system level.

To eliminate the prospect of having this problem, all Cypress CMOS products use a substrate bias generator. The substrate is maintained at a negative 3V potential, so the substrate diodes cannot be forward biased unless the voltage at the input pin becomes a diode drop more negative than −3V. (See *Figure 9* in "Input/Output Characteristics of Cypress Products" for a schematic of the input protection circuit used in all Cypress CMOS products.) To the systems designer, this translates to approximately five times (3.8V divided by 0.8V =4.75) the negative undershoot safety margin for Cypress CMOS integrated circuits versus those that do not use a bias generator.

Voltage reflections should be eliminated by using impedance matching techniques and passive components that dissipate excess energy before it can cause soft errors. Crosstalk should be reduced to acceptable levels by careful PCB layout and attention to details.

Crosstalk

The rise and fall times of the waveforms generated by Cypress CMOS circuit outputs are 2 to 4 ns between levels of 0.4 and 4V. The fast transition times and the large voltage swings could cause capacitive and inductive coupling (crosstalk) between signals if insufficient attention is paid to PCB layout.

Crosstalk is reduced by avoiding running PCB traces parallel to each other. If this is not possible, run ground traces between signal traces.

In synchronous systems, the worst time for the crosstalk to occur is during the clock edge that samples the data. In most systems it is sufficient to isolate the clock, chip select, output enable, and write and read control lines from each other and from data and address lines so that the signals do not cause coupling to each other or to the data lines.

It is standard practice to use ground or power planes between signal layers on multilayered PCBs to reduce crosstalk. The capacitance of these isolation planes increases the propagation delay of the signals on the signal layers, but this drawback is more than compensated for by the isolation the planes provide.

The Theory of Transmission Lines

A connection (trace) on a PCB should be considered as a transmission line if the wavelength of the applied frequency is short compared to the line length. If the wavelength of the applied frequency is long compared to the length of the line, conventional circuit analysis can be used.

In practice, transmission lines on PCBs are designed to be as nearly lossless as possible. This simplifies the mathematics required for their analysis, compared to a lossy (resistive) line.

Ideally, all signals between ICs travel over constant-impedance transmission lines that are terminated in their characteristic impedances at the load. In practice, this ideal situation is seldom achieved for a variety of reasons.

Perhaps the most basic reason is that the characteristic impedances of all real transmission lines are not constants, but present different impedances depending upon the frequency of the applied signal. For "classical" transmission lines driven by a single frequency signal source, the characteristic impedance is "more constant" than when the transmission line is driven by a square wave or a pulse.

According to Fourier series expansion, a square wave consists of an infinite set of discrete frequency components—the fundamental plus odd harmonics of decreasing amplitude. When the square wave propagates down a transmission line, the higher frequencies are attenuated more than the lower frequencies. Due to dispersion, the different frequencies do not travel at the same speed.

Dispersion indicates the dependence of phase velocity upon the applied frequency (see Reference 1 pg. 192). The result is that the square wave or pulse is distorted when the frequency components are added together at the load.

A second reason why practical transmission lines are not ideal is that they frequently have multiple loads. The loads may be distributed along the line at regular or irregular intervals or lumped together, as close as practical, at the end of the line. The signal-line reflections and ringing caused by impedance mismatches, non-uniform transmission line impedances, inductive leads, and non-ideal resistors could compromise the dynamic system noise margins and cause inadvertent switching.

One system design objective is to analyze the critical signal paths and design the interconnections such that adequate system noise margins are maintained. There will always be signal overshoot and undershoot. The objective is to accurately predict these effects, determine acceptable limits, and keep the undershoot and overshoot within the limits.

The Ideal Transmission Line

An equivalent circuit for a transmission line appears in *Figure 1*. The circuit consists of subsections of series resistance (R) and inductance (L) and parallel capacitance (C) and shunt admittance (G) or parallel resistance, R_p. For clarity and consistency, these parameters are defined per unit length. Multiply

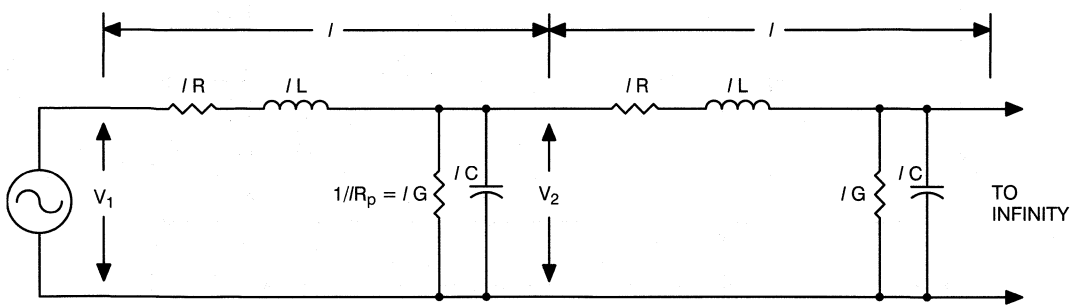


Figure 1. Transmission Line Model

the values of R, L, C, and R_p by the length of the subsection, *l*, to find the total value. The line is assumed to be infinitely long.

If the line of *Figure 1* is assumed to be lossless (R = 0, R_p = infinity), *Figure 1* reduces to *Figure 2*. A small series resistance has little effect upon the line's characteristic impedance. In practice and by design, the series resistance is quite small. For 1-ounce (0.0015-inch-thick), 1-mil-wide (0.010-inch) copper traces on G-10 glass epoxy PCBs, the trace resistance is between 0.5 and 0.3Ω per foot. 2-ounce copper has a resistance 50 percent lower than that of 1-ounce copper.

Input or Characteristic Impedance

To calculate the characteristic impedance (also called AC impedance or surge impedance) looking into terminals a-b of the circuit in *Figure 2*, use the following procedure.

Let Z_1 be the input impedance looking into terminals a-b,with Z_2 for terminals c-d, Z_3 for terminals e-f, etc. Z_1 is the series impedance of the first inductor (*l*L) in series with the parallel combination of Z_2 and the impedance of the capacitor (*l*C).

From AC theory:

$$X_L = j\omega lC \qquad \text{Eq. 2}$$

where X_L is the inductive reactance.

$$X_C + \frac{1}{j\omega lC} \qquad \text{Eq. 3}$$

where X_C is the capacitive reactance.

Then

$$Z_1 = X_L + \frac{Z_2 X_C}{Z_2 + X_C} \qquad \text{Eq. 4}$$

If the line is reasonably long, $Z_1 = Z_2 = Z_3$. Substituting $Z_1 = Z_2$ into *Equation 4* yields

$$Z_1 = X_L + \frac{Z_1 X_C}{Z_1 + X_C}$$

or

$$Z_1^2 - Z_1 X_L - X_C X_L = 0 \qquad \text{Eq. 5}$$

Substituting the expressions for X_C and X_L yields

$$Z_1^2 - j\omega lL = \frac{L}{C} \qquad \text{Eq. 6}$$

Equation 6 contains a complex component that is frequency dependent. The complex component can be eliminated by allowing *l* to become very small and by recognizing that the ratio L/C is constant and independent of *l* or ω:

$$Z_1 = \sqrt{L/C} \qquad \text{Eq. 7}$$

The AC input impedance of a purely reactive, uniform, lossless line is a resistance. This is true for AC or DC excitation.

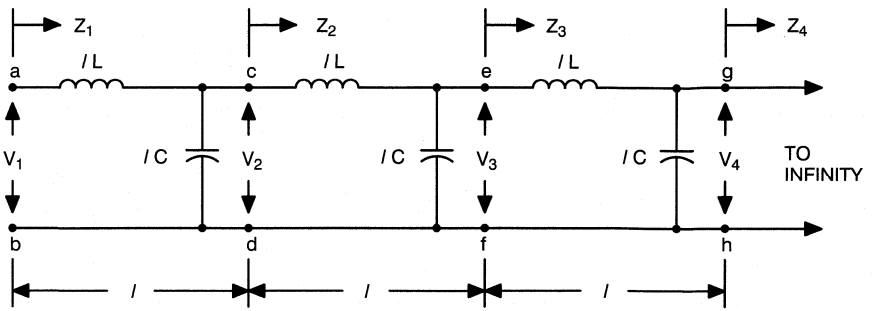


Figure 2. Ideal Transmission Line Model

Propagation Velocity and Delay

The propagation velocity (or phase velocity) of a sinusoid traveling on an ideal line (see Reference 1) is

$$\alpha = \frac{1}{\sqrt{L/C}} \qquad \text{Eq. 8}$$

The propagation delay for a lossless line is the reciprocal of the propagation velocity:

$$t_{pd} = \sqrt{LC} = Z_1 C \qquad \text{Eq. 9}$$

where L and C are once again the intrinsic line inductance and capacitance per unit length.

Adding additional stubs or loads to the line (see Reference 2 of this application note) increases the propagation delay by the factor

$$\sqrt{1 + C_D/C} \qquad \text{Eq. 10}$$

where C_D is the load capacitance.

Therefore, the propagation delay of a loaded line, T_{pdL}, is

$$t_{pdL} = t_{pd}\sqrt{1 + C_D/C} \qquad \text{Eq. 11}$$

This application note shows later that a transmission line's unloaded or intrinsic propagation delay is proportional to the square root of the dielectric constant of the medium surrounding or adjacent to the line. Propagation delay is not a function of the line's geometry.

The characteristic impedance of a capacitively loaded line decreases by the same factor that the propagation delay increases:

$$Z_1' = \frac{Z_1}{\sqrt{1 + C_D/C}} \qquad \text{Eq. 12}$$

Note that the capacitance per unit length must be multiplied by the line length, *l*, to calculate an equivalent lumped capacitance.

The Condition for Voltage Reflection

It is relatively straightforward to obtain a closed-form solution for a transmission line's maximum allowable length, which, if exceeded, might cause a voltage reflection. If the line is not terminated in its characteristic impedance, a reflection is guaranteed to occur. The reflection's amplitude depends on the amount of impedance mismatch between the line and the load and whether the rise time of the signal at the source equals or is greater (slower) than two times the propagation delay of the line.

The condition for a voltage reflection to occur is

$$L \geq \frac{t_r}{2t_{pdL}} \qquad \text{Eq. 13}$$

Solving for the loaded propagation delay yields

$$t_{pdL} = \frac{t_r}{2L} \qquad \text{Eq. 14}$$

However, the actual physical length of the line is

$$l = \frac{t_r}{t_{pd}} \qquad \text{Eq. 15}$$

The intrinsic capacitance of the line from *Equation 9* is

$$C_O = \frac{T_{pd}}{Z_O} \qquad \text{Eq. 16}$$

It is standard practice to use C_O to designate the intrinsic line capacitance, L_O the intrinsic line self inductance, and Z_O the intrinsic line characteristic impedance.

Substituting *Equations 14*, *15*, and *16* into *Equation 11* gives the relationship for the line length at which voltage reflections might occur. Two conditions must be present for voltage reflections to occur: the line must be long and there must be an impedance mismatch between the line and the load.

$$\frac{t_r}{2L} = t_{pd}\sqrt{1 + \frac{C_D}{\frac{t_r}{t_{pd}} \times \frac{t_{pd}}{Z_O}}} \qquad \text{Eq. 17}$$

Solving Equation 17 for the line length, L, yields

$$L = \frac{t_r}{2t_{pd}} \times \frac{1}{\sqrt{1 + \frac{C_D Z_O}{t_r}}} \qquad \text{Eq. 18}$$

Equation 18 is very useful to the system designer. It is generic and applies to all products irrespective of circuit type, logic family, or voltage levels. The equation allows you to estimate when a line requires termination, using variables you can easily determine.

When driving a distributed or non-lumped load, the signal's rise time depends on the source—not the load, as you might expect. The intrinsic, or unloaded, line propagation delay per unit length is a function of the dielectric constant and can be easily calculated. The intrinsic line characteristic impedance is a function of the dielectric constant and the PCB's physical construction or geometry and can also be calculated. Finally, you can estimate the equivalent (lumped) load capacitance by adding up the number of loads (device inputs) being driven and multiplying by 10 pF. For I/O pins, use 15 pF per pin.

Signal Transition Times

The standard Cypress 0.8μ (L drawn) CMOS process yields output buffers whose signals transition approximately 4V in 2 ns, or, have a slew rate of 2V per nanosecond. The rise time/fall time is 2 ns. Products fabricated using the Cypress BiCMOS process have the same rise times.

The Cypress ECL process yields products with 500-ps output signal rise times and fall times, or slew rates of 1V/0.5 ns = 2V per nanosecond. Internal signal slew rates are 10V per nanosecond, but only for short (usually less than 500 mV) voltage excursions. Thus, high-frequency noise is generated on chip, which you can eliminate by using 100- to 500-pF ceramic or mica filter capacitors between V_{CC} and ground.

The values in *Table 1* come from using *Equation 18* to calculate the line length at which voltage reflections may occur. The calculations assume a 50Ω intrinsic line characteristic impedance and that the PCB is multilayer, using stripline construction on G-10 glass epoxy material (dielectric constant of 5). These conditions result in an unloaded line propagation delay of 2.27 ns per foot.

Table 1. Line Length at Which a Voltage Reflection Occurs

t_r (ns)	C_D (pF)	L (inches)
2	10	4.73
2	20	4.32
2	40	3.74
2	80	3.05
1	10	2.16
1	20	1.87
1	40	1.53
1	80	1.18
0.5	10	0.93
0.5	20	0.76
0.5	40	0.59
0.5	80	0.44

Table 1 reveals that decreasing the source rise time from 2 to 0.5 ns (a factor of 4) decreases the line length at which a voltage reflection might occur by a factor of 5 (4.73 divided by 0.93 = 5.09) for the same load (10 pF) and intrinsic propagation delay (2.27 ns/ft.). A second observation is that for signals with rise times of 0.5 ns, all lines should be terminated.

Reflection Coefficients

Another attribute of the ideal transmission line, reflection coefficients, are not actually line characteristics. The line is treated as a circuit component, and reflection coefficients are defined that measure the impedance mismatches between the line and its source and the line and its load. The reason for defining and presenting the reflection coefficients becomes apparent later when it is shown that if the impedance mismatch is sufficiently large, either a negative or positive voltage might reflect back from the load to the source, and the voltage might either add to or subtract from the original signal. A mismatch between the source and line impedance may also cause a voltage reflection, which in turn reflects back to the load. Therefore, two reflection coefficients are defined.

For classical transmission lines driven by a single frequency source, the impedance mismatches cause standing waves. When pulses are transmitted and the source's output impedance changes depending upon whether a LOW-to-HIGH or a HIGH-to-LOW transition occurs, the analysis is complicated further.

You can use classical transmission line analysis—where pulses are represented by complex variables with exponentials—to calculate the voltages at the source and the load after several back and forth reflections. However, these complex equations tend to obscure what is physically happening.

Energy Considerations

Now consider the effects of driving the ideal transmission line with digital pulses and analyze the behavior of the line under various driving and loading conditions. The first task is to define the load and source reflection coefficients.

Figure 3 shows the circuit to be analyzed. The ideal transmission line of length l is driven by a digital source of internal resistance R_S and loaded with a resistive load R_L. The characteristic impedance of the line appears as a pure resistance,

$$Z_o = \sqrt{L/C} \qquad \text{Eq. 19}$$

to any excitation.

The ideal case is when $R_S = Z_O = R_L$. The maximum energy transfer from source to load occurs under this condition, and no reflections occur. Half the energy is dissipated in the source resistance, R_S, and the other half is dissipated in the load resistance, R_L (the line is lossless).

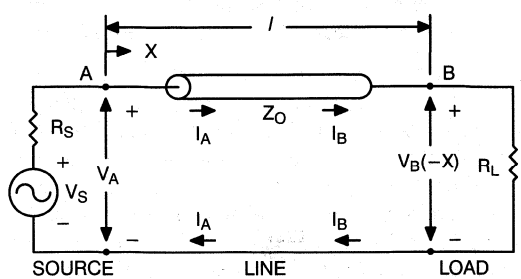


Figure 3. Ideal Transmission Line Loaded and Driven

If the load resistor is larger than the line's characteristic impedance, extra energy is available at the load and is reflected back to the source. This is called the underdamped condition, because the load underuses the energy available. If the load resistor is smaller than the line impedance, the load attempts to dissipate more energy than is available. Because this is not possible, a reflection occurs that signals the source to send more energy. This is called the overdamped condition. Both the underdamped and overdamped cases cause negative traveling waves, which cause standing waves if the excitation is sinusoidal. The condition $Z_O = R_L$ is called critically damped.

The safest termination condition, from a systems design viewpoint, is the slightly overdamped condition, because no energy is reflected back to the source.

Line Voltage for a Step Function

To determine the line voltage for a step function excitation, you apply a step function to the ideal line and analyze the behavior of the line under various loading conditions. The step function response is important because any pulse can be represented by the superposition of a positive step function and a negative step function, delayed in time with respect to each other. By proper superposition, you can predict the response of any line and load to any width pulse. The principle of superposition applies to all linear systems.

According to theory, the rise time of the signal driven by the source is not affected by the characteristics of the line. This has been substantiated in practice by using a special coaxially constructed reed relay that delivers a pulse of 18A into 50Ω with a rise time of 0.070 ns (see Reference 1).

The equation representing the voltage waveform going down the line (see *Figure 3*) as a function of distance and time is

$$V_L(X,t) = V_A(t)U(t - Xt_{pd}) \text{ for } t < T_O \qquad \text{Eq. 20}$$

$$V_A(t) = V_S(t)\left(\frac{Z_O}{Z_O + R_S}\right) \quad \text{Eq. 21}$$

where

V_A = the voltage at point A

X = the voltage at a point X on the line

l = the total line length

t_{pd} = the propagation delay of the line in nanoseconds per foot

$T_O = l\ t_{pd}$, or the one-way line propagation delay

U(t) = a unit step function occurring at x = 0

$V_S(t)$ = the source voltage

When the incident voltage reaches the end of the line, a reflected voltage, V′, occurs if R_L does not equal Z_O. The reflection coefficient at the load, ϱL, can be obtained by applying Ohm's Law.

The voltage at the load is $V_L + V_L'$, which must be equal to $(I_L + I_L')R_L$. But

$$I_L = \frac{V_L}{Z_O} \quad \text{Eq. 22}$$

and

$$I_L' = -\frac{V_L'}{Z_O} \quad \text{Eq. 23}$$

(The minus sign is due to I_L being negative; i.e., I_L is opposite to the current due to V_L.) Therefore,

$$V_B = V_L + V_L' = \left(\frac{V_L}{Z_O} - \frac{V_L'}{Z_O}\right)R_L \quad \text{Eq. 24}$$

By definition:

$$\varrho_L = \frac{\textit{reflected voltage}}{\textit{incident voltage}} = \frac{V_L'}{V_L} \quad \text{Eq. 25}$$

Solving for V_L'/V_L in *Equation 24* and substituting in the equation for ϱ_L yields

$$\varrho_L = \frac{R_L - Z_O}{R_L + Z_O} \quad \text{Eq. 26}$$

The reflection coefficient at the source is

$$\varrho_S = \frac{R_S - Z_O}{R_S + Z_O} \quad \text{Eq. 27}$$

Re-arranging *Equation 24* yields

$$V_B = V_L + V_L' = \left(1 + \frac{V_L'}{V_L}\right)V_L$$

$$= (1 + \varrho_L)V_L \quad \text{Eq. 28}$$

Equation 28 describes the voltage at the load (V_B) as the sum of an incident voltage (V_L) and a reflected voltage (ϱ_L V*l*) at time t = T_O. When $R_L = Z_O$, no voltage is reflected. When $R_L < Z_O$, the reflection coefficient at the load is negative; thus, the reflected voltage subtracts from the incident voltage, giving the load voltage. When $R_L > Z_O$, the reflection coefficient is positive; thus, the reflected voltage adds to the incident voltage, again giving the load voltage.

Note that the reflected voltage at the load has been defined as positive when traveling toward the source. This means that the corresponding current is negative, subtracting from the current driven by the source.

This piecewise analysis is cumbersome and can be tedious. However, it does provide an insight into what is physically happening and demonstrates that a complex problem can be solved by dividing it into a series of simpler problems. Also, eliminating the exponentials—which provide phase information in the classical transmission line equations—simplifies the mathematics. To use the piecewise method, you must do careful bookkeeping to combine the reflections at the proper time. This is quite straightforward, because a pulse travels with a constant velocity along an ideal or low-loss line, and the time delay between reflected pulses can be predicted.

The rules to keep in mind are that at any location and time the voltage or the current is the algebraic sum of the waves traveling in both directions. For example, two voltage waves of the same polarity and equal amplitudes, traveling in opposite directions, at a given location and time add together to yield a voltage of twice the amplitude of one wave. The same reasoning applies to all points of termination and discontinuities on the line. The total voltage or current is the algebraic sum of all the incident and reflected waves. Polarities must be observed. A

positive voltage reflection results in a negative current reflection and vice versa.

Step Function Response of the Ideal Line

Before examining reflections at the source due to mismatches between the source and line impedances, consider the behavior of the ideal line with various loads when driven by a step function. The circuit for analysis appears in *Figure 3*. *Figure 4* shows the voltage and current waveforms at point A (line input) and point B (the load) for various loads. (These values are drawn from Reference 1 pg. 158 – 159.) Note that $R_S = Z_O$ and that V_A at t = 0 equals $V_S/2$. This means that no impedance mismatch exists between the source and the line; thus, there is no reflection from the source at $t = 2\ T_O$. T_O is the one-way propagation delay of the line.

The time-domain response of the reactive loads are obtained by applying a step function to the LaPlace transform of the load and then taking the inverse transform.

Note that the reflection coefficient at the load is not the total reflection coefficient (a complex number) but represents only the real part of the load. The piecewise method eliminates the complex (jωt) terms by performing the bookkeeping involving the phase relationships, which the complex terms account for in classical transmission line analysis.

Note that for the open-circuit condition in *Figure 4b*, Z_L = infinity, so that $\varrho_L = +1$. The voltage is reflected from the load to the source (at amplitude $V_O = V_S/2$). Thus, at time $t = 2\ T_O$, the reflected voltage adds to the original voltage, $V_O = V_S/2$, to give a value of $2V_O = V_S$. While the voltage wave is traveling down to and back from the load, a current of

$$I_o = \frac{V_o}{Z_o} = \frac{V_s}{2}Z_o \qquad \text{Eq. 29}$$

exists. This current charges up the distributed line capacitance to the value V_S, then the current stops.

The waveforms at the source and load for the series RC termination shown in *Figure 4g* are of particular interest because this network dissipates no DC power; you can use this network to terminate a transmission line in its characteristic impedance at the input to a Cypress IC. *Figure 4h* represents the equivalent circuit of a Cypress IC's input. Combining both networks models a Cypress IC driven by a transmission line terminated in the line's characteristic impedance, when the values of R and C are properly chosen.

Reflections Due to Discontinuities

Figure 5 illustrates three types of common discontinuities found on transmission lines. Any change in the characteristic impedance of the line due to construction, connectors, loads, etc., causes a discontinuity, which causes a reflection that directs some energy back to the source. The amount of energy reflected back is determined by the discontinuity's reflection coefficient. Because discontinuities are usually small by design, most of the energy is transmitted to the load.

In general, a discontinuity has series inductance, shunt capacitance, and series resistance. An example is a via from a signal plane through a ground plane to a second signal plane in a multilayer PCB or module. IC sockets and other connectors can also cause discontinuities.

The Ideal Transmission Line's Pulse Response

Consider next the behavior of the ideal transmission line when driven by a pulse whose width is short compared to the line's electrical length—when the pulse width is less than the line's one-way propagation delay time, T_O.

Figure 6 shows another series of response waveforms for the circuit in *Figure 3*, this time for a pulse instead of a step (drawn from Reference 1 pg. 160 – 161). Note that $R_S = Z_O$ and that V_A at t = 0 equals $V_S/2$. This means that there is no impedance mismatch between the source and the line; thus, there is no reflection from the source at $t = 2\ T_O$.

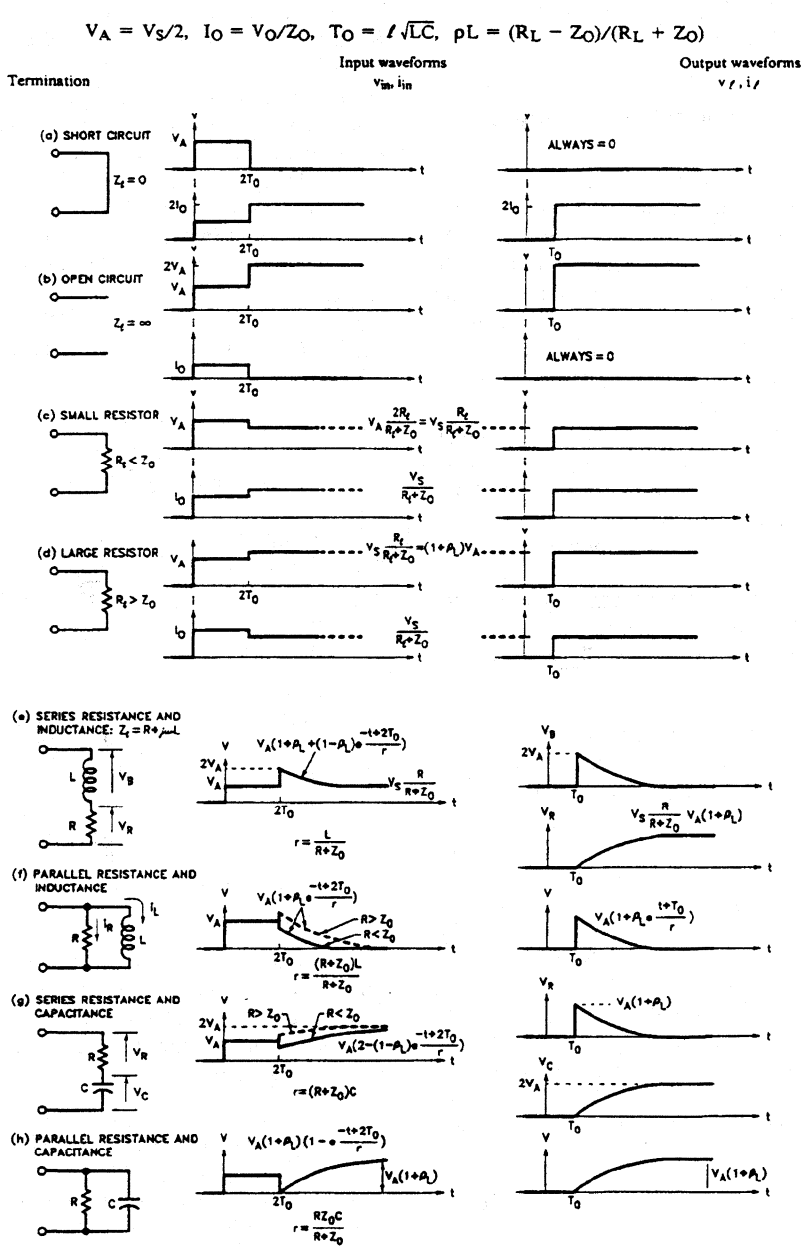


Figure 4. Step Function Response of Figure 3 for Various Terminations

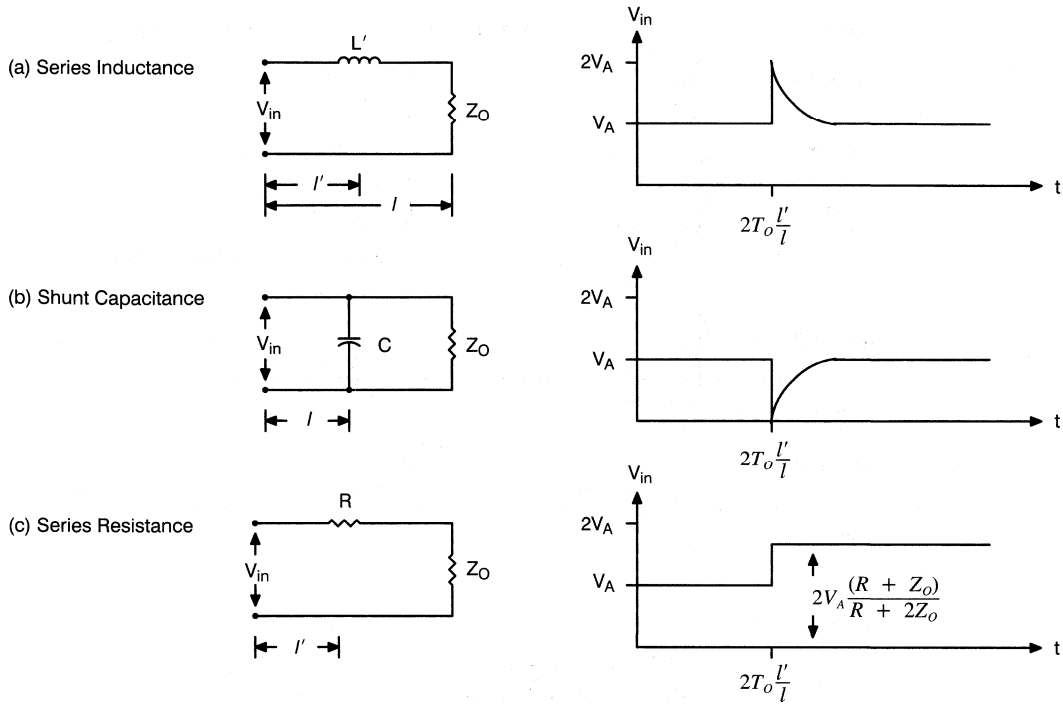


Figure 5. Reflections from Discontinuities with an Applied Step Function

Finite Rise Time Effects

Now consider the effects of step functions with finite rise times driving the ideal transmission line. During the rise time of a pulse, half the energy in the static electric field is converted into a traveling magnetic field and half remains as a static electric field to charge the line.

If the rise time is sufficiently short, the voltage at the load changes in discrete steps. The amplitude of the steps depends on the impedance mismatch, and the width of the steps depends on the line's two-way propagation delay.

As the rise time and/or the line gets shorter (smaller T_O), the result converges to the familiar RC time constant, where C is the static capacitance. All devices should be treated as transmission lines for transient analysis when an ideal step function is applied. However, as the rise time becomes longer and/or the traces shorter, the transmission line analysis reduces to conventional AC circuit analysis.

Reflections from Small Discontinuities

Figure 7 shows a pulse with a linear rise time and rounded edges driving the transmission line of *Figure 5a* and *Figure 5b*. The expressions for V_r are derived on pages 171 and 172 of Reference 1. The reflection caused by the small series inductance is useful for calculating the value of the inductor, L′, but little else.

The reflection caused by the small shunt capacitor is more interesting. If this capacitor is sufficiently large, it can cause a device connected to the transmission line to see a logic 0 instead of a logic 1.

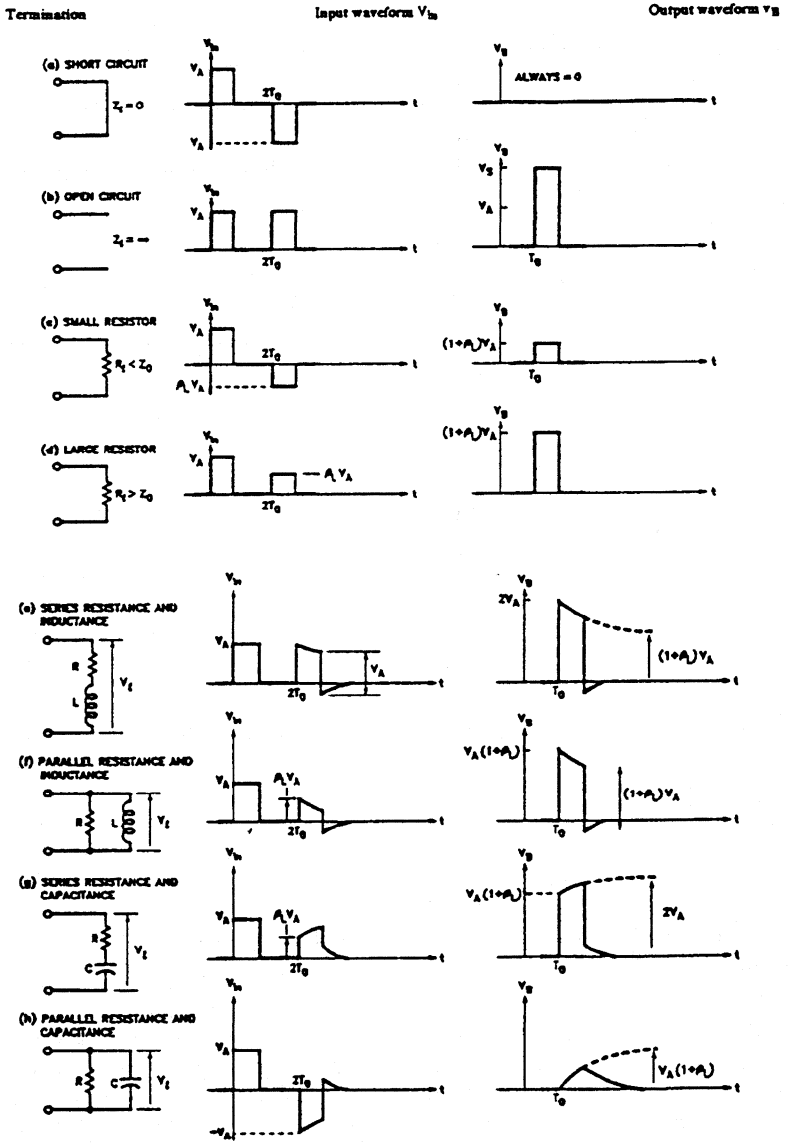


Figure 6. Pulse Response of Figure 3 for Various Terminations

$$V_A = V_S/2,\quad I_O = V_O/Z_o,\quad T_o = l\sqrt{LC},\quad \rho_L = \frac{(R_L - Z_o)}{(R_L + Z_o)}$$

The Effect of Rise Time on Waveforms

Next, consider the ideal line terminated in a resistance less than its characteristic impedance and driven by a step function with a linear rise time. The stimulus, the circuit, and the response appear in *Figure 8a*, *Figure 8b*, and *Figure 8c*, respectively. Once again, note that because the source resistance equals the line characteristic impedance, there are no reflections from the source.

The resulting waveforms are similar to those of *Figure 4c* when modified as shown in *Figure 8c*. The final value of the waveform must be the same as before (*Figure 4c*).

The resultant wave at the line input (V_{in}) is easily obtained by superposition of the applied wave and the reflected wave at the proper time. In *Figure 8*, because the step function's rise time is less than the line's two-way propagation delay, the input wave reaches its final value, $V_S/2$. At $t = 2\ T_O$, the reflected wave arrives back at the source and subtracts from the applied step function (the load reflection coefficient is negative). *Figure 9* illustrates waveforms for two relationships between the step function rise time and the propagation delay.

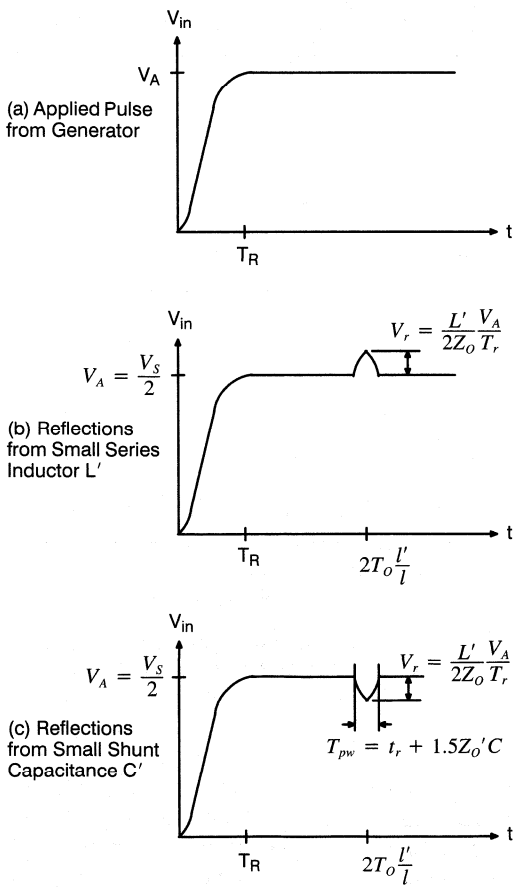


Figure 7. Reflections from Small Discontinuities with a Finite Rise Time Pulse

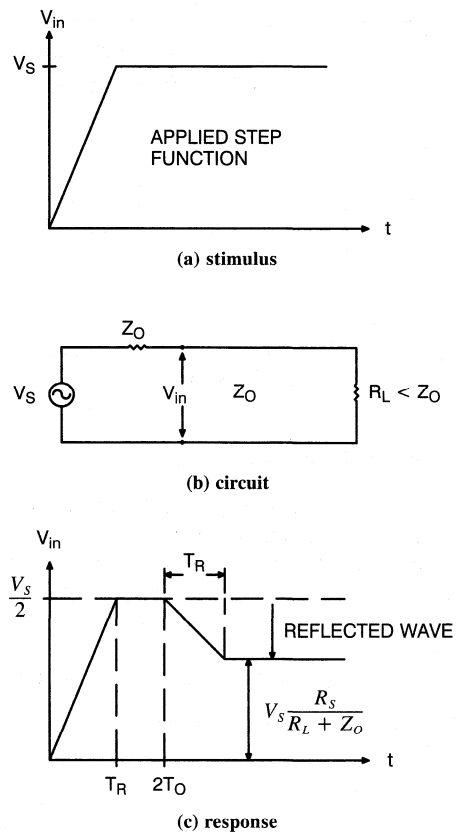


Figure 8. Effect of Rise Time on Response of Mismatched Line with $R_L < Z_O$

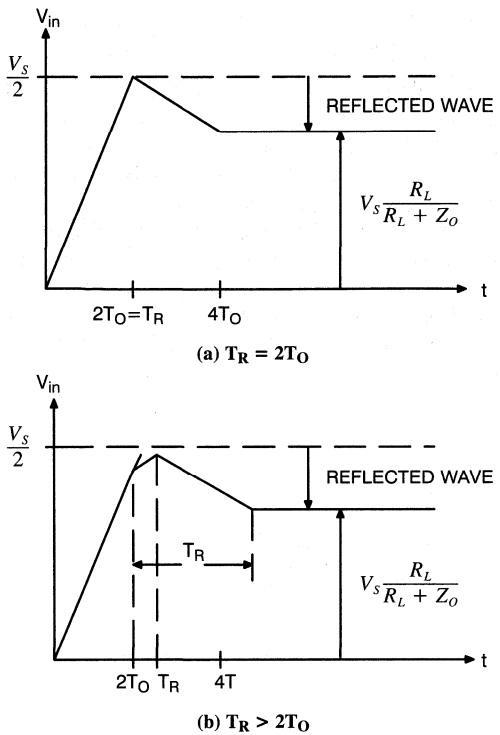


Figure 9. Effects of Rise Time on Response for $R_L < Z_O$

Multiple Reflections

Now consider the case of an ideal transmission line with multiple reflections caused by improper terminations at both ends of the line. The circuit and waveforms appear in *Figure 10*. The reflection coefficients at the source and the load are both negative—the source resistance and the load resistance are both less than the line characteristic impedance.

When the switch is initially closed, a step function of amplitude

$$V_O = V_{in} = \frac{V_S Z_O}{R_S + Z_O} \qquad \text{Eq. 30}$$

appears on the line and travels toward the load. After a one-way propagation delay time, T_O, the wave reflects back with an amplitude of $\varrho_L V_O$.

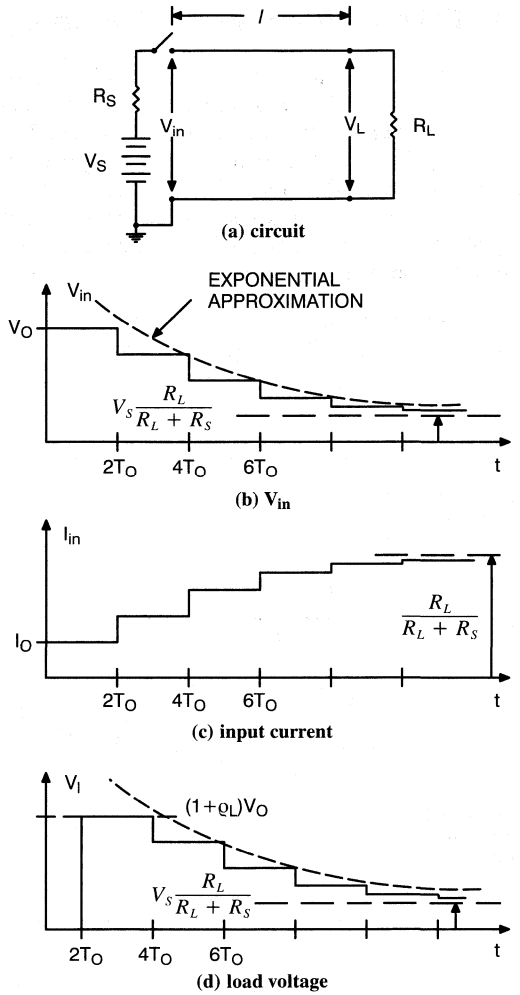


Figure 10. Step Function Applied to Line Mismatched on Both Ends; Shown for Negative Values of ϱ_S and ϱ_L

This first reflected wave than travels back to the source, and at time t = 2 T_O, the wave reaches the input end of the line. At this time, the first reflection at the source occurs, and a wave of amplitude ϱ_S (ϱ_L V_O) reflects back to the load. At time t = 3 T_O, this wave again reflects from the load back to the source with amplitude

$$\varrho_L\varrho_S\ (\varrho_L V_O) = \varrho_S\varrho_L{}^2 V_O \qquad \text{Eq. 31}$$

This back and forth reflection process continues until the amplitudes of the reflections become so small that they cannot be observed. The circuit is then said to be in a quiescent state.

Effective Time Constant

Voltage reflections in small increments and of short durations approximate an exponential function, as indicated by the dashed line in *Figure 10b*. The smaller and narrower the steps become, the more closely the waveform approaches an exponential curve.

The mathematical derivation is presented on pages 178 and 179 of Reference 1. The time constant is

$$K = -\frac{2T_O}{1 - \varrho_S\varrho_L} \qquad \text{Eq. 32}$$

Thus, the resultant voltage waveform at the load can be approximated by

$$V(t) = V_O e\left(\frac{t}{K}\right) \qquad \text{Eq. 33}$$

For *Equation 32* to be accurate, ϱ_L and ϱ_S must be reasonably large (approaching ± 1) so that the incremental steps are small. Because the product $\varrho_S\varrho_L$ is a positive number, less than one, the time constant is a negative number, which indicates that the exponential decreases with time. This is usually the case in transient circuits.

Both reflection coefficients must also have the same sign to yield a continually decreasing or increasing waveform. Opposite signs give oscillatory behavior that cannot be represented by an exponential function.

From Transmission Line to Circuit Analysis

When a transmission line is terminated in its characteristic impedance, the line behaves like a resistor. It usually does not matter if you use transmission line or circuit analysis, provided that you take the propagation delays into account.

Consider the case of a short-circuited transmission line driven by a step function with a source impedance unequal to the characteristic line impedance. The general case is shown in *Figure 10a*. For $R_L = 0$ the reflection coefficients are

$$\varrho_S = \frac{Z_S - Z_O}{Z_S + Z_O} \qquad \varrho_L = -1 \qquad \text{Eq. 34}$$

The approximate time constant is

$$-k = \frac{2T_O}{1 - \varrho_S\varrho_L} = \frac{2T_O}{1 + \varrho_S} = \frac{T_O(Z_S + Z_O)}{Z_S}$$

or

$$-k = T_O + \frac{T_O Z_O}{Z_S} \qquad \text{Eq. 35}$$

Recall that

$$T_O = l\sqrt{LC} \qquad \text{Eq. 36}$$

(one-way delay) and

$$Z_O = \sqrt{L/C} \qquad \text{Eq. 37}$$

where l is the physical length of the line, and L and C are the per-unit-length parameters. Substituting these variables into *Equation 35* yields

$$-k = T_O + l\frac{L}{Z_S} \qquad \text{Eq. 38}$$

It is necessary to have Z_S smaller than Z_O. Thus,the reflection coefficients have the same sign to give exponential behavior. Opposite signs give oscillatory behavior.

If $Z_S < Z_O$, the exponential approximation becomes more accurate. If Z_S is very small compared to Z_O, then T_O is negligible compared to lL/Z_O, so that Equation 35 reduces to

$$k = -l\frac{L}{Z_S} \qquad \text{Eq. 39}$$

But lL is the total loop inductance, and Z_S is the circuit's total series impedance. The time constant is then

$$k = \frac{L'}{R_S} \qquad \text{Eq. 40}$$

This is the same time constant you would obtain by a circuit analysis approach if you considered the line a series combination of L′ and R_S. By open-circuiting the line and performing a similar analysis, it can be shown that an RC time constant results.

Types of Transmission Lines

The types of transmission lines include

- Coaxial cable
- Twisted pair
- Wire over ground
- Microstrip lines
- Strip lines

Coaxial Cable

Coaxial cable offers many advantages for distributing high-frequency signals. The well-defined and uniform characteristic impedance permits easy matching. The cable's ground shield reduces crosstalk, and the low attenuation at high frequencies make the cable ideal for transmitting the fast rise-time and fall-time signals generated by Cypress CMOS ICs. However, because of high cost, coaxial cable is usually restricted to applications that permit no alternatives. These applications usually involve clock distribution systems on PCBs or backplanes.

Because coaxial cable is not easily handled by automated assembly techniques, its application requires human assemblers. This requirement further increases costs.

Coaxial cables have characteristic impedances of 50Ω, 75Ω, 93Ω, or 150Ω. These values are the most common, although special cables can be made with other impedances.

Coaxial cable's propagation delay is very low. You can compute it using the formula

$$t_{pd} = 1.017\sqrt{e_r} \quad (ns/ft) \qquad \text{Eq. 41}$$

where e_r is the relative dielectric constant and depends upon the dielectric material used. For solid Teflon and polyethylene, the dielectric constant is 2.3. The propagation delay is 1.54 ns per foot. For maximum propagation velocity, you can use coaxial cables with dielectric Styrofoam or polystyrene beads in air. Many of these cables have high-characteristic impedances and are slowed considerably when capacitively loaded.

Twisted Pair

You can make twisted pairs from standard wire (AWG 24 – 28), twisted about 30 turns per foot. The typical characteristic impedance is 110Ω.

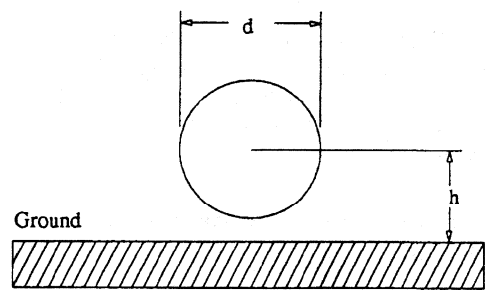
Because the propagation delay is directly proportional to the characteristic impedance (*Equation 9*), the propagation delay is approximately twice that of coaxial cable. Twisted pairs are used for backplane wiring, sometimes for driving differential receivers, and for breadboarding.

Wire Over Ground

Figure 11 shows a wire over ground. This configuration is used for breadboarding and backplane wiring. The characteristic impedance is approximately 120Ω. This value can vary as much as ±40 percent, depending upon the distance from the groundplane, the proximity of other wires, and the configuration of the ground.

Microstrip Lines

A microstrip line (*Figure 12*) is a strip conductor (signal line) on a PCB separated from a ground plane by a dielectric. If the line's thickness, width, and distance from the ground plane are controlled, the line's characteristic impedance can be predicted with a tolerance of ±5 percent.



$$Z_o = \frac{60}{\sqrt{e_r}} \ln\left(\frac{4h}{d}\right)$$

Figure 11. Wire Over Ground

The formula given in *Figure 12* has proven to be very accurate for width-to-height ratios between 0.1:1 and 3.0:1 and for dielectric constants between 1 and 15.

The inductance per foot for microstrip lines is

$$L = (Z_O)^2 C_O \qquad \text{Eq. 42}$$

where Z_O is the characteristic impedance and C_O is capacitance per foot.

The propagation delay of a microstrip line is

$$t_{pd} = 1.017\sqrt{0.45e_r + 0.67} \quad (ns/ft) \qquad \text{Eq. 43}$$

Note that the propagation delay depends only upon the dielectric constant and is not a function of the line width or spacing. For G-10 fiberglass epoxy PCBs (dielectric constant of 5), the propagation delay is 1.74 ns per foot.

Strip Lines

A strip line consists of a copper strip centered in a dielectric between two conducting planes (*Figure 13*). If the line's thickness, width, dielectric constant, and distance between ground planes are all controlled, the tolerance of the characteristic impedance is within ±5 percent. The equation given in *Figure 13* is accurate for W/(b − t) < 0.35 and t/b < 0.25.

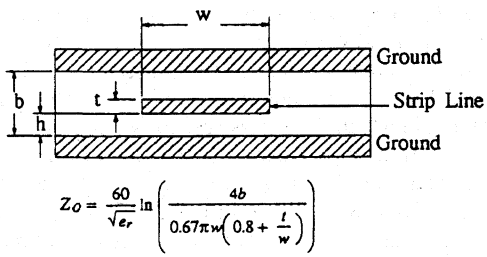


Figure 13. Strip Line Construction

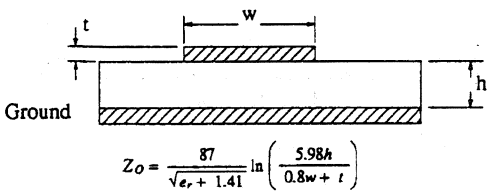


Figure 12. Microstrip Line

The inductance per foot is given by the formula

$$L = (Z_O)^2 C_O \qquad \text{Eq. 44}$$

The propagation delay of the line is given by the formula

$$t_{pd} = 1.017\sqrt{e_r} \quad (ns/ft) \qquad \text{Eq. 45}$$

For G-10 fiberglass epoxy boards, the propagation delay is 2.27 ns per foot. The propagation delay is not a function of line width or spacing.

Modern PCBs

Most PCBs employ microstrip, stripline, or some combination of the two. Microstrip construction on a double-sided board with power and ground nets can suffice for low- to medium-performance, and low-density PCBs.

For high-performance, high-density PCBs, stripline construction is preferred. Power planes isolate signal layers from each other and provide higher-quality power and grounds than those of a two-layer board. Manufacturing quality control assures that the metalization is of uniform thickness and that the layers are properly laminated, thus ensuring uniform, predictable electrical characteristics.

When to Terminate Transmission Lines

Transmission lines should be terminated when they are long. From the preceding analysis, it should be apparent that

$$Long\ Line > \frac{t_r}{2t_{pdL}} \qquad \text{Eq. 46}$$

where t_{pdL} is the loaded propagation delay of the line per unit length. For Cypress CMOS and BiCMOS products, the rise time, t_r, is typically 2 ns.

For stripline construction (multilayer PCBs), the line length at which voltage reflections might occur has been shown to vary from 4.73 inches for a 10-pF load to 3.05 inches for an 80-pF load (see *Equation 18* and *Table 1*).

Not all lines exceeding these lengths need to be terminated. Terminations are usually required on control lines (such as clock inputs, write and read strobe lines on SRAMs and FIFOs) and chip select or output-enable lines on RAMs, PROMs, and PLDs. Address lines and data lines on RAMs and PROMS usually have time to settle because they are normally not the highest-frequency lines in a system. However, if very heavily loaded, address and databus lines might require terminations.

Line Termination Strategies

There are two general strategies for transmission line termination:

1. Match the load impedance to the line impedance
2. Match the source impedance to the line impedance

In other words, if either the load reflection coefficient or the source reflection coefficient can be made to equal zero, reflections are eliminated. From a systems design viewpoint, strategy 1 is preferred. Eliminating the reflection at the load (i.e., dissipating the excess energy) before the energy travels back to the source causes less noise, electromagnetic interference (EMI), and radio frequency interference (RFI).

Multiple Loads, Buses, and Nodes

In the case where multiple loads are connected to a transmission line, only one termination circuit is required. The termination should be located at the load that is electrically the greatest distance from the source. This is usually the load that is the greatest physical distance from the source. A point-to-point or daisy chain connection of loads is preferred.

Bidirectional buses should be terminated at each end with a circuit whose impedance equals the intrinsic, characteristic line impedance. The reason is that each transmitting device sees the characteristic impedance of the line when the device is transmitting.

Consider next a line that has three bidirectional nodes: one on each end and one in the middle. The middle node, when driving the line, sees an impedance equal to $Z_O/2$, because the node is looking into two lines in parallel with each other. The end nodes, however, see an impedance of Z_O. In this case, as in a backplane, each end of the line should be terminated in an impedance equal to $Z_O/2$. When heavily loaded, *Equation 12* must be used to calculate the loaded characteristic impedance, and this must be used instead of Z_O

Types of Terminations

There are three basic types of terminations: series damping, pull-up/pull-down, and parallel AC terminations. Each has its advantages and disadvantages.

Except for series damping, the termination network should be attached to the input (load) that is electrically the greatest distance from the source. Component leads should be as short as possible to prevent reflections due to lead inductance.

Series Damping

Series damping is accomplished by inserting a small resistor (typically 10Ω to 75Ω) in series with the transmission line, as close to the source as possible (*Figure 14*). Series damping is a special case of damping in which the series resistor value plus the circuit output impedance equals the transmission line impedance. The strategy is to prevent the wave reflected back from the load from reflecting back from the source. This is done by making the source reflection coefficient equal to zero.

The channel resistance (on resistance) of the pull-down device for Cypress ICs is 10Ω to 20Ω, depend-

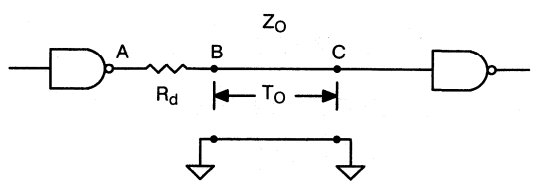


Figure 14. Series Damping Termination

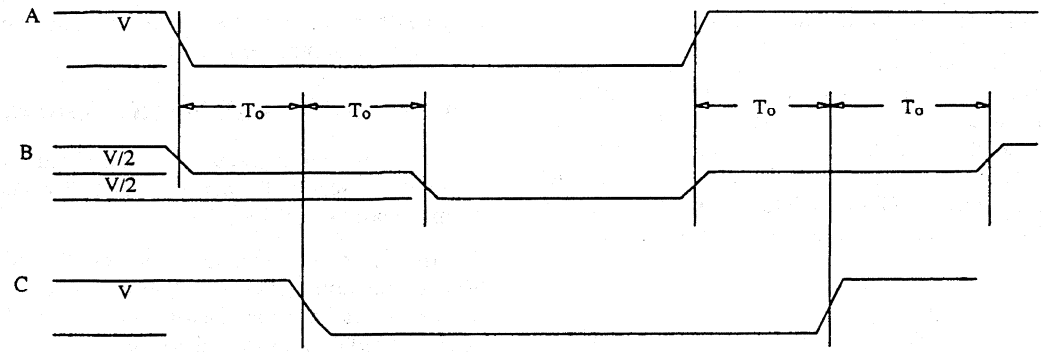


Figure 15. Series Damping Timing

ing upon the current-sinking requirements. Thus, subtract this value from the series-damping resistor, R_d.

$$Z_O = R_S + R_d \qquad \text{Eq. 47}$$

A disadvantage of the series-damping technique, as illustrated in *Figure 15*, is that during the two-way propagation delay time of the signal edges, the voltage at the input to the line is halfway between the logic levels, due to the voltage divider action of R_S. The "half voltage" propagates down the line to the load and then back from the load to the source. This means that no inputs can be attached along the line, because they would respond incorrectly during this time. However, you can attach any number of devices to the load end of the line because all the reflections are absorbed at the source. If two or more transmission lines must be driven in parallel, the value of the series-damping resistor does not change.

The advantages of series termination are:

- Requires only one resistor per line
- Consumes little power
- Permits incident wave switching at the load after a T_O propagation delay
- Provides current limiting when driving highly capacitive loads; the current limiting also helps reduce groundbounce

The disadvantages of series termination are:

- Degrades rise time at the load due to increased RC time constant
- Should not be used with distributed loads

The low input current required by Cypress CMOS ICs results in essentially no DC power dissipation. The only AC power required is to charge and discharge the parasitic capacitances.

Pull-Up/Pull-Down Termination

The pull-up/pull-down resistor termination shown in *Figure 16* is included for historical reasons and for the sake of completeness. For TTL driving long cables, such as ribbon cables, the values $R_1 = 220\Omega$ and $R_2 = 330\Omega$ are recommended by several bus interface standards. If the cable is disconnected, the voltage at point B is 3V, which is well above the 2V minimum high TTL specification. Because most

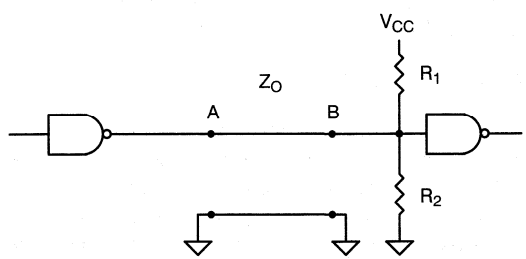


Figure 16. Pull-Up/Pull-Down

control signals are active LOW, a disconnected cable results in the unasserted state.

The maximum value of R_1 is determined by the maximum acceptable signal rise time, which is a function of the charging RC time constant. The minimum value of R_1 is determined by the amount of current the driver can sink. The value of R_2 is chosen such that a logic HIGH is maintained when the cable is disconnected. The equivalent Thévenin resistance is

$$R_T = \frac{R_1 R_2}{R_1 + R_2} \quad \text{Eq. 48}$$

The value of R_1 and R_2 in parallel is slightly less than the cable's characteristic impedance. Ribbon cables with characteristic impedances of 150Ω are typical.

If both resistors are used, DC power is dissipated all the time. If only a pull-down resistor (R_2) is used, DC power is dissipated when the input is in the logic HIGH state. Conversely, if only a pull-up resistor (R_1) is used, power is dissipated when the input is in the LOW state. Due to these power dissipations, this termination is not recommended.

If an unterminated control signal on a PCB is suspected of causing a problem, a resistor whose value is slightly less than the characteristic impedance of the line (e.g., 47Ω) can be connected between the input pin and ground. Be sure that the driver can source sufficient current to develop a TTL high voltage level (2.0V) across the resistor.

In special cases where inputs should be either pulled up (HIGH) for logic reasons or because of very slow rise and fall times, you can use a pull-up resistor to V_{CC} in conjunction with the terminating network shown in *Figure 17*. DC power is dissipated when the source is LOW.

Parallel AC Termination

Figure 17 illustrates the recommended general-purpose termination. It does not have the disadvantage of the half-voltage levels of series damping terminations, and it causes no DC power dissipation. You can attach loads anywhere along the line, and they see a full voltage swing.

The disadvantage is that a parallel AC termination requires two components, versus the one-component series-damping termination.

Commercially Available RC Networks

A variety of combinations of R and C values are available as series RC networks in SIP packages from at least two sources.

Bourns calls these networks the Series 701 and 702 RC Termination Networks. You can obtain datasheets by calling the factory in Logan, Utah (801-750-7200) or a local sales office.

Thin Film Technology also refers to the networks as RC Termination Networks. You can obtain datasheets by calling the factory in North Mankato, Minnesota at 507-635-8445.

Dale Electronics calls their product Resistor/Capacitor Networks. Call 915−595−8139 for information.

California Micro Devices calls their product R−C Networks. Call 408−263−3214 for information.

Low-Pass Filter Analysis

The parallel AC termination has another advantage: it acts as a low-pass filter for short pulses. You can verify this by analyzing the response of the circuit illustrated in *Figure 18* to a positive and a negative step function. The positive step function is generated by moving the switch from position 2 to position 1. The negative step function is generated by moving the switch from position 1 to position 2. The response of the circuit to a pulse is the superposition of the two separate responses. The input impedance of the Cypress circuits connected to the

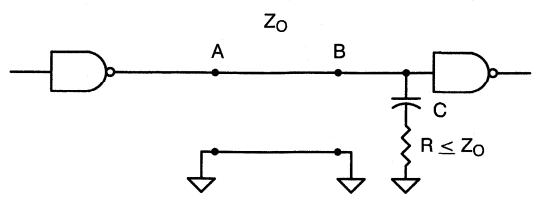


Figure 17. Parallel AC Termination

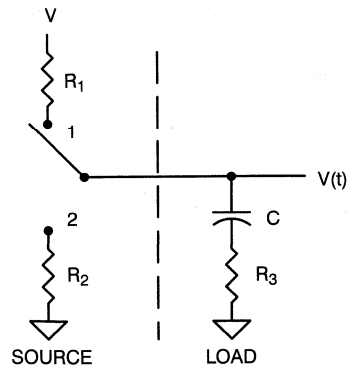


Figure 18. Lumped Load; AC Termination

termination network are so large that they can be ignored for this analysis.

Classic circuit analysis usually assumes an ideal source ($R_1 = R_2 = 0$). In real-world digital circuits, the source output impedance is not only non-zero, but also varies depending upon whether the output is changing from LOW to HIGH or vice versa.

For Cypress ICs, $100\Omega > R_1 > 50\Omega$ and $20\Omega > R_2 > 10\Omega$, depending upon speed and output current-sinking requirements.

Positive Step Function Response

The initial voltage on the capacitor is zero. At t = 0, the switch is moved from position 2 to position 1. At t = 0+, the capacitor appears as a short circuit, and the voltage V is applied through R_1 to charge the load (R_3C). The voltage across the capacitor $V_C(t)$, is

$$V_C(t) = V\left(1 - e^{\left[\frac{-t}{(R_1 + R_3)C}\right]}\right) \quad \text{Eq. 49}$$

In theory, the voltage across the capacitor reaches V when t equals infinity. In practice, the voltage reaches 98 percent of V after 3.9 RC time constants. You can verify this by setting $V_C(t)/V = 0.98$ in *Equation 49* and solving for t.

Negative Step Function Response

The capacitor is charged to approximately V. At t = 0, the switch is moved from position 1 to position 2, and the capacitor is discharged. The voltage across the capacitor, $V_C(t)$ is

$$V_C(t) = Ve^{\left[\frac{-t}{(R_2 + R_3)C}\right]} \quad \text{Eq. 50}$$

The voltage decays to 2 percent of its original value in 3.9 RC time constants. You can verify this by setting $V_C(t)/V = 0.02$ in *Equation 50* and solving for t.

The Ideal Case

Consider the ideal case where $R_1 = R_2 = 0$. Let $R_3 = R$ in *Equations 49* and *50*. If a positive pulse of width T is applied to the modified circuit of *Figure 18*, the pulse disappears if 4RC > T.

Because the discharging time constant is the same as the charging time constant for the ideal case, a negative-going pulse of width T also disappears if 4RC > T. That is, if the applied signal is normally HIGH and goes LOW, as does the write strobe on an SRAM, the termination filters out all negative glitches less than 4 RC time constants in width.

The maximum frequency that the circuit passes is

$$F(max.) = \frac{1}{2T} \quad \text{Eq. 51}$$

This is true because the charging and discharging time constants are equal for the ideal case.

Capacitance for the Ideal Case

The value of the capacitor, C, must be chosen to satisfy two conflicting requirements. First, the capacitor should be large enough to either absorb or supply the energy contained or removed when positive-going or negative-going glitches occur. Second, the capacitor should be small enough to avoid either delaying the signal beyond some design limit or slowing the signal rise and fall times to more than 5 ns.

A third consideration is the impedance caused by the capacitor's capacitive reactance, X_C. The digital waveforms applied to the AC termination can be ex-

pressed as a Fourier Series so that they can be manipulated mathematically. However, because these signals are not periodic in the classical meaning of the word, it is not clear that the AC steady-state analysis model of X_C applies here.

In most applications, the degradation of the signal's rise and fall times beyond 5 ns determines the maximum value of the capacitor. The procedure is to calculate the rise time between the 10- and 90-percent amplitude levels, equate this rise time to 5 ns, and solve for C in terms of R:

$$V(t) = V(1 - e^{\left[\frac{-t}{RC}\right]}) \qquad \text{Eq. 52}$$

for t yields

$$t = RC \ln\left[\frac{1}{1 - \frac{V(t)}{V}}\right] \qquad \text{Eq. 53}$$

For $\frac{V(t)}{V} = 0.1$, $t = 0.10\ RC$.

For $\frac{V(t)}{V} = 0.9$, $t = 2.3\ RC$.

The time for the signal to transition from 10 to 90 percent of its final value is then T = 2.2 RC. Solving for C yields

$$C = \frac{T}{2.2R} \qquad \text{Eq. 54}$$

For T = 5 ns, *Table 2* can be constructed. This table indicates that 50Ω transmission lines on PCBs that are terminated with RC networks should use a 47Ω resistor and a capacitor of 48 pF max; 47 pF is a standard value. This network eliminates glitches of 9 ns or less. The table's second column applies to wirewrapping construction, which is not recommended for systems operating at frequencies over 10 MHz. An exception is if the system consists of less than six MSI or SSI ICs.

Table 2. Termination Value for an Ideal Case

	PCB	Wirewrapped
Z_O (Ω)	50	120
R (Ω)	47	110
C (max., pF)	48	20
RC (ns)	2.25	2.2
4RC (ns)	9	8.8

The Real World

To go from the ideal to the real world, calculate the values of R_1 and R_2 from the curves on the datasheet of the device driving the line. R_1 is the slope of the output source current vs. output voltage between 2 and 4V. R_2 is the slope of the output sink current vs output voltage between 0 and 0.8V.

Add the value of R_1 to 47Ω and calculate C, using *Equation 54*. Then check to see that the RC charging time constant does not violate some minimum positive pulse-width specification for the line. If so, reduce C.

Add the value of R_2 to 47Ω and calculate C. Then check to see if the discharging RC time constant violates some minimum pulse-width specification for the line. If so, reduce C.

If the line is heavily loaded, *Equation 12* must be used to calculate the loaded characteristic impedance, which determines the maximum value of R. The Maximum value of C is then calculated using *Equation 54*.

Schottky Diode Termination

In some cases it can be expedient to use Schottky diodes or fast-switching silicon diodes to terminate lines. The diode switching time must be at least four times as fast as the signal rise time. Where line impedances are not well defined, as in breadboards and backplanes, the use of diode terminations is convenient and can save time.

A typical diode termination appears in *Figure 19*. The Schottky diode's low forward voltage, V_f (typically 0.3 to 0.45V), clamps the input signal to a V_f below ground (lower diode) and V_{CC} + V_f (upper diode). This significantly reduces signal undershoot

and overshoot. Some applications may not require both diodes.

The advantages of diode terminations are:

- Impedance matched lines are not required
- The diodes replace terminating resistors or RC terminations
- The diodes' clamping action reduces overshoot and undershoot
- Although diodes cost more than resistors, the total cost of layout might be less because a precise, controlled transmission-line environment is not required
- If ringing is discovered to be a problem during system debug, the diodes can be easily added

As with resistor or RC terminations, the leads should be as short as possible to avoid ringing due to lead inductance.

A few of the types of Schottky diodes commercially available are

- HSMS−2822 (Hewlet−Packard)
- 1N5711
- MBD101, MBD102 (Motorola)
- SN74S1050/52/56 (TI, single-diode arrays)
- SN74S1051/53 (TI, double-diode arrays)

Unterminated Line Example

The following example illustrates the procedure for calculating the waveforms when a Cypress PLD generates the write strobe for four Cypress FIFOs. The PLD is a PALC16L8 device and the FIFOs are CY7C429s.

The equivalent circuit appears in *Figure 20* and the unmodified driving waveform in *Figure 21*. The rise and fall times are 2 ns. The length of the stripline trace on the PCB is 8 inches and the intrinsic characteristic line impedance is 50Ω. The voltage waveforms at the source (point A) and the load (point B) must be calculated as functions of time. Stripline construction is used for this example because in most modern high-performance digital systems, the PCBs have multiple layers.

The equivalent ON channel resistance of the PLD pull-up device, 62Ω, is calculated using the output

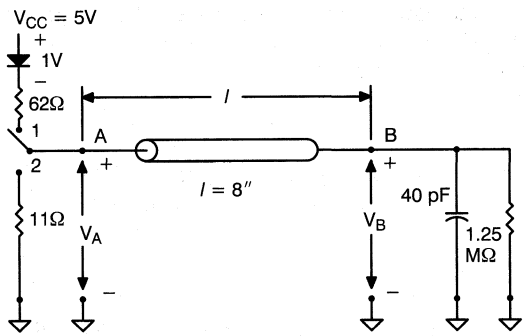


Figure 20. Equivalent Circuit for Cypress PAL Driving

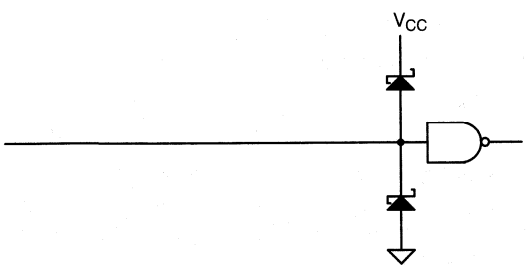


Figure 19. Schottky Diode Termination

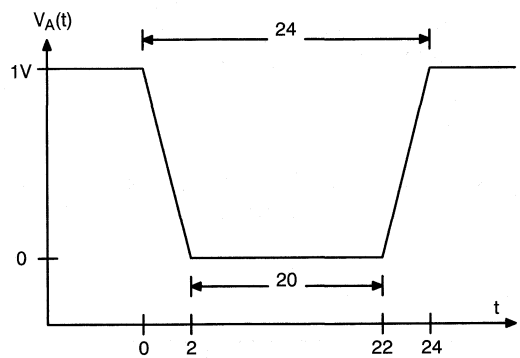


Figure 21. $V_A(t)$, Unmodified

source current versus voltage graph, over the region of interest (2 to 4V), from the PALC20 series datasheet. The equivalent resistance of the pull-down device, 11Ω, is calculated in a similar manner, using the output sink current versus output voltage graph, over the region of interest (0.4 to 2V), also on the datasheet.

The equivalent input circuit for the FIFO is constructed by approximating the input and stray capacitance with a 10-pF capacitor and the input resistance with a 5-MΩ resistor. The input leakage current for all Cypress products is specified as a maximum of ±10 μA, which guarantees a minimum of 500 KΩ at V_{in} = 5V. Typical leakage current is 10 pA.

Because the PLD is driving four FIFOs in parallel, the equivalent lumped capacitance is 4 × 10 pF = 40 pF, and the equivalent lumped resistance is 5,000,000/4 = 1.25 MΩ

The next step is to calculate the propagation delay and the loaded characteristic impedance of the line. The unloaded propagation delay of the line is calculated using *Equation 45* with a dielectric constant of 5:

$$t_{pd} = 2.27 \ (ns/ft) \qquad \text{Eq. 55}$$

To calculate the loaded line propagation delay, the intrinsic capacitance must first be calculated using *Equation 9*.

$$t_{pd} = Z_O C_O \qquad \text{Eq. 56}$$

where Z_O is the intrinsic characteristic impedance, and C_O is the intrinsic capacitance.

$$C_O = \frac{t_{pd}}{Z_O} = \frac{2.27 \ ns/ft}{50} = 45.4 \ pF/ft. \qquad \text{Eq. 57}$$

Because the line is loaded with 40 pF, *Equation 11* is used to compute the loaded propagation delay of the line.

$$t_{pdL} = t_{pd}\sqrt{1 + C_D/C_O}$$

$$t_{pdL} = 2.27 \ ns/ft \sqrt{1 + \frac{40 \ pF}{45.4 \ pF/ft \times \frac{8 \ in.}{12 \ in./ft}}}$$

$$t_{pdL} = 3.46 \ ns/ft \qquad \text{Eq. 58}$$

Note that the capacitance per unit length must be multiplied by the line length to arrive at an equivalent lumped capacitance.

The intrinsic line impedance is reduced by the same factor by which the propagation delay is increased (1.524; see *Equation 12*):

$$Z_O' = \frac{50\Omega}{1.524} = 32.8\Omega \qquad \text{Eq. 59}$$

Initial Conditions

At time t = 0, the circuit shown in *Figure 20* is in a quiescent state. The voltage at points A and B must be the same. By inspection:

$$V_A = V_B = (V_{CC} - Vf)\left(\frac{R_L}{R_S + R_L}\right)$$

$$= (5 - 1)\left(\frac{1.25 \times 10^6}{28 + 1.25 \times 10^6}\right) = 4V \qquad \text{Eq. 60}$$

At t = 0, the driving waveform changes from 4V to approximately 0V with a fall time of 2 ns. This is shown in *Figure 20* by the switch arm moving from position 1 to position 2.

The wave propagates to the load at the rate of 3.46 ns per foot and arrives there

$$T_O = 3.46 \ ns/ft \times \frac{8 \ in.}{12 \ in./ft} = 2.3 \ ns \qquad \text{Eq. 61}$$

later, as illustrated in *Figure 22b.*

Because the reflection coefficient at the load is ϱL = 1, an early equal and opposite polarity waveform is propagated back to the source from the load. The reflection arrives at t = $2T_O$ = 4.6 ns (*Figure 22a*). Note that the fall time is preserved.

The reflection coefficient at the source is

$$\varrho_S = \frac{R_S - Z_O'}{R_S + Z_O'} = \frac{11 - 32.8}{11 + 32.8} = -0.498 \qquad \text{Eq. 62}$$

To simplify the calculations that follow, consider −0.5 to be the low-level source reflection coefficient. The magnitude of the reflected voltage at the source is then

$$V_{S1} = -4V \times (-0.5) = 2V \qquad \text{Eq. 63}$$

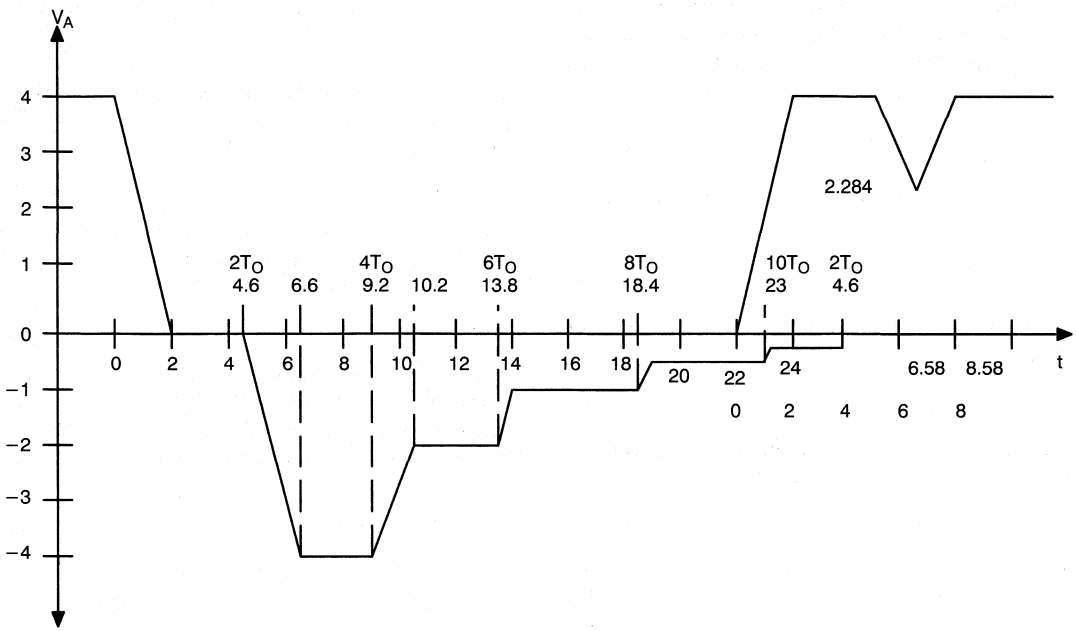


Figure 22a. Unterminated Line Example; $V_A(t)$

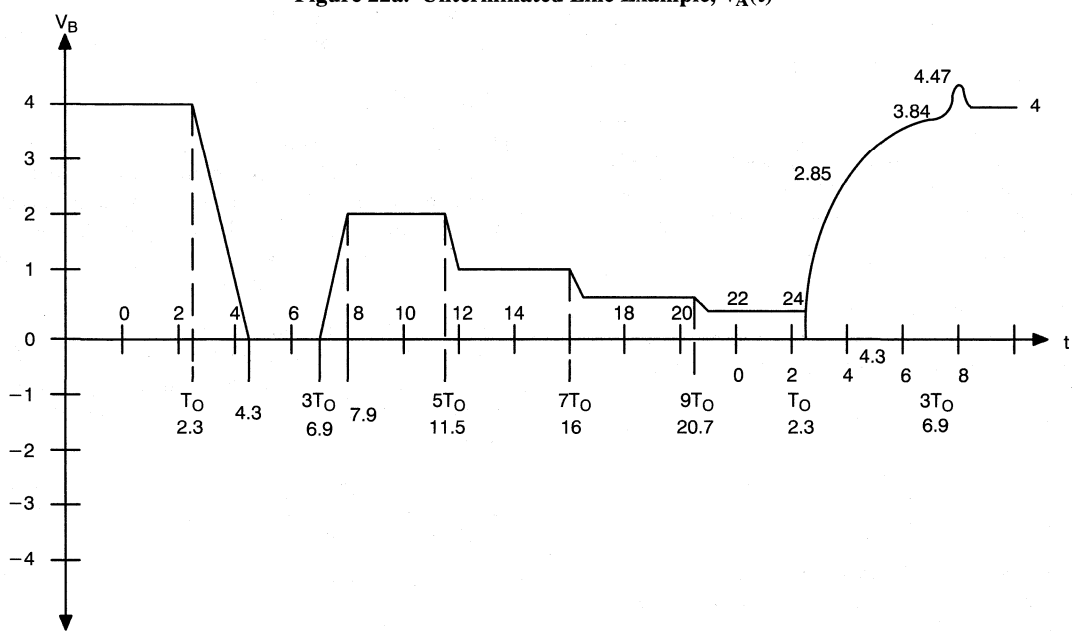


Figure 22b. Unterminated Line Example; $V_B(t)$

This wave propagates from the source to the load and arrives at t = 3 T_O. The wave adds to the 0V signal. The rise time is preserved, and thus the time required for the signal to go from 0 to 2V is

$$t_r = \frac{2V \times 2\ ns}{4V} = 1\ ns \qquad \text{Eq. 64}$$

The signal at the load thus reaches the 2V level at time

$$t = 3T_O + 1\ ns = 7.9\ ns \qquad \text{Eq. 65}$$

and remains at that level until the next reflection occurs at

$$t = 5T_O \qquad \text{Eq. 66}$$

The wave that arrives at the load at 3 T_O reflects back to the source and arrives at

$$t = 4T_O = 9.2\ ns \qquad \text{Eq. 67}$$

The 2V level adds to the −4V level, for a total of −2V. The rise time is preserved, so that this level is reached at

$$t = 4T_O + 1\ ns = 10.2\ ns \qquad \text{Eq. 68}$$

and maintained until the next reflection occurs at

$$t = 6T_O \qquad \text{Eq. 69}$$

The 2V wave that arrives at the source at t = 4T_O reflects back to the load and arrives at t = 5T_O. The portion that is reflected back to the load is

$$V_{S2} = 2 \times (-0.5) = -1V \qquad \text{Eq. 70}$$

This value subtracts from the 2V level to give 2 − 1 = 1V. Because the fall time is preserved, the time required for the signal to go from 2 to 1V is

$$t_f = \frac{1V \times 2\ ns}{4V} = 0.5\ ns \qquad \text{Eq. 71}$$

The 1V level is thus reached at time

$$t = 5T_O + 0.5\ ns = 12\ ns \qquad \text{Eq. 72}$$

At t = 6T_O, the 1V wave arrives back at the source, where it subtracts from the −2V level to give −1V. The rise time is

$$t_r = 1 \times 0.5\ ns/V = 0.5\ ns \qquad \text{Eq. 73}$$

The signal at the source reaches the −1V level at

$$t = 6T_O + 0.5 = 14.3\ ns \qquad \text{Eq. 74}$$

The 1V wave that arrives at the source at t = 6T_O is reflected back to the load and arrives at t = 7T_O. The portion that is reflected back is

$$V_{S3} = 1 \times (-0.5) = -0.5V \qquad \text{Eq. 75}$$

This value subtracts from the 1V level to give 0.5V. The fall time is 0.25 ns. The 0.5V level remains until the next reflection reaches the load at

$$t = 9T_O \qquad \text{Eq. 76}$$

At t = 8T_O the 0.5V wave that reflects from the load at t= 7T_O arrives back at the source, where it subtracts from the −1V level to give −0.5V. The rise time is 0.25 ns. The portion that reflects back to the load is

$$V_{S4} = 0.5 \times (-0.5) = -0.25V \qquad \text{Eq. 77}$$

The −0.25V signal arrives at the load at t = 10T_O = 23 ns and subtracts from the 0.5V signal to give 0.25V.

This process continues until the voltages at points A and B decay to approximately 0V.

Observations

The positive reflection coefficient at the load and the negative reflection coefficient at the source result in an oscillatory behavior that eventually decays to acceptable levels. The voltage at point A reaches −1V after 6T_O delays and the voltage at point B reaches 0.5V after 7T_O delays.

The reflection at the load that causes the voltage to equal the TTL minimum one level (2V) at T = 3T_O causes a problem. The actual input voltage threshold level is 1.5V for TTL-compatible devices that do not exhibit hysteresis.

The voltage at the load falls from 4V to 0V in 2 ns, beginning at t = T_O. Because T_O = 2.3 ns, the voltage reaches zero at

$$2.3\ ns + 2\ ns = 4.3\ ns \qquad \text{Eq. 78}$$

The 1.5V level occurs at

$$4.3\ ns - \frac{2\ ns}{4V} \times 1.5V = 3.55\ ns \qquad \text{Eq. 79}$$

The rising edge begins at

$$t = 3T_O = 6.9\ ns \qquad \text{Eq. 80}$$

The 1.5V level occurs at

$$6.9\ ns + \frac{2\ ns}{4V} \times 1.5 = 7.65\ ns \qquad \text{Eq. 81}$$

The time difference (7.65 − 3.55 = 4.1 ns) is long enough for the FIFO to interpret the signal as a LOW.

Next, consider the width of the positive pulse that begins at the load at t = $3T_O$. Because the rise time is preserved, the signal takes 1 ns to reach 2V, or 0.75 ns to reach 1.5V. The signal begins to fall at t = $5T_O$, reaching 1.5V at

$$t = 5T_O + 0.25\ ns = 11.75\ ns \qquad \text{Eq. 82}$$

The difference (11.75 − 7.65) is 4.1 ns, which is wide enough for the FIFO to interpret as a second clock. To eliminate this pulse, the line must be terminated.

Strobe Shortening Considerations

In this example the width of the negative strobe is 22 to 24 ns. If a CY7C429-20 FIFO is used, the write (or read) strobe must not be shorter than 20 ns. Even if the FIFO does not recognize the 4.5-ns negative pulse, the shortening of the write strobe by $5T_O$ = 11.5 ns is sufficient to violate the minimum negative-pulse-width specification.

This strobe-shortening phenomenon might also occur on other active-LOW control lines such as output enables and chip selects. Clock lines must also be analyzed for this problem; in general, these lines should be terminated.

Now consider an analysis of the write strobe's rising edge to assure that the reflections associated with this edge do not cause multiple clocks or false triggering of the FIFO. At t = 22 ns, the rising edge of the write strobe begins, which is the equivalent of closing the switch in *Figure 20* in the 1 position. For this analysis, it is convenient to start the timescale over at zero, as appears in *Figure 22a* and *b*.

If the forcing function were a step function, the equations of *Figure 4h* would apply. The time constant in the equation is

$$T = \frac{RZ_O{}'C_e}{R + Z_O{}'} \qquad \text{Eq. 83}$$

Because

$$R > Z_O{}',\quad T = Z_O{}'C_e \qquad \text{Eq. 84}$$

where $Z_O{}'$ = 32.8Ω, and C_e = 45.4 pF.

This is the equivalent of saying that you can ignore the1.25-MΩ device input resistance for transient circuit analysis. Substituting $Z_O{}'$ and C_e into the preceding equation yields a time constant of T = 1.489 ns.

Writing the equation for the voltages for the circuit of *Figure 20* yields

$$V_B(t) = iZ_O{}' + \frac{1}{C_e}\int_o^t i\ dt \qquad \text{Eq. 85}$$

Also,

$$V_B(t) = K_tU(t) - K(t - T1)\ U(t - T1) \qquad \text{Eq. 86}$$

where K_t is the rising edge of the write strobe (K = 2V/ns) applied at t = 0 using a unit step function, U(t); and −K(t − T1) represents an equal but opposite waveform applied at t = T1 (after the rise time) using a unit step function, U(t − T1).

Equating the expressions and taking the LaPlace transforms of both sides yields

$$\frac{K}{s^2} - \frac{Ke^{-T1s}}{s^2} = Z_O{}'I(s) + \frac{I(s)}{C_es} = \left(Z_O{}' + \frac{1}{C_es}\right)I(s) \qquad \text{Eq. 87}$$

However,

$$V_B(t) = \frac{1}{C_e}\int_0^t i\ dt, \quad \text{or,} \quad V_B(s) = \frac{I(s)}{C_es} \qquad \text{Eq. 88}$$

Therefore,

$$\frac{K}{s^2} - \frac{Ke^{-T1s}}{s^2} = \left(Z_O{}' + \frac{1}{C_es}\right)C_esV_B(s) \qquad \text{Eq. 89}$$

Solving for $V_B(s)$ yields

$$V_B(s) = \frac{\frac{K}{s^2}(1 - e^{-T1s})}{C_es\left(Z_O{}' + \frac{1}{C_es}\right)} \qquad \text{Eq. 90}$$

which is equivalent to

$$\frac{\frac{K}{Z_O'C_e}(1 - e^{-T1s})}{s^2\left(s + \frac{1}{Z_O'C_e}\right)} \quad \text{Eq. 91}$$

Taking the inverse LaPlace transform yields

$$V_B(t) = \left[KZ_O'C_e\left(e^{\frac{-t}{Z_O'C_e}} - 1\right) + K_t\right] U(t) -$$

$$\left[KZ_O'C_e\left(e^{\left[\frac{-(t-T1)}{Z_O'C_e}\right]} - 1\right) + K(t - T1)\right]U(t - T1) \quad \text{Eq. 92}$$

The first term in *Equation 92* applies from time zero up to and including T1, and the second term applies after T1:

$$V_B(t) = \frac{KZ_O'C_e}{T1}\left(e^{\left[\frac{-t}{Z_O'C_e}\right]} - 1\right) + \frac{K}{T1}(t) \quad \text{Eq. 93}$$

for t ≤ T1.

$$V_B(t) = \frac{KZ_O'C_e}{T1}\left(1 - e^{\left[\frac{T1}{Z_O'C_e}\right]}\right) e^{\left[\frac{-t}{Z_O'C_e}\right]} + K1 \quad \text{Eq. 94}$$

for t > T1.

where K1 is the final value, which is 4V.

Substituting the correct values for t = T1 = 2 ns yields

$$V_B(t = T1) = \frac{2 \times 32.8 \times 45.4 \times 10^{-12}}{2 \times 10^{-9}}(\varepsilon^{-1.489} - 1) + \frac{2V}{ns} \times 2\ ns$$

$$= -1.15 + 4 = 2.85V \quad \text{Eq. 95}$$

If the forcing function is a step function, the equation is

$$V_B(t) = 4V\left(1 - e^{\left[\frac{-t}{Z_O'C_e}\right]}\right) \quad \text{Eq. 96}$$

at t = 2 ns, V_B = 3V, which is more than the 2.85V calculated using *Equation 93*.

At t = 22 ns + T_O, the voltage waveform begins to build up at the load and continues to build until the first reflection from the source occurs at t = $3T_O$.

Equation 94 is used to calculate the voltage at the load at t = $2T_O$, because $1T_O$ is used for propagation delay time:

$$V_B\ (t = 2T_O) = \frac{-2V \times 32.8 \times 45.4 \times 10^{-12}}{2 \times 10^{-9}}(1 - e^{-1.489})(e^{-2}) + 4$$

$$= -1.489\ (0.774)\ (0.1353) + 4$$

$$= -1.559 + 4 = 3.84V \quad \text{Eq. 97}$$

The voltage at the load remains at this value until the first reflection from the source reaches the load at t = $3T_O$.

Meanwhile, at t = T_O, the wave at the load reflects back to the source and arrives at t = $2T_O$. The wave subtracts from the 4V level at the source, as illustrated in *Figure 6c*. The amplitude of the droop is given by

$$V_r = \frac{C'Z_O'}{2}\frac{V_O}{T_r} \quad \text{Eq. 98}$$

for $R_S = Z_O$.

If R_S does not equal Z_O', *Equation 98* must be modified. Instead of $V_O/2$, the voltage is

$$V_O\left(\frac{R_S}{R_S + Z_O'}\right) \quad \text{Eq. 99}$$

so that *Equation 98* becomes

$$V_r = \frac{C'Z_O'V_O}{T_r}\left(\frac{R_S}{R_S + Z_O'}\right) \quad \text{Eq. 100}$$

where C′ = 40 pF, Z_O' = 32.8Ω, R_S = 62Ω, T_r = 2 ns, and V_O = 4V. Substituting these values into *Equation 100* yields

$$V_r = 1.716V \quad \text{Eq. 101}$$

Because 4V − 1.716 = 2.284, the voltage does not drop below the minimum TTL V_{IH} level of 2V, but it does come close.

The reflection coefficient at the source is

$$\varrho_S = \frac{R_S - Z_O'}{R_S + Z_O'} \quad \text{Eq. 102}$$

where, R_S = 62 ohms, Z_O' = 32.8 ohms, ϱ_s = 0.308.

The amount of voltage reflected from the source back to the load is then

$$V_{S1} = 1.716 \times 0.308 = 0.53V \quad \text{Eq. 103}$$

The 40-pF capacitor reduces the rise time of the waveform at the load. The reflection at the source caused by the load capacitor is insufficient to reduce the 4V level to less than the TTL one level (2V).

The reflection coefficient at the source is small enough so that the energy reflected back to the load is insufficient to cause a problem.

References

1. Matick, Richard E. *Transmission Lines for Digital and Communications Networks*. McGraw Hill, 1969.

2. Blood, Jr., William R. *MECL System Design Handbook*. Motorola Inc., 1983.

Document #: GEN0001–0192

Protection, Decoupling, and Filtering of Cypress CMOS Circuits

This application note explains how to protect your ICs with a low-cost zener diode and why it is good insurance against inadvertent voltage transients. Also explained is the reason why decoupling and high-frequency-filtering capacitors are required. A method is provided for determining the capacitors' values.

Zener Diode Protection

Linear power supplies can cause large voltage transients. The transient is negative when it is caused by the collapse of a magnetic field and is positive when the supply is turned on.

Some commercially available laboratory bench supplies behave the same way. When they turn on, they can overshoot several volts. When they turn off, lead inductance can cause a negative transient voltage at the V_{CC} pin. If there is enough energy, this inductance can break down internal gate oxides, destroying or weakening the IC to the extent that it might fail later.

You can avoid this problem by adding a 20¢ zener diode (also called a voltage-regulator diode) between V_{CC} and ground. Connect the diode's cathode to V_{CC} and the anode to ground (see *Figure 1*). A 400-mW, 6.2V 1N525 or equivalent is recommended. You can also use the 1N753, a 500-mW, 6.2V zener diode.

If a voltage greater than the zener voltage (6.2V) occurs on V_{CC}, the diode breaks down, clamping the voltage to 6.2V and shunting the current to ground (see *Figure 2*). The diode can be destroyed if the current multiplied by the zener voltage exceeds the diode's power rating. Because zener diodes *always* fail shorted, they cause the power supply to "crowbar" and thus protect the ICs.

A negative voltage on the V_{CC} line puts a forward bias on the diode. This turns on the diode, which clamps the voltage to approximately −0.8V. If the negative voltage multiplied by the current exceeds the diode's power rating, the diode fails shorted, as in the reversed-bias case, and protects the ICs.

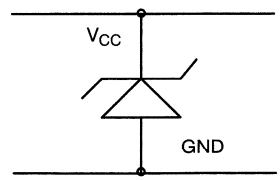


Figure 1. Zener Diode Connection

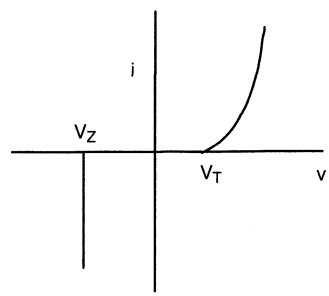


Figure 2. Zener Diode Characteristic

High-Frequency Filtering

In addition to the protection offered by zener diodes, decoupling and high-frequency filter capacitors are required on high-performance CMOS circuits. To use these capacitors effectively, you must understand why they are required.

To realize the fast rise and fall times that Cypress CMOS integrated circuits are capable of achieving, the power-distribution system must be able to supply the instantaneous current required when the device outputs switch from LOW to HIGH. The energy converted to current is stored as charge on the local decoupling capacitors. They decouple or isolate the circuit from the power-distribution system. It is standard practice to use one decoupling capacitor for each IC that drives a transmission line and one capacitor for every three devices that do not.

The PCB trace inductance plus the IC lead inductance can "current-starve" the output circuits, causing rise-time degradation. Remember that the current through an inductor cannot change instantaneously. Therefore, you must minimize any series inductance, including the lead inductance of the decoupling capacitors.

Decoupling-Capacitor Calculations

To determine the value of the decoupling capacitor, you must estimate the instantaneous current required when all the outputs of an IC switch from LOW to HIGH, assuming a reasonable droop of the voltage on the capacitor. The charge stored on the local decoupling capacitor is

$Q = CV$

Differentiating yields

$$i(t) = \frac{dQ}{dt} = C\frac{dV}{dt} \qquad \text{Eq. 1}$$

The characteristic impedance of a typical transmission line is 50Ω. Lines with a heavy capacitive load have lower characteristic impedances.

Next, assume that the IC is a nine-output FIFO, such as the CY7C429. The outputs reach

$V_{CC} - V_t = 5V - 1V = 4V$

Each output requires 4V/50Ω = 80 mA. Because the FIFO has nine outputs, it requires a total of 720 mA during the rise times of the outputs.

Solving *Equation 1* for C yields

$$C = i\frac{dt}{dv} \qquad \text{Eq. 2}$$

The last step is to assume a reasonable, tolerable droop in the capacitor voltage. Assume dV = 100 mV. Additionally, the signal rise and fall times are 2 ns. Substituting these values in *Equation 2* yields

$$C = \frac{720 \times 10^{-3} \times 2 \times 10^{-9}}{100 \times 10^{-3}}$$

$$= 14.4 \times 10^{-9}$$

$$= 0.0144\ \mu F$$

It is standard practice to use 0.01 to 0.1-μF decoupling capacitors. A 0.1-μF capacitor can supply 5A under the conditions assumed in the preceding calculations. Another way to look at the situation is that a 0.1-μF capacitor supplies 720 mA of instantaneous current in 2 ns with only 14.4 mV of voltage droop across the capacitor.

Decoupling capacitors for high-speed Cypress CMOS circuits should be of the high-K ceramic type with a low Effective Series Resistance (ESR). Capacitors using Z5U dielectric are a good choice.

High-Frequency Filter Capacitors

The 0.1 to 0.01-μF decoupling capacitors usually do not provide high-frequency decoupling or filtering. These capacitors do not behave like capacitors at high frequencies because their series resonance frequency is not high enough. This is primarily because of lead inductance in their construction, which is a result of the capacitor's relatively large value.

For high-frequency filter analysis, you can use the simplified capacitor equivalent circuit shown in *Figure 3*. R_S is the ESR, L is the Effective Series Inductance (ESL), and C is the capacitance.

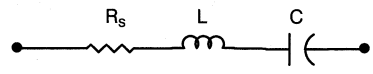


Figure 3. Simplified Capacitor Equivalent Circuit

The impedance of the simplified equivalent circuit is:

$$Z_c = R_s + j\,\omega L + \frac{1}{j\,\omega C} \qquad \text{Eq. 3}$$

$$Z_c = R_s + j\left[\omega L - \frac{1}{\omega C}\right] \qquad \text{Eq. 4}$$

The magnitude of the impedance is

$$Z_c = \sqrt{R_s^2 + \left[\omega L - \frac{1}{\omega C}\right]^2} \qquad \text{Eq. 5}$$

At the series resonant frequency:

$$\omega L = \frac{1}{\omega C}$$

or,

$$\omega = \frac{1}{\sqrt{LC}}$$

At the resonant frequency, $Z_c = R_s$, which is the minimum impedance.

Figure 4 shows how the impedance varies with frequency. The series resistance usually increases as the capacitance decreases. Also, as the capacitance decreases, the inductance typically decreases, which means that the resonant frequency increases. This is usually due to the capacitor's physical construction. Note that a surface-mounted capacitor's lead inductance is at least an order of magnitude less than that of an axial-lead capacitor.

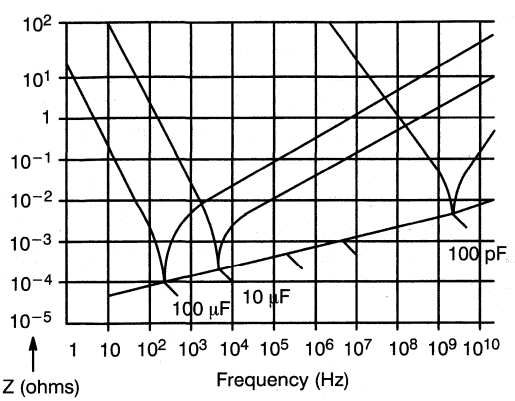


Figure 4. Capacitor Impedance Versus Frequency

The next step in high-frequency filter analysis is to determine a typical system's expected high-frequency components. Begin by assuming that the circuit is driven by a series of digital pulses with finite rise and fall times, then perform a Fourier transform on the series to determine their frequency components.

Fourier Transform of a Periodic Pulse

Figure 5 illustrates a periodic pulse of amplitude A, period T, rise and fall times of t_r, and pulse width of T_p, as measured between the 50-percent-amplitude points.

The approximate frequency-domain transform appears in *Figure 6*. The amplitude of the frequency-domain voltage is a function of the signal's amplitude and duty cycle in the time domain. The fundamental frequency, F_0, is related to the pulse train's period. The first harmonic, F_1, is of equal energy and is a function of the pulse width. The second

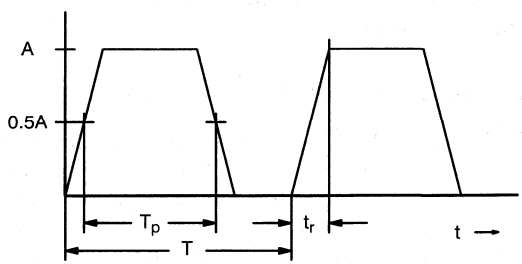


Figure 5. Periodic Pulse Waveform

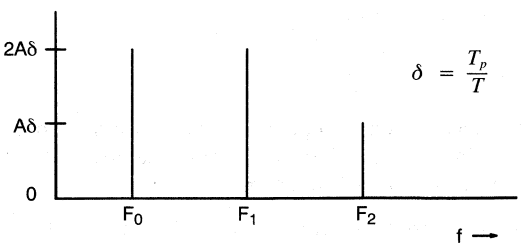


Figure 6. Fourier Transform of Periodic Pulse

harmonic, F_2, contains half the energy of F_0 and is a function of the pulse rise time.

The rise and fall times of Cypress's CMOS and BiCMOS circuits are 2 ns, by design. If a Cypress PLD is driving the write- or read-strobe inputs of a CY7C429–20 FIFO at the maximum frequency of 33.3 MHz (T = 30 ns) with a 10-ns/30-ns duty cycle signal (T_p = 10 ns), the following signal frequencies are generated:

$$F_0 = \frac{1}{\pi T} = \frac{1}{3.1416 \times 30 \times 10^{-9}} = 10.61\ MHz$$

$$F_1 = \frac{1}{\pi T_p} = \frac{1}{3.1416 \times 10 \times 10^{-9}} = 31.83\ MHz$$

$$F_2 = \frac{1}{\pi t_r} = \frac{1}{3.1416 \times 2 \times 10^{-9}} = 159.15\ MHz$$

Within the IC, signal rise and fall times can be as fast as 300 ps (picoseconds), which means that F2 = 1.061 GHz (1,061 MHz). In some ICs short timing pulses are generated internally, but they are usually longer than the 300-ps rise time, so the preceding F2 is the highest harmonic present.

Because the IC's data outputs can normally change no faster than those of the inputs, the outputs do not generate additional higher-frequency harmonics.

Parallel the Filter Capacitors

It will not be possible to find a capacitor with three series resonant frequencies that correspond to F0, F1, and F2. Instead, select one capacitor with a resonant frequency greater than 160 MHz and connect it in parallel with the decoupling capacitor, between V_{CC} and ground, as close to the IC as possible. It will act like a bandpass filter, shunting the unwanted, high frequency signals to ground. The sum of the values of the capacitors should be greater than or equal to the value of capacitance given by *Equation 2*.

The AVX Corporation, Myrtle Beach, South Carolina (803–448–9411), makes a series of "RF/Microwave NPO Capacitors." Their "Ultra Low ESR, 'U' Series" have an ESR of 0.06 Ohm at 500 MHz. A value of 470 pF in the EIA standard size 1210 "chipcap" is recommended. Its series resonant frequency is approximately 180 MHz.

Low-Frequency Filter Capacitors

A solid tantalum capacitor of 10 µF is recommended for every 50 to 100 ICs to reduce power-supply ripple. Place this capacitor as close as physically possible to where the V_{CC} and ground enter the PCB or module.

Document #: GEN0002–0592

Power Characteristics of Cypress Products

This application note presents and analyzes the power dissipation characteristics of Cypress products. The knowledge and tools presented here will help you manage power when using Cypress CMOS products.

Design Philosophy

The design philosophy for all Cypress products is to achieve superior performance at reasonable power dissipation levels. The CMOS technology, circuit design techniques, architecture, and topology are carefully combined to optimize the speed/power ratio.

Power Dissipation Sources

Power is dissipated both inside and outside ICs. The internal and external power have a quiescent (or DC) component and a frequency-dependent component. The relative magnitudes of each depend upon the circuit design objectives.

In circuits designed to minimize power dissipation at low to moderate performance, the frequency-dependent component is significantly greater than the DC component. In the high-performance circuits designed and manufactured by Cypress, the frequency-dependent power component is much lower than the DC component. This is because a large percentage of the internal power is dissipated in linear circuits such as sense amplifiers, bias generators, and voltage/current references, which are required for high performance.

Frequency-Dependent Power

CMOS circuits inherently dissipate significantly less power than either bipolar or NMOS circuits. The ideal CMOS circuit has no direct current path between V_{CC} and V_{SS}. In circuits using other technologies, such paths exist, and DC power is dissipated while the device is in a static state.

The principal component of power dissipation in a power-optimized CMOS circuit is the transient power required to charge and discharge the capacitances associated with the inputs, outputs, and internal nodes. This component is commonly called CV^2 power and is directly proportional to the operating frequency, f.

The charge, Q, stored in a capacitor, C, that is charged to a voltage, V, is given by the equation:

$$Q = CV \qquad \text{Eq. 1}$$

Dividing both sides of *Equation 1* by the time required to charge and discharge the capacitor (one period, or T) yields:

$$\frac{Q}{T} = \frac{CV}{T} \qquad \text{Eq. 2}$$

By definition, current (I) is the charge per unit time and

$$f = \frac{1}{T}$$

Therefore,

$$I = CVf \qquad \text{Eq. 3}$$

The power (P = VI) required to charge and discharge the capacitor is obtained by multiplying both sides of *Equation 3* by V:

$$P = VI = CV^2f \qquad \text{Eq. 4}$$

It is standard practice to assume that the capacitor is charged to the supply voltage (V_{CC}), so that

$$P = V_{CC} I = CV_{CC}^2 f \quad \text{Eq. 5}$$

The total power consumption for CMOS systems depends upon the operating frequency, the number of inputs and outputs, the total load capacitance, the internal equivalent (device) capacitance, and the static (quiescent) or standby power consumption. In equation form:

$$P_d = [C_{INT}F_{INT} + C_{load}F_{load}]V_{CC}^2 = I_{CC}V_{CC} \quad \text{Eq. 6}$$

The first four quantities are frequency dependent, the last is not. This same equation can be used to describe the power dissipation of every IC in the system. The total power dissipation is then the algebraic sum of the individual components.

The relative magnitudes of the various terms in the equation are device dependent. Note that *Equation 6* must be modified if all of the internal nodes or all of the outputs are not switching at the same frequency.

Transient Power

Cypress devices incorporate N-well CMOS inverters that can affect the devices' transient power consumption. In an ideal N-well CMOS inverter, the P-channel pull-up transistor and the N-channel pull-down transistor (which are in series with each other between V_{CC} and V_{SS}) are never on at the same time. Thus, there is no direct current path between V_{CC} and ground, and the quiescent power is very nearly zero.

In the real world, when the input signal makes the transition through the linear region (i.e., between logic levels), both the N-channel and P-channel transistors are partially turned on. This creates a low-impedance path between V_{CC} and V_{SS} whose resistance equals the sum of the N- and P-channel resistances.

DC or Static Power

In addition to conventional gates, Cypress devices contain sense amplifiers, input and output buffers, and bias and reference generators that all dissipate power. RAMs and FIFOs also have memory cells that dissipate standby power whether the IC is selected or not. PROM and PAL products have EPROM memory cells that do not dissipate as much standby power as a RAM cell.

Power-Down Options

Some Cypress static RAMs offer a power-down option that enables you to reduce the devices' power dissipation by approximately an order of magnitude when they are not accessed. The power-down technique disables or turns off the input buffers and sense amplifiers.

Power Dissipation Model

The rest of this application note presents power dissipation models for various Cypress CMOS products as well as information on each product's typical and worst-case power dissipation. The information is presented as functions of frequency, V_{CC}, and temperature.

A general-purpose power dissipation model for all Cypress ICs appears in *Figure 1*.

To obtain power dissipation data on an IC, you must isolate the three components of power dissipation included in *Equation 6* by controlling the IC's inputs. The standby current ($I_{CC(Q)}$) is measured with the inputs to the IC at 0.4V or less. Under this

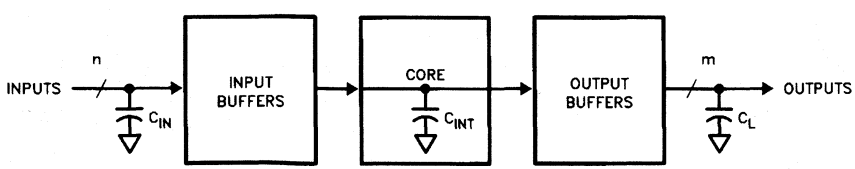


Figure 1. Power Dissipation Model

condition, the input buffers and unloaded output buffers draw only DC leakage currents. All other direct currents derive from the substrate bias generator, sense amplifiers, other internal voltage or current references, and NMOS memory circuits.

At V_{IN} = 1.5V, the input buffers draw maximum I_{CC}. To find the total input buffer I_{CC} current, you measure the total current and subtract the quiescent current. You can then calculate the current per input buffer by dividing the total input-buffer current by the number of input buffers.

Input Buffers

Cypress products use three different types of input buffers. For purposes of illustration, they are referred to as types A, B, and C. *Table 1* lists the buffer types used in various products.

Table 1. Types of Input Buffers

Buffer Type	I_{CC} (Max. in mA)
A	1.3
B	0.8
C	0.6

Figure 2 shows schematics and input characteristics for the three types of buffers. A circle on a transistor's gate means that the transistor is a P-channel device.

As *Figure 2* shows, the input buffers draw essentially zero I_{CC} when V_{IN} is 0.4V or less. This is also true when V_{IN} is 4V or more, except for type A. In other words, if the inputs are driven rail to rail, the B and C input buffers dissipate power only during the input signal transitions.

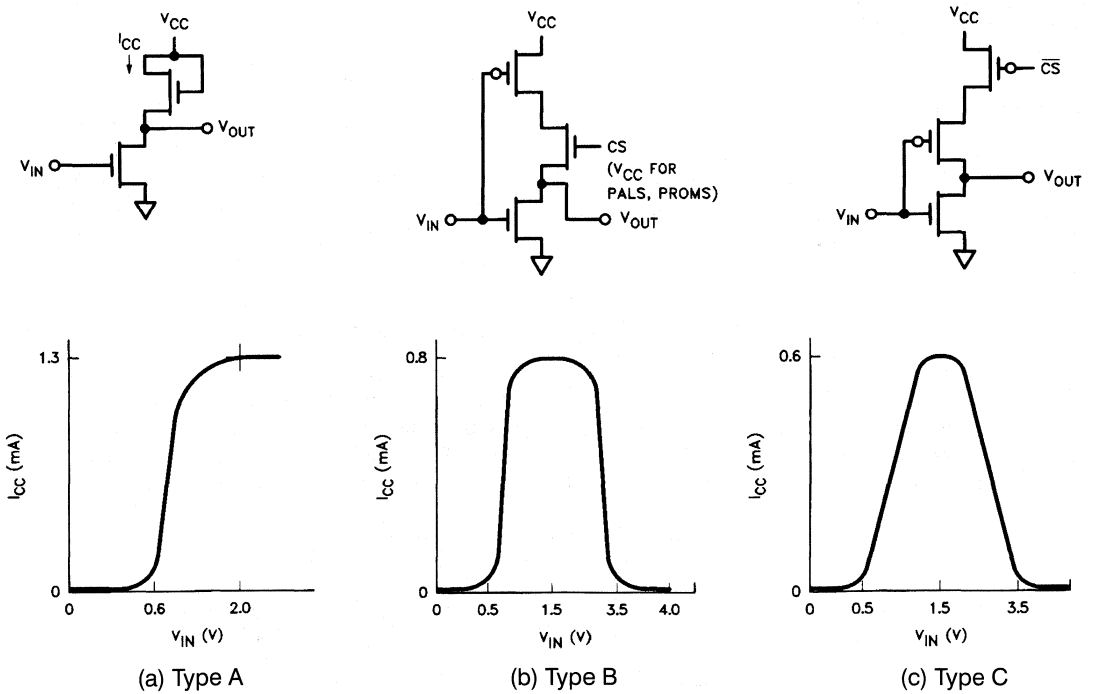


Figure 2. Three Buffer Types

Core and Output Buffers

The standby power dissipation of an IC's core derives from the substrate bias generator, reference generators, sense amplifiers, and polyload RAM cells or EPROM cells. This current is measured with $V_{IN} = 0V$ so that the input buffers draw no current. Under these conditions, the output buffers draw only leakage current and dissipate essentially no power.

Programming either PROMs or PALs stores charge on the floating gate of an NMOS transistor, which increases the transistor's threshold voltage. This higher threshold prevents the transistor from turning on during normal operation; unprogrammed transistors do turn on. Therefore, unprogrammed PALs and PROMs draw more current and dissipate more power than programmed devices.

The output buffers on Cypress products have N-channel pull-up devices that cause the output voltage level to reach

$$V_{OH} = V_{CC} - V_T = 5V - 1V = 4V$$

The capacitance of the output buffers, including stray capacitance, is typically 10 pF. If

$$C_L = 10\,pF,\ V_{OH} \approx 4V$$

Again, using *Equation 3*

$$I_{CC}(f) = 40 \times 10^{-12} f$$

for the output buffers.

Current Measurement

Figure 3 illustrates the instantaneous current drawn by a Cypress RAM. The instantaneous power is calculated by multiplying this current times the constant supply voltage, V_{CC}. Most of the power is dissipated during the access time. This is also true for PROMs and PALs.

The current measurement unit in an automatic tester integrates the instantaneous current over the measurement cycle and arrives at an equivalent average current. In other words, the average current, I_2, during time T_{CY} equals the area between the instantaneous current, i(t), and the X axis during T_{CY}. Thus, because the "current pulse" is effectively spread over a longer time when the frequency is decreased, the average current is proportionately lower.

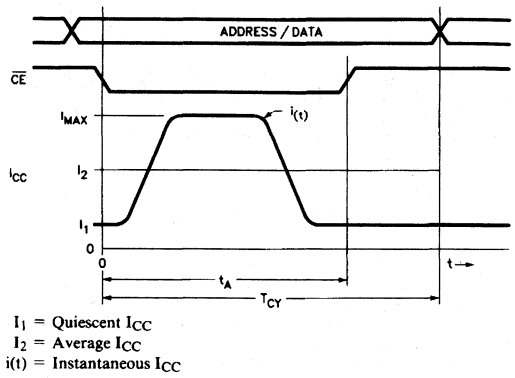


Figure 3. RAM I_{CC}

Note that the preceding calculations have not accounted for any DC loads. You must calculate these separately.

Product Characteristic Tables

Tables 2 through *5* allow you to calculate the current requirements for Cypress products. C_{INT} is the equivalent device internal capacitance, $I_{CC}(Q)$ is the quiescent or DC current, and $I_{CC(max.)}$ is the maximum I_{CC} (as specified on the datasheet) for the commercial operating temperature range. Conditions are $V_{CC} = 5V$ and $T_A = 25°C$.

Note that for the 16L8, 16R8, 16R6, and 16R4 PALs, the number of inputs and outputs is user configurable. All the PALs use type B buffers.

SRAM Calculation Example

To illustrate how to use *Tables 2* through *5*, consider an example of estimating the typical I_{CC} for the CY7C169–35 RAM at room temperature ($T_A = 25°C$) and nominal (5V) V_{CC}. Assume the duty cycle is 100 percent at the specified access time. The procedure shown here calculates the typical and worst-case I_{CC} with all inputs and outputs changing and with an output loading of 10 pF.

Table 2. Static RAMs

Part No.	Buffer Type	No. Inputs	No. Outputs	C_{INT} (pF)	I_{CC} (Q) (mA)	I_{CC} (max.) (mA)
CY7C109	C	29	8	170	45	160
CY7C122/123	A	16	4	24	50	90
CY7C128	B	14	8	27	59	120
CY7C130/131	C	21	10	120	6	170
CY7C132/136	C	22	9	120	6	170
CY7C140/141	C	22	9	120	6	170
CY7C142/146	C	22	9	120	6	170
CY7C147	B	15	1	34	28	90
CY7C148/149	B	12	1	32	45	90
CY7C150	B	18	4	20	44	90
CY7B160	C	25	4	55	40	130
CY7B161/162	C	22	4	55	40	130
CY7C161/162	B	22	4	300	13	70
CY7B164	C	20	4	55	40	130
CY7C164	B	20	4	300	13	70
CY7B166	C	21	4	55	40	130
CY7C166	B	21	4	300	13	70
CY7C167A	C	17	1	75	25	70
CY7C168A/169A	C	18	4	75	50	70
CY7C170A	B	18	4	50	33	90
CY7C171A/172A	B	18	4	100	27	70
CY7B185/186	C	24	8	60	47	150
CY7C185/186	B	25	8	330	13	100
CY7C187	B	19	1	150	7	100
CY7C189/190	B	10	4	21	32	90
CY7C191/192/ 194/195/196/197/ 198/199	C	18	8	115	15	170

Table 3. PROMs

Part No.	Buffer Type	No. Inputs[1]	No. Outputs	C_{INT} (pF)	I_{CC} (Q) (mA)	I_{CC} (max.) (mA)
CY7C225	B	12	8	32	35	90
CY7C235	B	13	8	35	35	90
CY7C245	B	13	8	35	50	90
CY7C245A	C	14	8	52	22	90
CY7C251	C	18	8	43	9.5	100
CY7C254	C	18	8	43	35	100
CY7C261/3/4	C	14	8	60	45	100
CY7C265	C	15	8	60	60	140
CY7C266	C	14	8	60	45	140
CY7C269	C	17	1/8	60	60	100
CY7C271/274	B	16	8	200	45	120
CY7C277/279	B	18	8	200	60	120
CY7C281/282	B	14	8	35	35	100
CY7C285	B	18	8	60	50	180
CY7C286	B	18	8	25	45	120
CY7C287	B	18	8	25	45	120
CY7C291/292	B	14	8	35	50	100
CY7C291A	C	14	8	52	22	120
CY7C292A	C	14	8	52	22	120
CY7C293A	C	14	8	52	22	120

Table 4. PALs

Part No.	C_{INT} (pF)	I_{CC} (Q) (mA)	I_{CC} (max.) (mA)
PALC16L8/R8/R4	40	25	45
PLDC20G10	50	30	55
PALC22V10	50	40	80
PALC22V10B	36	61	90
CY7C330	300	42	120

Table 5. Logic

Part No.	Buffer Type	No. Inputs[1]	No. Outputs	C_{INT} (pF)	I_{CC} (Q) (mA)	I_{CC} (max.) (mA)
CY7C401	B	6	6	53	30	75
CY7C402	B	7	7	53	30	75
CY7C403	B	7	6	53	30	75
CY7C404	B	8	7	53	30	75
CY7C408A	B	11	12	100	42	135
CY7C409A	B	11	13	100	42	135
CY7C420/421	C	14	12	190	18	142
CY7C421C	C	14	12	80	25	142
CY7C424/425	C	14	12	190	18	142
CY7C428/429	C	14	12	190	18	80
CY7C432/433	B	8	8	390	14	142
CY7C439A	C	13	11	40	55	147
CY7C441/443	C	13	11	200	15	180
CY7C451/453	C	15	12	200	15	180
CY7C510	C	24	19/16	60	30	100
CY7C516	C	28	16/16	60	30	100
CY7C517	C	28	16/16	60	30	100
CY3341	B	6	6	53	30	45
CY7C601	C	25	19/64	950	89	600
CY7C901	C	24	10/4	160	25	80
CY7C909	C	21	5	80	25	55
CY7C910	C	22	16	150	2.6	70
CY7C911	C	13	5	80	25	55
CY7C9101	C	36	22/4	70	30	60
CY7C9116	C	22	1/20	1000	35	150
CY7C9117	C	38	1/4	1000	35	150

From the RAM product characteristic table:

- Number of inputs = 18
- Number of outputs = 4
- C_{INT} = 75 pF
- I_{CC} = 50 mA

Because the input buffers on the CY7C169 are type C, the average current is 0.3 mA. If the input-signal-level transitions are 4V and the transition times are 2V/ns, the transition time is

$$T_t = \frac{4V}{2V/ns} = 2\ ns$$

The duty cycle is then

2 *ns*/35 *ns* = 0.057

Each input buffer thus draws

0.3 *mA* × 0.057 = 0.0171 *mA*

If all inputs change, the total transient input buffer current is

18 × 0.0171 = 0.31 *mA*

To calculate the CVf input buffer current:

I = *CVf*
C_{IN} = 5 *pF*
I = 0.57 *mA*
V = 4*V*
f = 1/35 *ns*

TOTAL = 18 × 0.57 = 10.28 *mA*

To calculate the internal CVf current:

I = *CVf*
C_{INT} = 75 *pF*
I = 10.71 *mA*
V = 5*V*
f = 1/35 *ns*

To calculate the output CVf current:

I = *CVf*
C_{OUT} = 10 *pF*
I = 1.15 *mA*
V = 4*V*
f = 1/35 *ns*

TOTAL = 4 × 1.15 = 4.6 *mA*

The quiescent current is 50 mA. The total current at T_{CY} = 35 ns is:

Input Transient	0.31 mA
Input CVf	10.28 mA
Internal CVf	10.71 mA
Output CVf	4.6 mA
Quiescent	50 mA
Total I_{CC}	75.9 mA (all inputs/outputs changing)

The worst-case transient current is 25.9 mA. If half the inputs and outputs change, the worst-case transient current decreases to 12.95 mA, which gives a total current of 63 mA (typical I_{CC}).

Note also that the input CVf current and the output CVf current have the same values for a bipolar device.

Worst-Case I_{CC}

Now consider a procedure for estimating I_{CC} for worst-case V_{CC} and low temperature, in addition to all inputs and outputs changing. Then you can compare the result with the I_{CC} specified on the datasheet.

I_{CC} is greater at high V_{CC}, which is 5.5V, or 1.1 times the nominal 5V V_{CC}. Because the increase in I_{CC} due to the lower temperature is 3 percent, the total increase is 13 percent. These factors apply to the internal CVf current (10.71 mA), the output CVf current (4.6 mA), and the quiescent current (50 mA), which together total 65.31 mA.

Total I_{CC} = *Input Transient* I_{CC}
+ *Input CVf* I_{CC}
+ *[Internal CVf + Output CVf +* $I_{CC}(Q)$*]* × 1.13
= 0.31 + 10.28 + [65.31 × 1.13] = 84.4 *mA*

This value is approximately 94 percent of the 90 mA specified on the datasheet. Note, however, that the datasheet I_{CC} maximum does not include the output CVf current.

I_{CC}-Versus-Frequency Characteristic

The I_{CC}-versus-frequency curves for all Cypress products have the same basic shape, which is illustrated by the PAL16R8 curve in *Figure 4*. The current remains essentially constant at the quiescent I_{CC} value until the frequency increases to the point where the capacitances begin to cause appreciable currents. The location of this point depends upon the input, internal, and output capacitances; the number of inputs and outputs; the rate at which the inputs and outputs change; and the voltage levels the inputs and outputs are switched between. For Cypress products, this point is in the 1- to 10-MHz range.

The PAL16R8 devices that were tested to obtain the data for the curve were exercised such that all inputs and outputs changed every cycle. Curve A shows the total I_{CC} for a 50-pF load on each of the eight outputs. Curve B shows the total I_{CC} when the outputs

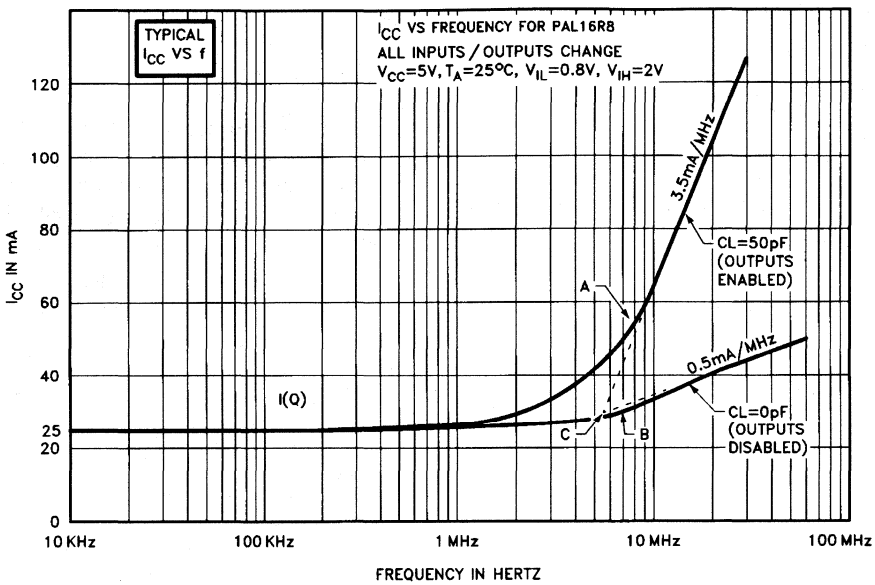


Figure 4. Typical I_{CC} vs. f

are disabled. The B curve results from the input and the internal capacitances. In most applications, the actual operation of the device falls somewhere between the A and B curves.

You can extrapolate the A and B curves backwards until they intersect the quiescent current, which occurs at point C in *Figure 4*. Point C is approximately 5.6 MHz. This gives you an easy-to-use formula for calculating I_{CC}. For frequencies less than 5.6 MHz:

$I_{CC} = I_{CC}(Q) = 25\ mA$

For frequencies greater than 5.6 MHz:

$I_{CC} = I_{CC}(Q) + 3.5\ mA/MHz$ *(all outputs changing)*

or

$I_{CC} = I_{CC}(Q) + 0.5\ mA/MHz$ *(no outputs changing)*

Calculating Power for the pASIC380

Since the pASIC380 family of devices is programmable, determining active power is difficult in that it is dependent on the functions implemented in the pASIC and the frequency of the internal nodes. To obtain a reasonably accurate estimate of the power consumption for a particular pASIC™ design, a calculation must be made that sums the power for each contributor in the device. This section presents the details of how this is accomplished.

Static Power

The pASIC family of devices does not have sense amplifiers, but they do have an internal bias generator, which typically uses about 2 mA for the 5-volt versions. This current is less for the 3-volt versions. The worst case static current is 10 mA for all pASIC380 devices.

Active Power

Active power arises from the energy required to move charge in and out of the load capacitances on the CMOS gates. A simple model of this is shown in *Figure 5*. The capacitance is composed of the in-

trinsic capacitance of the gate, the interconnect wire capacitance to ground, and the input capacitance of the gates to which the driving gate is connected. For the purpose of the model, the capacitance is lumped into the one capacitor in *Figure 5*. From energy considerations, the power consumed in moving charge on and off the capacitance is given by

$$P = CV^2f$$

where C is the capacitance, V is the voltage swing, which is usually the supply voltage (V_{CC}), and f is the average frequency at which the gate output changes logic states.

The calculation can be a consuming task. Each logic cell contains multiple gates each toggling at different average frequencies. Each logic cell can be connected to the inputs of other logic cells through various types and lengths of interconnect wires. With several thousand gates in the device, a power calculation based on the simple model is not a useful approach. A simplification is obtained by attributing an average capacitance to elements easily identifiable by the user.

These elements are:

- logic cell
- input buffer
- output buffer (unloaded externally)
- loads on high drive input buffers
- express interconnect wire
- clock input buffer
- clock distribution (internal column) buffer
- clock load

These elements are identified in *Figure 6*, which is an architectural representation of the pASIC family devices. Three of the elements in this list are explicitly identified load capacitances: loads on high drive input buffers, express interconnect wires, and clock loads. The average capacitances for each of these elements may not be directly due to a named element but will include interconnect wire capacitance and loads the element is connected to (given some

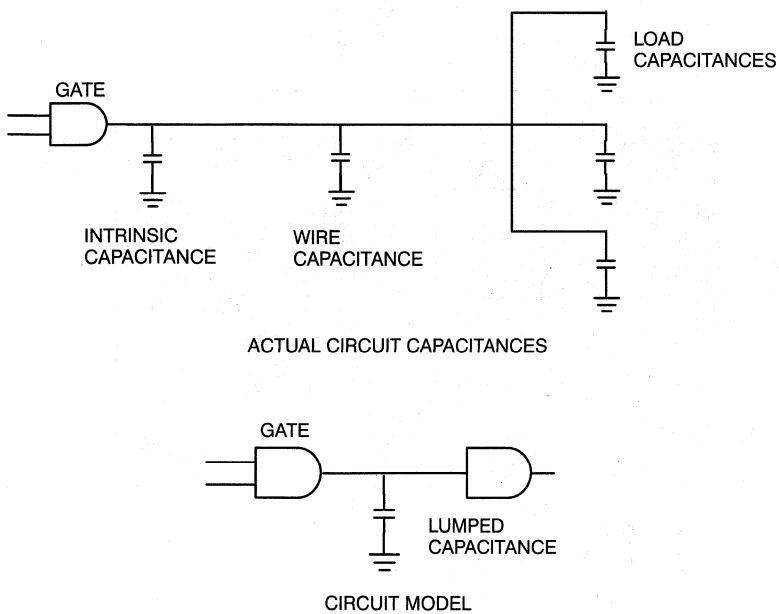


Figure 5. Capacitances in CMOS Circuits

average fanout). The capacitances for these elements will be referred to as an equivalent capacitance to reflect this averaging and the fact that the capacitance includes loads not necessarily in that particular element.

The equivalent capacitances are derived from empirical data to insure accuracy. Moreover, the measurements verify the averaging process and they verify the way the capacitances are attributed to the various elements. The equivalent capacitances for all the elements are given in *Table 6* for various average frequencies from 10 to 100 MHz. The equivalent capacitances are also plotted vs frequency in *Figure 7*. Given this data, the user only needs to know how many of each of these elements are used and their average frequency in order to estimate the power consumption. Not all elements are included in all members of the family. The individual datasheets should be consulted for further details.

With the capacitance in pF, the frequency in megahertz, and the resulting power in mW, the power equation can be expressed as

$$P_{mW} = C_{EQ} V_{CC}^2 f * 10^{-3}$$

This equation is in a form for practical use. The equivalent capacitance values, C_{EQ}, are obtained form the table or curves for the frequency of interest.

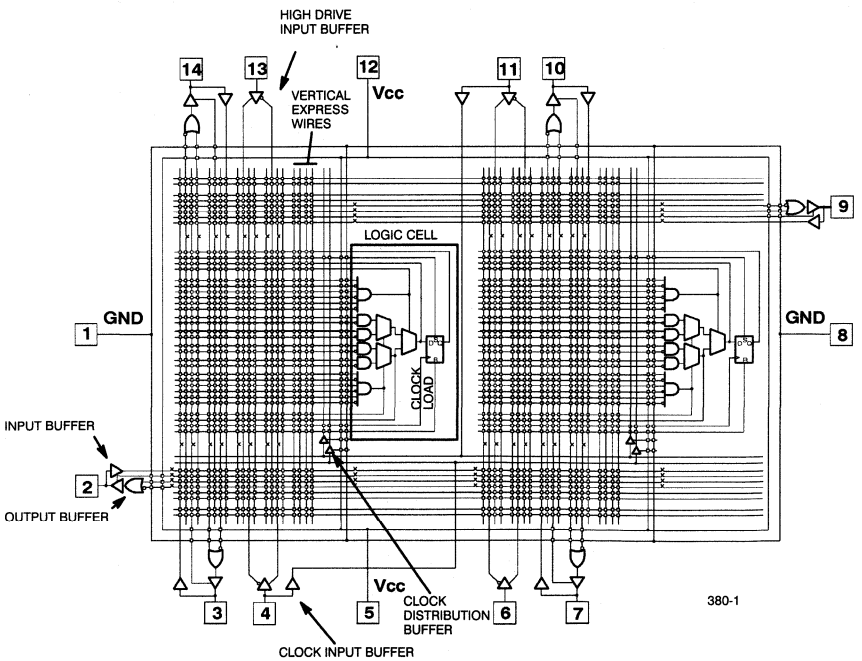


Figure 6. pASIC Internal Architecture

Table 6. pASIC380 Equivalent Capacitance (C_{EQ})

	10 MHz	20 MHz	33 MHz	50 MHz	66 MHz	80 MHz	100 MHz
Input	9.60	9.30	8.89	8.13	7.62	7.23	6.83
Output	15.73	15.83	16.04	15.55	16.60	16.04	14.39
Macro	17.47	17.02	16.57	15.78	14.15	13.42	12.69
HDbuffer Load	2.25	2.19	2.10	1.98	1.87	1.77	1.71
Vert HD Line	3.38	3.29	3.16	2.97	2.80	2.66	2.57
Clock Buffer	10.75	10.75	10.75	10.75	10.75	10.75	10.75
Clock Colbuf	4.67	4.67	4.67	4.67	4.67	4.67	4.67
Clock Load	1.11	1.11	1.11	1.11	1.11	1.11	1.11

Power Estimation Example

As an example of a power estimate, consider a 16-bit synchronous counter operating at 33 MHz. This example is for illustrative purposes only and the results should not be used for any other purposes. The counter clock is placed on a high-drive input-only pad (not one of the specialized clock input buffers) and is routed to the counter flip-flops through vertical express wires. When laid out, the counter occupies four columns in the array. All of the counter outputs are sent to output pins. Equivalent capacitance numbers are obtained from *Table 6* under the 33 MHz column.

First examine the 16 bit counter and the logic cells used to implement it. An analysis of this gives a simple result that can be used for the logic and output cell power calculation. The first flip-flop toggles at f/2, the next flip-flop toggles at f/4, the third at f/8, etc. The average per flip-flop toggle rate is

(f/16)*(1/2 + 1/4 + 1/8 ... + 1/65536)

or approximately f/16. The counter and the outputs can be considered as 16 logic cells and 16 outputs each toggling at f/16.

The power for all the elements can now be easily calculated.

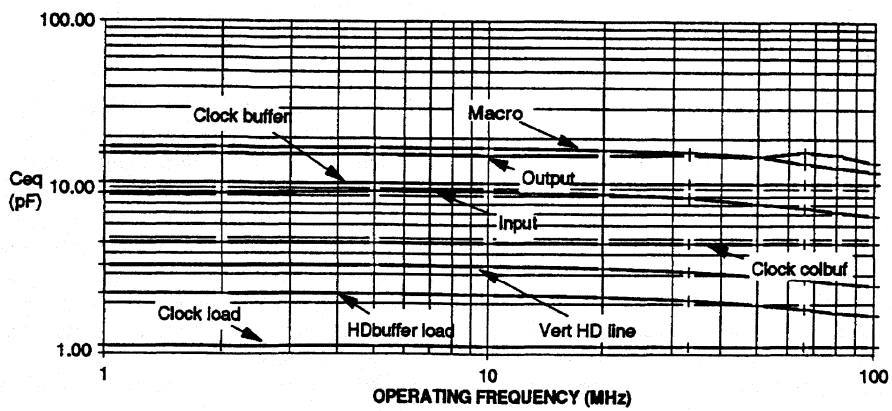


Figure 7. pASIC380 C_{EQ} vs. Operating Frequency

Logic Cells (each cell)

C_{EQ} = 16.57 pF

f = 33/16 MHz

P = 16.57 (33/16) * 5^2 * 10^{-3} = 0.83 mW

and for all 16

P = 16 * 0.83 = 13.28 mW

Input Buffer

This is a high drive input buffer for the clock

The input buffer itself

C_{EQ} = 8.89 pF

f = 33 MHz

P = 8.89 (33) * 5^2 * 10^{-3} = 7.33 mW

The vertical express wires (4)

C_{EQ} = 3.16 pF

f = 33 MHz

P = 4 * 3.16 (33) * 5^2 * 10^{-3} = 10.43 mW

The high drive buffer loads (clocks on 16 flip−flops)

C_{EQ} = 2.10 pF

f = 33 MHz

P = 16 * 2.10 (33) * 5^2 * 10^{-3} = 27.72 mW

The output buffers (total for all 16)

C_{EQ} = 16.04 pF

f = 33/16 MHz

P = 16 * 16.04 (33/16) * 5^2 * 10^{-3} = 13.23 mW

Adding all of the contributors to the power, the total dynamic power for the counter is 72.51 mW. To this must be added the maximum quiescent power of 50 mW (10 mA max. specification) giving a total of 123 mW. This power calculation does not include the power resulting from external loads on the device pins.

Obtaining Values for the Calculations

The difficult part of the calculations is obtaining values for the number of logic cells used, the number of clock buffers used, and the average toggle frequency. There is no prescription for determining these numbers. However, there are aids to this process. These aids will be discussed in this section.

Consider a 16-bit counter different from the one in the previous example. This new counter will use the internal clock distribution tree. The task of obtaining the number of logic cells and clock buffers used is aided by the Physical View in the *Warp3*™ tool. *Figure 8* shows the physical view for a 16-bit synchronous counter using the internal clock distribution tree. The number of logic cells used in the counter can be easily counted; there are 27. With the physical view displayed in SpDE, the user can obtain a summary of the cell utilization. This is done by selecting Cell Utilization under Info. There are 16 output buffers and one clock input buffer, as expected. The upper/lower column division is between row 5 and row 6. Therefore, the clock is distributed to both the upper and lower half of columns A and B, whereas only the lower half of columns C, D, E, and H receive clocks. Columns A and B will use two clock internal buffers each (one for the lower half column and one for the upper half column) and there will be one each for columns C, D, E, and H. The results for clock distribution are:

clock input buffer	1
clock internal buffers	8
clock loads	16

All of the components of the active power have been identified. From the previous example, the average frequency for the flip-flop logic cells, the output buffers, and the clock related buffers and loads is known. Eleven of the logic cells are combinatorial and need to be examined more closely. These logic cells must be part of the excitation logic for the counter flip-flops. There are several approaches. The most direct approach is to examine the physical view and, for each cell in question, examine the origin of the inputs and the destination of the output and determine heuristically the approximate logic function being implemented in the cell. Knowing this, the average toggle rate can be estimated. This

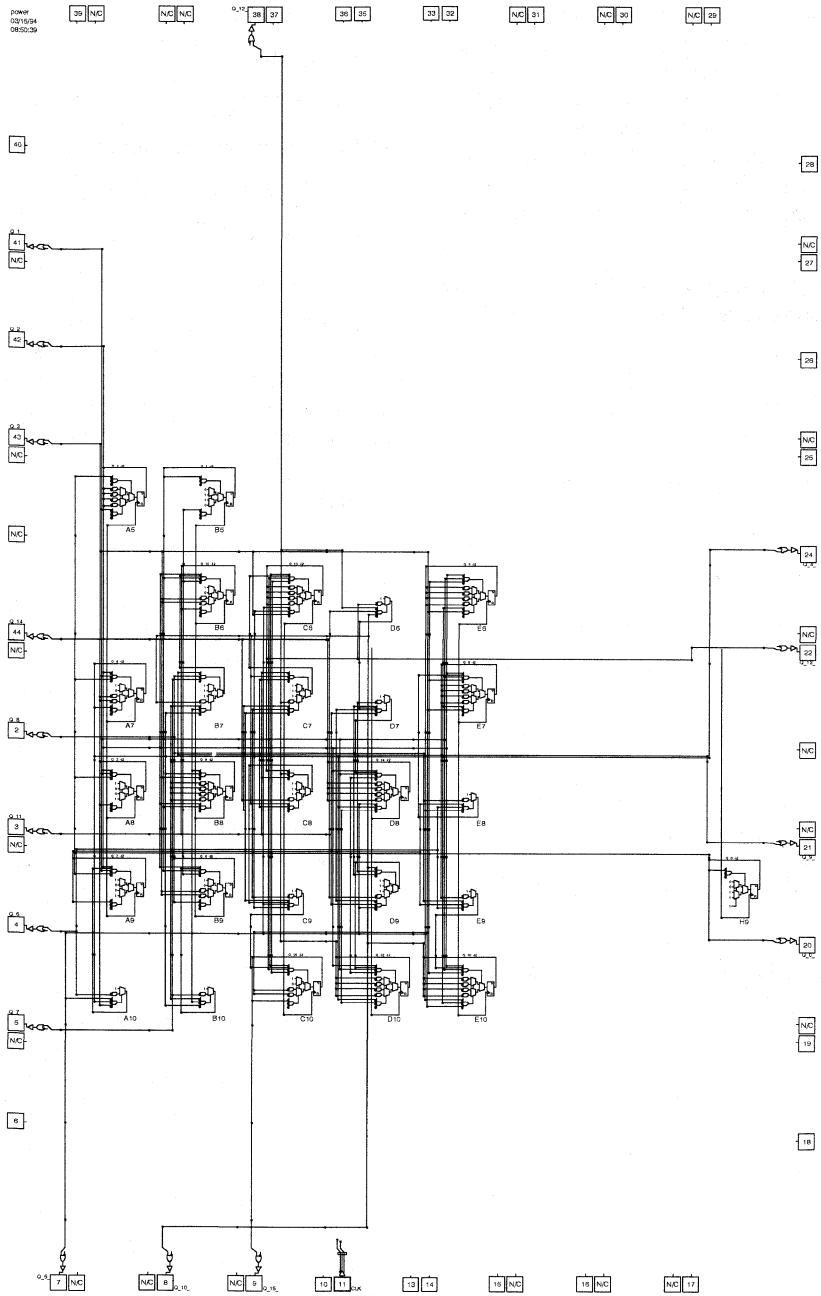


Figure 8. 16-Bit Counter Physical View

approach can be time consuming and difficult if there is a large amount of circuitry in the design. An alternative is to use approximations to an advantage. An approach used earlier was to attribute an average toggle frequency to each flip-flop of the counter. A simple extension of this approximation suggests that each of these combinatorial cells be estimated as having an average toggle frequency of f/16.

Using the above data, the power for this 16-bit counter is determined as follows:

Logic Cells (each cell)

C_{EQ} = 16.57 pF

f = 33/16 MHz

$P = 16.57\ (33/16) * 5^2 * 10^{-3} = 0.85$ mW

and for all 27

$P = 27 * 0.85 = 22.95$ mW

Clock Input Buffer and Distribution Tree

The input buffer itself

C_{EQ} = 10.75 pF

f = 33 MHz

$P = 10.75\ (33) * 5^2 * 10^{-3} = 8.87$ mW

The clock column buffers (8)

C_{EQ} = 4.67 pF

f = 33 MHz

$P = 8 * 4.67\ (33) * 5^2 * 10^{-3} = 30.82$ mW

The loads (16)

C_{EQ} = 1.11 pF

f = 33 MHz

$P = 16 * 1.11\ (33) * 5^2 * 10^{-3} = 14.65$ mW

The output buffers (total for all 16)

C_{EQ} = 16.04 pF

f = 33/16 MHz

$P = 16 * 16.04\ (33/16) * 5^2 * 10^{-3} = 13.23$ mW

The total dynamic power for the counter is 90.52 mW. Adding, as before, the maximum quiescent power of 50 mw, the total power becomes 140 mW. This power calculation does not include the power resulting from external loads on the device pins.

Conclusion

This application note provides algorithms and reference data for calculating power consumption in a broad range of Cypress devices. The calculation methodology varies from structured devices such as PROMs to programmable devices such as pASICs. All calculations for active power are based on *Equation 4*. The accuracy of the results is related to the determination of the capacitance and the frequency. In many cases, significant power dissipation is a result of driving external loads. Users should make certain that the device power calculations include the power associated with the external loads.

Note

1. Bidirectional pins.

Document #: GEN0003–0592

Input/Output Characteristics of Cypress Circuits

Introduction

This application note presents, explains, and analyzes the I/O (Input/Output) characteristics of the ICs (Integrated Circuits) manufactured by Cypress Semiconductor.

A brief history of bipolar, ECL, and CMOS digital ICs is presented to illustrate I/O circuits, topology, and fabrication. The input current, output current, fan-in, fan-out, maximum/minimum I/O voltage levels, and noise immunity were originally defined for RTL Micrologic and are explained here. The application of these definitions to subsequent logic families is then examined.

The I/O characteristics of CMOS, BiCMOS, FCT-T, and ECL ICs designed and manufactured by Cypress are then presented and analyzed.

A Brief History of Bipolar Integrated Circuits

The first family of digital ICs was the Micrologic family, introduced in 1961 by Fairchild Semiconductor. In 1963 Fairchild introduced a lower-power version, called Milliwatt Micrologic.

Both of these families used RTL (Resistor Transistor Logic) circuits and were fabricated using bipolar NPN transistors. Motorola, TI (Texas Instruments), and others second-sourced RTL.

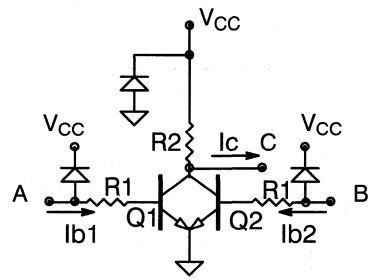


Figure 1. RTL NOR Gate

RTL Circuit Fabrication

Figure 1 shows the basic RTL two input NOR gate and *Figure 2* shows a cross section of the topology. The P substrate is grounded and N wells are used for the NPN transistors and P resistors. The NPN base and the P material resistors are fabricated during the same diffusion step. The N wells are reverse biased by connecting them to the V_{CC} power supply. The P substrate and the P+ isolation between the wells are connected to ground. As a result, PN junction diodes are formed between each of the wells and the substrate, thus isolating the transistors from the resistors.

Input Clamping Diodes to V_{CC}

The input diodes are called input clamping diodes because they clamp the input signal to V_{CC} + Vf if a voltage more positive than V_{CC} is present at the input pin. Under "normal" operating conditions an

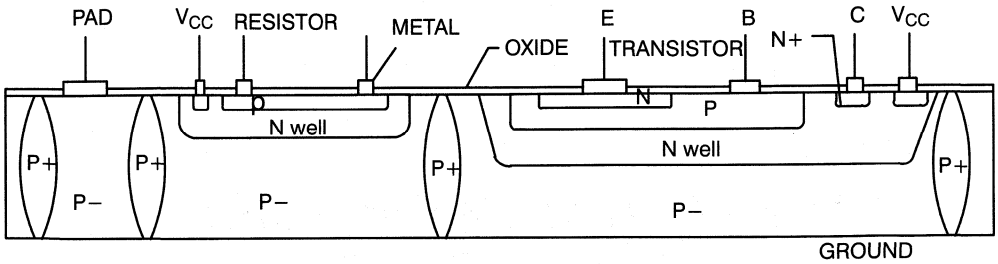


Figure 2. RTL Circuit Topology

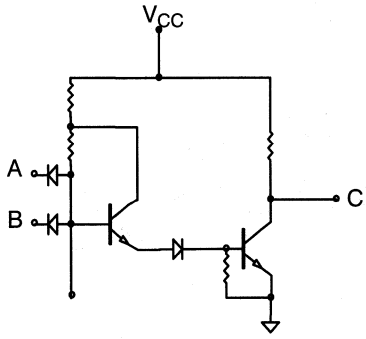


Figure 3. Early DTL

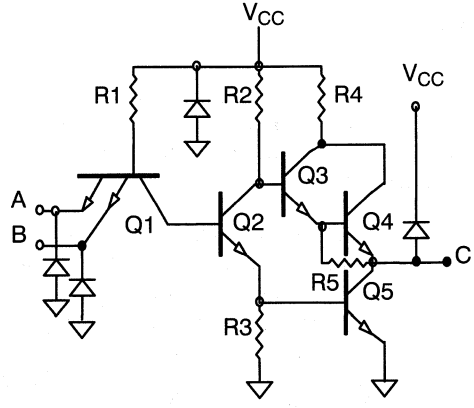


Figure 4. Early TTL

input pin will not see this high a voltage. However, signal overshoot due to lead inductance could cause positive voltages greater than V_{CC} on input pins.

Output Clamping Diodes

The same parasitic diodes occur at the output pin. They were present on all early junction isolated bipolar ICs.

DTL and TTL Logic Families

In the mid 1960s Signetics introduced the SE 101 DTL (Diode Transistor Logic) family of bipolar ICs. The diodes at each input are fabricated using multi-emitter transistors whose base and collector are connected together. Fairchild second-sourced the Signetics DTL family, calling it the "Fairchild DTuL 930" family. The schematic of a two-input gate is shown in *Figure 3*.

For the positive logic convention the DTL gate performs the logical NAND function, and for the negative logic convention it performs the logical NOR function. The reason for this is that the NAND and NOR functions are logical duals; i.e., if all of the zeros are replaced with ones and all of the ones are replaced with zeros in a NAND truth table, it becomes a NOR truth table. The converse is also true.

A clever circuit designer at Fairchild, Dr. Bob Seeds, realized that if the base-collector connection of the input stage was removed, the input stage would still perform the same logical function, but the emitter current would be multiplied by the hfe, or beta of the input transistor (see *Figure 4*). Thus TTL was born.

Signetics SE 101 DTL died instantly, and Fairchild blazed ahead; but not for long. A sleeping giant of a company named TI (Texas Instruments) in Dallas, Texas, was observing all of this and made its move. While Fairchild was trying to manufacture its 9300 family of TTL using one-tenth mil technology in 1964, TI introduced its 7400 family, which Motoro-

la, National, and Signetics quickly second-sourced. By 1966 the 7400 series was the TTL standard. Fairchild, being a proud company, did not second-source 7400 until Lester Hogan became its president in June of 1968, after Noyce, Moore, and Grove departed to found Intel.

Input Clamping Diodes to Ground

Advances in process technology, as well as active pull-ups (pull-up transistors, not resistors), caused signal rise and fall times to decrease to approximately five nanoseconds. Voltage reflections caused by transmission line effects began to appear. To suppress negative voltages, clamping diodes to ground were added to all inputs and outputs by performing an N diffusion under the bonding pads.

These diodes are called input clamping diodes because they clamp the input to −0.8V if a negative voltage is present at the input pin. Under "normal" operating conditions an input pin will not see a negative voltage. However, either signal undershoot due to lead inductance or a voltage reflection due to transmission line effects could cause negative voltages on input pins.

The presence of these input diodes is verified by applying a negative voltage on each input pin (that is sufficient to forward bias the parasitic diodes) and measuring the current (15 to 20 mA) that can be "pulled out" of the pin. Note that this test does not require that the V_{CC} power supply be connected. For silicon PN junction diodes, the forward voltage threshold is approximately 0.7 to 0.8V at room temperature (25 degrees Celsius). The datasheet test typically specifies a minimum of 18 mA at −1.2V.

Series 7400 and its variations (AS, LS, S, ALS) dominated the general-purpose logic family market until approximately 1984, at which time PALs (Programmable Array Logic) and PLDs (Programmable Logic Devices) overtook them in sales.

Early ECL (Emitter Coupled Logic)

Motorola introduced the MECL III family in 1968, followed by the MECL 10K family in 1971, the MECL 10KH family in 1981, and the ECLiPSE family in 1987.

Bipolar NPN transistors and resistors are used in these ICs. The output is usually an open emitter; sometimes a small series damping resistor is included. The input and output impedances are approximately seven ohms.

Most ECL ICs have 50-kΩ input pull-down resistors to ground so that unused inputs may be left open. Because bipolar transistors are current-sensing devices, they will not turn on if their bases are left open.

Early CMOS

The RCA 4000 series (circa 1975) was the first CMOS logic family. The combination of high input impedances and thin (1000 Angstroms in 1975) gate oxides make MOS transistors susceptible to ESD (ElectroStatic Discharge). The substrate clamping diode will not provide ESD protection unless it breaks down in the reverse direction, in which case the high current through the low-resistance reverse biased diode will cause it to short circuit.

Personnel and handling equipment can generate static voltage potentials in excess of 10 kV in a low-humidity environment.

Current-limiting input resistors and junction isolation dual-diode protection circuits were (and still are) used to prevent high voltages from either rupturing the thin gate oxides or overstressing them such that they fail later. The strategy is to prevent the high voltages from reaching the thin gate oxides by (1) causing the non-destructive breakdown of either a bipolar or a MOS transistor, and (2) limiting the current when the breakdown occurs so that it does not destroy the transistor.

Logic designers using series 4000 discovered another unique characteristic of CMOS, which is also true today. If an input (pin) is left open (unconnected or floating), within a few hundred milliseconds charge will accumulate at the gate until it is charged up to the device threshold and both the N-channel pull-down and the P-channel pull-up transistors will turn on, resulting in a low impedance path between V_{CC} and ground. In the 4000 series the channel ON resis-

tances were so low that the I_{CC} current destroyed the input devices. In normal operation both transistors will be partially on only during the time that the input signal transitions between the logic levels.

Logic Circuit Conventions: Currents

The input and output current conventions were defined for RTL, the first logic family, and are still followed today. With reference to *Figure 1*, the input currents Ib1 and Ib2 are defined as being positive into the device. The output current, Ic, is defined as being positive out of the device.

The purpose of the base resistors (R1= 450Ω) is to limit base current and reduce "current hogging" by one of the inputs when more than one input is driven by an output. Current hogging is caused by the variation of $V_{BE\,(sat)}$ of the transistors due to process parameter variations (between wafer runs of devices in different packages) and is exacerbated by variations in the integrity of the ground between packages.

The threshold or turn-on voltage occurs at approximately V_{BE}=0.7V for NPN transistors at 25 degrees Celsius. When either Q1 or Q2 are on, the voltage at point C becomes approximately 0.2V, which is $V_{CE(sat)}$ of the respective transistor. These fundamental values of $V_{BE(sat)}$ and $V_{CE(sat)}$, and the fact that they are different, form the basis of all saturated logic circuit design.

For PNP transistors the same relationships exist.

When the voltage at the A and B inputs are less than the V_{BE} thresholds of the transistors, both Q1 and Q2 are off. The output, node C, is then at V_{CC}. In this condition the minimum current, Ic, which can be supplied (sourced) by node C is determined by the maximum value of R2. The minimum value of R2 is determined by the hfe or beta of each transistor (since Ie= Ib + Ic, where hfe=Ic/Ib), which is in turn a function of the transistor geometry and process parameters. The maximum value of the base resistor, R1, is thus determined by hfe also, because a minimum amount of base current is required to sustain a given collector current.

RTL Transfer Characteristic

Figure 5 illustrates the voltage transfer function of the RTL NOR gate. All inverting logic has the same shaped curve. When the input voltage is less than or equal to V_{IL} (voltage In Low, or maximum logic zero) the output is guaranteed to be greater than or equal to V_{OH} (voltage Out High, or minimum logic one output), which is point A on the curve. Similarly, when the input voltage is equal to or greater than V_{IH} (voltage In High, or minimum logic one input) the output is guaranteed to be less than or equal to V_{OL} (voltage Out Low, or maximum logic zero output), which is point B on the curve. All inverting logic has the same backward-S-shaped transfer characteristic.

It is important for input signals to transition quickly through the "linear region" between V_{IL} and V_{IH}. Within this region they amplify noise, which could cause unpredictable results.

Noise Immunity Defined

The low or logic zero noise immunity is defined as $N_{IL} = V_{IL} - V_{OL}$ and the high or logic one noise immunity is defined as $N_{IH} = V_{OH} - V_{IH}$. These are typically shown as in *Figure 6*. The switching point or logic threshold, V_T, should be centered between V_{IL} and V_{IH} and should be insensitive to variations in processing, as well as supply voltage and operating temperature.

The absolute value of the noise immunity is not as important as the ratio of the noise immunity to the

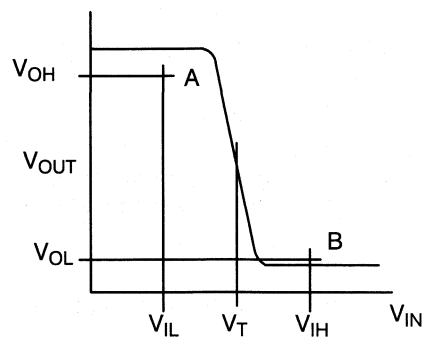


Figure 5. Inverting Logic Transfer Function

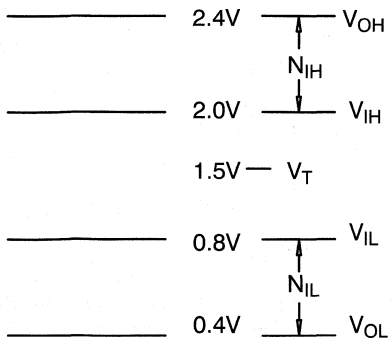


Figure 6. TTL Voltage Levels

logic swing; i.e., as a percentage of the output voltage swing, $V_{OH} - V_{OL}$. We shall also see that it is not quite correct to just specify a voltage for noise margin. It is really an energy consideration, so a voltage across an impedance for an amount of time is what should really be specified.

TTL Logic

RTL and DTL are obsolete and, except for HC and ECL, the rest of the world is TTL compatible. GTL (Gunning Transceiver Logic) is emerging as a low-voltage, high-performance interface standard, but that is another story. Cypress CMOS and BiCMOS ICs are TTL compatible. What this means is that circuits conforming to the TTL voltage levels are guaranteed to interface to each other.

Operation of the TTL Gate

The classical TTL gate shown in *Figure 4* performs the logical NAND function if the positive logic convention is used, or the logical NOR function if the negative logic convention is used. We will use the positive logic convention.

Consider first the condition where the output, C, is HIGH, which means that either one or both of the inputs, A or B, must be LOW. This is a given because we know that the TTL gate performs the logical NAND function. Transistors Q3 and Q4 form a darlington pair, whose output current is limited by R3. The output high voltage, V_{OH}, is thus $V_{CC} - V_{BE}(Q4) - V_{BE}(Q3)$. Q5 must be off, and for it to be off Q2 must also be off. Transistor Q1 must be on if either node A or node B is grounded. Current flows out of nodes A or B, opposite to the direction of current flow of the RTL gate of *Figure 1*, so it is negative. The original amount of TTL input current (I_{IN}) was -1.6 mA per input. The output pull-down transistor, Q5, was designed to sink 16 mA. Therefore, the fanout was ten.

Next, in order to determine the threshold point, connect nodes A and B together and begin to raise the voltage at the input until the gate output, C, switches from HIGH to LOW.

The threshold point is in the range of 1.4V to 1.6V (most folks say 1.5V), depending upon whether 0.7V or 0.8V is used for V_{BE}; i.e., The threshold point is $2V_{BE} - V_{CE(sat)}$, where $V_{CE(sat)}=0.2V$. This may be verified by analysis of *Figure 4* as follows.

If node C is LOW, Q5 must be on, so Q2 must be on and Q1 must be off. In order to turn Q1 off, the voltage at nodes A and B must just be equal to $V_T = V_{BE}(Q5) + V_{BE}(Q2) - V_{CE(sat)}(Q1)$. The base emitter junction is reverse biased and the base collector junction of Q1 then becomes forward biased, thus clamping the base of Q1 at $V_{BC}(Q1) + V_{BE}(Q2) + V_{BE}(Q5)$. The threshold is, therefore, very precisely determined between $3V_{BE}$ and $2V_{BE} - V_{CE}$. Note how the threshold has been increased over that of RTL.

TTL Problems and Performance

Early bipolar transistors suffered from excess charge storage in the base collector and base emitter regions, which slowed down their turn-off time. Gold doping the base was used for many years to reduce minority carrier storage time, but was replaced by less expensive Schottky diodes. Because the transistor turn-on time was faster than the turn-off time "current spiking" occurred in the output totem pole drivers (Q3 through Q5) during high-to-low output signal transitions. In other words, Q5 would turn on before Q3 and Q4 were off, thus causing transient short circuits between V_{CC} and ground, which generated electrical noise (EMI and RFI).

Maximum t_{PLH} (propagation delay, LOW to HIGH) was 22 ns, and maximum t_{PHL} (propagation

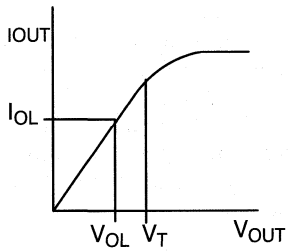


Figure 7. Output Low

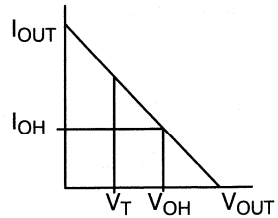


Figure 8. Output High

delay, HIGH to LOW) was 15 ns. Typical delays were one half of the maximums. Signal rise and fall times were ten nanoseconds.

TTL Voltage Levels

The TTL voltage levels are: V_{IL} = 0.8V, V_{OL} = 0.4V, V_{IH} = 2.0V, V_{OH} = 2.4V. On paper, the low-level voltage noise immunity ($V_{IL} - V_{OL}$ = 0.4V) and the high level voltage noise immunity ($V_{OH} - V_{IH}$ = 0.4V) are the same. However, because the output impedance in the logic HIGH state is much greater than the output impedance in the logic LOW state, the high-level and low-level noise immunities are not the same. It takes more energy to develop the same voltage across a low impedance than across a high impedance.

TTL Output Characteristics

Output buffers must drive PCB traces and capacitive loads, whether they are fabricated using bipolar or MOS transistors. From an equivalent circuit point of view, looking back into the output of an output buffer is, to a first order approximation, independent of technology.

Output Pull-Down Device

Figure 7 illustrates a typical output buffer characteristic in the low state. The pull-down transistor is turned on and appears as a resistance to ground. The slope of the I_{OL} vs. V_{OL} curve is the equivalent resistance of the pull-down transistor. The original TTL specification is I_{OL} = 16 mA at V_{OL} = 0.4 V, which gives an output (worst case) resistance of 25 ohms. Typical resistance was 10 ohms.

In the mid 1970s TI introduced LS (Low-power Schottky) that was the same speed as TTL, but used one-fifth the power. The input current for LS is −0.4 mA and the output sink current is 8 mA, giving a fanout of 20. The LS output specification is I_{OL} = 8 mA at V_{OL} = 0.4 V, which gives an output resistance of 50 ohms.

The multiple emitters of the LS inputs were eventually replaced with substrate PNP transistors, which further reduced the input (negative) current to tens of microamperes.

Output Pull-Up Characteristic

Figure 8 illustrates the output source current vs. voltage for a typical output buffer. The slope of the line, typically 50 to 100 ohms, is the equivalent on resistance of the pull-up transistor plus any current limiting resistor. The Y-axis intercept is the short circuit current. The X-axis intercept (zero current) is $V_{CC} - 2\ V_{BE}$.

HC CMOS

In 1983 the HC (High Current) CMOS and HCT (High Current TTL compatible) CMOS families were introduced. HC is CMOS I/O compatible and HCT is TTL I/O compatible. Maximum input current is plus or minus one microampere. There are two output current specifications; one for driving TTL loads and the other for driving CMOS loads. The TTL specification guarantees 4 mA at $V_{OH}=V_{CC}$ − 0.8V and −4 mA at V_{OL}=0.4V. The CMOS specification guarantees 20 microamperes at $V_{OH}=V_{CC}$ − 0.1V and −20 microamperes at 0.1V.

TTL Driving HC CMOS

The input voltage threshold for HC is $V_{CC}/2$. $V_{IH} = 0.7\,V_{CC}$ and $V_{IL} = 0.2\,V_{CC}$. $V_{OL} = 0.1V$ and $V_{OH} = V_{CC} - 0.1$ V. This means that HC cannot be reliably driven by standard TTL at $V_{CC} = 5$ V, because the output V_{OH} of TTL is too low: 2.4V for TTL and 3.5V for HC V_{IH}. In reality, the TTL V_{OH} is $V_{CC} - 2\,V_{BE} = 3.6V$, which is still marginal.

A one-to-ten-thousand-ohm pull-up resistor to V_{CC} is recommended on every TTL (compatible) output that drives a HC input. The maximum value of the pull-up resistor is determined by the acceptable rise time of the signal due to the RC time constant. The minimum value of the pull-up resistor is determined by the amount of current the output pull-down transistor driving the node is capable of sinking.

HC CMOS Driving TTL

HC CMOS can drive circuits with TTL-compatible inputs without any problems.

Cypress CMOS

Cypress CMOS ICs are TTL compatible. The input voltage threshold is 1.5V and the output V_{OH} is equal to V_{CC} - V_T, where $V_T = 0.7V$ to 1V (worst case). The threshold temperature coefficients of NMOS transistors are −1.1 mV per degree Celsius and those of PMOS transistors are −1.8 mV per degree Celsius.

CMOS Input Buffers

Figure 9 shows a typical Cypress input buffer, or TTL to CMOS level translator. A circle on the gate of a transistor indicates that it is a P-channel device. The gate of the P-channel pull-up device whose drain is connected to V_{CC} may either be controlled by a chip enable or tied to ground.

Figure 10 shows the current drawn from the V_{CC} supply as the input voltage varies from 0 to 3.5 volts. At the threshold voltage of 1.5V, all three devices are on and there is a low-impedance path between V_{CC} and ground. Normally this is a transient condition if the input is driven by a Cypress CMOS output buffer whose signal rise and fall times are two nanoseconds between (approximately) zero and four volts.

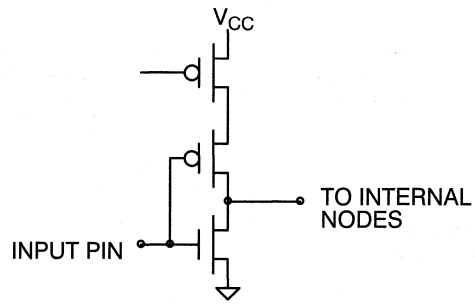


Figure 9. Input TTL to CMOS Buffer

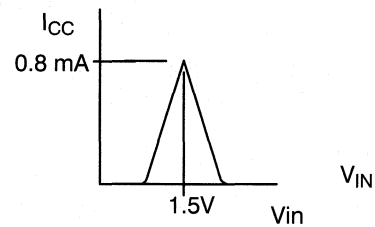


Figure 10. Input Buffer Current

However, if an input is left open (floating), within 100 milliseconds it will charge up to the threshold, all three devices will turn on, and a low impedance path will exist between V_{CC} and ground. This needlessly wastes current and, because the circuit operates in the linear region, it amplifies noise and applies it to following (internal) circuits, as well as couples it to nearby (internal) circuits. All unused inputs should be connected either to ground or to V_{CC}; whichever is logically correct, but preferably ground.

Input Current

The equivalent circuit of a Cypress CMOS input pin (under normal operating conditions) is a 10-pF capacitor in parallel with a 5-million-ohm resistor. Very little energy is required for the input N-channel transistor to turn on. For all practical purposes,

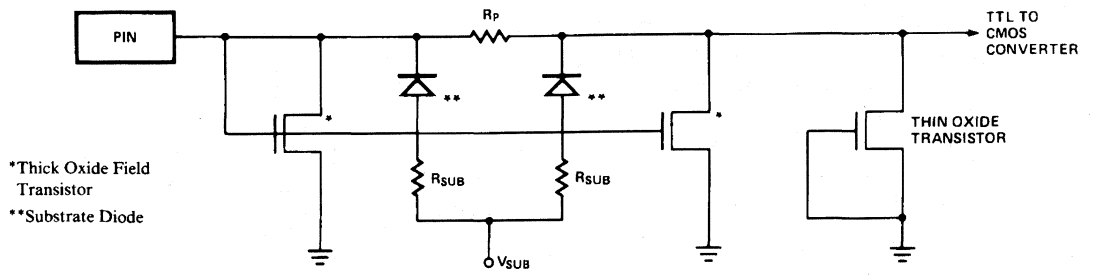


Figure 11. Input CMOS Protection Circuit

driving CMOS inputs is simply charging and discharging the parasitic input capacitances.

Input ESD Protection

Large electrical fields can damage the thin gate oxides of MOS transistors. The thicker the oxide, the larger the voltage it can withstand before it is either ruptured or stressed to the point that it may fail later.

The circuit shown in *Figure 11* is used at every input pin in all Cypress CMOS products to provide protection against ESD. This circuitry has been designed to withstand repeated applications of high voltages without failure or performance degradation. This is accomplished by preventing the high (ESD) voltage from reaching the thin gate oxides of the internal transistors.

The circuit consists of two thick oxide (field) transistors wrapped around an input resistor (Rp) and a thin oxide (gate) transistor with a relatively low breakdown of 12 volts. Large input voltages cause the thick oxide transistors to turn on, discharging the ESD current to ground. The thin oxide transistor breaks down when the voltage across it (drain to source) exceeds 12 volts. It is protected from destruction by the current limiting action of Rp.

Experiments have confirmed that this input protection circuitry results in ESD protection of in excess of 2000 volts. The next question to be answered is what happens when a negative voltage appears on an input pin. This can occur due to either a voltage reflection as a result of transmission line effects, or lead inductance.

The anodes of the substrate diodes are connected through the bulk resistance (R_{SUB}) of the substrate to V_{SUB}, which is generated by the substrate bias generator (*Figure 10*). The substrate is maintained at a negative-three volt potential, so that the substrate diodes cannot be forward biased unless the voltage at the input pin becomes a diode drop more negative than three volts. To the systems designer, this translates to approximately five times (3.8V divided by 0.8V = 4.75) the negative undershoot safety margin for Cypress CMOS ICs versus those that do not use a bias generator; i.e., where the substrate is grounded.

Definition of Latch-Up

Latch-up is a regenerative phenomenon that occurs when the voltage at an input pin or an output pin is either raised above the power-supply voltage potential or lowered below the substrate voltage potential, which is usually ground. Current rapidly increases until, in effect, a short circuit from V_{CC} to ground exists. If the (V_{CC}) current is not limited, it will destroy the device, usually by melting a metal trace.

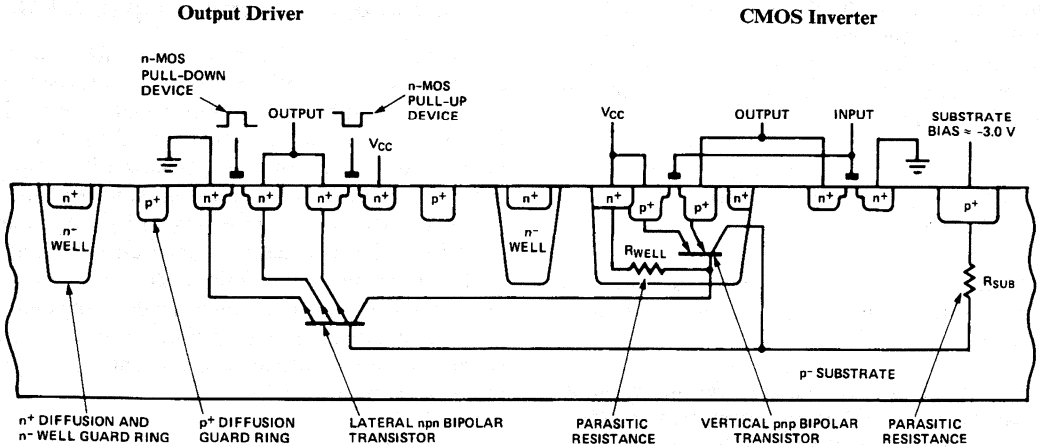


Figure 12. CMOS Cross Section and Parasitic Circuits

Causes of Latch-Up

The CMOS processing, which provides both N-channel and P-channel MOS transistors, also inherently provides parasitic bipolar transistors; both NPNs and PNPs. Latch-up is caused when these parasitic transistors are inadvertently turned on.

As long as the voltages that are applied to the package pins of the CMOS IC remain within the limits of the power supply voltages (usually 0 volts to 5 volts), these parasitic bipolar transistors will remain dormant (i.e., off). However, when either negative voltages or positive voltages greater than the V_{CC} supply voltage are applied to input or output pins, these parasitic bipolar transistors may turn on and cause latch-up.

Conditions for Latch-Up

A cross section of a typical CMOS inverter using a P-channel pull-up transistor and an N-channel pull-down transistor is shown in *Figure 12*. Also shown is a N-channel output driver that is isolated from the CMOS inverter by a guard ring (channel stopper) that is necessary to prevent parasitic MOS transistors between devices. P+ guard rings surround N-channel devices and N+ guard rings surround P-channel devices. The parasitic SCR (PNPN) and bias generator are illustrated in *Figure 13*. The output driver schematic is not shown.

In order for latch-up to occur two conditions must be satisfied: (1) the product of the betas of the NPN and PNP transistors must be greater than one, and (2) a trigger current must exist that turns on the SCR.

Since the SCR structure in bulk CMOS cannot be eliminated, preventing latch-up is reduced to keeping the SCR from turning on. If either $R_{well}=0$ or $R_{sub}=0$ the SCR cannot turn on because the base emitter junction of the PNP cannot be forward biased because they are tied together and the base

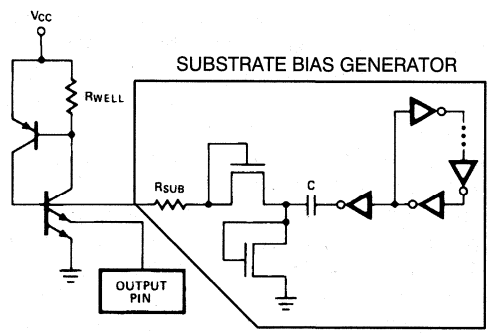


Figure 13. Parasitic SCR and Bias Generator

emitter junction of the NPN cannot be forward biased because the base is connected to ground. Note, however, that the NPN could be turned on by a negative voltage on the output pin (if the right end of R_{sub} is grounded).

Prevention of Latch-Up: Traditional Approaches

The traditional cures include increased horizontal spacing, diffused guard rings and metal straps to critical areas.

A brute-force approach that has been successful in reducing latch-up has been to increase the conductivity of the N well and the substrate. Changing the well conductivity is unacceptable because it affects the characteristics of the P-channel MOS transistors. Wafers with an epitaxial layer that reduces the substrate resistivity (Rsub) are used for all Cypress BiCMOS, CMOS RAM2.5 and greater, and ECL products.

The Cypress Solution to Latch-Up

Cypress uses several design techniques in addition to careful circuit layout and conservative design rules to avoid latch-up.

NMOS Output Pull-Up Transistors

Conventional CMOS technology uses a P-channel MOS as a pull-up transistor on the output drivers. This has the advantage of being able to pull the output voltage HIGH level to within 100 millivolts of the positive voltage supply. However, this is of marginal value when TTL compatibility is required. In addition, the P-channel pull-up is sensitive to overshoot and introduces another vertical PNP transistor that further compounds the latch-up problem. Cypress uses N-channel pull-up transistors on all CMOS outputs, which eliminate all of these problems and still maintain TTL compatibility. The outputs of all CMOS (not BiCMOS) devices rise to $V_{CC} - V_T$, where V_T is the threshold of the N-channel pull-up transistor, and is (worst case) one volt.

Substrate Bias Generator Up to RAM2

Cypress is the first company to use a substrate bias generator with CMOS technology. The bias generator keeps the substrate at approximately −3 volts DC, which serves several purposes. As previously mentioned, the parasitic diodes shown in *Figure 9* cannot be forward biased unless the voltage at an input pin is at least one diode drop more negative than −3 volts. This translates into increased device tolerance to (negative voltage) undershoot at the input pins, caused by inductance in the leads. If the undershoot is this large, the output impedance of the bias generator itself is sufficient to prevent trigger current from being generated.

Output Pins

The same reasoning applies to negative voltages at the output pins as shown in *Figure 13*. In order to turn on the NPN transistor the voltage at the output pin must be at least one VBE more negative than −3 volts.

To protect the "core" of the die from free-floating holes and stray currents, a diffused collection guard ring that is strapped with metal and connected to the bias generator is used. This provides an effective wall against transient currents that could cause misreading of the EPROM cells in PROMs or PLDs, or could cause memory cells to "flip" in RAMs, dual-port RAMs, or FIFOs.

BiCMOS Products

Bipolar and CMOS technologies are combined in many of the newer Cypress ICs and the resultant products are called BiCMOS. In these products, as well as in CMOS RAM2.5 and greater, the substrate bias generator has been eliminated and there are internal clamping diodes, one to V_{CC} and one to ground at most input and output pins. There are two types of BiCMOS products; those that are TTL compatible and those that are ECL compatible.

The TTL-compatible BiCMOS products are generally identified by a "B" in the part number (e.g., CY7B144) and will be discussed first. The I/O characteristics are very similar to those of the CMOS devices, except that the V_{OH} output level is $V_{CC} - 2V_T$

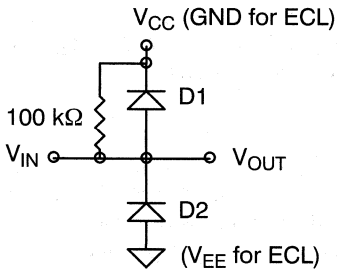


Figure 14. BiCMOS (ECL) Input Protection

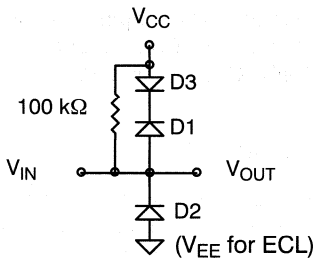


Figure 15. ECL Programming Pin Protection

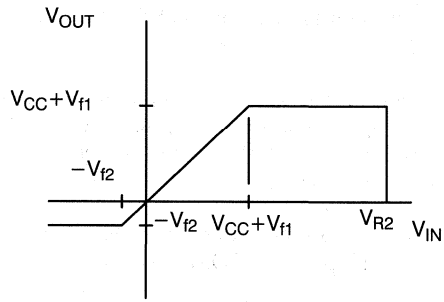


Figure 16. Transfer Characteristic

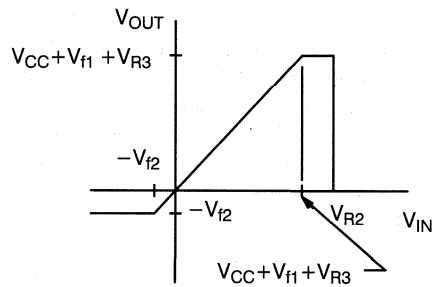


Figure 17. Transfer Characteristic

(n), or V_{CC} minus two N-channel transistor thresholds. The V_{OH} level specified on the datasheet is 2.4V.

Figure 14 illustrates the protection circuit used on input pins as well as output pins on most pins of most BiCMOS products. The pins used for high voltages during the programming of programmable devices use the circuit shown in *Figure 15*. All diodes have a forward voltage threshold of 0.7V and a reverse breakdown voltage of approximately fifteen volts.

If a voltage greater than V_{CC} plus the forward voltage drop of diode D1 is applied to the input pin or the output pin of the circuit of *Figure 14*, D1 will turn on and clamp it to V_{CC} + 0.7V until the voltage reaches fifteen volts (V_{R2}), at which time diode D2 will breakdown, thus clamping the voltage to zero volts.

Similarly, if a negative voltage is applied to the same circuit, diode D2 will turn on at −0.7V and clamp it to −0.7V. This transfer characteristic is shown in *Figure 16*.

The circuit of *Figure 14* cannot be used on certain programming pins because the programming voltages required are between 5 and 10 volts. Therefore, the circuit of *Figure 15* is used on these pins. As can be seen from the transfer characteristic of *Figure 17*, adding diode D3 moves the clamping point out by the reverse breakdown voltage of diode D3, which is shown as V_{R3}.

The 100-kΩ pull-up resistor to V_{CC} causes open inputs or outputs to be pulled up to V_{CC}.

PROMs

The first Cypress PROMs (actually EPROMs) used 1.2-micron PROM I technology, the substrate bias generator, and the ESD protection circuit of *Figure 11*. The 256-Kbit PROMs and those with an "A" suffix use the 0.8-micron PROM II technology, which

uses the same bias generator and ESD protection as PROM I.

PROM output pins are used as data inputs during programming. Therefore, they must have ESD protection. The circuit of *Figure 11* is used on these output pins with R_P=0.

Note that there are no clamping diodes to V_{CC} on input or output pins of PROMs.

FCT-T (Fast CMOS TTL-compatible) Logic Family

The FCT-T logic family consists of high-performance, low power, CMOS integrated circuits that either meet or exceed the speed and drive capability of their bipolar functional equivalents. This family achieves the performance of bipolar devices, but uses CMOS technology and typically one-third the power of the bipolar devices.

FCT-T is TTL compatible, which means that it conforms to the industry-standard TTL voltage levels and threshold point, and operates from a five Volt V_{CC} power source. The TTL threshold point is 1.5 Volts. All inputs are designed to have 200 mV of hysteresis, which means that the LOW to HIGH threshold point is 1.6V and the HIGH to LOW threshold point is 1.4V. The benefit to the user is increased static and dynamic noise immunity, as well as less sensitivity to noise superimposed on slowly rising or falling inputs.

FCT-T

The FCT-T logic family features output buffers that use n-channel pull-up transistors and controlled rise and fall time edge rates. Typical unloaded output signal rise and fall times are two nanoseconds. The maximum unloaded output high voltage, V_{OH}, is V_{cc} minus the n-channel threshold, V_T. The transistor drain is connected to V_{CC}, so V_T is approximately 1V worst case and typically 0.5V. The unloaded output is typically 4 Volts and the loaded V_{OH} is typically 3.3 Volts when sourcing 15 mA with a V_{CC} of five Volts.

The reduced output voltage swing of FCT-T results in lower crosstalk. The controlled edge rates reduce crosstalk as well as groundbounce.

FCT2-T

The FCT2-T logic family is identical to the FCT-T logic family, except that the FCT2-T devices have a 25-Ohm resistor in series with the output. The purpose of the resistor is to provide series damping when driving a transmission line. These products with series damping resistors should be used only when driving lumped (or single) loads, and should not be used for driving multiple or distributed loads. For a description of series damping, see the application note "System Design Considerations When Using Cypress CMOS Circuits" in the Cypress *Applications Handbook.*

CMOS Process Technology

FCT-T products are manufactured using the Logic 2.7 process and are fabricated in a Class 1 facility on six inch wafers. The minimum drawn channel length is 0.65 microns and the effective channel length is 0.5 microns. The process uses one layer of polysilicon, two layers of metal, and an epitaxial layer to achieve a 400-picosecond loaded internal gate delay. There is no substrate bias generator. In addition to providing high density, the technology assures latch-up protection, single event upset protection, and excellent ESD protection.

Input and Output Clamping Diodes

All input and output signal pins have clamping diodes to ground. There are no clamping diodes to V_{CC} on any input or output signal pin. Input and bidirectional pins have ESD protection circuits, but they will not become active until the voltage at the pin reaches twelve to fifteen volts.

ECL I/O Characteristics

The ESD and clamping diode circuits used at input and output pins of non-programmable ECL products manufactured by Cypress are shown in *Figures 18* and *19*, respectively. The circuit of *Figure 18* is used at ECL inputs and the circuit of *Figure 19* is used at ECL outputs. Some ECL products have

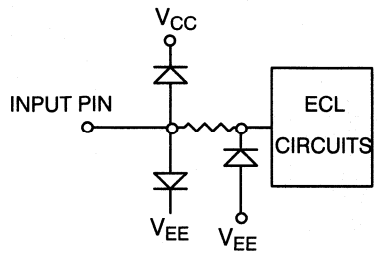


Figure 18. ECL Input ESD Protection

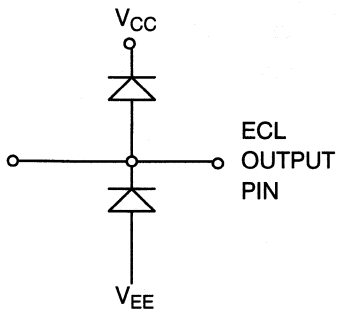


Figure 19. ECL Output ESD Protection

some TTL compatible inputs and outputs, e.g., CY101E383 ECL/TTL Translator and High-Speed Bus Driver, and use the circuit shown in *Figure 15* on both input and output pins. The reverse breakdown voltage of the diodes is approximately 20 volts and their series resistance is 5 ohms in the reverse direction. The CY10E383 and CY101E383 are the only Cypress ECL products that have internal 2-KΩ pull-down resistors to V_{EE} on all ECL outputs.

On the chip select inputs of ECL RAMs there are 50-kΩ pull-down resistors to ground. There are no series resistors or pull-down resistors on ECL outputs; only the emitter of the output NPN transistor.

The characteristic impedance looking into an ECL input or an output is approximately seven ohms.

ECL Input and Output Levels

ECL is a type of CML (Current Mode Logic) in which current is steered (switched) through one half or the other half of a non-saturating differential amplifier. Because of this, ECL has a constant current verses frequency characteristic.

The first ECL family, 10K, is not temperature compensated. The 100K family has temperature compensation for the internal voltage reference and the input threshold. The 10K compatible devices manufactured by Cypress are not temperature compensated and the 100K devices are.

The transfer curves for 10K and 100K ECL are shown in *Figures 20* and *21*, respectively. The cross-hatched area in *Figure 20* represents the operating ranges of the input and output voltages.

ECL Switching Characteristics

The output voltage swing of Cypress ECL circuits is approximately one volt in 500 ps (picoseconds), which is a slew rate of 2V/ns. The rise and fall times are specified between the 20% and 80% voltage points for ECL. Typical rise and fall times are 0.7 ns, which translate to 6.25 ns between 0 and 100%. Rise and fall times must not exceed 2 ns (20% to 80%) or 3.3 ns (0 to 100%), or circuit operation may be unreliable and cause unpredictable results.

In clocked circuits, if the clock is slowly changing during its transition through the linear region and there is noise superimposed on it, the circuit may interpret the noise as multiple clocks, thus causing mis-sampling of data, mis-counting of counters, or both. Because of the fast rise and fall times of ECL circuits, all signals must be terminated.

Bus Interface Products

The I/O characteristics of the following bus interface products are described in this section, starting with *Table 1*.

Table 1. Bus Interface Products

Product Number	Description
VIC068A	VMEbus Interface Controller
VAC068A	VMEbus Address Controller
VIC64	VMEbus Interface Controller with D64 Functionality
CY7C964	Bus Interface Logic Circuit

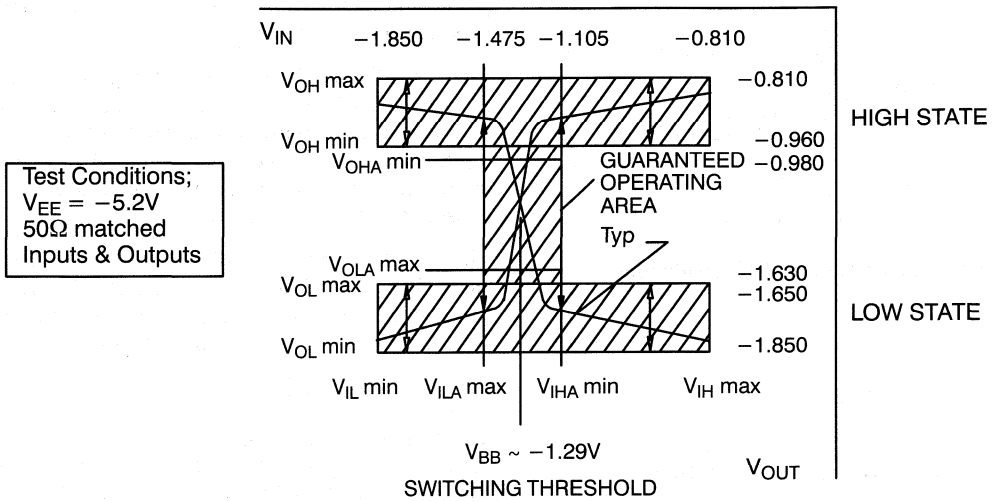


Figure 20. 10K ECL Transfer Curves and Specification Points

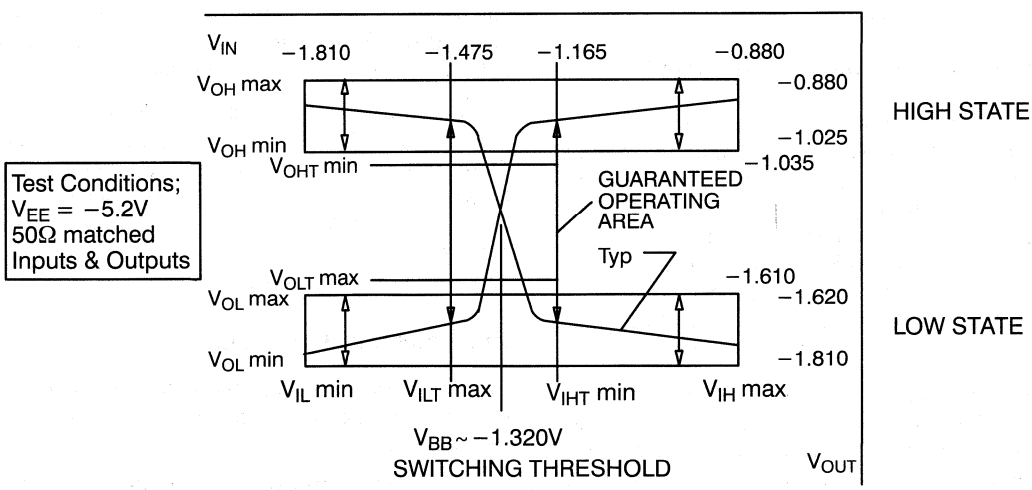


Figure 21. 100K ECL Transfer Curves and Specification Points

These circuits are fabricated using a 1.2-micron CMOS process with an epitaxial layer. The outputs that are required to interface to the VMEbus are able to drive the bus directly and to meet the applicable electrical specifications.

Three types of circuits drive the VMEbus; totem-pole (or push-pull), three-state, and open collector (drain). The Cypress totem-pole and three-state outputs use P-channel pull-up transistors that pull-up to V_{CC} − 0.1V on all output and I/O pins EX-

CEPT those that sink 64 mA, which have N-channel pull-ups.

The VMEbus electrical signal level specifications are as follows:

V_{OH} = 2.4V at I_{OH} = 3 mA, 400 µA (depending upon signal)

V_{IH} = 2.0V at $I_{IH} \leq$ 400 µA, 450 µA, 600 µA, 700 µA

V_{IL} = 0.8V at $I_{IL} \leq$ 50 µA, 100 µA, 150 µA

V_{OL} = 0.6V at I_{OL} = 8, 48, 64 mA (depending upon signal)

The V_{OL} specification is 200 mV higher than the TTL standard, but the VMEbus is TTL compatible.

The maximum capacitive loading (per slot) is 20 pF for bidirectional signals and 12 pF for inputs or outputs.

The VMEbus specification states that all signals should have clamping that shall prevent negative voltage excursions below −1.5V. Clamping diodes to ground are not mandatory.

The bus interface circuits capable of sinking 64 mA have clamping diodes to V_{CC} and ground as shown in *Figure 19* on the I/O pins.

Also, there is a 400-ohm series resistor (not shown) between the pin and the input buffer.

The bus interface circuits capable of sinking 16 mA have the circuit illustrated in *Figure 22* on the I/O pins. Note that there are no diodes to ground.

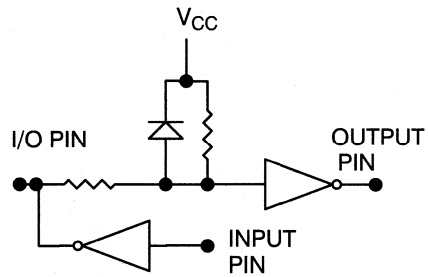


Figure 22. Bus Interface IC ESD

Document #: GEN0004−0892

PROMs – 2



PROM Section Contents and Abstracts

Introduction to the State Machine PROM 2–1

This application note introduces the CY7C258/9 2K x 16 State Machine PROMs (designed specifically for state machine applications) and discusses the design of a video signal generator using these devices. First, a description of the requirements of a state machine is presented. Second, the benefits of using PROMs as state machines are enumerated. This is followed by an architectural description of the State Machine PROM with emphasis on the various features that make the device especially well-suited for state machine applications. Finally, the design of a video signal generator using the State Machine PROM is described.

Generating PROM Programming Files 2–19

This application note introduces PROMs to the user and then explains the methods of generating PROM programming files. A brief description of PROM usage in systems is presented followed by a discussion of various PROM programming file formats, including the Intel, Motorola, DEC, and Tektronix formats. Finally, the application note discusses various methods of using high-level languages (ABEL HDL, ISDATA, LOG/iC HDL, BASIC, C) to generate PROM programming files.

Introduction to the CY7C270 Processor-Intelligent PROM 2–32

This application note introduces the CY7C270 Processor-Intelligent (PI) PROM, the industry's first UV-erasable PROM with a user-programmable processor interface. First, an architectural description of the CY7C270 PROM is presented followed by a discussion of the advantages of using the CY7C270 PROM instead of conventional PROMs. Finally, the application note describes the design of non-volatile memory for various popular RISC and CISC processors (960, 29K, x86, 68040) and DSP devices (TMS320C30) using the CY7C270.

Interfacing the CY7C276 High-Speed PROM to the AT&T, AD, Motorola, and TI DSPs 2–51

This application note discusses how to use the CY7C276 PROM as program memory for various DSPs. It will cover the topic of interfacing the CY7C276 high-speed PROM to some of today's most popular DSPs for program memory only. Data memory storage is typically done with SRAM and its interface is not included in this application note. The AT&T DSP1616, Analog Devices ADSP 2100A, Motorola DSP56000 and TI TMS320C5x family of devices are discussed. Also included is a detailed description of the CY7C276 (including architecture, programming options, and signal descriptions) and brief descriptions of the DSPs (architectures, signals and timing requirements). For ease of explanation, only one example from each product family is included. The other devices in each product family are similar and are left as an exercise for the reader. Detailed timing calculations that show code sizes up to 16K words in depth are included in the examples. Finally, a table is provided to help summarize the analysis.

Designing Custom ALUs and Multipliers with PROMs 2–59

PROMs can be used in a multitude of applications where discrete parts can be numerous and slow. The PROM approach can increase system reliability by reducing part counts and signal routing problems. With the advent of high-speed, dense PROMs, the number of application opportunities for PROMs continues to increase. Remember, when your high-speed application requires a special function, use a PROM.

Introduction to the State Machine PROM

Overview

This application note introduces the CY7C258/9 2K x 16 State Machine PROMs (designed specifically for state machine applications) and discusses the design of a video signal generator using these devices. We begin with a description of the requirements of a state machine. Next, the benefits of using PROMs as state machines are enumerated. This is followed by an architectural description of the State Machine PROM with emphasis on the various features that make the device especially well suited for state machine applications. Finally, the design of a video signal generator using the State Machine PROM is described.

Requirements of a State Machine

Figure 1 shows the block diagram of a synchronous state machine. The state machine can be logically segmented into the following sections.

Inputs

If an input is asynchronous with respect to the state machine clock, it needs to be synchronized to avoid race conditions and metastability. A synchronous input can be fed directly into the state machine.

State Vector and Feedback

The state vector describes the present state of the state machine. The state vector bits are fed back to the input of the decode logic where they form the to-

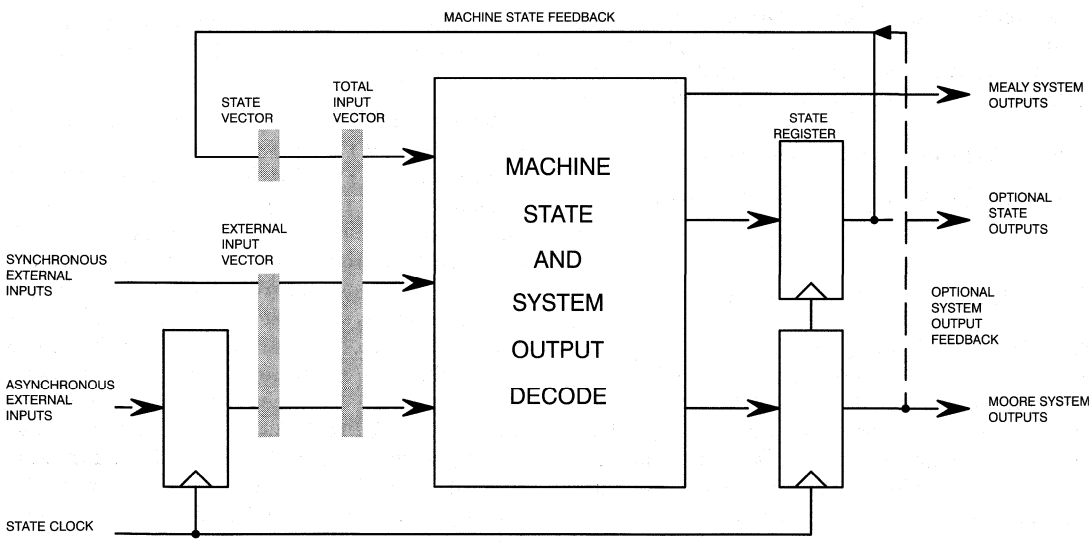


Figure 1. Synchronous State Machine

tal input vector along with the external inputs. The next state of the state machine depends on the total input vector at the inputs. The feedback is always registered to eliminate glitches at the outputs.

Outputs

There are two kinds of outputs from a state machine, *Mealy* and *Moore*. State machines having these outputs are called Mealy and Moore machines respectively. In a Mealy machine, the input conditions can alter the system output during a specific state. Hence Mealy outputs are implemented in state machines where the response to a change in the input vector needs to be immediate and not delayed until the next clock. On the other hand, in a Moore machine the system outputs depend only on the present machine state. The system outputs will remain stable until the next clock cycle, when the state machine samples the total input vector to determine the next state.

Moore machines are relatively safer to implement than Mealy machines because every signal in a Moore machine is clocked. If the various set-up and hold times are met, the outputs will be glitch-free. On the other hand, a state machine may have Mealy outputs if the machine needs to respond to a certain input condition immediately. However, the Mealy outputs will change only after a time equal to the propagation delay through the device on which the state machine is implemented (PROM, PLD, etc.).

Machine State and System Output Decode

This block generates the state bits and system outputs.

A PROM as a State Machine

A PROM is the simplest example of a programmable logic device. It consists of a fixed AND array and a programmable OR array. A PROM with n inputs can generate 2^n product terms and all product terms are available at the outputs. Hence, no logic reduction is required for state machine designs implemented in PROMs. The only design limitation of using PROMs for state machine design is the number of inputs and outputs. If the design meets the I/O constraints of a particular PROM, the design will always fit, no matter what last-minute logic or pinout changes are made.

The advantages of using PROMs as state machines are

- No state minimization is required, because each state corresponds to a unique memory location. When using a PLD for state machine designs, software is required to minimize the number of states in the state machine.
- They can hold large state machines. The only size restriction is the number of I/O pins available, as opposed to using PLDs, where the number of product terms limits the size of state machines.
- Last-minute design changes will always fit into the PROM, whereas in PLDs the product term limitations may restrict last-minute design changes.
- Users can tell immediately if the state machine will fit in the device. Two conditions determine if the design will fit:
  - The number of outputs of the state machine + number of state bits $\leq$ the number of outputs of the PROM.
  - The number of inputs of state machine + number of state bits $\leq$ number of inputs to the PROM.

When using PROMs as state machines, an external feedback path is required to return the state vector to the input of the PROM. These vectors need to be registered. Cypress integrated an output register on-chip to create output-registered PROMs, which have been very successfully used in state machine applications. However, when using output registered PROMs as state machines, feedback is external, forcing the designer to account for the set-up time of the register and resulting in a slower speed of operation. Besides, the output register is not individually configurable bit-by-bit, making it impossible to implement both Mealy and Moore outputs from the same device. These limitations were the primary factors in designing the State Machine PROM.

Architectural Features of the State Machine PROM

The State Machine PROM family consists of two devices, the CY7C258 and the CY7C259. Each device has a 2K x 16 PROM array and differs only in packaging. The State Machine PROM tries to incorporate all the requirements of state machines on-chip to make designing easier. *Figure* 2 illustrates the architecture of the State Machine PROM.

Input Registers

The input registers are useful for capturing addresses that appear for very short periods of time. Each address line has an individual register which can be configured to capture the address at the rising edge of the clock or be bypassed.

In addition, the set-up and hold times of these registers are programmable. For addresses that appear centered around the rising edge of clock, the register can be programmed to have a 2-ns set-up time and a 2-ns hold time (on the fastest part). If the address disappears immediately after the rising edge of clock, then the input registers can be programmed to have a 5-ns set-up time and a 0-ns hold time. Note that the set-up and hold time configuration is valid for all input registers simultaneously, meaning that the user cannot have different set-up and hold times for different input registers.

Output Registers

Output registers are used to hold both state information and output data. Each output register can be individually bypassed, giving the user the flexibility to implement a state machine with either Mealy outputs or Moore outputs or a combination of both. This is not possible with an output-registered PROM because the output registers are not individually bypassable.

Internal Feedback

The design feature that gives the State Machine PROM its high speed is the internal feedback. Since there are 11 address inputs to the PROM, 11 of the 16 outputs of the PROM array are fed back to the

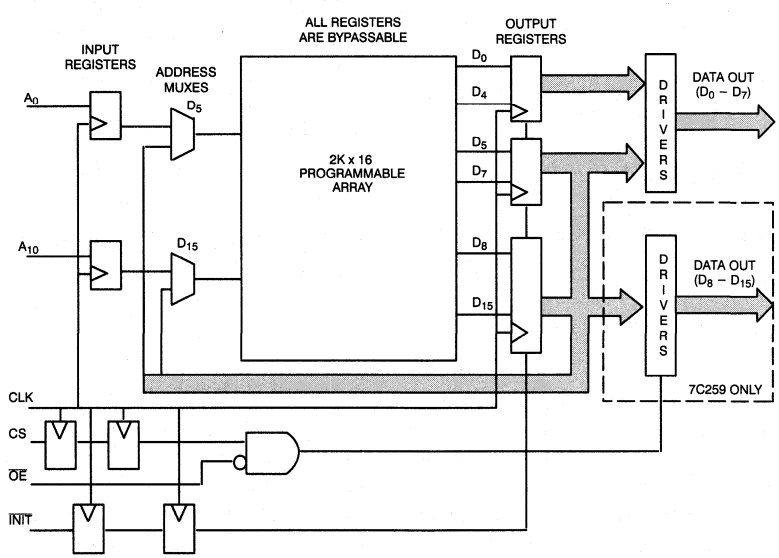


Figure 2. Architecture of the State Machine PROM

inputs. The difference between the CY7C258 and the CY7C259 is in the number of the feedback lines available at the output pins. On the CY7C258, only three of the feedback lines are available at the output pins. With a total of 8 output pins, the CY7C258 fits in a 28-pin package. On the CY7C259, all 11 feedback lines are available at the output, thus providing greater flexibility to the user. The CY7C259 is available in 44-pin LCC and PLCC packages.

2K x 16 Programmable Array

The 2K x 16 programmable array offers the user 16 bits at the output to design the state machine. Comparing this to presently available output registered PROMs, which have an 8-bit wide PROM array, the State Machine PROM provides a lot more flexibility and eliminates the need for additional external devices when implementing a state machine design.

Input Multiplexer

Each input address line has a 2:1 multiplexer in its path. One input is connected to the output of the address register, and the other input comes from the internal feedback lines. This means the user can individually configure the address lines to accept input from either the input pins or the feedback lines.

Programmable Initialize Word

The State Machine PROM allows the user to have a user-programmable initialize word stored in the device. When the $\overline{\text{INIT}}$ pin is sampled LOW on the rising edge of clock, this initialize word is sent to the output (and hence the feedback lines) of the State Machine PROM, thus allowing the user to reset the state machine into a known state. Note that the initialize word is loaded into the output registers. Therefore, if the output register for an individual output is configured to be bypassed, the corresponding bit of the initialize word will not appear at the outputs. However, since the initialize word is loaded into the output registers, it will be fed back to the inputs, thus initializing the state machine into a known state.

Registered Chip Select

The CS input to the State Machine PROM has two registers, each of which can be bypassed just like the input registers. In addition, the chip select input is polarity programmable.

Versatility of the State Machine PROM

Although the State Machine PROM is ideally suited to state machine applications, it can be used as a general-purpose PROM because its inputs and outputs are completely configurable. This makes the State Machine PROM a very flexible part that can be used in the following applications.

State Machine Designs

As discussed above, the State Machine PROM is ideal for state machine applications. The State Machine PROM features that are useful for state machine design are

- Output registers, used to hold state information. Each register can be individually configured to be either clocked or bypassed. When configured as clocked, the output on that register line is a Moore output. When the register is bypassed, the output is a Mealy output.
- Internal feedback of state information, eliminating the external feedback path, which enables the device to run at 100 MHz.
- Input multiplexer, which selects between address inputs and feedback state bits.
- Programmable initialize word, to set the state machine to a known state.
- x16 array, providing wider word for flexible state machine implementation.

Output-Registered PROM

For applications that need data clocked out of the PROM, the input registers are configured to be bypassed, while all output registers are configured to capture data at the rising edge of clock. The address multiplexer is configured to take inputs only from the address pins. With this configuration, the device is very similar to an output-registered PROM like the CY7C245A, but with a wider 16-bit data path.

High-Speed Asynchronous PROM

When both the input and output registers of the PROM are configured to be bypassed and the mux is configured to take input from the address pins, the device resembles a fast, asynchronous PROM. The propagation delay through the device (prime bin) is 18 ns, which is faster than any other PROM of this size. Hence, the State Machine PROM also performs well as a high-speed asynchronous PROM.

Pipelined Memory

The State Machine PROM can be used as a pipelined memory, when both input and output registers are configured to capture data at the rising edge of clock. This makes the device useful in high-performance embedded-control applications using pipelined processors.

Using the State Machine PROM as a Video Signal Generator

In this design, the State Machine PROM is used to generate the blanking and sync pulses for PAL, SECAM and NTSC video standards. To better understand this application, we will describe the operation of a CRT, beginning with some basic terms.

Scanning: The picture on a CRT is scanned in a sequential series of horizontal lines, one under another. Scanning is further categorized into horizontal and vertical scanning. In horizontal scanning, the beam is deflected from the beginning of a line to the end of a line. In vertical scanning, the beam is deflected from a position on screen to a position below it. Vertical scanning allows horizontal lines to be scanned sequentially.

Frame: One picture screen is called a frame. To include more detail in a picture, the number of horizontal lines in a frame should be high. There are 525 lines per frame in NTSC TV systems and 625 lines per frame in PAL and SECAM TV systems.

Blanking: Because each line is scanned sequentially, there is a time when the beam returns from the end of one line to the beginning of the next line. During this time, no picture information should be displayed on screen. This time is called the *retrace* time or the *blanking* time. Horizontal blanking occurs at the end of each line, while vertical blanking occurs at the end of each field (see Interlacing).

Synchronization: The electron beam in a CRT scans different picture elements to assemble the image, so timing is extremely important, and the beam has to be in the right place at the right time. Synchronization pulses are added to the video signal at the source to help align the beam when assembling the image on the CRT. Two kinds of synchronization pulses exist. Horizontal sync pulses align the beam to the beginning of each line, and vertical sync pulses align the beam to the beginning of each frame.

Without the horizontal synchronization pulses, the picture would roll horizontally and ultimately break up into diagonal components on the picture screen. Without the vertical sync pulses, the picture would roll up or down on the picture screen.

Interlacing: In an interlaced display, the picture frame is divided into groups of odd and even lines. Each group is called a *field*. During scanning, the odd lines are scanned first and even lines later. To display one picture frame, both fields (odd and even) are scanned. This process of breaking up a picture frame into two fields and scanning them sequentially is called interlacing.

The impression made by light falling on the human eye persists for 0.1 second after the source is removed. If many views are presented to the eye during this time interval of persistence of vision, the eye will integrate them and give the impression of seeing all the images at the same time. This phenomenon of persistence of vision makes it possible for CRT screens to display one basic picture element at a time. When the elements are scanned rapidly enough, they appear to the eye as a complete picture. Interlaced scanning uses this phenomenon to achieve an illusion of motion.

The frame repetition rate in NTSC television receivers is 30 per second. Because the display is interlaced, 60 views of the screen are presented to the eye during each second. The frame repetition rate of 30 per second is exactly half of the power supply frequency in most American homes. Hence, the field frequency is equal to the power supply fre-

quency. In countries where the power supply frequency is 50 Hz, the frame repetition rate is 25 frames per second and the field frequency is 50 Hz.

Figures 3a–d show the sequence of events that occur in interlaced scanning.

In *Figure 3a*, the beam starts off at the top left of the screen. The picture information for this horizontal line is assembled on screen in the time portion from A to B. Time BC is the retrace or blanking time. C is the start of the next odd line on the screen, and the same process is repeated until point D is reached. *Figure 3b* shows the vertical retrace (or blanking) time, during which the beams moves from the bottom of the picture to the beginning of the first even line on screen, located by point E. *Figure 3c* shows the same process being repeated for all the even lines on screen. *Figure 3d* shows the vertical retrace time, which brings the beam back to the start of the first odd line.

This design deals with the generation of the sync and blanking signals for a CRT for PAL, SECAM, and NTSC systems, and can be extended to generate signals for various other standards too. *Figure 4* shows a black-box representation of the system. The inputs to the system are

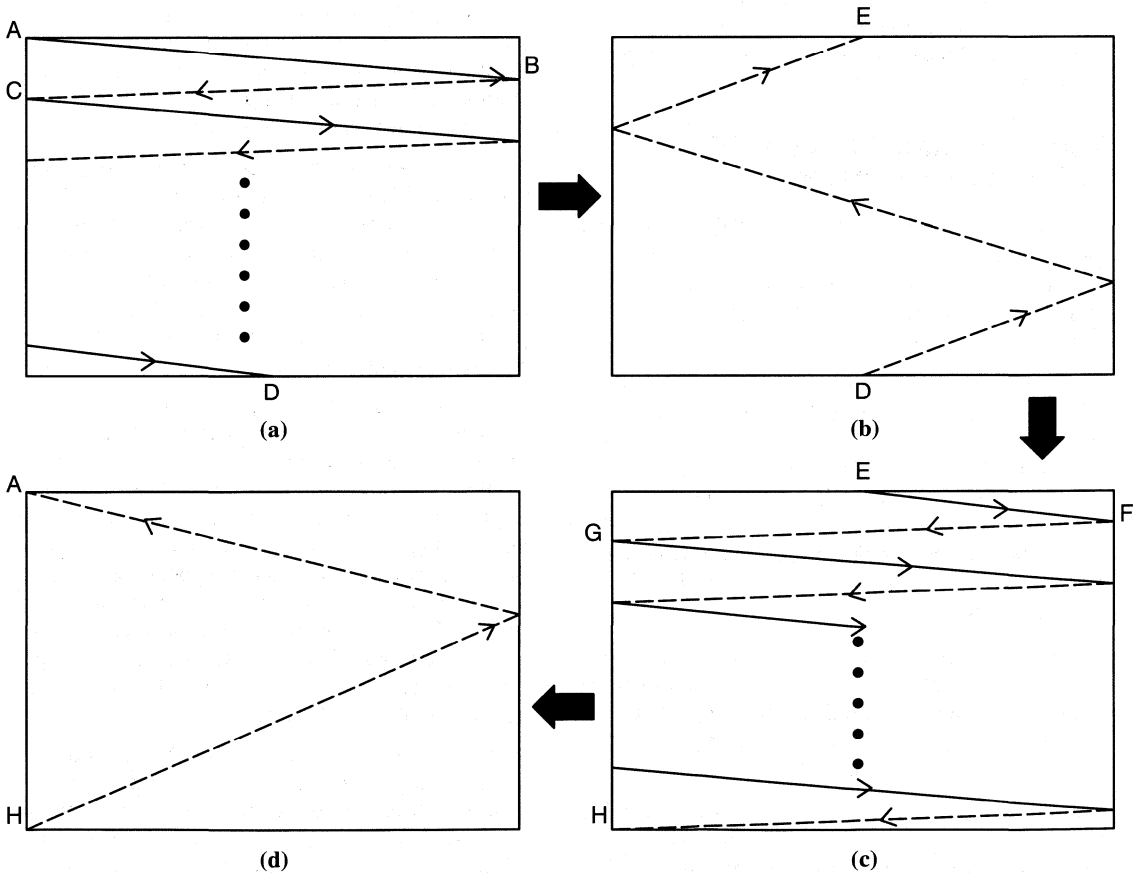


Figure 3. Interlaced Scanning Procedure

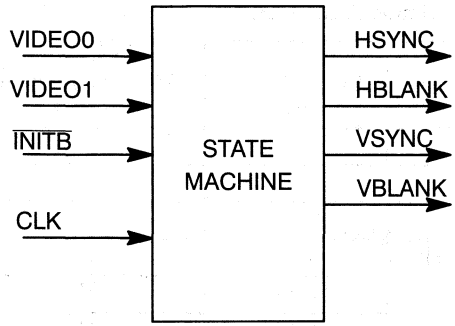


Figure 4. Black Box Representation

- Video Selector inputs Video0 and Video1. The Video Selector inputs select the system (PAL, SECAM, or NTSC) for which signals will be generated. PAL and SECAM systems have the same sync and blanking timing parameters and therefore, in this case only VIDEO0 is used.
- Clock: The clock input represents the horizontal system clock.

The outputs from the system are

- HBLANK: This signal generates the horizontal blanking required when the beam moves from the end of one line to the beginning of the next line.
- HSYNC: This signal generates the horizontal sync signal required when the beam moves back to the beginning of the next horizontal line.
- VBLANK: This signal generates the vertical blanking required when the beam moves from the bottom on one field to the top of the next field.
- VSYNC: This signal generates the vertical sync signal required when the beam moves back to the beginning of the next field.

The waveforms for all these signals along with the timing parameters are shown in *Figure 5* and *Tables 1* and *2*.

System Design

The system design was logically divided into two state machines, one generating horizontal signals and other generating vertical signals. The horizontal state machine can run off the system clock. The vertical state machine, however, needs to keep track of the number of horizontal lines scanned. In PAL and SECAM systems this number is 625, while in NTSC systems it is 525. Therefore, each field has 312.5 or 262.5 lines, depending on the standard selected. The vertical state machine counter needs to keep track of half lines, which implies that the vertical counter clock needs to run at twice the frequency of the horizontal blanking pulse. (The horizontal blanking pulse is generated for each line.)

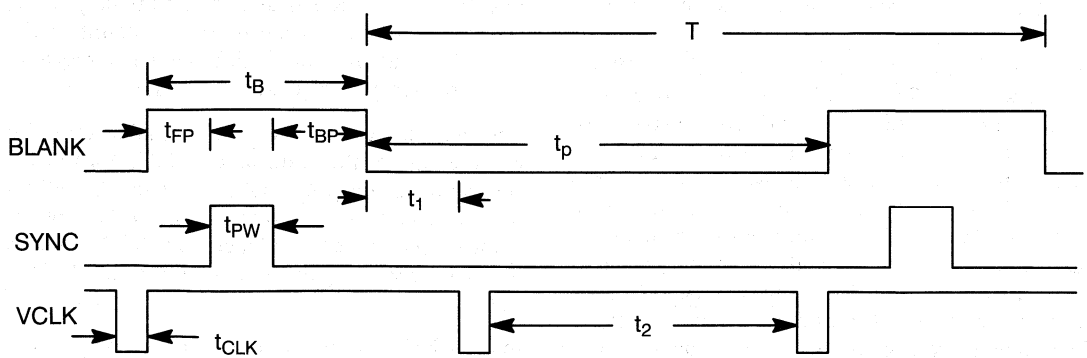


Figure 5. Blanking and Syncing Signals

Table 1. Horizontal State Machine Timing Parameters

Timing Parameter	Formula	NTSC (µs)	Count (Timing Parameter / t_{CLK})	SECAM/PAL (µs)	Count (Timing Parameter / t_{CLK})
T	H	63.5	–	64	–
t_B	0.16H	10.16	–	10.24	–
t_P	H – t_B	53.34	–	53.76	–
t_{FP}	≥ 0.02H	1.27	11	1.28	11
t_{BP}	≥ 0.06H	3.81	31	3.84	31
t_{PW}	t_B – (t_{TP} + t_{BP})	5.08	40	5.12	41
t_1	t_P – (t_2 + t_{CLK})	21.465	172	21.635	173
t_2	H/2 – t_{CLK}	31.625	253	31.875	255
t_{CLK}	–	0.125		0.125	–

Table 2. Vertical State Machine Timing Parameters

Timing Parameter	Formula	NTSC (µs)	Count (Timing Parameter / t_{CLK})	SECAM/PAL (µs)	Count (Timing Parameter / t_{CLK})
T	1/Fsupp	16,667	–	20,000	–
t_B	0.08T	1333	–	1,600	–
t_P	T – t_B	15334	483	18,400	575
t_{FP}	3H	190.5	6	192	6
t_{BP}	t_B – (t_{FP} + t_{PW})	952	30	1216	6
t_{PW}	3H	190.5	6	192	38
t_{CLK}	–	31.75	–	32	–

In this case, the best way of generating the vertical counter clock is from the horizontal state machine.

In this design, it was decided to use external 8-bit counters. The 74HC40103 was selected as the counter for the system. In addition, each state machine was implemented on a separate CY7C259. Hence, the system consists of two CY7C259s and two 74HC40103s. The schematic of the system is shown in *Figure 6*. Two additional signals need explanation. The $\overline{LD}$ signal loads the count presented on the data outputs into the counter. The $\overline{TC}$ signal from the counter indicates that the count is over. The state diagrams for these state machines are shown in *Figures 7* and *8*, respectively.

From the waveforms, it can be seen that the time period of the VCLK signal is approximately 32 µs which corresponds to the maximum count in the horizontal state machine (255). Hence, the system clock frequency is selected to be 8 MHz.

The user has to use the programmable initialize word feature of the State Machine PROM to start generating the output signals. Asserting the INITB pin will force a user-programmable word to appear at the output of the State Machine PROM, which is then fed back and puts the device into the RESET state.

In the RESET state, the state machine looks at the video selector inputs and then branches appropri-

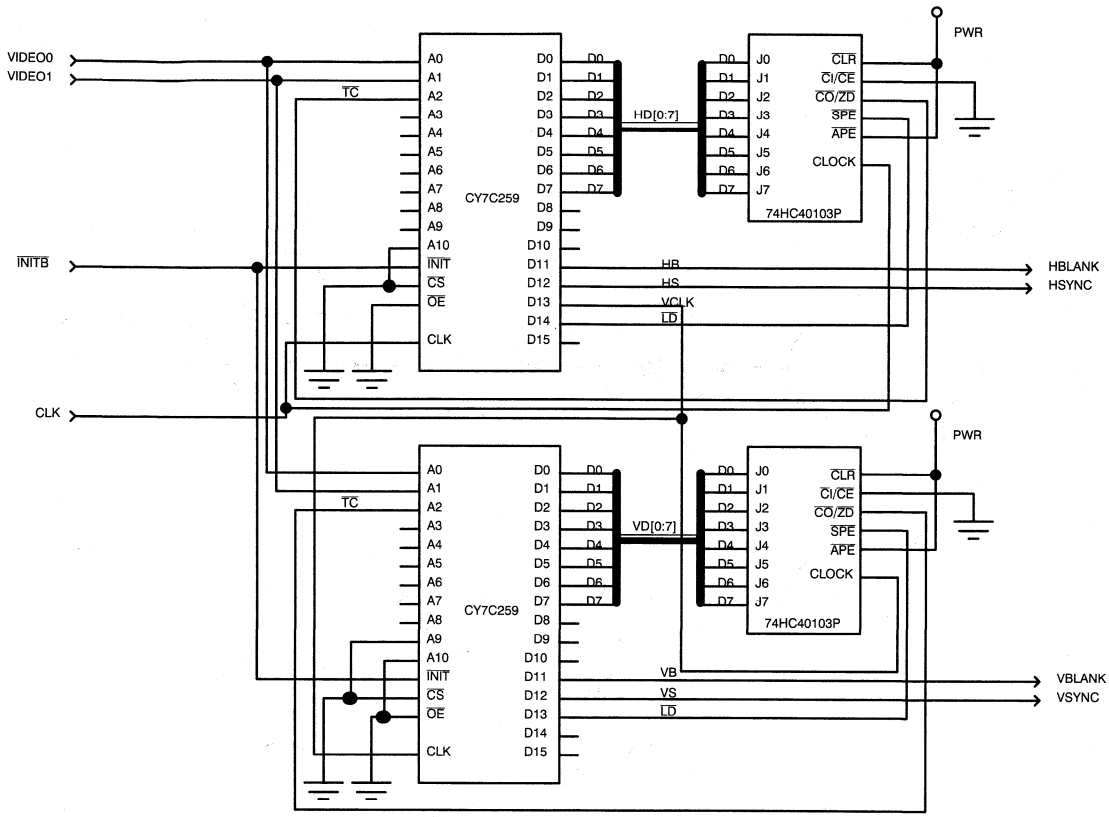


Figure 6. Schematic for Video Signal Generator

ately to generate the signals for PAL/SECAM systems or NTSC systems.

The design was compiled in LOG/iC and the PROM array contents were generated in an Intel MCS−86 format file. The configuration bits (to configure the input and output registers, the input multiplexer and the programmable initialize word register) were later appended by hand. *Tables 3* and *4* show the bit breakdown of the configuration bits.

Appendices A and B show the design files created in LOG/iC.

Reference

Grob, Bernard. *Basic Television: Principles and Servicing*. 4th Edition. McGraw−Hill, Inc.: New York, 1975.

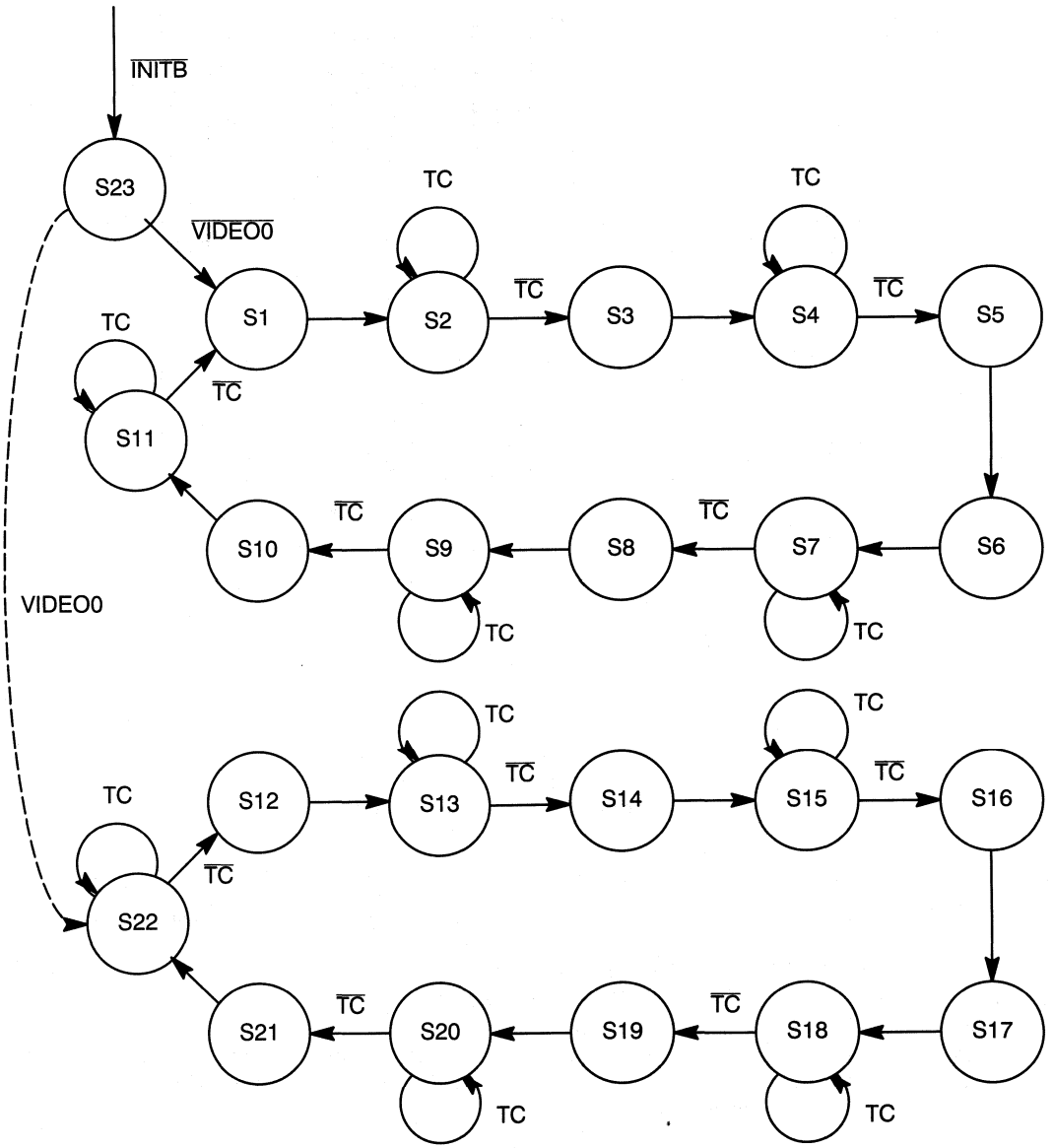


Figure 7. State Diagram of the Horizontal State Machine

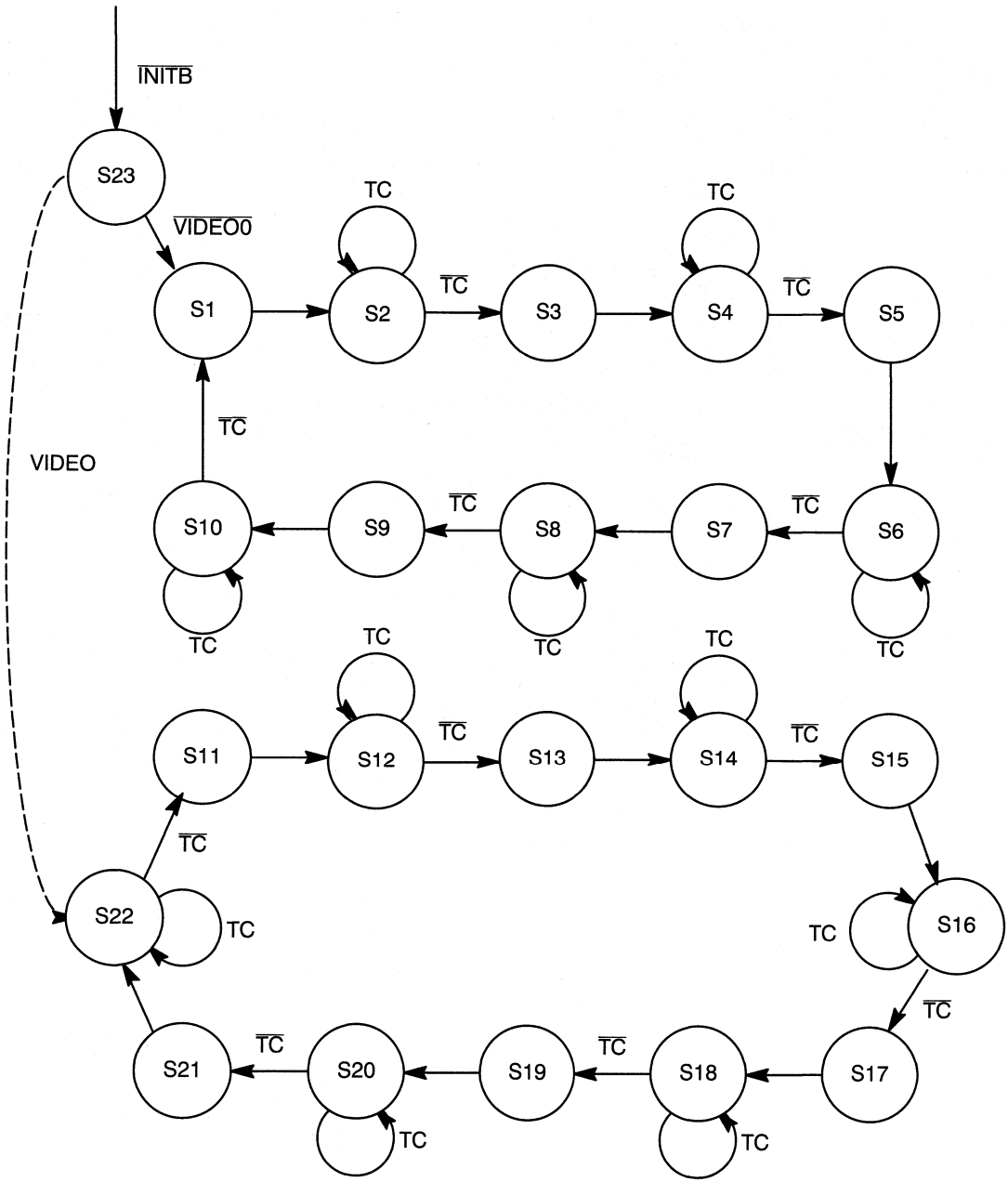


Figure 8. State Diagram of the Vertical State Machine

Table 3. Configuration Bits for Horizontal State Machine

Programmer Address Decimal	Programmer Address Hex	Programmer Memory 7C259	Bit Breakdown D_{15} D_{14} D_{13} D_{12} D_{11} D_{10} D_9 D_8 D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
2048	800	Address Register Select (1 = Bypassed Register)	A_{10} A_9 A_8 A_7 A_6 A_5 X X X X X A_4 A_3 A_2 A_1 A_0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
2049	801	Array Input Select (1 = Feedback)	A_{10} A_9 A_8 A_7 A_6 A_5 X X X X X A_4 A_3 A_2 A_1 A_0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
2050	802	Output Register Select (1 = Bypassed Register)	D_{15} D_{14} D_{13} D_{12} D_{11} D_{10} D_9 D_8 D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
2051	803	INIT WORD (1 = INIT Bit 1)	D_{15} D_{14} D_{13} D_{12} D_{11} D_{10} D_9 D_8 D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0 0 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0
2052	804	Architecture	SH C_1 C_2 CP IB IA X X X X X X X X X X 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1

Table 4. Configuration Bits for Vertical State Machine

Programmer Address Decimal	Programmer Address Hex	Programmer Memory 7C259	Bit Breakdown D_{15} D_{14} D_{13} D_{12} D_{11} D_{10} D_9 D_8 D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
2048	800	Address Register Select (1 = Bypassed Register)	A_{10} A_9 A_8 A_7 A_6 A_5 X X X X X A_4 A_3 A_2 A_1 A_0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
2049	801	Array Input Select (1 = Feedback)	A_{10} A_9 A_8 A_7 A_6 A_5 X X X X X A_4 A_3 A_2 A_1 A_0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0
2050	802	Output Register Select (1 = Bypassed Register)	D_{15} D_{14} D_{13} D_{12} D_{11} D_{10} D_9 D_8 D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
2051	803	INIT WORD (1 = INIT Bit 1)	D_{15} D_{14} D_{13} D_{12} D_{11} D_{10} D_9 D_8 D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0
2052	804	Architecture	SH C_1 C_2 CP IB IA X X X X X X X X X X 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1

Appendix A. LOG/iC™ Design File for Horizontal State Machine

```
*IDENTIFICATION
VIDEO Signal Generator using the State Machine PROM
Horizontal State Machine Implementation
Cypress Semiconductor

;*****************************************************************
;*********                                               **********
;*********      D E S I G N   D E S C R I P T I O N      **********
;*********                                               **********
;*****************************************************************

*X-NAMES
 VIDEO1, VIDEO0, TC, DUMMY ;

; (VIDEO1,VIDEO0) = (00, NTSC), (01, PAL/SECAM)
; DUMMY required to generate INTEL HEX file of correct size (2K x 16).
; If not included, file is half of correct size.

*Y-NAMES
 D7, D6, D5, D4, D3, D2, D1, D0 ;

*Z-NAMES
 Q2, Q1, Q0, HB, HS, VCLK, LD ;

*Z-VALUES

; Sn = Q2, Q1, Q0, HB, HS, VCLK, LD ;
; Q2 is to indicate the RESET state.  Q1 indicates the video type for which the vid-
eo
; is generated.  Q0 is to distinguish between different states.

S23       = 1100011  RESET state, waiting for correct assertion of VIDEO0 signal

; S[1-11] generate waveforms for NTSC signals

S1        = 0010010 ; Load Counter with Count(t1)
S2        = 0010011 ; Wait for TC
S3        = 0010000 ; Assert VCLK for 1 clock cycle, Load Counter with Count(t2)
S4        = 0000011 ; Wait for TC
S5        = 0000001 ; Assert VCLK for 1 clock cycle
S6        = 0000010 ; Load Counter with Count(tFP)
S7        = 0001011 ; Wait for TC
S8        = 0001110 ; Load Counter with Count(tPW)
S9        = 0001111 ; Wait for TC
S10       = 0001010 ; Load Counter with Count(tBP)
S11       = 0011011 ; Wait for TC

; S[12-22] perform the same functions as S[1-11] but generate waveforms for PAL/
; SECAM signals

S12       = 0110010 ;
S13       = 0110011 ;
```

Appendix A. LOG/iC Design File for Horizontal State Machine (continued)

```
S14      = 0110000 ;
s15      = 0100011 ;
S16      = 0100001 ;
S17      = 0100010 ;
S18      = 0101011 ;
S19      = 0101110 ;
S20      = 0101111 ;
S21      = 0101010 ;
S22      = 0111011 ;
 *FLOW-TABLE

RELEVANT = DUMMY, VIDEO1, VIDEO0 ;

S23,     X       000,      Y       10101011,       F1 ;
S23,     X       001,      Y       10101100,       F12 ;

RELEVANT = TC ;

; NTSC system horizontal state machine flow table

S1,      X       -,        Y       10101011,       F2 ;

S2,      X       1,        Y       10101011,       F2 ;
S2,      X       0,        Y       11111100,       F3 ;

S3,      X       -,        Y       11111100,       F4 ;

S4,      X       1,        Y       11111100,       F4 ;
S4,      X       0,        Y       11111100,       F5 ;

S5,      X       -,        Y       00001010,       F6 ;

S6,      X       -,        Y       00001010,       F7 ;

S7,      X       1,        Y       00001010,       F7 ;
S7,      X       0,        Y       00001010,       F8 ;

S8,      X       -,        Y       00100111,       F9 ;

S9,      X       1,        Y       00100111,       F9 ;
S9,      X       0,        Y       00100111,       F10 ;

S10,     X       -,        Y       00011110,       F11 ;

S11,     X       1,        Y       00011110,       F11 ;
S11,     X       0,        Y       00011110,       F1 ;

; PAL and SECAM system horizontal state machine flow table

S12,     X       -,        Y       10101100,       F13 ;
```

Appendix A. LOG/iC Design File for Horizontal State Machine (continued)

```
S13,     X       1,      Y       10101100,       F13 ;
S13,     X       0,      Y       11111110,       F14 ;

S14,     X       -,      Y       11111110,       F15 ;

S15,     X       1,      Y       11111110,       F15 ;
S15,     X       0,      Y       11111110,       F16 ;

S16,     X       -,      Y       00001010,       F17 ;

S17,     X       -,      Y       00001010,       F18 ;

S18,     X       1,      Y       00001010,       F18 ;
S18,     X       0,      Y       00001010,       F19 ;

S19,     X       -,      Y       00101000,       F20 ;

S20,     X       1,      Y       00101000,       F20 ;
S20,     X       0,      Y       00101000,       F21 ;

S21,     X       -,      Y       00011110,       F22 ;

S22,     X       1,      Y       00011110,       F22 ;
S22,     X       0,      Y       00011110,       F12 ;

*STATE-ASSIGNMENT
 Z-VALUES ;

;*********************************************************************
;***********                                              ************
;***********   D E V I C E  C O N F I G U R A T I O N      ************
;***********                                              ************
;*********************************************************************

*ROM
 TYPE = 2K_BY_16 ;
 INPUTS = 11 ;
 OUTPUTS = 16 ;
 X-PINS = VIDEO0, VIDEO1, TC, Q0, Q1, Q2, HB, HS, VCLK, LD, DUMMY ;
 Y-PINS = D0, D1, D2, D3, D4, D5, D6, D7, Q0, Q1, Q2, HB, HS, VCLK, LD ;

*RUN-CONTROL
 PROG = INTEL-HEX ;
 LIST = EQUATIONS,PINOUT, HEX ;

*END
```

Appendix B. LOG/iC Design File for Vertical State Machine

```
*IDENTIFICATION
 VIDEO Controller using the State Machine PROM
 Vertical State Machine Implementation
 Cypress Semiconductor

;*****************************************************************
;*********                                               **********
;*********      D E S I G N   D E S C R I P T I O N      **********
;*********                                               **********
;*****************************************************************

*X-NAMES
 VIDEO1, VIDEO0, TC, DUMMY0, DUMMY1 ;

; (VIDEO1,VIDEO0) = (00, NTSC), (01, PAL), (11, SECAM)
; DUMMY0, DUMMY1 required to generate INTEL HEX file of correct size (2K x 16).
; If not connected, file size is one quarter of correct size.

*Y-NAMES
 D7, D6, D5, D4, D3, D2, D1, D0 ;

*Z-NAMES
 Q2, Q1, Q0, VB, VS, LD ;

; Sn = Q2, Q1, Q0, VB, VS, LD ;
; Q2, Q1, Q0 are the additional state bits.
; Q2 indicates the video type for which the signals are generated.  A value of 0
; indicates that NTSC signal waveforms are generated.

*Z-VALUES
S23     =       011001 ; RESET State, waiting for correct assertion of VIDEO0 signal

;S[1-10] generate waveforms for NTSC signals

S1      =       000000 ; Load Counter with Count = 256
S2      =       000001 ; Wait for Terminal Count
S3      =       001000 ; Load Counter with (Count(tP) - 256)
S4      =       001001 ; Wait for Terminal Count
S5      =       010000 ; Load Counter with Count(tFP)
S6      =       010101 ; Wait for Terminal Count
S7      =       010110 ; Load Counter with Count(tPW)
S8      =       010111 ; Wait for Terminal Count
S9      =       000100 ; Load Counter with Count(tBP)
S10     =       000101 ; Wait for Terminal Count

; S[11-22] generate waveforms for PAL/SECAM signals

S11     =       100000 ; Load Counter with Count = 256.
S12     =       100001 ; Wait for TC
S13     =       101000 ; Load Counter with Count = 256.
S14     =       101001 ; Wait for TC
```

Appendix B. LOG/iC Design File for Vertical State Machine (continued)

```
S15     =       110000 ; Load Counter with (Count(tP) - 2*256)
S16     =       110001 ; Wait for TC
S17     =       110100 ; Load Counter with Count(tFP)
S18     =       110101 ; Wait for TC
S19     =       110110 ; Load Counter with Count(tPW)
S20     =       110111 ; Wait for TC
S21     =       100100 ; Load Counter with Count(tBP)
S22     =       100101 ; Wait for TC

*FLOW-TABLE

RELEVANT = DUMMY1, DUMMY0, VIDEO1, VIDEO0 ;

S23,    X       0000,     Y       11111111,       F1 ;
S23,    X       0001,     Y       11111111,       F11 ;

RELEVANT = TC ;

; NTSC system vertical state machine flow table

S1,     X       -,      Y       11111111,       F2 ;

S2,     X       1,      Y       11111111,       F2 ;
S2,     X       0,      Y       11100010,       F3 ;

S3,     X       -,      Y       11100010,       F4 ;

S4,     X       1,      Y       11100010,       F4 ;
S4,     X       0,      Y       00000101,       F5 ;

S5,     X       -,      Y       00000101,       F6 ;

S6,     X       1,      Y       00000101,       F6 ;
S6,     X       0,      Y       00000101,       F7 ;


S7,     X       -,      Y       00000101,       F8 ;

S8,     X       1,      Y       00000101,       F8 ;
S8,     X       0,      Y       00011101,       F9;

S9,     X       -,      Y       00011101,       F10 ;

S10,    X       1,      Y       00011101,       F10 ;
S10,    X       0,      Y       11111111,       F1 ;


; PAL and SECAM system vertical state machine flow table

S11,    X       -,      Y       11111111,       F12 ;
```

Appendix B. LOG/iC Design File for Vertical State Machine (continued)

```
S12,    X       1,      Y       11111111,       F12 ;
S12,    X       0,      Y       11111111,       F13 ;

S13,    X       -,      Y       11111111,       F14 ;

S14,    X       1,      Y       11111111,       F14 ;
S14,    X       0,      Y       00111110,       F15 ;

S15,    X       -,      Y       00111110,       F16 ;

S16,    X       1,      Y       00111110,       F16 ;
S16,    X       0,      Y       00000101,       F17 ;

S17,    X       -,      Y       00000101,       F18 ;

S18,    X       1,      Y       00000101,       F18 ;
S18,    X       0,      Y       00000101,       F19 ;

S19,    X       -,      Y       00000101,       F20 ;

S20,    X       1,      Y       00000101,       F20 ;
S20,    X       0,      Y       00100101,       F21 ;

S21,    X       -,      Y       00100101,       F22 ;

S22,    X       1,      Y       00100101,       F22 ;
S22,    X       0,      Y       11111111,       F11 ;


*STATE-ASSIGNMENT
 Z-VALUES ;

;*********************************************************************
;***********                                             *************
;***********   D E V I C E  C O N F I G U R A T I O N     *************
;***********                                             *************
;*********************************************************************

*ROM
 TYPE = 2K_BY_16 ;
 INPUTS = 11 ;
 OUTPUTS = 16 ;
 X-PINS = VIDEO0, VIDEO1, TC, Q0, Q1, Q2, VB, VS, LD, DUMMY0, DUMMY1 ;
 Y-PINS = D0, D1, D2, D3, D4, D5, D6, D7, Q0, Q1, Q2, VB, VS, LD ;

*RUN-CONTROL
 PROG = INTEL-HEX ;
 LIST = PINOUT, HEX ;

*END
```

Document #: PRO0002–0892

LOG/iC is a trademark of Isdata.

Generating PROM Programming Files

PROMs are nonvolatile memory devices that were first conceived as instruction and data storage devices for microprocessor systems. Since their introduction, PROMs have benefited from improvements in processing and manufacturing technology. The evolution of PROMs has included a tremendous increase in their density and speed and has added new features such as built-in registers and reprogrammability. Now these devices can be used in a wide variety of applications other than instruction storage. PROMs are commonly found in state machines, decoders, encoders, complex counters, controllers, sequencers, and look-up tables as well as in their traditional role of instruction or microcode storage.

PROMs are simply an array of data coupled with an input address decoder. The address presented to the device drives a simple 1-of-n decoder. The decoder selects one preprogrammed memory location whose data flows to the output pins of the device. PLAs (Programmable Logic Array) and PALs (Programmable Array Logic) are also programmable devices and, along with PROMs, make up the majority of devices that are considered to be programmable logic elements. The difference between the three types of programmable logic elements can be seen by observing the internal structure of the programmable array of each of the devices. PLAs have both a programmable "AND" array and a programmable "OR" array. PALs have a similar AND-OR structure, but the number of inputs to the OR function is fixed, so only the AND array is programmable. Both the PLA and PAL have a fixed number of AND-OR terms dedicated to each output. Therefore, the number of functions controlling each output is significantly reduced. PROMs, on the other hand, can realize every possible combination or function of n input lines for a given output. There are 2^n product terms (where n = number of address lines) per PROM output. This makes PROMs useful in very complex functions that exhaust the sum-of-product resources of a traditional PAL or PLA architecture. Some PROMs have additional features, such as output registers, that enable them to operate synchronously, which is required for state machines. The Cypress CY7C245A is one of these PROMs. Presets, clears, and initialization words are also available for dealing with power-on and reset conditions.

After understanding the basic function of a PROM, the designer must now create the PROM data in the form of a programming file. Creating the PROM data can be intimidating to engineers who are not familiar with the process. Looking back, we can see that PROMs were mainly used for instruction or microcode storage in a microprocessor or bit-slice-based system. Therefore, the PROM data for such systems is generated by the compilers, assemblers, and linkers that are resident on the CPU development station or emulator. Generating the PROM files for such systems is almost trivial because the programming data file is simply a listing of the CPU's executable instructions generated by the compiler. But creating the programming file for a complex decoder, look-up table, sequencer, or state machine can be pretty complicated and overwhelming. In fact, just figuring out where to start or what tools to use can become very time consuming. In this brief application note we will discuss the structure of PROM data files and show several ways to create them. Examples using simple languages such

as C and BASIC, as well as PLD development tools such as ABEL and LOG/iC, will be discussed.

In order to understand how to create programming files, you must first be familiar with the actual structure or format of such a file. Again, a PROM is simply an array of programmable memory locations. The data file that is transmitted to the PROM programmer must therefore contain data for each of the locations to be programmed. There are many standard formats for PROM data files.

Generic PROM programmers, such as those manufactured by Data I/O, Stag, Logical Devices, Digelic, SMS, and Kontron, are generally compatible with the following formats:

- ASCII—HEX (Space)
- Binary
- DEC Binary
- Motorola Exorciser
- Motorola Exormax
- Intel "Intellec" 8/MDS
- Intel MCS86 "intellec 86"
- Tektronix "HEX"
- Extended Tektronix "HEX"

The following section describes each format in detail. Each format has its own set of required fields, delimiters, and special characters. When writing code in C or BASIC, you must know exactly where to place each field and special character so that a programmer will interpret your data correctly.

ASCII–HEX (Space)

One of the simplest and probably the most universal file formats is HEX or HEX-Space ASCII. This format does not support checksum or address field conventions. Therefore, the data in the file must be in order incrementing from address 0. However, many times the program that reads the file into programmer memory can manipulate the data to start at any address location.

Three hidden instructions are used in this format:

1. ASCII STX Character (ASCII 02) marks the beginning of the file.
2. ASCII ETX Character (ASCII 03) marks the end of the file.
3. ASCII Space (ASCII 20) is between each data byte.

Figure 1 shows a data file for a 64-byte PROM implemented in ASCII–HEX (space) format.

Note that each data byte is separated by a "space" character and that no addressing information is present.

ASCII Binary

ASCII Binary files, like ASCII–HEX, contain no addressing or checksum information. ASCII Binary allows for very fast file transfers to the programmer due to its simplicity. The data format begins with the ASCII STX character and is terminated by an ETX. Data is grouped into four-byte lines separated by a space. Each line of data begins with a "B" character and ends with an "F" character.

Figure 2 shows a 64-byte PROM file containing all zeros using ASCII Binary format. All data is loaded into the PROM sequentially starting at location 0.

Simple Binary

The simple Binary format consists of just binary data. There are no start or end characters. Although the binary file is simple to produce, it is not a recommended output format for the following examples because binary files cannot be easily read by text editors.

```
(STX)FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF(ETX)
```

Figure 1. ASCII–HEX Format

DEC Binary

DEC Binary is a modification of the basic ASCII Binary file format. DEC Binary adds a starting address and a checksum for each line of data.

Motorola Exorcisor

Motorola Exorcisor is one of the most widely used formats. Motorola Exorcisor files are commonly referred to as "S" records because each line starts with an "S" followed by the record type. Each line also contains a byte count, starting address, and a checksum, which are delineated by carriage returns and line feeds.

Figure 3 shows an example of a 64-byte PROM file implementing "S" Records.

Calculating Record Checksum

The Checksum is calculated by first stripping off the start code ("S"), the record type, and the checksum. The remaining bytes are added together, converted to binary, and complimented (one's compliment). For example, the optional sign on "S0" line reads:

S0 06 00 01 00 01 F7

Stripping the appropriate characters leaves:

06 00 01 00 01

Adding the bytes yields

```
(STX)  B00000000F B00000000F B00000000F B00000000F
       B00000000F B00000000F B00000000F B00000000F
       B00000000F B00000000F B00000000F B00000000F
       B00000000F B00000000F B00000000F B00000000F(ETX)
```

Figure 2. ASCII Binary Format

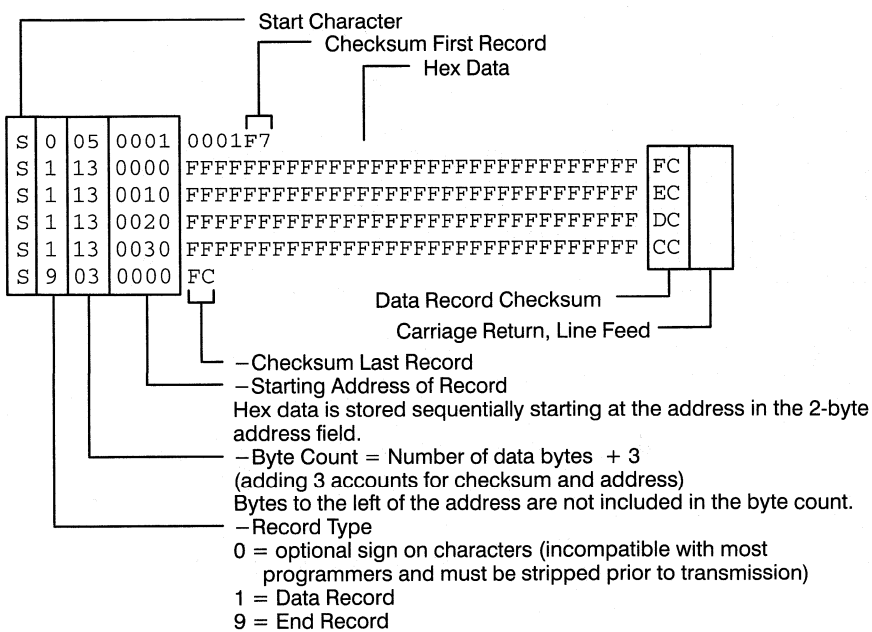


Figure 3. S Record Format

08 hex

The compliment of the value

F7..... Record checksum

End of Each Record

It is important to end each record with a carriage return and a line feed, which is used as a delineator.

"S" records are useful because they are so universal. However, this format can only be used for PROMs smaller than 64 Kbytes because the address field is limited to 4 bytes.

Motorola Exormax

Exormax is another "S" record file and is identical to Exorcisor with only one exception. Exormax allows for a 6-digit address field, which makes it useful for PROMs that are much larger than 64 Kbytes.

Exormax Record Number:

S0– Optional sign on record
S1– Data record (2 Byte Address field)
S2– Data Record (3 Byte Address Field)

Figure 4 shows an example of a 64-byte PROM file implementing "Exormax S" records.

Intel "Intellec" 8/MDS

Intellec is similar to S records in that each line contains a starting address, byte count, and checksum. However, each line begins with a colon.

Intellec Record Example:
":", Byte Count, Address, Record Type, Data, Checksum

Byte Count: Total number of data bytes ONLY.

Starting Address: 2-byte field where record will be placed in memory.

Record Type:
00 — Data Record
01 — End Record

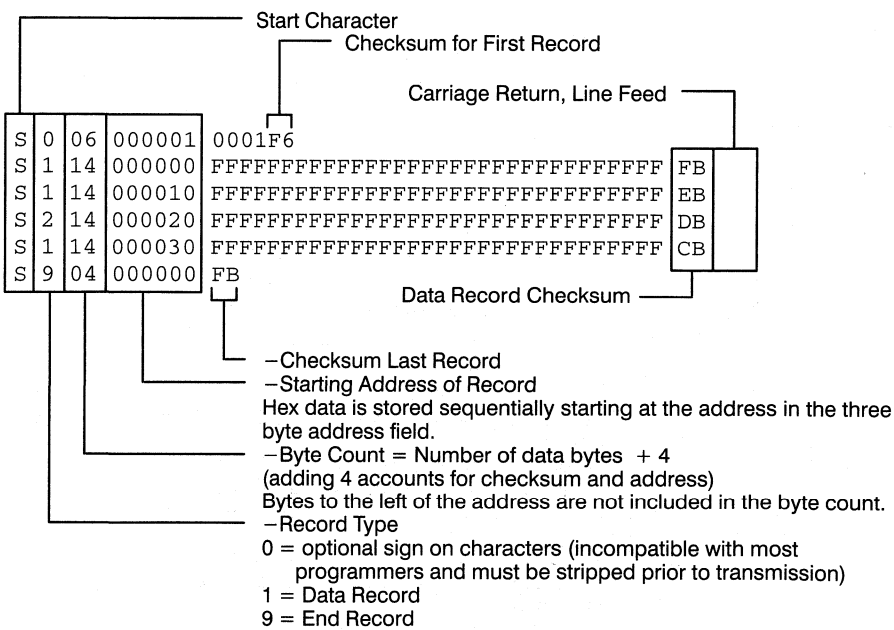


Figure 4. Exormax S Format

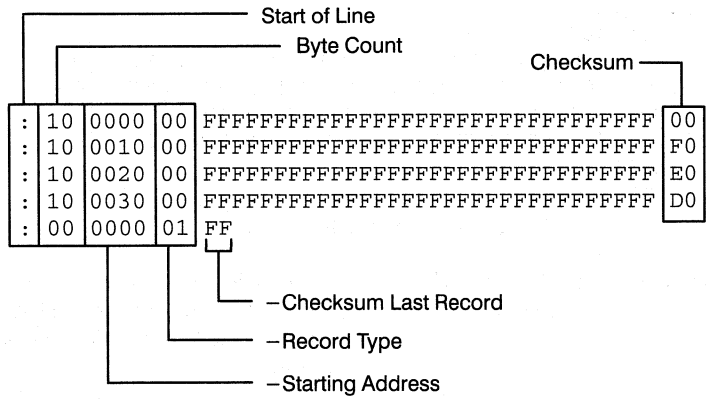


Figure 5. Intellec Format

Checksum: The sum of all preceding bytes including byte count, Address, and all data bytes. This number is expressed in two's compliment notation.

The end of each record is marked by a carriage return.

Figure 5 shows a 64-byte PROM file using Intellec format.

Since there is only a 2-byte address field, Intellec is generally used for PROMs smaller than 64 Kbytes.

Intel MCS86 (Intellec 86)

Intellec 86 is an extension of the standard Intellec format. It adds the feature of a Segment Base Address record (SBA). Adding the SBA to the 2-byte address field increases the total addressing capability to 1M locations. The file must begin with an SBA record because physical addresses are calculated using the Starting Address field and the most recent SBA.

Intellec 86 Data or End Record Example:
":", Byte Count, Address, Record Type, Data, Checksum

Intellec 86 SBA Record Example:
":", Byte Count, Address, Record Type "02", SBA, Checksum

Byte Count: Total number of data bytes ONLY.

Segment Base Address (SBA): A 2-byte field that extends the starting address fields of the following records by 4 bits. A new SBA can be inserted as many times as needed. Records sent after a new SBA will use the new SBA to calculate the address.

Starting Address: A 2-byte field where record will be placed in memory. The actual physical address for data placement must be calculated by using the SBA and Starting Address.

Record Type:
00 — Data Record
01 — End Record
(02 — SBA Record)

Checksum: The sum of all preceding bytes including byte count, address, and all data bytes. This number is expressed in two's compliment notation.

The end of each record is marked by a carriage return and line feed.

Figure 6 shows a 64-byte PROM file using Intellec 86 format. This example has an SBA Value of 8000h, which offsets the starting addresses as shown.

To calculate the starting address: (third data record)

Take the value of the most recent SBA	(8000h)
Shift the SBA left	8000
Add value of start address field	+ 0030
Result: Physical Start Address	80030

```
:02 0000 02 8000 7C
:10 0000 00 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF 00
:10 0010 00 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF F0
:10 0020 00 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF E0
:10 0030 00 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF D0
:00 0000 01 FF
```

Figure 6. Intellec 86 Format

Again, the segment base address can be updated at any time and will affect the records that follow the change.

Tektronix Hex (TEK HEX)

TEK HEX is another simple file format that is accepted by most programming systems. It uses the "/" character as a start-of-record marker and includes a starting address for each record, byte count, data and two checksums. The first checksum is the summation of the bytes for the address and byte count fields. The second checksum is simply the summation of all of the data bytes. *Figure 7* shows an example of a file stored in TEK HEX format.

Start Character: "/" is used to mark the beginning of each line. Most programmers ignore any characters sent before the "/".

Start Address: This value is a 2-byte absolute address. It represents the starting address for the first data byte in the record. All following bytes in the record are stored sequentially.

Byte Count: The number of data bytes in the record are represented by the byte-count field. The end of record is marked by setting the byte count equal to "00".

First Checksum: The simple summation of the nibbles in the address and byte-count fields are represented by the first checksum in each record.

Second Checksum: Calculated by summing all of the nibbles of the data bytes in the record, the second checksum is placed at the end of the record.

Each record is terminated by a carriage return/line feed.

Extended Tektronix Hex (XTEK)

XTEK is a variation of the standard TEK HEX format. It uses the "/" character as a start of record marker and includes a starting address for each record, byte count, data, and two checksums. The first checksum is the summation of the nibbles for the address and byte-count fields. The second checksum is simply the summation of all of the data nibbles. *Figure 8* shows an example of a file stored in XTEK format.

Start Character: "/" is used to mark the beginning of each line. Most programmers ignore any characters sent before the "/".

Start Address: This value is a 2-byte absolute address. It represents the starting address for the first data byte in the record. All following bytes in the record are stored sequentially.

```
/0000 10 01 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF E0
/0010 10 02 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF E0
/0020 10 03 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF E0
/0030 10 04 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF E0
/0000 00 00
```

Figure 7. TEK HEX Format

Byte Count: The number of data bytes in the record are represented by the byte-count field. The end of record is marked by setting the byte count equal to "00".

First Checksum: The simple summation of the nibbles in the address and byte count fields are represented by first checksum in each record.

Second Checksum: The summation of the data nibbles is represented by the second checksum, which is placed at the end of the record.

Each record is terminated by a carriage return/line feed.

Using High-Level Languages to Create Files

Depending on the application, there are many different ways to create the actual file. PROMs that contain data derived from mathematical formulas such as look-up tables are easily implemented using a high-level language such as C or BASIC. These languages can easily deal with the complicated data types and mathematical data manipulation that is required for many applications. The data created by the program must be stored in a file so that it can be transferred to a programmer at some later time. The following examples show that opening a new file and writing the data is simple when using high-level languages.

As shown in *Figure 9*, this method written in C follows the simple form:

1. Header documentation – The Header documentation is usually written as comments to help the user understand the purpose and flow of the program. Documentation is not essential, but it is good practice.
2. Constant declaration – Following the Header, the constants can be declared as symbols to help the user update the program to accommodate changes in the design.
3. Variable definition – The variables should be defined to agree with the type of data being used.
4. Body – The body of the program will contain the commands necessary to create the PROM data. This usually takes the form of an outer "For"-type loop to iteratively step through all the possible combinations of address inputs, followed by nested commands that create a data instance to correspond to that combination of address lines.

The C program in *Figure 9* generates ASCII-Space or HEX-Space format output files for downloading to a PROM programmer.

Figure 10 is an example of using BASIC to produce a PROM programming file in the HEX space format.

PLD Development Packages

In general, most of the standard PLD development packages support PROMs. ABEL, CUPL, and LOG/iC are three of the most popular third-party packages. They support most of the industry standard PAL, PLA and PROM devices. These PLD development tools are well suited to creating PROM files that can be described by Boolean equations, truth tables, or state-machine syntax.

ABEL

ABEL, produced by Data I/O Corporation, is one of the most popular PLD development software packages on the market. The fact that ABEL supports PROMs is one of the industry's best-kept secrets. Since a PROM can be thought of as a PLD with a large number of product terms per output, it is relatively easy for a PLD compiler to generate code for a PROM. In fact, the source file (filename.abl) for

```
% 1A 6 06 4 1000 FFFFFFFFFFFFFFFF
% 1A 6 0E 4 1008 FFFFFFFFFFFFFFFF
% 1A 6 07 4 1010 FFFFFFFFFFFFFFFF
 % 0A 8 16 4 0000
```

Figure 8. XTEK Format

```
/* Example Program 1 */

/* The purpose of this program is to create a data file that could be
used as a COSINE look-up table.  The table has an angular resolution of
256 points per period and an amplitude resolution 256 steps or 8 bits.
*/


#include <stdio.h>        /* defines the input-output of PC */
#include <math.h>         /* defines the math package of PC */

int i,j;                  /* integers for loop variables */
float y,x,z;              /* floating pt variables for COSINE */
int data;                 /* data  variables for result */
int outfile;

main()
/* main denotes the start of the active part of the program */
{
  FILE *outfile;
/* makes outfile a pointer to the output file */

  outfile=fopen("promfile","w");
/* opens the output file for writing */

  fprintf(outfile,"%c",2);

/* prints control data to output file for download STX */

/* This section consists of 2 nested loops to generate every possible
combination of address inputs.  An incrementing variable z is used to
generate the angle y in radians.  x = the cosine of y.  Then x is justi-
fied to use the dynamic range of 256 states.  The result is stored as an
integer in data.  The data is written directly to the output file.  The
data is broken into blocks for easier reading.  */
  z=0;
  for(i=0;i<=15;i++)  {
    for(j=0;j<=15;j++)  {
      y=M_PI*((z)/128.0);
      x=(cos(y));
      x=x*127.99999;
      data = x+128;
      fprintf(outfile,"%02X ",data);
      z=z+1.0;
    }
    fprintf(outfile,"\n");
  }
  fprintf(outfile,"%c",3);
/* prints control char ETX to output file */

  fclose(outfile);
/* closes output file */

}
```

Figure 9. C Program to Generate ASCII-Space or HEX-Space Format Files

```
10 'Example program 2
20 '
30 'The purpose of this program is to create a data file
40 'that could be used as a COSINE look-up table.  The table
50 'has an angular resolution of 256 points per period and
60 'an amplitude resolution of 256 steps or 8 bits.
70 '
80 PI = 3.14159
90 OPEN "O",#1,"PROMFILE.HEX"     'open the file for output
100 '
110 'This section consists of 2 nested loops to generate every
120 'possible combination of address inputs.  An incrementing
130 'variable z is used to generate the angle y in radians.
140 'X = cosine of y.  Then X is justified to use the dynamic
150 'range of 256 states.  The result is stored as an integer
160 'in RANGE.  The data is written directly to the output file
170 'in the HEX SPACE format.
180 '
190 PRINT#1,CHR$(2)      'start the file with the STX char
200 Z = 0                'initialize the loop
210 FOR I = 0 TO 15
220     FOR J = 0 TO 15
230         Y = PI*((Z)/128)
240         X = COS(Y)
250         RANGE = INT(X*127.99999# + 128)
260         IF RANGE > 15 THEN 290
270         PRINT#1,"0";HEX$(RANGE);" ";
280         GOTO 300
290         PRINT#1,HEX$(RANGE);" ";
300         Z = Z + 1
310     NEXT J
320     PRINT#1,""
330 NEXT I
340 PRINT#1,CHR$(3);     'end the file with the ETX char
350 CLOSE
360 END
```

Figure 10. BASIC Program to Generate HEX-Space Format

a PROM and a PLD are almost identical. The only difference is in the device declaration. In the logic diagram package for ABEL, there are pin descriptions for 4-, 8-, and 16-bit PROMS.

Figure 11 shows how to use truth tables and equations to generate a PROM file that is a comparator with some additional built-in logic.

All methods of generating PLD files in ABEL are also available for generating PROM files.

```
module COMP_OR
title '4 bit comparator '
"

     PROMB device     'RA8P8';
"                        8 address lines and 8 data lines

"INPUTS           PIN #    PROM ADDRESS/DATA BIT

     A0          PIN  1;   " A0
     A1          PIN  2;   " A1
     A2          PIN  3;   " A2
     A3          PIN  4;   " A3
     B0          PIN  5;   " A4
     B1          PIN 17;   " A5
     B2          PIN 18;   " A6
     B3          PIN 19;   " A7

"OUTPUTS          PIN #

     AGB         PIN 14;   " D8     A IS GREATER THAN B
     ALB         PIN 13;   " D7     A IS LESS THAN B
     EQUAL       PIN 12;   " D6     A IS EQUAL TO B

     ALL_HIGH    PIN 11;   " D5     ALL BITS ARE HIGH
     OR_BITS_3   PIN  9;   " D4     Misc.  logical functions
     OR_BITS_2   PIN  8;   " D3
     OR_BITS_1   PIN  7;   " D2
     OR_BITS_0   PIN  6;   " D1

     X = .X.;

Declarations
     A_NIB  = [A3,A2,A1,A0];
     B_NIB  = [B3,B2,B1,B0];

Equations
  ALL_HIGH  = (A_NIB==15) & (B_NIB==15);
  OR_BITS_3 = A3 # B3;
  OR_BITS_2 = A2 # B2;
  OR_BITS_1 = A1 # B1;
  OR_BITS_0 = A0 # B0;
```

Figure 11. Using Truth Tables and Equations in ABEL to Generate a Comparator PROM File

```
truth_table
     ([A3,A2,A1,A0, B3,B2,B1,B0]->[AGB,ALB,EQUAL])
      [ 0,  0,  0,  0,   0,  0,  0,  0]->[ 0 ,  0 ,  1];"A = B CONDITIONS
      [ 0,  0,  0,  1,   0,  0,  0,  1]->[ 0 ,  0 ,  1];
      [ 0,  0,  1,  0,   0,  0,  1,  0]->[ 0 ,  0 ,  1];
      [ 0,  0,  1,  1,   0,  0,  1,  1]->[ 0 ,  0 ,  1];
      [ 0,  1,  0,  0,   0,  1,  0,  0]->[ 0 ,  0 ,  1];
      [ 0,  1,  0,  1,   0,  1,  0,  1]->[ 0 ,  0 ,  1];
      [ 0,  1,  1,  0,   0,  1,  1,  0]->[ 0 ,  0 ,  1];
      [ 0,  1,  1,  1,   0,  1,  1,  1]->[ 0 ,  0 ,  1];
      [ 1,  0,  0,  0,   1,  0,  0,  0]->[ 0 ,  0 ,  1];
      [ 1,  0,  0,  1,   1,  0,  0,  1]->[ 0 ,  0 ,  1];
      [ 1,  0,  1,  0,   1,  0,  1,  0]->[ 0 ,  0 ,  1];
      [ 1,  0,  1,  1,   1,  0,  1,  1]->[ 0 ,  0 ,  1];
      [ 1,  1,  0,  0,   1,  1,  0,  0]->[ 0 ,  0 ,  1];
      [ 1,  1,  0,  1,   1,  1,  0,  1]->[ 0 ,  0 ,  1];
      [ 1,  1,  1,  0,   1,  1,  1,  0]->[ 0 ,  0 ,  1];
      [ 1,  1,  1,  1,   1,  1,  1,  1]->[ 0 ,  0 ,  1];

      [ 0,  0,  0,  0,   X,  X,  X,  1]->[ 0 ,  1 ,  0];"A < B CONDS.
      [ 0,  0,  0,  X,   X,  X,  1,  X]->[ 0 ,  1 ,  0];
      [ 0,  0,  X,  X,   X,  1,  X,  X]->[ 0 ,  1 ,  0];
      [ 0,  X,  X,  X,   1,  X,  X,  X]->[ 0 ,  1 ,  0];
      [ 1,  0,  X,  X,   1,  1,  X,  X]->[ 0 ,  1 ,  0];
      [ 1,  0,  0,  X,   1,  0,  1,  X]->[ 0 ,  1 ,  0];
      [ 1,  0,  0,  0,   1,  0,  0,  1]->[ 0 ,  1 ,  0];
      [ 0,  1,  0,  X,   0,  1,  1,  X]->[ 0 ,  1 ,  0];
      [ 0,  1,  0,  0,   0,  1,  X,  1]->[ 0 ,  1 ,  0];
      [ 0,  0,  1,  0,   0,  0,  1,  1]->[ 0 ,  1 ,  0];

      [ 1,  X,  X,  X,   0,  X,  X,  X]->[ 1 ,  0 ,  0];"A > B CONDS.
      [ 0,  1,  X,  X,   0,  0,  X,  X]->[ 1 ,  0 ,  0];
      [ 0,  0,  1,  X,   0,  0,  0,  X]->[ 1 ,  0 ,  0];
      [ 0,  0,  0,  1,   0,  0,  0,  0]->[ 1 ,  0 ,  0];
      [ 1,  1,  X,  X,   1,  0,  X,  X]->[ 1 ,  0 ,  0];
      [ 1,  0,  1,  X,   1,  0,  0,  X]->[ 1 ,  0 ,  0];
      [ 1,  0,  0,  1,   1,  0,  0,  0]->[ 1 ,  0 ,  0];
      [ 0,  1,  1,  X,   0,  1,  0,  X]->[ 1 ,  0 ,  0];
      [ 0,  1,  0,  1,   0,  1,  0,  0]->[ 1 ,  0 ,  0];
      [ 0,  0,  1,  1,   0,  0,  1,  0]->[ 1 ,  0 ,  0];

 end  COMP_OR
```

Figure 11. Using Truth Tables and Equations in ABEL to Generate a Comparator PROM File (continued)

LOG/iC

LOG/iC by ISDATA probably has the best support of PROM devices due to its ability to create a PROM file of any size. All the programmer has to do is to tell the compiler how many inputs and how many outputs the PROM should have. The above ABEL file is reproduced in *Figure 12* using LOG/iC.

Although not illustrated in the last two examples, both ABEL and LOG/IC are capable of using state machine input formats.

```
*IDENTIFICATION
 This example uses an 8 bit prom as a 4 bit comparator and does some
 additional misc.  logic

*X-NAMES                  !Define the input pins.
 B[3..0], A[3..0];        !Pins are defined MSB first,
                          !Therefor, B3 will be connected
                          !to address bit 7 and A0 will
                          !be connected to address bit 0.
*Y-NAMES                  !Define the output pins
 AGB,ALB,EQUAL,ALL_HIGH,
 OR_BITS[3..0];

*BOOLEAN EQUATIONS
 ALL_HIGH = B3&B2&B1&B0&A3&A2&A1&A0;
 OR_BITS3 = A3 # B3;
 OR_BITS2 = A2 # B2;
 OR_BITS1 = A1 # B1;
 OR_BITS0 = A0 # B0;

*FUNCTION-TABLE
$ ((A3,A2,A1,A0, B3,B2,B1,B0)):((AGB,ALB,EQUAL));
     0  0  0  0   0  0  0  0  :   0   0    1; A=B CONDITIONS
     0  0  0  1   0  0  0  1  :   0   0    1;
     0  0  1  0   0  0  1  0  :   0   0    1;
     0  0  1  1   0  0  1  1  :   0   0    1;
     0  1  0  0   0  1  0  0  :   0   0    1;
     0  1  0  1   0  1  0  1  :   0   0    1;
     0  1  1  0   0  1  1  0  :   0   0    1;
     0  1  1  1   0  1  1  1  :   0   0    1;
     1  0  0  0   1  0  0  0  :   0   0    1;
     1  0  0  1   1  0  0  1  :   0   0    1;
     1  0  1  0   1  0  1  0  :   0   0    1;
     1  0  1  1   1  0  1  1  :   0   0    1;
     1  1  0  0   1  1  0  0  :   0   0    1;
     1  1  0  1   1  1  0  1  :   0   0    1;
     1  1  1  0   1  1  1  0  :   0   0    1;
     1  1  1  1   1  1  1  1  :   0   0    1;
```

Figure 12. Using LOG/iC to Generate a Comparator PROM File

```
      0  0  0  0   -  -  -  1  :   0   1    0; A < B CONDS.
      0  0  0  -   -  -  1  -  :   0   1    0;
      0  0  -  -   -  1  -  -  :   0   1    0;
      0  -  -  -   1  -  -  -  :   0   1    0;
      1  0  -  -   1  1  -  -  :   0   1    0;
      1  0  0  -   1  0  1  -  :   0   1    0;
      1  0  0  0   1  0  0  1  :   0   1    0;
      0  1  0  -   0  1  1  -  :   0   1    0;
      0  1  0  0   0  1  0  1  :   0   1    0;
      0  0  1  0   0  0  1  1  :   0   1    0;

      1  -  -  -   0  -  -  -  :   1   0    0; A > B CONDS.
      0  1  -  -   0  0  -  -  :   1   0    0;
      0  0  1  -   0  0  0  -  :   1   0    0;
      0  0  0  1   0  0  0  0  :   1   0    0;
      1  1  -  -   1  0  -  -  :   1   0    0;
      1  0  1  -   1  0  0  -  :   1   0    0;
      1  0  0  1   1  0  0  0  :   1   0    0;
      0  1  1  -   0  1  0  -  :   1   0    0;
      0  1  0  1   0  1  0  0  :   1   0    0;
      0  0  1  1   0  0  1  0  :   1   0    0;

*ROM
 TYPE = 8_IN_8_OUT;
 INPUTS = 8;
 OUTPUTS = 8;

*RUN
 PROG = INTEL;          Produce an INTEL-HEX output format

*END
```

Figure 12. Using LOG/iC to Generate a Comparator PROM File (continued)

Conclusion

PROM files can be easily generated in a variety of ways. If a complex function is desired, a high-level language approach is probably the best method. However, if a logical function is the desired result, PLD development tools will more than suffice.

Document #: PRO0003–0292

Introduction to the CY7C270 Processor-Intelligent PROM

This application note introduces the CY7C270 Processor-Intelligent (PI) PROM, the industry's first UV-erasable PROM with a user-programmable processor interface. We begin with an architectural description of the CY7C270 PROM. Next, the advantages of using the CY7C270 PROM instead of conventional PROMs and PROMs designed for specific processors are highlighted. Finally, we will discuss designing non-volatile memory for various popular RISC and CISC processors and DSP devices using the the CY7C270.

Architectural Features of the CY7C270

Figure 1 is a block diagram of the CY7C270. In addition to the 16K x 16 core PROM array, the PROM has a register/latch on the address lines, an internal counter, three polarity programmable chip selects, and a polarity programmable output enable, each of which will be described in more detail in the following sections.

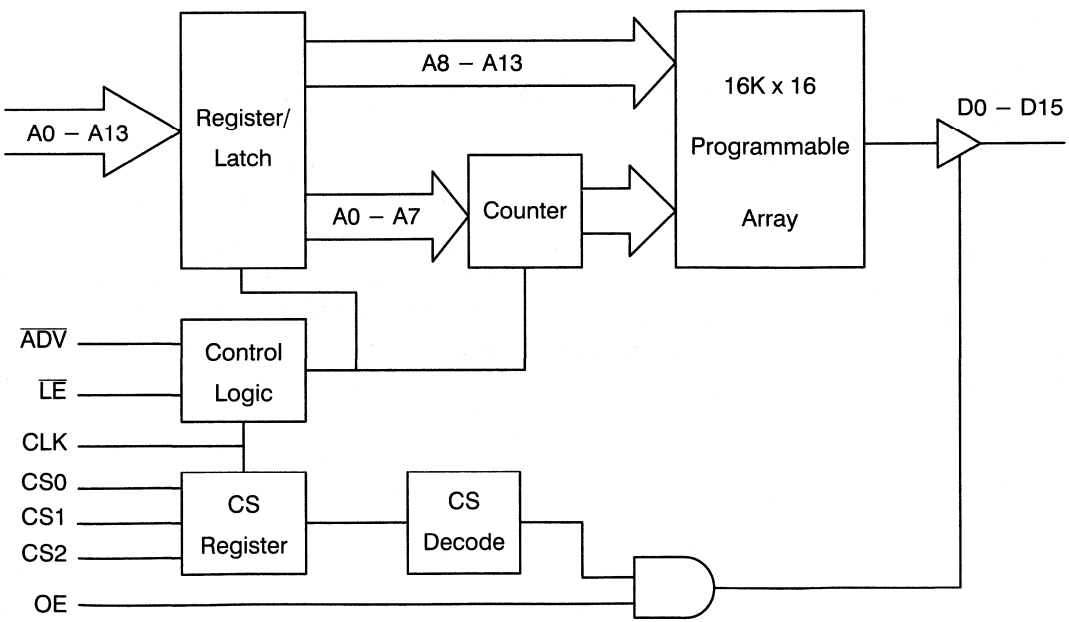


Figure 1. Block Diagram of the CY7C270 Processor-Intelligent PROM

Input Register/Latch

A RISC processor puts a valid address on the address bus just before the rising edge of clock. This means that in order to achieve the highest performance in RISC-processor-based systems, a register is required to capture the address. On the other hand, a CISC processor puts a valid address on the address bus midway through a clock cycle, so a latch is required to start the access in CISC-processor-based systems. The CY7C270's programmable register/latch on the address lines can be configured to be either a register or a latch, thus allowing for easier interface to both RISC- and CISC-processor-based systems.

Two control signals, CLK and $\overline{LE}$, are associated with the registered and latched modes of operation. A read access in registered mode is initiated on the rising edge of CLK if all three chip selects are asserted, $\overline{LE}$ is sampled LOW, and the PROM is configured for registered mode of operation. A read access in latched mode is initiated on the falling edge of $\overline{LE}$ if all three chip selects are asserted and the PROM is configured for latched mode of operation.

On-Chip Programmable Counter

The CY7C270 has an on-chip programmable counter that can be configured to follow a 2-, 4-, or 8-bit linear burst sequence and the non-linear 486 and Pentium burst sequence. A burst counter on the PROM eliminates the need to implement the counter in external logic, a requirement when interfacing PROMs to high performance burst processors. Additionally, with a initial access time of 25 ns and burst access time of 11 ns (in the fastest version, CY7C270-15), the PROM can be interfaced to most 25-MHz RISC and 40-MHz CISC systems with zero wait states. Some processors can be interfaced to the slower versions of the CY7C270 as shown in the design examples section of this application note.

The $\overline{ADV}$ signal controls the burst counter advance logic. The signal is active LOW, is sampled on the rising edge of clock, and when asserted, increments the counter to the next counter sequence. Some processors, like the AMD 29000, can suspend a burst. The CY7C270 incorporates support for this capability via the $\overline{ADV}$ signal. When this signal is taken HIGH, the burst is automatically suspended. To continue the burst from where it left off, the $\overline{ADV}$ signal just needs to be reasserted. Note that if both $\overline{LE}$ and $\overline{ADV}$ are asserted at the same time, $\overline{ADV}$ is ignored.

Additionally, the CY7C270 can be configured with the counter turned off, making it the fastest 256K asynchronous PROM available in the market today.

Polarity Programmable Chip Selects and Output Enable

The CY7C270 has three programmable chip selects, allowing up to 256 Kbytes of PROM to be interfaced to a processor with no external decode logic. The CY7C270 allows the user to individually program the polarity of each chip-select pin. The polarity of the asynchronous $\overline{OE}$ pin is also user programmable.

16-Bit Wide Outputs

The CY7C270 provides a 16-bit-wide output, thus halving the number of PROMs required when interfacing with 16-bit or wider processors.

Zero wait-state performance

The CY7C270 has an initial access time of 25 ns and a burst access time of 11 ns, allowing it to be interfaced to 40-MHz CISC systems and 25-MHz RISC systems with zero wait states. With a burst access time of 11 ns, the CY7C270 can provide zero-wait-state performance on burst accesses with 66-MHz systems. Note that in this case, a wait state may be needed on the initial access of a burst sequence.

Benefits of Using the CY7C270 Processor-Intelligent PROM

The CY7C270 offers several advantages over conventional PROMs. Here is a summary of these advantages:

- 25-ns initial access and 11-ns burst access times allow zero-wait-state performance with most 40-MHz CISC and 25-MHz RISC processors and high-performance DSP devices.
- Input register/latch and on-chip burst counter allow glueless interface to most CISC and RISC processors and DSP devices.

- 16-bit-wide organization reduces the number of PROMs required when interfacing to 16-bit or wider processors.

This translates into the following benefits for the user:

- Enhanced system performance
- Reduced component count
- Board space savings
- Increased system reliability
- Reduced system cost
- Compatibility with many different processors

Interfacing the CY7C270 to CISC Processors

386 at 33 MHz

The 386 processor is offered by both Intel and AMD at a frequency of 33 MHz. The memory read timing diagram for this processor is shown in *Figure 2*.

CPU-Memory Interface Signals

The 386 uses the following signals to interface with memory.

CLK2: Input Clock.

$\overline{ADS}$: Address Status, indicates that a valid bus cycle definition and address are driven on the processor pins.

A[31:2]: Address Bus, outputs memory or I/O address.

D[31:0]: Data Bus, transfers data to and from the processor.

M/$\overline{IO}$: Memory I/O is a bus cycle definition pin that differentiates between memory cycles and I/O cycles.

W/$\overline{R}$: Write/Read is a bus cycle definition pin that differentiates between write and read cycles.

$\overline{READY}$: Bus Ready terminates the bus cycle.

Interface Considerations

Interfacing the CY7C270 Processor Intelligent PROM to the 386 processor requires using the clock

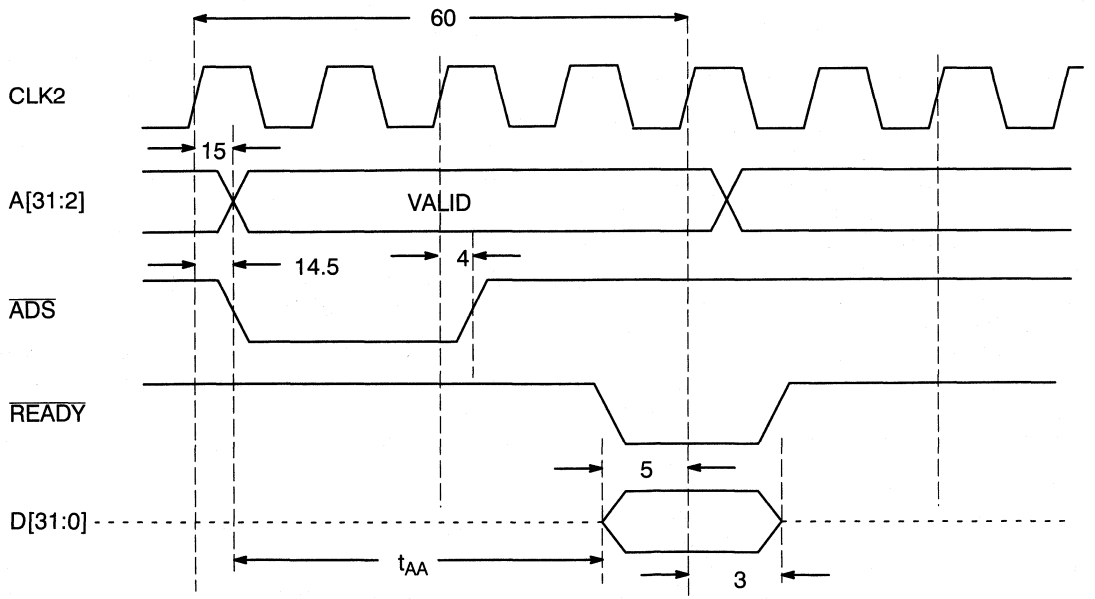


Figure 2. Memory Read Timing for the 386

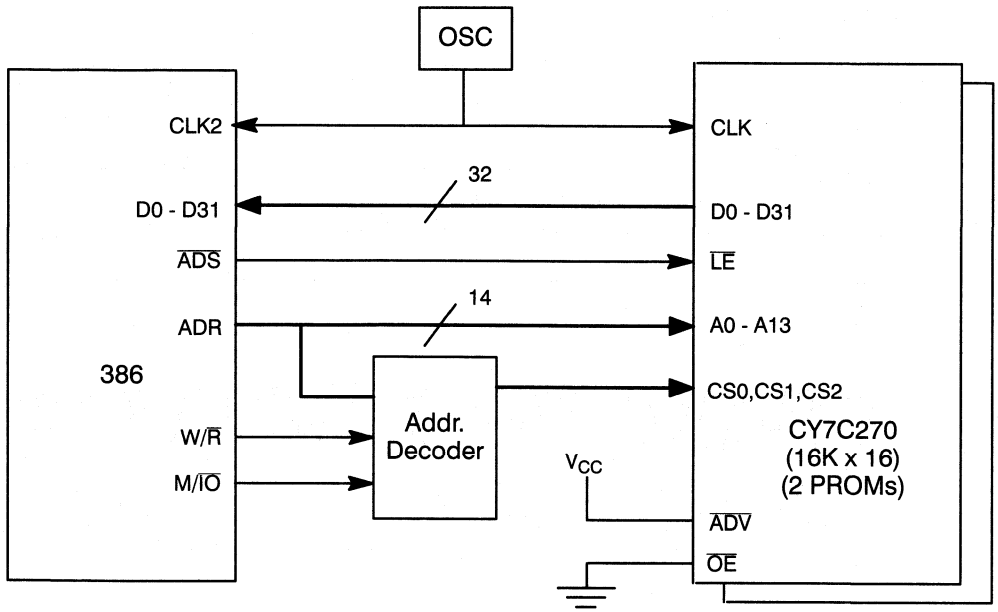


Figure 3. Interfacing the CY7C270 to the 386

signal, the address bus, the data bus, and the address status signal. An address decoder PLD uses the address signals and the W/$\overline{R}$ and M/$\overline{IO}$ signals to select the PROM bank. Additionally, the $\overline{READY}$ signal, which is dependent on the SRAM and I/O in the system, can be generated in this PLD.

Since the 386 processor does not perform burst reads, the $\overline{ADV}$ signal on the PROM is tied HIGH. The $\overline{ADS}$ signal from the processor is fed directly into the $\overline{LE}$ pin on the PROM. CLK2 from the processor drives the PROM clock. The PROM is used in latched mode, since the address comes out much later than the rising edge of clock.

CY7C270 Configuration

Output Enable: Active LOW
Counter Configuration: Don't Care
Mode: Latched
Chip Selects: User Defined
Burst Enable: No Burst

Timing Requirement

PROM access time required:

t_{AA} = 2 clock periods – address valid delay – i/p data set-up

t_{AA} = 60 – 15 – 5 = 40 ns

The above number shows that the CY7C270 can easily interface to the 386 processor at 33 MHz with zero wait states. In this case, the CY7C270-30 with an initial access time of 35 ns can be used. *Figure 3* shows the CPU-PROM interface.

AMD 386 DXL at 40 MHz

The AMD 386 DXL is the only 386 processor currently available at a 40-MHz clock frequency. In this section, we will discuss the PROM access time requirements for interfacing to this processor. The CPU-memory read timing and the CPU-PROM interface figures for this processor are not shown here because they are identical to the figures shown in the previous section, except the address delay and input data set-up are changed to 13 ns and 4 ns, respectively.

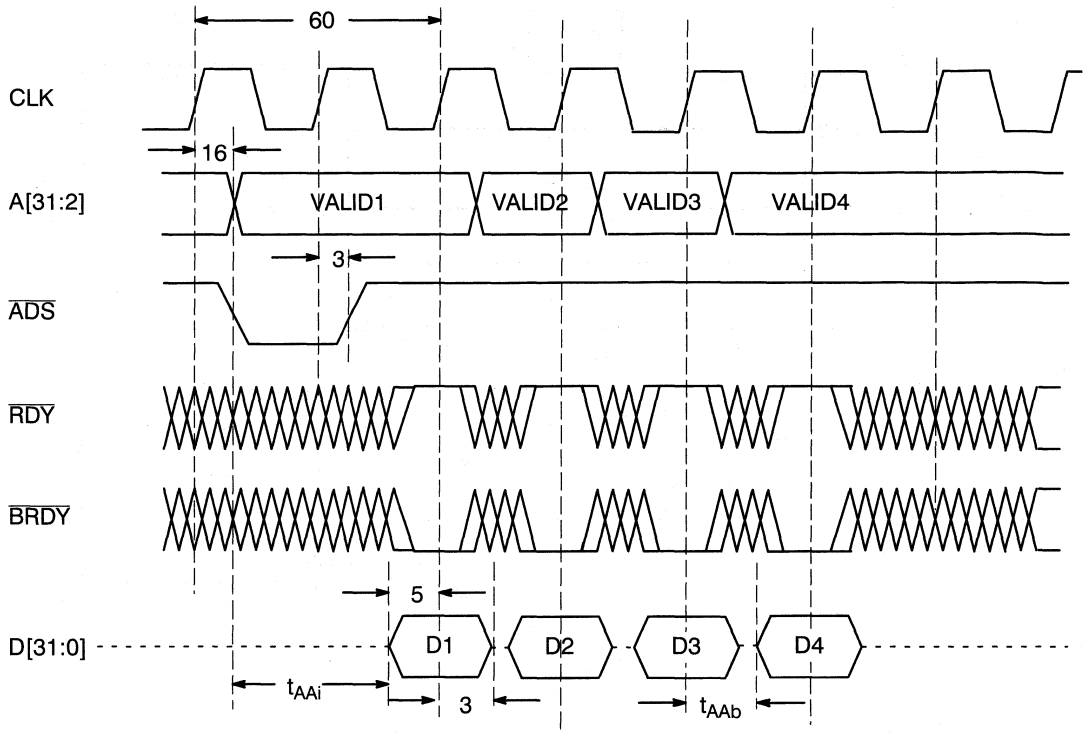


Figure 4. Memory Read Timing for the 486

Timing Requirement

PROM access time required:

t_{AA} = 2 clock periods – address valid delay – i/p data set-up

t_{AA} = 50 – 13 – 4 = 33 ns

The above number shows that the CY7C270 can easily interface to the 386 processor at 40 MHz with zero wait states. In this case, the CY7C270-20, with an initial access time of 28 ns can be used.

486 at 33 MHz

The 486 processor is currently offered by Intel at 25, 33 and 50 MHz. The 486 differs from the 386 by having a non-linear burst sequence. The memory read timing for the 486 is shown in *Figure 4*.

CPU-Memory Interface Signals

The 486 uses the following signals to interface with memory.

CLK: Input clock

$\overline{ADS}$: Address Status, indicates that a valid bus cycle definition and address are driven on the processor pins.

A[31:2]: Address Bus, outputs memory or I/O address.

D[31:0]: Data Bus, transfers data to and from the processor.

M/$\overline{IO}$: Memory I/O is a bus cycle definition pin that differentiates between memory cycles and I/O cycles.

W/$\overline{R}$: Write/Read is a bus cycle definition pin that differentiates between write and read cycles.

$\overline{RDY}$: The Non-Burst Ready input indicates that the current bus cycle is complete.

$\overline{BRDY}$: The Burst Ready signal indicates that the current burst bus cycle is complete. It is sampled on the second and subsequent clocks of a burst cycle.

Interface Considerations

Interfacing the CY7C270 Processor-Intelligent PROM to the 486 processor requires the use of the clock signal, the address bus, the data bus and the address status signal. An address decoder uses the address signals and the W/$\overline{R}$ and M/$\overline{IO}$ signals to select the PROM bank. The address decoder PLD can also generate the $\overline{RDY}$ and $\overline{BRDY}$ signals which depend on various other components (SRAM, I/O) in the system, too.

The $\overline{ADS}$ signal from the processor is fed into the $\overline{LE}$ signal on the PROM. Note that the $\overline{ADV}$ signal on the PROM is tied LOW. This is because the 486 always bursts on a cachable read. If a burst is terminated by the processor and a new address is placed on the address bus, the $\overline{ADS}$ signal goes LOW, forcing $\overline{LE}$ on the PROM to go LOW, which overrides the $\overline{ADV}$ signal.

CY7C270 Configuration

Output Enable: Active LOW
Counter Configuration: 486 2-bit counter
Mode: Latched
Chip Selects: User defined
Burst Enable: Burst (follow counter configuration)

Timing Requirement

PROM access time required on initial access:

t_{AAi} = 2 clock periods – address valid delay – i/p data set-up

t_{AAi} = 60 – 16 – 5 = 39 ns

PROM access time required on burst access:

t_{AAb} = 1 clock period – i/p data set-up

t_{AAb} = 30 – 5 = 25 ns

The address valid delay parameter is not used in calculating t_{AAb} since the burst address is generated internally on the PROM by the on-chip counter. The above numbers show that the CY7C270 can easily interface to the processor with zero wait states. In this case, the CY7C270-30 with an initial access time of 35 ns and a burst access time of 15 ns can be used. *Figure 5* shows the CPU-PROM interface.

Motorola 68040 at 40 MHz

The Motorola 68040 processor is offered at speeds of 25, 33 and 40 (preliminary) MHz. In this section, we will interface the CY7C270 to the 40-MHz version of the 68040. The memory read timing is shown in *Figure 6*.

CPU-Memory Interface Signals

The 68040 uses the following signals to interface with memory.

CLK: Processor clock input.

A[31:0]: Address Bus, outputs memory and I/O addresses.

D[31:0]: Data Bus, transfers data to and from the processor.

$\overline{TS}$: Transfer Start, indicates the beginning of a bus transfer.

$\overline{TA}$: Transfer Acknowledge, is asserted to acknowledge a bus transfer.

Interface Considerations

Interfacing the CY7C270 Processor Intelligent PROM to the 68040 processor will require the use of the CLK signal, the address bus, the data bus and the $\overline{TS}$ signal. The $\overline{TS}$ signal is fed directly into the $\overline{LE}$ pin of the CY7C270. Additionally, the $\overline{ADV}$ pin on the PROM is tied LOW, since the 68040 processor always bursts on an instruction read from external memory. The chip selects to the PROM are generated from an address decoder, which also generates chip selects for other memory and I/O components in the system. The $\overline{TA}$ (Transfer Acknowledge) signal is generated in the address decoder too.

CY7C270 Configuration

Output Enable: Active LOW
Counter Configuration: Linear 2-bit counter
Mode: Latched

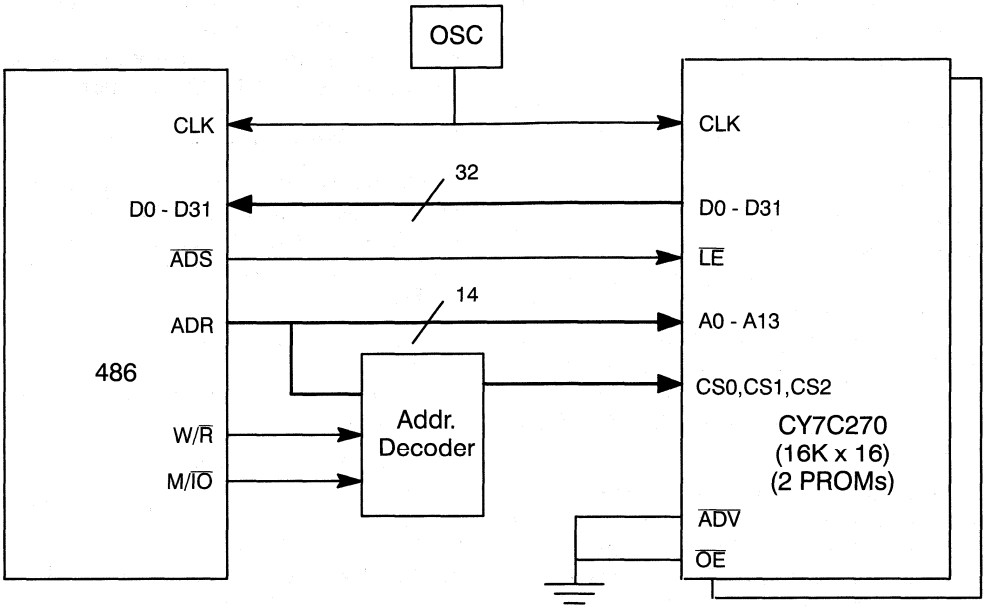


Figure 5. Interfacing the CY7C270 to the 486

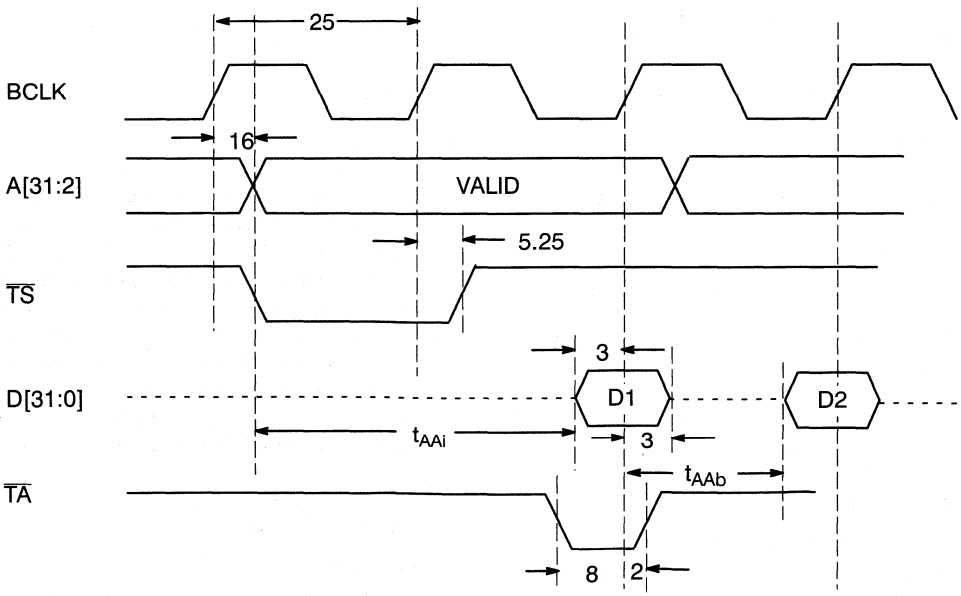


Figure 6. Memory Read Timing for the 68040

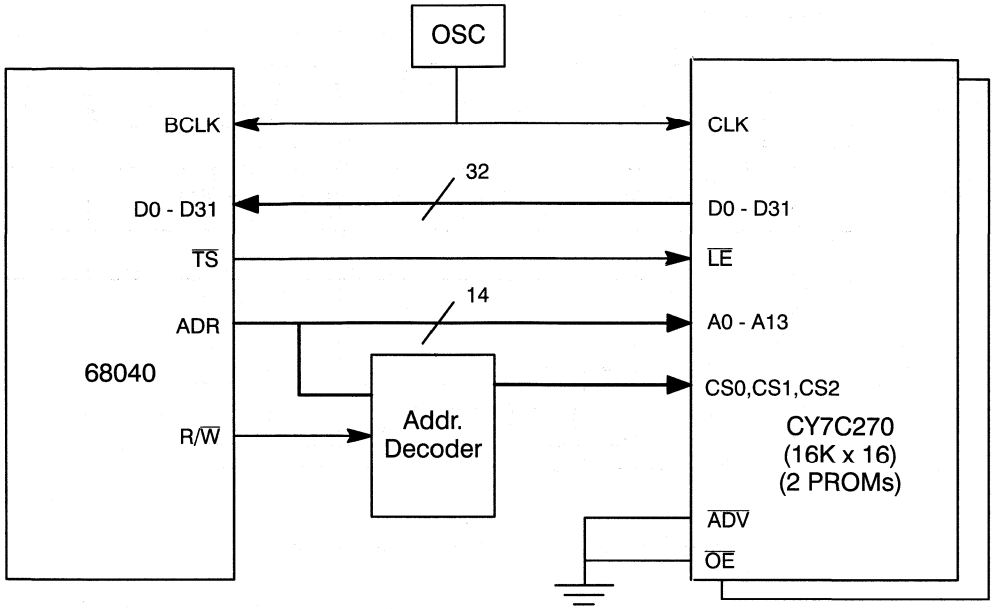


Figure 7. Interfacing the CY7C270 to the 68040

Chip Selects: User defined
Burst Enable: Burst (follow counter configuration)

Timing Requirement

PROM access time required on initial access:

t_{AAi} = 2 clock periods – address valid delay – i/p data set-up

t_{AAi} = 50 – 16 – 3 = 31 ns

PROM access time required on burst access:

t_{AAb} = 1 clock period – i/p data set-up

t_{AAb} = 25 – 3 = 22 ns

The address valid delay parameter is not used in calculating t_{AAb} since the burst address is generated internally on the PROM by the on-chip counter. As can be seen, the CY7C270 can easily interface to the processor with zero wait states. In this case the CY7C270-20 with an initial access time of 28 ns and a burst access time of 12 ns can be used. The CPU-memory interface is as shown in *Figure 7.*

Interfacing the Processor Intelligent PROM to RISC Processors

Intel 960CA at 33 MHz

The 960 processor is offered by Intel in the CA, KB and CF configurations. In this section we will consider timing for the 960CA configuration, though it must be noted that the CY7C270 can also interface to the other configurations. The memory read timing for the 960CA is shown in *Figure 8.*

CPU-Memory Interface Signals

The 960 uses the following signals to interface with memory.

PCLK: Processor Output Clock. The processor has two output clock pins, PCLK2 and PCLK1, which can be tied together if only one output clock is required. All input and output timings are specified with respect to this clock signal.

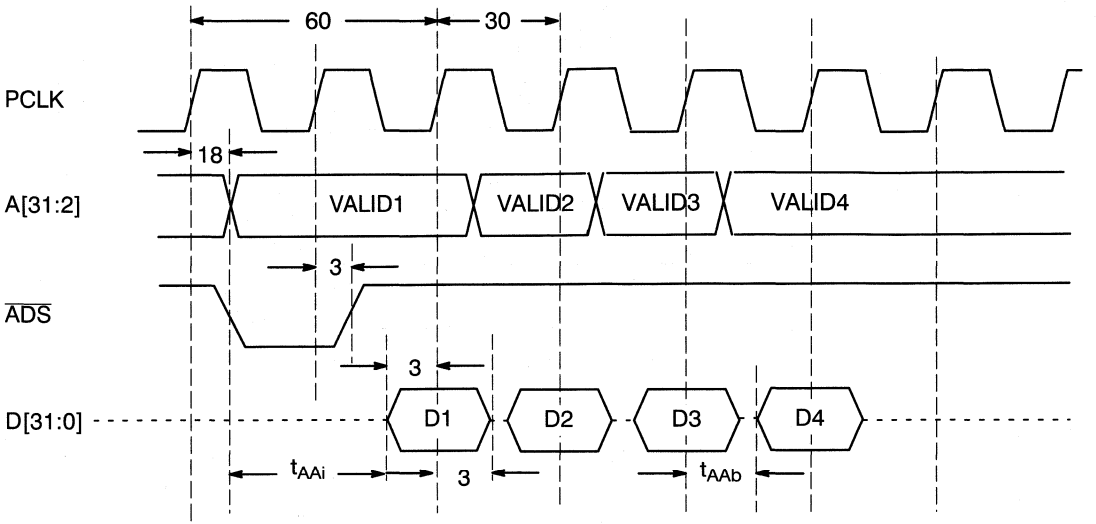


Figure 8. Memory Read Timing for the 960CA

$\overline{ADS}$: Address Status, indicates that a valid bus cycle definition and address are driven on the processor pins.

A[31:2]: Address Bus, outputs memory or I/O address.

D[31:0]: Data Bus, transfers data to and from the processor.

W/$\overline{R}$: Write/Read is a bus cycle definition pin that differentiates between write and read cycles.

$\overline{READY}$: The Ready input signals the termination of a data transfer.

Interface Considerations

Interfacing the CY7C270 Processor-Intelligent PROM to the 960CA processor will require the use of the clock signal, the address bus, the data bus and the address strobe signal. The $\overline{ADS}$ signal of the processor is tied to the $\overline{LE}$ pin of the PROM. The $\overline{ADV}$ signal on the PROM is tied LOW so that the PROM will burst on every instruction access. If the processor outputs a new address or breaks up the burst sequence with the $\overline{BTERM}$ signal, the $\overline{ADS}$ signal will go LOW, driving the $\overline{LE}$ pin on the PROM LOW which will override the signal on the $\overline{ADV}$ pin. The PROM chip selects are generated by an address decoder.

CY7C270 Configuration

Output Enable: Active LOW
Counter Configuration: Linear 2-bit counter
Mode: Latched
Chip Selects: User defined
Burst Enable: Burst (follow counter configuration)

Timing Requirement

PROM access time required on initial access:

t_{AAi} = 2 clock periods – address valid delay – i/p data set-up

t_{AAi} = 60 – 18 –3 = 39 ns

PROM access time required on burst access:

t_{AAb} = 1 clock period – i/p data set-up

$t_{AA(burst)}$ = 30 – 3 = 27 ns

The address valid delay parameter is not used in calculating t_{AAb} since the burst address is generated internally on the PROM by the on-chip counter. The above numbers show that the CY7C270 can easily interface to the processor with zero wait states. In this case, the CY7C270-30 with an initial access time of 35 ns and a burst access time of 15 ns can be used. *Figure 9* shows the CPU-PROM interface.

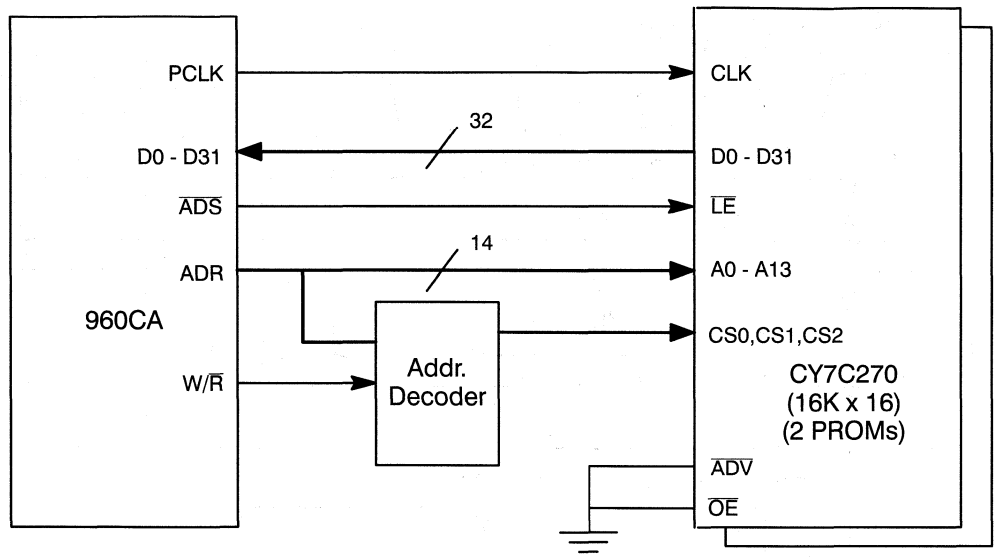


Figure 9. Interfacing the CY7C270 to the 960CA

AMD Am29000 at 33 MHz

The AMD 29K processor is offered by AMD in different configurations, including the 29000, 29005, 29200 etc. In this section, we will interface the CY7C270 to the Am29000 processor. The memory read timing diagram for this processor is shown in *Figure 10*.

CPU-Memory Interface Signals

The Am29000 processor uses the following signals to interface with instruction memory.

SYSCLK: Processor clock input from external clock generator at processor's clock frequency.

A[31:0]: Address Bus, outputs memory and I/O addresses.

I[31:0]: Instruction Bus, transfers instructions to the processor from external memory.

$\overline{\text{IREQ}}$: Instruction Request, requests an instruction access. When this signal is active, the address of an instruction appears on the address bus.

$\overline{\text{IBREQ}}$: Instruction Burst Request, used to request burst-mode instruction accesses.

R/$\overline{\text{W}}$: Read/Write, signal differentiates between read and write cycles.

Interface Considerations

The Am29000 processor family has a unique three bus organization: One Address bus, one Instruction bus and a Data bus. The Instruction bus is unidirectional, i.e., it is a read-only bus. Because of the unidirectional nature of this bus, PROMs are required to store instructions. If RAMs were used, the user would have to derive methods to load the RAMs.

Interfacing the CY7C270 Processor-Intelligent PROM to the Am29000 processor will require the use of the SYSCLK signal, the address bus, the instruction bus, the $\overline{\text{IREQ}}$ signal, the $\overline{\text{IBREQ}}$ signal and the R/$\overline{\text{W}}$ signal. The address decoder uses the address signals and the R/$\overline{\text{W}}$ signal to select the PROM bank.

Figure 10 shows that the processor outputs a valid address 1 ns/12 ns (min./max.) after the rising edge of clock cycle T0. Additionally, it expects instruction/data on the next rising edge of clock. For zero-

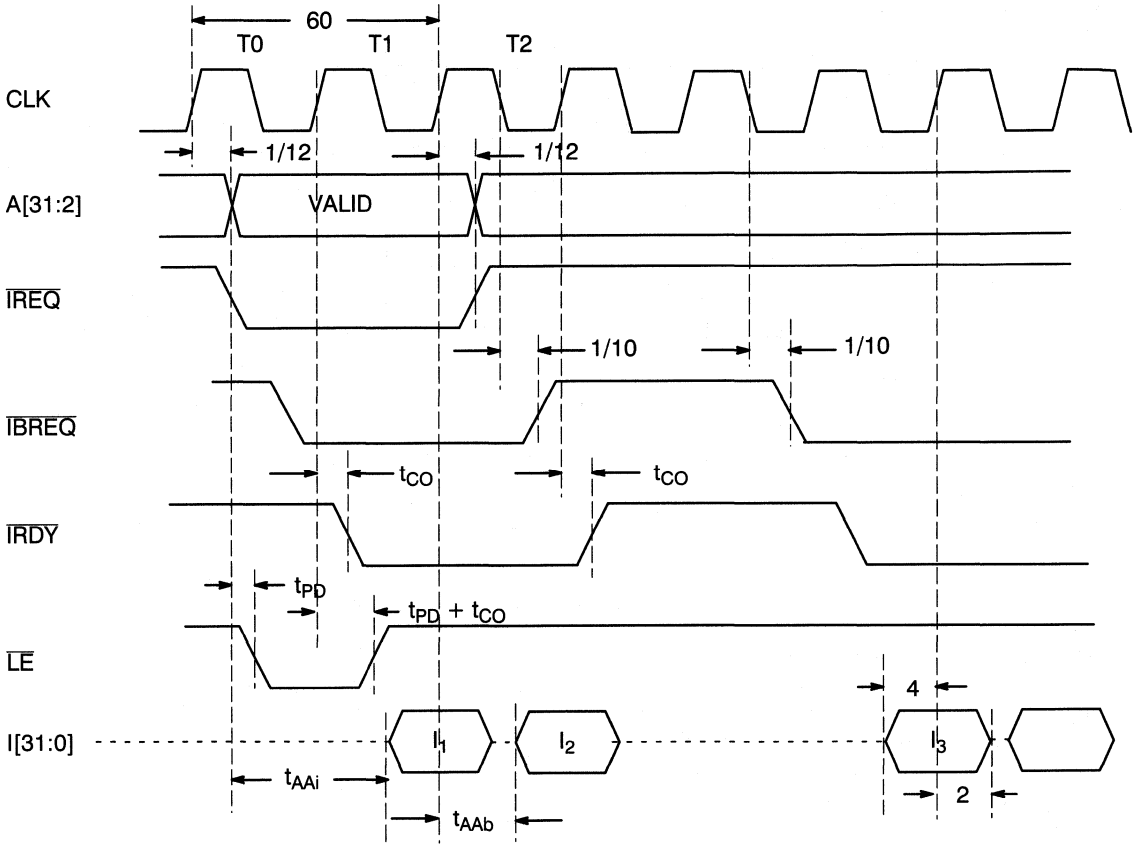


Figure 10. Memory Read Timing for the Am29000

wait-state operation on the initial access, the PROM access time requirement is:

t_{AAi} = 1 clock period – address valid delay – i/p data set-up

t_{AAi} = 30 – 12 – 4 = 14ns

This is not possible even with a very high performance PROM such as the Processor-Intelligent PROM. Hence, a wait state on the initial access becomes mandatory.

However, on burst accesses, the Processor-Intelligent PROM can generate the new address internally. By eliminating the address valid delay parameter in the above equation, we get a PROM burst access time requirement of 26 ns, which is easily met by the Processor-Intelligent PROM. The Am29000 also has an unusually long burst sequence: up to 1 Kbyte or 256 32-bit words long. The CY7C270 can support this long burst sequence with its on-chip configurable 8-bit burst counter.

Therefore, the CPU-Memory system will operate with one wait-state on the initial access and zero-wait-states on the burst access. Certain control signals will also need to be generated, which is discussed in the following section.

Control Signal Generation

Referring to *Figure 11*, the $\overline{\text{IREQ}}$ from the processor appears after the rising edge of clock cycle T0 and stays valid until the first instruction is presented.

$\overline{IBREQ}$ appears after the falling edge of clock cycle T0. Both signals are asserted on the rising edge of clock cycle T2, when the first instruction is presented to the processor and when the CY7C270 has to start bursting. If the $\overline{IREQ}$ signal were fed directly into the $\overline{LE}$ input of the CY7C270, and the $\overline{IBREQ}$ signal to the $\overline{ADV}$ input, the $\overline{LE}$ input would override the $\overline{ADV}$ input and the CY7C270 would not perform a burst read. Hence, only one of the two signals $\overline{IREQ}$ and $\overline{IBREQ}$ can be fed into the corresponding $\overline{LE}$ or $\overline{ADV}$ inputs of the CY7C270.

The Am29000 processor has the capability to suspend a burst by driving the $\overline{IBREQ}$ signal HIGH during a burst access. The CY7C270 incorporates support for this feature on the $\overline{ADV}$ input, implying that $\overline{IBREQ}$ can be fed directly into the $\overline{ADV}$ input of the CY7C270.

Therefore, a new signal, $\overline{LE}$, needs to be generated and fed to the $\overline{LE}$ input of the CY7C270. $\overline{LE}$ should be asserted when $\overline{IREQ}$ is asserted, and deasserted at least a time t_{LES} ($\overline{LE}$ set-up to Latch Close parameter on the CY7C270) before the rising edge of clock cycle T2. To generate $\overline{LE}$, a 2-bit counter is set up in a PLD, and the count is incremented when $\overline{IREQ}$ is sampled LOW on the rising edge of clock. $\overline{LE}$ is asserted when $\overline{IREQ}$ is asserted, and deasserted on the count of 1. Accounting for the propagation delay through the PLD, $\overline{LE}$ is asserted a time t_{PD} after $\overline{IREQ}$ and deasserted a time (t_{PD} + t_{CO}) after the rising edge of clock cycle T1.

The $\overline{IRDY}$ signal also needs to be generated to indicate to the processor that the CY7C270 has placed valid data on the data bus. $\overline{IBREQ}$ is used to generate this signal.

The complete ABEL equations for generating $\overline{LE}$ and $\overline{IRDY}$ are shown in Appendix A. Note that the generation of the above control signals can be incorporated into the address decoder PLD.

CY7C270 Configuration

Output Enable: Active LOW
Counter Configuration: Linear 8-bit counter
Mode: Latched
Chip Selects: User defined
Burst Enable: Burst (follow counter configuration)

Timing Requirement

PROM access time required on initial access:

t_{AAi} = 2 clock periods – $\overline{LE}$ valid delay – i/p data set-up

$t_{AAi} = 60 - (12 + t_{PD}) - 4 = (44 - t_{PD})$ ns

A 15-ns PLD can be used to generate the control signals. The PROM access time required on the initial access will then be 29 ns. Using such a PLD will also ensure that the $\overline{LE}$ signal meets the t_{LES} parameter on the CY7C270.

PROM access time required on burst access:

t_{AAb} = 1 clock period – i/p data set-up

$t_{AAb} = 30 - 4 = 26$ ns

In this case, the CY7C270-20, with an initial access time of 28 ns and a burst access time of 12 ns can be used. Note that using a faster PLD will allow the use of the slower CY7C270–30, with an initial access time of 35 ns and a burst access time of 15 ns. *Figure 11* shows the CPU-PROM interface.

Cypress CY7C611A at 25 MHz

The CY7C611A is an embedded SPARC processor, running at 25 MHz. The memory read timing for this processor is shown in *Figure 12*.

CPU-Memory Interface Signals

The CY7C611A uses the following signals to interface with memory.

CLK: Processor Clock Input

A[23:0]: Address Bus, outputs memory and I/O addresses.

D[31:0]: Data Bus, transfers data to and from the processor.

Interface Considerations

Interfacing the CY7C270 Processor-Intelligent PROM will require the use of the clock signal, the address bus and the data bus. From the timing diagram, it can be seen that input data to the PROM requires a hold time of 5 ns. The CY7C270 has a data

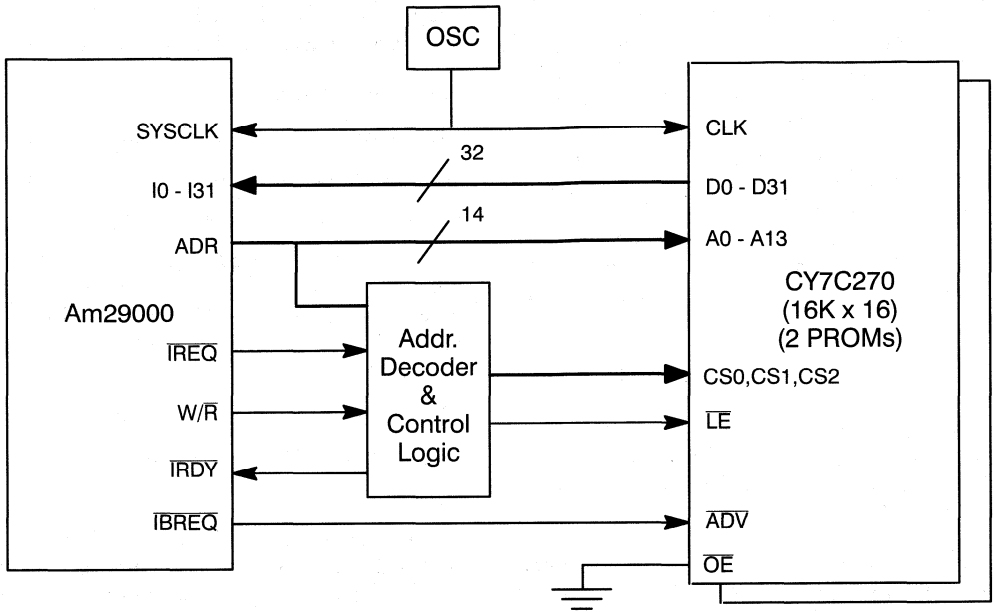


Figure 11. Interfacing the CY7C270 to the Am29000

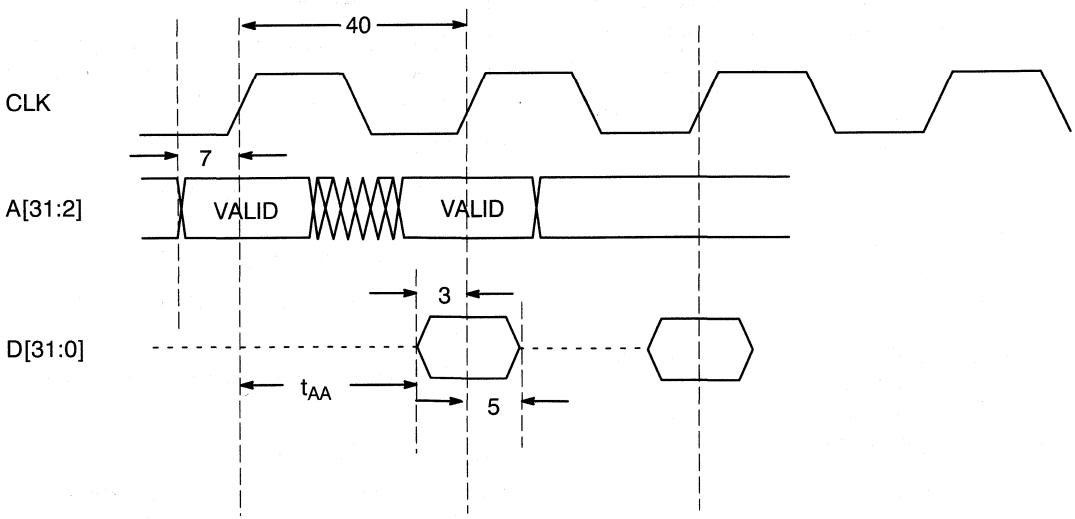


Figure 12. Memory Read Timing for the CY7C611

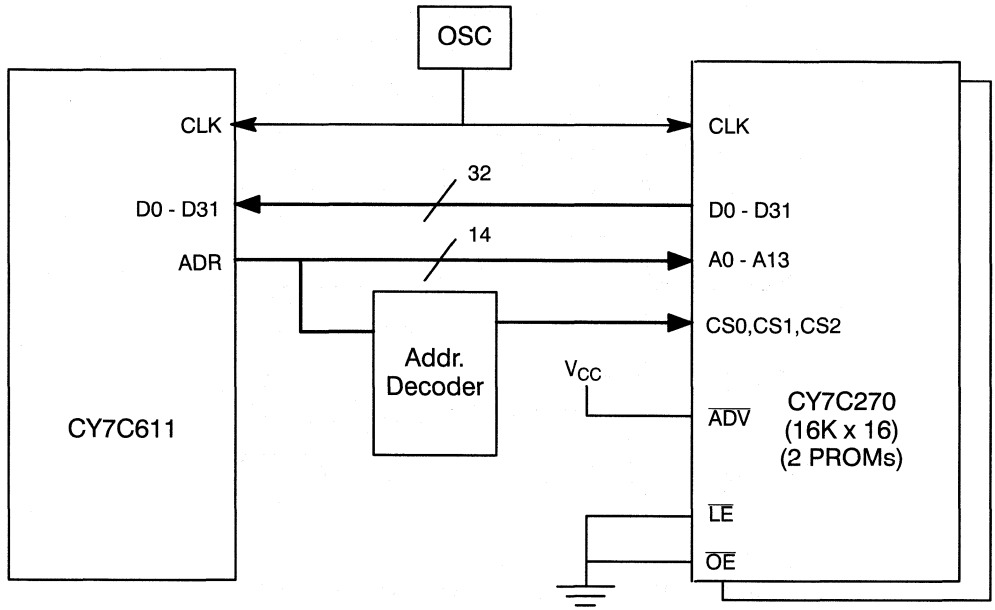


Figure 13. Interfacing the CY7C270 to the CY7C611A SPARC Processor

hold time of 3 ns. The additional 2 ns can be obtained by:

- Delaying the clock to the PROM by a minimum of 2 ns. The simplest delay element is a non-inverting buffer. With a 25-ns access time, there is sufficient margin to accommodate this delay.

- Putting a latch on the outputs of the PROM to capture and hold the data. For example, a transparent latch using the LOW portion of the clock signal as the latch enable should be able to hold data until mid-clock.

The $\overline{LE}$ signal on the PROM is tied LOW, since the PROM needs to sample $\overline{LE}$ LOW on the rising edge of clock in registered mode, and there is no appropriate signal available from the CPU. The $\overline{ADV}$ signal is tied HIGH, since the CPU does not perform burst accesses.

CY7C270 Configuration

Output Enable: Active LOW
Counter Configuration: Don't care
Mode: Registered
Chip Selects: User defined
Burst Enable: No burst

Timing Requirement

PROM access time required on each access:

t_{AA} = 1 clock period – data set-up

t_{AA} = 40 – 3 = 37 ns

These numbers show that the CY7C270 can interface to the 25-MHz CY7C611A embedded SPARC processor with zero wait states. In this case, the CY7C270-30 with an initial access time of 35ns can be used. *Figure 13* shows the CPU-PROM interface.

Motorola 88100 at 25 MHz

The 88100 is first in Motorola's family of RISC processors. In this section, we will interface the

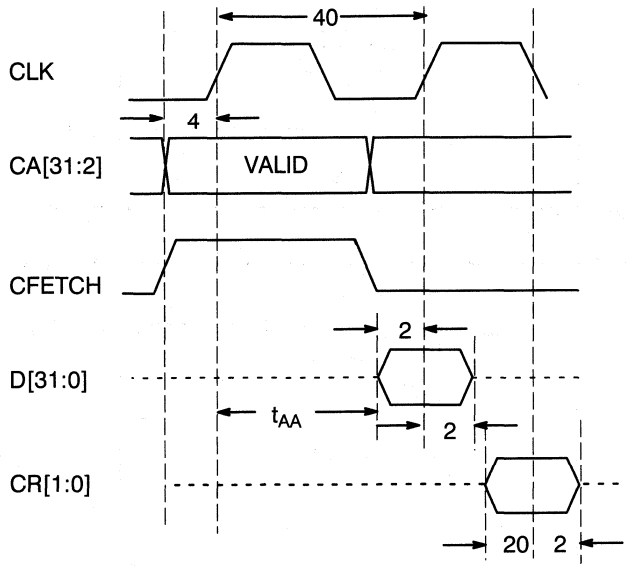


Figure 14. Memory Read Timing for the Motorola 88100

CY7C270 to the 88100 running at 25 MHz. The memory read timing is shown in *Figure 14*.

CPU-Memory Interface Signals

The 88100 processor uses the following signals to interface with memory:

CLK: System clock, running at 25 MHz.

CA[31:2]: Code Address Bus, provides 30-bit word address to the instruction memory space.

C[31:0]: Code bus, transfers instructions to the processor from external memory.

CFETCH: Code Fetch, when asserted indicates that an instruction fetch is occurring.

CR[1:0]: Code Reply, indicates the status of an instruction memory access.

Interface Considerations

Note that the CFETCH signal from the 88100 is active HIGH and becomes valid 4 ns before the rising edge of CLK. The $\overline{LE}$ input on the CY7C270 needs to be sampled active LOW on the rising edge of CLK, to initiate a read in registered mode. Using an inverted CFETCH signal to drive the $\overline{LE}$ pin is not a feasible option, as explained below:

- If CFETCH is inverted and fed into the $\overline{LE}$ input on the PROM while in the registered mode of operation, the t_{LES} parameter ($\overline{LE}$ setup to CLK rise, 3 ns) is not met, since there is a finite delay through the inverter.
- If CFETCH is inverted and fed into the $\overline{LE}$ input on the PROM while in the latched mode of operation, the t_{AHL} parameter (Address Hold from Latch Close, 2 ns) is not met, since the address and CFETCH signals change at the same time.

The solution to this problem is to tie the $\overline{LE}$ signal on the CY7C270 LOW and configure the PROM for registered mode of operation. The PROM will capture the address on the rising edge of CLK, then it will sample $\overline{LE}$ LOW, and present data to the processor t_{AA} ns (Address to Data for Single Read) later.

A signal which needs to be generated is the CR[1:0] signal. This signal indicates the status of the instruction memory access to the processor. This signal can be generated in the address decoder PLD. Finally,

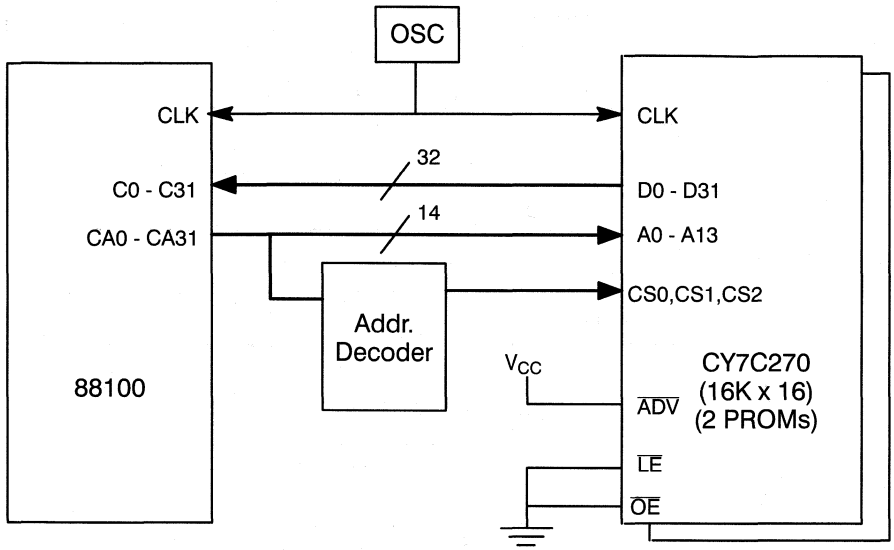


Figure 15. Interfacing the CY7C270 to the Motorola 88100 Processor

the $\overline{ADV}$ signal on the PROM is tied HIGH, since the CPU does not perform burst accesses.

CY7C270 Configuration

Output Enable: Active LOW
Counter Configuration: Don't care
Mode: Registered
Chip Selects: User defined
Burst Enable: No burst

Timing Requirement

PROM access time required on each access:

t_{AA} = 1 clock period – i/p data set-up

t_{AA} = 40 – 2 = 38 ns

The above number shows that the CY7C270 can interface to the 25-MHz Motorola 88100 processor with zero wait states. In this case, the CY7C270-30, with an initial access time of 35 ns can be used. *Figure 15* shows the interface between the processor and the PROM.

Interfacing the Processor-Intelligent PROM to DSP Devices

In this section, we will discuss how to interface the Processor-Intelligent PROM to a popular DSP device. Only one example is provided, since most DSP devices have very similar memory interfaces.

TI TMS320C30

The TI TMS320C30 is a popular high-performance floating-point DSP device used extensively in embedded-control applications. It is offered in 27-, 33- and 40-MHz speeds. In this section, we discuss the interface of the TMS320C30-40 processor with the CY7C270. The CPU-memory read timing diagram for the CY7C270 and the C30 processor is shown in *Figure 16*.

CPU Memory Interface

The TMS320C30 uses the following signals to interface with memory.

H1: External H1 Clock.

H3: External H3 Clock.

A[23:0]: Address Bus, addresses memory and I/O.

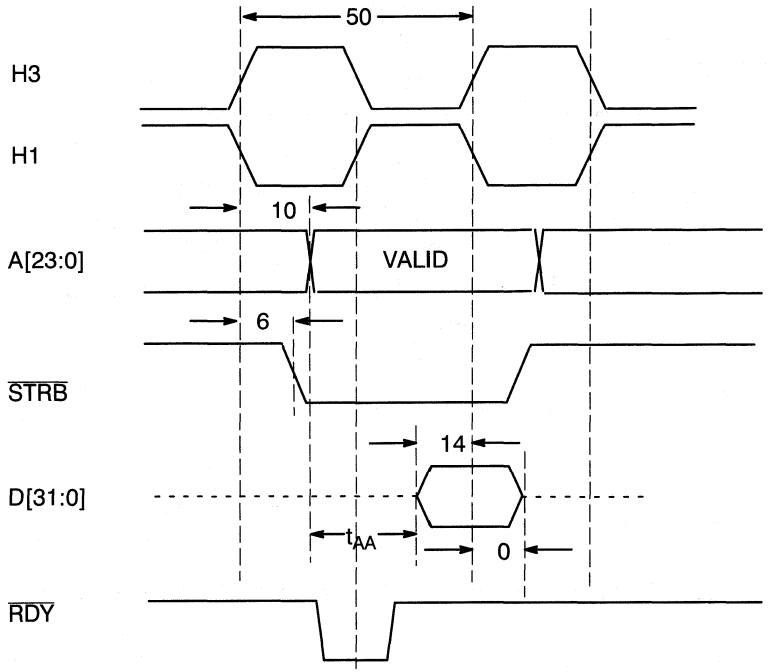


Figure 16. Memory Read Timing for the TI TMS320C30 Processor

D[31:0]: Data Bus, transfers data to and from processor.

$\overline{\text{STRB}}$: Strobe, external access strobe to the primary bus interface.

$\overline{\text{RDY}}$: Ready Signal, indicates that the external device is ready for a primary bus interface transaction to complete.

Interface Considerations

Because the CY7C270 is not performing burst reads and is operating in latch mode, it does not need a clock input. Hence the CLK pin on the CY7C270 is tied HIGH. The $\overline{\text{STRB}}$ signal is fed into the $\overline{\text{LE}}$ pin of the CY7C270. The $\overline{\text{ADV}}$ pin on the PROM is tied HIGH, since the CPU does not perform burst accesses. Generations of the $\overline{\text{RDY}}$ signal can be incorporated into the address decoder PLD. Note that this signal is also dependent on the other memory and I/O components in the system. Finally, the chip selects of the PROM are generated from the address decoder PLD.

CY7C270 Configuration

Output Enable: Active LOW
Counter Configuration: Don't care
Mode: Latched
Chip Selects: User defined
Burst Enable: No burst

Timing Requirement

PROM access time required on each access:

t_{AA} = 1 clock period – address valid delay – i/p data set-up

t_{AA} = 50 – 10 – 14 = 26 ns

It can be seen that the CY7C270 interfaces to the TMS320C30-40 processor with zero wait states. In this case, the CY7C270-15, with an initial access time of 25 ns can be used. *Figure 17* shows the CPU-PROM interface.

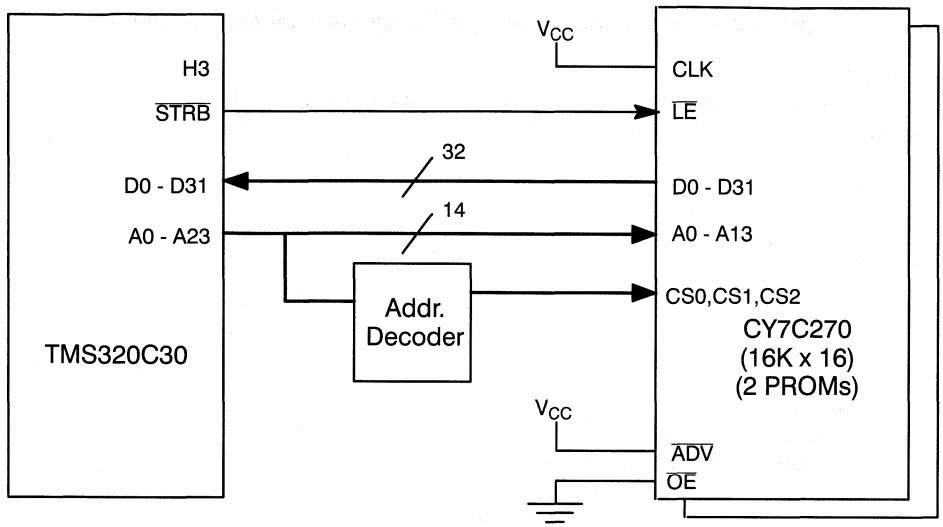


Figure 17. Interfacing the CY7C270 to the TI TMS320C30 Processor

Conclusion

This application note describes methods of interfacing the CY7C270 Processor-Intelligent PROM to various CISC and RISC processors and DSP devices. The CY7C270 can interface to many processors with zero wait states and no glue logic, offering embedded system designers a high-performance, highly integrated, cost-effective solution to their non-volatile memory needs.

Appendix A. Generating $\overline{LE}$ and $\overline{IRDY}$ Using ABEL for the Am29000

```
Module pip29K;
Title 'Generating LE and IRDY to interface PIP to AMD 29K
       Cypress Semiconductor'

Declarations

  Clk       pin;
  Q1,Q0     pin;
  ireq      pin;
  ibreq     pin;
  le        pin;
  !irdy     pin;
  reset     pin ;
  Count     = [Q1,Q0];
  Count     istype 'reg_D,buffer';

Equations

  Count.CLK = Clk;
  irdy.clk = Clk;
  Count := (Count.fb + 1) & !ireq & reset ;
  le = Q0.fb # Q1.fb # ireq # !reset ;
  irdy := (!ibreq # !le) & reset ;

End pip29K;
```

Document #: PRO0004–0192

Interfacing the CY7C276 High-Speed PROM to the AT&T, AD, Motorola, and TI DSPs

Introduction

Digital signal processors (DSPs) have typically required two external storage devices—a relatively slow PROM (Programmable Read Only Memory) for non-volatile code storage, and an SRAM (Static Random Access Memory), faster than the PROM, from which to run the code. The reason for this is that PROM access times are typically too slow to meet the requirements of the DSP cycle times.

The Cypress CY7C276 is a 16K x 16 UV-erasable PROM that can meet the fast cycle time requirements of a DSP design. It can help reduce component count and cost by eliminating the need for SRAM. If the goal is to eventually place the code in the internal Mask ROM (MROM) of the DSP, (which exists on the AT&T, Motorola and TI devices) then the CY7C276 can be utilized for prototyping until the code is completely debugged. This can save the expense of going to MROM prematurely.

This application note discusses how to use the CY7C276 PROM as program memory for various DSPs. It will cover the topic of interfacing the CY7C276 high-speed PROM to some of today's most popular DSPs for program memory only. Data memory storage is typically done with SRAM and its interface is not included in this application note. The AT&T DSP1616, Analog Devices ADSP 2100A, Motorola DSP56000 and TI TMS320C5x family of devices are discussed. Also included is a detailed description of the CY7C276 (including architecture, programming options, and signal descriptions) and brief descriptions of the DSPs (architectures, signals and timing requirements). For ease of explanation, only one example from each product family is included. The other devices in each product family are similar and are left as an exercise for the reader. Detailed timing calculations that show code sizes up to 16K words in depth are included in the examples. Finally, a table is provided to help summarize the analysis.

An Introduction to the CY7C276

The CY7C276 is a 16K x 16 asynchronous UV-erasable PROM with an access time of 25 ns. There are three polarity-programmable chip selects (CS[2:0]), which provide on-chip decoding of up to eight banks of PROM for a total of 256 Kbytes of PROM. The polarity of the asynchronous output enable (OE) pin is also user programmable. The CY7C276 provides a 16-bit-wide output, thus halving the number of PROMs required when interfacing to 16-bit or wider DSPs. With an access time of 25 ns, the CY7C276 can be used in 40-MHz DSP systems with zero-wait-states.

In all following examples, the CY7C276 is programmed to have all three chip enables (CS[2:0]) and the output enable (OE) active LOW. This is achieved by programming a Hex 0008 in location H4000.

AT&T – DSP1616

The DSP1616 is a 16-bit fixed-point DSP based on the popular DSP1600 core. It is object code upward-compatible with the DSP16, 16A, 16C, and 1610 devices from AT&T.

The DSP1616 can run out of either the 12K 16-bit words of on-chip MROM or external memory of up to 56K 16-bit words. Data memory space can also be accessed externally, although this application note only describes the external program memory interface. *Table 1* shows memory maps for the various configurations of DSP1616. IROM is Internal ROM, EROM is External ROM and RAM1/RAM2 are internal memory locations that are not utilized in this design and are therefore left as "don't cares." Two parameters, LOWPR and EXM, determine which memory map is used. LOWPR controls the address in memory assigned to the RAM1 and RAM2 areas. EXM (EXternal Memory) is an input signal that determines whether the internal ROM (IROM) or the external ROM (EROM) will be addressed in the memory map at location 0.

Initialization

The DSP must be reset after power-up to begin executing code. Reset is administered by asserting the RSTB (Reset Bar) pin LOW. When the RSTB pin is driven HIGH, the DSP1616 comes out of reset and fetches an instruction from location zero of the program space. The physical location of address zero is determined by which memory map is selected. The DSP1616 is configured for external ROM by holding the EXM (External ROM enable) signal HIGH at the rising edge of RSTB. With the

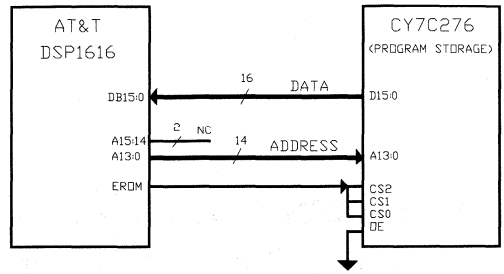


Figure 1. AT&T DSP1616 to PROM Interface

LOWPR bit set to 0 at reset, the corresponding memory map selected is Map2 (*Table 1*). This locates EROM at address 0.

DSP to Memory Interface

The DSP1616 uses the following signals to interface with external memory.

AB[15:0] – Address Bus. Outputs memory and I/O addresses.

DB[15:0] – Data Bus. Used to transfer data to and from the processor.

$\overline{\text{EROM}}$ – (Program address External ROM enable) Access enable for external memory. Active LOW.

The implementation of the DSP to PROM interface is shown in *Figure 1*.

Table 1. DSP1616 Memory Maps

Decimal Address	MAP1 EXM = 0 LOWPR = 0	MAP2 EXM = 1 LOWPR = 0	MAP3 EXM = 0 LOWPR = 1	MAP4 EXM = 1 LOWPR = 1
0	IROM	EROM	RAM1	RAM1
1K			RAM2	RAM2
2K			Reserved	Reserved
8K			IROM	EROM
12K	RAM1	RAM1		
13K	RAM2	RAM2		
14K	Reserved	Reserved		
20K 64K−1	EROM	EROM	EROM	

Timing Notes

The DSP1616 has numerous mask-programmable options for the clock source. It can use a crystal oscillator, or a small signal (TTL, or CMOS level) oscillator. The external source can be supplied by either a crystal or an oscillator. These options are discussed in detail in the AT&T DSP1616 datasheet. The DSP1616 can run at either the same or half the input frequency. The datasheet calls this a 1x or 2x clock, referring to the ratio of the input clock to the processor (internal) clock. *Table 4* notes this as the 1x and 2x clock options respectively.

The calculations for the required access time of the CY7C276 are illustrated below. The timing diagram for this example is shown in *Figure 2*.

$t_{AA}(PROM) = (t_{c(CKO)} - 2) - t_{ASKW(DSP)} - t_{SU(D)R}$

where:

$t_{AA}(PROM)$ = PROM address access time required,

$t_{c(CKO)}$ = DSP CKO cycle time [$t_{c(CKO)} - 2$ is to compensate for the worst-case EROM cycle time, which would be 2 ns shorter than CKO, as specified in the DSP1616 datasheet]. CKO is the clock out signal from the DSP,

$t_{ASKW}(DSP)$ = Worst-case address valid time from edge of cycle, and

$t_{SU(D)R}$ = Read data set-up time required by DSP before EROM goes HIGH.

Substituting values from a DSP1616 datasheet gives us:

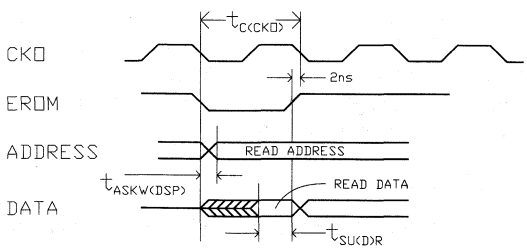


Figure 2. DSP1616 External Program Memory Timing

$t_{AA}(PROM) = (50 \text{ ns} - 2) - 2 \text{ ns} - 17 \text{ ns} = 29 \text{ ns}.$

for a 40-MHz DSP with the 2x input clock option.

Summary

From the above analysis, it can be seen that AT&T's DSP1616 can run code directly out of a CY7C276 at 40 MHz with the 2x input clock option with zero wait states. The CY7C276–25 (25-ns access time) can be used to satisfy this requirement.

Analog Devices – ADSP2100A

The ADSP2100A is a 24-bit fixed-point DSP that utilizes 24-bit instructions. It has a 14-bit address bus that directly addresses 16K 24-bit words externally and is expandable to 32K 24-bit words by using the program memory data access (PDMA) signal as a chip select. This example discusses the memory interface for 16K 24-bit words of program size. There is no on-chip memory for code/data storage. The ADSP2100A is the fastest available 24-bit DSP from Analog Devices that can access external ROM. (As of writing this application note.) It has a maximum clock frequency of 12.5 MHz.

Initialization

The DSP must be reset after power-up to begin executing code. Reset is administered by driving the $\overline{\text{RESET}}$ pin LOW. When the $\overline{\text{RESET}}$ pin is driven HIGH, the ADSP2100A comes out of reset and fetches an instruction from location H0004 of the program space (The first three locations of memory contain the interrupt vector addresses). In the case of the ADSP2100A, where there is no internal memory, the address (H0004) appears on the program memory address (PMA) bus followed by assertion of the program memory select ($\overline{\text{PMS}}$) and program memory read ($\overline{\text{PMRD}}$) signals. The DSP has completed the reset sequence and is now ready to execute code.

DSP to Memory Interface

The ADSP2100A uses the following signals to interface with external memory.

PMA[13:0] – Address Bus. Outputs memory and I/O addresses.

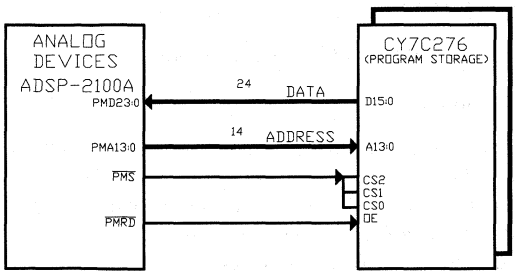


Figure 3. ADSP2100A to PROM Interface

PMD[23:0] – Data Bus. Used to transfer data to and from the processor.

$\overline{\text{PMS}}$ – Program Memory Select. Used to access external memory.

$\overline{\text{PMRD}}$ – Program Memory Read. Used for external memory output enable.

The implementation of the DSP to PROM interface is shown in *Figure 3*.

Timing Notes

The ADSP2100A datasheet directly specifies the maximum allowable access times to run code directly out of PROM. The timing diagram for this example is shown in *Figure 4*.

The result with the DSP running at a frequency of 12.5 MHz is:

t_{AA}(PROM)max = 32 ns.

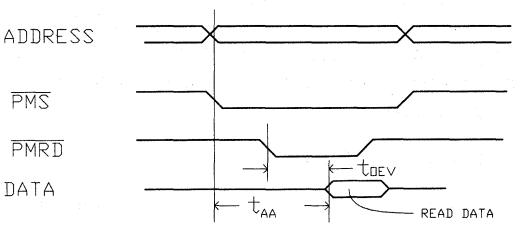


Figure 4. ADSP2100A External Program Memory Timing

where:

t_{AA}(PROM) = PROM address access time required. This is specified in the datasheet as PMA valid to PMD input valid.

Because this device uses the $\overline{\text{PMRD}}$ signal to control the OE of the CY7C276, the OE to data valid time must also be calculated. The following was also directly specified in the datasheet as $\overline{\text{PMRD}}$ LOW to PMD input valid.

t_{OEV}(PROM)max = 18 ns.

The CY7C276–30 satisfies both of these timing requirements.

Summary

From the above analysis, it can be seen that Analog Devices' ADSP2100A can run code directly out of two CY7C276s with zero wait states at its maximum frequency of 12.5 MHz. The CY7C276–25 or CY7C276–30 (25-ns and 30-ns access times, respectively) can be used to satisfy this requirement.

Motorola – DSP56000

The DSP56000 is a 24-bit general purpose DSP. It has 3.75K x 24-bits of on-chip ROM and can also run from external memory of 64K 24-bit words of program space. *Table 2* shows the memory maps for the various configurations of DSP56000. Mode 0 is the single-chip mode for use with internal ROM only. Mode 1 on the DSP56000 is for test purposes only and should not be invoked. Mode 2 is the normal expanded mode and is identical to Mode 0 except that the reset vector is in a different location. Mode 3 is Development Mode, which disables the internal ROM. All references to program memory space in this mode are directed to external program memory. There are two pins (MODA and MODB) that are sampled at the end of reset to determine which memory map is used.

Initialization

The DSP must be reset after power-up to begin executing code. Reset is administered by driving the $\overline{\text{RESET}}$ pin LOW. When the $\overline{\text{RESET}}$ pin is driven HIGH, the DSP56000 comes out of reset and

fetches an instruction from the reset vector location of the program space. The physical location of the reset vector is determined by which memory map is selected. If MODE 0 or 3 is selected, the reset vector is at location H0000 (location 0). If MODE 2 is selected, the reset vector is at location HE000. The DSP56000 is configured for external ROM by setting MODA and MODB HIGH at the rising edge of $\overline{\text{RESET}}$. The state of MODA and MODB will determine which memory map is selected for use (see *Table 2*). This example will use MODE 3 (Development Mode).

So, for MODE3, DSP56000 will start executing code from location zero of the external memory after reset is complete. All 64K words of external memory, except the first 64 locations (used for interrupts), are available for program storage. Location H0000 contains the reset vector which holds the programs starting address. This example will use two CY7C276 16K x 16 PROMs to achieve the 24-bit-wide program memory bus that is required. The upper eight bits of the PROM will not be used.

DSP to Memory Interface

The DSP56000 uses the following signals to interface with external memory.

A[13:0] – Address Bus. Outputs memory and I/O addresses.

D[23:0] – Data Bus. Used to transfer data to and from the processor.

$\overline{\text{PS}}$ – Program Memory Select. Used to access external memory.

$\overline{\text{RD}}$ – Read Select. Used for external memory output enable.

The implementation of the DSP to PROM interface is shown in *Figure 5*.

Timing Notes

The DSP56000 takes two external clock cycles for a read operation. The address becomes available at about the midpoint of the first cycle referenced from the falling edge of clock. The data is read into the DSP at the middle of the second cycle or the second rising edge of clock (see *Figure 6*). This actually works to the PROM's advantage by lengthening the required access time.

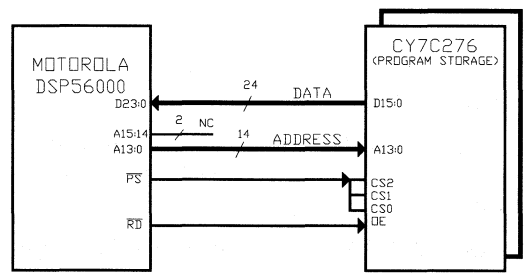


Figure 5. Motorola DSP56000 to PROM Interface

Table 2. DSP56000 Memory Maps

Decimal Address	MODE 0 MB = 0 MA = 0 Internal ROM Internal Reset	MODE 1* MB = 0 MA = 1 TEST MODE DO NOT USE	MODE 2 MB = 1 MA = 0 Internal ROM External Reset	MODE 3 MB = 1 MA = 1 No Int. ROM External Reset
0	Reset			Reset
3839 (H0EFF)	INTERNAL ROM		INTERNAL ROM	
3840	EXTERNAL		EXTERNAL	EXTERNAL
60K			Reset	
64K−1				

* Mode 1 is for test purposes only on the DSP56000 and should not be invoked by the user.

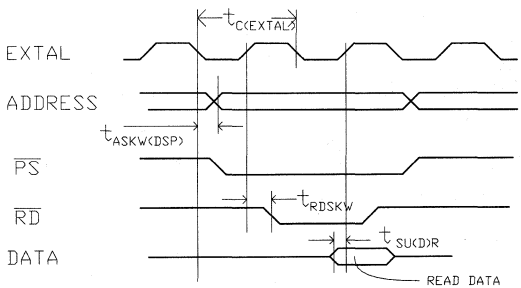


Figure 6. Motorola DSP56000 External Program Memory Timing

The calculations for the required access time of the CY7C276 are shown below. The timing diagram for this example is shown in *Figure 6*.

$t_{AA}(PROM)max = t_{c+} - t_{ASKW}(DSP) - t_{SU(D)R}$

where:

$t_{AA}(PROM)$ = PROM address access time required,

t_{c+} = DSP CLOCK IN cycle time x 1.5 (due to data read into DSP occurring at midpoint of second external clock cycle),

$t_{ASKW}(DSP)$ = Worst case address valid from edge of cycle, and

$t_{SU(D)R}$ = Read data set-up time before end of cycle.

This device uses the $\overline{RD}$ signal to control the OE of the CY7C276. Therefore, the OE access time must also be calculated. The following equation is for calculating the maximum allowable OE time of the CY7C276.

$t_{OEV}(PROM)max = t_{c(EXTAL)} - t_{RDSKW} - t_{SU(D)R}$

where:

$t_{c(EXTAL)}$ = clock cycle,

$t_{RDSKW}(DSP)$ = Worst case $\overline{RD}$ valid from edge of cycle, and

$t_{SU(D)R}$ = Read data set-up time before end of cycle.

The result with the DSP running at a frequency of 33 MHz is:

$t_{AA}(PROM)max = (30 \times 1.5)\ ns - 19\ ns - 0\ ns$
$= 45\ ns - 19\ ns$
$= 26\ ns.$

$t_{OEV}(PROM)max = 30ns - 16ns - 0ns = 14ns.$

Summary

Based on the above analysis, it can be seen that Motorola's DSP56000 can run code directly out of a CY7C276 with zero-wait-states at its maximum frequency of 33 MHz. The CY7C276–25 (25-ns access time) can be used to satisfy this requirement.

Texas Instruments – TMS320C5X

The TMS320C5X devices are a family of 16-bit fixed-point DSPs based on the older TMS320C25 CPU core. Significant modifications were added to improve performance. These devices are capable of running at twice the speed of the 'C2x family and are source-code upward compatible with all previous fixed-point DSPs from TI.

There are three devices in the 'C5x family, the 'C50, 'C51 and 'C53. They have 2K, 8K and 16K words of on-chip ROM, respectively. All three devices can run out of external ROM of up to 64K 16-bit words. All 64K words of external memory are available for program storage. *Table 3* shows the memory map for the TMS320C50 as an example. The SARAM is 9K words of program/data Single-Access RAM. This is memory that can only be read or written in a single machine cycle. It can reside on-chip or externally depending on the setting of the RAM bit in the PMST register. The DARAM is 1056 words of Dual-Access data RAM. It can be read from or written to in the same cycle. The DARAM can reside on-chip or externally depending on the setting of the CNF bit on the PMST register. The map in *Table 3* gives an example of the use of these two sections of memory.

Initialization

The DSP must be reset after power-up to begin executing code. Reset is administered by asserting the $\overline{RS}$ pin LOW. When the $\overline{RS}$ pin is driven HIGH,

the TMS320C5x comes out of reset and fetches an instruction from location 0. Location 0 (either on-chip or externally) contains the reset vector. The TMS320C5x is configured for external memory by holding the MP/$\overline{MC}$ pin HIGH at the rising edge of $\overline{RS}$.

Table 3. TMS320C50 Memory Maps

Decimal Address	'C50 MAP 1 MP/$\overline{MC}$ = 1 Internal	'C50 MAP 2 MP/$\overline{MC}$ = 0 External
0	Interrupts & Reserved (On-Chip)	Interrupts & Reserved (External)
48	On-Chip ROM	External
2K	On-Chip SARAM (RAM=1)	SARAM (RAM=0)
11K	External	External
63.5K 64K−1	On-Chip DARAM B0 (CNF=1)	DARAM External (CNF=0)

DSP to Memory Interface

The TMS320C5x uses the following signals to interface with external memory.

A[15:0] – Address Bus. Outputs memory and I/O addresses.

D[15:0] – Data Bus. Used to transfer data to and from the processor.

$\overline{PS}$ – Program Memory Select. Used to access external memory.

$\overline{RD}$ – Read Select. Used for external memory output enable.

The implementation of the DSP to PROM interface is shown in *Figure 7*.

Timing Notes

The TMS320C5x can use either its internal oscillator or an external frequency source for a clock. The external source can either be a crystal or an oscillator. These options are discussed in detail in the TI User's Guide for the TMS320C5x. The TMS320C5x can run at either the same or half the input frequency. This means that the internal machine cycle and, subsequently, external accesses, can cycle at the same or one-half times the external frequency. The table at the end of this document notes this as the ÷1 and ÷2 clock options respectively.

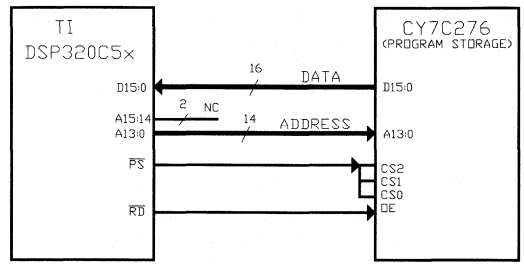


Figure 7. TMS320C5x to PROM Interface

The calculations for the required access time of the CY7C276 are illustrated below. The timing diagram for this example is shown in *Figure 8*.

$t_{AA}(PROM) = t_{c(CO)} - t_{ASKW}(DSP) - t_{SU(D)R}$

where:

$t_{AA}(PROM)$ = PROM address access time,

$t_{c(CO)}$ = DSP CLKOUT1 cycle time,

$t_{ASKW}(DSP)$ = Worst case address valid from edge of cycle, and

$t_{SU(D)R}$ = Read data set-up time before $\overline{RD}$ goes HIGH.

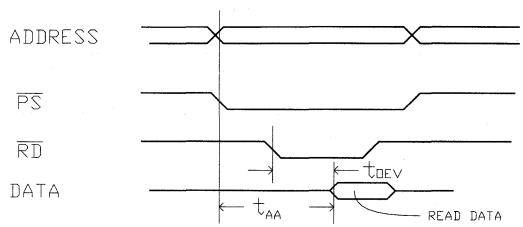


Figure 8. TMS320C5x External Program Memory Timing

The result with a frequency of 40 MHz and using the divide-by-two clock option the result is:

t_{AA}('276)max = 48.8 ns − 2 ns − 10 ns = 36.8 ns.

Summary

Based on the above analysis, it can be seen that TI's TMS320C5x series of DSPs can run code directly out of a CY7C276 at 40 MHz with zero-wait-states. The CY7C276−25 or CY7C276−30 (25-ns and 30-ns access times, respectively) can be used to satisfy this requirement.

Table 4 below has been provided to give a quick synopsis of the processors covered in this application note. It provides a quick cross reference of the CY7C276 PROM access times to DSP clock speeds and the number of wait states required.

Summary

These examples show how effectively the CY7C276 PROM can be used for executing code in DSP applications. The need for more costly SRAM is eliminated. There is no additional logic required to interface the PROM to the DSP thereby reducing pin count and cost in the design. As *Table 4* illustrates, most of the DSP speed grades can run code directly out of the CY7C276 with zero-wait-states.

Table 4. Wait State Requirements

DSP PART NUMBER	DSP Frequency and Clock Option If Applicable	# of Wait States Required for each PROM		
		CY7C276−25	CY7C276−30	CY7C276−35
AT&T DSP1616	20 MHz w/1x clock	0	1	1
	40 MHz w/2x clock	0	1	1
	20 MHz w/2x clock	0	0	0
ADSP2100A	12.5 MHz	0	0	1
	10.24 MHz	0	0	0
DSP56000	33 MHz	0	1	1
	27 MHz	0	0	0
	20.5 MHz	0	0	0
TMS320C5X	57 MHz with ÷1 clock option	2	3	3
	40 MHz with ÷1 clock option	1	2	2
	57 MHz with ÷2 clock option	1	1	1
	40 MHz with ÷2 clock option	0	0	1

Document #: PRO0001−1293

Designing Custom ALUs and Multipliers with PROMs

Have your designs ever called for an ALU that would subtract one number from another and add five? Or a comparator that, instead of just telling which number was larger, presented the larger number at the outputs? Or even a multiplier that, after multiplying, subtracted a constant from the result? The trouble is, ALUs and multipliers can only add, subtract, do some logic, or multiply. High-density PLDs are not a solution because of the high number of product terms necessary to do arithmetic operations in a single pass. When you need a high-speed, dense, custom solution, it's time to use a PROM. Depending upon the size of the coefficients, design speeds can be as high as 100 MHz!

Lets take a look at the first example. To implement the first function discussed above (subtract and add five) in discrete logic using 8-bit coefficients requires the components shown in *Figure 1*. The maximum frequency of this system is determined by counting the propagation delays through each ALU (t_{PDALU}), adding them to the clock to out (t_{CO}) of the register providing the coefficients, adding the set up time (t_{SU}) of the register receiving the result, and inverting it. Using F logic, $t_{PDALU} = \sim 20$ ns, $t_{CO} = \sim 10$ ns, and $t_{SU} = \sim 2$ ns.

$t_{CO} + 2(t_{PDALU}) + t_{SU}$= cycle time

10 ns + 40 ns + 2 ns = 52 ns

1/52 ns = 1/cycle time = f_{max} = 19 MHz

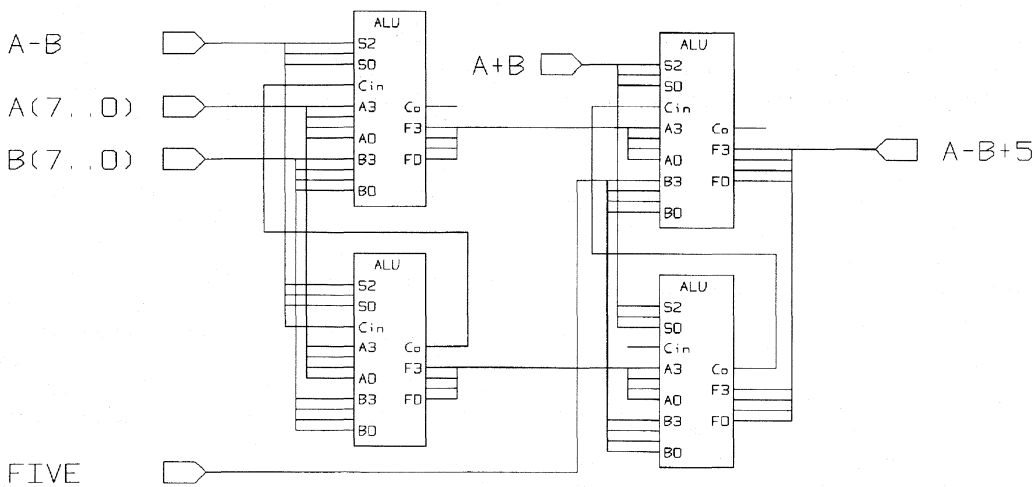


Figure 1. First Example of a Discrete Solution

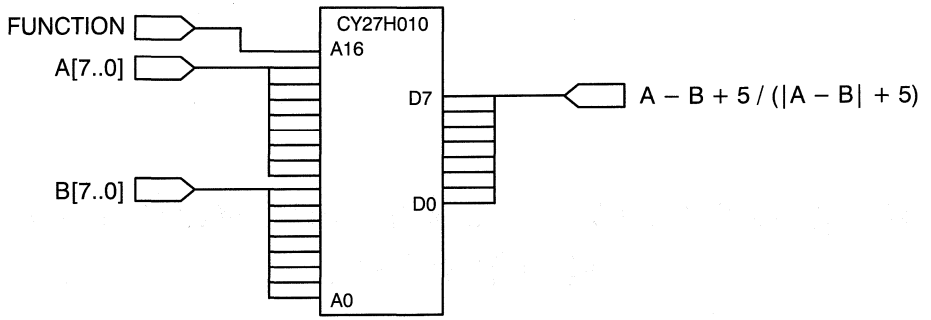


Figure 2. First Example of a PROM Implementation

If the design called for the absolute value of the subtraction, |a−b| + 5, the size and complexity of the design becomes quite unwieldy and complex in discrete logic. Using a high-speed PROM reduces the problem to child's play. The program in Appendix A illustrates the simplicity of fitting this design into a CY27H010. In addition, one address bit (*Figure 2*) on the PROM is used to determine if the equation being implemented is a−b + 5, or |a−b| + 5. The maximum frequency of this design is

$t_{CO} + t_{PDPROM} + t_{SU}$ = cycle time

10 ns + 25 ns + 2 ns = 37 ns

1/37 ns = f_{max} = 27 MHz

In the second example, the design calls for a comparator that produces the larger of the two values on the following clock cycle. Implementing this discretely for 7-bit coefficients is shown in *Figure 3*.

The maximum frequency of this solution is the inverse of the t_{CO} of the register supplying the inputs, plus the propagation delay through the comparators (t_{PDCOMP}) plus the delay through the muxes (t_{PDMUX}), plus the t_{SU} of the receiving components.

$t_{CO} + 2t_{PDCOMP} + t_{PDMUX} + t_{SU} = 1/f_{MAX}$; f_{MAX} = 20 MHz.

Sometimes an algorithm also needs to know how much larger one coefficient is over the other (image-processing applications). This would make discrete implementation quite cumbersome and slow (ALUs would be needed). Once again the PROM solution makes this all possible in a faster, more dense, and cost-effective solution. All the above-mentioned features can be easily implemented in a Cypress CY7C276 PROM (see *Figure 4*). A simple program (Appendix B) is used to "write the equations."

The maximum frequency of this design is simply the inverse of the t_{CO} of the register feeding the data plus the access time of the CY7C276.

$t_{CO} + t_{PDPROM} = 1/f_{MAX}$

2 ns + 25 ns = cycle time = 27 ns; 1/27ns = 37 MHz

If the design had only 5-bit coefficients, the maximum frequency of the discrete implementation would remain the same, while the PROM solution would accelerate to 100 MHz in a Cypress CY7C258! In addition the registers at the start of the timing chain would be not be needed do to the CY7C258 having them built in.

The final example calls for a multiplier that can subtract a constant from its final result, all in one clock cycle. To implement this discretely for 7-bit coefficients requires a 7-bit multiplier followed by a 16-bit ALU. If the absolute value of this operation (|A * B − C|) is required, the part count for the operation soar as the frequency of operation plummets. The entire solution can easily be placed in a single Cypress CY7C276. The program for this implementation is available in Appendix C.

The number of functions a PROM solution can perform is only limited by the number of inputs and out-

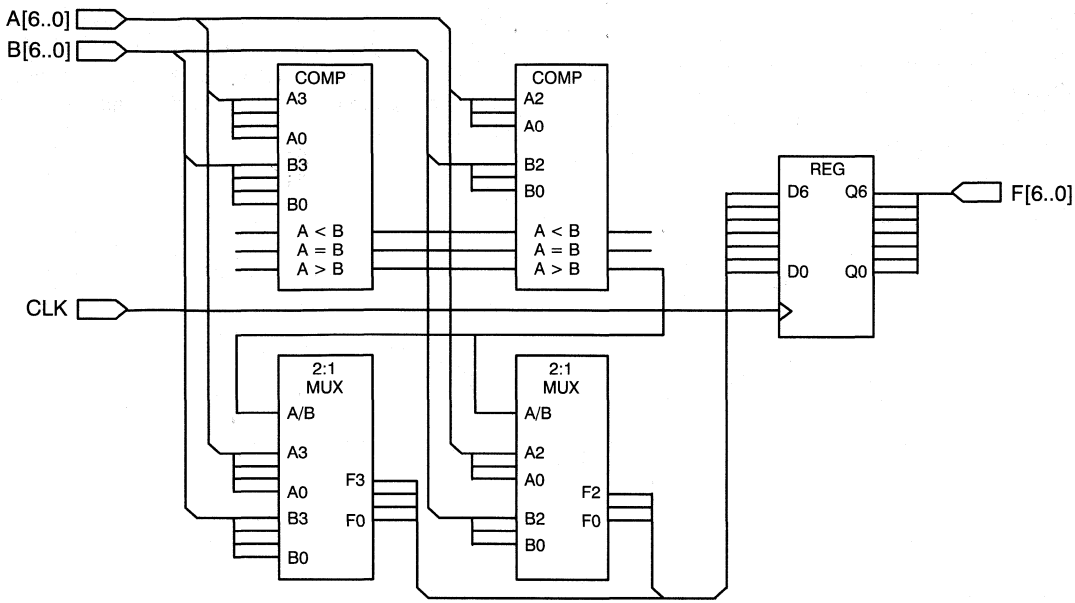


Figure 3. Second Example of a Discrete Solution

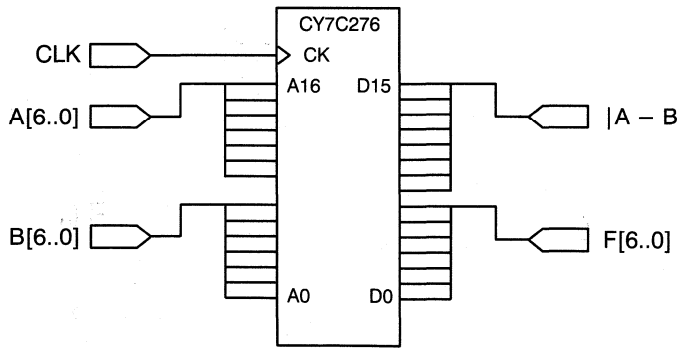


Figure 4. Second Example of a PROM Implementation

puts available on the PROM. A 1-MB PROM with seventeen address lines and eight data (CY27H010) lines can perform 512 different operations on 4-bit coefficients, 128 on 5-bit coefficients, and thirty-two on 6-bit coefficients! Unlike ALUs and multiplexers, if one coefficient is larger than the other, no input pins are left wasted. As illustrated above, with the new wider output PROMs (CY7C259 and CY7C276 with x16 outputs) multiple functions can be performed at the same time!

Conclusion

PROMs can be used in a multitude of applications where discrete parts can be numerous and slow. The PROM approach can increase system reliability by reducing part counts and signal routing problems. With the advent of high-speed, dense PROMs, the number of application opportunities for PROMs continues to increase. Remember, when your high-speed application requires a special function, use a PROM.

Appendix A. Source Code for *Figure 2*

```
10 ' ---------- APPENDIX A ----------
20 ' This program is written for Example 1.
30 ' The function that will be performed is determined by
40 ' the polarity of address bit A16.  If this bit is high
50 ' the function that will be performed is |a-b|+5.  If
60 ' it is low, the function performed is a-b+5.
70 ' Please note that in the overflow condition, the MSB is
80 ' truncated.
90 '
100 'A0 - LSB of a, A7 - MSB of a, A8 - LSB of b, A15 - MSB of b.
110 'D0 - LSB of f, D7 - MSB of f.
120 '
130 OPEN "O",#1,"PROMFILE.HEX"   'Open the file for output.
140 ' Data is written to a file in the HEX SPACE format
150 '
160 PRINT#1,CHR$(2)      'start the file with the STX char
170 FOR B = 0 TO (2^8-1)
180   FOR A = 0 TO (2^8-1)
190     F = A-B+5
200     IF F > 255 THEN F = F-256
210     IF F > 15 THEN 250
220     IF F < 0 THEN PRINT#1,RIGHT$(HEX$(F),2);" ";: GOTO 260
230     PRINT#1,"0";HEX$(F);" ";
240     GOTO 260
250     PRINT#1,HEX$(F);" ";
260   NEXT A
270 NEXT B
280 ' change the function to |a-b|+5
290 FOR B = 0 TO (2^8-1)
300   FOR A = 0 TO (2^8-1)
310     F = ABS(A-B)+5
320     IF F > 255 THEN F = F-256
330     IF F > 15 THEN 360
340     PRINT#1,"0";HEX$(F);" ";
350     GOTO 370
360     PRINT#1,HEX$(F);" ";
370   NEXT A
380 NEXT B
390 PRINT#1,CHR$(3);     ' end the file with the ETX char
400 CLOSE
410 END
```

Appendix B. Source Code for *Figure 4*

```
10 ' ---------- APPENDIX B ----------
20 ' This program is written for Example 2.
30 ' The function that will be performed is a comparator
40 ' that places the larger of the two numbers in the LSB
50 ' (D7..D0) of the output and the magnitude of the
60 ' difference in the MSB (D15..D8) of the output.
70 '
80 'A0 - LSB of a, A6 - MSB of a, A7 - LSB of b, A13 - MSB of b.
90 'D0 - LSB of f, D6 - MSB of f, D7 - LSB of d, D13 - MSB of d.
100 '
110 OPEN "O",#1,"EXAMPLE2.HEX"   'Open the file for output.
120 ' Data is written to a file in the HEX SPACE format
130 PRINT#1,CHR$(2)      'start the file with the STX char
140 FOR B = 0 TO (2^7-1)
150   FOR A = 0 TO (2^7-1)
160     D = ABS(A-B)
170     IF A-B > 0 THEN F = A ELSE F = B
180     IF D > 15 THEN 200
190     PRINT#1,"0";HEX$(D);:GOTO 210
200     PRINT#1,HEX$(D);
210     IF F > 15 THEN 230
220     PRINT#1,"0";HEX$(F);" ";:GOTO 240
230     PRINT#1,HEX$(F);" ";
240   NEXT A
250 NEXT B
260 PRINT#1,CHR$(3);     ' end the file with the ETX char
270 CLOSE
280 END
```

Appendix C. Source Code for Multiply w/Subtract of 7-Bit Coefficients

```
10 ' ---------- APPENDIX C ----------
20 ' This program is written for Example 3.
30 ' The function that will be performed is:
40 '                        |A x B - C|
50 ' Please note that since we are using a x16 EPROM (CY7C276),
60 ' there is no carry out function needed.
70 '
80 'A0 - LSB of a, A6 - MSB of a, A7 - LSB of b, A13 - MSB of b.
90 'D0 - LSB of f, D15 - MSB of f.
100 '
110 OPEN "O",#1,"EXAMPLE3.HEX"   'Open the file for output.
120 ' Data is written to a file in the HEX SPACE format
130 '
140 CONSTANT = 22 'Choose a value for the constant
150 '
160 PRINT#1,CHR$(2)      'start the file with the STX char
170 FOR B = 0 TO (2^7-1)
180   FOR A = 0 TO (2^7-1)
190     F = ABS(A*B-CONSTANT)
200     IF F < 16   THEN PRINT#1,"000";HEX$(F);" ";: GOTO 240
210     IF F < 256  THEN PRINT#1,"00";HEX$(F);" ";:  GOTO 240
220     IF F < 4096 THEN PRINT#1,"0";HEX$(F);" ";:   GOTO 240
230     PRINT#1,HEX$(F);" ";
240   NEXT A
250 NEXT B
260 PRINT#1,CHR$(3);     ' end the file with the ETX char
270 CLOSE
280 END
```

Document #: PRO0005–0992

UltraLogic/PLDs – 3



UltraLogic/PLD Section Contents and Abstracts

Are Your PLDs Metastable? **3–1**

This application note provides a detailed description of the metastable behavior in PLDs from both circuit and statistical viewpoints. Additionally, the information on the metastable characteristics of Cypress PLDs presented here can help achieve any desired degree of reliability.

State Machine Design Considerations and Methodologies **3–27**

This application note describes many of the options encountered during a state machine design cycle. The different methods of describing a state machine design are covered briefly. The different types of state machines are described. Most of the application note is a design example of a clock generator for a bit-slice processor. The last section shows the necessary steps to implement the clock generator in a CY7C361 device. The appendices include source code, reduced equations, pinouts, and simulation results.

Designing with the CY7C335 and *Warp2*™ VHDL Compiler **3–73**

This application note provides an overview of the CY7C335 Universal Synchronous EPLD architecture and *Warp2* VHDL Compiler for PLDs. Example designs demonstrate how the *Warp2* VHDL compiler takes advantage of the rich architectural features of the CY7C335.

The FLASH370™ Family of CPLDs and Designing with *Warp2* **3–102**

This application note introduces Cypress's high-density complex programmable logic device family of products. The innovative architectural features of this family are discussed relative to competitor implementations. Some simple VHDL examples are shown that demonstrate usage or the features of the architecture using VHDL hardware description language available from Cypress's design development tool called *Warp*™.

Implementing a Reframe Controller for the CY7B933 HOTLink™ Receiver in a CY7C371 CPLD ... **3–121**

This application note gives some criteria that can be used to determine when the CY7B933 HOTLink Receiver should be forced to reframe its data, and it describes in detail a specific design of a reframe controller that implements these criteria. The design is described in VHDL and is implemented in the 32-macrocell CY7C371 FLASH CPLD.

Architectures and Technologies for High-Performance FPGAs **3–137**

Key issues in FPGA architectures are identified and are related to the interconnect technology (the technology used to connect two wires under user programmability). Logic cell architecture, number of interconnects available, routability, and performance are related to SRAM based, large anti-fuse based, and ViaLink™ fused based interconnect technologies. The relationships are used to explain characteristics of certain device families using the various fuse technologies. Characteristics include fitting capability, internal propagation delays, and other factors of interest to FPGA users.

Designing with FPGAs **3–149**

This application note describes the design process for pASIC™ (FPGA). A design flow is described that takes the reader from an abstract VHDL description of the design all the way through design place and route, and simulation. The flow description includes discussions of design decomposition for optimization by going to

structural VHDL and schematic descriptions of the design. Performance analysis and optimization at the place and route step and at the simulation step are also presented. The path analyzer is presented along with its use in analysis and applying constraints. Operation of the simulation tool is presented including setting up of simulation stimulus descriptions and making timing measurements from the simulation result. An actual design is presented to clearly illustrate the concepts presented in the note.

CY7C380 Family Quick Power Calculator 3–172

Calculating power consumption for a pASIC device may be required prior the completion of the detailed design. This can be difficult without detailed knowledge of the number of logic cells used and the toggle rate for each of the cells. However, with a general knowledge of the percent of the device used and the average toggle rate for various sections of the design, the power can be easily estimated. This application note presents a quick power estimation procedure. A worksheet along with graphs for rapid estimation of worksheet power values is included. An example is also provided.

Using Scan Mode on pASIC380 for In-Circuit Testing 3–177

It is sometimes desirable to test devices when they are installed in the application circuit board (in-circuit testing). The pASIC380 family of FPGAs contains an internal scan path that can be used in in-circuit tests. This application note describes the scan path and how to shift data in and out of the internal registers. Interface electrical and timing specifications are given.

Getting Started Converting .ABL Files to VHDL 3–180

This application note is intended to assist *Warp* users in converting designs written in DATA I/O's ABEL hardware description language to IEEE 1076 VHDL. It contains several language cross reference tables and many helpful hints. It also includes two real-world designs that have been converted from MACH™ 210-ABEL descriptions to FLASH371-VHDL descriptions.

Top Down Design Methodology with VHDL 3–207

The purpose of this application note is two-fold. One of the goals is to design an interrupt controller for the CY7C611 using the power of VHDL and a 'top down design methodology'. The second purpose is to introduce several features of VHDL and to give examples of how to use them. The features used include packages, truth tables (the ttf[ ] function), loop, generate, bufoe, bit vectors, and x01z vectors.

Abel–HDL vs. IEEE–1076 VHDL 3–217

The purpose of this application note is to compare and contrast the complexity and basic features of Abel-HDL with those of IEEE–1076 VHDL. Both of these languages are very robust in their support of different types of constructs that can be used to describe the same functionality at different levels of abstraction. It is beyond the scope of this document to exhaustively describe these possibilities or to present a complete tutorial for writing code in either language because of the great variety of constructs and syntax available with which to describe the functionality of a given circuit. Rather, a simple sample design that contains a mixture of synchronous and asynchronous logic circuits will be shown. Sample code is written in both Abel–HDL and VHDL that describes the example's functionality and synthesizes to create functionally identical hardware. The code written here represents a typical level of abstraction that balances readability with compactness. With experience, designers can develop their own preferences for style. For instance, state machines can be described in a number of ways: state tables, IF-THEN-ELSE statements, CASE-WHEN statements, or explicitly using a combination of Register-Transfer-Level (RTL) code (individually describe each gate/register as a component with its inputs and outputs) and/or Boolean equations.

VHDL Techniques for Optimal Design Fitting **3–231**

There are three simple rules to follow when the *Warp2* software informs you that your VHDL design did not fit into the target device. First, complete all IF-THEN-ELSE chains with an ELSE statement. Second, complete all CASE statements with a WHEN OTHERS statement. Third, assign default values to all output signals at the beginning of each process. Following these rules will ensure the most optimal fit for your design.

Describing State Machines with *Warp2* VHDL **3–236**

Converting state machines from a concept to an implementation is a simple process using VHDL and the *Warp2* compiler. The basic types of state machines are described and the implementation of each of these types of state machines is shown. Through the use of multiple examples, designers are shown how to enter their state machines into VHDL and target them to any PLD or PROM.

Using Hierarchical VHDL Design **3–269**

This application note describes how to construct a hierarchical design using *Warp* VHDL. It first discusses the features of VHDL that are designed to simplify hierarchical design. The reader is then walked through a design sample that is modified to illustrate increasingly advanced topics.

Are Your PLDs Metastable?

This application note provides a detailed description of the metastable behavior in PLDs from both circuit and statistical viewpoints. Additionally, the information on the metastable characteristics of Cypress PLDs presented here can help you achieve any desired degree of reliability.

Metastable is a Greek word meaning "in between." Metastability is an undesirable output condition of digital logic storage elements caused by marginal triggering. This marginal triggering is usually caused by violating the storage elements' minimum set-up and hold times.

In most logic families, metastability is seen as a voltage level in the area between a logic HIGH and a logic LOW. Although systems have been designed that did not account for metastability, its effects have taken their toll on many of those systems.

In most digital systems, marginal triggering of storage elements does not occur. These systems are designed as synchronous systems that meet or exceed their components' worst-case specifications. Totally synchronous design is not possible for systems that impose no fixed relationship between input signals and the local system clock. This includes systems with asynchronous bus arbitration, telecommunications equipment, and most I/O interfaces. For these systems to function properly, it is necessary to synchronize the incoming asynchronous signals with the local system clock before using them.

Figure 1 shows a simple synchronizer, whose asynchronous input comes from outside the local system. The synchronizer operates with a system clock that is synchronous to the local system's operation. On each rising edge of this system clock, the synchronizer attempts to capture the state of the asynchronous input. *Figure 2* shows the expected result. Most of the time, this synchronizer performs as desired.

Digital systems are supposed to function properly all the time, however. But because there is no direct relationship between the asynchronous input and the system clock, at some point the two signals will both be in transition at very nearly the same instant. *Figure 3* shows some of the synchronizer's possible

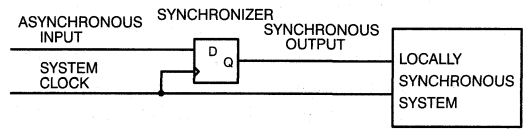


Figure 1. Simple Synchronizer

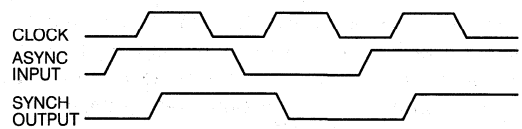


Figure 2. Expected Synchronizer Output

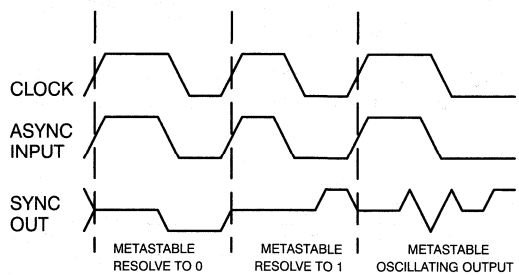


Figure 3. Possible Metastable States of Synchronizer

metastable outputs when this input condition occurs. These types of outputs would not occur if the synchronizer made a decision one way or the other in its specified clock-to-output time. A flip-flop, when not properly triggered, might not make a decision in this time. When improperly triggered into a metastable state, the output might later transition to a HIGH or a LOW or might oscillate.

When other components in the local system sample the synchronizer's metastable output, they might also become metastable. A potentially worse problem can occur if two or more components sample the metastable signal and yield different results. This situation can easily corrupt data or cause a system failure.

Such system failures are not a new problem. In 1952, Lubkin (Reference 1) stated that system designers, including the designers of the ENIAC, knew about metastability. The accepted solution at that time was to concatenate an additional flip-flop after the original synchronizer stage (*Figure 4*). This added flip-flop does not totally remove the problem but does improve reliability. This same solution is still in wide use today.

Recovery from metastability is probabilistic. In the improved synchronizer, the first flip-flop's output might still be in a metastable state at the end of the sample clock period. Because the flip-flops are sequential, the probability of propagating a metastable condition from the second flip-flop stage is the square of the probability of the first flip-flop remaining metastable for its sample clock period. This type of synchronizer does have the drawback of adding one clock cycle of latency, which might be unacceptable in some systems.

As system speeds increase and as more systems utilize inputs from asynchronous external sources, metastability-induced failures become an increasingly significant portion of the total possible system failures. So far, no known method totally eliminates the possibility of metastability. However, while you cannot eliminate metastability, you can employ design techniques that make its probability relatively small compared with other failure modes.

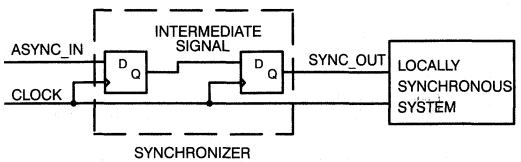


Figure 4. Two-Stage Synchronizer

Explanation of Metastability

In a flip-flop, a metastable output is undefined or oscillates between HIGH and LOW for an indefinite time due to marginal triggering of the circuit. This anomalous flip-flop behavior results when data inputs violate the specified set-up and hold times with respect to the clock.

In the case of a D-type flip-flop, the data must be stable at the device's D input before the clock edge by a time known as the set-up time, t_s. This data must remain stable after the clock edge by a time known as the hold time, t_h (*Figure 5*). The data signal must satisfy both the set-up and hold times to ensure that the storage device (register, flip-flop, latch) stores valid data and to ensure that the outputs present valid data after a maximum specified clock-to-output delay t_{co_max}. As used in this application note, t_{co_max} refers to the interval from the clock's rising edge to the time the data is valid on the outputs. In most cases, t_{co_max} refers to the maximum t_{co} specified by a datasheet, as opposed to the average or typical t_{co} value.

If the data violates either the set-up or hold specifications, the flip-flop output might go to an anomalous state for a time greater than t_{co_max} (*Figure 5*).

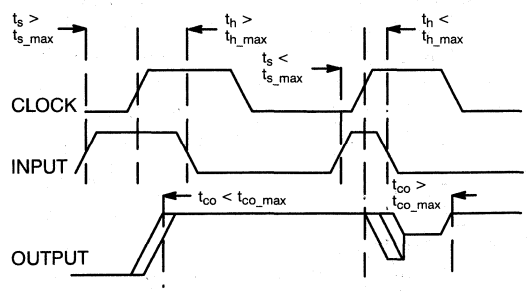


Figure 5. Triggering Modes of a Simple Flip-Flop

The additional time it takes the outputs to reach a valid level can range from a few hundred picoseconds to tens of microseconds. The amount of additional time beyond t_{co_max} required for the outputs to reach a valid logic level is known as the metastable walk-out time. This walk-out time, while statistically predictable, is not deterministic.

Figure 6, from Reference 2, shows the variation in output delay with data input time. The left portion of the graph shows that when the data meets the required set-up time, the device has valid output after a predictable delay, which equals t_{co}. The middle portion of the graph indicates the metastable region. If the data transitions in this region, valid output is delayed beyond t_{co_max}. The closer the input transitions to the center of the metastable region, violating the device's triggering requirements, the longer the propagation delay. If the data transitions after the metastable region, the device does not recognize the input at that clock edge, and no transition occurs at the output. As given in Reference 3, you can predict the region t_w, where data transitions cause a propagation delay longer than t, from the formula:

$$t_w = t_{co}\, e^{\frac{-(t - t_{co})}{\tau}} \qquad \text{Eq. 1}$$

where τ depends on device-specific characteristics such as transistor dimensions and the flip-flop's gain-bandwidth product.

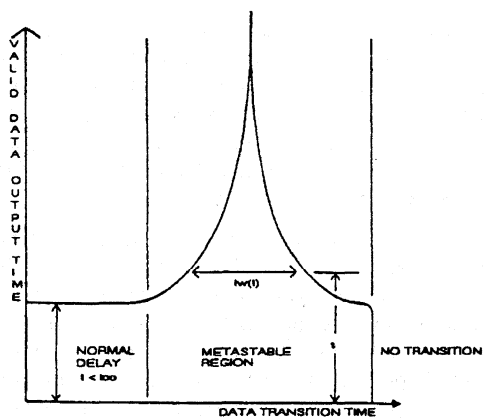


Figure 6. Output Propagation vs. Data Transition



Figure 7. Triggering Modes of a Simple Flip-Flop

Figure 7 shows another way of looking at metastability. A flip-flop, like any other bistable device, has two minimum-potential energy levels, separated by a maximum-energy potential. A bistable system has stability at either of the two minimum-energy points. The system can also have temporary stability—metastability—at the energy maximum. If nothing pushes the system from the maximum-energy point, the system remains at this point indefinitely.

A hill with valleys on either side is another bistable system. A ball placed on top of the hill tends to roll toward one of the minimum-energy levels. If left undisturbed at the top, the ball can remain there for an indeterminate amount of time. As this figure indicates, the characteristics of the top of the hill as well as natural factors affect how long the ball stays there. The steepness of the hill is analogous to the gain-bandwidth product of the flip-flop's input stage.

Causes of Metastability

Systems with separate entities, each running at different clock rates, are called globally asynchronous systems (Reference 4). The entities might include keyboards, communication devices, disk drives, and processors. A system containing such entities is asynchronous because signals between two or more entities do not share a fixed relationship.

Metastability can occur between two concurrently operating digital systems that lack a common time reference. For example, in a multiprocessing system, it is possible that a request for data from one system can occur at nearly the exact moment that this signal is sampled by another part of the system. In this case, the request might be undefined if it does not obey the set-up and hold time of the requested system.

When globally asynchronous systems communicate with each other, their signals must be synchronized. Arbitration must occur when two or more requests

for a shared resource are received from asynchronous systems. An arbiter decides which of two events should be serviced first. A synchronizer, which is a type of arbiter with a clock as one of the arbited signals, must make its decision within a fixed amount of time. A device can synchronize an input signal from an external, asynchronous device in cases such as a keyboard input, an external interrupt, or a communication request.

Care must be taken when two locally synchronous systems communicate in a globally asynchronous environment. A synchronization failure occurs when one system samples a flip-flop in the other system that has an undefined or oscillating output. This event can distribute non-binary signals through a binary system (Reference 5).

In synchronizers, the circuit must decide the state of the data input at the clock input's rising edge. If these two signals arrive at the same time, the circuit can produce an output based on either decision, but must decide one way or the other within a fixed amount of time.

Attacking Metastability

The design of synchronous systems is much different than the design of globally asynchronous systems. The design of a synchronous digital system is based on known maximum propagation delays of flip-flops and logical gates. Asynchronous systems by definition have no fixed relationship with each other, and therefore, any propagation delay from one locally synchronous system to the next has no physical meaning.

Two different methods are available to produce locally synchronous systems from globally asynchronous systems. The first method involves creating self-timed systems. In a self-timed system, the entity that performs a task also emits a signal that indicates the task's completion. This handshaking signal allows the use of the results when they are ready instead of waiting for the worst-case delay. Such handshaking signals allow communications between locally synchronous systems.

The advantage of the self-timed method is that it permits machines to run at the average speed instead of the worst-case speed. The disadvantages are that a self-timed system must have extra circuitry to compute its own completion signals and extra circuitry to check for the completion of any tasks assigned to external entities.

Petri Nets, data flow machines, and self-timed modules all use the self-timed method of communication among locally synchronous systems. Self-timed structures do not completely eliminate metastability, however, because they can include arbiters that can be metastable. Most systems do not include self-timed interfaces due to the additional circuitry and complexity.

The second method of producing locally synchronous systems from globally asynchronous systems is the simple synchronizer. This is the most common way of communicating between asynchronous objects. The metastability errors that might arise from these systems must be made to play an insignificant role when compared with other causes of system failure.

Many metastability solutions involve special circuits (References 6 and 7). Some of these solutions do not reduce metastability at all (References 13 and 8). Others, however, do reduce metastability errors by pushing the occurrence of metastability to a place where sufficient time is available for resolving the error. Most of these circuits are system dependent and do not offer a universal solution to metastability errors.

The easiest and the most widely used solution is to give the synchronizing circuit enough time to both synchronize the signal and resolve any possible metastable event before other parts of the system sample the synchronized output. This solution requires knowledge of the metastable characteristics of the device performing the synchronization.

Many semiconductor companies have developed circuits such as arbiters, flip-flops, and latches that are specifically designed to reduce the occurrence of metastability. Although these parts might have good metastability characteristics, they have very limited application. The circuits can only function as flip-flops or arbiters and do not have the flexibility of PLDs. Cypress Semiconductor has designed the flip-flops in the company's PLDs to be metast-

ability hardened. This allows you to use Cypress PLDs in a wide range of systems requiring synchronization.

Circuit Analysis of Metastability

Many authors have written papers detailing the analysis of metastability from a circuit standpoint (References 5, 7, 8, 9, 10, 11, and 12). In Reference 11, for example, Kacprzak presents a detailed analysis of an RS flip-flop's metastable operation. He states that a flip-flop has two stages of metastable operation (*Figure 8*).

During the initialization phase, the Q and $\overline{Q}$ outputs move simultaneously from their existing levels to the metastable voltage V_m, which is the voltage at which $V_q = \overline{V}_q$.

The second or resolving phase occurs when the outputs once again drift toward stable voltages. Once a flip-flop has entered a metastable state, the device can stay there for an indeterminate length of time. The probability that the flip-flop will stay metastable for an unusually long period of time is zero, however, due to factors such as noise, temperature imbalance within the chip, transistor differences, and variance in input timing. During the second phase of metastability, for very small deviations around the metastable voltage, V_m, the flip-flop behaves like two cross-coupled linear amplifier stages that gain $V_d = V_q - \overline{V}_q$. When the gain of the cross-coupled loop exceeds unity, the differential voltage increases exponentially with time.

The length of time the flip-flop takes to resolve cannot be exactly determined. The probability that the flip-flop will resolve within a specific length of time, however, can be predicted. This probability depends on the electrical parameters of the flip-flop acting as a linear amplifier around the metastability voltage. The solution (Reference 11) to the differential voltage $V_d(t)$ driving the resolving phase is given by

$$V_d(t) = V_d(t_0)\, e^{\frac{(t - t_0)}{\tau}} \qquad \text{Eq. 2}$$

where τ depends directly on the amplifier gain and capacitance, and where $V_d(t_0)$ represents the differential voltage at some time t_0. You can use this equation to determine the length of time that the output voltage will take to drift from the metastable voltage V_m to a specified voltage difference V_d.

Horstmann (Reference 5) states that a flip-flop, like any other system with two stable states, can be described by an energy function with two local energy minima where P(x) = 0 (*Figure 9*). Any bistable system has at least one metastable state, which is an unstable energy level within the system and represents the local maximum of the energy function. The system's gradient can be represented by a force, F(x), that is zero at stable and metastable states (inflection points of the energy function).

Figure 10 shows a simplified first-order model of an RS flip-flop used to predict and visualize metast-

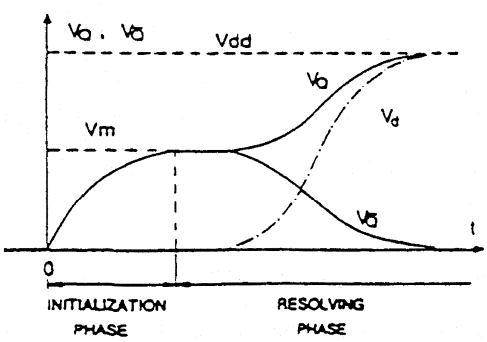


Figure 8. Two Phases of Metastability

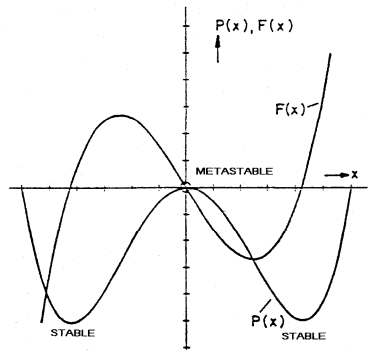


Figure 9. Energy/Force Function of a Bistable System

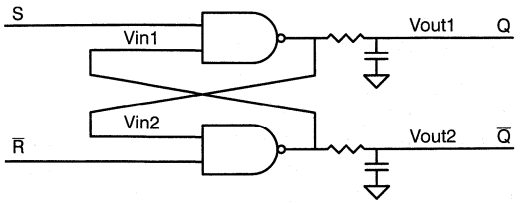


Figure 10. First-Order Flip-Flop Approximation

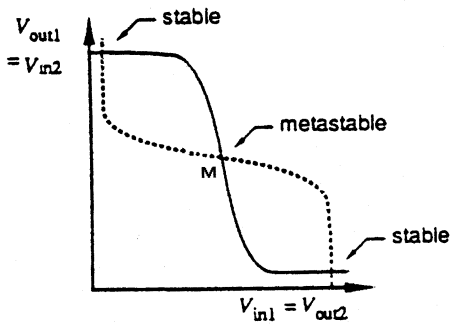


Figure 11. Energy Transfer Diagram of Simple RS Flip-Flop

ability. A flip-flop energy transfer curve (*Figure 11*) shows the relationship between the two outputs. The two stable states are local energy minima of the system. The metastable state, M, is a local energy maximum and represents an unstable state with loop gain near M that is greater than one.

Figure 12 shows the trigger line for the first-order approximation of the flip-flop. The dashed line RS represents the device's normal trigger line, which does not follow the transfer curve because, during triggering, the feedback loop has not been established. If at varying points along the trigger line the feedback loop is re-established, the nodes of the device follow the curves that lead to the line $S_0 - S_1$. Once on this line, the circuit exponentially drifts toward stability at either S_0 or S_1, depending on which side of the line $Q = \overline{Q}$ the feedback loop was re-established. The curves are solutions to the first-order model circuit equations for the device shown in *Figure 10*.

When the feedback loop is restored near the line $Q = \overline{Q}$, the system moves toward the unstable state M

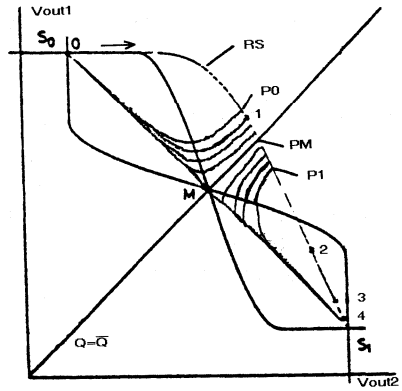


Figure 12. Energy Transfer Curves showing Trigger Paths

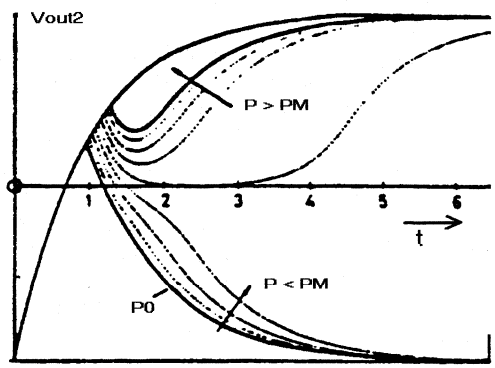


Figure 13. Time Scale Showing Trigger Paths

and can take an indefinite amount of time to exit from this metastable state. You can see this from the graph by noticing that S_0 and S_1 are equally likely solutions for system stability from M. Once the feedback loop is re-established, the system exponentially decays toward M and then exponentially grows toward S_0 or S_1.

Figure 13 shows the system's possible trigger events using the implied time scale of the state-space curves. The solution of these simplified first-order equations indicates that the fastest metastable resolution time occurs when the circuit's gain-bandwidth product is maximized.

Flannagan (Reference 12), in an attempt to maximize the gain-bandwidth product, solves simplified flip-flop equations to determine the phase trajectory near the metastable point. His results, which are supported by other authors, indicate that p and n devices with equal geometries produce the optimal gain-bandwidth product for metastable event resolution.

Statistical Analysis of Metastability

To begin the analysis of metastability, assume that the flip-flop's probability of resolving its metastable state does not depend on its previous metastable state. In other words, the metastable device has no memory of how long it has been in a metastable region. The analysis of metastability also assumes that the flip-flop's probability of resolving its metastable state in a given time interval does not depend on the metastable resolution in another disjoint time interval. The probability that a metastable event will resolve in a given interval (0,t) is only proportional to the length of the interval.

These assumptions yield an exponential distribution that describes the probability that the flip-flop resolves its metastability at a time t. The exponential distribution has the form

$$f(x) = \mu\, e^{-\mu t} \qquad \text{Eq. 3}$$

where μ is the expected value of metastability resolution per unit time (settling rate).

Using this equation and given that the flip-flop was metastable at time t = 0, the probability of a metastable event lasting a time t or longer is

$$P\ (met_t \mid met_{t=0}) = \int_t^{\infty} \mu\, e^{-\mu t}\, dt = e^{-\mu t} \qquad \text{Eq. 4}$$

The next part of the analysis involves the probability that the flip-flop is metastable at time t = 0. This part of the analysis assumes that the probability that the data transitions in a given time interval depends only on the length of the interval. A Poisson process with rate f_d describes the probability of the data transitioning at a time t:

$$p(x) = \frac{e^{-f_d t}\ (f_d\ t)^x}{x\ !} \qquad \text{Eq. 5}$$

where x is the number of transitions.

If a data transition within a bounded time interval, W, of the clock edge causes a metastable condition, the expected number of transitions of this Poisson process with rate f_d in time interval W is

$$E(X) = \sum_{x=0}^{\infty} \frac{x\ e^{-f_d W}\ (f_d\ W)^x}{x\ !} = f_d\ W \qquad \text{Eq. 6}$$

Because this expected number of transitions is the same as the probability that the flip-flop is metastable at t = 0, the equation for the probability at t = 0 is

$$P\ (met_{t=0}) = f_d\ W \qquad \text{Eq. 7}$$

Using *Equations 5* and *7*, the probability that a given clock cycle results in metastability that lasts at most a time t is

$$P\ (met_t) = P\ (met_t \mid met_{t=0})\ P\ (met_{t=0}) = f_d\ W\ e^{-\mu t} \qquad \text{Eq. 8}$$

Substituting $\frac{1}{t_{sw}}$ for μ allows this variable to be expressed as a settling time constant of the flip-flop. Further, a synchronization failure for a given clock cycle exists whenever a metastable event lasts a specified time (t_r) or longer. Using these two substitutions, the probability that the flip-flop is metastable in a given clock cycle is

$$P\ (fail_{1\ clock}) = f_d\ W\ e^{\frac{-t_r}{t_{sw}}} \qquad \text{Eq. 9}$$

Because the data transitions are independent, the number of failures in n clock cycles has a binomial distribution with an expected number of failures:

$$E\ (fail_{n\ cycles}) = n\ P\ (fail_{1\ cycle}) \qquad \text{Eq. 10}$$

Assuming a sample clock frequency, f_c, that represents the number of clock cycles, n, per unit time, the expected number of failures per unit time is

$$E\ (fail_{unit\ time}) = f_c\ f_d\ W\ e^{\frac{-t_r}{t_{sw}}} \qquad \text{Eq. 11}$$

Assuming that all data transitions are independent and that the clock has a fixed period, the mean time between failures (MTBF) is

$$MTBF = \frac{1}{E\ (fail_{unit\ time})} = \frac{e^{\frac{t_r}{t_{sw}}}}{f_c\ f_d\ W} \qquad \text{Eq. 12}$$

where MTBF is a measure of how often, on the average, a metastable event lasts a time t_r or longer.

Metastability Data

The strong resemblance between *Equation 12* and *Equation 2* is based on the predictions of the first-order circuit analysis of an RS flip-flop. In fact, the metastability resolving time constant, t_{sw}, is directly related to the variable τ, which is based on the flip-flop's gain-bandwidth product.

The device-dependent variable W depends mostly on the window of time within which the combination of the input and clock generate a metastable condition. This parameter also depends on process, temperature, and voltage levels. The MTBF equation is usually plotted with t_r (the resolving time allowed for metastable events) on the X axis and the natural log of the MTBF plotted on the Y axis (see the appendix in this note). Because the metastability equation is plotted on a semi-log scale, the graph of t_r vs ln(MTBF) is a line described by the equation

$$\ln\ (MTBF) = \frac{t_r}{t_{sw}} - \ln(f_c\ f_d\ W) \qquad \text{Eq. 13}$$

Graphically, the parameter t_{sw} is 1/slope of the line on this graph. The equation for t_{sw} from the graph is

$$t_{sw} = \frac{t_{r1} - t_{r2}}{\ln(MTBF_1) - \ln(MTBF_2)} \qquad \text{Eq. 14}$$

To determine how often, on the average, a given synchronizer in a system will go metastable (MTBF), you must know the two device-specific parameters W and t_{sw}, which should be available from the manufacturer. *Table 5*, discussed later in this note, lists these values for Cypress PLDs. Additional values you need are the average frequency of both the system data and the synchronizer clock and the amount of time after the synchronizer's maximum clock-to-Q time that is allowed to resolve metastable events.

For example, consider the method for determining the MTBF for a Cypress PALC22V10 registered PLD used as a synchronizer in a system with the following characteristics:

W = 0.125 ps

t_{sw} = 190 ps

f_c = system clock frequency = 25 MHz

f_d = average asynchronous data frequency = 10 MHz

In addition to these values, the PLD's maximum operating frequency, f_{max}, is taken directly from the datasheet. The frequency is specified as the internal feedback maximum operating frequency. It is calculated as

$$f_{max} = \frac{1}{t_{cf} + t_s} = 41.6\ MHz$$

where t_{cf} is the clock-to-feedback time. If the data sheet does not specify t_{cf}, you can use t_{co} as t_{cf}'s upper bound.

Using f_{max}, you calculate the amount of time that a metastable event is allowed to resolve, t_r, with

$$t_r = \frac{1}{f_c} - \frac{1}{f_{max}} = \frac{1}{25\ MHz} - \frac{1}{41.6\ MHz} = 16\ ns$$

Now you enter these values into the MTBF equation, making sure to keep all units in seconds:

$$MTBF = \frac{e^{\frac{t_r}{t_{sw}}}}{f_c\ f_d\ W}$$

$$= \frac{e^{\frac{16 \times 10^{-9}\ s}{190 \times 10^{-12}\ s}}}{25 \times 10^6 s^{-1} \times 20 \times 10^6 s^{-1} \times 0.125 \times 10^{-12} s}$$

$$= 59.7 \times 10^{33} s$$

$$= 1.89 \times 10^{27}\ years = Almost\ forever$$

If the operating frequency of the system, f_c, is simply changed to 33.3 MHz,

$$MTBF = \frac{e^{\frac{6 \times 10^{-9} s}{190 \times 10^{-12} s}}}{33.3 \times 10^6 s^{-1} \times 20 \times 10^6 s^{-1} \times 0.125 \times 10^{-12} s}$$

$$= 623 \times 10^9\ s$$

the system fails, on the average, about every 19,700 years—still beyond the system's normal lifetime.

And if f_c is changed to f_{max} (41.6 MHz),

$$MTBF = \frac{e^{\frac{0 \times 10^{-9} s}{190 \times 10^{-12} s}}}{41.6 \times 10^6 s^{-1} \times 20 \times 10^6 s^{-1} \times 0.125 \times 10^{-12} s}$$

the system fails, on the average, every 9.62 ms.

A 16-ns difference in resolve time, t_r, results in almost 36 orders of magnitude difference in MTBF. Obviously, accurate data is needed to design a system with a high degree of reliability without being overly cautious.

Characterization of Metastability

Many authors (References 6, 8, 9, 10, 11, and 12) have performed numerous experiments on circuits to predict the likelihood of device metastability. These researchers have used several testing theories and apparatus that can be classified into three basic types (Reference 14).

Intermediate voltage sensors constitute the first type. Two voltage comparators determine whether the output voltage, Q, lies between two given voltages. The fixture produces an error output if Q has a level that is neither HIGH nor LOW, hence metastable. *Figure 14* shows an intermediate voltage sensor.

The second type of apparatus uses an output proximity sensor to determine if the Q and $\overline{Q}$ outputs have approximately the same voltages, which would indicate that the device is metastable. *Figure 15* shows an output proximity sensor.

The last type of apparatus uses a late-transition sensor to test for metastability. Note that if one or more gates separate the sensor from the metastable signal, the metastability might not be detected. The test circuitry must infer the occurrence of metastability by some other means. *Figure 16* shows an example of a late-transition sensor. The sample input is detected at time t_1, then at a later time t_2. If these two signals disagree, the device under test was metastable at t_1.

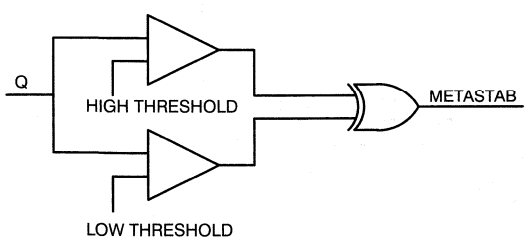


Figure 14. Intermediate Voltage Sensor

Information from Manufacturers

Many semiconductor companies provide metastability data on their parts. However, most companies do not present the data in a format the engineer can use. They either present inconclusive and incomplete data or they assume the engineer can use the data without further explanation. Few companies compare their devices with similar devices.

PLD manufacturers provide little data largely because of a fear that telling the design community that devices can fail in synchronizing applications will cause designers to use a competitor's parts. The truth is that no company can provide a device that is guaranteed never to become metastable when used as a synchronizer. At a given operating frequency, with a given asynchronous input, and given enough time, the device becomes metastable.

Cypress provides you with data you can use to build a system to any given level of reliability when using Cypress PLDs. Cypress has performed numerous tests and collected extensive data on Cypress PLDs, as well as PLDs from other companies. This data

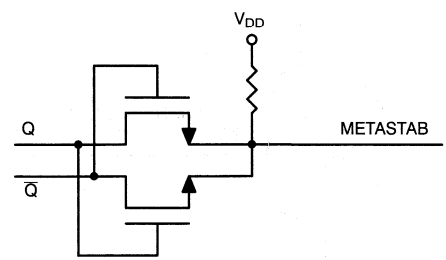


Figure 15. Output Proximity Sensor

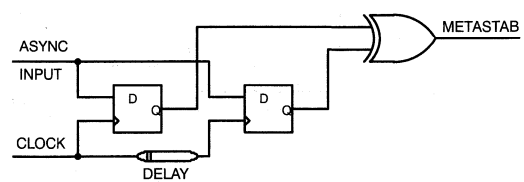


Figure 16. Late Transition Sensor

gives you a perspective of the parts that are best suited for a specific application. Specific data on the metastability characteristics of Cypress PLDs is found in this application note in the Test Results section. Metastability data collected by Cypress for other companies' PLDs is available upon request.

The Test Circuit

Cypress uses a test that falls into the category of the late-transition detection. Directly measuring the outputs of the flip-flop in a PLD are impossible due to the additional circuitry that lies between the flip-flop and the outside world. The metastability detection circuitry must, instead, infer the flip-flop's state.

Figure 17 shows the metastability test circuit implemented in each test PLD. This circuit allows the PLD under test to effectively test itself. The device under test will both produce and record metastable conditions.

Figure 18 is a state diagram showing the operation of the device. During normal operation, the two flip-flops' outputs (F_1, F_2) transition between states S_1 and S_2, depending on the synchronizer's state. During normal operation, the Exclusive-OR on these

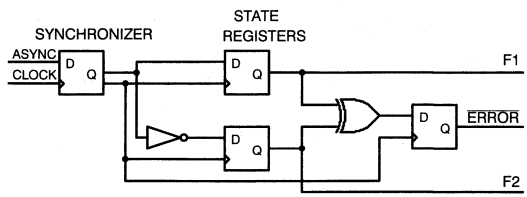


Figure 17. Metastability Test Circuit

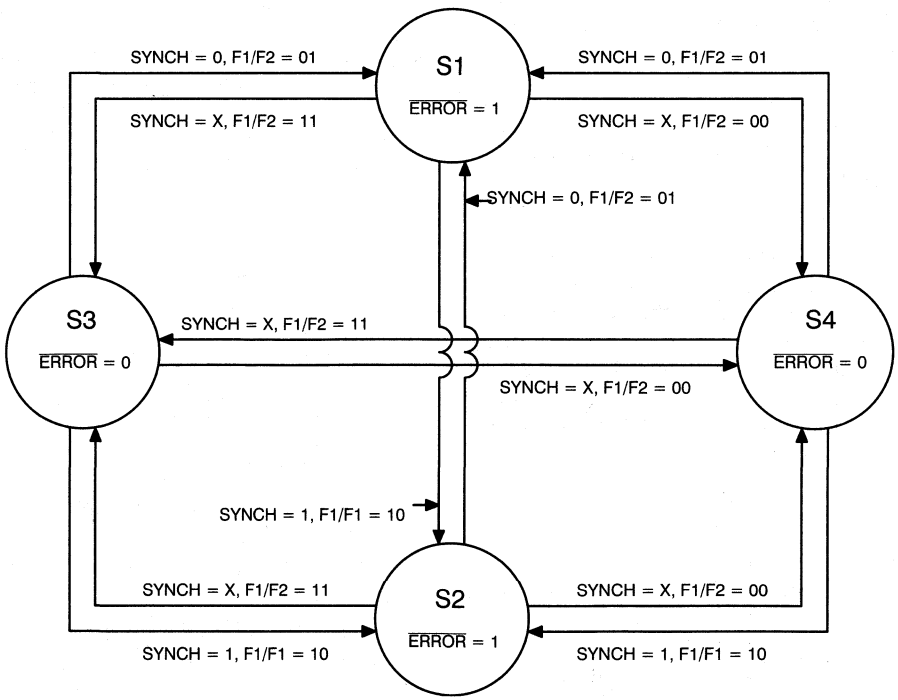


Figure 18. Metastability Testing State Diagram

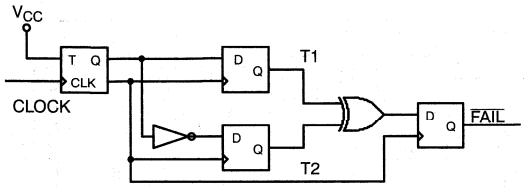


Figure 19. Maximum Operating Frequency Test

outputs produces a HIGH. This indicates either that metastability has not occurred within the device or that metastability that has occurred has resolved before the next clock cycle.

If a metastable event cannot resolve before the next clock cycle, the state machine move to states S_3 or S_4. In this case, the state flip-flops have interpreted the signal from the synchronization register differently; exclusive-ORing this signal produces a LOW at the device's output, indicating that unresolved metastability has occurred.

This test circuit does not catch all metastable events. Specifically, it does not record metastable events that resolve before the next clock cycle. But metastability causes an error only when it has not resolved by the time the signal is needed. The Cypress tests thus reveal the information designers need to know: how often metastability creates an error in the system.

The test circuit also includes the ability to check the maximum operating frequency of the device under test (*Figure 19*). At each clock edge, the first register's output toggles. When the device reaches its maximum operating frequency, the PLD array cannot resolve the changing signal fast enough to produce a valid output. At this speed, one register might resolve the signal correctly and one might not, or both might produce invalid signal resolutions. In any case, when Exclusive-ORing the state T_1/T_2 of the two maximum-frequency testing registers results in anything other than a HIGH, the part's maximum operating frequency is exceeded.

The Test Board

A four-layer printed circuit board with two signal planes, a ground plane, and a power plane is used to

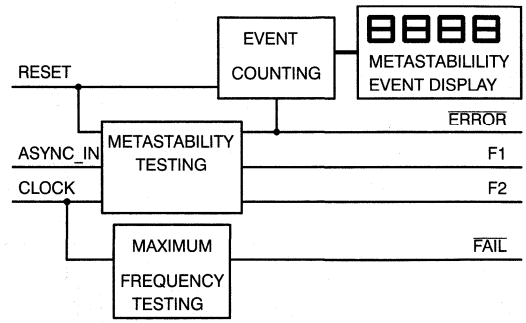


Figure 20. Metastability Test Board Block Diagram

perform the metastability measurements. Using this four-layer board gives a quiet testing environment with reliable, repeatable results. *Figure 20* shows a block diagram of the test board, with the complete schematic shown in *Figure 21*. The device under test (DUT) is decoupled with 0.01-µF and 100-pF capacitors. The test circuit is designed to fit all industry-standard and Cypress-proprietary PLDs. The socket allows DUT pins 1, 2, and 4 to serve as clock pins. Pin 3 is the device's asynchronous input. The $\overline{\text{ERROR}}$ condition is located on pin 27 of a 28-pin device, and the $\overline{\text{FAIL}}$ condition is on pin 20. Two additional outputs, F_1 and F_2, monitor the state of the metastability test circuit flip-flops.

All inputs and outputs connect with BNC connectors located around the board. The clock line, which is terminated with a 50Ω resistor to match the coax input impedance, is buffered with a 74AS04 and isolated from other signals by a ground trace. The input line is also terminated with a 50Ω resistor and buffered with a 74AS04. Four PLDs drive a four-digit LED display that counts metastability occurrences.

After going LOW in response to a metastable event, the $\overline{\text{ERROR}}$ signal automatically transitions HIGH again at the next system clock. This LOW-to-HIGH pulse produces a clock to the input of the first PLD, which in turn increments the display of metastable events. When a digit reaches 9, the next occurrence of metastability generates a cascade signal to the next higher digit.

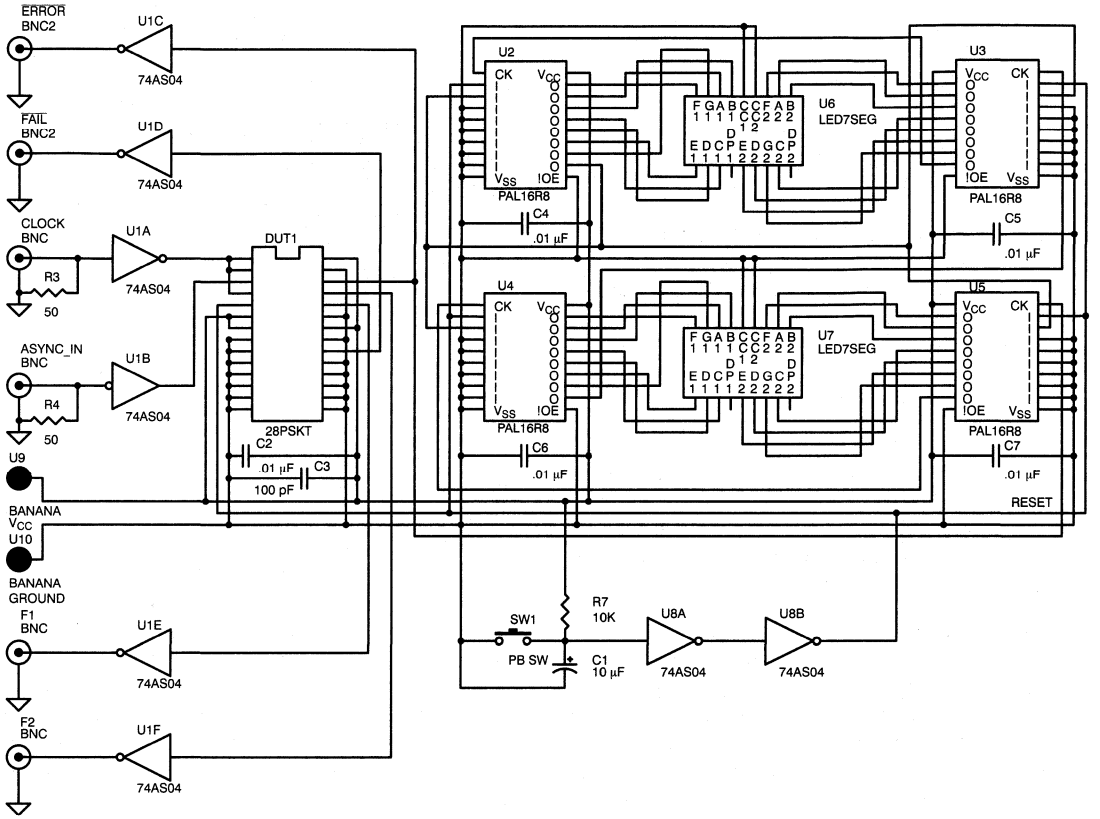


Figure 21. Metastability Test Board Schematic

In this way, the test board can record a maximum of 9,999 metastable events. If a metastable event is received at 9,999, all LEDs switch to E, indicating that an overflow condition occurred. A reset button resets all counters and initializes the DUT.

Test Set-Up

Figure 22 shows a block diagram of the test set-up used for metastability testing. Two independent pulse generators (Hewlett-Packard 8082As) produce the CLOCK and the ASYNC_IN signal to the test board. A Tektronix DAS9200 logic analyzer records metastable events. A 2465 CTS digital oscilloscope with frequency counter accurately determines the DUT's maximum operating frequency and the ASYNC_IN and CLOCK frequencies.

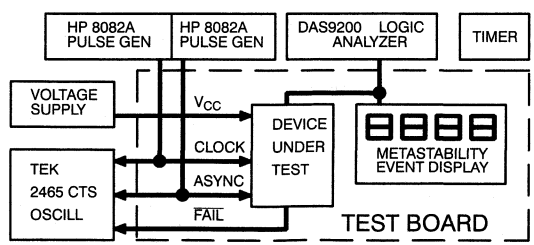


Figure 22. Metastability Test Set-Up

Test Procedure

Cypress has tested all its 20-, 24-, and 28-pin PLDs. The fastest speed grades of each device type were tested because these devices have the best metastable resolution time and thus make the best synchronizers. Several parts from each device type were tested to ensure an average metastability characteristic for that product. Where possible, parts from different date codes were selected to eliminate variations among different wafer lots.

Testing for a specific device starts by creating the high−level description written in VHDL to be used with the *Warp2* VHDL Compiler. *Figures 23 and 24* list the behavioral description used for generating a JEDEC file. All devices were programmed using JEDEC files generated by *Warp2*, except for the CY7C344. The MAX+PLUS development environment was used to produce a design file for this device.

Each part is programmed, then tested for its maximum operating frequency, f_{max}. By attaching the $\overline{FAIL}$ output to the oscilloscope and observing the clock frequency at which the device started to malfunction ($\overline{FAIL}$ going LOW periodically), the maximum operating frequency for that part is determined. f_{max} indicates the maximum rate at which metastability measurements can be taken with accurate results. Above this frequency, metastable events are indistinguishable from errors caused by exceeding f_{max}.

To determine each device's metastability characteristics, measurements are taken of the number of metastable events that occurred in a given time interval for several different clock and data frequencies.

Equation 13 can be used to describe the graph of the metastability characteristics of the device:

$$\ln(MTBF) = \frac{t_r}{t_{sw}} - \ln(f_c\, f_d\, W)$$

The slope of the line, t_{sw}, can be determined only by forcing the Y intercept of the graph ($\ln(f_c f_d W)$) to a constant value when using *Equation 14*:

$$t_{sw} = \frac{t_{r1} - t_{r2}}{\ln(MTBF_1) - \ln(MTBF_2)}$$

Note that t_{sw} is a constant, device-specific parameter.

Because W is also a constant, device-specific parameter, it is only necessary to hold the product $f_c f_d$ constant to make $\ln(f_c f_d W)$ constant. The independent variable t_r is varied by changing f_c to produce changes in the dependent variable ln(MTBF). Decreasing the frequency f_c from its f_{max} value increases the metastable resolution time, t_r, and decreases the probability that a metastable event will last longer than t_r.

As f_c is decreased below a certain limit, the MTBF becomes too large to measure accurately. A metastable event occurring every minute is chosen as the upper limit for MTBF measurements. The range of clock rates for metastability testing is then between f_{max} and the metastable-event-per-minute clock rate. Between these two rates, a selected frequency constant ($f_c f_d$) ensures that no point in this range has a clock frequency less than twice the data frequency. This is because a data signal that transitions more than once per clock period cannot be effectively sampled.

After determining this constant, data is taken from several test points within the test range by varying f_c and f_d. The data at each test point is averaged among all test devices, and the equation for the line through these points is determined using a linear regression analysis. The correlation between the line and the data points verifies that the metastability equation accurately describes the test data. From the calculated results, the constants W and t_{sw} are extracted.

Test Results

Table 5 and the Appendix list the results of the metastability analysis of Cypress PLDs. *Table 5* also lists the maximum data book operating frequency, f_{max}; the metastability equation constants, W and t_{sw}; the metastability resolve time, t_r, required for a 10-year MTBF; and the process for that part.

You can use this data to determine the maximum metastability resolve time (t_r) that you must use in a system to yield a given degree of reliability. The graphs and constants (W and t_{sw}) can be used with any speed grade of the device, but it is suggested that the fastest speed grade of the specific PLD be used

```
package test is
  component metastability port (
    clock, async_in, reset : in bit;
      fail, perror, f1, f2 : out bit);
  end component;
end test;

entity metastability is port (
  clock, async_in, reset : in bit;
    fail, perror, f1, f2 : out bit);
end metastability;

use work.bv_math.all

architecture fsm of metastability is

signal  sync  : bit;
signal  tsync : bit;
signal  t1,t2 : bit;
signal  f1_tmp, f2_tmp : bit;
signal  error_tmp : bit;
signal  fail_tmp : bit;

begin
  proc1: process begin

    wait until clock = '1';
    sync <= async_in;

  end process;

  proc2: process begin

    wait until clock = '1';
    f1_tmp <= sync;
    f2_tmp <= inv(sync);

  end process;

  proc3: process begin

    wait until clock = '1';
    error_tmp <= ((((inv(reset) and f1_tmp) and inv(f2_tmp))
      or ((inv(reset) and inv(f1_tmp)) and f2_tmp))
      or (reset and inv(error_tmp)));

  end process;
```

Figure 23. *Warp2* VHDL Behavioral Description for Metastability Testing

```
  proc4: process begin

    wait until clock = '1';
    if (tsync = '1') then
      tsync <= '0';
    else
      tsync <= '1';
    end if;

  end process;

  proc5: process begin

    wait until clock = '1';
    t1 <= tsync;
    t2 <= inv(tsync);

  end process;

  proc6: process begin

    wait until clock = '1' ;
    fail_tmp <= (t1 xor t2);

  end process;

fail <= inv(fail_tmp);
perror <= inv(error_tmp);
f1 <= inv(f1_tmp);
f2 <= inv(f2_tmp);

end fsm;
```

Figure 23. *Warp2* VHDL Behavioral Description for Metastability Testing (continued)

for optimum synchronizer performance. These graphs indicate the time (t_r) and the device's minimum clock period that must be used to produce a desired degree of reliability.

For example, to determine the operating parameters of the Cypress PALC22V10−20 from *Table 5* when using the device as a synchronizer, determine the desired MTBF. With a 10-yr (315×10^6s) MTBF, for instance, a synchronization failure will occur once every 10 years on the average. The maximum operating frequency (f_{max}) from the PALC22V10's data sheet is 41.6 MHz. From this information, you can determine the minimum time (t_r) beyond the device's minimum operating period that must be added for metastability resolution:

$$MTBF = \frac{e^{\frac{t_r}{t_{sw}}}}{f_c f_d W}$$

$$t_r = t_{sw} (\ln(MTBF) + \ln(f_c f_d W))$$

$$t_r = (0.190 \times 10^{-9}s) [\ln(315 \times 10^6 s)$$

$$+ \ln(41.6 \times 10^6 \times 41.6 \times 10^6 \times 0.125 \times 10^{-12})]$$

$$= 4.73\ ns$$

This analysis assumes that the clock, f_c, operates at f_{max} (41.6 MHz) and that the average asynchronous data frequency is no more than half the clock fre-

quency. The latter condition ensures effective data sampling by the synchronizer. f_d, as explained in the Statistical Analysis of Metastability section represents the rate at which the data changes state. f_d is twice the average frequency of the asynchronous data input because, during any given asynchronous data period, the asynchronous data changes state twice: once from LOW to HIGH and again from HIGH to LOW. Because either of these state changes can cause a metastable event, f_d must be set to twice the average asynchronous data frequency when determining the worst-case MTBF.

Due to the real-world uncertainty in factors such as trace delays and the skew in clock generators, 5 ns is used instead of 4.73 ns for t_r. The synchronizer's maximum operating frequency, f_c, in this system is then

$$f_c = \frac{1}{t_s + t_{cf} + t_r} = \frac{1}{10ns + 12ns + 5ns} = 37.0\ MHz$$

The effective MTBF using these new values for t_r and f_c is

$$MTBF = \frac{\frac{5 \times 10^{-9} s}{e^{0.190 \times 10^{-9} s}}}{37.0 \times 10^{6} s^{-1} \times 37.0 \times 10^{6} s^{-1} \times 0.125 \times 10^{-12} s}$$

$$= 1.57 \times 10^{9} = 49.7\ yrs$$

Another example focuses on the CY7C330–50 used as a synchronizer in a system whose output registers are clocked at an f_c of 35.7 MHz, and the data has an average frequency of 10 MHz. The MTBF for this device used as a synchronizer is calculated by first determining the metastable resolution time, t_r, allowed for synchronization. The maximum operating frequency of the part is specified in Cypress's *Data Book* as

$$f_{max} = \frac{1}{t_{co} + t_{is}}$$

where t_{co} in this case specifies the clock-to-feedback delay, and t_s specifies the set-up time of the output registers. t_r is calculated with the equation:

$$t_r = \frac{1}{f_c} - \frac{1}{f_{max}} = \frac{1}{35.7\ MHz} - \frac{1}{50.0\ MHz} = 8\ ns$$

With this result, the MTBF is

$$MTBF = \frac{\frac{8 \times 10^{-9} s}{e^{0..290 \times 10^{-9} s}}}{35.7 \times 10^{6} s^{-1} \times 20.0 \times 10^{6} s^{-1} \times 1.02 \times 10^{-12} s}$$

$$= 1.31 \times 10^{9}\ s = 41.6\ yrs$$

This equation uses the same values for W and t_{sw} with this 50-MHz device as with the 66-MHz device listed in *Table 5*. As stated previously, the constants listed in *Table 5* are valid for all speed grades of a specific device. Also note that the 10-MHz average data frequency is doubled to produce the frequency of data transitions, f_d.

The last example illustrates how to use a Cypress PALC22V10C–10 as a synchronizer. For a 10-year MTBF, assuming the maximum f_c from Cypress's *Data Book* and f_d, the required t_r is

$$t_r = (0.547 \times 10^{-9} s)\ [\ln(315 \times 10^{6} s) + \ln(90.9 \times 10^{6} \times 90.9 \times 10^{6} \times 8.08 \times 10^{-15})]$$

$$= 13.0\ ns$$

Using this result, the synchronizer's maximum operating frequency is reduced from 90.9 MHz to

$$f_c = \frac{1}{\frac{1}{f_{max}} + t_r} = \frac{1}{\frac{1}{90.9\ MHz} + 13.0\ ns} = 41.6\ MHz$$

Two-Stage Synchronization

As explained earlier, you can use a second register in series to perform two-stage synchronization (*Figure 4*). This is accomplished by feeding the output of the first synchronization register to the input of the second synchronization register. In PLDs, this method is common because the first synchronization stage can synchronize the asynchronous input signal, and the second synchronization stage can perform a Boolean function on a combination of the input and output signals. Boolean functions can be performed at either stage; the metastability characteristics listed in *Table 5* apply to PLD registers' asynchronous inputs that are used directly as well as asynchronous inputs used as a Boolean combination of existing inputs and outputs.

Table 5. Metastability Characteristics of Cypress PLDs

Device	f_{max} (MHz)	W (s)	t_{sw} (s)	t_r for 10-yr MTBF (ns)
PALC16R8–25	28.5	9.503E–12	0.515E–9	14.68
PAL16R8–5	125	94.48E–12	0.299E–9	9.48
PALC20G10–20	41.6	3.730E–12	0.173E–9	4.91
PALC20RA10–15	33.3	2.860E–12	0.216E–9	5.87
PAL22V10C–7	111	0.389E–12	0.546E–9	15.50
PAL22V10CF–7	111	0.398E–12	0.570E–9	16.21
PALC22V10D–7	100	32.35E–12	0.347E–9	10.56
PALC22V10B–15	50.0	55.76E–12	0.261E–9	8.19
PALC22V10–20	41.6	0.125E–12	0.190E–9	4.73
CY7C330–66	66.6	1.020E–12	0.290E–9	8.12
CY7C331–20	31.2	0.298E–9	0.184E–9	5.91
CY7C335–100	58.8	0.288E–12	0.189E–9	4.95
CY7C344–20	41.6	0.966E–9	0.223E–9	7.55

When implementing a two-stage synchronizer in a PLD, the probability that a synchronizer is metastable after the second stage of synchronization is the square of the probability that a synchronizer is metastable after the first stage of synchronization. The MTBF equation is

$$MTBF = \left(\frac{e^{\frac{t_r}{t_{sw}}}}{f_c\, f_d\, W}\right)^2$$

From this result, the equation for t_r becomes

$$t_r = \frac{t_{sw}\,(\ln(MTBF) + 2 \times \ln(f_c\, f_d\, W))}{2}$$

Using this result for a two-stage synchronizer in a Cypress PALC22V10C, the t_r for a 10-year MTBF is reduced from 13.0 ns to

$$t_r = (0.5)(0.547 \times 10^{-9}s)\ [\ln(315 \times 10^{6}s) + \ln(90.9 \times 10^{6} \times 90.9 \times 10^{6} \times 8.08 \times 10^{-15})]$$
$$= 7.65\ ns$$

The maximum f_c increases from 41.6 MHz to

$$f_c = \frac{1}{\frac{1}{f_{max}} + t_r} = \frac{1}{\frac{1}{90.9\ MHz} + 7.65\ ns} = 53.6\ MHz$$

This example shows that if the cycle of latency caused by the additional synchronization stage is acceptable, you can dramatically increase the synchronizer's maximum operating frequency.

References

1. Lubkin, S., (Electronic Computer Corp.), "Asynchronous Signals in Digital Computers," *Mathematical Tables and Other Aids to Computation*, Vol. 6, No. 40, October 1952, pp. 238–241.

2. Nootbaar, Keith, (Applied Microcircuits Corp.), "Design, Testing, and Application of a Metastable-Hardened Flip-Flop," WESCON 87 (San Francisco, CA, Nov. 17–19, 1987), Electronic Conventions Management, Los Angeles, CA 90045.

3. Stoll, Peter A., "How to Avoid Synchronization Problems," *VLSI Design*, November/December 1982, pp. 56–59.

4. Chapiro, Daniel M., *Globally-Asynchronous Locally-Synchronous Systems*, Stanford University, Department of Computer Science Report No. STAN–CS–84–1026, October 1984.

5. Horstmann, Jens U., Eichel, Hans W., Coates, Robert L., "Metastability Behavior of CMOS ASCI Flip-Flops in Theory and Test," *IEEE Journal of Solid-State Circuits*, Vol. 24, No. 1, February 1989, pp. 146–157.

6. Wormald, E.G., "A Note on Synchronizer or Interlock Maloperation," *Professional Program Session Record 16*, WESCON 87, November 17–19, 1987, Electronic Conventions Management, Los Angeles, CA 90045.

7. Pechouchek, Miroslav, "Anomalous Response Times of Input Synchronizers," *IEEE Trans. Computers*, Vol. C–25, No. 2, February 1976, pp. 133–139.

8. Chaney, T. J., "Comments on 'A Note on Synchronizer or Interlock Maloperation," *IEEE Trans. Computing*, Vol. C–28, No. 10, Oct. 1979, pp. 802–804.

9. Couranz, George R., Wann, Donald F., "Theoretical and Experimental Behavior of Synchronizers Operating in the Metastable Region," *IEEE Trans. Computers*, Vol. C–24, No. 6, June 1975, pp. 604–616 .

10. Veendrick, Harry J.M., "The Behavior of Flip-Flop Used as Synchronizers and Prediction of Their Failure Rate," *IEEE Journal of Solid-State Circuits*, Vol. SC–15, No. 2., April 1980, pp. 169–176.

11. Kacprzak, Tomasz, Albicki, Alexander, "Analysis of Metastable Operation in RS CMOS Flip-Flops," *IEEE Journal of Solid-State Circuits*, Vol. SC–22, No. 1, February 1987, pp. 57–64.

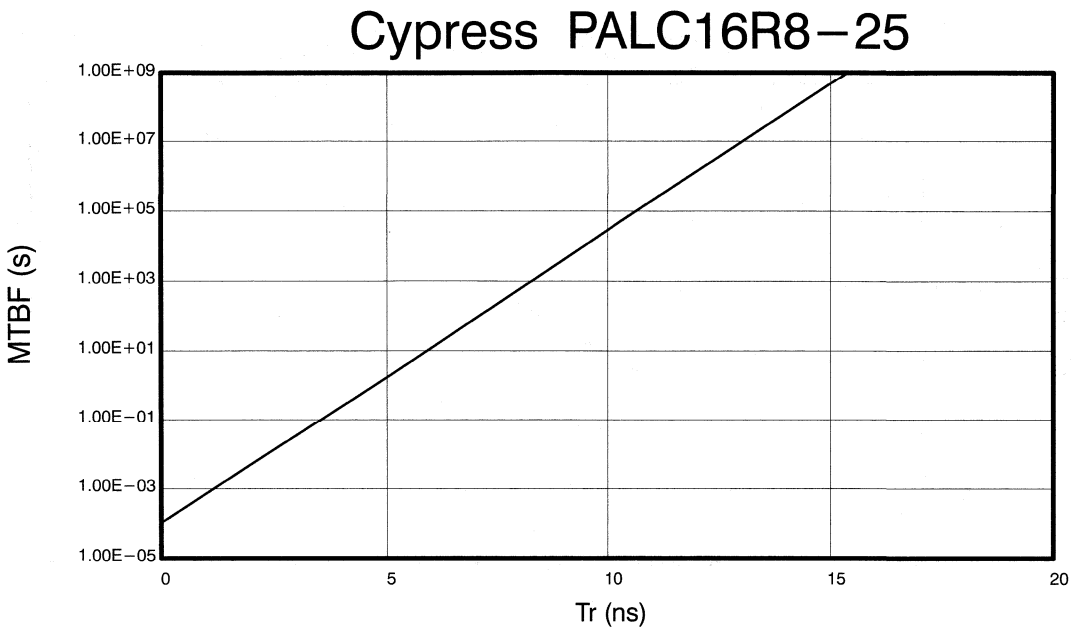
12. Flannagan, Stephen T., "Synchronization Reliability in CMOS Technology," *IEEE Journal of Solid-State Circuits*, Vol. SC–20, No. 4, Aug 1985, pp. 880–882.

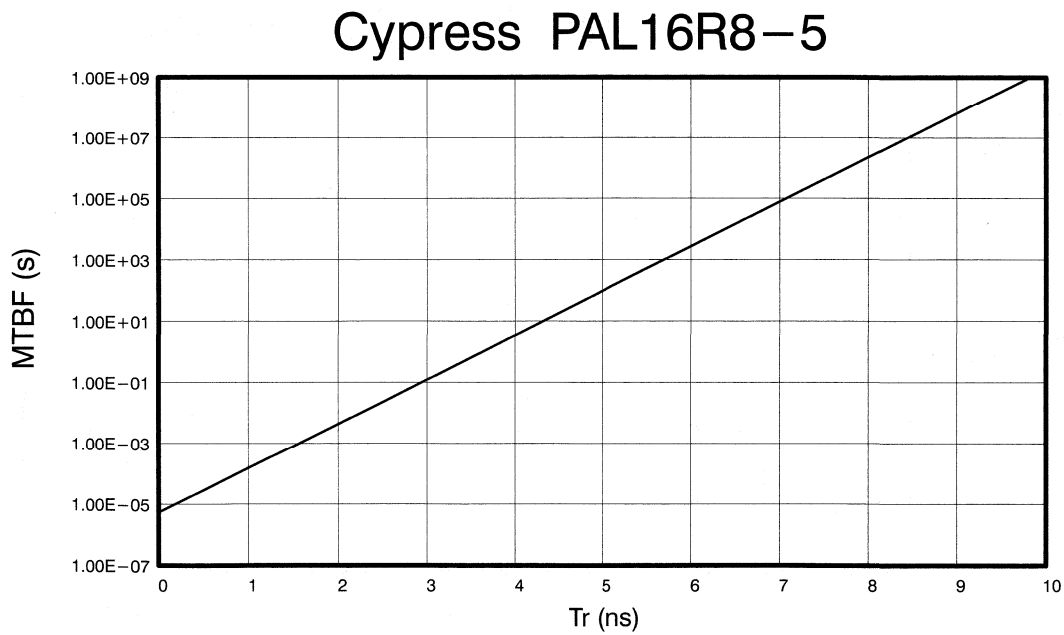
13. Wakerly, John F., *A Designers Guide to Synchronizers and Metastability*, Center for Reliable Computing Technical Report, CSL TN #88–341, February, 1988 Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA.

14. Freeman, Gregory G., Liu, Dick L., Wooley, Bruce, and McClusky, Edward J., *Two CMOS Metastability Sensors*, CSL TN# 86–293, June 1986, Computer Systems Laboratory, Electrical Engineering and Computer Science Departments, Stanford University, Stanford, CA.

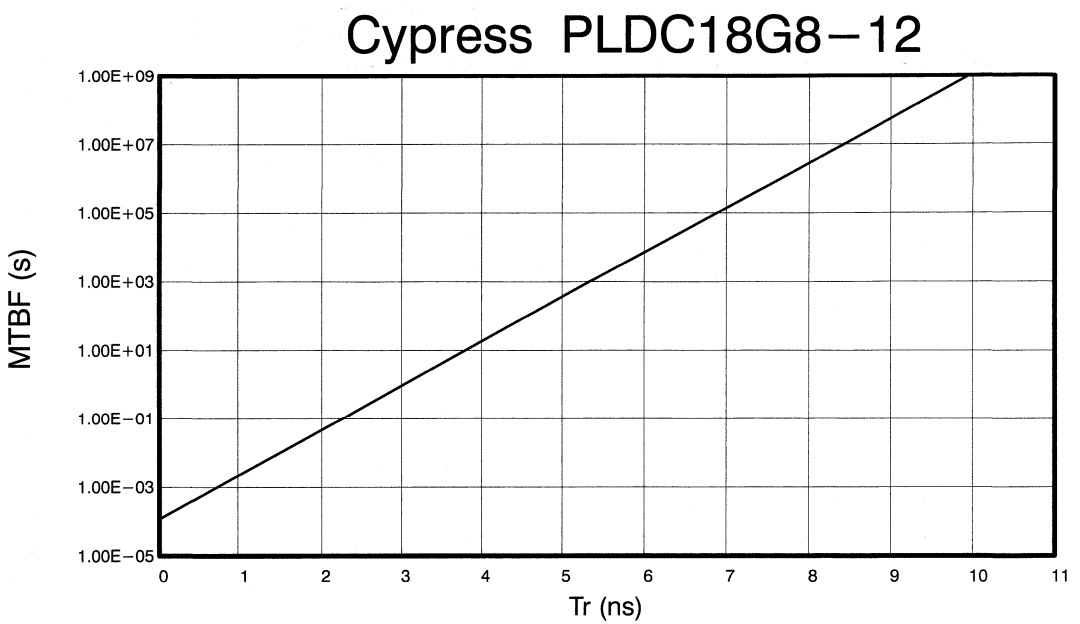
15. Rubin, Kim, "Metastability Testing in PALs," WESCON 87 (San Francisco, CA, Nov. 17–19, 1987), Electronic Conventions Management, Los Angeles, CA 90045.

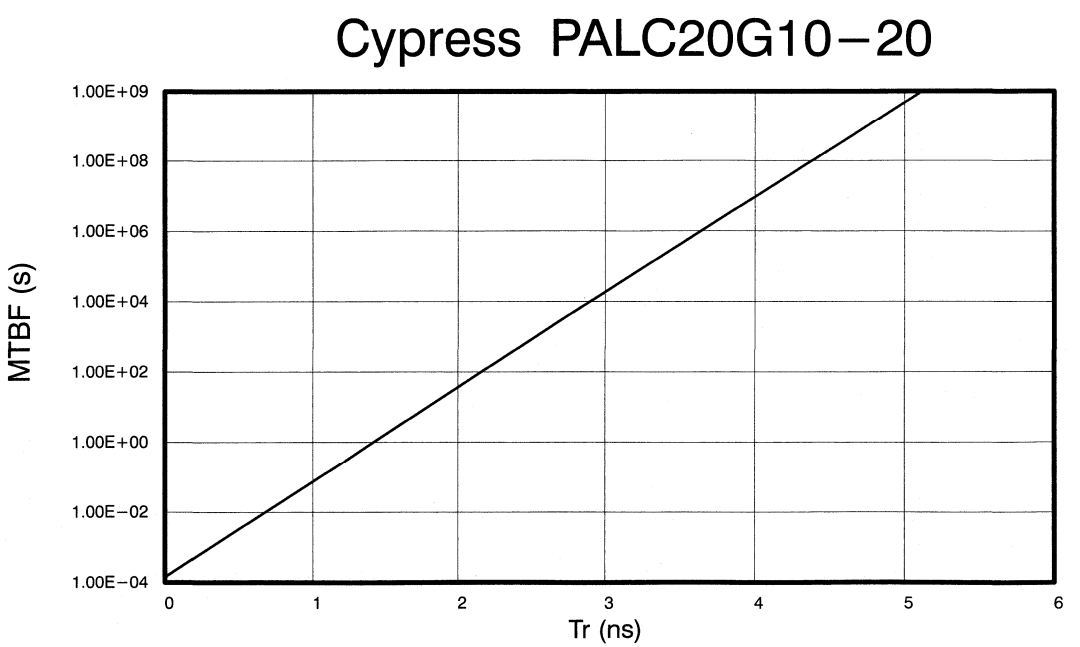
Appendix A. Metastability Graphs of Cypress Devices



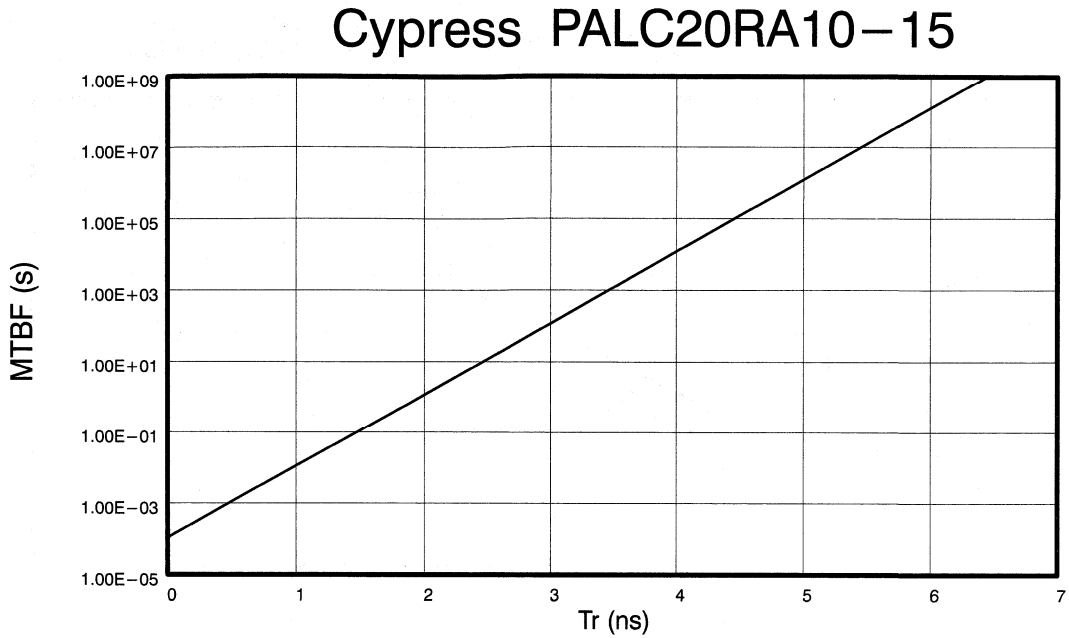


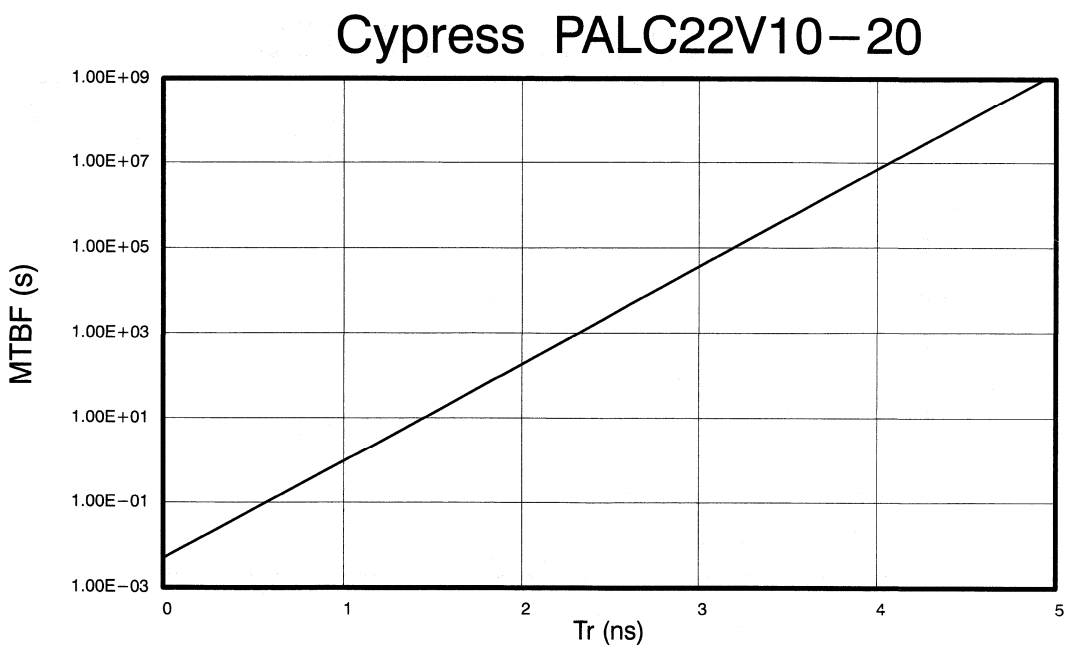
Appendix A. Metastability Graphs of Cypress Devices (continued)



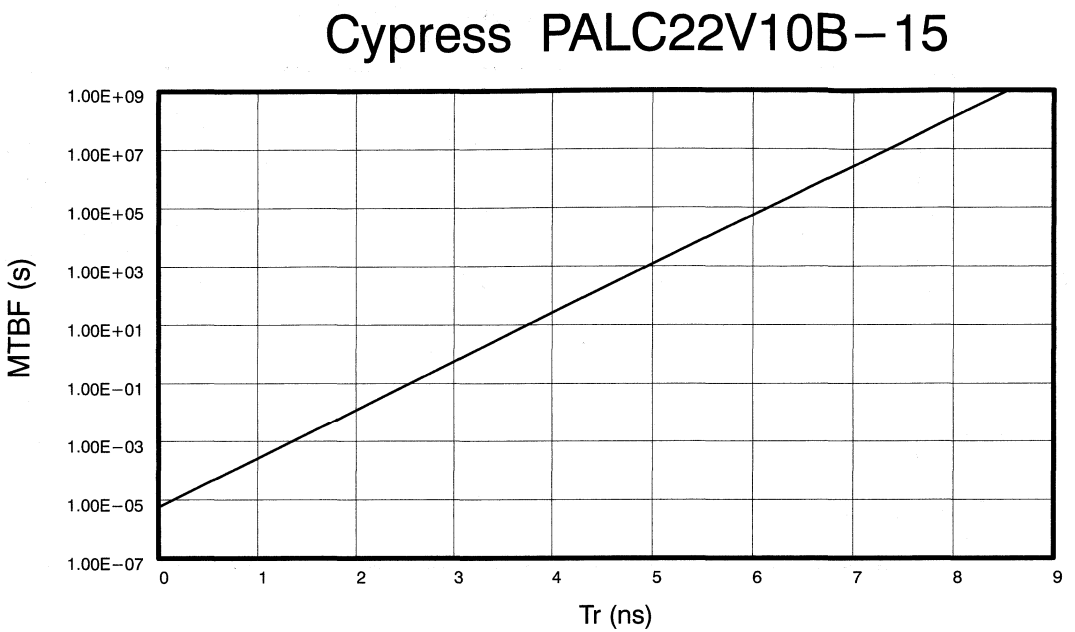


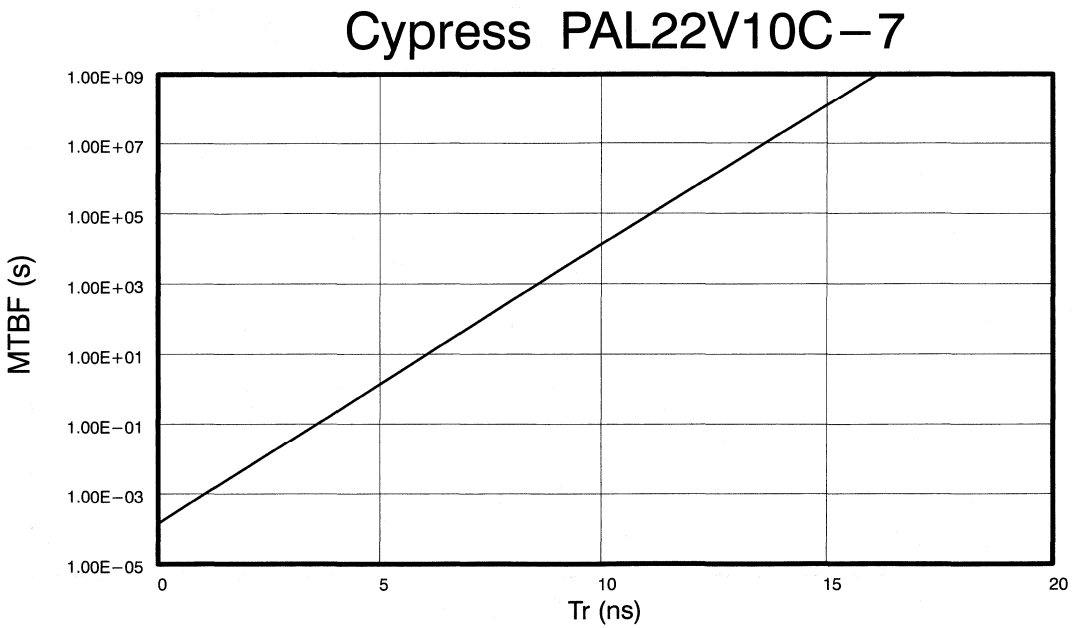
Appendix A. Metastability Graphs of Cypress Devices (continued)



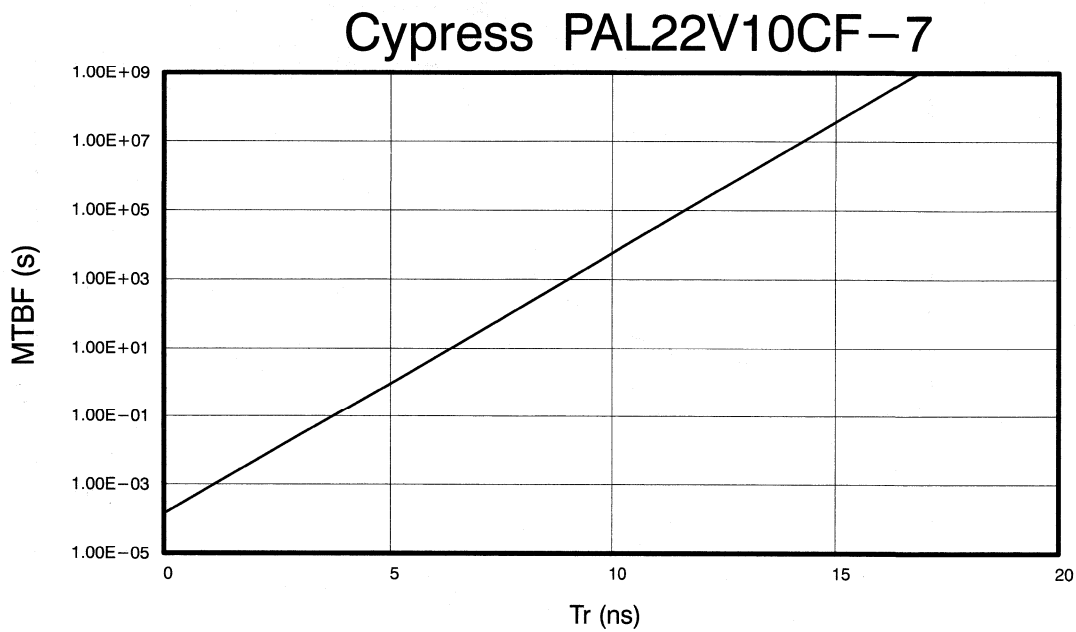


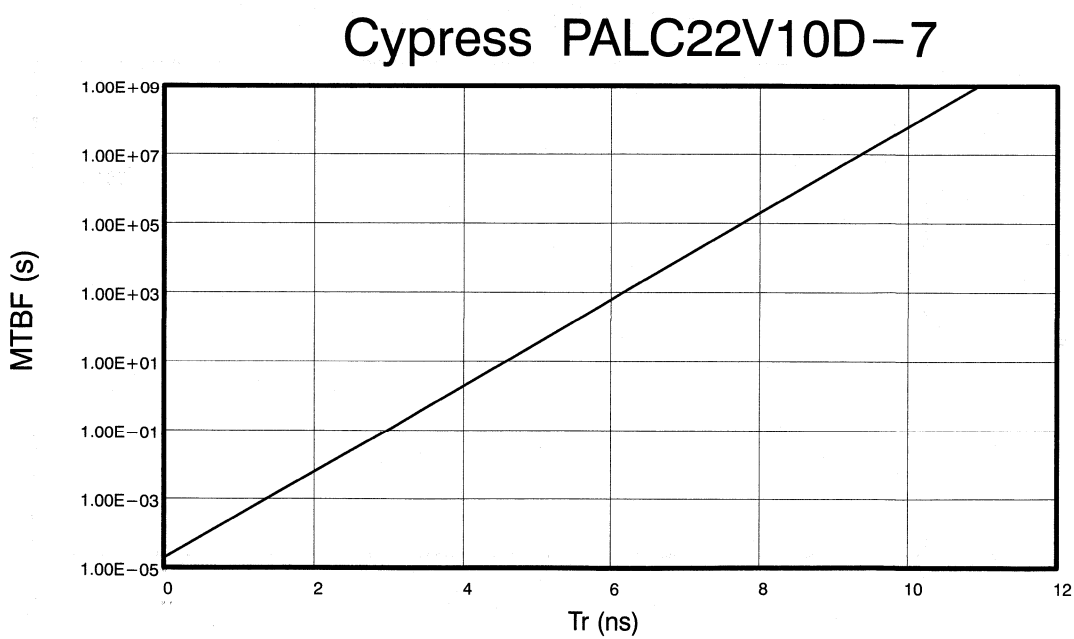
Appendix A. Metastability Graphs of Cypress Devices (continued)



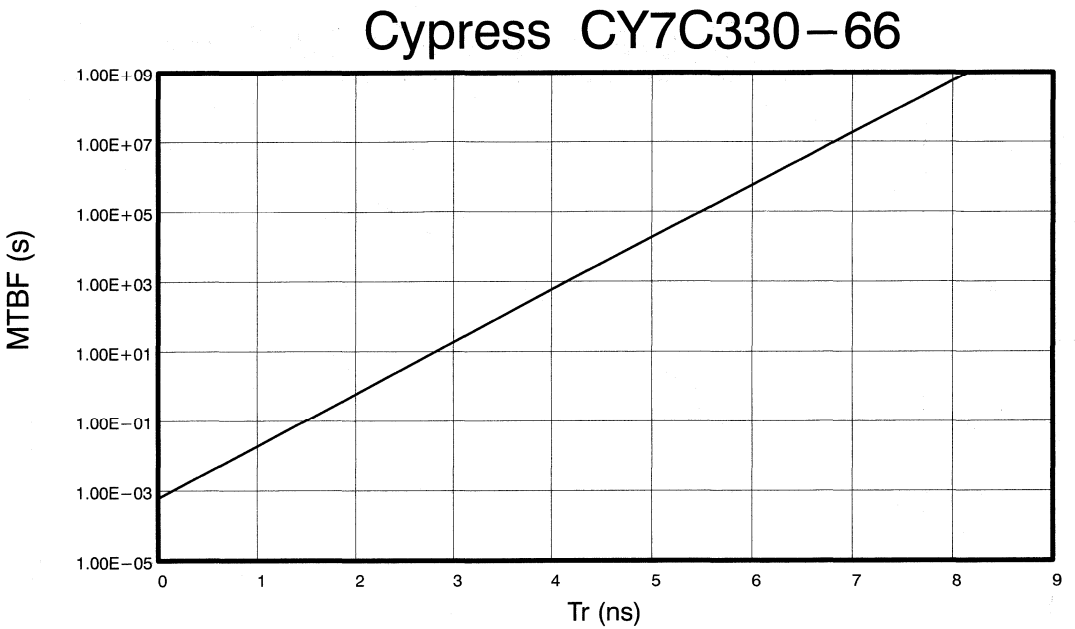


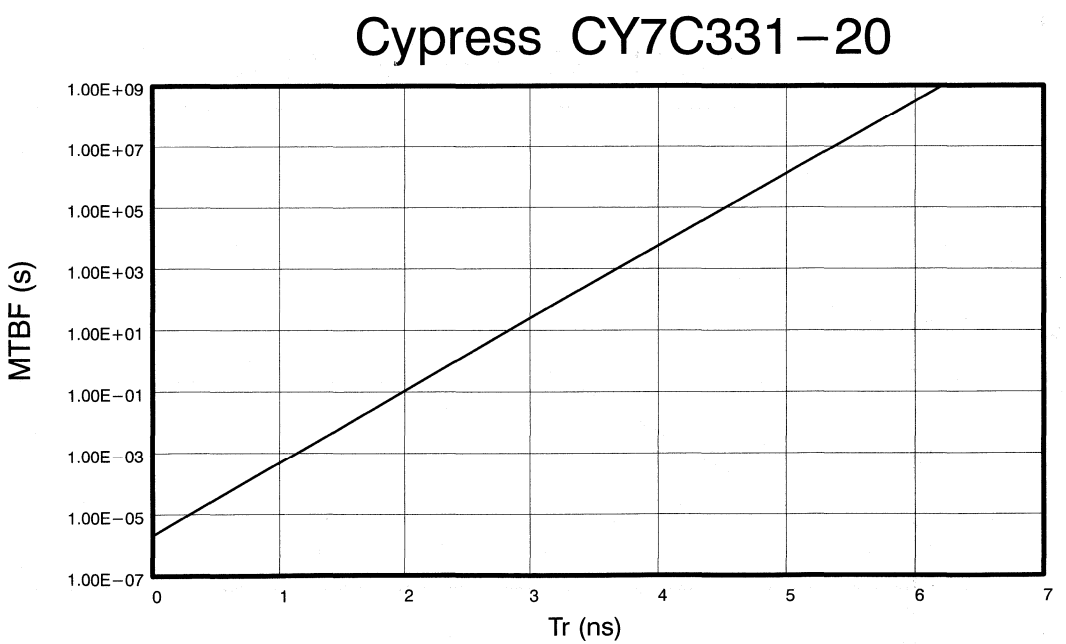
Appendix A. Metastability Graphs of Cypress Devices (continued)



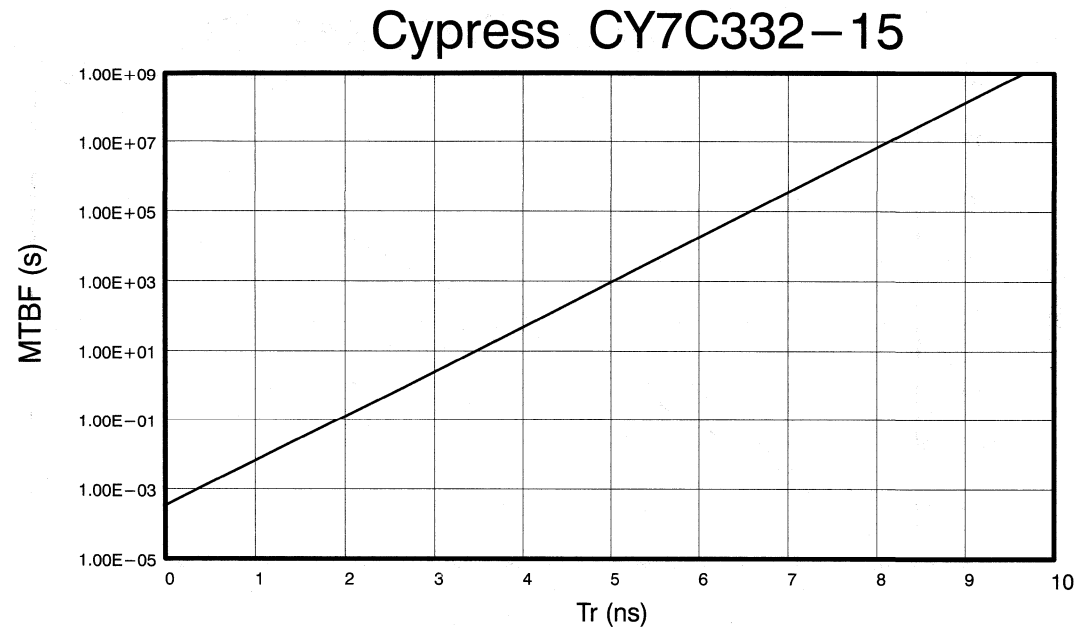


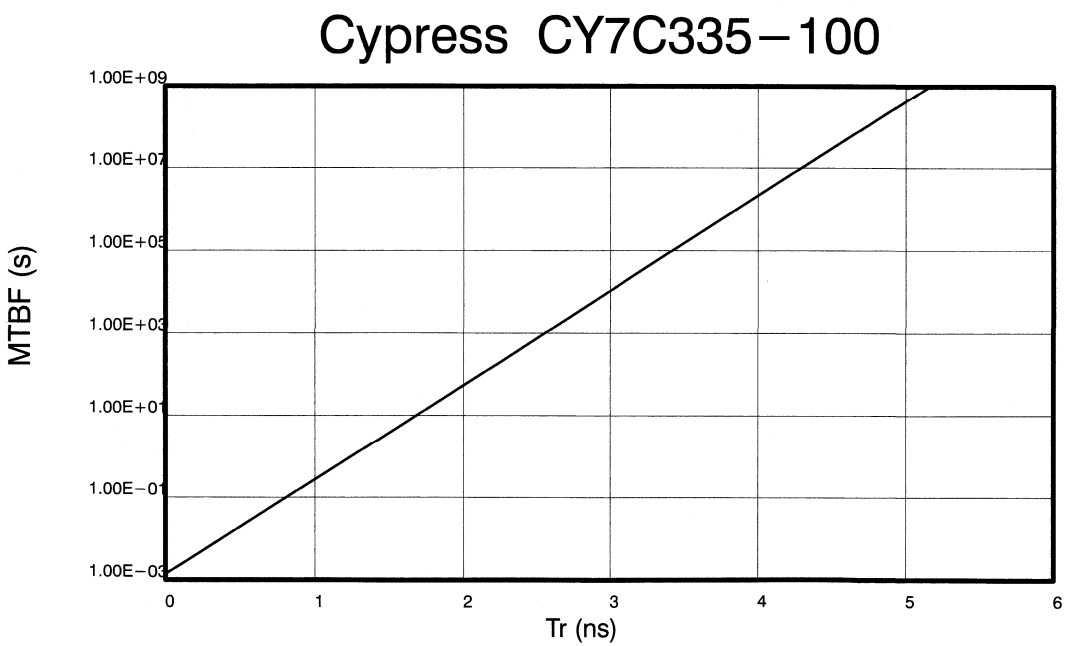
Appendix A. Metastability Graphs of Cypress Devices (continued)



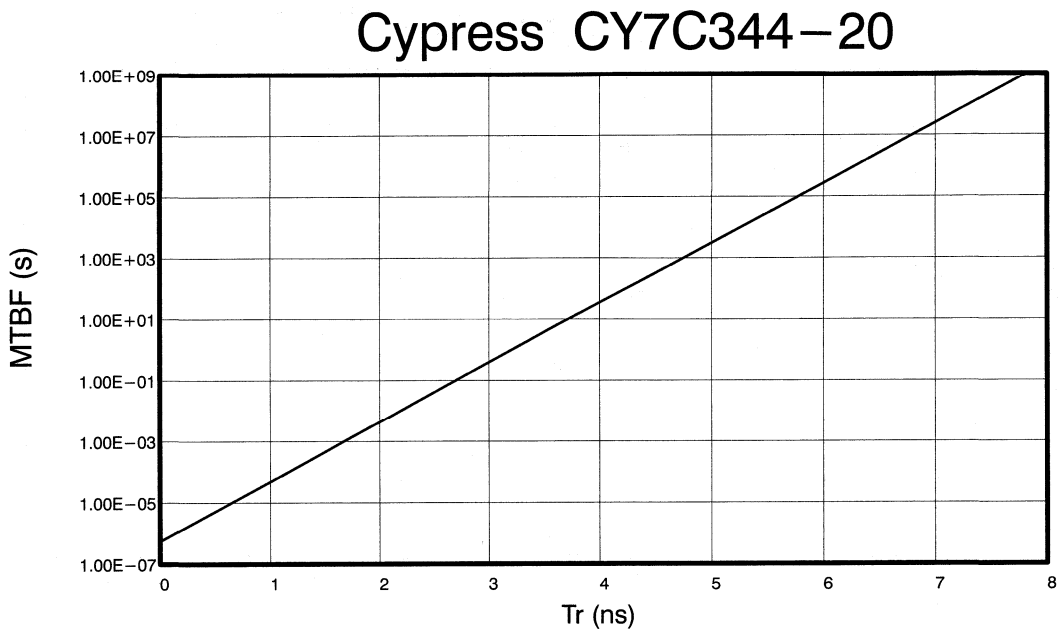


Appendix A. Metastability Graphs of Cypress Devices (continued)





Appendix A. Metastability Graphs of Cypress Devices (continued)



Document #: PLD0007−0294

State Machine Design Considerations and Methodologies

The use of state machines provides a systematic way to design complex sequential logic circuits—an increasingly popular approach since the advent of PLD (Programmable Logic Device) circuitry. This application note describes the many options encountered during the state machine design cycle. By exhaustively walking through the PLD-based design example presented here, you can weigh the merits of several design approaches.

Definitions of Commonly Used Terms

External input vector—External signals (stimulus) applied to the state machine.

System outputs—Signals generated by the state machine that are explicitly designed for availability to the external system (hardware outside of the state machine). Registered system outputs can also be fed back into the state machine as part of the State Vector, which is then used in the decode of the state machine's next state.

State registers—Registers used exclusively for determining the next state of the machine (feedback).

State outputs—Outputs of the state registers that are available to the external system. (They are typically available to the external machine for debug or due to the lack of buried registers.)

State vector or machine state—The registered feedback information defining the present state of the machine and required to determine the next state of the machine.

State path—The transitional condition that must be met for the state machine to progress from one state to another. The state path typically consists of one or more product terms generated from external inputs, although other state paths are possible.

Total input vector—The combination of the external input vector and the state vector. The total input vector is decoded to generate the next state of the machine.

State Machine Entry Methods

There are many ways of describing a state machine, each with distinct advantages and disadvantages. Three popular description methods are state diagrams, state tables, and high-level languages (HLLs). The state diagram provides an easily observable flow description of the state machine. Because the ability to view the flow of states provides distinct documentation advantages, state diagrams will be used throughout this application note to describe the example state machine.

Upon completing a state diagram, you can easily convert the diagram's visual information into the other types of state machine description or directly into Boolean equations. Several available software programs accept their own forms of state table, HLL, and/or Boolean entry. You can enter all these formats easily via your favorite text editor. The software then translates the inputs into suitable forms (usually a JEDEC map) for hardware implementation.

Another method of describing a state machine, the state table, offers perhaps the most concise description. Its major advantage over the other entry methods is the availability of state table reduction methods (see Reference 1). When applied to your state

table definition, a reduction program generates a minimal model for the function. The software used for state machine synthesis throughout this application note uses the state table method of entry. The program is called LOG/iC™ from Isdata Corporation.

Finally, high-level language (HLL) state machine entry is probably the most popular form of state machine design. HLLs typically offer C-language-like instructions (e.g., case, if-then-else, etc.) to describe the machine.

A Sample State Machine

The sample state machine is a clock generator for a pipelined (three system execution stages), bit-slice-based, central processing unit (CPU). Each of the three system execution stages contains two clocks for a total of six system clocks for every instruction execution. With pipelining enabled, each instruction takes an average of two clock periods. Further, external hardware unaffected by CPU wait and stop states (e.g., cache memory) needs both polarities of an additional free-running clock.

To minimize clock edge skew, the state machine provides both versions of the clock. To put the timing of this application into perspective, executing each pipeline stage in an 80-ns period (or 12.5 MHz) requires the state machine to run at 25 MHz. This speed is well within the range of the available PALs, EPLDs and PROMs that can be used to implement the state machine.

Each of the pipeline's three execution stages has a specific function. Briefly, the first stage of the pipeline accesses the Writable Control Store (WCS) RAM. The Arithmetic Logic Unit (ALU) execution occurs during the second stage of the pipeline. Finally, the third pipeline stage clocks status and memory address registers. The function(s) performed during each of the three stages are described in greater detail in the State Machine Output Definition section of this application note.

If this design only generates a simple set of pipelined clocks, why not use shift registers and miscellaneous glue logic instead of a state machine? There are two reasons to consider a state machine. First, it is usually desirable to minimize the number of chips required; the state machine in PLD form might need external glue logic, but significantly less than the shift register solution.

The second reason for considering a state machine is that this application requires more then just a simple set of pipeline clocks. The function of the clock signals is to provide control of the CPU in multiple modes of operation. The desired modes of operation follow.

PIPELINED RUN Mode

In this mode, the CPU simultaneously performs the instructions in all three stages of the pipeline. For example, while instruction n does an ALU operation, instruction n+1 accesses WCS, and instruction n−1 clocks ALU status.

NONPIPELINED RUN Mode

NONPIPELINED RUN mode performs all three stages of instruction execution without overlap. The time to complete one nonpipelined instruction equals the average of three pipelined instructions.

CPU STOP

The system must have a way to perform an orderly stop of CPU execution from both of the above run modes. This stop might be the result of several possible conditions, including a utility stop from a system control unit, a single step, a breakpoint, or a response to external hardware (e.g., a logic analyzer). The free-running clocks continue to run during the CPU STOP mode and remain running at all times, except during a reset condition.

CPU WAIT

In CPU WAIT mode, an external condition causes a delay in an instruction's execution. The instruction pauses until the external condition is removed. One application for the CPU WAIT mode is to handle a cache miss. When a cache miss occurs, the CPU remains in the CPU WAIT mode until the cache completes its memory transfer.

SINGLE STEP

The ability to execute one instruction at a time is needed to debug the CPU. You can easily imple-

ment SINGLE STEP external to the clock state machine by pulsing the RUN signal. SINGLE STEP mode is described further in the State Machine Input Definition section of this application note.

INTERRUPT

A variety of system conditions can interrupt the CPU out of its normal execution sequence and immediately start the execution of the interrupt handler. The influence of the INTERRUPT mode on the system clocks will be discussed in greater detail later in this application note.

REPEAT INSTRUCTION

The REPEAT INSTRUCTION mode is a CPU debug feature. It is a good idea to implement this mode external to the clock state machine. By dubbing the clock to the instruction register and the interrupt line to the clock state machine, the CPU continually executes the instruction in the instruction register.

Synchronous vs. Asynchronous Machine

At this point in the state machine design, an appropriate type of state machine must be chosen to match the application. Two major types are the asynchronous and the synchronous implementations. The asynchronous machine changes state when one or more of its inputs changes from a previously stable input state. After a state change, the outputs of the state machine settle, while the machine stabilizes once again. A basic example of an asynchronous state machine would be a simple SR latch built from two NAND gates (*Figure 1*). For the clocking application considered in this application note, the asynchronous state machine implementation would be a poor choice, due to the instability of the system outputs.

The synchronous state machine offers a better choice. A synchronous state machine block diagram appears in *Figure* 2. Generally, a synchronous state machine samples the total input vector at specific periods to determine the machine's next state. When designing synchronous state machines, it is important to avoid state register metastability. External inputs to the machine must be synchronized to guarantee stable state register inputs, and the feedback time plus data set-up time to the state register clock must be less then or equal to the state clock period.

The modern theory of synchronous state machines was pioneered by Mealy and Moore (see Reference 1). Mealy and Moore machines differ slightly from each other in the way they control the system outputs. During a specific machine state, a Mealy machine allows the input conditions to alter the system outputs (the outputs depend on the "total" input state). In contrast, a Moore machine system outputs depend only on the present machine state. Thus, the system outputs remain stable until the next time period, when the Moore machine samples the total input vector to determine the next state. If all design conditions are met (external inputs are stable prior to the next state clock), the Moore machine provides glitch-free system outputs—a desirable characteristic for the CPU system clock. The design described here is therefore implemented as a Moore machine.

Clock Generator Output Definition

As explained earlier, each of the three system execution stages contains two clocks for a total of six system clocks for every instruction execution. The naming convention for these clocks is

`CLK_xy`

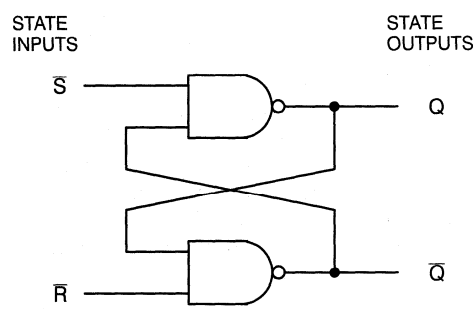


Figure 1. SR Latch, Asynchronous State Machine Example

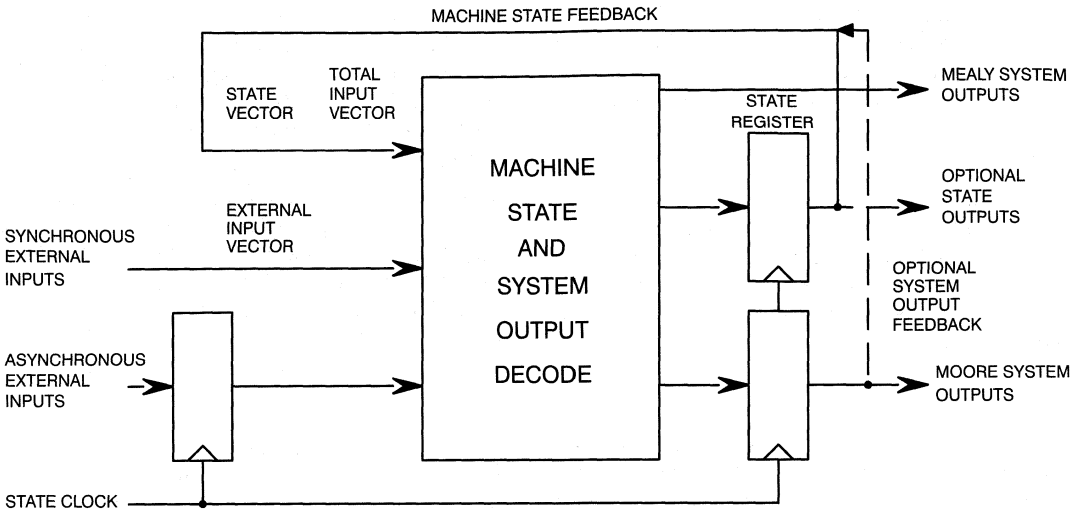


Figure 2. Synchronous State Machine Block Diagram

where x = 1, 2, or 3, representing the first, second, or third stage of the instruction execution and y = A or B, representing the first or second half of the execution stage.

Following this convention, the state machine's two free-running clocks are named CLK_A and CLK_B. These clocks run at half the state clock frequency and 180 degrees out of phase. The free-running clocks occur at the same time as their respective CLK_xA and CLK_xB clocks.

The major clock functions for this application are:

CLK_1B: The leading edge of this clock updates the instruction register.

CLK_2A: This clock's leading edge marks the start of ALU execution. The information on the ALU input bus clocks into the appropriate input registers at this time. The instruction cycle is considered recoverable up through and including CLK_2A (i.e., the status of the machine from the previous instruction has not been altered).

CLK_2B: Used to control the second half of the ALU execution stage, this clock initiates a write to RAM, triggers counters, gates ALU output into its latch, and clocks the ALU output information into any of the distributed destination registers.

CLK_3A: On this clock the memory address register can be updated. The ALU output bus status and ALU status is also clocked into the CPU status register.

Clock Generator Inputs

A set of inputs (external stimulus to the state machine) controls the state machine. The clock state machine described here has eight external inputs, including the state machine clock. These inputs are:

STATECLK: The state machine clock.

RESET: An asynchronous or synchronous reset input that can be connected directly to the state registers' preset or clear or to all clocked register inputs (D or T input). If connected to the preset or clear, RESET need not be synchronized. In this case, RESET forces the state machine into the machine's initial state, regardless of the present state. RESET can result from any combination of the following sources:

- Power up circuit (system reset)
- System controller software decodes system reset

- System controller software decodes module reset
- CPU software decodes module reset

RUN: This signal controls the start and stop sequence of the CPU clocks. In PIPELINE RUN mode, the start sequence generates the proper clock progression to fill up the pipeline registers, and the stop sequence empties the pipeline. RUN is externally manipulated to implement the single step and breakpoint functions.

NPL: Used to select NONPIPELINED RUN vs. PIPELINED RUN modes, this signal must be set to the selected mode prior to activating the RUN signal. Setting NPL = 1 selects NONPIPELINED RUN mode, and NPL = 0 selects PIPELINED RUN mode. The single step function operates properly in NONPIPELINED RUN mode only.

INTR: This signal indicates an external interrupt. When INTR is received, and IEN (interrupt enable, described below) is active, the CPU executes its interrupt handler. An interrupt inhibits the instruction register update clock (CLK_1B) and the ALU update clock (CLK_2B). CLK_1A for the interrupt instruction executes on the next cycle. The interrupt condition has priority over a wait condition and therefore starts generating clocks to permit execution of the interrupt instructions.

IEN: This interrupt enable signal qualifies INTR. IEN is likely to be a bit in the instruction word, allowing the user to define sections of un-interruptable code.

WAIT: The wait condition is initiated when both WAIT and WEN (wait enable, described below) are active. The CPU remains in the wait condition until WAIT goes inactive.

WEN: This wait enable signal qualifies WAIT for entrance into the wait condition. Like IEN, WEN is usually a bit in the instruction word, allowing the user to define sections of wait-sensitive code.

State Machine Partitioning

When architecting a state machine, it is generally a good practice to break up large machines into workable blocks, with each of the smaller machines containing states that require common inputs and generate common outputs. The example clock state machine is small enough to be designed as a single state machine, although it would be trivial to design logic to generate the free-running clocks as a separate machine from the rest of the clock state machine. Equations for the free-running clocks are:

$$CLK_A := \overline{RESET} \bullet \overline{CLK_A}$$

$$CLK_B := \overline{RESET} \bullet CLK_A$$

where ":=" indicates a registered output.

By examining these output equations, you can see that the free-running clocks have only two dependencies in common with the remaining portion of the clock state machine, i.e., RESET and STATECLK. The free-running clocks are required as inputs to the other state machine to synchronize the additional system outputs, however.

The example presented here implements the free-running clocks and the other system outputs within the same state definition. The resulting output equations can be verified against the equations for the free-running clocks alone.

The Initial Machine State

Regardless of the preferred state machine entry method, attacking the problem starts with defining the initial state of the machine. This initial state (INIT in the example) must be consistent with the power-on condition and/or an external input used to initialize the machine (RESET).

The state of the machine can be decoded from the present values of the system outputs, state registers, or a combination of the two. (The advantages and disadvantages of the state definition options will be discussed in greater detail later in this application note.) The initial machine state is generally, but not always, a decode of all 0s or all 1s. In the example design, INIT is the decode of all 0s.

Naming the States

With the exception of INIT, each state in the example design is named to indicate the active system clocks occurring during that state. For example, during state A, only CLK_A is active. Similarly,

state 123B has only CLK_1B, CLK_2B, CLK_3B, and CLK_B active. Additionally, an "N" suffix designates a nonpipelined state and a "W" suffix designates a wait condition state; this convention differentiates between states with identical active system outputs.

CPU Inactive States

The RESET input causes the state machine to enter the INIT state from any state in the machine. From the INIT state, the machine unconditionally starts to generate the free-running clocks. As shown in *Figure 3*, a line pointing from the INIT state to the A state, with a path equation equal to 1, indicates an unconditional branch. The state machine progression continues from the A state unconditionally into the B state. In the B state a multi-branch condition exists. If the RUN input remains inactive, then the A and B states continue to toggle, generating only the free-running clocks. Hence the INIT, A, and B states are referred to as "CPU inactive states."

Nonpipelined States

If the NPL input is active while the RUN input becomes active, the state machine operates in NONPIPELINED RUN mode and follows the model portrayed in *Figure 4*.

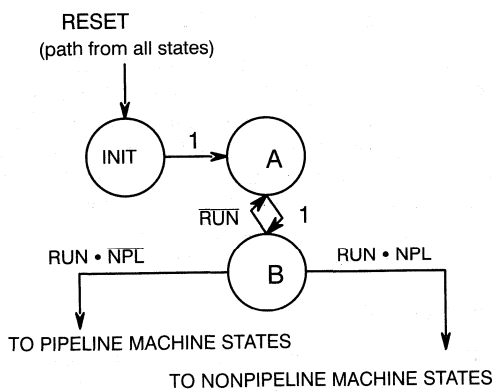


Figure 3. CPU Inactive States

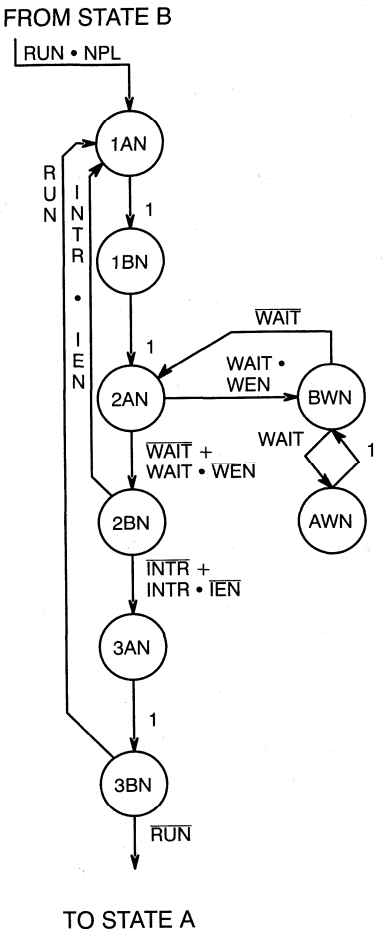


Figure 4. Non-Pipelined States

Pipelined States

If the NPL input is inactive when the RUN input goes active, thus indicating PIPELINED RUN mode, the state machine operates as depicted in *Figure 5*.

Unique States

When the RUN input goes active, the next state executed is either the 1A or the 1AN state, depending upon the value of the NPL input (refer to *Figures 4* and 5). Notice that the active system outputs in these two states are identical. Why generate two

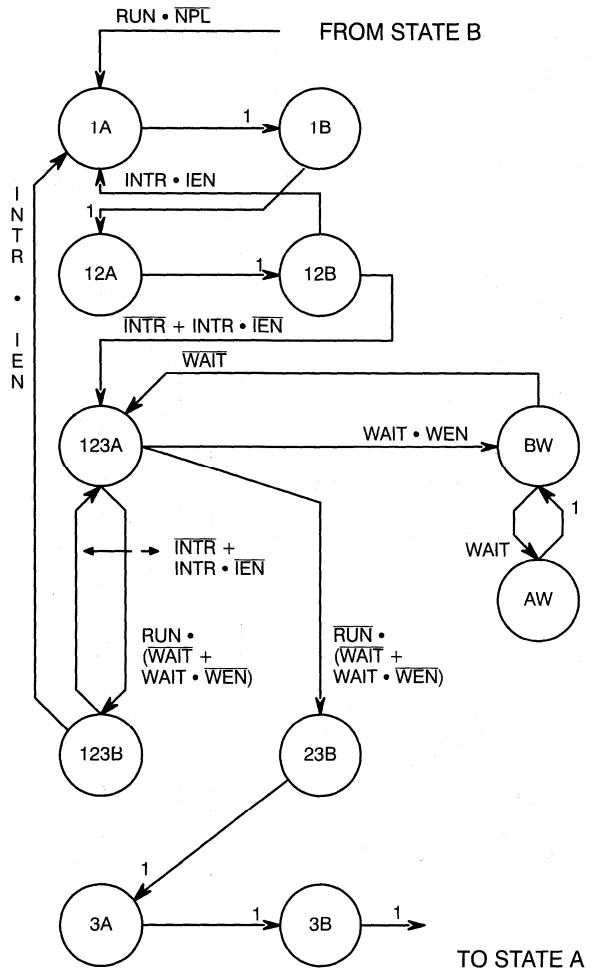


Figure 5. Pipelined States

identical states—when an additional state register might be required to differentiate between the states? (This assumes you use the system outputs to decode the machine's states.) The redundant states are not a problem because the additional state register needed to differentiate between the states is not an issue. There are two reasons for this. First, if you eliminate the redundant states, the state machine would require at least one additional state register anyway to differentiate between the B and the BW or BWN states, which would be needed without 1A and 1AN. (Separation of states BW and BWN from state B is required for correct functionality.) Second, adding another state only increases the number of state registers if the new total number of states exceeds an additional binary boundary (2, 4, 8, 16, ...). This is not a problem here.

You might also choose to widen your state machine (increase the number of state registers) to reduce

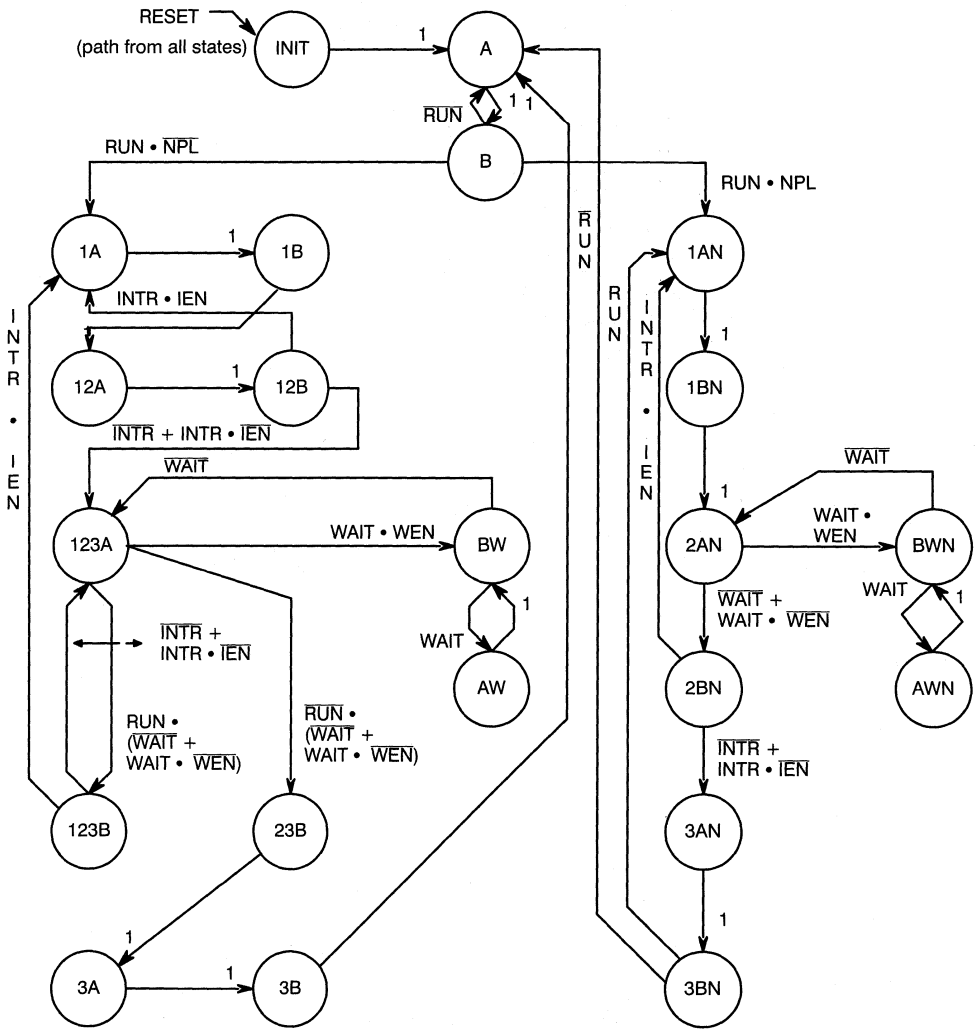


Figure 6. CPU Clock State Machine

the number of product terms to the state or system output registers. This decision should take into account the desired circuit implementation (PLDs, PROMS, discrete hardware, etc.) and is often an iterative process. In general, you can initially architect the state machine in the manner that is the easiest for you to understand, then make additional changes or small adjustments later if they become necessary.

State Description Verification

Now that all the pieces of the state machine are functionally defined (refer to *Figure 6* for the com-

pleted state diagram), consider methods for verifying the validity of the design. Some software you can use to describe and implement state machines would already offer verification at this point in a design. For other methods, read on!

One way to verify a state machine design is to recognize a rule of thumb: Out of every state, there should be a state path to another state for every possible combination of relevant external inputs. For example, there are two paths out of state 123B, with INTR and IEN as the relevant external inputs:

Path 1 = INTR • IEN

Path 2 = $\overline{\text{INTR}}$ + INTR • $\overline{\text{IEN}}$

If there are no known restrictions on the external inputs, a simple method of verifying the above rule of thumb is to generate an equation where all of the paths out of a state are ORed together as follows:

```
OUT_STATE_123B      = Path 1 + Path 2;
OUT_STATE_123B      = (INTR • IEN)
                    + INTR
                    + (INTR • IEN);
= OUT_STATE_123B    = 1
```

If the equation's terms equal 1 after Boolean reduction, then every state path out of the state is accounted for. The main advantage to this verification method is that you can easily do it using readily available Boolean reduction software.

If there are known restrictions to the external inputs, you can use this information to reduce the complexity of the machine. If it is impossible for the INTR • $\overline{\text{IEN}}$ condition to occur externally, for example, then you can leave this condition out of the Path 2 equation. In that case, the reduction of the OUT_STATE_123B equation yields a non-1 result.

Because the method of verification just described does not detect redundant path equations, it is useful to revise the original rule of thumb to: Out of every state, there should be one and only one state path to another state for every possible combination of relevant external inputs.

This revised condition is not as easily verified as the original statement. The easiest way to verify the more restrictive case is to simulate the state machine. To do this, you must generate a test vector for every possible external input that is relevant to each state simulated. Automatic test vector generation programs are available that produce every possible combination. After running the vectors against the design, you must visually inspect the output to verify that the machine never enters an illegal state.

System and State Register Output Generation

The model defining the clock state machine is complete, but there are still quite a few important decisions to be made regarding the final circuit implementation. Some of the major alternatives for final implementation are:

- System output vs. exclusive state register state decode
- D flip-flop vs. T flip-flop implementation
- PLD vs. PROM implementation

To gain some insight into these choices, consider how the output or feedback equations are assembled. Take, for example, the generation of CLK_3A using a D flip-flop (FF) implementation. By referring to *Figure 6*, you can find all the states in which CLK_3A is active. These are 123A, 3A, and 3AN. The CLK_3A output is generated by ORing the state decodes that, when ANDed with their respective state paths, advance the state machine into the three states listed above. Specifically:

```
CLK_3A :=
(Decode of 12B)•(INTR+INTR•IEN)     ;-123A
+(Decode of BW)•(WAIT)              ;-123A
+(Decode of 23B)•(1)                 ;-3A
+(Decode of 2BN)•(INTR+INTR•IEN)    ;-3AN
```

When you define the state decodes, the CLK_3A equations are completely specified in terms of the state machine inputs (state path), state registers, and/or system outputs (state decode). Typically, you then multiply the equation out to form a sum of products. This format provides for easy implementation in a PLD, which has a sum-of-products architecture, and also provides a useful foundation for further equation reduction.

State Decode

As discussed earlier, the next state of the machine can be decoded from the present values of the system outputs, the state registers, or a combination of the two. The choice typically comes down to weighing the maximum number of product terms verses the maximum number of flip-flops available in an implementation. For a Moore machine with registered system outputs, using the system outputs to uniquely define the states uses the smallest number of flip-flops to define the state machine. However, it is often necessary to add one or more state registers to uniquely define the states.

State assignment for this state decoding method is quite simple, but also rigidly defined, allowing limited flexibility when assigning the additional state registers. After reduction, the feedback and output equations of this "narrow" state machine might contain too many product terms to be implemented in a specific PLD, although product term complexity is never a problem with a PROM implementation.

Exclusive State Registers

Another consideration in state machine design is that you might be able to distribute the number of product terms more evenly among the equations implementing the state machine by using state registers exclusively to decode the states. Because the state decodes in the state registers can be selected to assist in Boolean reduction, proper state assignment enables the more complex equations to fit into a specific implementation.

This type of decode is useful in a PLD implementation, where there is a shortage of product terms for a specific state flip-flop, but extra flip-flops are available. Adding an extra state register can simplify the decode logic enough to fit the design in a single PLD.

The total number of exclusive state registers required to implement a state machine varies from a minimum of LOG(2)X (rounded up to the nearest integer) to a maximum of X, where X is the total number of states in the machine. You can iteratively change this number, along with the state assignment, to obtain a suitable solution.

The state assignment itself is a non-trivial issue, with almost limitless possibilities and no known method of obtaining the optimal solution. There are, however, some guidelines that can be used to obtain workable solutions:

1. Two or more states that potentially enter the same state with identical path equations should be adjacent (their binary codes differ in exactly one position). As an example, refer to *Figure* 5. States 12B and 123B both proceed into state 1A if the path condition INTR • IEN is true. When generating the CLK_1A equation, two of the terms of the equation look like this:

```
CLK_1A :=
   (Decode of 12B) • (INTR • IEN)          ;-1A
+ (Decode of 123B) • (INTR • IEN)          ;-1A
   . . .
```

If the decode of 12B and 123B differ in exactly one position, then Boolean reduction (which uses the $A \cdot B + \overline{A} \cdot B = B$ relationship) converts the two product terms into one smaller product term.

2. Two or more states that might proceed into different states with identical path equations, and an identical active output, should be adjacent. This situation occurs in the previous CLK_3A equation, shown again here:

```
CLK_3A :=
(Decode of 12B)•(INTR+INTR•IEN)      ;-123A
+(Decode of BW)•(WAIT)               ;-123A
+(Decode of 23B)•(1)                 ;-3A
+(Decode of 2BN)•(INTR+INTR•IEN);    -3AN
```

Note that if states 12B and 2BN are adjacent, then you can reduce the CLK_3A equation to three product terms.

Clock Generator Implementation

As mentioned earlier, there are many ways to implement state machines. The following sections discuss some of the pros and cons associated with some of the more common state machine implementations.

D Flip-Flop Implementation

There are more products available that support a D flip-flop solution than any other implementation. Therefore, it is usually the most cost-effective solution for a state machine.

Table 1 lists the number of product terms per output obtained by compiling the clock generator state machine definition with the LOG/iC software, using D flip-flops. The compiler input file appears in Appendix A. Optimizing the design (*Table 2*) significantly reduces the number of product terms needed.

Table 1. Optimized Results for Clock Generator: T Flip-Flop Implementation

LOG/iC Optimization Summary (FACT)				
CPU Time Quota per Function: 100 sec				
Function	**INV**	**P-Terms**	**CPU-Time**	**Flags**
CLK_1A.T	No	6	<1	
	Yes	7	1	
CLK_1B.T	No	4	1	
	Yes	3	1	
CLK_2A.T	No	5	1	
	Yes	4	<1	
CLK_2B.T	No	4	1	
	Yes	3	<1	
CLK_3A.T	No	5	<1	
	Yes	6	2	
CLK_3B.T	No	4	<1	
	Yes	2	<1	
CLK_A.T	No			C
	Yes			C
CLK_B.T	No	2	1	
	Yes	1	<1	
QQ1.T	No	3	<1	
	Yes	5	1	
QQ2.T	No	6	<1	
	Yes	11	2	

C: Constant function
FACT Minimization: 11 sec

Table 2. Non-optimized Results for Clock Generator: D Flip-Flop Implementation

Log/IC Optimization Summary (FACT)				
CPU Time Quota per Function: 100 sec				
Function	**INV**	**P-Terms**	**CPU-Time**	**Flags**
CLK_1A.D	No	12	<1	N
	Yes	27	<1	N
CLK_1B.D	No	5	<1	N
	Yes	34	1	N
CLK_2A.D	No	8	<1	N
	Yes	31	<1	N
CLK_2B.D	No	7	<1	N
	Yes	32	<1	N
CLK_3A.D	No	8	<1	N
	Yes	31	<1	N
CLK_3B.D	No	6	<1	N
	Yes	33	<1	N
CLK_A.D	No			NT
	Yes			NT
QQ1.D	No	6	<1	N
	Yes	5	<1	N
QQ2.D	No	10	<1	N
	Yes	9	<1	N

N: No Optimization
T: Trivial Function
FACT Minimization: 11 sec

T Flip-Flop Implementation

Even though D flip-flop solutions are more widely available, there are times when the logic needed for this implementation is prohibitively complex. Under these circumstances, a T flip-flop implementation might be more cost effective, because using T flip-flops reduces the logic significantly.

The best example of this situation is a simple synchronous binary counter. While the most significant bit (MSB) of an N-bit counter in a D flip-flop implementation requires N product terms, the T flip-flop solution requires only one product term. Note that the Cypress family of CY7C33x devices offers you a configurable T- or D-type implementation if you

place an XOR gate prior to the D flip-flop; route the AND/OR array to one of the XOR's inputs and the flip-flop's Q output (via an additional product term) to the other XOR input.

It isn't clear from simple observation, however, whether the T flip-flop implementation is beneficial for the clock generator state machine. One way to clarify this question is to change three command lines in the state machine description shown in Appendix A and recompile to produce a T flip-flop implementation. *Table 3* contains the product term results using T flip-flops. A quick study of the results reveals that the optimized version using D flip-flops (*Table 2*) requires fewer product terms than the T flip-flop version.

PLD Implementation

With the LOG/iC PLD Database option, the software assists in selecting a PLD, and it shows that the non-optimized version of the clock state machine fits in a PALC22V10 without further reduction. If the equations are reduced using Boolean reduction, however, a lower-cost solution is available. The results shown in *Table 3* indicate that the less expensive PALC20G10 would work. Appendix A shows the listing for the 20G10 LOG/iC implementation. Waveforms for the completed design appear in Appendix B. You can verify the CLK_A and CLK_B equation results against the equations generated in the State Machine Partitioning section of this application note.

PROM Implementation

You can obtain very high speed solutions by implementing state machines using PROMs. A PROM uses a look-up table to decode the machine's next state, as opposed to the AND/OR array in a PLD. The main advantage of using a look-up table to decode the next state is that every combination of the inputs can be decoded. Thus, you can create an extremely complex machine, without equation reductions.

The look-up table's drawback is that the PROM's depth grows exponentially (2N, where N = # of inputs to the look-up table) with every additional input to the look-up table. To determine the depth required, notice that the present total input vector provides the inputs to the look-up table. The clock generator state machine has seven external inputs, six system outputs, and two state outputs, which indicates a feasible implementation using the CY7C277 (32K x 8) registered PROM.

Table 3. Optimized Results for Clock Generator: D Flip-Flop Implementation

Log/IC Optimization Summary (FACT)				
CPU Time Quota per Function: 100 sec				
Function	**INV**	**P-Terms**	**CPU-Time**	**Flags**
CLK_1A.D	No	6	1	
	Yes	11	2	
CLK_1B.D	No	3	1	
	Yes	4	<1	
CLK_2A.D	No	4	1	
	Yes	7	<1	
CLK_2B.D	No	3	1	
	Yes	4	<1	
CLK_3A.D	No	4	1	
	Yes	9	1	
CLK_3B.D	No	3	<1	
	Yes	3	1	
CLK_A.D	No	1	<1	
	Yes	2	<1	
CLK_B.D	No	1	1	
	Yes	2	<1	
QQ1.D	No	3	<1	
	Yes	3	1	
QQ2.D	No	6	16	
	Yes	6	2	

FACT Minimization: 29 sec

Using a registered PROM such as the CY7C277 to implement the machine also helps to reduce the parts count, because the PROM implements both the state and system output registers. LOG/iC offers support for implementing state machines in PROMs, and only a few minor changes to the state machine description shown in Appendix A are re-

quired. *PROM replaces the *PAL command, some simple statements indicating the CY7C277 architecture (INPUTS = 15 AND OUTPUTS = 8) replaces the TYPE = statement, and PROGFORMAT = INTEL–HEX.

CY7C361 Implementation

A new way to obtain high-speed operation of state machines became available with Cypress Semiconductor's development of a revolutionary architecture that enables a CMOS PLD state machine part to operate at speeds in the 125-MHz range. The first part in this family is the CY7C361. The architectural innovations used to obtain 125-MHz operation require that you approach state machine design slightly differently than you would when designing with traditional PLD architectures.

To fully understand the information in this section, consult the Cypress Semiconductor application note titled "Understanding the CY7C361."

Using the clock generator state machine example, this section shows how you can generate a state diagram for the CY7C361 by following some simple rules. This diagram allows you to determine whether the design can fit in a CY7C361. The rule of thumb is that a state diagram with 32 or fewer state nodes will probably fit. (The likelihood of the implementation at that point depends totally upon split-input-array fitting issues.) You can convert the state diagram directly into Boolean equations (with no Boolean reduction required) and compile the equations into JEDEC code for the final implementation.

The CY7C361's condition-decode array has been optimized for use in state machine applications. As shown in *Figure 7*, the CY7C361 condition decoder contains the necessary logic to generate two kinds of state machine operations. The Entering a State operation should look familiar. The process used to generate the system output and state register equations (in the System and State Register Output Generation section of this application note) utilizes a similar equation form. There are two small differences, though.

First, the Entering a State equation shown in *Figure 7* assumes the present state conditions are available as single entities on the input array. That is, one state register uniquely defines each state, and therefore the present state is not encoded using multiple flip-flops, as is typical in traditional state machines. There is a special case, however, that allows you to encode the states into 32 or fewer state registers (providing a maximum total of 232 states) so long as only one product term per state register is required in the decode of the next state. An example of this is a simple synchronous 32-bit binary counter using the Toggle (or T flip-flop) configuration.

The second design difference with the CY7C361 is that the Entering a State equation also shows all states (SA, SB, SC) that have an identical state path to S0. This is not necessarily the case when designing with traditional PLDs (as shown in *Figure 6*). The CY7C361, however, requires a machine definition in which all state paths into any given state are identical. You can easily convert an existing state diagram and satisfy the new condition by simply adding additional states for those states that do not meet the above condition.

To remain consistent with the naming conventions already defined for the clock generator example, two additional suffixes, "X" and "Y", indicate the additional states. For example, state BW in *Figure 6* has two state paths entering into it: WAIT • WEN from state 123A and a path from state AW. To meet

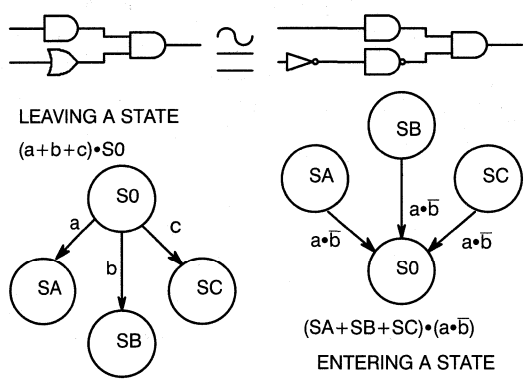


Figure 7. The Condition Decoder – Optimized for Two State Machine Operations

the design conditions for the CY7C361, you add an additional state, BWX, such that state AW enters BWX with a state path of 1, and state 123A enters state BW with a state path of WAIT • WEN.

The CY7C361 implementation of the clock generator state machine appears in *Figure 8*. Note that both of the new states (BW and BWX) have exits with the same state path equation. Thus, the number of states in the state machine does not grow geometrically due to this new methodology.

In addition to the normal Entering a State equation, the CY7C361 supports operations in which multiple state paths go from one state to another, and each state path term contains only one input. *Figure 9* shows a diagram of this condition.

Another operation, called Leaving a State, proves especially useful in conjunction with the Wait-Until-Terminate state macrocell configuration in the CY7C361. The flip-flop in this macrocell configuration is unconditionally set by the active previous state macrocell. The flip-flop remains set until the condition decoder equation (a Leaving a State equation) for the Terminate macrocell goes active. *Figure 10* shows how the Terminate configuration looks within a state diagram.

When implementing the clock generator state machine in the CY7C361 using the conversion techniques discussed above, the number of states slightly

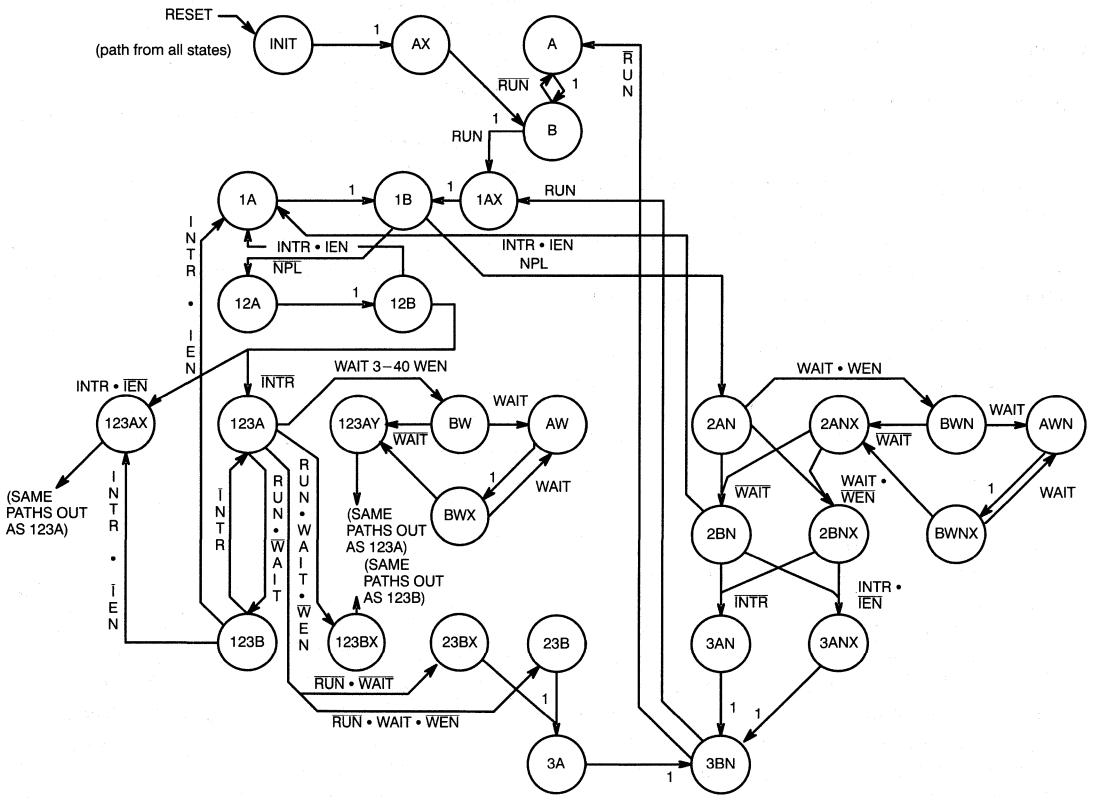
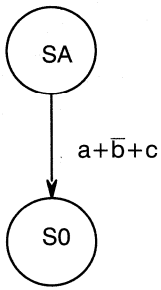


Figure 8. CY7C361 Implementation, CPU Clock State Machine



ENTERING A STATE
EQ. # 2
(SA)(a+b̄+c)

Figure 9. Entering a State Along Multiple Paths

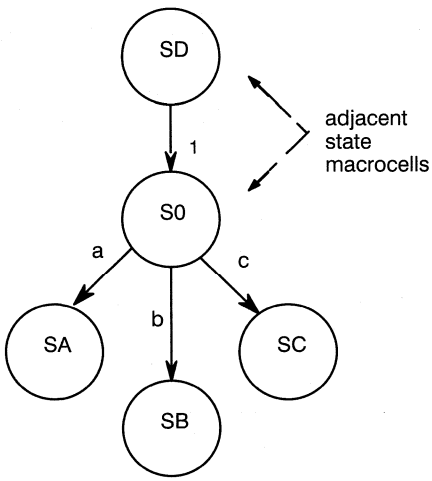


Figure 10. Leaving a State (Terminate Configuration)

exceeds 32. But by allowing the machine's pipelined and nonpipelined portions to share common states, (1A, 1B, and 3B) the total number of states reduces to less than 32.

Note that you can use this same kind of state reduction for the original implementation (refer to the Unique States section). *Figure 8* shows the resulting state diagram.

It is a simple matter to convert information from the state diagram to PLD ToolKit™ equations (refer to Appendix C for the PLD ToolKit source file). You must generate an Entering a State equation for every state node in the diagram. (The Terminate configuration was not used in this example, but it can be useful for implementing wait states.)

You generate the equations in Appendix C using the <PROD> and <INV_PROD> connectives for the AND and NAND terms, respectively. Then generate the system outputs by ORing the appropriate states in the OR-based output array. For example, the CLK_1B output is active during the 1B, 12B, 123B, or 123BX states. The PLD ToolKit connective for the OR array is <INV_SUM>. The CY7C361 implementation of the clock generator state machine was simulated using the PLD ToolKit (see Appendix D).

Reference

1. Donald D. Givone, Introduction to Switching Circuit Theory (New York: McGraw-Hill, Inc., 1970)

Appendix A. LOG/iC PLD Source Code: Clock State Machine

```
LOG/iC-PAL              Rel 3.2/2-2328-1721/00034 #32-5955 90/03/15  23:49:45
------------------------------------------------------------------------------
LOG/iC - COPYRIGHT (C) 1985,1988 BY ISDATA GMBH, 7500 KARLSRUHE WEST-GERMANY
Cypress Semiconductor                                  LICENCE FOR IBM-PC/XT/AT
Data Set: OD20G10.DCB
  1     1: *IDENTIFICATION
  2     2: PIPELINED CLOCKING SYSTEM OD20G10
  3     3: ERIC B.  ROSS
  4     4: CYPRESS SEMICONDUCTOR
  5     5: NAMING CONVENTION
  6     6:  OD      = SYSTEM OUTPUTS ARE DFLOPS AND ARE USED FOR STATE DEF
  7     7:  20G10   = PALC20G10 IMPLEMENTATION
  8     8: *PAL
  9     9:  TYPE=PALC20G10
 10 I  10:
 11    11: *X-NAMES
 12 I  12:
;-------------------------------------------------------------------------
 13 I  13: ;INPUT DEFINITIONS :
 14 I  14: ; RUN    = START & STOP EXECUTION OF OUTPUT CLOCKS (NORMAL, SINGLE
 15 I  15: ;          STEP, & BREAK PT.  EXECUTION
 16 I  16: ; NPL    = PIPELINED VS NON-PIPELINED MODE OF EXECUTION
 17 I  17: ; INTR   = EXTERNAL INTERRUPT CONDITION (TLB MISS, PARITY ERROR,...)
 18 I  18: ; IEN    = INTERRUPT ENABLE
 19 I  19: ; WAIT   = WAIT ENABLE (CACHE MISS)
 20 I  20: ; WEN    = WAIT ENABLE
 21 I  21:
;-------------------------------------------------------------------------
 22 I  22: ;
 23    23:  RUN, NPL, INTR, IEN, WAIT, WEN, RESET;
 24 I  24:
 25    25: *Z-NAMES
 26 I  26:
;-------------------------------------------------------------------------
 27 I  27: ;OUTPUT DEFINITIONS :
 28 I  28: ;
 29 I  29: ; 3 CLOCK STAGES 1, 2, 3
 30 I  30: ; 2 CLOCKS PER STATE A, B
 31 I  31: ;    CLK_XX WHERE XX = 1A,1B,2A,2B,3A,3B
 32 I  32: ;
 33 I  33: ; 2 FREE RUNNING CLOCKS
 34 I  34: ;    CLK_A, CLK_B
 35 I  35: ;
 36 I  36: ; ADDITIONAL REGISTERS FOR STATE DEFINITION
 37 I  37: ;    QQ1, QQ2
 38 I  38:
;-------------------------------------------------------------------------
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
 39 I  39: ;
 40    40: CLK_1A, CLK_1B, CLK_2A, CLK_2B, CLK_3A, CLK_3B, CLK_A, CLK_B, QQ1,
QQ2;
 41 I  41:
 42    42: *Z-VALUES
 43 I  43:
 44 I  44: ;                                ADDITIONAL OUTPUTS
 45 I  45: ;       SYSTEM OUTPUTS           FOR STATE DEFINITION
 46 I  46: ;       ______________           ____________________
 47 I  47: ;
 48 I  48: ;       C C C C C C C C   Q Q
 49 I  49: ;       L L L L L L L L   Q Q
 50 I  50: ;       K K K K K K K K   1 2
 51 I  51: ;       1 1 2 2 3 3 A B
 52 I  52: ;       A B A B A B
 53 I  53:
 54    54:  S1   = 0 0 0 0 0 0 0 0   - -          ; INIT  COMMON STATES
 55    55:  S2   = 0 0 0 0 0 0 1 0   0 -          ; SA       - INACTIVE
 56    56:  S3   = 0 0 0 0 0 0 0 1   0 -          ; SB         MODE STATES
 57 I  57:
 58    58:  S4   = 1 0 0 0 0 0 1 0   - 0          ; S1A    PIPELINE STATES
 59    59:  S5   = 0 1 0 0 0 0 0 1   - 0          ; S1B
 60    60:  S6   = 1 0 1 0 0 0 1 0   - -          ; S12A
 61    61:  S7   = 0 1 0 1 0 0 0 1   - -          ; S12B
 62    62:  S8   = 1 0 1 0 1 0 1 0   - -          ; S123A
 63    63:  S9   = 0 1 0 1 0 1 0 1   - -          ; S123B
 64    64:  S10  = 0 0 0 1 0 1 0 1   - -          ; S23B 65    65:  S11 = 0 0 0
0 1 0 1 0   - 0             ; S3A
 66    66:  S12  = 0 0 0 0 0 1 0 1   - 0          ; S3B
 67    67:  S13  = 0 0 0 0 0 0 1 0   1 0          ; SAW
 68    68:  S14  = 0 0 0 0 0 0 0 1   1 0          ; SBW
 69 I  69:
 70    70:  S15  = 1 0 0 0 0 0 1 0   - 1          ; S1AN   NON-PIPELINE
 71    71:  S16  = 0 1 0 0 0 0 0 1   - 1          ; S1BN
 72    72:  S17  = 0 0 1 0 0 0 1 0   - -          ; S2AN
 73    73:  S18  = 0 0 0 1 0 0 0 1   - -          ; S2BN
 74    74:  S19  = 0 0 0 0 1 0 1 0   - 1          ; S3AN
 75    75:  S20  = 0 0 0 0 0 1 0 1   - 1          ; S3BN
 76    76:  S21  = 0 0 0 0 0 0 1 0   1 1          ; SAWN
 77    77:  S22  = 0 0 0 0 0 0 0 1   1 1          ; SBWN
 78 I  78:
 79    79: *STRING
 80    80:  INIT    = 1                           ;  COMMON STATES
 81    81:  SA      = 2                           ;   -INACTIVE MODE
 82    82:  SB      = 3                           ;    STATES
 83 I  83:
 84    84:  S1A     = 4                           ;  PIPELINE STATES
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
 85     85:  S1B      = 5                                ;
 86     86:  S12A     = 6                                ;
 87     87:  S12B     = 7                                ;
 88     88:  S123A    = 8                                ;
 89     89:  S123B    = 9                                ;
 90     90:  S23B     = 10                               ;
 91     91:  S3A      = 11                               ;
 92     92:  S3B      = 12                               ;
 93     93:  SAW      = 13                               ;
 94     94:  SBW      = 14                               ;
 95 I   95: 96    96:  S1AN     = 15                              ;  NON-PIPELINE
 97     97:  S1BN     = 16                               ;
 98     98:  S2AN     = 17                               ;
 99     99:  S2BN     = 18                               ;
100    100:  S3AN     = 19                               ;
101    101:  S3BN     = 20                               ;
102    102:  SAWN     = 21                               ;
103    103:  SBWN     = 22                               ;
104    104:  LASTSTATE = 22;
105 I 105:
106    106: *FLOW-TABLE
107 I 107: ;
108 I 108:
;-------------------------------------------------------------------------
109 I 109: ;RESET STATE
110 I 110: ;ALL STATES MUST RESET TO THE INITIAL STATE (ALL OUTPUTS REGISTERS 0
UPON
111 I 111: ;AN ACTIVE RESET INPUT.  SINCE THE 20G10 HAS NO GLOBAL OR INDIVIDUAL
112 I 112: ;RESETS TO THE OUTPUT REGISTERS, RESET TO INITIAL STATE MUST BE EM-
BEDDED
113 I 113: ;INTO THE STATE MACHINE
114 I 114: ;
115    115: RELEVENT = RESET                            ;
116    116: S[1..'LASTSTATE'], X 1      , F 'INIT'   ;ALL STATE > INIT UPON RESET
117    138: RELEVENT = RESET = 0                       ;
118 I 139: ;
119 I 140:
;-------------------------------------------------------------------------
120 I 141: ;INACTIVE MODE STATES
121    142: RELEVANT = RUN, NPL                        ;
122    143: S 'INIT'    , X - -         , F 'SA'     ;INITIAL STATE AFTER RESET
123 I 144:
124    145: S 'SA'      , X - -         , F 'SB'     ;INACTIVE MODE STATE, ONLY
125 I 146:
126    147: S 'SB'      , X 0 -         , F 'SA'     ;FREE RUN CLKS A & B ARE AC-
TIVE
127    148:               X 1 0         , F 'S1A'    ; PIPELINE VS.
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
128   149:              X 1 1          , F 'S1AN'   ; NON-PIPELINE DECISION
129 I 150:
130 I 151:
;-------------------------------------------------------------------------
131 I 152: ;PIPELINE MODE STATES
132 I 153:
133   154: RELEVANT = INTR, IEN                     ;*PRIMING THE PIPELINE*
134   155: S 'S1A'    , X - -          , F 'S1B'    ;
135 I 156:
136   157: S 'S1B'    , X - -          , F 'S12A'   ;
137 I 158:
138   159: S 'S12A'   , X - -          , F 'S12B'   ;
139 I 160:
140   161: S 'S12B'   , X 1 1          , F 'S1A'    ;   INTERRUPT CONDITION ? YES
141   162:             X 1 0          , F 'S123A'  ;                           NO
142   163:             X 0 -          , F 'S123A'  ;                           NO
143 I 164:
144   165: RELEVANT =  RUN, INTR, IEN, WAIT, WEN; *FULL PIPELINE*
145   166: S 'S123A'  , X - - - 1 1 , F 'SBW'      ;   WAIT CONDITION
146   167:             X 0 - - 0 - , F 'S23B'     ;   /RUN COND., EMPTY PIPELIN
147   168:             X 0 - - 1 0 , F 'S23B'     ;   /RUN COND., EMPTY PIPELIN
148   169:             X 1 - - 0 - , F 'S123B'    ;   RUN CONDITION
149   170:             X 1 - - 1 0 , F 'S123B'    ;   RUN CONDITION
150 I 171:
151   172: S 'S123B'  , X - 1 1 - - , F 'S1A'      ;   INTERUPT CONDITION
152   173:             X - 0 - - - , F 'S123A'    ;   RUN CONDITION
153   174:             X - 1 0 - - , F 'S123A'    ;   RUN CONDITION
154 I 175:
155   176: RELEVANT = RUN                           ; *EMPTY PIPELINE*
156   177: S 'S23B'   , X -            , F 'S3A'    ;
157 I 178:
158   179: S 'S3A'    , X -            , F 'S3B'    ;
159 I 180:
160   181: S 'S3B'    , X -            , F 'SA'     ;   BACK TO INACTIVE STATE
161 I 182:
162   183: RELEVANT = WAIT                          ; *PIPELINE WAIT STATES*
163   184: S 'SBW'    , X 1            , F 'SAW'    ;   WAIT
164   185:             X 0            , F 'S123A'  ;   /WAIT
165 I 186:
166   187: S 'SAW'    , X -            , F 'SBW'    ;
167 I 188:
168 I 189:
;-------------------------------------------------------------------------
169 I 190: ;NON-PIPELINE MODE STATES
170 I 191:
171   192: S 'S1AN'   , X -            , F 'S1BN'   ;
172 I 193:
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
173    194: S 'S1BN'    , X -           , F 'S2AN'   ;
174  I 195:
175    196: RELEVANT = WAIT, WEN                    ;
176    197: S 'S2AN'    , X 1 1         , F 'SBWN'   ; WAIT CONDITION
177    198:               X 0 -         , F 'S2BN'   ; /WAIT CONDITION
178    199:               X 1 0         , F 'S2BN'   ; /WAIT CONDITION
179  I 200:180    201: RELEVANT = INTR, IEN                    ;
181    202: S 'S2BN'    , X 1 1         , F 'S1AN'   ; INTERRUPT CONDITION
182    203:               X 0 -         , F 'S3AN'   ; /INTERRUPT CONDITION
183    204:               X 1 0         , F 'S3AN'   ; /INTERRUPT CONDITION
184  I 205:
185    206: RELEVANT = RUN                          ;
186    207: S 'S3AN'    , X -           , F 'S3BN'   ;
187  I 208:
188    209: S 'S3BN'    , X 1           , F 'S1AN'   ;
189    210:               X 0           , F 'SA'     ; BACK TO INACTIVE STATE
190  I 211:
191    212: RELEVANT = WAIT                         ;*NON-PIPELINED WAIT STATES*
192    213: S 'SBWN'    , X 1           , F 'SAWN'   ;  REMAIN IN WAIT
193    214:               X 0           , F 'S2AN'   ;  END OF WAIT CONDITION
194  I 215:
195    216: S 'SAWN'    , X -           , F 'SBWN'   ;  REMAIN IN WAIT
196  I 217:
197    218: *STATE-ASSIGNMENT
198    219:  Z-VALUES
199  I 220:
200  I 221:
201    222: *PIN
202    223:  STATECLK = 1, RUN = 2, NPL = 3, INTR = 4, IEN = 5, WAIT = 6, WEN =
7,
203    223:  RESET = 8, CLK_1A = 14, CLK_1B = 15, CLK_2A = 16, CLK_2B = 17,
204    223:  CLK_3A = 18, CLK_3B = 19, CLK_A = 20, CLK_B = 21, QQ1 = 22, QQ2 =
23;
205  I 224:
206    225: *RUN-CONTROL207   226:  LISTING= LONG,SYMBOL-TABLE,EQUATIONS,PIN-
OUT;208   227:  PROGFORMAT= L-EQUATIONS
209    228:  OPTIMIAZATION= P-TERMS;
210    229: *END
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

LOG/IC SYMBOL TABLE				
SYMBOL	TYPE	REG	LEVEL	PIN/NODE
GND	LOCAL	-	HIGH	
VCC	LOCAL	-	HIGH	
RUN	X-VARIABLE	-	HIGH	2
NPL	X-VARIABLE	-	HIGH	3
INTR	X-VARIABLE	-	HIGH	4
IEN	X-VARIABLE	-	HIGH	5
WAIT	X-VARIABLE	-	HIGH	6
WEN	X-VARIABLE	-	HIGH	7
RESET	X-VARIABLE	-	HIGH	8
CLK_1A	X-VARIABLE	-	HIGH	14
CLK_1B	X-VARIABLE	-	HIGH	15
CLK_2A	X-VARIABLE	-	HIGH	16
CLK_2B	X-VARIABLE	-	HIGH	17
CLK_3A	X-VARIABLE	-	HIGH	18
CLK_3B	X-VARIABLE	-	HIGH	19
CLK_A	X-VARIABLE	-	HIGH	20
CLK_B	X-VARIABLE	-	HIGH	21
QQ1	X-VARIABLE	-	HIGH	22
QQ2	X-VARIABLE	-	HIGH	23
CLK_1A.D	Z-VARIABLE	DFF	HIGH	14
CLK_1B.D	Z-VARIABLE	DFF	HIGH	15
CLK_2A.D	Z-VARIABLE	DFF	HIGH	16
CLK_2B.D	Z-VARIABLE	DFF	HIGH	17
CLK_3A.D	Z-VARIABLE	DFF	HIGH	18
CLK_3B.D	Z-VARIABLE	DFF	HIGH	19
CLK_A.D	Z-VARIABLE	DFF	HIGH	20
CLK_B.D	Z-VARIABLE	DFF	HIGH	21
QQ1.D	Z-VARIABLE	DFF	HIGH	22
QQ2.D	Z-VARIABLE	DFF	HIGH	23

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
EXPANDED FUNCTION TABLE (INCLUDING LOCAL VARIABLES):
----------------------------------------------------
                                :  CCCC CC
                                :  LLLL LLCC
              CCC CCC           :  KKKK KKLL
             RLLL LLLC C        :  ____ __KK QQ
      I W    EKKK KKKL L        :  1122 33__ QQ
  GVRN NIAW  S___ ___K KQQ      :  ABAB ABAB 12
  NCUP TEIE  E112 233_ _QQ      :  ....  ....  ..
  DCNL RNTN  TABA BABA B12      :  DDDD DDDD DD
  ---------------------------------------
  ---- ---- 1000 0000 0--   :  0000 0000 --;     1/  116
  ---- ---- 0000 0000 0--   :  0000 0010 0-;     2/  143
  ---- ---- 1000 0001 00-   :  0000 0000 --;     3/  117
  ---- ---- 0000 0001 00-   :  0000 0001 0-;     4/  145
  ---- ---- 1000 0000 10-   :  0000 0000 --;     5/  118
  --0- ---- 0000 0000 10-   :  0000 0010 0-;     6/  147
  --10 ---- 0000 0000 10-   :  1000 0010 -0;     7/  148
  --11 ---- 0000 0000 10-   :  1000 0010 -1;     8/  149
  ---- ---- 1100 0001 0-0   :  0000 0000 --;     9/  119
  ---- ---- 0100 0001 0-0   :  0100 0001 -0;    10/  155
  ---- ---- 1010 0000 1-0   :  0000 0000 --;    11/  120
  ---- ---- 0010 0000 1-0   :  1010 0010 --;    12/  157
  ---- ---- 1101 0001 0--   :  0000 0000 --;    13/  121
  ---- ---- 0101 0001 0--   :  0101 0001 --;    14/  159
  ---- ---- 1010 1000 1--   :  0000 0000 --;    15/  122
  ---- 11-- 0010 1000 1--   :  1000 0010 -0;    16/  161
  ---- 10-- 0010 1000 1--   :  1010 1010 --;    17/  162
  ---- 0--- 0010 1000 1--   :  1010 1010 --;    18/  163
  ---- ---- 1101 0101 0--   :  0000 0000 --;    19/  123
  ---- --11 0101 0101 0--   :  0000 0001 10;    20/  166
  --0- --0- 0101 0101 0--   :  0001 0101 --;    21/  167
  --0- --10 0101 0101 0--   :  0001 0101 --;    22/  168
  --1- --0- 0101 0101 0--   :  0101 0101 --;    23/  169
  --1- --10 0101 0101 0--   :  0101 0101 --;    24/  170
  ---- ---- 1010 1010 1--   :  0000 0000 --;    25/  124
  ---- 11-- 0010 1010 1--   :  1000 0010 -0;    26/  172
  ---- 0--- 0010 1010 1--   :  1010 1010 --;    27/  173
  ---- 10-- 0010 1010 1--   :  1010 1010 --;    28/  174
  ---- ---- 1000 1010 1--   :  0000 0000 --;    29/  125
  ---- ---- 0000 1010 1--   :  0000 1010 -0;    30/  177
  ---- ---- 1000 0101 0-0   :  0000 0000 --;    31/  126
  ---- ---- 0000 0101 0-0   :  0000 0101 -0;    32/  179
  ---- ---- 1000 0010 1-0   :  0000 0000 --;    33/  127
  ---- ---- 0000 0010 1-0   :  0000 0010 0-;    34/  181
  ---- ---- 1000 0001 010   :  0000 0000 --;    35/  128
  ---- ---- 0000 0001 010   :  0000 0001 10;    36/  187
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
---- ---- 1000 0000 110  :  0000 0000 --;     37/  129
---- --1- 0000 0000 110  :  0000 0010 10;     38/  184
---- --0- 0000 0000 110  :  1010 1010 --;     39/  185
---- ---- 1100 0001 0-1  :  0000 0000 --;     40/  130
---- ---- 0100 0001 0-1  :  0100 0001 -1;     41/  192
---- ---- 1010 0000 1-1  :  0000 0000 --;     42/  131
---- ---- 0010 0000 1-1  :  0010 0010 --;     43/  194
---- ---- 1001 0001 0--  :  0000 0000 --;     44/  132
---- --11 0001 0001 0--  :  0000 0001 11;     45/  197
---- --0- 0001 0001 0--  :  0001 0001 --;     46/  198
---- --10 0001 0001 0--  :  0001 0001 --;     47/  199
---- ---- 1000 1000 1--  :  0000 0000 --;     48/  133
---- 11-- 0000 1000 1--  :  1000 0010 -1;     49/  202
---- 0--- 0000 1000 1--  :  0000 1010 -1;     50/  203
---- 10-- 0000 1000 1--  :  0000 1010 -1;     51/  204
---- ---- 1000 0101 0-1  :  0000 0000 --;     52/  134
---- ---- 0000 0101 0-1  :  0000 0101 -1;     53/  207
---- ---- 1000 0010 1-1  :  0000 0000 --;     54/  135
--1- ---- 0000 0010 1-1  :  1000 0010 -1;     55/  209
--0- ---- 0000 0010 1-1  :  0000 0010 0-;     56/  210
---- ---- 1000 0001 011  :  0000 0000 --;     57/  136
---- ---- 0000 0001 011  :  0000 0001 11;     58/  216
---- ---- 1000 0000 111  :  0000 0000 --;     59/  137
---- --1- 0000 0000 111  :  0000 0010 11;     60/  213
---- --0- 0000 0000 111  :  0010 0010 --;     61/  214
REST                     :  ---- ---- --;     62
----------------------------------------
1234 5678 9012 3456 789     1234 5678 90
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
  STATE ASSIGNMENT:
-----------------
    CCCC CC
    LLLL LLCC
    KKKK KKLL
    _____ __KK QQ
    1122 33__ QQ
    ABAB ABAB 12
    ------------
    0000 0000 --; 1
    0000 0010 0-; 2
    0000 0001 0-; 3
    1000 0010 -0; 4
    0100 0001 -0; 5
    1010 0010 --; 6
    0101 0001 --; 7
    1010 1010 --; 8
    0101 0101 --; 9
    0001 0101 --; 10
    0000 1010 -0; 11
    0000 0101 -0; 12
    0000 0010 10; 13
    0000 0001 10; 14
    1000 0010 -1; 15
    0100 0001 -1; 16
    0010 0010 --; 17
    0001 0001 --; 18
    0000 1010 -1; 19
    0000 0101 -1; 20
    0000 0010 11; 21
    0000 0001 11; 22
EXPANDED FUNCTION TABLE (LOCAL VARIABLES REMOVED):
--------------------------------------------------
                           :  CCCC CC
                           :  LLLL LLCC
           C CCCC C        :  KKKK KKLL
          RL LLLL LCC      :  _____ __KK QQ
      I  W EK KKKK KLL     :  1122 33__ QQ
    RNNI AWS_ _____ _KKQ Q :  ABAB ABAB 12
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
    UPTE IEE1 1223 3__Q Q  :  ....  ....  ..
    NLRN TNTA BABA BAB1 2  :  DDDD DDDD DD
    ---------------------------------------
    ---- --10 0000 000- -  :  0000 0000 --;     1/  116
    ---- --00 0000 000- -  :  0000 0010 0-;     2/  143
    ---- --10 0000 0100 -  :  0000 0000 --;     3/  117
    ---- --00 0000 0100 -  :  0000 0001 0-;     4/  145
    ---- --10 0000 0010 -  :  0000 0000 --;     5/  118
    0--- --00 0000 0010 -  :  0000 0010 0-;     6/  147
    10-- --00 0000 0010 -  :  1000 0010 -0;     7/  148
    11-- --00 0000 0010 -  :  1000 0010 -1;     8/  149
    ---- --11 0000 010- 0  :  0000 0000 --;     9/  119
    ---- --01 0000 010- 0  :  0100 0001 -0;    10/  155
    ---- --10 1000 001- 0  :  0000 0000 --;    11/  120
    ---- --00 1000 001- 0  :  1010 0010 --;    12/  157
    ---- --11 0100 010- -  :  0000 0000 --;    13/  121
    ---- --01 0100 010- -  :  0101 0001 --;    14/  159
    ---- --10 1010 001- -  :  0000 0000 --;    15/  122
    --11 --00 1010 001- -  :  1000 0010 -0;    16/  161
    --10 --00 1010 001- -  :  1010 1010 --;    17/  162
    --0- --00 1010 001- -  :  1010 1010 --;    18/  163
    ---- --11 0101 010- -  :  0000 0000 --;    19/  123
    ---- 1101 0101 010- -  :  0000 0001 10;    20/  166
    0--- 0-01 0101 010- -  :  0001 0101 --;    21/  167
    0--- 1001 0101 010- -  :  0001 0101 --;    22/  168
    1--- 0-01 0101 010- -  :  0101 0101 --;    23/  169
    1--- 1001 0101 010- -  :  0101 0101 --;    24/  170
    ---- --10 1010 101- -  :  0000 0000 --;    25/  124
    --11 --00 1010 101- -  :  1000 0010 -0;    26/  172
    --0- --00 1010 101- -  :  1010 1010 --;    27/  173
    --10 --00 1010 101- -  :  1010 1010 --;    28/  174
    ---- --10 0010 101- -  :  0000 0000 --;    29/  125
    ---- --00 0010 101- -  :  0000 1010 -0;    30/  177
    ---- --10 0001 010- 0  :  0000 0000 --;    31/  126
    ---- --00 0001 010- 0  :  0000 0101 -0;    32/  179
    ---- --10 0000 101- 0  :  0000 0000 --;    33/  127
    ---- --00 0000 101- 0  :  0000 0010 0-;    34/  181
    ---- --10 0000 0101 0  :  0000 0000 --;    35/  128
    ---- --00 0000 0101 0  :  0000 0001 10;    36/  187
    ---- --10 0000 0011 0  :  0000 0000 --;    37/  129
    ---- 1-00 0000 0011 0  :  0000 0010 10;    38/  184
    ---- 0-00 0000 0011 0  :  1010 1010 --;    39/  185
    ---- --11 0000 010- 1  :  0000 0000 --;    40/  130
EXPANDED FUNCTION TABLE (LOCAL VARIABLES REMOVED)- continued :
    ---- --01 0000 010- 1  :  0100 0001 -1;    41/  192
    ---- --10 1000 001- 1  :  0000 0000 --;    42/  131
    ---- --00 1000 001- 1  :  0010 0010 --;    43/  194
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
      ---- --10 0100 010- -  :  0000 0000 --;    44/  132
      ---- 1100 0100 010- -  :  0000 0001 11;    45/  197
      ---- 0-00 0100 010- -  :  0001 0001 --;    46/  198
      ---- 1000 0100 010- -  :  0001 0001 --;    47/  199
      ---- --10 0010 001- -  :  0000 0000 --;    48/  133
      --11 --00 0010 001- -  :  1000 0010 -1;    49/  202
      --0- --00 0010 001- -  :  0000 1010 -1;    50/  203
      --10 --00 0010 001- -  :  0000 1010 -1;    51/  204
      ---- --10 0001 010- 1  :  0000 0000 --;    52/  134
      ---- --00 0001 010- 1  :  0000 0101 -1;    53/  207
      ---- --10 0000 101- 1  :  0000 0000 --;    54/  135
      1--- --00 0000 101- 1  :  1000 0010 -1;    55/  209
      0--- --00 0000 101- 1  :  0000 0010 0-;    56/  210
      ---- --10 0000 0101 1  :  0000 0000 --;    57/  136
      ---- --00 0000 0101 1  :  0000 0001 11;    58/  216
      ---- --10 0000 0011 1  :  0000 0000 --;    59/  137
      ---- 1-00 0000 0011 1  :  0000 0010 11;    60/  213
      ---- 0-00 0000 0011 1  :  0010 0010 --;    61/  214
      REST                   :  ---- ---- --;    62
      ---------------------------------------
      1234 5678 9012 3456 7     1234 5678 90
  PIPELINED CLOCKING SYSTEM OD20G10
  CYPRESS SEMICONDUCTOR
  90/03/15  23:49:45
  ***************************************************
  ***  NET DESCRIPTION TABLE FOR AND/OR STRUCTURE  ***
  ***************************************************
                         :  CCCC CC
                         :  LLLL LLCC
        C CCCC C         :  KKKK KKLL
       RL LLLL LCC       :  ____ __KK QQ
     I  W EK KKKK KLL     :  1122 33__ QQ
RNNI AWS_ ____ _KKQ Q    :  ABAB ABAB 12
UPTE IEE1 1223 3__Q Q    :  ....  ....  ..
NLRN TNTA BABA BAB1 2    :  DDDD DDDD DD
---------------------------------------
                        INV  ....  ....  ..
                        REG  DDDD DDDD DD
---------------------------------------
---- 0-0- --0- 0-11 0  :  A...  ....  ..  ;   1
1--- --0- --0- 1--- 1  :  A...  ....  ..  ;   2
---- --0- 1--- ---- 0  :  A...  ....  ..  ;   3
---- --0- 1-1- ---- -  :  A...  ....  ..  ;   4
--11 --0- --1- 0--- -  :  A...  ....  ..  ;   5
1--- --0- 0-0- 0-10 -  :  A...  ....  ..  ;   6
---- --01 ---0 ---- -  :  .A..  ....  ..  ;   7
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

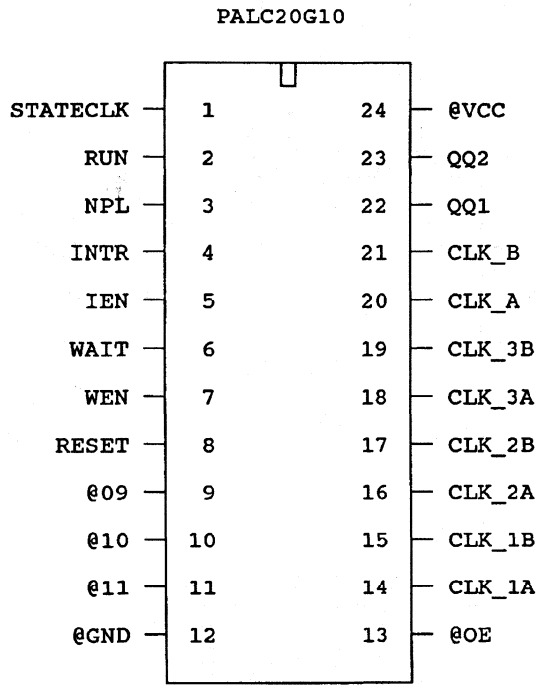
```
 1---  -001  ----  ----  -   :   .A..   ....   ..   ;   8
 1---  0-01  ----  ----  -   :   .A..   ....   ..   ;   9
 ---0  --0-  1---  ----  -   :   ..A.   ....   ..   ;  10
 --0-  --0-  1---  ----  -   :   ..A.   ....   ..   ;  11
 ----  0-0-  --0-  0-11  -   :   ..A.   ....   ..   ;  12
 ----  --0-  1-0-  ----  -   :   ..A.   ....   ..   ;  13
 ----  0-0-  -1--  ----  -   :   ...A   ....   ..   ;  14
 ----  -00-  -1--  ----  -   :   ...A   ....   ..   ;  15
 ----  --01  -1-0  ----  -   :   ...A   ....   ..   ;  16
 ---0  --0-  --1-  ----  -   :   ....   A...   ..   ;  17
 --0-  --0-  --1-  ----  -   :   ....   A...   ..   ;  18
 ----  --0-  0-1-  1---  -   :   ....   A...   ..   ;  19
 ----  0-0-  0-0-  0-11  0   :   ....   A...   ..   ;  20
 ----  0-0-  ---1  ----  -   :   ....   .A..   ..   ;  21
 ----  -00-  ---1  ----  -   :   ....   .A..   ..   ;  22
 ----  --0-  -0-1  ----  -   :   ....   .A..   ..   ;  23
 ----  --0-  ----  -0--  -   :   ....   ..A.   ..   ;  24
 ----  --0-  ----  -1--  -   :   ....   ...A   ..   ;  25
 ----  ----  ----  -1-1  -   :   ....   ....   A.   ;  26
 ----  ----  ----  0-11  -   :   ....   ....   A.   ;  27
 ----  ----  -1--  ----  -   :   ....   ....   A.   ;  28
 ----  ----  --0-  1---  -   :   ....   ....   .A   ;  29
 ----  ----  0-1-  0---  -   :   ....   ....   .A   ;  30
 ----  ---0  -1--  ----  -   :   ....   ....   .A   ;  31
 ----  ----  -0--  -1--  1   :   ....   ....   .A   ;  32
 -1--  ---0  0--0  0--0  -   :   ....   ....   .A   ;  33
 ----  ----  00--  0--1  1   :   ....   ....   .A   ;  34
 ---------------------------------------
1234 5678 9012 3456 7   :  1234 5678 90
   PIPELINED CLOCKING SYSTEM OD20G10
   CYPRESS SEMICONDUCTOR
   90/03/15  23:49:45
   ****************************************************
   ***              BOOLEAN EQUATIONS                 ***
   ****************************************************
```

Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
 CLK_1A.D     :=
          /WAIT   & /RESET  & /CLK_2B       & /CLK_3B       &  CLK_B
                  &  QQ1    & /QQ2
        +  RUN    & /RESET  & /CLK_2B       &  CLK_3B       &  QQ2
        + /RESET  &  CLK_1B       & /QQ2
        + /RESET  &  CLK_1B       &  CLK_2B
        +  INTR   &  IEN    & /RESET  &  CLK_2B       & /CLK_3B          +  RUN
 & /RESET  & /CLK_1B       & /CLK_2B       & /CLK_3B
                  &  CLK_B  & /QQ1    ;
CLK_1B.D     :=
          /RESET  &  CLK_1A       & /CLK_3A
       +  RUN    & /WEN    & /RESET  &  CLK_1A
        +  RUN    & /WAIT   & /RESET  &  CLK_1A       ;
CLK_2A.D     :=
          /IEN    & /RESET  &  CLK_1B
        + /INTR   & /RESET  &  CLK_1B
        + /WAIT   & /RESET  & /CLK_2B       & /CLK_3B       &  CLK_B
                  &  QQ1
        + /RESET  &  CLK_1B       & /CLK_2B       ;
CLK_2B.D     :=
          /WAIT   & /RESET  &  CLK_2A
        + /WEN    & /RESET  &  CLK_2A
        + /RESET  &  CLK_1A       &  CLK_2A       & /CLK_3A       ;
CLK_3A.D     :=
          /IEN    & /RESET  &  CLK_2B
        + /INTR   & /RESET  &  CLK_2B
        + /RESET  & /CLK_1B       &  CLK_2B       &  CLK_3B
        + /WAIT   & /RESET  & /CLK_1B       & /CLK_2B       & /CLK_3B
                  &  CLK_B  &  QQ1    & /QQ2    ;
CLK_3B.D     :=
          /WAIT   & /RESET  &  CLK_3A
        + /WEN    & /RESET  &  CLK_3A
        + /RESET  & /CLK_2A       &  CLK_3A       ;
CLK_A.D      :=
          /RESET  & /CLK_A  ; CLK_B.D      :=
          /RESET  &  CLK_A  ;
QQ1.D  :=  CLK_A  &  QQ1
        +  /CLK_3B       &  CLK_B  &  QQ1
        +  CLK_2A        ;
QQ2.D  := /CLK_2B        &  CLK_3B
        + /CLK_1B        &  CLK_2B       & /CLK_3B
        + /CLK_1A        &  CLK_2A
        + /CLK_2A        &  CLK_A  &  QQ2
        +  NPL    & /CLK_1A       & /CLK_1B       & /CLK_3A
                  & /CLK_3B       & /QQ1
        + /CLK_1B        & /CLK_2A       & /CLK_3B       &  QQ1    &  QQ2    ;
```

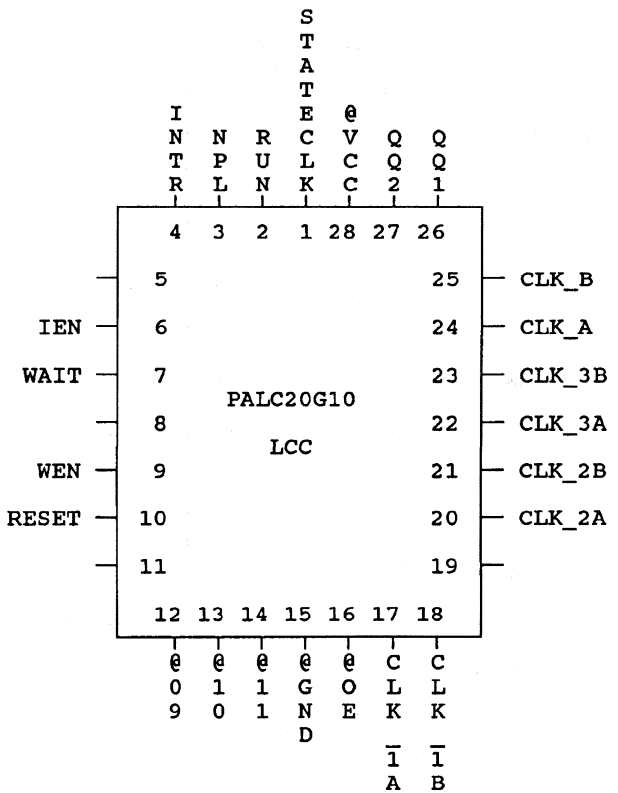
Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
 PIPELINED CLOCKING SYSTEM OD20G10
CYPRESS SEMICONDUCTOR
90/03/15  23:49:45
```



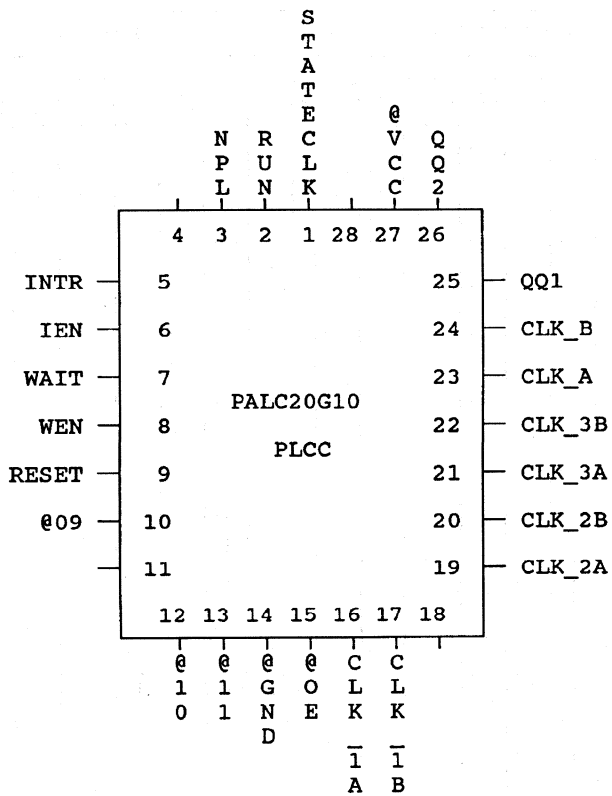
Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
PIPELINED CLOCKING SYSTEM OD20G10
CYPRESS SEMICONDUCTOR
90/03/15  23:49:45
```



Appendix A. LOG/iC PLD Source Code: Clock State Machine (continued)

```
PIPELINED CLOCKING SYSTEM OD20G10
CYPRESS SEMICONDUCTOR
90/03/15  23:49:45
```



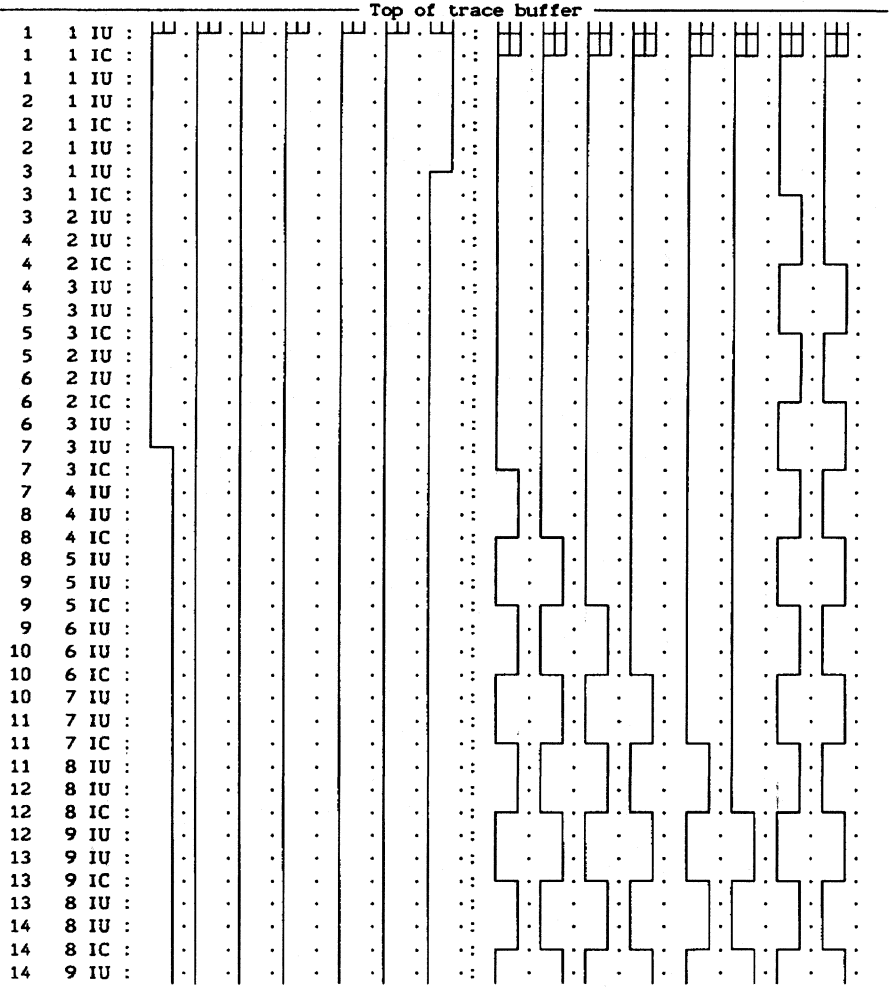

```
-------------------------------------------------------
LOG/iC  -  PAL              CPU TIME USED:      45 SEC
```

Appendix B. LOG/iC Simulation: Clock State Machine

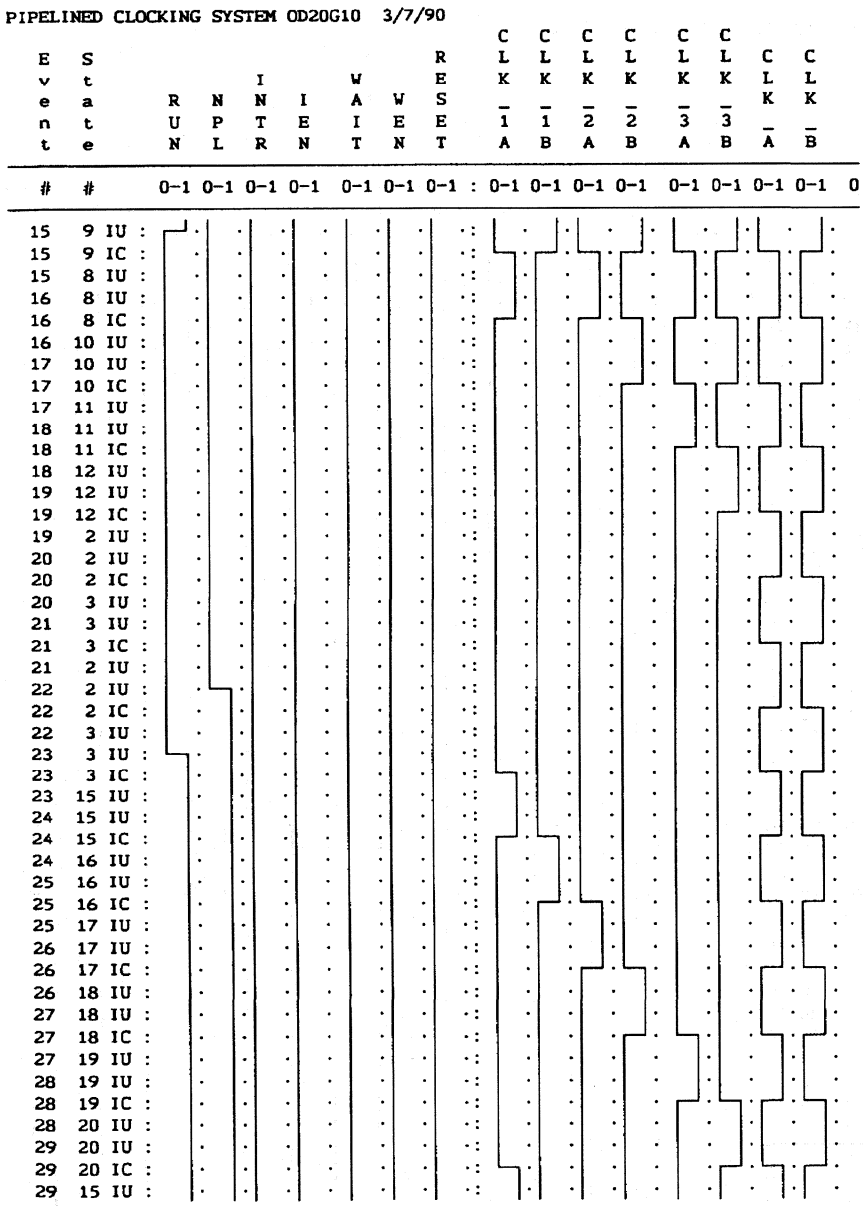
PIPELINED CLOCKING SYSTEM OD20G10 3/7/90

Event #	State #	RUN	NPL	INTR	IEN	WAIT	WEN	RESET	CLK1A	CLK1B	CLK2A	CLK2B	CLK3A	CLK3B	CLKA	CLKB
#	#	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0

Top of trace buffer



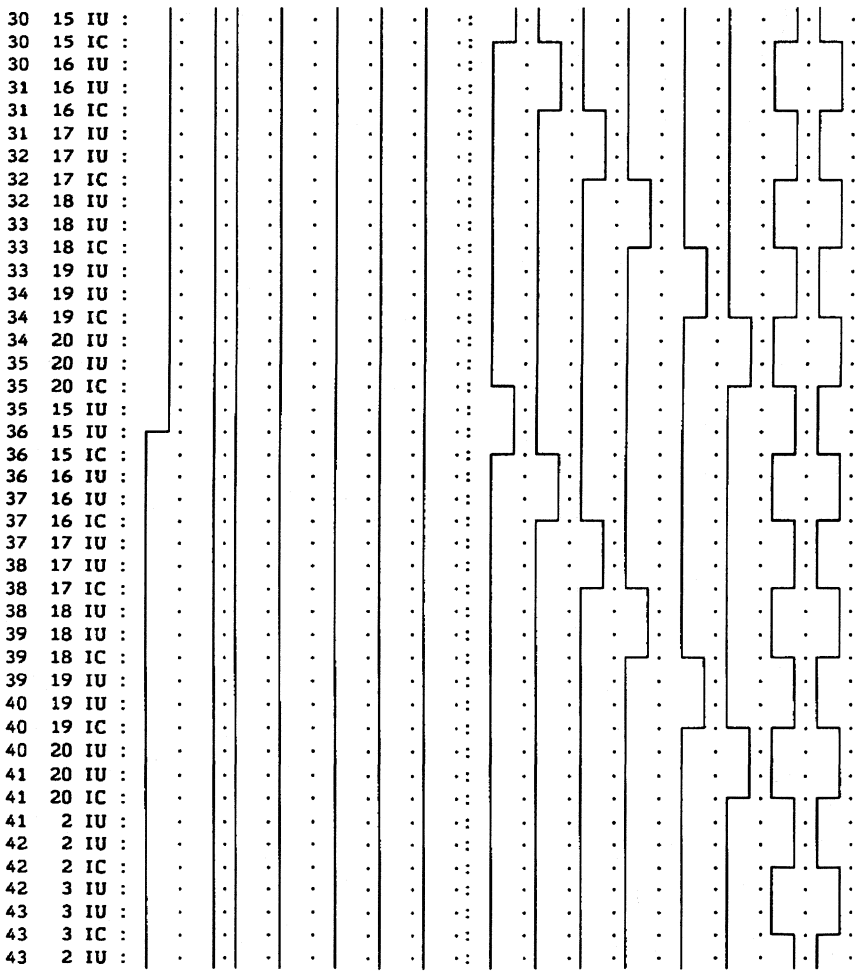
Appendix B. LOG/iC Simulation: Clock State Machine (continued)



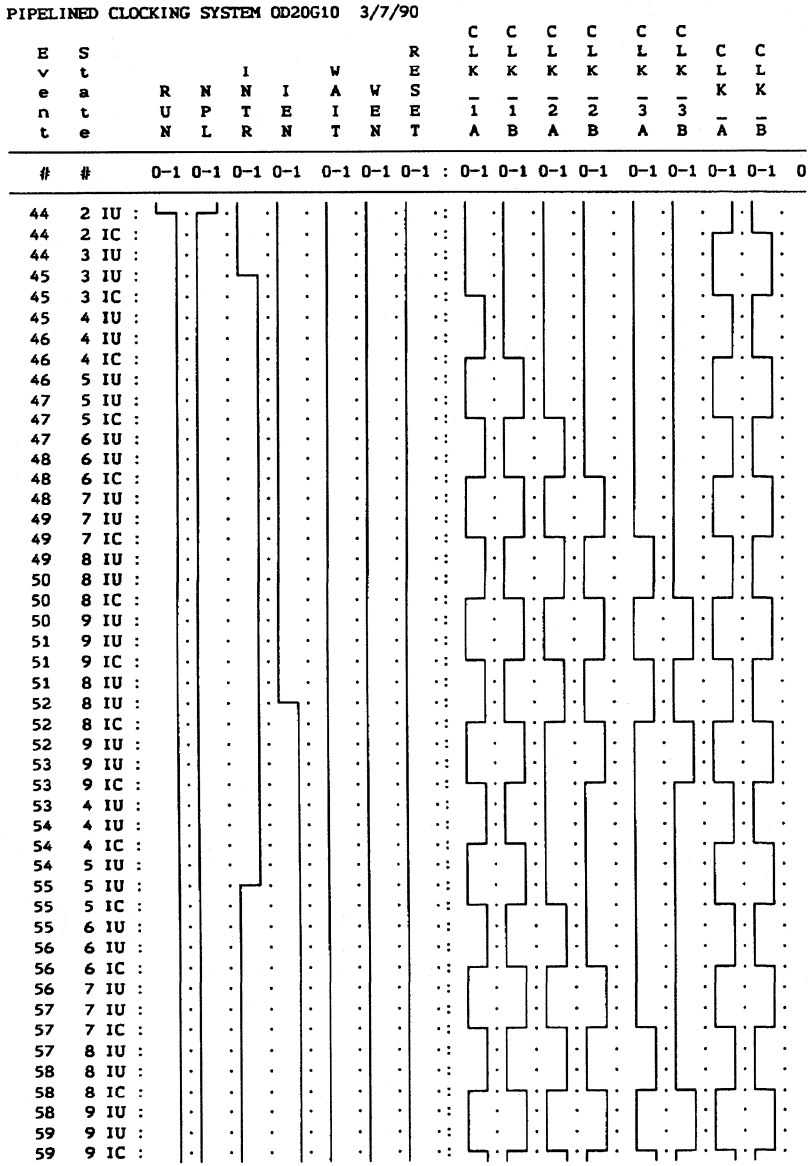
Appendix B. LOG/iC Simulation: Clock State Machine (continued)

PIPELINED CLOCKING SYSTEM QD20G10 3/7/90

Event	State	RUN	NPL	INTR	INEN	AIT	WEN	RES	CLK 1A	CLK 1B	CLK 2A	CLK 2B	CLK 3A	CLK 3B	CLK A	CLK B
#	#	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1



Appendix B. LOG/iC Simulation: Clock State Machine (continued)



Appendix B. LOG/iC Simulation: Clock State Machine (continued)

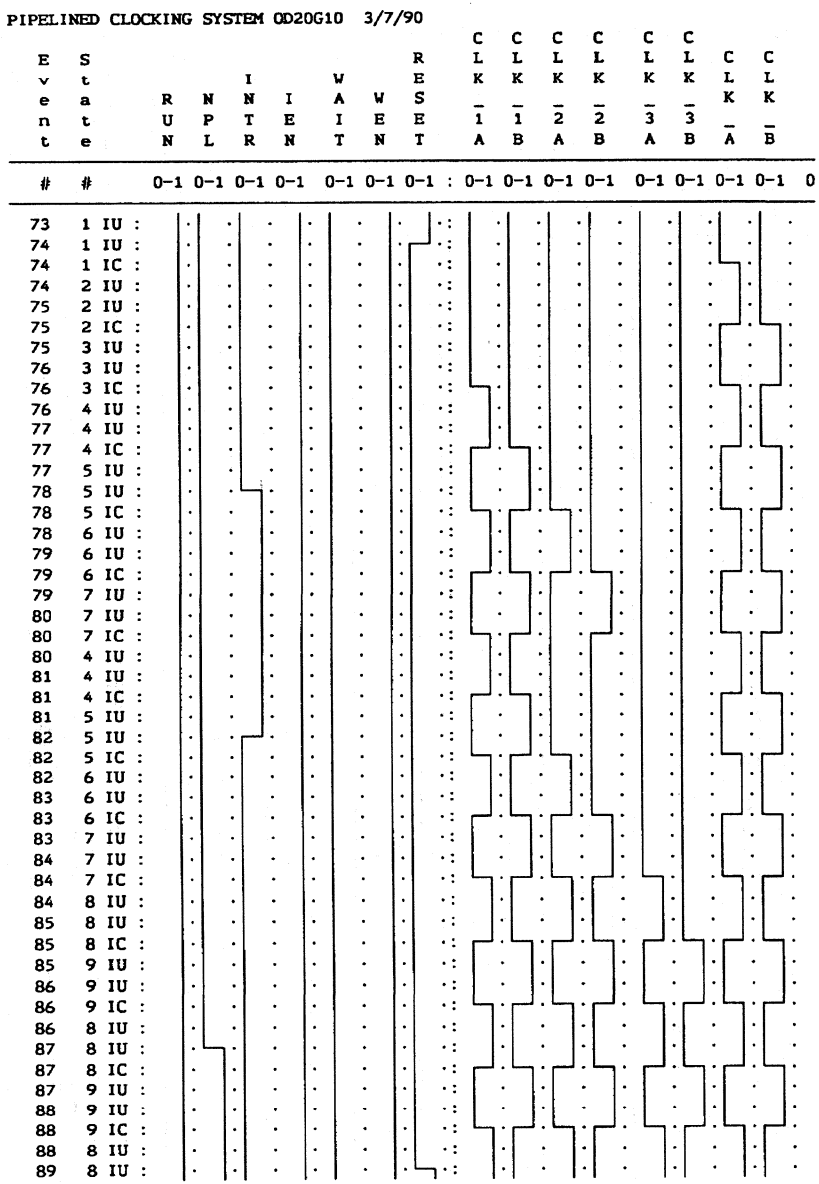
```
PIPELINED CLOCKING SYSTEM OD20G10  3/7/90

                                                                  C   C   C   C   C   C
  E   S                                                   R       L   L   L   L   L   L   C   C
  v   t                       I           W               E       K   K   K   K   K   K   L   L
  e   a       R   N   N   I   A   W   R   S                       _   _   _   _   _   _   K   K
  n   t       U   P   T   E   I   E   E                   1   1   2   2   3   3       _   _
  t   e       N   L   R   N   T   N   T                   A   B   A   B   A   B   A   B

  #   #       0-1 0-1 0-1 0-1 0-1 0-1 0-1 : 0-1 0-1 0-1 0-1 0-1 0-1 0-1 0-1 0

 59   8 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 60   8 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 60   8 IC  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 60   9 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 61   9 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 61   9 IC  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 61   8 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 62   8 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 62   8 IC  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 62  14 I   :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 63  14 I   :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 63  14 IC  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 63  13 I   :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 64  13 I   :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 64  13 IC  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 64  14 I   :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 65  14 I   :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 65  14 IC  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 65  13 I   :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 66  13 I   :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 66  13 IC  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 66  14 I   :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 67  14 I   :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 67  14 IC  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 67   8 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 68   8 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 68   8 IC  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 68   9 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 69   9 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 69   9 IC  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 69   8 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 70   8 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 70   8 IC  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 70   9 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 71   9 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 71   9 IC  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 71   1 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 72   1 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 72   1 IC  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 72   1 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 73   1 IU  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
 73   1 IC  :  .   .   .   .   .   .   .  :  .   .   .   .   .   .   .   .   .
```

Appendix B. LOG/iC Simulation: Clock State Machine (continued)



Appendix B. LOG/iC Simulation: Clock State Machine (continued)

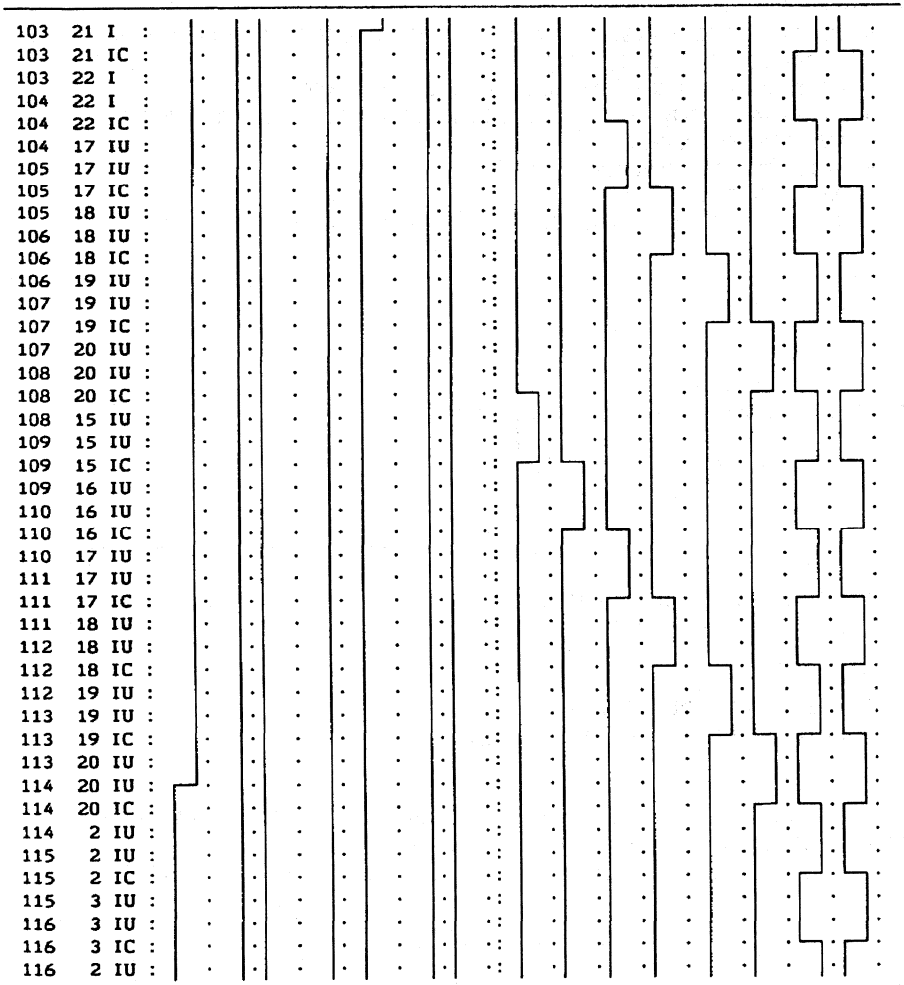
PIPELINED CLOCKING SYSTEM 0D20G10 3/7/90

Event	State	RUN	NPL	INTR	INIEN	WAIT	WEN	RESET	CLK_1A	CLK_1B	CLK_2A	CLK_2B	CLK_3A	CLK_3B	CLK_A	CLK_B
#	#	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1	0-1
89	8 IC															
89	1 IU															
90	1 IU															
90	1 IC															
90	2 IU															
91	2 IU															
91	2 IC															
91	3 IU															
92	3 IU															
92	3 IC															
92	15 IU															
93	15 IU															
93	15 IC															
93	16 IU															
94	16 IU															
94	16 IC															
94	17 IU															
95	17 IU															
95	17 IC															
95	18 IU															
96	18 IU															
96	18 IC															
96	15 IU															
97	15 IU															
97	15 IC															
97	16 IU															
98	16 IU															
98	16 IC															
98	17 IU															
99	17 IU															
99	17 IC															
99	22 I															
100	22 I															
100	22 IC															
100	21 I															
101	21 I															
101	21 IC															
101	22 I															
102	22 I															
102	22 IC															
102	21 I															

Appendix B. LOG/iC Simulation: Clock State Machine (continued)

```
PIPELINED CLOCKING SYSTEM OD20G10  3/7/90

                                                                  CLK  CLK  CLK  CLK  CLK  CLK
 Event  State                                          RESET      1_A  1_B  2_A  2_B  3_A  3_B   CLK_A  CLK_B
   #      #      RUN  NPL  INTR  IEN  WAIT  WEN  RST  : 0-1  0-1  0-1  0-1  0-1  0-1  0-1  0-1  0
 103    21 I   :
 103    21 IC  :
 103    22 I   :
 104    22 I   :
 104    22 IC  :
 104    17 IU  :
 105    17 IU  :
 105    17 IC  :
 105    18 IU  :
 106    18 IU  :
 106    18 IC  :
 106    19 IU  :
 107    19 IU  :
 107    19 IC  :
 107    20 IU  :
 108    20 IU  :
 108    20 IC  :
 108    15 IU  :
 109    15 IU  :
 109    15 IC  :
 109    16 IU  :
 110    16 IU  :
 110    16 IC  :
 110    17 IU  :
 111    17 IU  :
 111    17 IC  :
 111    18 IU  :
 112    18 IU  :
 112    18 IC  :
 112    19 IU  :
 113    19 IU  :
 113    19 IC  :
 113    20 IU  :
 114    20 IU  :
 114    20 IC  :
 114     2 IU  :
 115     2 IU  :
 115     2 IC  :
 115     3 IU  :
 116     3 IU  :
 116     3 IC  :
 116     2 IU  :
```



Appendix C. Cypress PLD ToolKit: CY7C361 Implementation

```
 'CY7C361;

{PIPELINED CLOCKING SYSTEM AN1_361
 CYPRESS SEMICONDUCTOR}

CONFIGURE;

{-------------------------------------------------------------------------
;INPUT DEFINITIONS :
; RUN     = START & STOP EXECUTION OF OUTPUT CLOCKS (NORMAL, SINGLE STEP,
;           & BREAK PT.  EXECUTION
; NPL     = PIPELINED VS NON-PIPELINED MODE OF EXECUTION
; INTR    = EXTERNAL INTERRUPT CONDITION (TLB MISS, PARITY ERROR,...)
; IEN     = INTERRUPT ENABLE
; WAIT    = WAIT ENABLE (CACHE MISS)
; WEN     = WAIT ENABLE
; RPT_EO = USED TO DUB CLK_1B, CLK USED TO UPDATE THE EO REG

;-------------------------------------------------------------------------
;OUTPUT DEFINITIONS :
;
; 3 CLOCK STAGES 1, 2, 3
; 2 CLOCKS PER STATE A, B
;    CLK_XX WHERE XX = 1A,1B,2A,2B,3A,3B
;
; 2 FREE RUNNING CLOCKS
;    CLK_A, CLK_B
;
;-------------------------------------------------------------------------
}
RUN(node=3), STATECLK, NPL, INTR,                {*INPUTS*}
IEN(node=9), WAIT, WEN, RESET,
/RPT_EO,

/CLK_A(node=16), /CLK_B, /CLK_1A,                {*OUTPUTS*}
/CLK_2B(node=24), /CLK_1B(and), /CLK_2A, /CLK_3A,
/CLK_3B,

{ LOCAL 8      LOCAL 8      HALF 16      GLOBAL 32     *STATE MACROCELLS*
  FEEDBACK     FEEDBACK     FEEDBACK     FEEDBACK
  _________    _________    _________    __________     START = DEFAULT}

AX(node=32),   A,           B,           1B,          {LOCAL 8 = 1, HALF = 1}
1A,            1AX,         12A,         123A,

BW,            AW,          12B,         123AX,       {LOCAL 8 = 2, HALF = 1}
```

Appendix C. Cypress PLD ToolKit: CY7C361 Implementation (continued)

```
BWX,                                        123AY(node=47),

AWN,          BWN,          2AN,          123B,       {LOCAL 8 = 1, HALF = 2}
              BWNX(node=53),2ANX,         123BX,
23B,          23BX,         2BNX,         2BN,        {LOCAL 8 = 2, HALF = 2}
3A,           3AN,          3ANX,         3BN,

IENA(node=29),IENB,                                   {*MISC*}
GLBRST(node=64),
GND(NODE=73),
CLKDB(NODE=74)


EQUATIONS;

    GLBRST = <prod> RESET;                            {*MISC*}

    IENA = <INV_SUM> /GND;

    IENB = <INV_SUM> /GND;

    AX   = <prod> /RESET;                             {*STATE MACROCELLS}

    A    = <prod> /RUN
           <inv_prod> /B * /3BN;

    B    = <prod>
           <inv_prod> /AX * /A;

    1B   = <prod>
           <inv_prod> /1A * /1AX;

    1A   = <prod> INTR * IEN
           <inv_prod> /123B * /123BX * /12B * /2BN;

    1AX  = <prod> RUN
           <inv_prod> /B * /3BN;

    12A  = <prod> /NPL
           <inv_prod> /1B;

    123A = <prod> /INTR
           <inv_prod> /12B * /123B * /123BX;

    BW   = <prod> WAIT * WEN
           <inv_prod> /123A * /123AX * /123AY;
```

Appendix C. Cypress PLD ToolKit: CY7C361 Implementation (continued)

```
AW    = <prod> WAIT
        <inv_prod> /BW * /BWX;

12B   = <prod> 12A;

123AX= <prod> INTR * /IEN
        <inv_prod> /12B * /123B * /123BX;

BWX   = <prod> AW;

123AY= <prod> /WAIT
        <inv_prod> /BW * /BWX;

AWN   = <prod> WAIT
        <inv_prod> /BWN * /BWNX;

BWN   = <prod> WAIT * WEN
        <inv_prod> /2AN;

2AN   = <prod> NPL
        <inv_prod> /1B;

123B = <prod> RUN * /WAIT
        <inv_prod> /123A * /123AX * /123AY;

BWNX = <prod> AWN;

2ANX = <prod> /WAIT
        <inv_prod> /BWN * /BWNX;

123BX= <prod> RUN * WAIT * /WEN
        <inv_prod> /123A * /123AX * /123AY;

23B   = <prod> /RUN * WAIT * /WEN
        <inv_prod> /123A * /123AX * /123AY;

23BX = <prod> /RUN * /WAIT
        <inv_prod> /123A * /123AX * /123AY;

2BNX = <prod> WAIT * /WEN
        <inv_prod> /2AN * /2ANX;

2BN   = <prod> /WAIT
        <inv_prod> /2AN * /2ANX;

3A    = <prod>
        <inv_prod> /23B * /23BX;
```

Appendix C. Cypress PLD ToolKit: CY7C361 Implementation (continued)

```
3AN   = <prod> /INTR
        <inv_prod> /2BN * /2BNX;

3ANX  = <prod> INTR * /IEN
        <inv_prod> /2BN * /2BNX;


3BN   = <prod>
        <inv_prod> /3A * /3AN * /3ANX;

CLK_A   = <inv_sum> /A * /AX * /1A * /1AX *    {*OUTPUTS*}
                    /12A * /123A * /123AX *
                    /123AY * /AW * /3A * /2AN *
                    /2ANX * /AWN * /3AN * /3ANX;

CLK_B   = <inv_sum> /B * /1B * /12B * /123B *
                    /123BX * /BW * /BWX * /23B *
                    /23BX * /2BN * /2BNX * /BWN *
                    /BWNX * /3BN;

CLK_1A = <inv_sum> /1A * /1AX * /12A * /123A *
                   /123AX * /123AY;

CLK_1B = <inv_sum> /1B * /12B * /123B * /123BX;

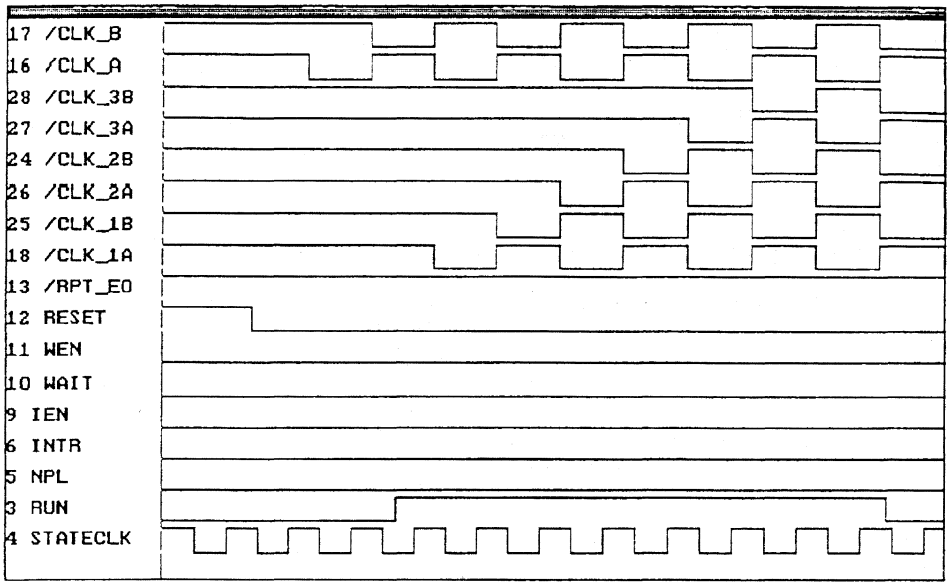
CLK_2A = <inv_sum> /12A *  /123A * /123AX *
                   /123AY * /2AN * /2ANX;

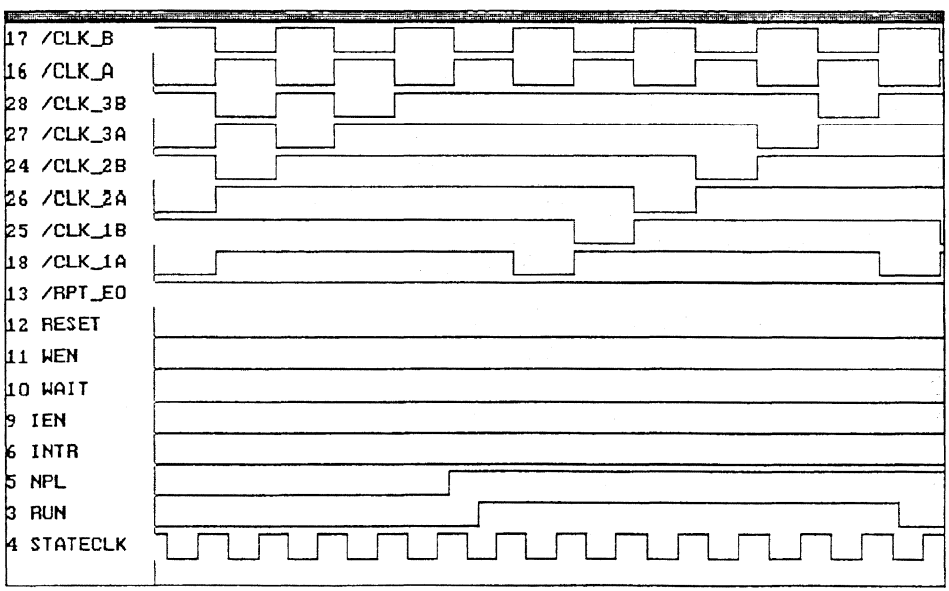
CLK_2B = <inv_sum> /12B * /123B * /123BX *
                   /23B * /23BX * /2BN * /2BNX;

CLK_3A = <inv_sum> /123A * /123AX * /123AY *
                   /3A * /3AN * /3ANX;

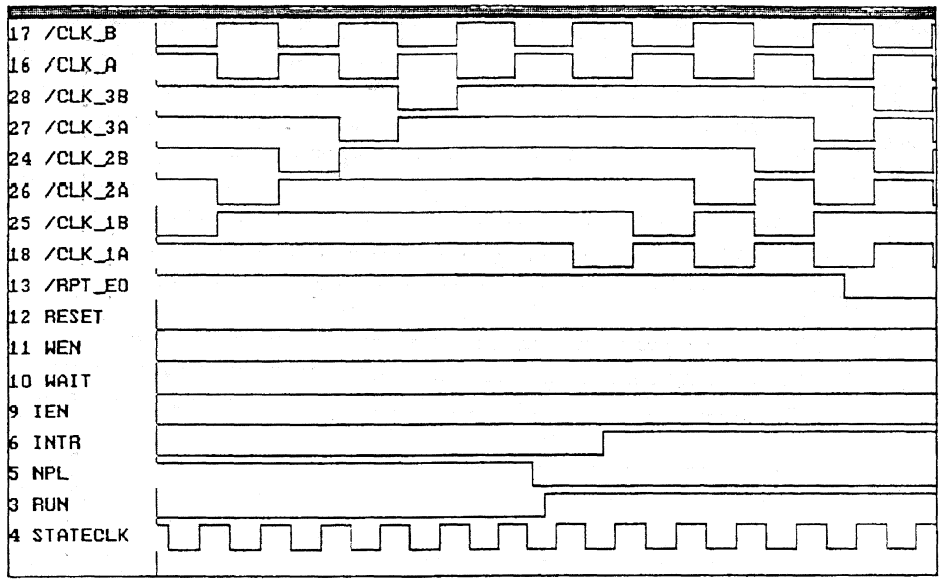
CLK_3B = <inv_sum> /123B * /123BX * /23B *
                   /23BX * /3BN;
```

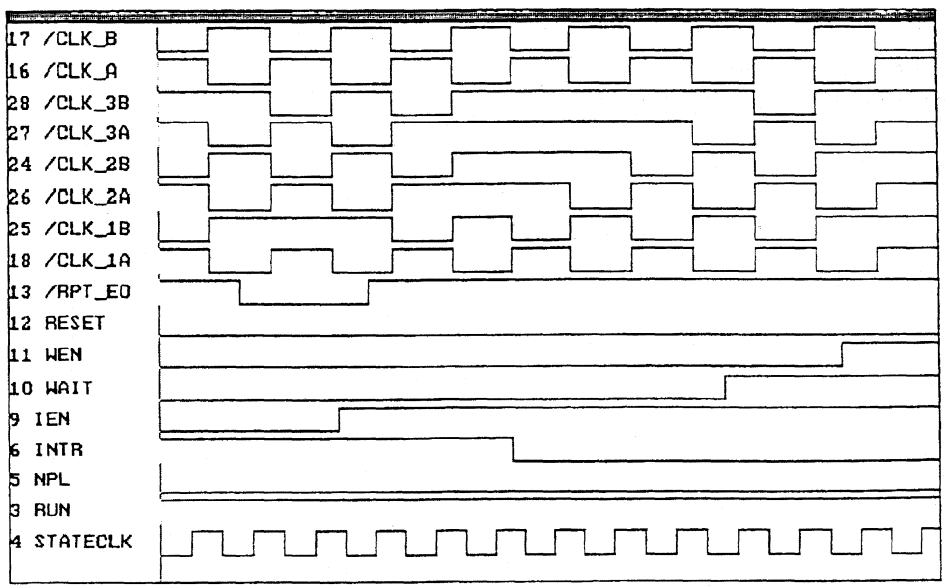
Appendix D. Cypress PLD ToolKit: CY7C361 Simulation



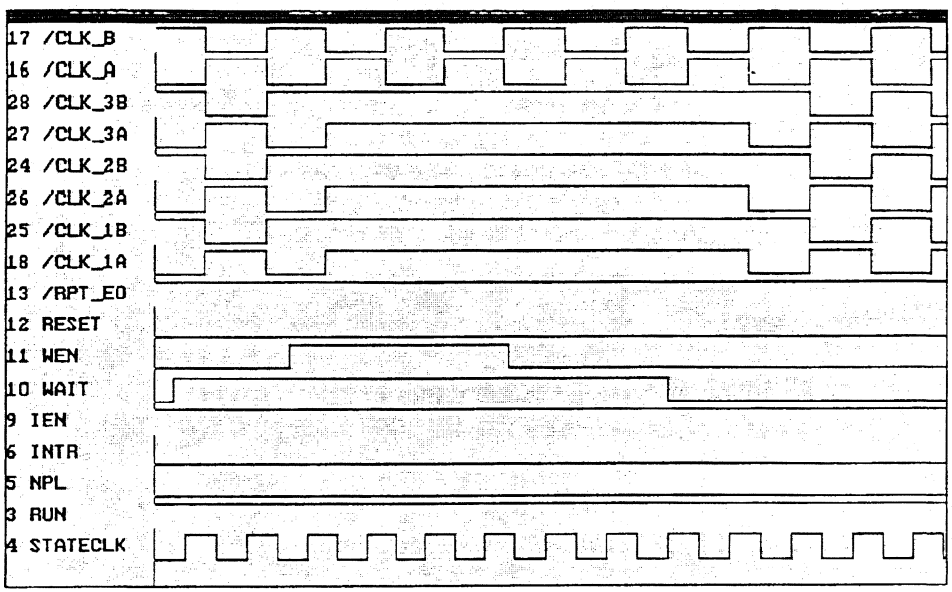


Appendix D. Cypress PLD ToolKit: CY7C361 Simulation (continued)





Appendix D. Cypress PLD ToolKit: CY7C361 Simulation (continued)



Document #: PLD0013−0492

LOG/iC is a trademark of Isdata Corporation.
PLD Toolkit is a trademark of Cypress Semiconductor Corporation.

Designing with the CY7C335 and *Warp2*™ VHDL Compiler

This application note provides an overview of the CY7C335 Universal Synchronous EPLD architecture and *Warp2*™ VHDL Compiler for PLDs. Example designs demonstrate how the *Warp2* VHDL compiler takes advantage of the rich architectural features of the CY7C335.

The CY7C335 is a synchronous EPLD optimized for high-performance state machines and other clocked systems that operate at speeds of up to 100 MHz. The CY7C335 uses Cypress's low-power, 0.8-micron CMOS UV erasable technology and is packaged in 28-pin, 300-mil dual in-line and LCC/PLCC packages.

The CY7C335 builds on the popularity of the high-speed CMOS PALC22V10 and exceeds the capability of the 26V12 and 26CV12. The CY7C335 offers significantly higher density solutions and can replace as many as four 22V10s. It has 258 variable product terms for 16 state registers (ranging from 9 to 19 product terms per macrocell), macrocells that can be configured as JK-, RS-, T-, or D-type, bidirectional pins, bypassable input registers, three clocks, and a product term output enable for each macrocell.

In addition to supporting the features of the CY7C335, the *Warp2* VHDL compiler enables the designer to create designs, using any combination of high-level behavioral descriptions, Boolean equations, state tables, or RTL structures, that can easily be retargetted to any Cypress PLD.

Warp2 is a state-of-the-art VHDL compiler that facilitates device-independent designs by synthesizing for a powerful subset of IEEE1076. Optimization and reduction algorithms automatically select T- or D-type flip-flops and perform automatic state and pin assignment. *Warp2* includes a graphical user interface (which runs under Windows™ for the PC, and OpenLook™ or Motif™ for the Sun) and comes complete with a functional simulator for graphical waveform simulation.

Overview of the CY7C335

Figure 1 is the block diagram of the CY7C335. Three separate clock signals—two input and two output clocks (one shared)—can be used on pins 1, 2, and 3. Alternatively, pins 2 and 3 can be used as two of twelve inputs that may be registered or fed directly to the programmable AND array. Pin 14 can be used as an input or as a common output enable for each I/O pin. Outputs can also be enabled by product terms. The device features center ground and supply pins that reduce ground bounce due to parasitic effects, particularly lead inductance.

Figure 2 illustrates the input macrocell. Each D-type input register can use either ICLK1 or ICLK2. Alternatively, the input register bypass multiplexer can be programmed to allow the signal to feed directly to the array as combinatorial input.

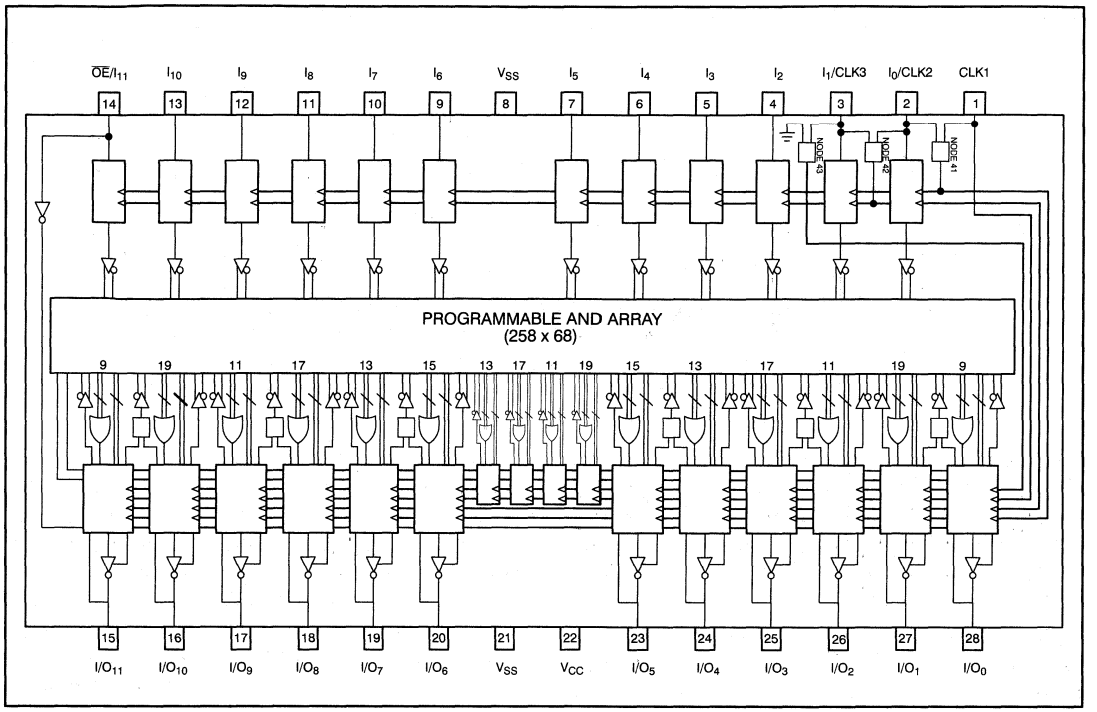


Figure 1. CY7C335 Block Diagram

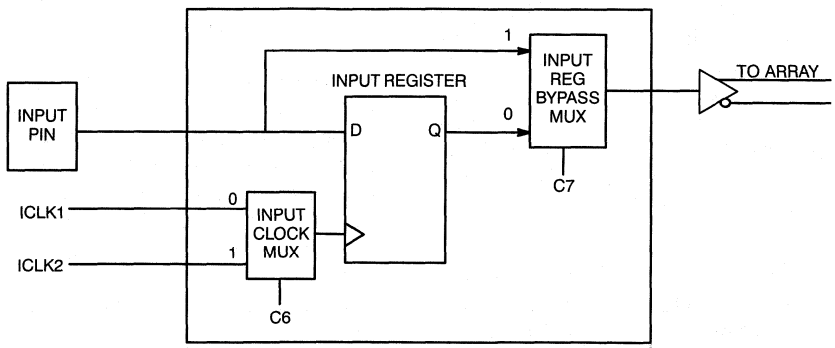


Figure 2. CY7C335 Input Macrocell

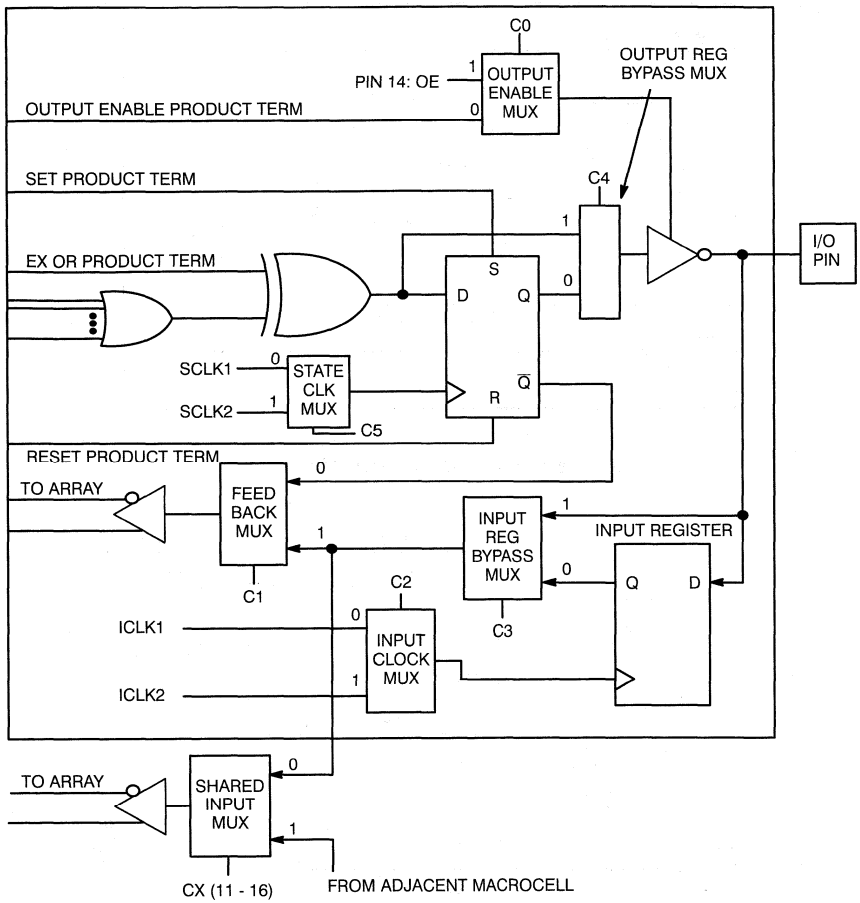


Figure 3. CY7C335 Input/Output Macrocell

Twelve configurable I/O macrocells enable JK-, RS-, T-, or D-type state registers to optimize for minimal product terms. *Figure 3* illustrates the I/O macrocell, which includes the following features: (1) registered or combinatorial output; (2) global (by pin 14) or product term output enable; (3) global, synchronous, product-term set and reset; (4) three clocks—two can be used as input clocks and two can be used as output clocks (with one shared); (5) input/output flexibility (the cell can be configured as input only, output only, or a dedicated input with a buried register by using the shared input multiplexer and thereby maximizing cell resource utilization).

In addition to the input and I/O macrocells, the CY7C335 features four hidden macrocells, one of which is shown in *Figure 4*. Buried registers are highly useful for state machines, internal counters, or other applications that need registers that are not also used as outputs.

The clocking scheme is shown in *Figure 5*. The CY7C335 can utilize three separate clocks. Two clocks are inputs to each of the input clock multiplexers and state clock multiplexers. If two clocks are used on both the input and the state registers, then one of the clocks is shared, because a total of

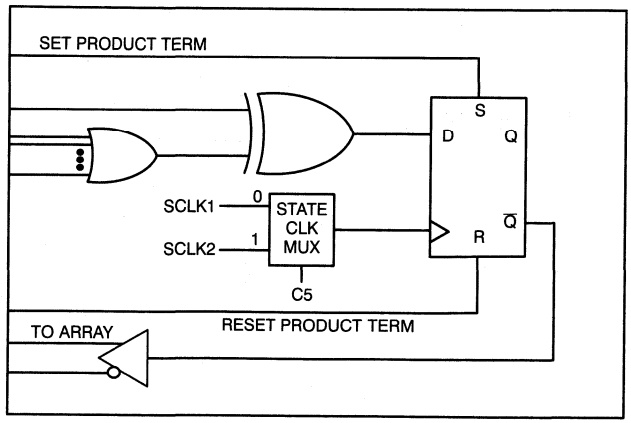


Figure 4. CY7C335 Hidden Macrocell

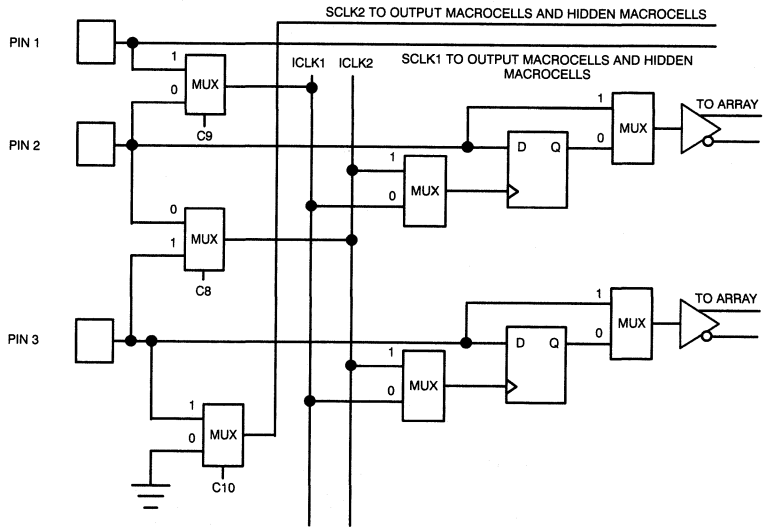


Figure 5. CY7C335 Input Clocking Scheme

three clocks are supported. Pin 1 is a dedicated state clock pin, designated SCLK1 (state clock). Pins 2 and 3 may be used as either inputs or clocks, as shown in *Figure 5*.

Overview of *Warp2*

Warp2 is a state-of-the-art VHDL compiler for designing with Cypress PLDs and PROMs. *Warp2* accepts VHDL designs, synthesizes and optimizes the entered design, and outputs a JEDEC file for the CY7C335. *Warp2* also provides a graphical wave-

form simulator for functional simulation. *Figure 6* illustrates the *Warp2* design flow.

VHDL Compiler

As an open, non-proprietary, IEEE1076 compliant language that is the standard for behavioral design entry and simulation, VHDL allows designers to easily describe complex hardware systems.

Warp2's VHDL enables designers to describe device-independent designs at different levels of abstraction, including behavioral descriptions, Boolean equations, state tables, and structural descriptions. In addition, VHDL and *Warp2* support hierarchical designs, allowing either a "top-down" or "bottom-up" approach to design.

Design Examples

The following design examples demonstrate how to use *Warp2* and VHDL to take advantage of the CY7C335 architectural features. The purpose is to show some VHDL constructs that are particularly useful for the CY7C335 architecture as well as point out designs that are well suited for the device. Further information on VHDL constructs may be found in the *Warp2 Reference Manual* or one of several texts available on the language. For each of the examples, the complete VHDL source code and an excerpt of the report file may be found in the appendices.

Pipelined Buffer

This example demonstrates how to use VHDL code to implement a pipelined buffer (see *Figure 7*) with multiple clocks and output enables. The CY7C335 is well suited for pipelined applications because it has input registers in both the input macrocells as well as the I/O macrocells.

The complete VHDL source code is printed in Appendix A of this note. The pipeline architecture is reprinted in *Figure 8*.

The pipeline is implemented in three processes. The first process registers (using CLK1 on the input registers) the upper four bits of the input signal I. The second process registers the lower four bits, using CLK2. The signal INTMP represents the q output of these registers. The third process registers the signal INTMP, with OUTTMP being the q output of these registers. This signal reaches the I/O pins if output is enabled, as explained below.

Below the three processes is a generation scheme which is used to instantiate eight *triout* components

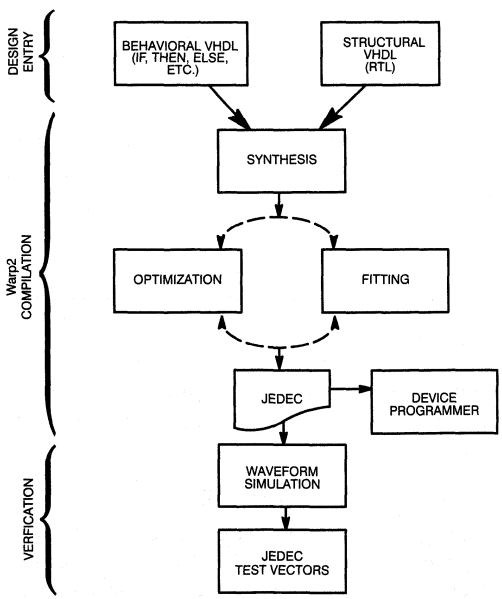


Figure 6. *Warp2* Design Flow

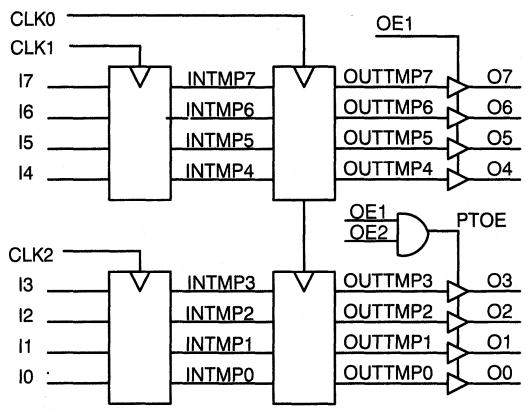


Figure 7. Pipelined Buffer Block Diagram

```
use work.rtlpkg.all;
architecture archpipe of pipe is
signal intmp, outtmp: bit_vector(7 downto 0);
signal ptoe: bit;
begin
proc1: process
  begin
    wait until clk1 = '1';
      intmp(7 downto 4) <= i(7 downto 4);
  end process;

proc2: process
  begin
    wait until clk2 = '1';
      intmp(3 downto 0) <= i(3 downto 0);
  end process;

proc3: process
  begin
    wait until clk0 = '1';
      outtmp <= intmp;
  end process;

  ptoe <= oe1 AND oe2;

g1:  for j in 7 downto 0 generate
  g2:  if j > 3 generate
       t1x: triout port map(outtmp(j), oe1, o(j));
    end generate;
  g3:   if j < 4 generate
       t2x: triout port map(outtmp(j), ptoe, o(j));
    end generate;
  end generate;

end archpipe;
```

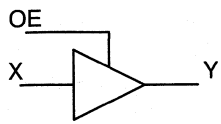
Figure 8. Pipeline Architecture

(see *Figure 9*). The *triout* components are used to implement an output enable. The upper four bits of the output are enabled by OE1 (which is assigned to pin 14 by *Warp*) and the lower four bits use a product term output enable, PTOE.

The complete VHDL source code for this example is in Appendix A. A report file excerpt, showing resource utilization, is shown in Appendix B. This excerpt shows that 8 of 12 I/O macrocells were utilized. However, not all resources (the input registers, for example) in those macrocells were utilized.

Comparator with Registered Inputs

In high-speed systems, such as microprocessor local buses that operate at 40, 50, or 66 MHz, data or addresses must be captured from the bus (when qualified with a strobe) with set-up times of 3 to 5 ns. Few logic functions can be implemented in this time, and




```
component triout
 port(
   x: in bit; --input to buffer
   oe: in bit; -- output enable
   y: out bit; -- output
 );
end component;
```

Figure 9. Triout Component

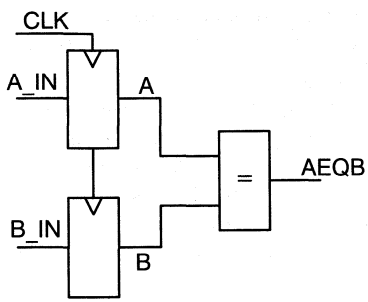


Figure 10. Comparator

for this reason data or addresses are captured and then processed in pipeline fashion. The CY7C335, with its input registers, is well suited for such high-speed systems.

In this simple, register-intensive example, all 18 inputs are registered and the output is combinatorial (*Figure 10*). As noted in Appendix D, this design leaves much of the CY7C335's resources free for additional logic. The 22V10, however, would be unable to fit a 5-bit comparator with registered inputs. Ten macrocells would be consumed when registering the inputs, leaving no macrocells for the AEQB combinatorial output. The 22V10 fares poorly in such pipelined systems because it does not have input registers and must therefore waste output macrocell resources.

The VHDL source code can be found in its entirety in Appendix C. The architecture is reprinted below.

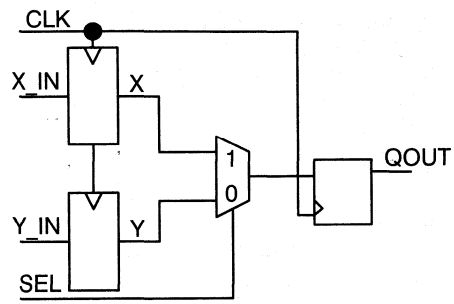


Figure 11. Multiplexer

```
architecture archcomp of comp is
signal a, b: bit_vector(0 to 8);
begin
proc1:  process begin
 wait until clk = '1';
   a <= a_in;
   b <= b_in;
 end process;

 aeqb <= '1' when a=b else '0';
end archcomp;
```

The process is used to register the inputs on the rising edge of CLK1. The equation for AEQB is placed outside of the process because it is a combinatorial output.

Multiplexer with Registered Inputs and Outputs

Registered multiplexers and demultiplexers demand a large number of inputs and outputs. This example (*see Figure 11*) takes advantage of the CY7C335 input and output registers, two groups of six-bit-wide signals are captured via the input registers and signal SEL selects one of the groups, which is then registered on the output. The complete VHDL source code can be found in Appendix E. The architecture is reprinted below.

```
architecture archmux of mux is
signal x, y: bit_vector(5 downto
0);
begin
proc1:  process begin
 wait until clk = '1';
   x <= xin;
   y <= yin;
```

```
    if sel = '1' then
      qout <= x;
    elsif sel = '0' then
      qout <= y;
    end if;
  end process;
end archmux;
```

On the rising edge of CLK1, the inputs are registered while the outputs are propagated. Thus, data on the inputs is not propagated to the outputs until the second rising edge.

Decoder

Faster microprocessors require decoders to operate at higher frequencies. Many high-density PLDs and FPGAs cannot meet speed requirements, leaving designers to opt for ASIC-based solutions which can be time consuming and expensive. The CY7C335 is another option.

Consider a 16-bit address that requires decoding to address system memory elements (SRAM, PROM, EEPROM and "shadow" RAM) and two peripheral ports. At times other than boot-up, the microprocessor fetches instructions from shadow RAM that is loaded from PROM during boot-up. *Figure 12* shows the VHDL architecture that decodes the memory map shown in *Figure 13*. Appendix H shows that the CY7C335's resources easily handle this application while operating at speeds to 100 MHz.

Up/Down Counter with Upper and Lower Limits

This example demonstrates how to use VHDL code to implement the up/down counter shown in *Figure 14*. The CY7C335 is particularly well suited for this design because it supports three clocks and has flexible I/O. This design requires the following resources: three clocks (two inputs and one state), eight input registers for the lower limit, eight input registers for the upper limit, one input each for the preset HIGH, preset LOW, reset, and output enable signals, eight state registers for the counter, one state register each for the comparators, and one state register for the counter direction signal.

A total of 20 inputs and 8 outputs are required; consequently, this design utilizes bidirectional signals.

The counter output is three-stated to load six bits of the upper limit into input registers of I/O macrocells. For example, the least-significant counter bit is stored in a state register and the least-significant upper-limit bit is stored in the input register of the same macrocell. The least-significant upper-limit bit feeds into the array via the shared input multiplexer. (The shared-input multiplexers are placed between adjacent I/O macrocells, and allow for input when the macrocell register is buried.) The CY7C335 provides six of these multiplexers. The two most significant bits of the upper limit are passed into the array through an I/O pin configured as a dedicated input. The two most significant bits of the upper limit and counter may be externally tied together so the design can be bidirectional.

The up/down counter counts between limits stored in the input registers. The lower-limit (LL) is loaded into the registers on the rising edge of CLK1 while the upper limit is loaded on the rising edge of CLK2. On CLK0, if preH is asserted, then the upper limit is loaded into the counter, and if preL is asserted, then the lower limit is loaded into the counter.

The 22V10 would not suffice for this design. Although the 22V10 has been an attractive choice of devices to implement counters and state machines, it suffers a limitation in addition to its poor handling of pipelined systems: it does not have any buried registers.

In counters and encoded state machines, registers often need not be apparent to the outside, meaning the registers can be buried within the device. In the 22V10, all macrocells are connected to I/O pins. Thus, even when a macrocell register is being used in a buried sense, the I/O pin is committed, thereby preventing the pin from being used as an additional input to the device.

In addition to overcoming the 22V10's shortcoming with pipelined systems by having input registers in both the input and I/O macrocells, the CY7C335 provides a solution to the 22V10's density problems with regards to counters and state machines by providing four buried registers. Additionally, pairs of macrocells have a shared input multiplexer that allows up to six additional inputs, even when all twelve

I/O macrocells have their registered outputs feeding back into the AND array.

The VHDL source code for this example is in Appendix I of this note, and the architecture is reprinted in *Figure 15*.

The up/down counter is implemented in three processes, a generation scheme, and two concurrent statements. In the first process, the lower limit is registered on the rising edge of CLK1. The signal LOWER registers the input signal LL. The second process registers the upper limit on the rising edge of CLK2. The third process implements (1) the up/down counter with reset, preset LOW, and preset HIGH, (2) two comparators, and (3) the direction signal (DIR) that indicates to count up (logic 1) or down (logic 0). The comparators and the direction signal are clocked by CLK0, forcing the counter to change direction from up to down or vice-versa two clock cycles after the count matches one of the limits. For this reason, the upper limit should be loaded with a value two less than the greatest desired count, and the lower limit should be loaded with a value two greater than the least desired count.

The generation scheme below the three processes is a means to instantiate 6 *bufoe* components (see *Figure 16*) and two *triout* components. The *bufoe* com-

```
use work.bv_math.all;
architecture behav of decode is
signal address: bit_vector(15 downto 0);
begin
  address <= a & "000";

proc1: process begin
  wait until clk = '1';
    promsel <= '0';
    shadowsel <= '0';
    periph1 <= '0';
    periph2 <= '0';
    sramsel <= '0';
    eesel <= '0';
    if valid = '1' then
      if address >= x"0000" and address < x"4000" then
        if bootover = '0' then
          promsel <= '1';
        else
          shadowsel <= '1';
        end if;
      elsif address >= x"4000" and address < x"4008" then
        periph1 <= '1';
      elsif address >= x"4008" and address < x"4010" then
        periph2 <= '1';
        elsif address >= x"8000" and address < x"C000" then
        sramsel <= '1';
      else address >= x"C000" then
        eesel <= '1';
      end if;
    end if;
  end process;
end behav;
```

Figure 12. VHDL Architecture

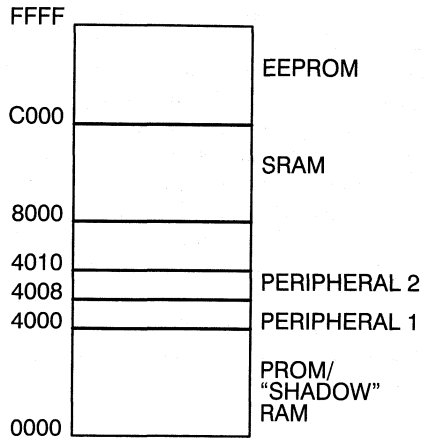


Figure 13. Decoder Memory Map

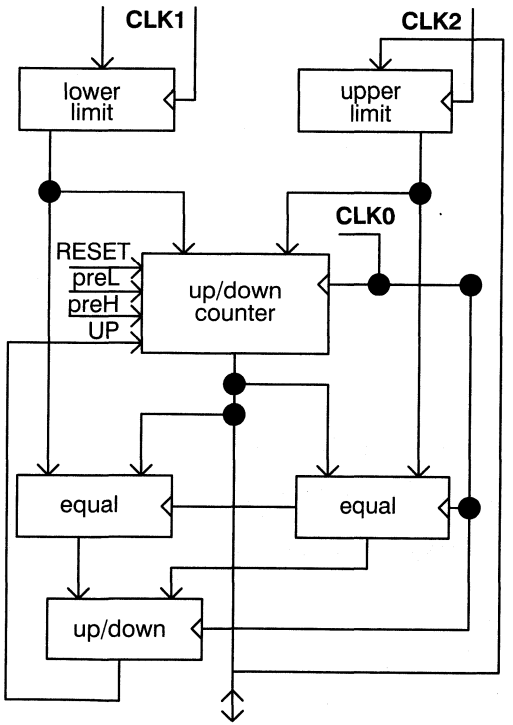


Figure 14. Up/Down Counter

ponents are used to implement the output enable and provide a feedback path for the upper limit. The CY7C335 has six shared input multiplexers that allow six bits of the signal count to utilize the state registers while enabling six bits of the upper limit to be loaded into the input register associated with the same macrocell. The remaining two bits of count will be placed in I/O macrocells in which the input registers are not used, and the two bits of the upper limit will be in two I/O macrocells configured as inputs. To enable bidirectional operation, the input and output pins for the associated upper limit and count bits can be connected externally. This is the reason for instantiating two triout components on the most significant two bits of the count.

Serial Decoder

The CY7C335's state registers and abundant product terms make it a good choice in which to implement state machines. The following VHDL code uses a state machine to implement a serial decoder that searches for a synchronization word within serially transmitted data. The sync word is the byte 11101000 and is expected to be repeated every 16 bytes. When the sync word is found, MATCH is asserted. When the sync word is found separated by 15 bytes three consecutive times, LOCK is asserted. The state diagram for this example is shown in *Figure 17.*

The architecture of this design is printed in *Figure 18* and the complete VHDL code is in Appendix K. The resources that this design uses (Appendix L) show that there is room for more logic within the device. For instance, the comparator with registered inputs described earlier could fit in the device along with this design.

The first process within the architecture defines the state transitions. The second process is one that is synchronized by the clock. The output MATCH is determined by the present inputs and the currents state. This implements a Mealy machine. The counter process counts the number of bits after a match, and the synchronizer process checks to see if a match occurs 15 bytes after the previous one. If a match is separated by 15 bytes for three consecutive times, then on the fourth consecutive match separated by 15 bytes, LOCK is asserted.

```
use work.bv_math.all;
use work.rtlpkg.all;

architecture archupdown of updown is
signal lower, upper, ul, count: bit_vector(0 to 7);
signal cequ, ceql, dir: bit;
begin
proc1: process
  begin
    wait until clk1 = '1';
      lower <= ll;
  end process;

proc2: process
  begin
    wait until clk2 = '1';
      upper <= ul;
  end process;

proc3: process
  begin
    wait until clk0 = '1';
    -- implement counter
    if reset = '1' then
      count <= x"00";
    elsif preL = '1' then
      count <= lower;
    elsif preH = '1' then
      count <= upper;
    elsif (dir = '1') then
      count <= inc_bv(count);
    else
      count <= dec_bv(count);
    end if;     -- implement comparators & direction signal
    if count = upper then
      cequ <= '1';
    else
      cequ <= '0';
    end if;
    if count = lower then
      ceql <= '1';
    else
      ceql <= '0';
    end if;
```

Figure 15. Architecture

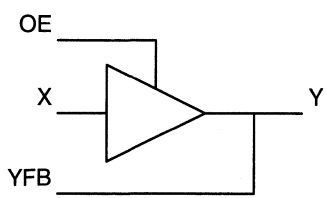
```
      if ceql = '1' then
        dir <= '1';
      elsif cequ = '1' then
        dir <= '0';
      else
        dir <= dir;
      end if;
    end process;

  g1: for i in 0 to 7 generate
    bidir: if i < 6 generate
      bx: bufoe port map(count(i), outen, countio(i), ul(i));
      end generate;
    trist: if i > 5 generate
      tx: triout port map(count(i), outen, countio(i));
      end generate;
    end generate;

    ul(6) <= ul6;
    ul(7) <= ul7;

  end archupdown;
```

Figure 15. Architecture (continued)




```
component bufoe
 port(
    x: in bit; --input to buffer
    oe: in bit; --output enable
    y: inout x01z; --x01z output
    yfb: out bit; -- feedback
    );
end component;
```

Figure 16. bufoe Component

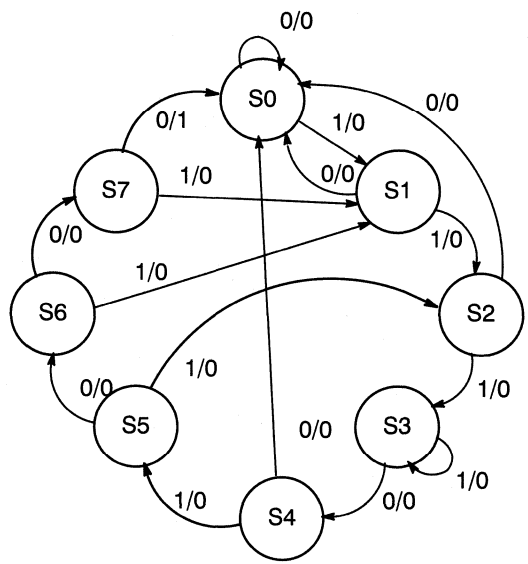


Figure 17. State Diagram

```
use work.int_math.all;
use work.bv_math.all;
architecture archserial of serial is
type states is (state0, state1, state2, state3, state4, state5, state6,
    state7);
signal state, nextstate: states;
signal match_cnt: bit_vector(1 downto 0);
signal bit_cnt: bit_vector(6 downto 0);

begin
fsm:
process begin
 match <= '0';
 case state is
   when state0 =>
     if data = '1' and (lock = '0' or bit_cnt = "1111000") then
       nextstate <= state1;
     else
       nextstate <= state0;
     end if;
   when state1 =>
     if data = '1' then
       nextstate <= state2;
     else
       nextstate <= state0;
     end if;
when state2 =>
     if data = '1' then
       nextstate <= state3;
     else
       nextstate <= state0;
     end if;
   when state3 =>
     if data = '0' then
       nextstate <= state4;
     else
       nextstate <= state3;
     end if;
   when state4 =>
     if data = '1' then
       nextstate <= state5;
     else
       nextstate <= state0;
     end if;
   when state5 =>
     if data = '0' then
       nextstate <= state6;
     else
       nextstate <= state2;
     end if;
```

Figure 18.

```
    when state6 =>
      if data = '0' then
        nextstate <= state7;
      else
        nextstate <= state1;
      end if;
    when state7 =>
      if data = '0' then
        nextstate <= state0;
        match <= '1';
      else
        nextstate <= state1;
      end if;
--No "when others" needed since CASE is completely defined.
 end case;
end process;

mealy:
process begin
 wait until clk = '1';
   state <= nextstate;
end process;

counter:
process begin
 wait until clk = '1';
 if match = '1' then
   bit_cnt <= "0000000";
 else
   bit_cnt <= inc_bv(bit_cnt);
 end if;
end process;

synchronizer:
process begin
 wait until clk = '1';
 if bit_cnt = "1111111" then
   if match = '1' then
     if match_cnt = "11" then
       lock <= '1';
     else
       match_cnt <= inc_bv(match_cnt);
     end if;
   else
     match_cnt <= "00";
     lock <= '0';
   end if;
 end if;
end process;

end archserial;
```

Figure 18. (continued)

Appendix A. *Warp2* VHDL Source Code for Pipelined Buffer

```
entity pipe is
  port( clk0, clk1, clk2: in bit;
    oe1, oe2: in bit;
    i: in bit_vector(7 downto 0);
    o: out x01z_vector(7 downto 0));
end pipe;

use work.rtlpkg.all;
architecture archpipe of pipe is
signal intmp, outtmp: bit_vector(7 downto 0);
signal ptoe: bit;
begin
proc1: process
  begin
    wait until clk1 = '1';
      intmp(7 downto 4) <= i(7 downto 4);
  end process;

proc2: process
  begin
    wait until clk2 = '1';
      intmp(3 downto 0) <= i(3 downto 0);
  end process;

proc3: process
  begin
    wait until clk0 = '1';
      outtmp <= intmp;
  end process;

  ptoe <= oe1 AND oe2;

g1: for j in 7 downto 0 generate
  g2: if j > 3 generate
      t1x: triout port map(outtmp(j), oe1, o(j));
    end generate;
  g3:  if j < 4 generate
      t2x: triout port map(outtmp(j), ptoe, o(j));
    end generate;
  end generate;

end archpipe;
```

Appendix B. *Warp2* Report File Excerpt for Pipelined Buffer

Information: Macrocell Utilization.

Description	Used	Max
Dedicated Inputs	9	9
Clock/Inputs	3	3
Enable/Inputs	1	1
I/O Macrocells	8	12
Buried Macrocells	0	4
	21 /	29 = 72 %

Information: Output Logic Product Term Utilization.

Node#	Output Signal Name	Used	Max
15	o_0_	1	9
16	Unused	0	19
17	o_2_	1	11
18	Unused	0	17
19	o_4_	1	13
20	o_7_	1	15
23	o_6_	1	15
24	o_5_	1	13
25	Unused	0	17
26	o_3_	1	11
27	Unused	0	19
28	o_1_	1	9
29	Unused	0	1
30	Unused	0	1
31	Unused	0	13
32	Unused	0	17
33	Unused	0	11
34	Unused	0	19
		8 / 230	= 3 %

Appendix C. *Warp2* Source Code for Comparator

```
entity comp is port (
  clk: in bit;
  a_in, b_in: bit_vector(0 to 8);
  aeqb: out bit);
end comp;

architecture archcomp of comp is
signal a, b: bit_vector(0 to 8);
begin
proc1: process begin
  wait until clk = '1';
    a <= a_in;
    b <= b_in;
  end process;

  aeqb <= '1' when a=b else '0';

end archcomp;
```

Appendix D. *Warp2* Report File Excerpt for Comparator

Information: Macrocell Utilization.

Description	Used	Max
Dedicated Inputs	9	9
Clock/Inputs	3	3
Enable/Inputs	1	1
I/O Macrocells	7	12
Buried Macrocells	0	4
	20 /	29 = 68 %

Information: Output Logic Product Term Utilization.

Node#	Output Signal Name	Used	Max
15	Used As Input	0	9
16	Used As Input	0	19
17	Used As Input	0	11
18	Used As Input	0	17
19	Used As Input	0	13
20	Used As Input	0	15
23	Unused	0	15
24	Unused	0	13
25	Unused	0	17
26	Unused	0	11
27	aeqb	18	19
28	Unused	0	9
29	Unused	0	1
30	Unused	0	1
31	Unused	0	13
32	Unused	0	17
33	Unused	0	11
34	Unused	0	19
		18 / 230	= 7 %

Appendix E. *Warp2* Source Code for Multiplexer

```
entity mux is port(
  clk, sel: in bit;
  xin, yin: in bit_vector(5 downto 0);
  qout: out bit_vector(5 downto 0));
end mux;

architecture archmux of mux is
signal x, y: bit_vector(5 downto 0);
begin
proc1: process begin
  wait until clk = '1';
    x <= xin;
    y <= yin;
    if sel = '1' then
      qout <= x;
    elsif sel = '0' then
      qout <= y;
    end if;
  end process;
end archmux;
```

Appendix F. *Warp2* Report File Excerpt for Multiplexer

Information: Macrocell Utilization.

Description	Used	Max
Dedicated Inputs	9	9
Clock/Inputs	3	3
Enable/Inputs	1	1
I/O Macrocells	7	12
Buried Macrocells	0	4
	20 /	29 = 68 %

Information: Output Logic Product Term Utilization.

Node#	Output Signal Name	Used	Max
15	qout_0_	2	9
16	Used As Input	0	19
17	qout_2_	2	11
18	Unused	0	17
19	qout_4_	2	13
20	Unused	0	15
23	Unused	0	15
24	qout_5_	2	13
25	Unused	0	17
26	qout_3_	2	11
27	Unused	0	19
28	qout_1_	2	9
29	Unused	0	1
30	Unused	0	1
31	Unused	0	13
32	Unused	0	17
33	Unused	0	11
34	Unused	0	19
		12 / 230	= 5 %

Appendix G. *Warp2* VHDL Source Code for Decoder

```
entity decode is port(
  a: in bit_vector(15 downto 3);
  rdwritebar, valid, bootover, clk: in bit;
  sramsel, promsel, eesel, shadowsel, periph1, periph2: out bit);
end decode;

use work.bv_math.all;
architecture behav of decode is
signal address: bit_vector(15 downto 0);
begin
  address <= a & "000";

proc1: process begin
  wait until clk = '1';
    promsel <= '0';
    shadowsel <= '0';
    periph1 <= '0';
    periph2 <= '0';
    sramsel <= '0';
    eesel <= '0';
    if valid = '1' then
      if address >= x"0000" and address < x"4000" then
        if bootover = '0' then
          promsel <= '1';
        else
          shadowsel <= '1';
        end if;
      elsif address >= x"4000" and address < x"4008" then
        periph1 <= '1';
      elsif address >= x"4008" and address < x"4010" then
        periph2 <= '1';
        elsif address >= x"8000" and address < x"C000" then
        sramsel <= '1';
      else address >= x"C000" then
        eesel <= '1';
      end if;
    end if;
  end process;
end behav;
```

Appendix H. *Warp2* Report File Excerpt for Decoder

Information: Macrocell Utilization.

Description	Used	Max
Dedicated Inputs	9	9
Clock/Inputs	3	3
Enable/Inputs	1	1
I/O Macrocells	9	12
Buried Macrocells	0	4
	22 /	29 = 75 %

Information: Output Logic Product Term Utilization.

Node#	Output Signal Name	Used	Max
15	eesel	1	9
16	Used As Input	0	19
17	periph2	1	11
18	Used As Input	0	17
19	shadowsel	1	13
20	Used As Input	0	15
23	Unused	0	15
24	promsel	1	13
25	Unused	0	17
26	periph1	1	11
27	Unused	0	19
28	sramsel	1	9
29	Unused	0	1
30	Unused	0	1
31	Unused	0	13
32	Unused	0	17
33	Unused	0	11
34	Unused	0	19
		6 / 230	= 2 %

Appendix I. *Warp2* Source Code for UpDown

```
entity updown is
  port( clk0, clk1, clk2: in bit;
    outen, preL, preH, reset: in bit;
    ll: in bit_vector(0 to 7);
    ul6, ul7: in bit;
    countio: inout x01z_vector(0 to 7));
end updown;

use work.bv_math.all;
use work.rtlpkg.all;

architecture archupdown of updown is
signal lower, upper, ul, count: bit_vector(0 to 7);
signal cequ, ceql, dir: bit;
begin
proc1: process
  begin
    wait until clk1 = '1';
      lower <= ll;
  end process;

proc2: process
  begin
    wait until clk2 = '1';
      upper <= ul;
  end process;

proc3: process
  begin
    wait until clk0 = '1';
    if reset = '1' then
      count <= x"00";
    elsif preL = '1' then
      count <= lower;
    elsif preH = '1' then
      count <= upper;
    elsif (dir = '1') then
      count <= inc_bv(count);
    else
      count <= dec_bv(count);
    end if;
  end process;

proc4: process
  begin
    wait until clk0 = '1';
    if count = upper then
      cequ <= '1';
```

Appendix I. *Warp2* Source Code for UpDown (continued)

```
    else
      cequ <= '0';
    end if;
    if count = lower then
      ceql <= '1';
    else
      ceql <= '0';
    end if;
    if ceql = '1' then
      dir <= '1';
    elsif cequ = '1' then
      dir <= '0';
    else
      dir <= dir;
    end if;
  end process;

g1: for i in 0 to 7 generate
  bidir: if i < 6 generate
    bx: bufoe port map(count(i), outen, countio(i), ul(i));
    end generate;
  trist: if i > 5 generate
    tx: triout port map(count(i), outen, countio(i));
    end generate;
  end generate;

  ul(6) <= ul6;
  ul(7) <= ul7;

end archupdown;
```

Appendix J. *Warp2* Report File Excerpt for UpDown

Information: Macrocell Utilization.

Description	Used	Max
Dedicated Inputs	9	9
Clock/Inputs	3	3
Enable/Inputs	1	1
I/O Macrocells	12	12
Buried Macrocells	3	4
	28 /	29 = 96 %

Information: Output Logic Product Term Utilization.

Node#	Output Signal Name	Used	Max
15	countio_2_	7	9
16	Used As Input	0	19
17	countio_0_	3	11
18	Used As Input	0	17
19	countio_6_	7	13
20	countio_4_	7	15
23	Used As Input	0	15
24	countio_5_	7	13
25	countio_3_	7	17
26	countio_7_	7	11
27	Used As Input	0	19
28	countio_1_	6	9
29	Unused	0	1
30	Unused	0	1
31	Unused	0	13
32	ceql_BEH_i27..	16	17
33	dir	2	11
34	cequ	16	19
		85 / 230	= 36 %

Appendix K. *Warp2* VHDL Source Code for Serial Decoder

```
entity serial is port(
  clk, reset, data: in bit;
  match: buffer bit;
  lock: buffer bit);
end serial;

use work.int_math.all;
use work.bv_math.all;
architecture archserial of serial is
type states is (state0, state1, state2, state3, state4, state5, state6,
    state7);
signal state, nextstate: states;
signal match_cnt: bit_vector(1 downto 0);
signal bit_cnt: bit_vector(6 downto 0);

begin
fsm:
process  begin
  match <= '0';
  case state is
    when state0 =>
      if data = '1' and (lock = '0' or bit_cnt = "1111000") then
        nextstate <= state1;
      else
        nextstate <= state0;
      end if;
    when state1 =>
      if data = '1' then
        nextstate <= state2;
      else
        nextstate <= state0;
      end if;
    when state2 =>
      if data = '1' then
        nextstate <= state3;
      else
        nextstate <= state0;
      end if;
    when state3 =>
      if data = '0' then
        nextstate <= state4;
      else
        nextstate <= state3;
      end if;
    when state4 =>
      if data = '1' then
        nextstate <= state5;
      else
```

Appendix K. *Warp2* VHDL Source Code for Serial Decoder (continued)

```
        nextstate <= state0;
      end if;
    when state5 =>
      if data = '0' then
        nextstate <= state6;
      else
        nextstate <= state2;
      end if;
    when state6 =>
      if data = '0' then
        nextstate <= state7;
      else
        nextstate <= state1;
      end if;
    when state7 =>
      if data = '0' then
        nextstate <= state0;
        match <= '1';
      else
        nextstate <= state1;
      end if;
  end case;
end process;

mealy:
process begin
  wait until clk = '1';
    state <= nextstate;
end process;

counter:
process begin
  wait until clk = '1';
  if match = '1' then
    bit_cnt <= "0000000";
  else
    bit_cnt <= inc_bv(bit_cnt);
  end if;
end process;

synchronizer:
process begin
  wait until clk = '1';
  if bit_cnt = "1111111" then
    if match = '1' then
      if match_cnt = "11" then
        lock <= '1';
```

Appendix K. *Warp2* VHDL Source Code for Serial Decoder (continued)

```
      else
        match_cnt <= inc_bv(match_cnt);
      end if;
    else
      match_cnt <= "00";
      lock <= '0';
    end if;
  end if;
end process;

end archserial;
```

Appendix L. *Warp2* Report File Excerpt for Serial Decoder

```
Information: Macrocell Utilization.
```

Description	Used	Max
Dedicated Inputs	0	9
Clock/Inputs	1	3
Enable/Inputs	0	1
I/O Macrocells	10	12
Buried Macrocells	4	4
	15 /	29 = 51 %

```
Information: Output Logic Product Term Utilization.
```

Node#	Output Signal Name	Used	Max
15	lock	9	9
16	Unused	0	19
17	data	5	11
18	bit_cnt_0_	1	17
19	serial_0_sta..	4	13
20	bit_cnt_2_	3	15
23	bit_cnt_1_	2	15
24	serial_0_sta..	4	13
25	match	1	17
26	bit_cnt_3_	4	11
27	Unused	0	19
28	bit_cnt_4_	5	9
29	Unused	0	1
30	Unused	0	1
31	match_cnt_0_	8	13
32	bit_cnt_6_	7	17
33	match_cnt_1_	9	11
34	bit_cnt_5_	6	19
		68 / 230	= 29 %

Document #: PLD0003–0194

Windows is a trademark of Microsoft Corporation.
OpenLook is a trademark of UNIX System Laboratories.
Motif is a trademark of Open Software Foundation, Inc.
Warp2 is a trademark of Cypress Semiconductor Corporation.

The FLASH370™ Family Of CPLDs and Designing with *Warp2*™

This application note covers the following topics: (1) a general discussion of complex programmable logic devices (CPLDs), (2) an overview of the CY7C370 family of CPLDs, and (3) using the *Warp2* VHDL Compiler for the CY7C370 family.

Overview of CPLDs

CPLDs extend the concept of the PLD to a higher level of integration to improve system performance, use less board space, improve reliability, and reduce cost. Instead of making the PLD bigger with more input terms and product terms, a CPLD architecture is composed of multiple PLDs or logic blocks (LABs) connected together with a programmable interconnect matrix (PIM). Multiple Logic Array Blocks (LABs) provide comparable speed to a PLD because the basic propagation path is through one LAB and each LABs product term array is comparable to a PLD array. Multiple LABs provide the higher integration. The number of LABs in a CPLD is typically between 2 for the smaller CPLDs and 16 for the larger ones. In addition to LABs interconnected by the PIM, are the input/output macrocells and the dedicated input macrocells. *Figures 1* and *2* show the CPLD generic block diagram and the logic block diagram respectively.

The architectural components of the LAB are: (1) the product term array, (2) the product term allocator, and (3) the macrocell. The product term array is the same in the CPLD as in the PLD except that the inputs into the array can now also come from the PIM. The product term allocator is a new concept in the CPLD where product terms are not fixed to a macrocell with its associated input/output pin but

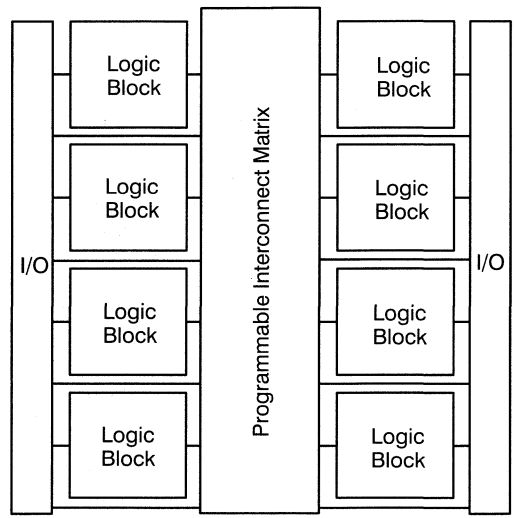


Figure 1. Generic Block Diagram

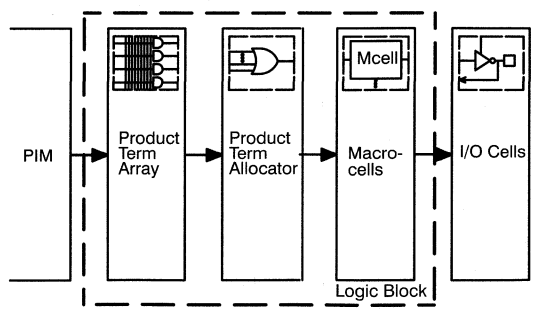


Figure 2. Logic Block Diagram

can be routed to different macrocells depending on where they are needed. The result is a more efficient allocation of product terms and higher integration. Implementation of the product term allocator varies across CPLD vendors which is more fully discussed in the section describing the features of the CY7C370 family.

The macrocell accepts the single output of the product term allocator which is the ORing of a variable number of product terms. In some macrocells this input feeds into a two input XOR gate with the other input potentially carrying the Q feedback. This configures the D flip flop to a T flip flop which can provide an improvement in capacity for certain designs such as counters. After the XOR gate, the macrocell is configurable as registered, combinatorial, and in some cases latched. There are two kinds of macrocells which are input/output dedicated and buried. Dedicated macrocells output to the input/output macrocell and also provide feedback into the product term array. Buried macrocells only provide feedback into the product term array.

The function of the PIM is to distribute the needed fraction of the total available resources, all outputs from the LAB and possibly also dedicated inputs and inputs/outputs, to the appropriate LAB. There are two common methods of PIM implementation: array based interconnect and mux based interconnect.

Figure 3 shows the data path of communication between two LABs using the array based interconnect. In the array based interconnect, each output of the LAB can potentially connect to any number of PIM input terms through a memory element. Each PIM input term is assigned to a specific LAB and functions as an input term into the LABs product term array. In this example only four PIM input terms are shown two going to LAB1 and two going to LAB2. There is a sense amp per input term to detect the logic level, buffer the signal, and drive it into the LAB. The true and complement of the PIM signal feed into the product term array (not shown in the figure). Since every LAB output can connect to any PIM input, the interconnect is considered 100 percent routable. It never limits the ability of the device to fit logic. A macrocell output can connect to one or multiple PIM input terms. The major drawback from using a memory element as an interconnect is the slower propagation delay than the muxed based interconnect.

Figure 4 shows the data path of communication between two LABs using the muxed based interconnect. In the muxed based interconnect a mux chooses one of a number of potential PIM input terms into the LAB. The PIM input terms differ from the array based interconnect in that they are output from a 1 of n (where "n" is the number of inputs of the mux) mux instead of the output of a wired nor memory array. The inputs into the muxes are all the outputs of the LABs as well as dedicated inputs and input/output pins. *Figure 3* shows two PIM input terms output from two 4-to-1 muxes. In this example, macrocell 2 from LAB1 and macrocell 2 from LAB2 both show 2 chances to route into the muxes with other inputs having only 1 chance The wider the mux (the number of inputs into the mux) the more likely all desired inputs into each LAB will be successfully routed and the more chances each signal gets to route into a LAB. The disadvantage of larger muxes is a larger slower propagation delay through the PIM and increased die size. Implementations of mux-based interconnect vary in the size of the mux.

Features of the FLASH370 CPLDs

The FLASH370 family of CPLDs offers densities from 2 to 16 LABs . *Figure 5* shows the block diagram of the CY7C374/5 with 8 LABs. The even numbers of the family (372,374,376) bury half of the macrocells for maximum integration with the same pinout as the (371,373,375) respectively. The 377 does not have a corresponding equivalent pinout with buried macrocells. *Table 4* shows the family members offered.

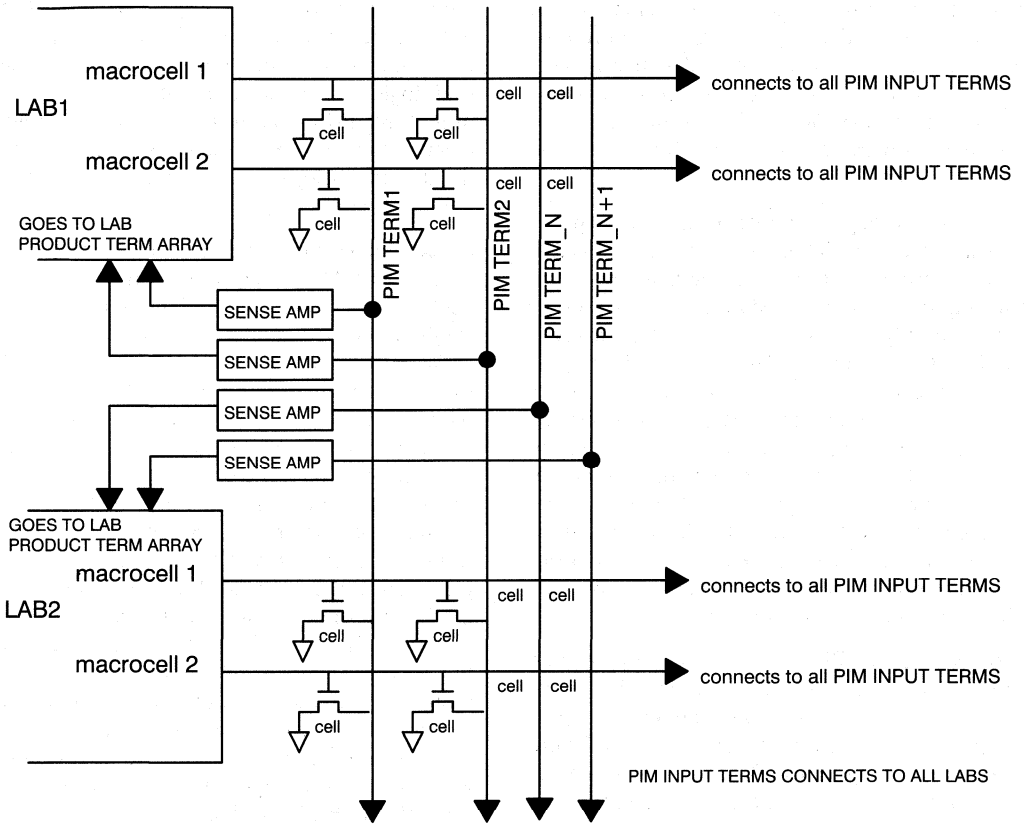


Figure 3. Array-Based Interconnect

Table 4. FLASH370 Family Members

Feature	CY7C371	CY7C372/3	CY7C374/5	CY7C376/7
Macrocells	32	64	128	256
Dedicated Inputs	6	6	6	6
I/O pins	32	32/64	64/128	128/256
Dedicated Inputs Usable as Clocks	2	2/4	4/4	4/4
Speed (t_{PD})	8.5 ns	10 ns	12 ns	15 ns
Primary Packages	44-PLCC	44/84-PLCC 100-TQFP	84-PLCC 100/160-TQFP	160-TQFP 289-BGA

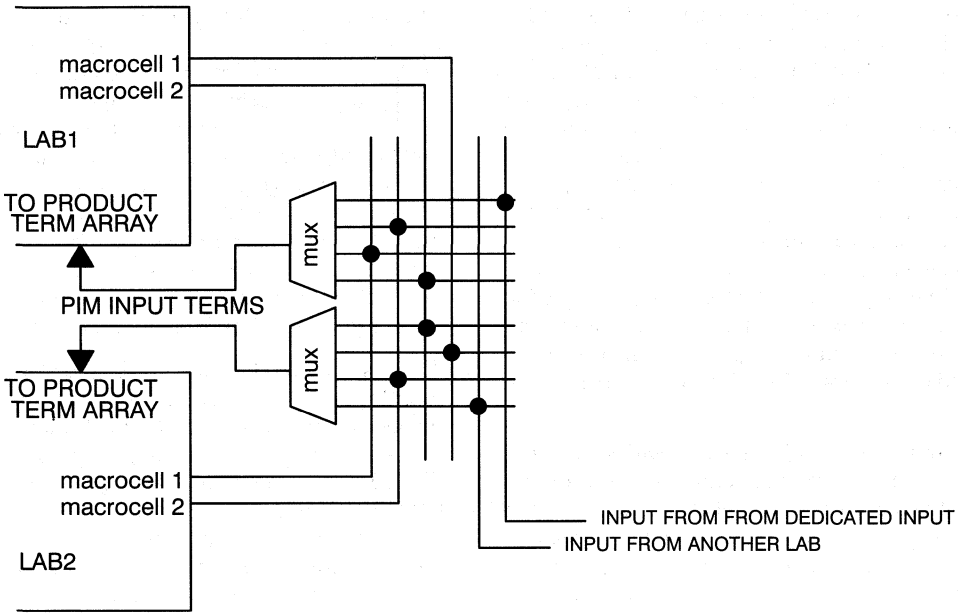


Figure 4. Mux Based Interconnect

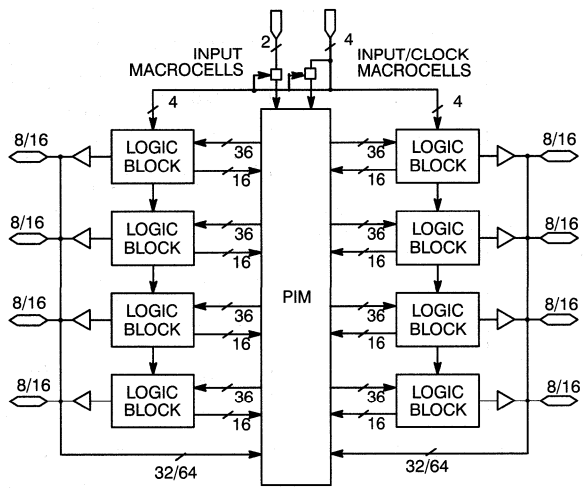


Figure 5. CY7C374/5 Block Diagram

Figures 6 and *7* show the product term array, product term allocator, macrocells, and input/output macrocells for the CY7C370 family. Each LAB features 36 inputs, which can adequately handle 32-bit operations plus control signals with one pass through the LAB. The product term array features the true and complement polarities of each PIM output signal for a total of 72 inputs. 80 standard product terms are provided to the product term allocator which allocates from 0 to 16 product terms to each of the 16 macrocells. Additionally, 6 special product terms are also generated in the product term array. They are an asynchronous preset, asynchronous reset, and two groups of 2 bank output enable product terms.

The output macrocell (*Figure 8*) provides a selection of four output controlling options: (1) control from one output enable, (2) control from a second output enable, (3) permanently enabled, or (4) permanently disabled. Each LAB contains 4 output enable product terms, 2 for the upper 8 macrocells and 2 for the lower 8 macrocells.

The state macrocell (*Figure 8*) contains options to register, latch, or send data through combinatorially. For the input/output macrocell there is an additional output polarity mux to improve capacity before the signal goes to the input/output macrocell. For buried macrocells there is an additional mux which can configure the state register as an input register. If the buried macrocell is configured as an input, zero product terms will be allocated from the array. In *Figure 8* architecture bit C7 can choose the feedback from the input/output pin as the input into the register instead of from the product term array.

There is one asynchronous preset and reset product term for each LAB. There are polarity muxes for the clocks, preset and reset. Each macrocell can choose among two clocking options for the CY7C371/372 and four clocking options for the CY7C373/374/375/376/377. All macrocells in a LAB receive the same polarity of the clock, set and reset. Polarities are configurable per LAB. *Figure 8* shows the input/output macrocell and input/output plus buried macrocell.

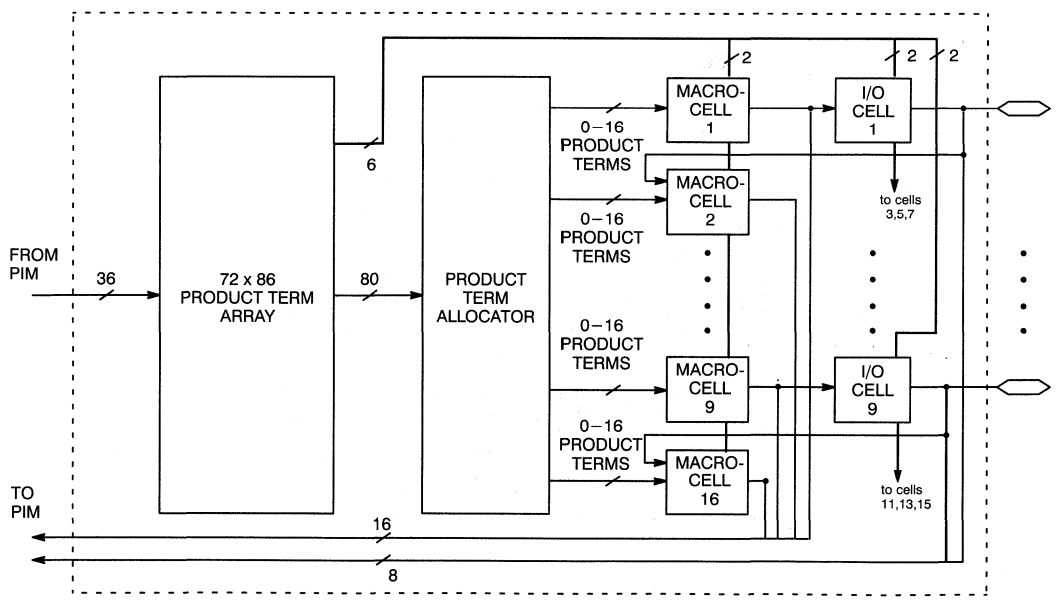


Figure 6. Logic Block for CY7C372, CY7C374, and CY7C376 (Register Intensive)

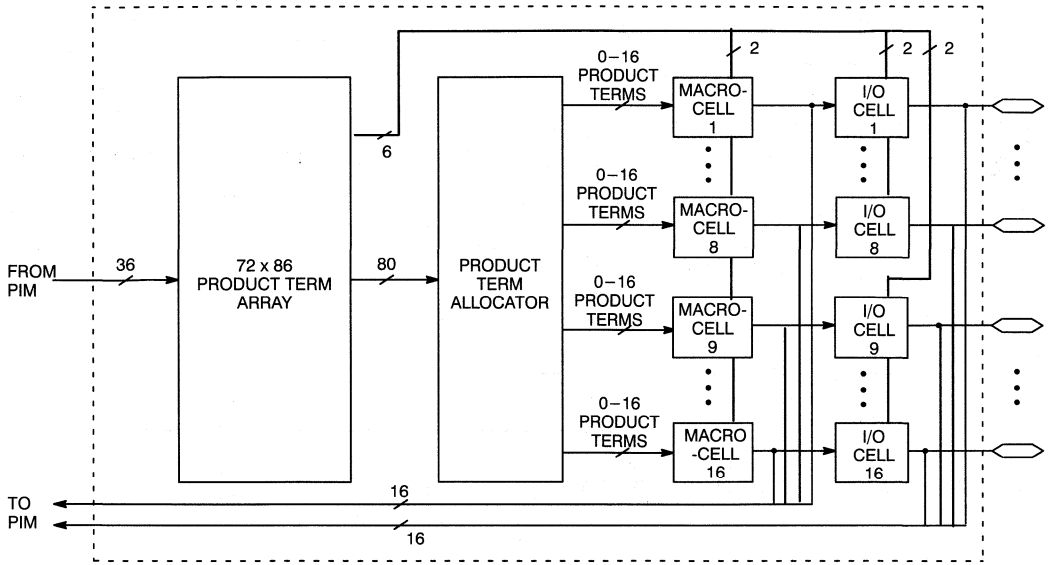


Figure 7. Logic Block for CY7C371, CY7C373, CY7C375, and CY7C377 (I/O Intensive)

Figure 9 and *10* show the input/clock and input macrocells. The input macrocell provides the flexibility to let the input enter combinatorially, latched, single registered, or double registered (for maximum metastability performance). For the 371/372 there are two input/clocks pins and four input pins. For the CY7C373/374/375/376/377 there are four input/clock pins and two input pins. For added flexibility, each clock can be configurable for either positive or negative polarity.

In order to fully understand the operation of the CY7C370 product term allocator, two important aspects of product term allocator design need to be introduced: product term steering and product term sharing. Steering refers to the assignment of a product term resource to a macrocell. In the traditional PLD there is no steering flexibility. Each macrocell has assigned product terms that can only be used by that macrocell. In many designs each macrocell requires a different number of product terms putting an emphasis on the ability to allocate product terms individually on an as needed basis. Product term sharing refers to a product term being used by multiple macrocells. The logic equations for different macrocells sometimes contain the same minterm. Instead of generating this same minterm multiple times, it is generated on only one product term and shared across macrocells, thereby improving capacity.

Figure 11 is a conceptual representation of the CY7C370 product term allocator. The product term allocator functions like a segmented OR array by ORing from 0 to 16 product terms for each macrocell. Product terms can be steered and shared on an individual basis. This architecture has several advantages over other implementations that steer product terms away from one macrocell to serve another.

Figure 12 is a conceptual representation of the MACH™ product term allocator. It shows no ability to share product terms across macrocells. Each cluster of four product terms can route to only one macrocell. The product terms are routed in groups of four which is a much higher granularity of product term allocation and not as efficient.

To demonstrate this inefficiency, consider a macrocell that needs five product terms to implement its logic. Two product term clusters with a total of eight

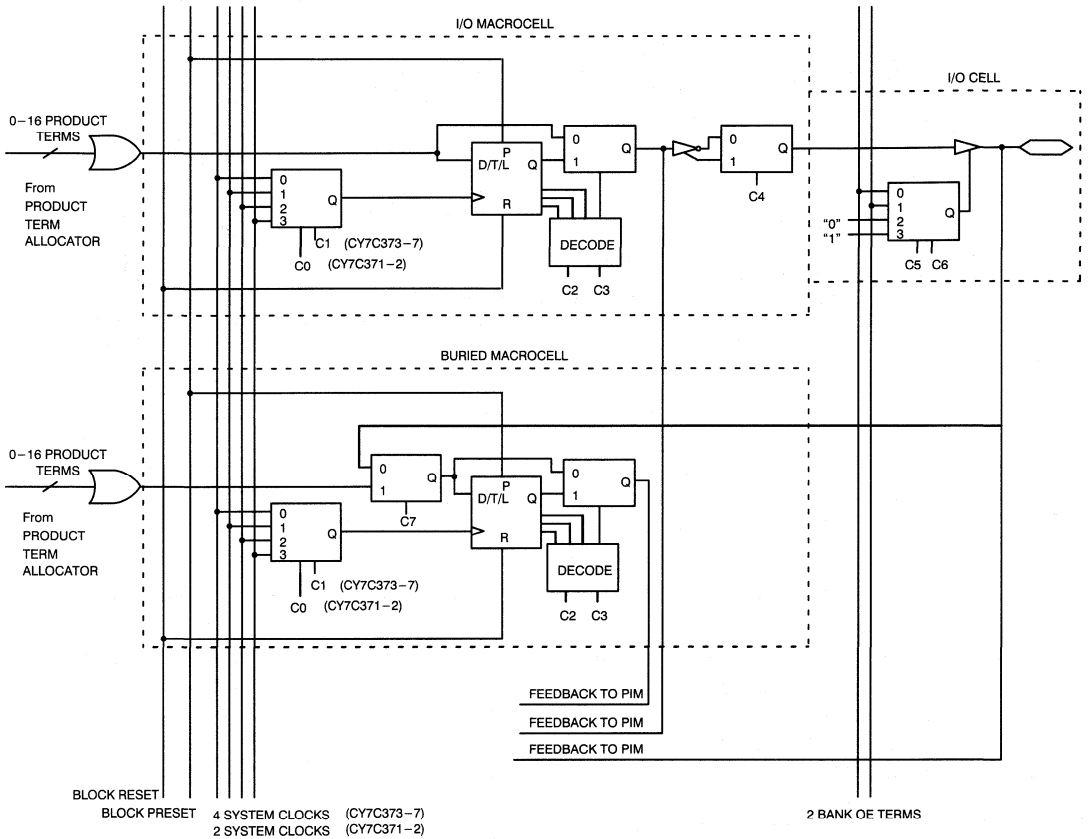


Figure 8. I/O and Buried Macrocells

available product terms are needed. This wastes the resources of three product terms from the borrowed cluster since these product terms can not be re-routed to another macrocell.

The MAX7000™ product term allocator representation (*Figure 13*) shows the use of expander terms. Expander terms allow two passes through the array which can produce very high capacity. These expanders are also shared among all product terms in the LAB. The problem with using the expanders is in the additional propagation delay of two passes through the array. This complicates the timing model and links the performance of the device to the use of the expander product terms. As with the MACH product term allocator, the MAX7000 allocator also has five product term clusters. It therefore suffers from the same problem of product term wasting when more than one cluster is routed to a macrocell.

The CY7C370 product term allocator provides the most effective method of steering and sharing product terms. The propagation of signals through the product term allocator is independent of the number of product terms allocated to each macrocell. Additionally the flexibility of this product term allocator, with the PIM, enables a change in the design without a modification to the external pinout of the device. There is no need for input and output switch matrices, which add extra delay and degrade performance.

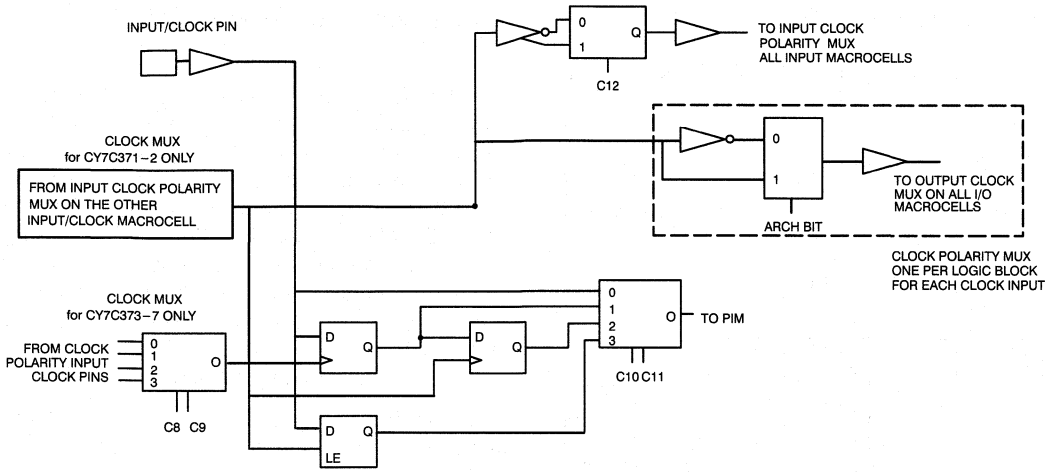


Figure 9. Input/Clock Pins

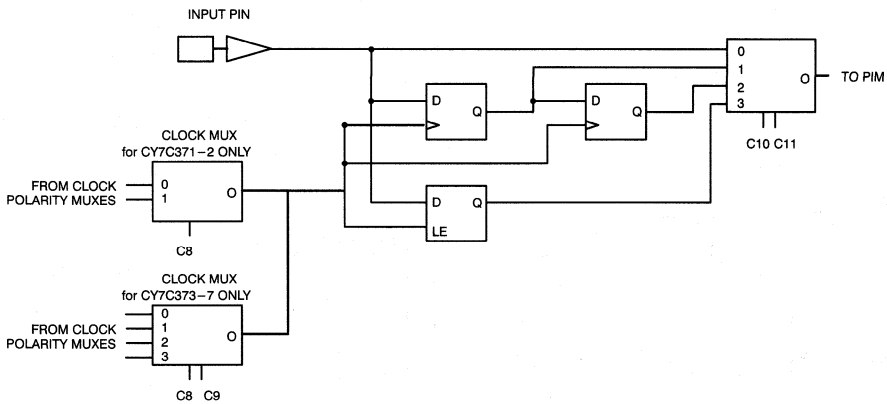


Figure 10. Input Pins

The timing model of the CY7C370 family is far simpler than for other CPLD solutions for two reasons. First, all input signals into the LAB pass through the PIM. This includes all input/outputs, feedbacks from macrocell outputs, and dedicated inputs. Secondly, the propagation time through the product term allocator is independent of the number of product terms allocated to a macrocell. As a result, there are no expander delays, no dedicated versus input/output pin delays, no penalties for using up to 16 product terms, or no delay penalties for steering and or sharing product terms. The CY7C370 family of products provides timing as predictable as PLDs like the 22V10.

The PIM in the CY7C370 was designed to approach the 100 percent routability of the array based interconnect but not made so wide that performance and die size suffered.

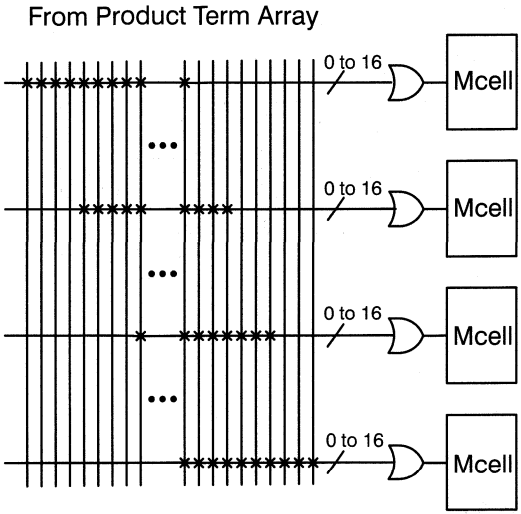


Figure 11. CY7C370 Product Term Allocator Representation

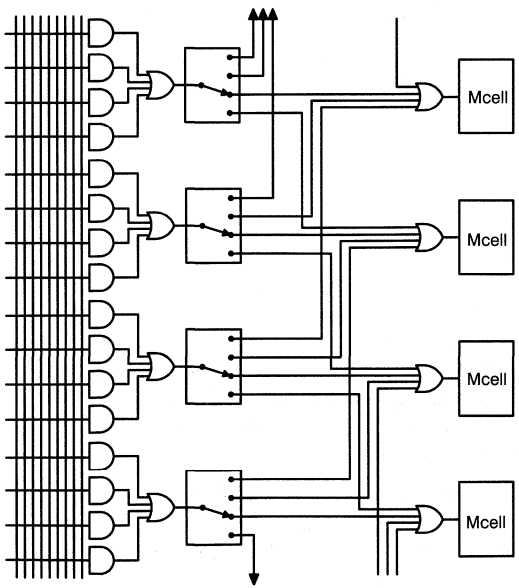


Figure 12. MACH Product Term Allocator Representation

Using *Warp*™ to Design with the CY7C370

Development software is extremely important for ease of use and efficiency of resource allocation when designing with CPLDs. Cypress offers two software packages that will fully support the CY7C370 family of products as well as all other PLDs, FPGAs, and state machine PROMs. *Warp2* provides full VHDL language support which is becoming the industry standard for describing hardware design. A functional simulator is also provided. *Warp3* additionally includes schematic capture and exact timing simulation capability.

The simplified timing model of the CY7C370 often makes exact timing simulation unnecessary because performance can be predicted directly from the datasheet. Therefore the functional simulator of *Warp2* may be a cost effective design solution. With *Warp* no manual intervention for fitting the designs into the devices are necessary. In addition to *Warp*, customers also have third party support from a variety of vendors.

Warp products take in VHDL designs and automatically fit them into the chosen device. The following

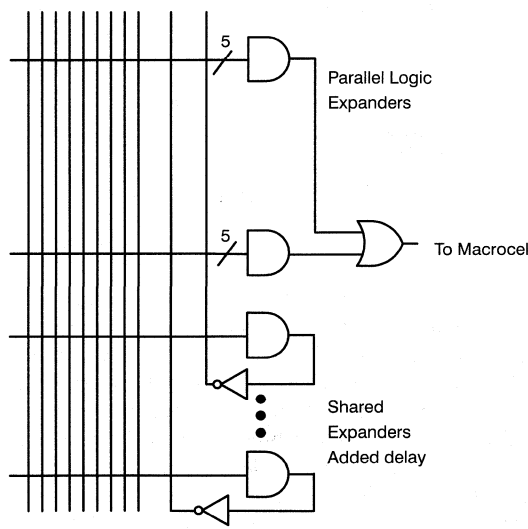


Figure 13. MAX Product Term Allocator Representation

section explains how to exploit the special features of the CY7C370 with VHDL. A thorough treatment of VHDL constructs is found in the *Warp2* Reference Manual. Topics covered here are: (1) using the single/double registered options for the dedicated inputs, and registering signals from the IO pins, (2) using the clock polarity mux feature, (3) describing registered versus latched versus combinatorial outputs, (4) using the output enable feature, (5) using the asynchronous preset/reset feature, and (6) Using the buried registers as for the (372/4/6).

To register the dedicated inputs one or two signals must be defined to represent the additional nodes for one and two registers respectively. Appendix A demonstrates how to use single and double registered inputs for a 4 bit loadable counter. In *proc2*, RESET1 and RESET2 are the outputs of the first and second registers. It requires 2 passes through *proc2* to activate RESET2. Signal RESET2 is then used in *proc1* to perform the reset. *Proc2* additionally registers the data to be loaded with the statement `regin <=temp.dat`. The signal REGIN is then used in process *Proc1* to load the counter with the statement `temp.cnt <= regin`. If the same clock is used for the inputs as for the state registers, then the statements in process *proc2* could be incorporated into *proc1* and only one process is needed. The assignment of the entity output pins is handled by the instantiation of the *bufoe* component (called in the statement `use work.rtlpkg.all`), which takes the signal TEMP.CNT as input and transfers it to the output (in this case called COUNT) when the output enable control (called OUTEN) is HIGH. Registering the inputs from the input/output pins is better suited for the 372/374/376 members of the family since the signal does not need to go through the PIM and logic block.

Clocking on the falling instead of the rising edge of the clock is simply done by changing the statement `wait until (clk = '1')` to `wait until (clk = '0')`. Events occurring on the rising and falling edge of a clock can be incorporated into the same design by defining a separate process for the event, provided that sufficient logic blocks are available.

VHDL describing combinatorial and registered outputs is identical to other part implementations as with the CY7C370. The registered equations must be inserted inside a process and after a `wait until clock=` statement.

Appendix B shows an example of how to implement the combinatorial macrocell option with maximum usage of output enable flexibility for the CY7C371. A total of eight different input signals control the output enable functionality. The entire function is handled by the *bufoe* component where the input into the buffer is the external input pin. No signals are necessary.

The latch option is unique to the CY7C370 family. Appendix C shows an example of how to latch a signal using the IF-THEN-ELSE construct. In this example the signal is latched when the clock is HIGH by setting the signal value to itself with the statements `signala <= signala` and `signalb <= signalb`. When the clock is LOW the path is combinatorial and the signal value gets the input. This is handled in the code `if clk='0' then signala <= inputa; signalb <= inputb`. Two signals are defined, SIGNALA and SIGNALB, to latch the data when the clock is in the right polarity (in this case HIGH).

Appendix D shows the full registered configuration. As in Appendix C, the signals SIGNALA and SIGNALB are defined and the function of the register is defined within a process. On the rising edge of the clock, SIGNALA gets INPUTA and SIGNALB gets INPUTB.

Appendix E uses latches for the output enable control. Signals need to be generated from the array and are passed as the output enable parameter into the *triout* component. This function behaves similarly to the *bufoe* but does not include the feedback parameter.

Appendix F shows how to use the buried registers to implement the least significant bits in a counter. A bit vector signal is defined to represent all the register states. Those states that are needed as outputs are assigned to the entity output pins outside of the process with the statement `count (0 to 11) <= fullcnt (4 to 15)`. If output enable control is desired then this last statement is omitted and

the signal to output assignment is handled with the *bufoe* component.

Appendix G is the same as Appendix F except that the registers are reset asynchronously. The format of the process is much different from Appendix F but functions exactly the same except for the asynchronous instead of synchronous reset. The process uses a "sensitivity list" that includes all the parameters that will activate the process. The synchronous part of the process is initiated by the statement `clk'event and clk='1'` instead of `wait until clk='1'`. The asynchronous preset/reset is similar to other Cypress PLDs except for the additional polarity mux feature that enables active HIGH or LOW. To specify clock polarity, the VHDL construct for active HIGH is `if reset = '1' then` and for active LOW is `if reset = '0' then`.

Appendix A. inregcnt

```
-- The bufoe port map parameters are:
-- bufoe port map(signal going to the input of the tristateable buffer,
--     tristate control signal,
--     the output signal that is the entity output pin,
--     the feedback signal from the entity input/output pin)
-- In this example the last entry is "open" meaning no feedback.

USE work.bv_math.all;          -- necessary for inc_bv();
USE work.rtlpkg.all;           -- necessary for bufoe

ENTITY inregcnt IS
       PORT   (clk, clkin, reset, load, outen: IN bit;
         count: INOUT x01z_VECTOR(0 TO 3));
END inregcnt;

ARCHITECTURE behavior OF inregcnt IS
TYPE bufRec IS                 -- record for bufoe
       RECORD                  -- inputs and feedback
            cnt: bit_vector(0 TO 3);
            dat: bit_vector(0 TO 3);
END RECORD;
SIGNAL temp: bufRec;
SIGNAL regin: bit_vector(0 to 3);-- for registering input loaded data
SIGNAL reset1, reset2:bit;      -- for registering the reset input
CONSTANT counterSize: integer := 3;
BEGIN
g1:    FOR i IN 0 TO counterSize GENERATE
            bx: bufoe PORT MAP(temp.cnt(i), outen, count(i), temp.dat(i));
       END GENERATE;
proc1: PROCESS
       BEGIN
            WAIT UNTIL (clk = '1');
            IF reset2 = '1' THEN -- uses the double registered signal
                 temp.cnt <= "0000";
            ELSIF load = '1' THEN
                 temp.cnt <= regin; -- uses the single registered signal
            ELSE
                 temp.cnt <= inc_bv(temp.cnt);  -- increment bit vector
            END IF;
       END PROCESS;
-- Proc2 single registers the load operation and double registers the reset
-- operation.  Note the two clkin's are needed for the double register.
proc2: PROCESS
       BEGIN
         WAIT UNTIL (clkin = '1');
          regin <= temp.dat; --single register for data load
          reset1 <= reset; --single register the reset signal
          reset2 <= reset1;--double register the reset signal
       END PROCESS;
END behavior;
```

Appendix B. usecomb

```
--uses the full functionality of the oe features of the 371.
--macrocell is in combinatorial mode

USE work.rtlpkg.all;

ENTITY usecomb IS
       PORT (outen1, outen2, outen3, outen4, outen5, outen6, outen7,
           outen8; IN bit; inputa, inputb: IN bit_vector(0 to 1);
           outa,outb: INOUT x01z_vector(0 to 7));
END usecomb;

ARCHITECTURE behavior OF usecomb IS
BEGIN
g1:    FOR i IN 0 TO 1 GENERATE
           bx1: bufoe PORT MAP(inputa(i), outen1, outa(i), open);
           bx2: bufoe PORT MAP(inputa(i), outen2, outa(i+2), open);
           bx3: bufoe PORT MAP(inputa(i), outen3, outa(i+4), open);
           bx4: bufoe PORT MAP(inputa(i), outen4, outa(i+6), open);
           bx5: bufoe PORT MAP(inputb(i), outen5, outb(i), open);
           bx6: bufoe PORT MAP(inputb(i), outen6, outb(i+2), open);
           bx7: bufoe PORT MAP(inputb(i), outen7, outb(i+4), open);
           bx8: bufoe PORT MAP(inputb(i), outen8, outb(i+6), open);
       END GENERATE;
END behavior;
```

Appendix C. uselatch

```
--uses the full functionality of the oe features of the 371.
--macrocell in latched mode

USE work.rtlpkg.all;

ENTITY uselatch IS
       PORT   (clk, outen1, outen2, outen3, outen4, outen5, outen6, outen7,
            outen8: IN bit;
            inputa, inputb: IN bit_vector(0 to 1);
            outa,outb: INOUT x01z_vector(0 to 7));
END uselatch;

ARCHITECTURE behavior OF uselatch IS
SIGNAL signala, signalb: bit_vector(0 to 1);
BEGIN
g1:    FOR i IN 0 TO 1 GENERATE
            bx1: bufoe PORT MAP(signala(i), outen1, outa(i), open);
            bx2: bufoe PORT MAP(signala(i), outen2, outa(i+2), open);
            bx3: bufoe PORT MAP(signala(i), outen3, outa(i+4), open);
            bx4: bufoe PORT MAP(signala(i), outen4, outa(i+6), open);
            bx5: bufoe PORT MAP(signalb(i), outen5, outb(i), open);
            bx6: bufoe PORT MAP(signalb(i), outen6, outb(i+2), open);
            bx7: bufoe PORT MAP(signalb(i), outen7, outb(i+4), open);
            bx8: bufoe PORT MAP(signalb(i), outen8, outb(i+6), open);
       END GENERATE;--the clk input is an active low latch enable
--the if then construct must be within a process.
PROCESS
  BEGIN
    IF clk='0' then
      signala <= inputa;
      signalb <= inputb;
    ELSE
      signala <= signala;
      signalb <= signalb;
    END IF;
  END PROCESS;
END behavior;
```

Appendix D. usereg

```
--macrocell in registered mode

ENTITY usereg IS
       PORT   (clk, outen1, outen2, outen3, outen4, outen5, outen6, outen7,
            outen8: IN bit; inputa, inputb: IN bit_vector(0 to 1);
            outa,outb: INOUT x01z_vector(0 to 7));
END usereg;

ARCHITECTURE behavior OF usereg IS
SIGNAL signala, signalb: bit_vector(0 to 1);
BEGIN
g1:    FOR i IN 0 TO 1 GENERATE
            bx1: bufoe PORT MAP(signala(i), outen1, outa(i), open);
            bx2: bufoe PORT MAP(signala(i), outen2, outa(i+2), open);
            bx3: bufoe PORT MAP(signala(i), outen3, outa(i+4), open);
            bx4: bufoe PORT MAP(signala(i), outen4, outa(i+6), open);
            bx5: bufoe PORT MAP(signalb(i), outen5, outb(i), open);
            bx6: bufoe PORT MAP(signalb(i), outen6, outb(i+2), open);
            bx7: bufoe PORT MAP(signalb(i), outen7, outb(i+4), open);
            bx8: bufoe PORT MAP(signalb(i), outen8, outb(i+6), open);
       END GENERATE;   --the clk input is a rising edge triggered clock for
                       --the register
--the wait until construct must be within a process.
PROCESS
  BEGIN
    WAIT UNTIL clk='1';
    signala <= inputa;
    signalb <= inputb;
  END PROCESS;
END behavior;
```

Appendix E. uselatch2

```
--This file shows the use of the triout component to perform the
--output enable function.

--COMPONENT triout
-- port (
--    x: IN bit; -- input to buffer
--    oe: IN bit; -- output enable
--    y: OUT bit); -- output
--END component

--The oe control is a function of the dedicated inputs and is latch
--controlled.

USE work.rtlpkg.all;          --to instantiate triout component

ENTITY uselatch2 IS
  PORT (clk1, clk2, in_oe1, in_oe2: IN bit;
        inputa, inputb: IN bit_vector(0 to 1);
         outa,outb: INOUT x01z_vector(0 to 7));
END uselatch2;

ARCHITECTURE behavior OF uselatch2 IS
SIGNAL signala, signalb: bit_vector(0 to 1);
SIGNAL sig_en1, sig_en2, sig_en3, sig_en4: bit;
BEGIN
g1:  FOR i IN 0 TO 1 GENERATE
       bx1: triout PORT MAP(signala(i), sig_en1, outa(i));
       bx2: triout PORT MAP(signala(i), sig_en2, outa(i+2));
       bx3: triout PORT MAP(signala(i), sig_en3, outa(i+4));
       bx4: triout PORT MAP(signala(i), sig_en4, outa(i+6));
       bx5: triout PORT MAP(signalb(i), sig_en1, outa(i));
       bx6: triout PORT MAP(signalb(i), sig_en2, outa(i+2));
       bx7: triout PORT MAP(signalb(i), sig_en3, outa(i+4));
       bx8: triout PORT MAP(signalb(i), sig_en4, outa(i+6));
     END GENERATE;

--The clock latches the data when high and is combinatorial when low
oecontrol: PROCESS
           BEGIN
             IF clk1= '0' then
               sig_en1 <= not(in_oe2) and not(in_oe1);
               sig_en2 <= not(in_oe2) and in_oe1;
               sig_en3 <= in_oe2 and not(in_oe1);
               sig_en4 <= in_oe2 and in_oe1;
             ELSE
               sig_en1 <= sig_en1;
               sig_en2 <= sig_en2;
               sig_en3 <= sig_en3;
```

Appendix E. uselatch2 (continued)

```
            sig_en4 <= sig_en4;
          END IF;
        END PROCESS;

latch: PROCESS
        BEGIN
          IF clk2= '0' then
            signala <= inputa;
            signalb <= inputb;
          ELSE
            signala <= signala;
            signalb <= signalb;
          END IF;
        END PROCESS;
END behavior;
```

Appendix F. buriedreg

```
-- The purpose of this example is to show how to use the
-- buried registers to create a 16 bit counter.  The 12
-- most significant bits are assigned to i/o registers
-- and the 4 least significant bits go to the buried registers.

USE work.bv_math.all;                -- necessary for inc_bv();

ENTITY buriedreg IS
       PORT   (clk, reset: IN BIT;
           count: INOUT bit_vector(0 TO 11));
END buriedreg;

ARCHITECTURE behavior OF buriedreg IS
SIGNAL fullcnt : bit_vector(0 to 15);
BEGIN
PROCESS
     BEGIN
       WAIT UNTIL (clk = '1');
            IF reset = '1' THEN        -- synchronous reset
                FOR i IN 0 TO 15 LOOP
                    fullcnt(i) <= '0';
                END LOOP;
            ELSE
                fullcnt <= inc_bv(fullcnt);
            END IF;
     END PROCESS;
count(0 to 11) <= fullcnt(4 to 15);
END behavior;
```

Appendix G. buriedreg2

```
-- The purpose of this example is to show how to use the
-- buried registers to create a 16 bit counter.  The 12
-- most significant bits are assigned to i/o registers
-- and the 4 least significant bits go to the buried registers.
-- This example also demonstrates how to do an asynchronous reset.

USE work.bv_math.all;                -- necessary for inc_bv();

ENTITY buriedreg2 IS
  PORT   (clk, reset: IN BIT;
    count: inout bit_vector(0 TO 11));
END buriedreg2;

ARCHITECTURE behavior OF buriedreg2 IS
SIGNAL fullcnt : bit_vector(0 to 15);
BEGIN
   PROCESS(clk,reset)--sensitivity list
         BEGIN
           IF reset = '1' THEN
             fullcnt <= x"0000";-- asychronous reset, the x stands for hex
           ELSIF (clk'event and clk = '1') then
             fullcnt <= inc_bv(fullcnt);-- synchronous count
           END IF;
         END process;
   count(0 to 11) <= fullcnt(4 to 15);  -- assigns signals to entity outputs
                                        -- and defines buried registers
END behavior;
```

Document #: PLD0010–0394

MAX7000 is a trademark of Altera Corporation.
MACH is a trademark of Advanced Micro Devices, Inc.
Warp, Warp2, Warp3, and FLASH370 are a trademarks of Cypress Semiconductor Corporation.

Implementing a Reframe Controller for the CY7B933 HOTLink™ Receiver in a CY7C371 CPLD

Introduction

This application note describes a reframe controller for the Cypress CY7B933 HOTLink Receiver. The primary function of the controller is to monitor the Receive Violation Symbol output, RVS, from the CY7B933 in order to detect framing errors and, under the correct conditions, assert the Reframe signal, RF, to the CY7B933. The controller function is designed with a state machine, a few counters, and some decode logic. All are implemented in VHDL and fit into a Cypress CY7C371 32-macrocell FLASH CPLD. The exact implementation in this application note makes several assumptions about the next-higher-level controller that may not be universally applicable. However, the source code for the design is provided in Appendix A at the end of this application note so that modification and customization for other interfaces is easily possible.

Why Reframing Is Necessary

The CY7B923 and CY7B933 HOTLink Transmitter and Receiver are a pair of chips for high-speed point-to-point serial data communication. The CY7B923 is the transmitter, and the CY7B933 is the receiver. The CY7B923 takes in an 8-bit byte at a frequency between 16 and 33 MHz, encodes it into 10 bits, does a parallel-to-serial conversion, and then transmits the serial data at ten times the byte-rate clock (about 160 to 330 Megabits per second (Mbps)). At the other end of the link, the CY7B933 receives the serial data, does a serial-to-parallel conversion, unencodes the data back into its original form, and shifts the 8-bit parallel data out at the same byte-rate clock frequency used by the transmitter. (Note: the chips can also transmit and receive 10 bits of unencoded data. For a full description of the encoding and decoding functions, see the CY7B923/933 datasheet.)

The key element in the data-and-clock-recovery circuit on the receiver is the PLL, i.e., phase-locked loop, on the chip. It is triggered by the transitions in the incoming data stream, and it is used to both separate the data stream into individual bits and to generate the byte-rate clock going out of the chip. Once the PLL achieves synchronization with the incoming serial data stream and is receiving bits properly, the receiver must be given a reference point that will set the byte boundaries in the bit stream. This is done by the framing circuitry. Whenever the receiver's RF (reframe) input is asserted, the receiver's framing logic will check the incoming bit stream for the special pattern that defines a byte boundary. When this is found, the receiver logic sets a reference point and simply counts bits from that point on so it can properly execute the serial-to-parallel conversion on subsequent byte boundaries, and properly align the byte-rate clock rising edge.

Thus, framing is always required when the receiver begins receiving data for the first time, either at power-up or after switching from one transmitter source to another. Periodic reframing may also be necessary, however, due to other conditions. If the PLL goes out of lock—that is, if it loses its synchronization with the incoming serial bit stream for any reason, the recovered data will be erroneous and the

framing boundary information will be lost. Once the PLL gets back into synchronization with the incoming bit stream, it will be necessary to force the receiver to reframe in order to re-establish the proper byte boundary point.

Using RVS to Know When to Reframe

The PLL out-of-lock condition can be detected by the behavior of the RVS output of the CY7B933 receiver. The CY7B933 asserts RVS when it detects an error in the bit stream. Infrequent errors, due to random noise in the environment or attenuation by the transmission medium, for example, are expected and do not necessarily mean that the PLL is out of lock or that the data needs to be reframed. Too many errors in too short a time indicates that the PLL has lost lock and reframing is necessary. The benchmark chosen in this controller is 16 errors occurring in a period of 64 bytes. If the controller counts RVS asserted 16 times during a 64-byte period, it will assume the PLL has lost lock and will assert RF to the receiver to force it to reframe.

The 16-out-of-64 benchmark is somewhat arbitrarily chosen, but it is justified by the fact that when the PLL is in lock, you would normally expect to see significantly fewer errors. The fact that 16 out of 64 is the criteria used does not mean that 15 out of 64, or 14 out of 64, etc., are acceptable error rates and that the PLL is not out of lock in these cases as well. But, it is fairly certain that if the PLL does go out of lock, you will get at least 16 errors in 64 byte-times, very quickly. Furthermore, there are counters inside the HOTLink Receiver that detect this same condition (16 errors in a 64-byte period) and when this detection occurs inside the CY7B933, it forces the PLL to re-lock onto the serial input data stream. Even if the PLL is out of lock, if fewer than 16 errors are detected in a 64-byte period, the PLL will not be forced to re-synchronize with the data stream and will stay out-of-lock until that condition is detected. Therefore, for consistency, the same criteria was selected for the reframe controller.

Additional Functionality of the Reframe Controller

The reframe controller itself interfaces to a higher-level controller that controls the entire receiver system. That higher-level controller can force the reframe controller to initiate framing in the CY7B933, regardless of any errors. There are two ways to do this. The first is with the DO_REFRAME signal, which the higher-level controller asserts when it wants the reframe controller to go through the same procedure it goes through to initiate framing when an out-of-lock condition occurs. If the reframe controller sees this signal asserted, it acts just like it had detected an out-of-lock condition. The other way the higher-level controller can force a reframe is by asserting its FORCE_RF output. This simply forces the reframe controller's RF output HIGH and does not cause the internal logic or state machine to change. The reframe controller's RF output will stay asserted as long as its FORCE_RF input remains asserted.

The higher-level controller will normally assert DO_REFRAME on power-up or when the transmitter source is switched on in order to find the initial byte-boundary, as described above. The FORCE_RF signal could be used for any reason depending on specific system requirements. The most likely reason to use it is to force multibyte framing. When the receiver does multibyte framing, instead of looking for a single byte-boundary-indicating character, the receiver looks to detect two of these special characters within any four-byte sequence. This is a more reliable way of finding the byte boundary, simply because it causes the framing circuitry to verify its first find with another one. This may be useful in particularly noisy environments. To cause the receiver to do multibyte framing, you must assert its RF input for 2048 consecutive cycles; this is something the reframe controller would not ordinarily do. The higher-level controller can cause this to happen by asserting FORCE_RF to the reframe controller for 2048 cycles, thus causing its RF output to be asserted for the same length of time.

The reframe controller also implements a basic handshake with the higher-level controller to make sure the two controllers' operations stay consistent

after forced reframes. Whenever the higher-level controller uses the DO_REFRAME signal to force the reframe controller to initiate framing, it will keep that signal asserted until the reframe controller asserts RFDONE_HS. This signal from the reframe controller indicates that the receiver has finished its reframing. The higher-level controller will then assert RFDONE_ACK, which acknowledges receipt of RFDONE_HS, and both the reframe controller and the higher-level controller will return to the state it normally returns to following a reframe.

In addition to the operations described above, the reframe controller also provides a decoding function. When the HOTLink Receiver detects a data error and asserts RVS, it also puts the code for the type of error on its eight data outputs, D7 – D0. The reframe controller decodes these signals and asserts one of two outputs, UNDEF_CHAR or RDISP_ERR, depending on the exact type of error decoded. The two types of errors are an undefined-character error and a running-disparity error. A running-disparity error means that the character received had too many consecutive 1s or 0s to be a valid byte of data (the purpose of the eight-bit-to-ten-bit encoding mentioned earlier is to encode the data in such a way as to minimize the imbalance of 1s and 0s in the bit stream). If the reframe controller detects the code for a running-disparity error, it will assert the RDISP_ERR output. If the received character has the correct running disparity but is not a valid code for any character, then it is an undefined-character error, and the reframe controller will assert the UNDEF_CHAR output instead.

Design and Implementation

The out-of-lock detection, RF control, higher-level controller interface, and error-type decoding are implemented with a simple state machine, a few internal counters, and some decoding logic, and it is all fit into a 32-macrocell CY7C371 FLASH CPLD (for more information on this CPLD, please refer to other application notes in the PLD section of this handbook and to the CY7C371 datasheet). The design was done in VHDL and compiled with Cypress' *Warp*™ PLD/FPGA design tool. The receiver system, the reframe controller's interface, and the details of the design of the internal state machine, counters, and logic are described in detail in the rest of this section.

Receiver System

Figure 1 shows where this reframe controller fits into the overall system. The CY7B933 receiver connects (through a physical connector) to the actual transmission medium, which can be either twisted pair, coaxial cable, or fiberoptic cable. The reframe controller interfaces to the receiver, and it also interfaces to the higher-level system controller.

Controller Interface

The complete set of reframe controller inputs and outputs is shown in *Figure 2*, and their source or destination, polarity, and functionality are described below.

Inputs

RF_ENABLE. Overall enable. It comes from a higher-level controller. When asserted (HIGH), reframe controller is enabled. When deasserted, reframe controller is disabled and does not operate.

CLK. Clock signal to the reframe controller that comes from the recovered byte-rate-clock output, RCLK, of the CY7B933, and is also used in the rest of the system as the system clock.

RESET. Resets the state machine and the internal counters and status registers (HIGH = asserted).

RVS. Received Violation Symbol. It comes from RVS output of the CY7B933 (HIGH = asserted).

FORCE_RF. When asserted, this forces the RF output to also be asserted regardless of other conditions. It comes from a higher-level controller (HIGH = asserted).

DO_REFRAME. When asserted, it causes internal state machine to initiate framing in the receiver just as if it had detected an out-of-lock condition. It comes from higher-level controller (HIGH = asserted).

RFDONE_ACK. Handshake signal from the higher-level controller acknowledging that it received confirmation that the reframe controller completed

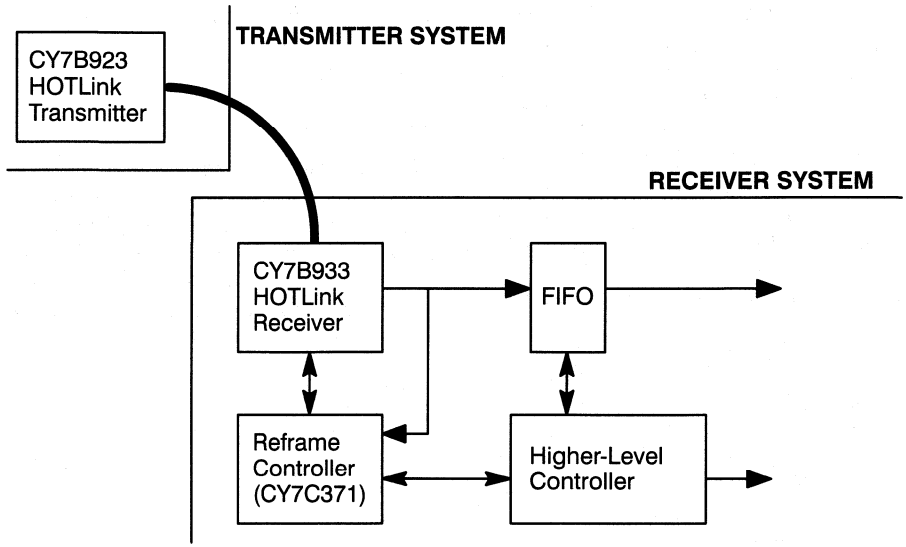


Figure 1. Block Diagram

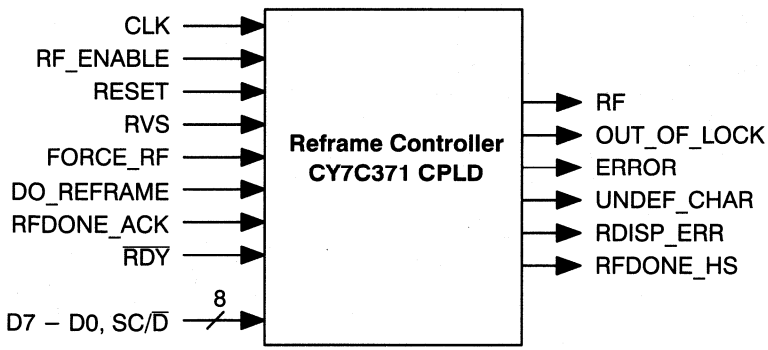


Figure 2. Controller Inputs and Outputs

the framing procedure. The handshake is only done when the framing was triggered by the DO_REFRAME signal, not by an out-of-lock condition (HIGH = asserted).

RDY. Ready signal. It comes from the CY7B933 RDY output and indicates to the reframe controller that the receiver has completed the reframe operation (LOW = asserted).

D[7:0], SC/D. Eight-bit data byte and control/data indicator bit from the CY7B933 receiver. The information on these lines can be decoded during a receive violation to determine the error type.

Outputs

RF. Reframe output. It goes to the RF input of the CY7B933 receiver and causes the HOTLink Re-

ceiver to begin a framing operation on the incoming data stream (HIGH = asserted).

RFDONE_HS. This is the handshake signal to the higher-level controller telling it that the reframe it requested with the DO_REFRAME signal has been completed (HIGH = asserted).

OUT_OF_LOCK. This signal indicates that the HOTLink Receiver's PLL has gone out of lock with the incoming serial bit stream. This is inferred by counting sixteen or more RVS assertions in a single 64-byte period. Once asserted, it remains asserted until the PLL regains lock and reframing has been accomplished (HIGH = asserted).

ERROR. When asserted (HIGH) it indicates to the higher-level controller that an error of some type (as indicated by the RVS signal from the receiver) has occurred.

UNDEF_CHAR. This is an undefined-character-error signal, one of two types of errors that can be decoded from the D7 – D0, SC/$\overline{D}$ inputs during receive violations. This signal is only valid when the ERROR output is also asserted, and it can only be asserted when RDISP_ERR is deasserted (HIGH = asserted).

RDISP_ERR. Running-disparity-error signal. This is the other of the two types of errors that can be decoded from the D7 – D0, SC/$\overline{D}$ inputs during data-receive violations. This signal is only valid when the ERROR output is also asserted, and it can only be asserted when UNDEF_CHAR is deasserted (HIGH = asserted.)

Counters

The primary function of the controller, which is to detect the out-of-lock condition by monitoring RVS and initiate a reframe when necessary, is implemented through the use of two counters. The VHDL for this function is shown in *Figure 3*. The first counter, rcvdbyts_count, is a seven-bit counter that counts the number of bytes received (0 to 64) and the second counter, error_count, is a five-bit counter that counts the number of times that RVS is asserted. If error_count reaches 16 before rcvdbyts_count reaches 64, then the out-of-lock condition will be declared. If rcvdbyts_count reaches 64 before error_count reaches 16, then fewer than 16 errors occurred in the given 64-byte window and out-of-lock is not declared. If rcvdbyts_count reaches 64 before error_count reaches 16, both rcvdbyts_count and error_count are set back to zero and a new 64-byte window begins. If the out-of-lock condition is declared (error_count = 16 and rcvdbyts_count $\leq$ 64), then the out-of-lock flip-flop is set to HIGH and a reframe operation is initiated. The out-of-lock flip-flop stays HIGH until the receiver successfully reframes. At that point, the out-of-lock flip-flop is set back to LOW and the search for the out-of-lock condition is started again.

State Machine

The state machine is described by the diagram in *Figure 4*, and the VHDL code that implements it is shown in *Figure 5*.

IDLE state

The normal, quiescent state of the state machine, and the state it enters upon reset, is IDLE. In this state, the RF output is deasserted and the state machine waits for either a DO_REFRAME input from the outside or for the counters to set the out-of-lock flip-flop. If neither of these conditions occur, the state machine simply stays in the IDLE state. Once either one of these conditions occurs, the state machine must initiate a reframe, so it will go to the START_REFRAME state.

START_REFRAME state

In the START_REFRAME state, RF is asserted, and the state machine unconditionally transitions to the COUNT_2_CLOCKS state.

COUNT_2_CLOCKS state

The COUNT_2_CLOCKS state enables a two-bit counter to start counting incoming clock cycles. After two clock cycles have been counted, the state machine transitions to the LOOK_FOR_xRDY state. Two clock cycles must be counted before looking for the $\overline{RDY}$ signal from the outside because a total of three clocks must pass after RF is asserted until the value of $\overline{RDY}$ can be guaranteed valid (see the "HOTLink CY7B933 $\overline{RDY}$ Pin Description" ap-

```
-- relevant VHDL code for counter functions

use work.bv_math.all;
use work.int_math.all;

signal count2: bit_vector(0 to 1);          -- 2-bit counter
signal error_count: bit_vector(0 to 4);     -- 5-bit counter
signal rcvdbyts_count: bit_vector(0 to 6);  -- 7-bit counter

counters: process (CLK, RVS, reset, rcvdbyts_count, error_count,
       out_of_lock) begin

  if (clk'event and clk = '1') then

    if (reset = '1') then
      fb_out_of_lock    <= '0';
      rcvdbyts_count <= "0000000";
      error_count    <= "00000";
    elsif (error_count = "10000") then
      fb_out_of_lock    <= '1';
      rcvdbyts_count <= "0000000";
      error_count    <= "00000";
    elsif (rcvdbyts_count = "1000000") then
      rcvdbyts_count <= "0000000";
      error_count    <= "00000";
    else
      rcvdbyts_count  <= rcvdbyts_count + 1;
      if (RVS = '1') then
        error_count <= error_count + 1;
      end if;
    end if;

    if (current_state = LOOK_FOR_xRDY) and (xRDY = '0') then
      fb_out_of_lock <= '0';
    end if;

    if (current_state = COUNT_2_CLOCKS) then
      count2 <= count2 + 1;
    else
      count2 <= "00";
    end if;

  end if;

end process; --counters
```

Figure 3. VHDL for Counter Functions

plication note for more details on this). One clock cycle passed during the START_REFRAME state, so the COUNT_2_CLOCKS state is used to count two more clock cycles to get to the requirement of three. RF is asserted throughout this state.

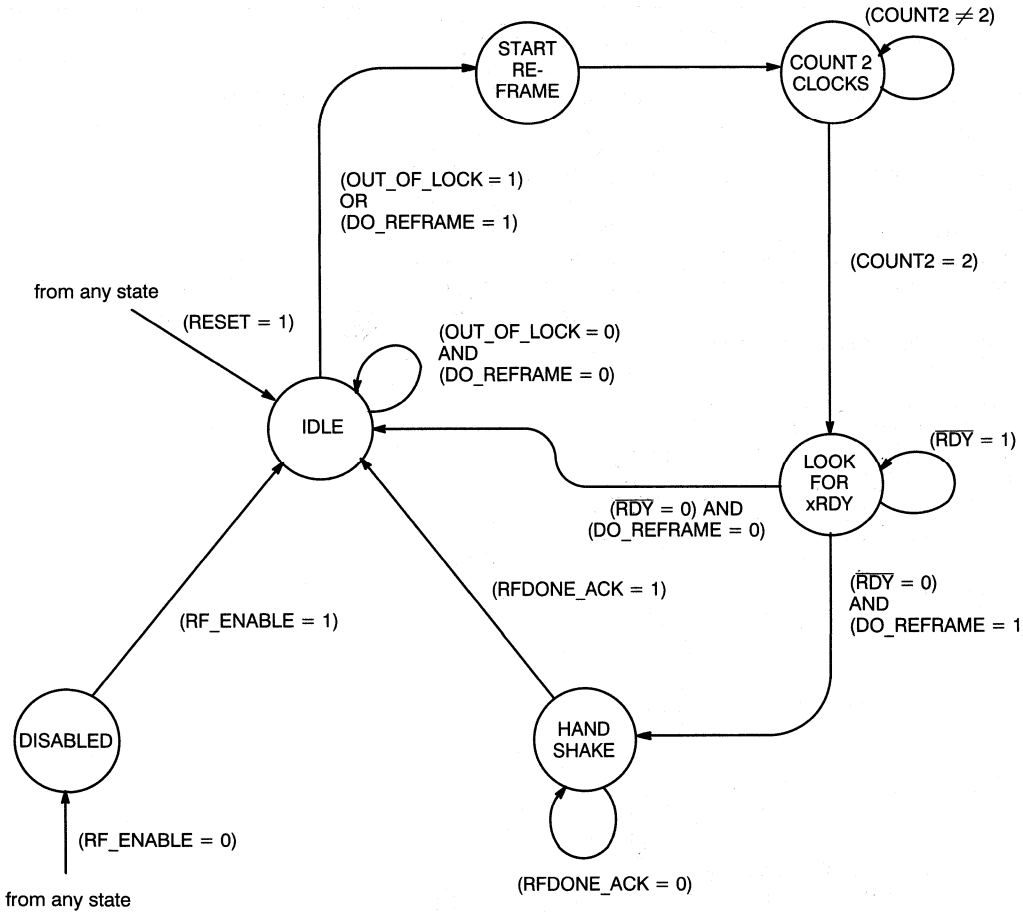


Figure 4. State Diagram

LOOK_FOR_xRDY state

On the fourth clock cycle from the start of RF, the value of $\overline{RDY}$ is guaranteed to be valid and the state machine, in the LOOK_FOR_xRDY state, continues to assert RF and waits until the HOTLink Receiver asserts $\overline{RDY}$. Once the receiver asserts RDY, it has successfully reframed and is ready to resume normal receiver operation. Thus, once an asserted $\overline{RDY}$ is detected in the LOOK_FOR_xRDY state, the state machine exits that state and goes back to the IDLE state. If the reframe was started by an out-of-lock detection, the transition back to the IDLE state is immediate; if the reframe was started by the DO_REFRAME input, then the state machine goes to the HANDSHAKE state first.

HANDSHAKE state

The HANDSHAKE state is used to make sure the reframe controller and the higher-level controller are consistent with each other. The only way this state will ever be entered is if the higher-level con-

```
-- Relevant VHDL code for state machine

subtype StateType is bit_vector(0 to 2);       -- State Type
constant       DISABLED: StateType := b"111"; -- State Defns.
constant           IDLE: StateType := b"000";
constant  START_REFRAME: StateType := b"001";
constant COUNT_2_CLOCKS: StateType := b"010";
constant  LOOK_FOR_xRDY: StateType := b"011";
constant      HANDSHAKE: StateType := b"100";
signal current_state, next_state : StateType; --State declaration

-- State Machine Description

if (RESET = '1') then
  next_state <= IDLE;
elsif (RF_ENABLE = '0') then
  next_state <= DISABLED;
else
  case current_state is

  when IDLE =>
    if (fb_OUT_OF_LOCK = '1') or (DO_REFRAME = '1') then
      next_state <= START_REFRAME;
    else
      next_state <= current_state;
    end if;

  when START_REFRAME =>
    next_state <= Count_2_Clocks;

  when COUNT_2_CLOCKS =>
    if (count2 = "10") then
      next_state <= LOOK_FOR_xRDY;
    else
      next_state <= current_state;
    end if;

  when LOOK_FOR_xRDY =>
    if (xRDY = '0') and (DO_REFRAME = '1') then
      next_state <= HANDSHAKE;
    elsif (xRDY = '0') and (DO_REFRAME = '0') then
      next_state <= IDLE;
    else
      next_state <= current_state;
    end if;
```

Figure 5. VHDL Code for State Machine

```
  when HANDSHAKE =>
    if (RFDONE_ACK = '1') then
      next_state <= IDLE;
    else
      next_state <= current_state;
    end if;

  when DISABLED =>
    if (RF_ENABLE = '0') then
      next_state <= current_state;
    else
      next_state <= IDLE;
    end if;

  end case;
end if;

if (clk'event and clk = '1') then
    current_state <= next_state;
end if;
```

Figure 5. VHDL Code for State Machine (continued)

troller initiated a reframe by asserting DO_REFRAME to the reframe controller. Once that reframe has been completed by the receiver, the reframe controller communicates this to the higher-level controller by asserting RFDONE_HS. Once the higher-level controller acknowledges this assertion and is ready to proceed with normal receiving operation, it will assert RFDONE_ACK as confirmation to the reframe controller. It will simultaneously deassert DO_REFRAME so that once the state machine goes back to the IDLE state, that input is deasserted and does not erroneously cause another immediate pass into the reframe procedure. Once the state machine detects the RFDONE_ACK assertion, it exits the HANDSHAKE state and returns to the IDLE state. The RF operation is deasserted throughout the HANDSHAKE state.

DISABLED state

There is one more state, the DISABLED state, which is treated separately. As long as RF_ENABLE, the overall controller enable, is asserted, the state machine will never enter this state. If RF_ENABLE gets deasserted, the state machine will transition to the DISABLED state no matter what state it was in, and it will stay there until RF_ENABLE is once again asserted. Once RF_ENABLE is reasserted, the state machine goes to the IDLE state and resumes normal operation.

It was mentioned previously that the out-of-lock flip-flop is set when the out-of-lock condition is detected, and it stays set until the reframe has been completed. The exact time when the OUT_OF_LOCK flip-flop gets cleared is at the rising clock edge when the state machine exits the LOOK_FOR_xRDY state. This is because that is the exact point where the receiver has signalled to the controller, with $\overline{\text{RDY}}$, that it has successfully competed the reframe.

Decode Logic

The error-decode logic is very straightforward, and the VHDL code for it is shown in *Figure 6*. The ERROR output is a registered version of the RVS input. The RDISP_ERR and UNDEF_CHAR outputs are decoded from the D7 – D0, SC/$\overline{\text{D}}$ inputs. These outputs are also registered.

When the receiver asserts RVS, it will also put a code for the error type on its eight data outputs. If this code is E4, E2, or E1 (hex), it indicates the error is a running-disparity error, (explained earlier), and the RDISP_ERR output is asserted. If it is any other hex code, the receiver has detected some kind of illegal or undefined character, and the UNDEF_CHAR output will be asserted instead. These outputs are mutually exclusive, i.e., if one is asserted, the other must be deasserted. However, it is only meaningful to decode the data outputs when an error condition is detected, so the ERROR signal must be examined by the higher-level controller as well. If ERROR is not asserted, the output from RDISP_ERR and UNDEF_CHAR is no longer valid.

VHDL, CY7C371 Utilization, and CY7C371 Speed Considerations

The complete VHDL description for this design is given in Appendix A. The full source code consists of the fragments shown throughout this application note along with the other code necessary to mesh it together, (process declarations, signal declarations, and package-entity declarations). As the fragments and complete source file show, VHDL is a very simple, efficient way for describing PLD designs. For example, the counter functions are simply bit vectors that are used in the manner: COUNT <= COUNT + 1. Upper limits for the counters, clearing functions, resets, and presets are all implemented with a few simple IF-THEN-ELSE statements. The entire state machine is implemented with a CASE statement and IF-THEN-ELSE statements that have a straightforward, natural, one-to-one correspondence with the bubble diagram shown in *Figure 4*. The entire set of decode logic is implemented in a single IF-THEN-ELSE clause. Furthermore, the VHDL code provided is easy to understand and can be very easily modified. For example, it can be modified to interface to different higher-level-controller interfaces than the one assumed in this application note, or it could be incorporated into the higher-level controller design, with that design consisting of other VHDL code and implemented in a larger FLASH370™ CPLD, a pASIC™ FPGA, or even a gate array.

```
-- relevant VHDL code for Decode Logic

if (clk'event and clk = '1') then

  if (RVS = '1') then
    ERROR <= '1';
    if (D = x"E4" or D = x"E2" or D = x"E1") then
      UNDEF_CHAR <= '0';
      RDISP_ERR  <= '1';
    else
      UNDEF_CHAR <= '1';
      RDISP_ERR  <= '0';
    end if;
  else
    ERROR <= '0';
    UNDEF_CHAR <= '0';
    RDISP_ERR  <= '0';
  end if;

end if;
```

Figure 6. VHDL Code for Decode Logic

This design used all 32 of the CY7C371's macrocells and 37 of its 38 I/O and input pins. It could have used fewer pins if necessary, by making the various counters be internal counters only. The outputs of the counters were brought out to output pins in this example, however, for easier simulation and debugging. The speed of the CY7C371 ranges from 66 MHz (with a 15-ns combinatorial propagation delay and a 12-ns clock-to-output time) to 143 MHz (with a 8.5-ns combinatorial propagation delay and a blazing 6-ns clock-to-output time). For this application, the maximum byte-rate clock of the CY7B933 is 33-MHz, and this and the corresponding set-up and hold times on the CY7B933 make the CY7C371-66 quite sufficient. The higher-level controller may have tighter timing requirements, but there is plenty of speed to be gained by going to the faster speed bins of the CY7C371. The design can, thus, easily meet much faster system timing requirements.

Conclusion

The serial data received by the CY7B933 needs to be framed, i.e., aligned to the proper byte boundaries. This must always be done when the serial communication first begins, and it must always be redone if the PLL loses lock on the incoming serial bit stream. This application note described a controller that will manage this operation and provided some guidelines for determining when the periodic reframing is necessary. It assumed a particular interface to a higher-level controller, but the design was done in VHDL, which is provided in the appendix, to make it very easily modifiable and adaptable to any other specific interface. The controller itself is implemented in a CY7C371 32-macrocell CPLD, which had sufficient resources and routability to implement this fairly substantial function. It was able to do this exceeding system speed requirements even in its slowest speed bin.

Appendix A. VHDL Description

```
-- Application Note
-- Using a CY7C371 as a HOTLink Reframe Controller
-- Cypress Semiconductor

use work.bv_math.all;
use work.int_math.all;

entity CONTROLLER is port (
    CLK, RVS, RESET, xRDY, DO_REFRAME, FORCE_RFOUT, RFDONE_ACK,
      RF_ENABLE : in bit;
    D           : in bit_vector(0 to 7);
    curr_st     : out bit_vector (0 to 2);
    rb_cntr     : out bit_vector (0 to 6);
    err_cntr    : out bit_vector (0 to 4);
    RF, RFDONE_HS, OUT_OF_LOCK, UNDEF_CHAR, RDISP_ERR, ERROR : out bit
);
end CONTROLLER;

architecture CNTRL933 of CONTROLLER is

    subtype StateType is bit_vector(0 to 2);        -- State Type
    constant       DISABLED: StateType := b"111";   -- State Definitions
    constant           IDLE: StateType := b"000";
    constant  START_REFRAME: StateType := b"001";
    constant COUNT_2_CLOCKS: StateType := b"010";
    constant  LOOK_FOR_xRDY: StateType := b"011";
    constant      HANDSHAKE: StateType := b"100";

    signal current_state, next_state : StateType;
    signal fb_OUT_OF_LOCK : bit;

    signal count2: bit_vector(0 to 1);          -- 2-bit counter
    signal error_count: bit_vector(0 to 4);     -- 5-bit counter
    signal rcvdbyts_count: bit_vector(0 to 6);  -- 7-bit counter

begin

counters: process (CLK, RVS, reset, rcvdbyts_count, error_count,
       out_of_lock) begin

  if (clk'event and clk = '1') then

    if (reset = '1') then
      fb_out_of_lock    <= '0';
      rcvdbyts_count <= "0000000";
      error_count    <= "00000";
```

Appendix A. VHDL Description (continued)

```
    elsif (error_count = "10000") then
      fb_out_of_lock    <= '1';
      rcvdbyts_count <= "0000000";
      error_count    <= "00000";
    elsif (rcvdbyts_count = "1000000") then
      rcvdbyts_count <= "0000000";
      error_count    <= "00000";
    else
      rcvdbyts_count  <= rcvdbyts_count + 1;
      if (RVS = '1') then
        error_count <= error_count + 1;
      end if;
    end if;

    if (current_state = LOOK_FOR_xRDY) and (xRDY = '0') then
      fb_out_of_lock <= '0';
    end if;

    if (current_state = COUNT_2_CLOCKS) then
      count2 <= count2 + 1;
    else
      count2 <= "00";
    end if;

  end if;

end process; --counters

next_st_comb:  process (CLK, fb_OUT_OF_LOCK, DO_REFRAME, FORCE_RFOUT, xRDY,
                        RFDONE_ACK, RESET, RF_ENABLE, current_state) begin

  if (RESET = '1') then
    next_state <= IDLE;
  elsif (RF_ENABLE = '0') then
    next_state <= DISABLED;
  else
    case current_state is

    when IDLE =>

      if (fb_OUT_OF_LOCK = '1') or (DO_REFRAME = '1') then
        next_state <= START_REFRAME;
      else
        next_state <= current_state;
      end if;

    when START_REFRAME =>
```

Appendix A. VHDL Description (continued)

```
      next_state <= Count_2_Clocks;

    when COUNT_2_CLOCKS =>

      if (count2 = "11") then
        next_state <= LOOK_FOR_xRDY;
      else
        next_state <= current_state;
      end if;

    when LOOK_FOR_xRDY =>

      if (xRDY = '0') and (DO_REFRAME = '1') then
        next_state <= HANDSHAKE;
      elsif (xRDY = '0') and (DO_REFRAME = '0') then
        next_state <= IDLE;
      else
        next_state <= current_state;
      end if;


    when HANDSHAKE =>

      if (RFDONE_ACK = '1') then
        next_state <= IDLE;
      else
        next_state <= current_state;
      end if;

    when DISABLED =>

      if (RF_ENABLE = '0') then
        next_state <= current_state;
      else
        next_state <= IDLE;
      end if;

    end case;
  end if;
end process; --next_st_comb

outp_comb:  process (current_state, FORCE_RFOUT) begin

  if (FORCE_RFOUT = '1') then
    RF <= '1';
  else
    case current_state is
```

Appendix A. VHDL Description (continued)

```
    when IDLE =>
      RF         <=  '0';
      RFDONE_HS  <=  '0';

    when START_REFRAME =>
      RF         <=  '1';
      RFDONE_HS  <=  '0';

    when COUNT_2_CLOCKS =>
      RF         <=  '1';
      RFDONE_HS  <=  '0';

    when LOOK_FOR_xRDY =>
      RF         <=  '1';
      RFDONE_HS  <=  '0';

    when HANDSHAKE =>
      RF         <=  '0';
      RFDONE_HS  <=  '1';

    end case;
  end if;
end process; --outp_comb

seq_assgnmnt:  process (clk) begin

  if (clk'event and clk = '1') then

    current_state <= next_state;

    if (RVS = '1') then
      ERROR <= '1';
      if (D = x"E4" or D = x"E2" or D = x"E1") then
        UNDEF_CHAR <= '0';
        RDISP_ERR  <= '1';
      else
        UNDEF_CHAR <= '1';
        RDISP_ERR  <= '0';
      end if;
    else
      ERROR <= '0';
      UNDEF_CHAR <= '0';
      RDISP_ERR  <= '0';
    end if;

  end if;

end process; --seq_assgnmnt
```

Appendix A. VHDL Description (continued)

```
-- concurrent assignment statements
-- outputs and local feedback signals made the same

curr_st     <=  current_state;
rb_cntr     <=  rcvdbyts_count;
err_cntr    <=  error_count;
OUT_OF_LOCK <=  fb_out_of_lock;

end CNTRL933;  -- end architecture
```

Document #: PLD0012−0394

HOTLink, *Warp*, and FLASH370 are trademarks of Cypress Semiconductor Corporation.
pASIC is a trademark of QuickLogic Corporation.

Architectures and Technologies for FPGAs

Introduction

The FPGA (Field Programmable Gate Array) is the newest concept in programmable logic. Previously the most complex programmable logic device was the Complex Programmable Logic Device, the CPLD. The CPLD concept is a simple extension of the basic PLD. Taking a small PLD device design and repeating it multiple times on the same die provides large resources in a single device. It is then necessary is to provide interconnect resources to allow each repeated cell to share resources and to communicate with one another and the I/O cells. The individual repeated cells are called macrocells which are, of course, relatively large and functionally complex.

Before the FPGA, the next level up from the PLD in solution alternatives was the sea-of-gates gate array. This is a fixed die, consisting of transistors, that is customized by the user by specifying the interconnect of the transistors. The user, in actuality, specifies the interconnection between a set of functional primitives such as NAND gates and flip-flops, which the gate array vendor has predefined and placed in a library. The gate array is not user programmable and must be customized by the vendor in the manufacturing process. Delivery of first articles is many weeks, and non-recurring costs are usually above ten thousand dollars.

Between the CPLD and the gate array is the FPGA which borrows from the solutions above and below. The FPGA logic cells are small and have less functionality than those of the CPLD. Thus they are a move toward the sea-of-gates concept in the Gate Array ASIC. Since the FPGA logic cells are smaller than those of the CPLD, there are many more of them in the same die. The FPGA logic cells are arranged in a rectangular array as in the gate array sea-of-gates concept. Between each logic cell is a routing channel so that multiple interconnect wires can run vertically and horizontally across the chip. Programmable connection points are provided where the logic cell I/O enters the routing channel and at the cross points where vertical routing channels meet horizontal routing channels. By appropriate programming of the cell I/O connections and the cross channel connections, signals can be routed throughout the chip.

Although the FPGA concept is relatively simple, realization of the FPGA is complex. There are many interrelated technology and architecture issues which must be addressed to produce a successful device. Success in this context means a device which can make maximum use of the available resources to accommodate large designs, achieve the highest possible performance that the semiconductor process technology has to offer, and give the designer flexibility (in, for example, pin assignments). This application note is intended to explain key factors in technology and architecture issues and how they relate. From this understanding, benefits to the designer will emerge. Different FPGA approaches have very different characteristics that can make the difference between a design achieving the required performance or being able to fit into a specific device. The material in this note is intended to help the design engineer make choices that will help achieve design goals.

Detailed Architecture

The global form of an FPGA is shown in *Figure 1*. The layout is a matrix of logic cells with a grid of routing channels running between the cells. I/O cells surround the array and allow access to the ex-

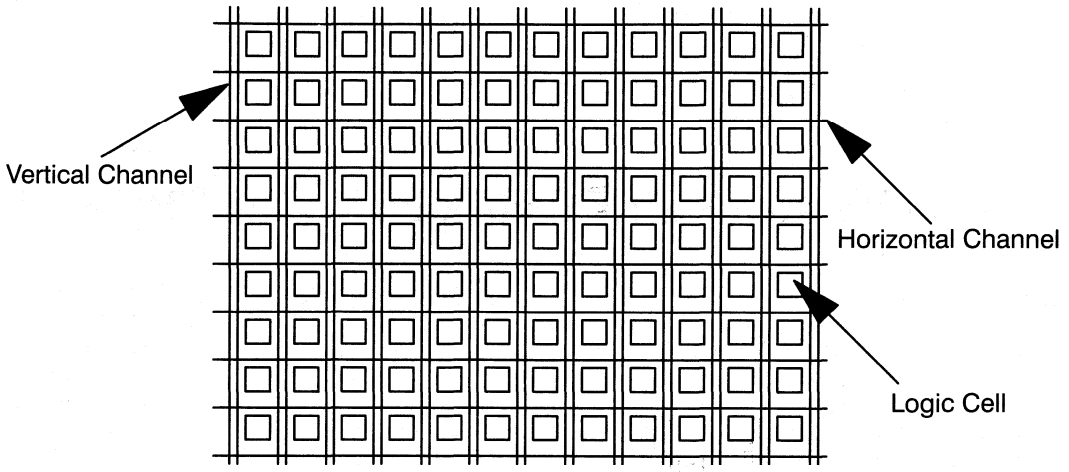


Figure 1. Global FPGA Architecture

ternal pins of the device. The programmable connections are located where the vertical and horizontal routing channels cross; where the I/O of the logic cells meet the routing channels; and where the I/O cells around the periphery meet the routing channels. In contrast to the CPLD, the FPGA usually has no programmability within the logic cells themselves (Some FPGA architectures do include programmable elements within the logic cell. Beyond this fundamental architecture, FPGAs can differ widely in the details. Key considerations are: (1) the number of wires in the routing channels, (2) the flexibility in the interconnect programmability where channels meet channels and where logic cells meet channels, and (3) the functionality contained within the logic cell. The details of the architecture choices in these key considerations are not obvious and are closely tied to the semiconductor process technology that is used to realize the device. The remainder of this section focuses on some of the more important aspects of these details: to show how they influence architecture choices and what impact the choices have on the performance, cost, and utility of the final product.

FPGA Logic Cells

There are three approaches to the form of a logic cell in FPGA implementations.

One approach is to make the logic cell as complex as in the CPLD. Such an implementation is termed a Coarse-Grain Logic Cell. An example is shown in *Figure 2*. This cell contains multiple flip-flops, several multiplexers, a combinatorial function block, and a variety of different inputs. Each of the flip-flops may be bypassed in order to implement combinatorial functions. From a first level analysis, it is clear that this logic cell can implement complex logic as well as register intensive functions. Three important characteristics are significant to note. First, the cell is complex and, as will be explained later, unless there is programmability in the cell, this can lead to inefficient use of the available logic. Second, there are a small number of cell outputs (two in this case). Third, many of the inputs are dedicated to flip-flop control and are not usable for other purposes if the flip-flop is not required. Timing analysis of this cell can be complicated. Since the cell possesses a large amount of functionality, it reduces the burden on the cell-to-cell interconnect.

The second approach in the cell architecture is to make the logic cell very simple. This approach is termed the Fine-Grain Logic Cell. An example is shown in *Figure 3*. This cell contains no flip-flops and very minimal logic. Only one output is available. The cell can realize simple AND-OR logic implementations and, because of its simplicity, it can

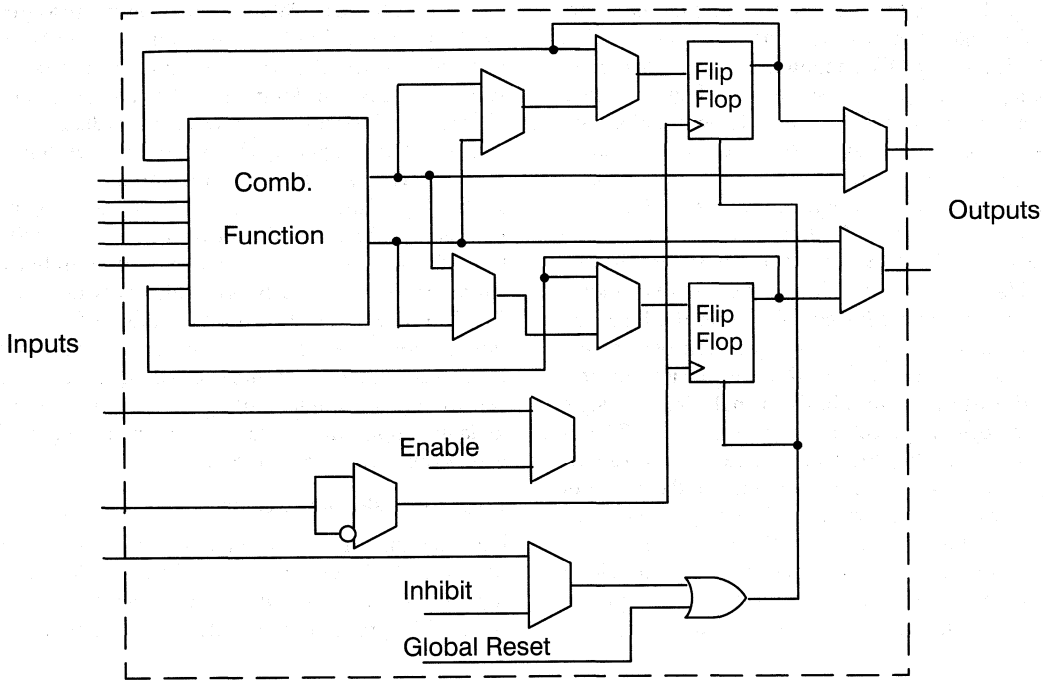


Figure 2. Coarse-Grain Logic Cell

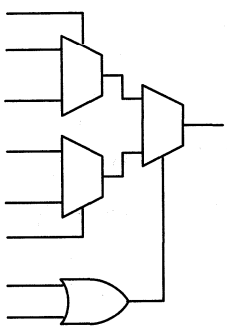


Figure 3. Fine-Grain Logic Cell

have very low propagation delays through the cell and there tends to be a high cell utilization. This cell will rely heavily on the cell-to-cell routing resources. The greater use of routing may add to the overall delay negating the low delay of the cell.

The third approach is a hybrid between the coarse-grain and fine-grain extremes but with some variations that enhance the other trade offs in the global architecture design. To understand the hybrid role, the relationship of the fine-grain and coarse-grain logic cell approaches to the global architecture design must be understood.

The global architecture relation to the logic cell type can be illustrated by starting from the basic FPGA concept and identifying and working through the implementation trade offs. Consider a sea of logic cells surrounded by the routing channels. Assume that the routing channels are very large and the interconnect completely flexible. Then, as in the custom sea-of-gates ASIC, the logic cells can be small, consisting of elementary logic primitives. The first trade off is that the FPGA is not a gate array where gates are sacrificed for routing. In the FPGA, the routing resources are limited and unused gates offer no increase in interconnect capability. This suggests

that the logic cell possess more than minimal functionality and should include at least one flip-flop and wide AND-OR combinations to realize complex logic. It also suggests that the logic cell have multiple outputs so that if some logic functions are very simple, more than one of these functions could be implemented in the same cell. This allows maximum utilization of the logic cell resources. Continuing this pattern would further increase functionality in the logic cell. Bear in mind, however, that the logic cell lacks programmability in the examples shown here and that directing signals within the logic cell is done with multiplexing and judiciously selecting input combinations. Further expansion of this would waste resources in the signal directing logic and increase logic cell propagation delay. This is detrimental to the objective of implementing the desired logic function. Added circuitry and controls in the logic cell to maximize its flexibility do not contribute to the realization of the desired function. That flexibility is the task of the interconnect. There is, therefore, an optimum logic cell complexity which lies somewhere between the coarse- and fine-grained extremes. With finite routing resources available, but without fuse related constraints, the optimal cell would look something like the cell in *Figure 4*. This cell design is simple and symmetric yielding small propagation delays regardless of the signal path through the cell. Since the interconnect is not a factor, a large number of inputs is provided so that logic functions of many variables can be implemented. The cell also provides for the realization of any logic function up to a given number of variables. Multiple outputs are also provided. The flip-flop may be used or the "D" input of the flip-flop is available as an output for combinatorial only functions. The large number of outputs allows the cell to be split, implementing two or more simple logic functions in the same cell or sharing logic across function in the same cell.

Programmable Connections

Now examine the interconnect surrounding the sea of logic cells. There are two key issues in the implementation of the programmable connections. First is size. The programmable interconnect circuit may

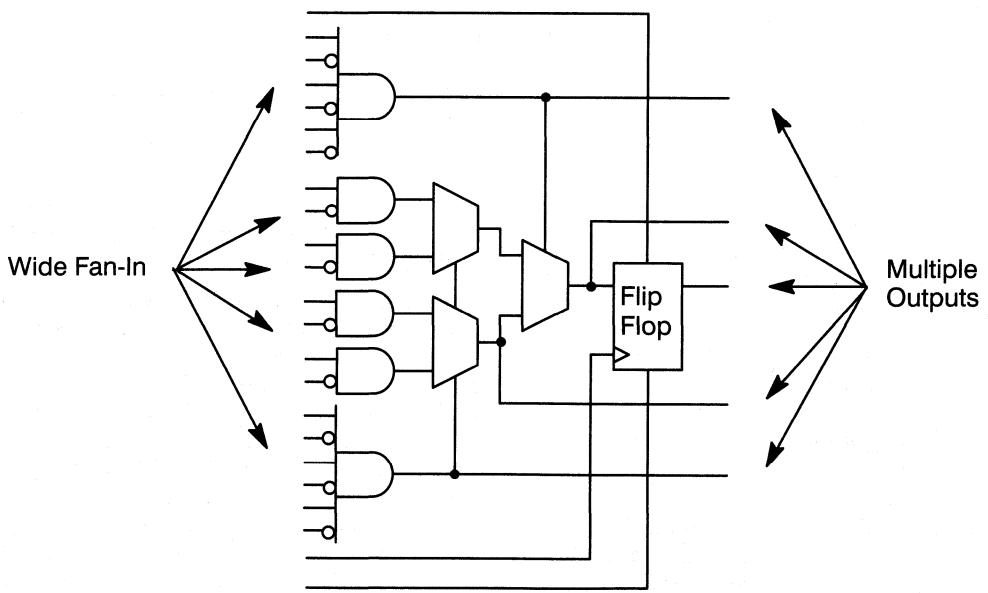


Figure 4. An Optimal Logic Cell

use an area which limits the number of interconnects which can be put into a given area. Second is the electrical characteristics of the interconnect. The interconnect may not look like a wire to the signal that it is carrying. Resistance and capacitance of the interconnect can affect the propagation delay of the signal.

Infinite routability is not realistic. The routing channels can contain only a finite number of wires and the interconnect possibilities where wires cross or meet is not endless. There are three classes of ways to connect two wires: RAM-based connections, large fuse technology, and via fuse technology. Consider the rectangular area where horizontal and vertical wiring channels cross. At this intersection there is potentially a connection possibility at each wire crossing (intersection). An ideal case of the user programmable interconnect possibilities are shown in *Figure 5*. Each circle at the intersection point represents a potential user-programmable connection. This scheme offers a great deal of interconnect capability. There is a lot of redundancy in the interconnect possibilities. Once a wire is connected to a signal, it is dedicated to that signal throughout the extent of that wire. Wires may be segmented so that they can support local interconnect. The implementation of the connection mechanism (fuse, RAM cell, etc.) may be larger than the dimension of the wire size and inter-wire spacing as shown in *Figure 6*. Because of the size of the connection cell for connection "O," programmable connection at points "X" are not possible. Further programmable connection cells cannot fit into the area unless the interconnect wires are spread apart. This latter approach is not an effective alternative since it upsets the regularity and fit of the wiring channels and the logic cells. The only alternative is to limit the number of connection possibilities.

An example of this situation is RAM-based programmable interconnect. RAM-based connectivity uses a memory cell to control the connection of one wire to another. The memory cell is far larger than the wire intersect area. What is done is to limit the connection possibilities to a small fraction of the possible signal paths and limit the number of wires in the routing channel. The result is that the routability of the FPGA becomes what is known as "interconnect constrained." That is, the number of wire connection possibilities is so small that the interconnect limits the realization of functions with a given logic cell architecture. To compensate for this, coarse-grained logic cells are usually chosen and programming internal to the logic cell may be added. The ability to perform more function in a logic cell tends to make up for the inability to implement the equivalent function in a set of interconnected logic cells. It is difficult to quantify the results of such choices. Can a large complex logic cell adequately compensate for interconnect constrained situations? There is no clear answer and the results are dependent on what is to be implemented in the FPGA. However, it can be determined by realizing various types of functions in various architecture forms that the interconnect-constrained architec-

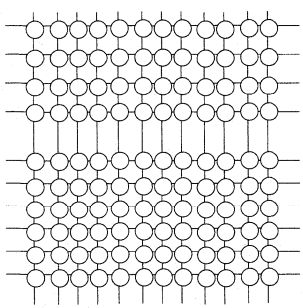


Figure 5. Connection Points at Routing Channel Intersection

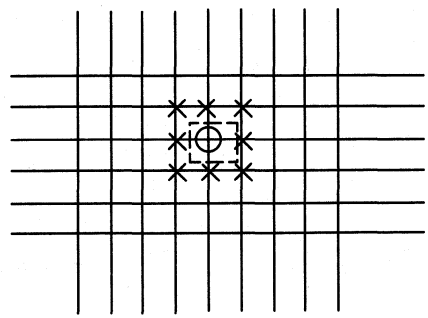


Figure 6. Connection Limitations due to Programmable Connect Cell Size

tures tend to have routability limitations which are exacerbated when any constraints are placed on the device pin/signal association.

Architectures implemented with large-fuse technology tend to have the same characteristics as RAM-based connectivity except they are not as severe. Both approaches limit the number of outputs in the logic cell to only one or two. This is because increased outputs add to the number of potential connections, which places a further burden upon an already stressed interconnect mechanism. Because the number of outputs is small, simple functions will tend to waste logic resources in the interconnect-constrained, complex logic cell architectures. Therefore, these architectures may tend to be more efficient in implementing complex state machines than the fine-grained logic cell architectures with the same interconnect capability.

Of the programmable interconnect technologies available for FPGAs, (RAM-based, large-fuse, and via-fuse) the optimum interconnect is achieved by via-fuse technology. This technology realizes an antifuse in the same physical area as that used by a normal semiconductor process via which connects two layers of metal. The via fuse will be described in detail in a later section. This technology approaches the interconnect characteristics found in sea-of-gates gate arrays and almost completely eliminates programmable interconnect as a factor in FPGA wiring channel and logic cell architectures.

The electrical characteristics of the interconnect technology play a major role in the performance of the FPGA. A first level summary of the technology impact on the interconnect (not including wire delay) is given in *Table 5*. When the connection is made, it exhibits some ON resistance in series with the logic signal path. When a connection is OFF it presents a shunt capacitance from the logic signal wire to ground. A single ON fuse followed by a single OFF fuse represents an RC combination in the signal path. The product of the ON resistance and the OFF capacitance of this combination is given in the Time Constant column of the table.

Timing Model

The timing model is a representation of signal delays in an actual FPGA that allow the designer to determine the performance of a design when it is realized in a particular device. The nature of the timing model is of concern to the designer since it affects the level of difficulty in determining performance. All FPGAs, by virtue of their architectures, inherently have variable timing models. The pin-to-pin propagation delay depends upon the number of logic cells cascaded together to achieve a given logic function. There are two types of variable timing models: simple and fine structured. In the simple variable timing model, the pin-to-pin propagation delay is chiefly dependent upon the number of cascaded logic cells, signal fan out, and the wire delay that would normally be encountered in a sea-of-gates gate array. The number of interconnect points in the signal path and the logic function implemented in the logic cell tend to have secondary and lesser effects on the timing. Architectures suitable for the simple timing model will have actual device delay characteristics which are independent of where the logic cell is placed in the array. In the fine structured variable timing model, pin-to-pin propagation delay is strongly dependent upon not only the simple model factors but also the number of programmable interconnects in the signal path and the function implemented in the logic cell. In these cases, the logic cell itself has a variable timing model due to its complex structure and antisymmetry. Programmable interconnect points with unfavorable electrical characteristics raise the effect of the number of interconnect points in a signal path to being a first order effect.

Table 5. Electrical Characteristics of Fuse Technologies

Technology	ON Resistance	OFF Cap.	T	Tgate
Via Fuse	50 ohms	1 fF	0.05 ps	50 ps
Large Fuse	400 ohms	5 fF	2 ps	400 ps
RAM Cell	800 ohms	10 fF	8 ps	800 ps

The actual performance of devices does not differ by these orders of magnitude. This is because the OFF capacitance of a large number of no-connect fuses is small compared to the metal and gate input capacitances. Therefore the fuse series resistance is the dominant component in limiting performance due to programmable interconnect. To put this factor into perspective, *Table 5* includes a column which is the time constant for one series fuse connected to a gate with a capacitance of 1 pF. Note that with as few as five programmable interconnects in the path from the signal source to one gate, the time constant, for some technologies, can be as large as the gate delay itself.

Interconnect and Logic Cell Trade Off Summary: Advantages and Weaknesses

The technology has a profound effect on the total FPGA architecture. SRAM and large fuse based technologies cause interconnect-constrained architectures which force non-optimal logic cell architectures. In general, interconnect-constrained architected logic cells tend to be large and complex to make up for the interconnect limitations. Moreover, these complex logic cells tend to be wasteful of resources in certain applications. Such architectures are characterized by routing and capacity limitations and an inability to fit a design when there are pinout constraints (user fixes signals to particular pins). Router and fitter software may take many iterations to fit a design.

It is clear that the small-size fuse technology, combined with well chosen routing channel wire complement and an optimum complexity logic cell, will yield a high performance, small die size device. *Figure 7* shows system performance of a fixed benchmark fitted into devices of the three technologies described above. The system performance is plotted versus the semiconductor process technology line width in order to perform an apples to apples

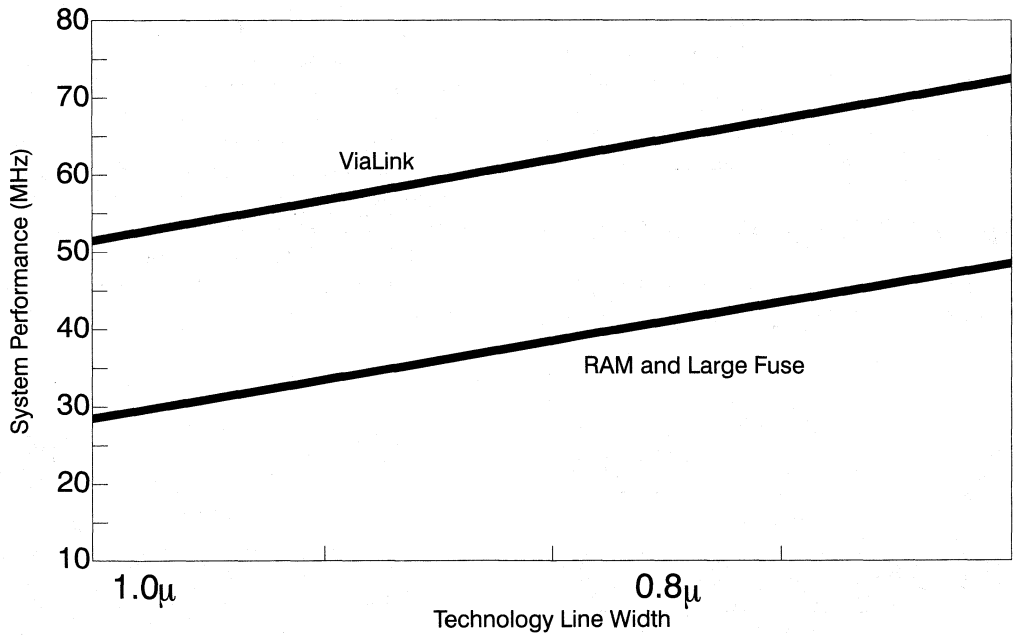


Figure 7. Performance Relative to Connect Technology

comparison of the key factors in the FPGA implementation. As expected, the architectures optimized for ViaLink™ exhibit a significant performance advantage over the large-fuse and RAM-based interconnect approaches.

Comparison to CPLDs

The architecture of the FPGA manifests itself in the device characteristics in much the same way that the sea-of-gates gate array architecture does. The two major influences on the device characteristics are the small logic cell size and the channel routing. First, the cell sizes are small and considerably less complex than those of the CPLD. Therefore, the propagation delay through the FPGA logic cell is much smaller than that through the CPLD macrocell. Functions such as multiplexing, which need only one cell per signal path, will typically achieve much higher performance in the FPGA. In contrast, functions which require cascading of many logic cells to implement may be at a performance disadvantage in the FPGA. Complex state machines with a lot of decoding are in this category. This does not mean that use of the FPGA is to be avoided. In the example to follow, a complex state machine is implemented successfully. Secondly, an abundance of routing resources can permit complicated interconnects as well as convenient handling of buses.

Cypress 380 Family FPGA Architectures

The previous architecture discussions have pointed out the strong relationship between the technology, the architecture of the FPGA, and the device characteristics. The 380 family possesses a unique technology which impacts all of the remaining architecture trade offs positively. The discussion of the 380 family begins, therefore, with a presentation of the interconnect technology.

380 Family Fuse Technology

In usual integrated circuits two crossing metal lines that are on different layers may be connected by a via. A via is a small hole in the insulating glass that lies between the two layers of metal. This small hole, which is about the size of the metal lines themselves, is filled with metal from above making the connection to the underlying metal line. The programmable via is a modified via used in standard CMOS semiconductor processing. The modification consists of depositing a thin layer of amorphous silicon in the via hole so that the silicon separates the two layers of metal. As manufactured, this special via has a resistance in excess of 1 gigaohm and an insignificantly small capacitance (about 1 fF). Its size is no larger than the standard via normally used to connect two layers of metal. A cross section of the programmable via is shown in *Figure 8*. A programming pulse applied across the programmable via causes a change in the characteristics of the silicon layer forming a bidirectional conductive link between the top and bottom metal. This programmed link has a series resistance of about 52 ohms and in practice is no more than 65 ohms. The parasitic capacitance is no larger than a normal metal to metal via. The technology is appropriately termed "ViaLink."

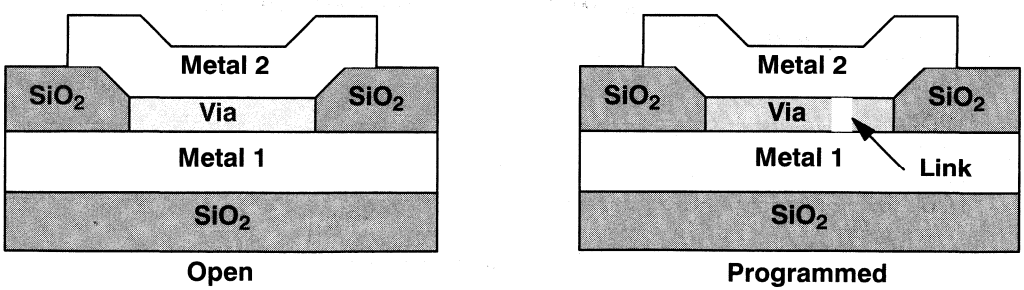


Figure 8. The ViaLink

Routing

ViaLink technology has significant impact on FPGA architecture. Since the programmable site is no larger than the associated metal interconnect wires, there is no real restriction on the number of interconnect points (fuses) and no fuse related restrictions on the number of wires in the interconnect channels. The 380 family routing scheme is architected with this added freedom.

Four types of signal wires are employed in the routing channels:

- segmented wires
- quad segmented wires
- express wires
- clock wires

Segmented wires are wires that extend only from one routing channel to the next, both vertically and horizontally. At the channel junction, a horizontal segmented wire may be programmed to interconnect to a vertical segmented wire at points called cross links. In *Figure 9*, programmable cross links are denoted by the open circle at intersections of vertical and horizontal wires. Also at the channel juncture, the segmented wire may be continued in the original horizontal or vertical direction by connection to another segmented wire running in the same channel. This connection is provided by a pass link. These links are denoted by an "x" in the figure. Segmented wires are most applicable for local wiring around or between adjacent logic cells.

Quad segmented wires are similar to the segmented wires described above except that the wire extends across four logic cells before it is segmented. Like segmented wires, the quad segmented wires may be continued to the next quad segmented wire by a pass link. The quad segmented wires are applicable to signal distribution over a larger but still local group of logic cells.

Express wires are similar to segmented wires except they do not include pass links. An express wire will therefore run the entire length of the device. These wires are most suitable for global signals within the device. Routing software with specific knowledge of the device architecture will automatically route signals over the appropriate wire type.

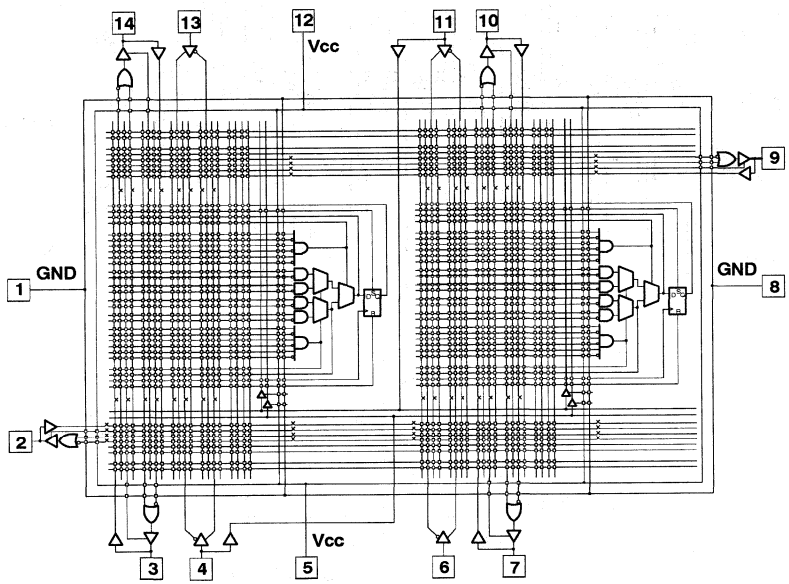


Figure 9. Simplified 380 Family Model

Clock wires are special signal lines that include an array of buffers for minimal skew. Clock wires are similar to express wires except that the cross links are limited. This is to insure that the clock wires are lightly loaded by programmable interconnects and can be used maximally in routing high-speed clocks or reset signals globally throughout the device with minimal skew. The source of the signal on the clock wires is specific device pins with the designation "I/CLK." After passing through the special input buffers, the signal is routed horizontally across the center of the die, as shown in *Figure 10*. There are four high drive buffers. One pair drive clock 1 and clock 2 to the upper half of the column of logic cells, and the other pair drive the two clocks to the lower half column of logic cells. There is a cluster of these buffers for each column of logic cells in the array. The buffers can be enabled to drive the clock lines or disabled if a clock is not required in a given column.

Vertical channels include all three wire types plus V_{CC} and ground wires. The V_{CC} and ground connections allow unused inputs of any logic cell to be tied to an appropriate logic level. The vertical channels run to the left of each logic cell column and extend the full height of the device. The I/O wires, which run from each of the logic cells to the right of the vertical channel, intersect the wires of the vertical channel with cross links at all segmented wires and at judicious points for express wires. At the extreme ends of the vertical channels are I/O cells that connect to the device pins. The number of wires in the vertical channel is chosen to be commensurate with the number of inputs and outputs of a logic cell, the added wires for V_{CC}, ground, and the I/O cells at the device periphery. There are 24 of these wires.

Horizontal channels provide connection by way of cross links from vertical channel to vertical channel

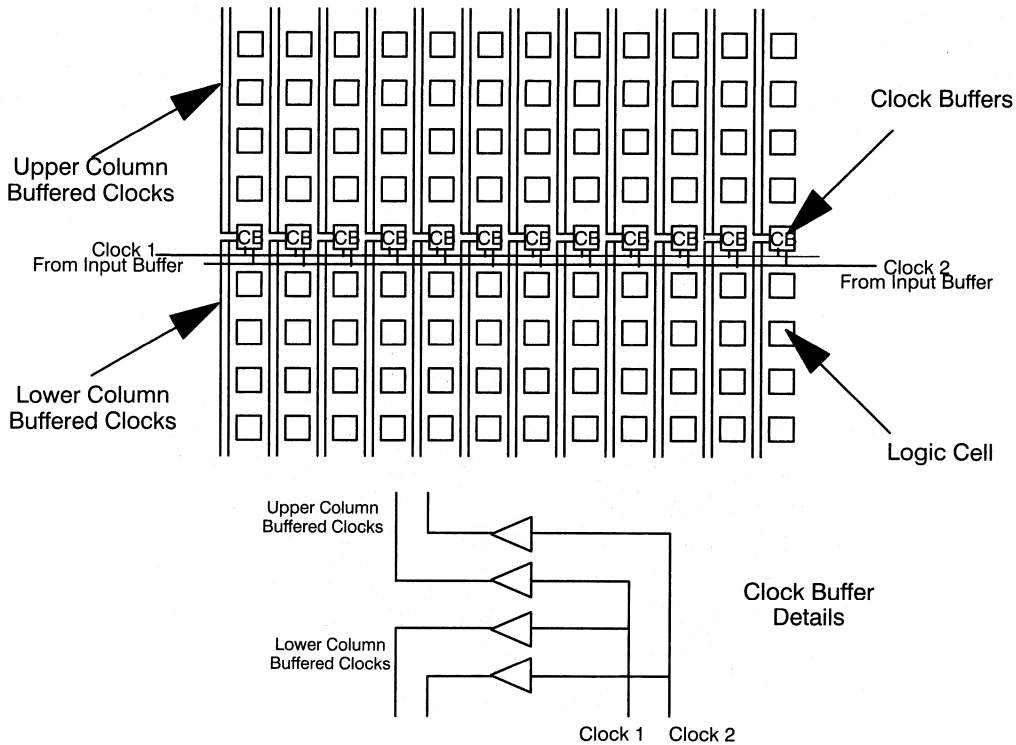


Figure 10. 380 Family Clock Distribution

and from the vertical channels to I/O cells on the left and right periphery of the device. All wire types are included in the horizontal channels (which contain 12 wires each) except for the clock wires. (These are the dedicated wires that carry the clocks to the buffers.)

I/O Cells

There are three types of interface buffers that connect the internal array to the device pins. The dedicated input buffer provides high drive internally and generates both true and complementary versions of the input signal. This high drive capability allows signals coming from these input only buffers to fan out to a larger number of cells than the normal I/O cell. The clock input buffer is similar to the dedicated input buffer except that it provides a third output that is routed to the internal clock distribution buffers described previously. The I/O cell provides a bidirectional connection to the devices pins. The cell can be used as input only, output only, or a bidirectional pin connection. Internally the cell has an output enable, an input data connection, and two output data connections which are ORed together to produce the output. This cell is shown schematically in *Figure 11*. The output driver provides 8 mA drive level (I_{OH} and I_{OL}).

Logic Cells in the 380 Family

Since the routing resources of the 380 family are abundant and without expectation of being interconnect constrained, there is freedom in the logic cell architecture to choose the optimum complexity. The 380 family logic cell is shown in *Figure 12*. This cell has been optimized to maintain the speed advantage of the ViaLink technology while insuring maximum logic flexibility.

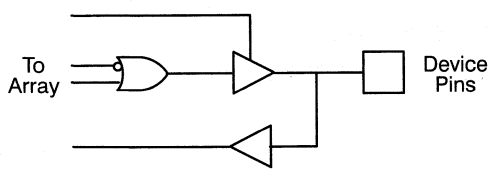


Figure 11. Bidirectional I/O Buffer

The logic cell consists of two 6-input AND gates, four 2-input AND gates, three 2-to-1 multiplexers and a D flip-flop. This cell represents approximately 30 gate equivalents of logic capability. The cell has 23 logic and control inputs and 5 outputs. The arrangement of the gates permits 14-bit-wide gating functions and can realize all possible Boolean transfer functions of up to three variables. The D flip-flop possesses asynchronous set and reset inputs to independently control the output state. The multiplexer and logic feeding the D input allow the flip-flop to be configured as D, T, JK, or SR.

The outputs of the logic cell include the Q output of the flip-flop (QZ) plus four other outputs tapped at selected points within the logic cell. The OZ output is the same as the D input to the flip-flop. The OZ output facilitates combinatorial functions. The three other combinatorial outputs tap the logic cell at selected places. If simple logic functions are to be implemented, the multiple outputs permit more than one of these functions to be realized in a single logic cell. Maximum use of the available logic can be made. Note the ability to provide this multifunction utilization without any significant impact on routing. The additional utilization factor is ob-

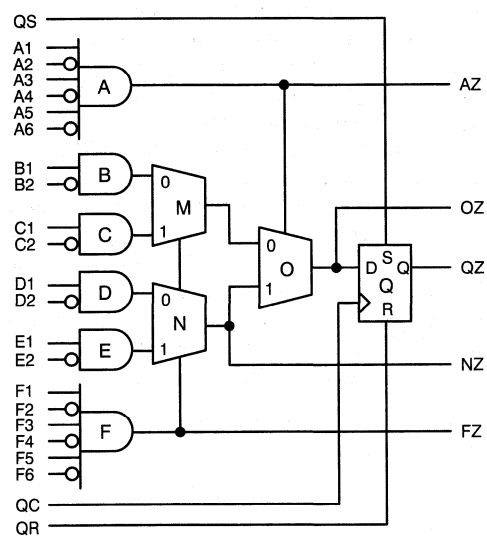


Figure 12. pASIC380 Internal Logic Cell

tained for free. When implementing multiple functions, the flip-flop may still be employed in many cases.

The logic cell is not so complex as to adversely impact propagation delay. The internal multiplexers are positioned to participate in implementing logic functions. Since the multiplexers are all in the path to the D input of the flip-flop, they contribute significantly to combinatorial logic function realization and are not expended on signal steering. The logic cell is also noticeably symmetric and regular. Combinatorial delays are thus also symmetric. That is, input to output delays tend to be roughly the same, although the AZ and FZ output will be faster than the others. Whereas some architectures bypass large sections of cell logic by the multiplexing, thereby making the cell delay dynamically changeable, the 380 logic cell delay is not subject to this condition.

Performance and Timing Model

An inherent characteristic of any FPGA is that the timing model is a variable model: logic implementation is accomplished by cascading a number of logic cells that is dependent upon the function to be implemented. For the non-ideal FPGA model, the device input to output propagation delay is a function of the number of logic cells in the signal path; the dynamics of the signal path in the logic cells; the number of programmable interconnects through which the signal traverses; the normal integrated circuit routing delay; and the I/O cell delay. This relationship can cause the variable timing model to be quite complex and depend upon the routing, placement, and cell dynamics. Since the 380 family has a fixed timing model for each logic cell, the cell configuration dynamics are not a factor in the 380 family timing model. Only the cell delays need to be summed for their contribution to the overall delay.

The 380 family programmable interconnect affects the propagation delay in much the same way as normal integrated circuit interconnects. That is, the fuses contribute to the delay as if they were slightly longer wires. This characteristic greatly reduces the complexity of the timing model and the variability in the timing results when a design is fitted to a device.

Conclusions and Summary

This application note is a first introduction to FPGAs. Specifics of the Cypress 380 family of devices were presented. It was shown that the fuse or connection technology has a very strong influence on the architecture of the FPGA logic cells and the interconnect scheme. Specifically the physical size of the interconnect (fuse or RAM cell) and its ON resistance are major influences on the FPGA logic cell complexity and interconnect architecture. The low ON resistance, physically small fuse permits

- an interconnect scheme virtually unconstrained by the number and location of fuses
- an optimized logic cell architecture
- device performance with minimal limitations from the fuse electrical characteristics

When an FPGA is architected with the freedoms listed above, the results are significant benefits to the user:

- FPGAs where designs are easy to fit, i.e., large capacity (non interconnect constrained)
- Flexibility in pin assignments
  (user can define/keep fixed after modifications)
- High performance/low propagation delays
- Easy to use timing models

When these are primary concerns, the small fuse technology offer the greatest opportunity for extracting these benefits.

Document #: PLD0001–0194

ViaLink is a trademark of QuickLogic Corporation.

Designing with FPGAs An Introduction to Cypress's 380 Family of FPGAs and the *Warp3*™ Design Tool

Introduction

Field Programmable Gate Arrays (FPGA) borrow the sea of gates concept from the gate array semicustom integrated circuit and add field programmability. The similarity of the FPGA to the semicustom approaches opens many possibilities for the design engineer. With a large number of gates available, complex designs can be implemented into a single device. In the semicustom approach it may be many weeks from sign off to prototypes. Moreover, simulation is usually exhaustive (due to the cost of a design change), taking many weeks of the design cycle. With field programmability, a design may be realized in a device in a week or two of design time, in contrast to the many months with a semicustom approach. The FPGA brings gate-array-like possibilities to many design projects. The lack of a non-recurring engineering charge for FPGAs makes this technology financially available to a large number of developments.

This application note is intended to be an introduction to using FPGAs by taking the reader through a complete design. The first part of this note presents the design tools for FPGA designs. Here a design flow is followed from design entry in its multiple forms. The design flow is top down. That is, the process starts from a description of high-level abstract entry of the design and progresses to adding more hardware-specific details as required for realization in a physical device. To illustrate the back-end part of the design process, an actual design is presented. In this part, details of speed optimization, simulation, and device specifics required in the design description are discussed.

Design Development and Cypress's *Warp3* Design Tool

FPGA devices are resource-rich entities capable of implementing designs that use up to 12000 gates. These designs will not be simple, consequently the development of the design, its debugging, and final performance analysis will be a complicated task. Fortunately, Gate Array design and analysis tools can be used to make these tasks comparatively easy. The design proceeds beginning with the entry of the design description. This may be in schematic form or in a high level VHDL language form. After entry, the VHDL code (or VHDL equivalent of the schematic) can be simulated directly to verify the functionality of the design description. The design is then synthesized and committed to a particular device. At this point the performance of the implementation can be analyzed and optimized if necessary to achieve a design target. Lastly, the actual device is programmed.

Warp3 is a modern, self-contained PLD design tool that supports all phases of the design process. At the front end, it includes VHDL and schematic entry design capture tools for efficient and convenient user entry. The synthesis tools include native compilation of VHDL (for accurate VHDL interpretation), and Cypress device-specific compilers for maximum utilization of device architectural features. The schematic capture, simulation, and the

framework are ViewLogic™ tools adapted specifically for PLD, CPLD, and FPGA designs.

VHDL Design

VHDL is a rich and powerful language for the description of logic circuits. The language offers the capability to use different styles of design entry. There are three styles of logic description that can be used in any combination: behavioral, Boolean, and structural. Behavioral descriptions are C- or Pascal-like constructs that specify the action of the logic in high level abstract terms. Language constructs such as the IF/ELSE statement or the CASE statement specify the behavior. Boolean equations can also be used to describe the logic. Tabular descriptions are also possible. These can be considered a subset of the behavioral descriptions but where the action of the logic is specified by a truth table. The structural description is much less abstract and can be considered a verbal description of a schematic. In one version of this form of VHDL, gates, flip-flops, and other primitives are instantiated, and their interconnection described through signals that tie the output of one primitive to the input of others. Conversely, several entities can be instantiated in a structural description of their interconnect, but the description of any of the individual entities can be abstract behavioral.

VHDL is a hierarchical language. Just as most programming languages support subroutines, functions, and procedures, VHDL supports components, packages, and the ability to combine a set of entities into a higher-level entity. A complex VHDL design can be built by successively combining building blocks in related layers. This allows two, very powerful design approaches. First, a complex design can be done from a top-down approach. In this method, the whole of the design can be described in very abstract, high level terms. Then the design is decomposed—breaking the design apart into specific functional blocks that are described in more specific terms. This moves the design from concept to implementation, from abstract description to near hardware level realization and optimization. At the top level, the designer need not be concerned with the exact details of the design. The concern at this top level is the conformance of the design to the given functional specification. Once this is achieved, the design can be decomposed for realization purposes while being assured that the overall functional requirements are still met. Even at the top level the design can be built up of manageable entities which can be debugged separately. Second, the design can be done from a bottom up approach. After a block diagram is sketched out, the individual blocks can be designed according to their function and the interfaces to the other blocks. The individual block designs can be done at the most detailed level. This can consist of schematics using components instantiated from a library or the design can be a structural VHDL description. With each block fully designed, they are then connected together to build the complete design.

As an example of VHDL, consider a 12-bit wide 4-to-1 multiplexer. A version of this multiplexer is used in the design example section of this application note. The first input is a 12-bit bus that is the column address for the DRAM. The second input is a select signal which controls whether the row (row_ad) or column (col_ad) is selected. The third input is a 12-bit bus that is the row address for the DRAM. The forth input is a 12-bit bus that is the refresh address for the DRAM, and the last input is the state of the controller finite state machine. The output of the multiplexer is a 12-bit address to be sent to the DRAM memory devices. When the finite state machine is in states *refad*, *wr1*, or *wr2*, the refresh address is placed on the multiplexer output. This selection is independent of the state of *col_sel*. When the finite state machine is not in states *refad*, *wr1*, or *wr2*, COL_SEL controls the multiplexing. When COL_SEL is 0, row_ad is placed on the output of the multiplexer, and when COL_SEL is 1, col_ad is placed on the output of the multiplexer. The VHDL code to implement this function is given in *Figure 1*. Note the compactness and simplicity of the description. One of the twelve synthesized logic equations is given in *Figure 2*. These logic equations are available in the report file generated during compilation. The equivalence of the behavioral code and the logic equation should be self evident: in the logic equation, the state machine state vector bits (for the states in the if statement) are ANDed with the refresh address bit (REF_AD(11)) and the

```
mux: process(col_ad,col_sel,row_ad,re_ad,state)
  begin
    if(state = refad or state or state = wr1 or state = wr2) then
      rc_ad <= re_ad;
    elsif(col_sel = '1') then
      rc_ad <= col_ad;
    else
      rc_ad <= row_ad;
    end if;
  end process;
```

Figure 1. Behavioral Description of Multiplexer

```
rc_ad_11_ = /controller_state_12_.Q * /controller_state_11_.Q *
            /controller_state_10_.Q * /controller_state_11_.Q *
            stored_ad_11_DFF.Q * col_sel.Q
           +
            /controller_state_12_.Q * /controller_state_11_.Q *
            /controller_state_10_.Q * /controller_state_11_.Q *
            stored_ad_23_DFF.Q * /col_sel.Q
           +
            controller_state_12_.Q * ref_ad_11_DFF.Q *
           +
            controller_state_11_.Q * ref_ad_11_DFF.Q *
           +
            controller_state_10_.Q * ref_ad_11_DFF.Q *
           +
            controller_state_9_.Q * ref_ad_11_DFF.Q *
```

Figure 2. Logic Equations for the Multiplexer

compliment of these bits are a factors in the remaining product terms. One of the other two product terms ANDs COL_SEL with col_ad(11); the other product term ANDs COL_SEL with row_ad(11). The logic equations refer to stored_ad_11 instead of col_ad[11] and stored_ad_23 instead of row_ad[11]. This is because col_ad and row_ad are aliases for these signals. Refer to the appendix for the alias definition.

Schematic Entry

In some cases VHDL may not be the preferred method of capturing the design. A discrete implementation of the design may already exist with the objective of reducing size and improving performance by putting the circuit into an FPGA. Many designers may feel more comfortable with schematic design capture than with a high-level language description.

Mixed Mode

Some functions are difficult to describe directly in schematic form. A state machine, for example, is far easier to describe in terms of a transition table (possible in VHDL) or VHDL conditional constructs (for example, a case statement). Not only is the description easier but design changes and debugging

are also far easier in nonschematic form. It is therefore important for a tool to be capable of mixed mode design description. In its most probable form, it is desirable to place and connect a component into a schematic where the function of the component is described in VHDL.

Whether the design is done in VHDL, schematic, or mixed mode, the design tool transforms the user's captured form of the design into VHDL. From this VHDL description, the the design synthesis and compilation takes place. This is important since a schematically captured design is collapsed in the synthesis process. Several layers of elementary gates are combined where possible into a single AND/OR plane, thus removing redundant gates. The final implementation may therefore look quite different than the original schematic.

Source Level Design Verification

It is very convenient to verify the functionality of the design at the VHDL source code level. Clearly, debugging at this stage saves considerable time and effort in that the design does not have to be synthesized and fitted to a device before simulation can take place. *Warp3* features a VHDL source level debugger that will simulate VHDL code and produce functional results. The results can be as graphical waveform displays, active line indication in the source code, and tabular displays of variable values. Various other debug facilities are included. The VHDL code can be conveniently debugged at this level leaving the post compilation simulation to speed optimization.

Design Decomposition

Up to this point the discussion has not implied any particular hardware. The assumption is that the designer has captured the scope of the design into one abstract level VHDL description. The design could also have been entered schematically. This description may be realizable in a broad range of hardware devices. Memory, interface, microprocessor, one or more programmable logic devices, and a variety of other integrated circuits may be required to realize the design. Therefore the design must be decomposed into those specific devices which the designer chooses for realization. There may also be more than one programmable logic device. This decomposition process must be done manually by the designer. The task includes the selection of the particular programmable logic devices which will be used and the logic functions to be included in each. Although discussed, there are no tools, as yet, which can automatically select device types and partition the logic among them.

Assuming that this partitioning has been performed and a particular FPGA has been selected for a certain function, further decomposition tasks must be performed. Assume that the partitioning has been done where a particular set of functions are to be implemented into an FPGA. The VHDL source code or schematic is decomposed into components which represent the real hardware. Once a pASIC™ device has been selected, the user must

1. connect clock(s) to the clock distribution network (if desired)
2. connect any high fanout (to internal cells) inputs to the high drive input cells (if desired)

The design then proceeds through the synthesis and place and route processes without user intervention. Most designs will produce excellent results in both performance and fit. The following paragraphs describe in detail how the two steps itemized above are performed.

As a specific case, consider the design example that is described in later sections. There are several signals which have high fanout. These are identified by the design tools, which issue warnings during the place and route process. These warnings list the signal names and the fanout (number of loads). These input signals are candidates for use of the high drive input buffers of the device. It is clear from the start that the CLK signal should take advantage of the internal clock distribution tree. Either anticipating these conditions or addressing the warnings posted by the tool, hardware-specific statements are added to the VHDL source file or schematic to address the issue. First, the statement

```
use work.rtlpkg.all;
```

is added to the VHDL source code above the architecture description. This invokes a library which

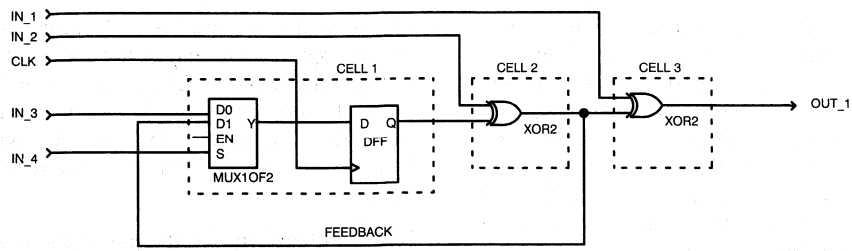


Figure 3a. Original Schematic Diagram

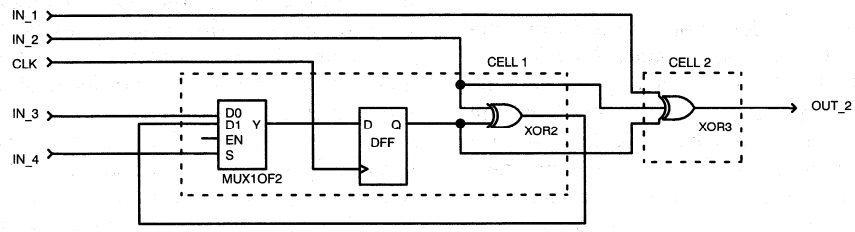


Figure 3b. Modified Schematic that Allows Packing of XOR2 into Cell with Mux and Flip-Flop

contains the special input cells. In the body of the architecture, these components are instantiated:

```
ck1:PAckcell
   PORT MAP(clk, ck_x, ck_y, clk_in);
```

defines the use of the clock buffer tree and associated input cell for the clock signal, and

```
hd1:PAincell
   PORT MAP(as, as_x, as_in);
```

defines the use of a high drive input cell for the AS (address strobe) signal. The place and route tool assigns pins but will place CLK and AS on a pin with the appropriate driver.

Similar statements can be added to address other signals whose fanout suggests that they be assigned to pin with high internal drive characteristics. With the addition of these statements, the design description has, of necessity, become less behavioral and more device specific.

Designs can have special requirements that require special attention in the design process. A typical concern is performance. The previous discussion has presented use of the internal clock distribution trees and the use of the special high drive input buffers. In this section, specific topics will be presented that are typical approaches to performance optimization. This is not intended to be a complete description of optimization procedures, but is intended to give a general introduction of a few approaches. The first discussion will focus on schematic entry using library components. The last part of this section will focus on path analysis in the place and route tool and optimizations that can be performed there. A place and route with a user applied constraint will be presented in the Design Example Section.

Take for example the logic shown schematically in *Figure 3a*. When synthesized, placed, routed, and packed, the multiplexer and flip-flop are packed into one logic cell, and the two exclusive-OR gates are each placed in separate logic cells. The clock to feedback path delay is two cell delays: one for the mux/flip-flop logic cell and one for exclusive-OR1. When the schematic is redrawn as shown in *Figure*

3b, the result is different. In this figure the functionality of the circuit is identical to that of *Figure 3a*. Schematically however, the output of exclusive-OR1 is dedicated to the feedback signal and its function, while this same function for the output signal is replicated in exclusive-OR2. When this schematic is synthesized, placed, routed, and packed, exclusive-OR1 is packed into the same logic cell as the multiplexer and flip-flop. This result reduces the feedback path delay from two cell delays to one cell delay with an accompanying improvement in performance.

Depending upon the design objectives, further design decomposition may be necessary. Given the example described above, if this circuit happens to be a critical speed path, the action taken to eliminate one logic cell of delay may have made the difference between a successful design and one that may have been marginal or require a higher speed bin device. There are two approaches to this decomposition: VHDL structural design and schematic entry. Schematic entry is by far the simplest approach. If the circuit is captured schematically, and if all signal paths of interest in the schematic are labeled, the realization of the circuit in the FPGA can be ascertained by opening the Place and Route tool and viewing the wiring placement. Identifying the labeled signals in this schematic view (Physical View) of the device allows tracing the signal paths through the logic cells and determining the details of the implementation. With potential reconfigurations identified, the schematic can be redrawn to effect propitious changes for optimization improvement.

To make the discussion about optimization clear, an explanation of the design flow and the tool functions needs to be done. When a design is entered, it can be initially done without regard to any target device. This allows choosing the target device later in the project at a time that is optimal for making the decision. VHDL is device independent unless device specific statements are made. Schematics are constructed from elements contained in a library. Likewise, VHDL descriptions can instantiate components from the same library. Even though library components are used, the design remains non-device specific. When a target device is specified, the synthesis tools use an architecture description of the library component specific to the target device. This permits optimum fit and performance of the design in the target device. In addition to the generic library of components, there is a library with components that are unique to a target device family. These components are not useable with any other family.

With this background, the discussion can return to the subject of optimization. With behavioral VHDL, the optimization task is more difficult than with schematic entry. First, because of the abstract nature of the description, individual signals cannot be given a label with which to assess the implementation. These signals simply may not exist at the VHDL behavioral level. Knowledge of the defined signals must be used to assess how particular behaviors are implemented. If a particular section of the circuit is identified as being speed critical either in the analysis phase of the place and route process or from the start of the design, it may be worthwhile to capture this part of the design as a structural rather than behavioral description. Without considerable experience this may not prove as useful as using schematic entry with its signal traceability to the layout, availability of library components, and more predictability in the results. (As stated in the previous paragraph, when a target device is specified, the optimum architecture of the library components is chosen.) If schematic entry is chosen as a path for design optimization purposes, two approaches are possible. The critical portion of the design may be schematic captured, exported to a VHDL description and then combined with the remaining behavioral VHDL described portions of the design. Alternatively, the behavioral portions of the design can be given symbols and incorporated into a schematic. The underlying VHDL code will pass directly through the schematic to the VHDL code created in the export process (export1076 tool). In extreme cases, experienced designers may use device-specific library components to achieve exacting requirements.

This optimization is more the exception than common practice. Most designs will compile directly from the VHDL or schematic description, be placed in the target device and operate satisfactorily. The optimization tools do an excellent job in producing

successful results. Only in special applications should the need arise to intervene to the level described above to achieve results.

Synthesis, Optimization, and Place and Route

After the design has been captured, the remainder of the design process is transparent to the user. In the process from the design capture to the final product, the user may interact with the development tool to select certain optimization options and to apply performance constraints.

Once the design has been captured and converted to its VHDL equivalent if the capture was done in schematic form, the design goes through a synthesis and compilation process. Synthesis is the first step; it is the transformation of the design input at a high level of abstraction to a lower level. In this case, the VHDL description is turned into an unoptimized Boolean level description of the design. This unoptimized Boolean description is flattened, that is, it has no hierarchy and all intermediate variables are removed. At this point the design is not device-specific and is in a sum-of-products form. The synthesis process should be native. Some tools have laid a VHDL input over an existing tool set. The VHDL is not synthesized directly in these cases (not native) but is first converted into the usual input form of the underlying tool. In such cases the output should be suspect since it is not derived from a direct interpretation of the original VHDL code.

Several steps are performed next. First, the synthesized design is transformed into a device-specific netlist. This netlist is then passed to the place and route tool. Here, the technology mapper transforms any structures that are not yet device specific into forms that can be realized by the logic cells. Second, the netlist is reduced to use a smaller number of logic cells, or adjusted to give better speed instead of reducing logic cells. Two tools are available for these purposes: the technology mapper (a second version of the one mentioned above) and the packer.

The packer does not perform any logic optimization. Assuming all necessary device-specific mapping is done in the first step, the packer maps multiple macro symbols into logic cells. The result is the densest and fastest possible implementation. The packer performs its operation in three phases. First it analyzes the physical implementation and determines which functions may be implemented in only part of a logic cell (a fragment). This is in preparation to split logic cells in order to use a single cell for more than one function. This was discussed earlier when the optimal logic cell architecture was presented. Second, the packer searches for functions that are logically related and offer packing opportunities. This not only reduces the number of logic cells to be used but reduces the place and route task by identifying functions that will be pre-placed in the same cell. Third, the packer searches for unrelated logic functions that may be grouped into the same logic cell.

The optimizing technology mapper is an alternative to the packer. This tool offers optimization choices. It may be desirable to optimize for speed or it may be desirable to minimize the die area used by the design. In the case of an FPGA, this latter choice may be necessary to get a design to fit into a particular device. In contrast to the packer, the technology mapper performs optimization.

The last step is the actual placing and routing of the design into the actual device. The place and route software is an extremely important part of the design compilation. It accepts the output of the technology mapper or the packer and generates a file for use by programming equipment and a timing model for use by analysis tools. The place and route determines the actual cell choice for each logic function to be implemented and the wire connections for the signals which pass to and from the logic cells and the I/O cells. Once the place and route is completed, a pictorial representation of the cells and the interconnection is provided through the physical viewer. Here the used logic cells and the used routing are displayed with appropriate annotation linking the physical view to the original source input (schematic or VHDL).

Once the place and route is completed, a performance analysis is usually done to verify that the design will meet the expected performance. This is done by a timing analyzer associated with the place and route tools. Once placed and routed, the output will contain results of the delay calculation for each

internal signal path. The timing analyzer uses this data to provide a tabular presentation of all signal delays. The user may select options for timing analysis. The tabular output can be filtered to display only certain types of signal paths and their delays. The user may select start points, such as clock input, general inputs, or internal flip-flop Q outputs, and end points such as output pads or flip-flop D inputs. The tabular display shows the selected signal path types and the associated delays. The signal path delays may be analyzed to a fine structure including the wire interconnection delay. Any signal path can be selected for a fine structure display of the individual delays involved in the start to end delay. Moreover, the design tool can display the selected signal paths on a pictorial representation of the die (physical view). This is useful to examine any user-imposed pinout impact on the signal path lengths and routing paths. Constraints may be applied to specific signal paths, both pin to pin and internal, in the tabular performance results. The user can then request a refit imposing the constraints. Through an interactive, iterative process, the user can direct a fit that optimally meets his design objectives. The final simulation verifies the performance specifications.

Automatic Test Vector Generation

Some programmers are equipped to exercise a programmed device with a user supplied functional test program. Such programmers have enough hardware to permit driving and sensing all pins of a device. *Warp3* can generate test vector files for these programmers.

A Design Example

A design example is presented in order to illustrate the significant features of the 380 family FPGAs and the *Warp3* design tools. The design is a DRAM memory controller that interfaces to a system address and control bus on one side and a DRAM memory array on the other. The controller includes a slave bus interface, a DRAM address generator with burst transfer capability, and a state machine to effect the DRAM control signal timing, refresh, and bus handshake. This example was chosen because of its wide variety of implementations: a state machine, counters, registers, signal multiplexing, and decoding. This example is not meant to be a full featured DRAM controller. Certain obvious functionality is left out so as not to obscure the illustration objectives of the example. Specifically,

- there are variable splits in the row/column address multiplexing depending upon the DRAM type populating a given address range.
- the RAS/CAS that is asserted depends upon the address range occupied by a given DRAM type that happens to be present
- the column address does not use bus addresses [3:0] for a 64-bit data bus because these are byte addresses
- the actual refresh counter is left out of the synthesized design

The design is done as a high-level behavioral design with no hierarchy. The design is divided into four sections: the address and control register, the state machine, the refresh counters, and the address multiplexer. This is a natural division producing design tasks that are easy to grasp. Each section is coded in VHDL and simulated separately to verify the functionality of each section. Simulation at these simpler levels is easier to accomplish and can be more thorough than simulation of the entire design. Simulations at this level attempt to verify all details of the operation of the particular section. After each section is functionally verified, they are combined into the top-level controller description, which is then functionally simulated again except with a different focus. After the design verification, it is then compiled and performance analyzed. The following paragraphs will describe the design process, present representative samples of the outputs at each step, and highlight significant results of tool and device architecture capabilities.

Detailed Design

The entire controller is described as a single entity with a behavioral level architecture description. Each section of the design is coded as a separate process. These sections are described below. The entity declaration includes hardware-specific attribute statements. These specify the target device and

particular pin assignments to take advantage of device-specific features for clock distribution and high fanout signal distribution. Other non-behavioral declarations are made later in the design for optimization purposes. In all other respects, the design is purely abstract.

The address register interfaces to the main system bus and captures the address and control information for the memory transaction. The register is synchronous. The system bus clock is applied to the register, which captures bus data on the rising edge of the system bus clock while the address strobe is asserted. The output of the register supplies the address and control information to the rest of the circuit. The VHDL description of the register is given in *Figure 4*. The register circuitry includes handshake flags to communicate with other parts of the controller. When an address strobe is detected, a flag is set to inform the state machine that a bus transaction has started. The handshake is closed by the state machine clearing the flag, indicating that it has been recognized. The flag is necessary since, as will be seen in the later parts of the design, the state machine may not always be able to respond to (recognize) this one clock cycle event. The address register circuitry also includes comparator circuitry to see if the address is valid for the memory array. This

```
adreg: process
   begin
     wait until clk = '1';
     if(reset = '1') then
       as_flag <= '0';
     elsif(as = '0') then
       as_flag <= '1';
     elsif(clr_as = '0') then
       as_flag <= '0';
     endif;

     if(as = '0') then
       stored_ad <= address;
       burst_stored <= burst;
     end if;
   end process;
```

Figure 4. VHDL Behavioral Description of Address Register

comparator produces the match result combinatorially from the register output. There are other architectural options: making the address register a transparent latch and/or placing the address comparator inputs on the input side of the latch. For simplicity, these options are not pursued.

A refresh counter is required to provide a refresh address to the DRAM memory as well as to make request of the controlling state machine to periodically execute a refresh operation. The refresh counter does not add any instructional value to the description here. It is therefore not included in the example details. The refresh circuitry provides a refresh request to the state machine and a refresh address to the address multiplexer. These are set to zero in the example VHDL source code.

The column address counter produces addresses for a burst transaction. In these transactions the system provides the first address of the burst and the remaining addresses are expected to be generated by the slave. The column address counter is loaded, under control of the state machine, with the low-order portion of the incoming address and is advanced also under control of the state machine. The counter also serves to hold the column address in those cases where a new address is received and the state machine is not yet finished with the present address. The counter is a fixed width for a given transaction burst length. The counter provides an address which wraps around the maximum length of the counter. This is consistent with burst transactions that do not start the transaction at the lowest address of the items in the data burst. The counter is configured to count in Intel order (refer to the *i486 Hardware Reference Manual*).

The address multiplexer is combinatorial logic that produces twelve address outputs for the DRAM array. Under control of the state machine, the multiplexer can output the refresh counter address, the column address, or the row address. The column address is the lowest-order bits of the address contained in the address register. The row address is a selected set of the upper-order bits of the address contained in the address register. The particular upper-order bits depend upon the DRAM type used in the array.

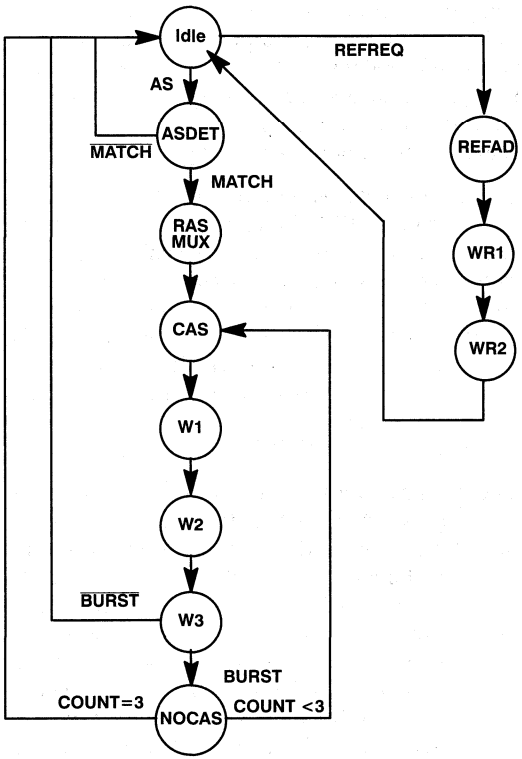


Figure 5. Controller State Machine State Diagram

The state machine controls all actions of the DRAM controller including the timing for the DRAM access. The state diagram is given in *Figure 5* and the VHDL code is given in *Figure 6*. The state machine is designed so that the outputs come directly from a dedicated flip-flop, principally so that the bus acknowledge back to the system has a favorable set-up time for high-speed buses. Upon recognizing the assertion of the address flag from the address register, the state machine advances to the next state where the validity of the address is checked. The additional clock cycle from the receipt of the address flag to the test of the address validity allows time to test the address after it has entered the address register. If the upper address bits are 0, the access is taken to be valid for the DRAM memory and the state machine advances to start the memory cycle. If there is no address match, the state machine asserts the address flag clear signal to prepare for a subsequent transaction.

From the state machine *idle* state, the address multiplexer is set to select the upper portion of the stored address (the ROW address). With an address strobe detected, the state machine advances to the *asdet* state where the address match is examined. If there is no match, the state machine returns to the *idle* state and if there is a match, the state machine advances to the *rasa* state. Here the state machine asserts RAS and simultaneously causes the address multiplexer to select the lower portion of the stored address (the COLUMN address). In the next cycle, the state machine advances to the *casa* state where the DRAM CAS signal is asserted. The next cycles, w1, w2, and w3, are wait states keeping CAS and RAS asserted for the required time given in the DRAM data sheets. Depending upon the clock speed, w1 and/or w2 may be omitted. In state w3, a decision is made to return to the idle state if no burst is to be performed, or to advance to the state *nocas* if the transaction is a burst. In the *nocas* state, CAS is deasserted and then the state machine then loops back to the *casa* state to continue with the burst. The burst is ended with an examination of the burst counter in state w3. If the burst is completed, the state machine returns to the *idle* state. Refreshes are performed by advancing from the *idle* state to the *refad* state. Here, the address multiplexer is directed to output the address from the refresh address counter. In the next states, RAS is asserted for three cycles: *ref_ras*, *wr1*, and *wr2*. Refresh is completed, and the state machine returns to the idle state. In the IDLE state the code gives priority to refresh requests over bus related requests (address strobe detection).

Although this is a high-level behavioral design, some aspects of the hardware have been taken into account. The statement

```
ATTRIBUTE state_encoding OF state:type IS
    ONE_HOT_ONE;
```

directs the synthesizer to use one flip-flop per state for the state machine. A portion of the synthesized logic is shown in *Figure 7*. Note the simplicity of the resulting logic equations. Outputs and transition

```
-- State Machine process

    control:process
    begin
      wait until clk_in = '1';
      if(rs_in = '1') then
        state <= idle;
        cas_en <= '1';
        ras_en <= '1';
        back <= '1';
        col_sel <= '0';
        ref_sel <= '0';
      else
      case state is
        when idle =>
          cas_en <= '1';
          ras_en <= '1';
          back <= '1';
          if (ref_req = '1') then
            state <= refad;
            ref_sel <= '1';
          elsif (as_flag = '1') then
            state <= asdet;
            clr_as <= '1';
          end if;
        when asdet =>
          clr_as <= '0';
          if (match = '1') then
            state <= rasa;
            ras_en <= '0';
            col_sel <= '1';
            burst_flag <= burst_stored;
          else
            state <= idle;
          end if;
        when rasa =>
            state <= casa;
            cas_en <= '0';
            --back <= '0';

        when casa =>
            state <= w1;
            --back <= '1';

        when w1 =>
            state <= w2;
```

Figure 6. State Machine Behavioral Description

```
            when w2 =>
                state <= w3;
                back <= '0';

            when w3 =>
              cas_en <= '1';
              back <= '1';
              if(burst_flag = '1' and bst_cnt /= "11") then
                state <= nocas;
                col_sel <= '1';
              else
                state <= idle;
                ras_en <= '1';
                col_sel <= '0';
              end if;

            when nocas =>
                state <= casa;
                cas_en <= '0';
    -- Refresh
            when refad =>
                state <= wr1;
                ras_en <= '0';

            when wr1 =>
                state <= wr2;
            when wr2 =>
                state <= idle;
                ref_sel <= '0';
                ras_en <= '1';

          end case;
          end if;
        end process;
```

Figure 6. State Machine Behavioral Description (continued)

terms include only the one flip-flop for the state in question. This results in a great simplification of the logic since complex state decoding is not required. Although this approach uses more flip-flops than encoded states, the decoding is far simpler. Most states and the related decoding should be realizable in one logic cell.

```
controller_0_state_bv_4_.D =
   controller_0_state_bv_3_.Q
 + controller_0_state_bv_8_.Q * burst_flag.Q * /bst_cnt_0__BEH_i42_0_DFF.Q
 + controller_0_state_bv_8_.Q * burst_flag.Q * /bst_cnt_1__BEH_i42_0_DFF.Q
```

Figure 7. A Portion of the Synthesized State Machine Logic Equations (State CAS)

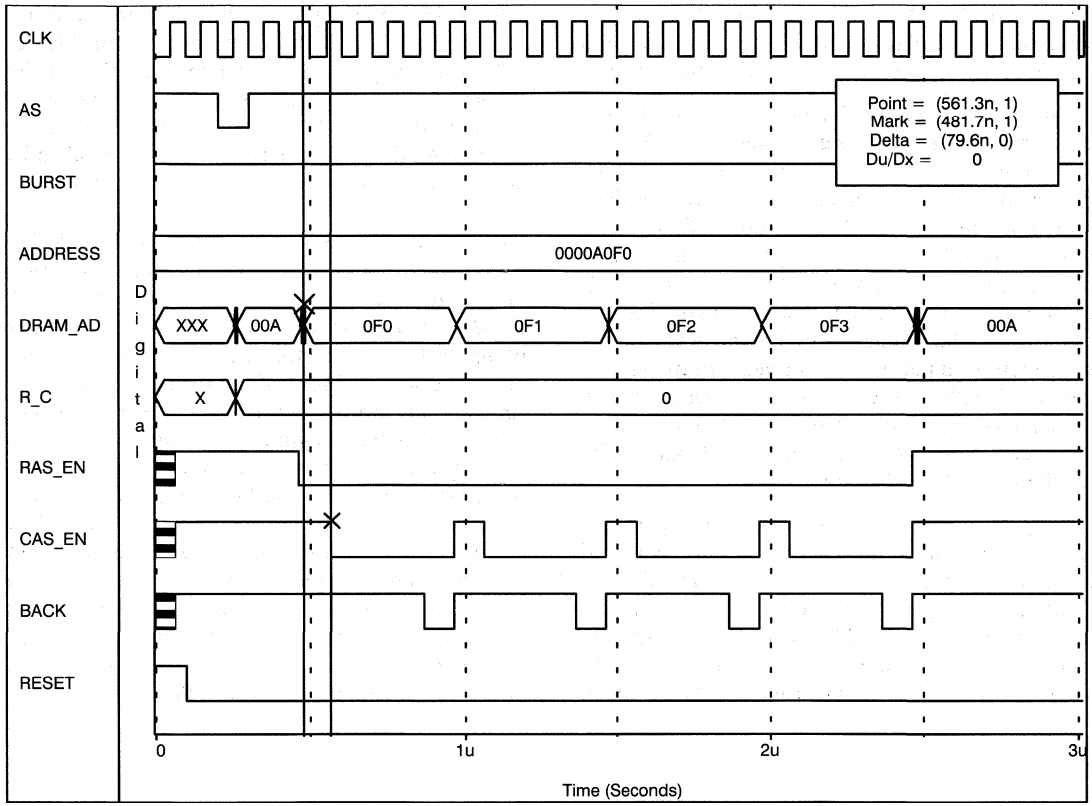


Figure 8. Graphic Output of the Simulator

The state machine, by virtue of the VHDL coding, forces the outputs to be derived directly from the output of a flip-flop. This gives signals such as BACK (bus acknowledge) a short clock-to-output propagation delay, allowing it to meet the requirements of a high speed bus.

Design Analysis

The final step in the design process is to analyze the device performance and make adjustments to the design to optimize certain speed paths. At this point, the functionality of the device should be correct per the design specification as verified by the VHDL source level debugger. The performance optimization is accomplished using the timing analyzer. Timing measurements can be made in the simulation as well. There are several concerns in this design: the set-up time for the address and control information to the input register, the clock-to-output delay for the BACK signal, and the delay in the column address output. This latter concern is to determine if the multiplexing from the row to the column address (in state *rasa*) will cause the column address to be too close to the assertion of CAS (state *casa*) potentially violating the DRAM column address set up time. *Figure 8* illustrates a timing measurement being made in the simulation environment. The figure shows one burst transaction. A Mark and Point are placed at the falling edge of CAS_EN (the CAS enable output) and the place where the DRAM address (DRAM_AD) has

switched to the column address and stabilized. The small window displays the Mark and Point times and DELTA is the time difference between these points.

As an example of an optimization the clock to output of the BACK signal is examined. *Table 6* shows the output of the Path Analyzer. The table shows a timing analysis result selecting the flip-flop to PAD options in the analyzer. The flip-flop to BACK path is then selected in the table of signal delays. The tabular results show a delay of 6.3 ns from the BACK flip-flop output to the pad. The schematic-like representation (the Physical View) of the device shows the generalized placement and signal routing in the device. The Physical View is shown in *Figure 9*. The selection of the flip-flop output to PAD path in the analyzer results table has caused that path to be highlighted in the schematic.

An improvement in this delay is sought by placing a timing constraint in the analyzer results table and rerunning the place and route. *Table 2* shows the new result after the place and route is rerun. The new place and route Physical View results are shown in *Figure 10*. Comparing *Figures 9* and *10* it can be observed that the flip-flop storing BACK has been moved closer to the pad resulting in a near 1 ns improvement in the clock to output delay for this signal. The analyzer shows that the variation in the clock pad to flip-flop delay is less than 100 ps, therefore it is not fruitful to attempt to improve the clock pad to BACK flip-flop delay. The other signal paths are similarly analyzed and optimized if necessary.

After the optimization is completed, the final design is simulated. There are two ways to define the stimulation for the simulation: graphical and textual. For designs with wide bus inputs or a large number of inputs, the most convenient method is textual input. The textual input of commands to drive the simulation is shown in *Figure 11*. Note in particular the command lines beginning with wfm. These lines describe the waveform of the input signals. Of particular interest is the line beginning with wfm address, which describes the input address signal. The entire vector, address, can be assigned values in hex—a task which would be very cumbersome in a graphic input only environment. This has been done for the simulation result shown in *Figure 8*. The figure shows the graphical results of the simulation output along with the input stimuli.

Table 6. SpDE Path Analyzer

Path #	Delay	Delay Path	Constraint
−1−	4.0	RAS_CAS_3_−I2 −− RAS_CAS_3_	
−2−	4.0	RAS_CAS_0_−I2 −− RAS_CAS_0	
−3−	4.0	RAS_CAS_1_−I2 −− RAS_CAS_1	
−4−	4.3	RAS_CAS_2_−I2 −− RAS_CAS_2	
−5−	5.7	RAS_EN−I2 −− RAS_EN	
−6−	6.0	CAS_EN−I2 −− CAS_EN	
−7−	6.3	BACK−I2 −− BACK	
−8−	6.8	STORED_AD_19_ −− RC_AD_7_	
−9−	7.0	STORED_AD_16_ −− RC_AD_4_	
−10−	7.0	STORED_AD_23_ −− RC_AD_11_	
−11−	7.0	STORED_AD_21_ −− RC_AD_9_	
−12−	7.1	STORED_AD_15_ −− RC_AD_3_	
−13−	7.2	STORED_AD_13_ −− RC_AD_1_	
−14−	7.4	STORED_AD_14_ −− RC_AD_2_	
−15−	7.4	STORED_AD_18_ −− RC_AD_6_	
−16−	7.5	STORED_AD_20_ −− RC_AD_8_	
−17−	7.8	STORED_AD_12_ −− RC_AD_0_	
−18−	7.8	STORED_AD_22_ −− RC_AD_10_	
−19−	8.0	STORED_AD_17_ −− RC_AD_5_	
−20−	8.7	STORED_AD_19_ −− RC_AD_7_	
−21−	8.8	STORED_AD_23_ −− RC_AD_11_	
−22−	8.8	STORED_AD_16_ −− RC_AD_4_	
−23−	8.8	STORED_AD_21_ −− RC_AD_9_	
−24−	8.9	COL_AD_9_ −− RC_AD_9_	

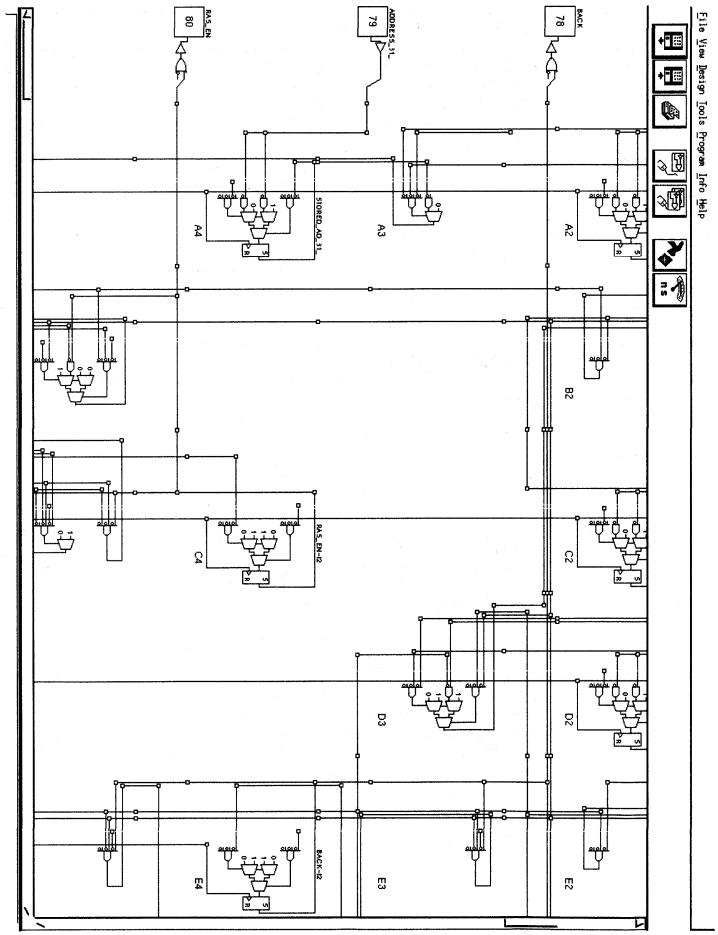


Figure 9. Physical View of Initial Design

Table 7. SpDE Path Analyzer with Applied Constraint

Path #	Delay	Delay Path	Constraint
−1−	4.0	RAS_CAS_3_−I2 −− RAS_CAS_3_	
−2−	4.0	RAS_CAS_0_−I2 −− RAS_CAS_0	
−3−	4.0	RAS_CAS_1_−I2 −− RAS_CAS_1	
−4−	4.3	RAS_CAS_2_−I2 −− RAS_CAS_2	
−5−	5.5	BACK−I2 −− BACK	4.0
−6−	5.7	RAS_EN−I2 −− RAS_EN	
−7−	6.0	CAS_EN−I2 −− CAS_EN	
−8−	6.8	STORED_AD_19_ −− RC_AD_7_	
−9−	7.0	STORED_AD_16_ −− RC_AD_4_	
−10−	7.0	STORED_AD_23_ −− RC_AD_11_	
−11−	7.0	STORED_AD_21_ −− RC_AD_9_	
−12−	7.1	STORED_AD_15_ −− RC_AD_3_	
−13−	7.2	STORED_AD_13_ −− RC_AD_1_	
−14−	7.4	STORED_AD_14_ −− RC_AD_2_	
−15−	7.4	STORED_AD_18_ −− RC_AD_6_	
−16−	7.5	STORED_AD_20_ −− RC_AD_8_	
−17−	7.8	STORED_AD_12_ −− RC_AD_0_	
−18−	7.8	STORED_AD_22_ −− RC_AD_10_	
−19−	8.0	STORED_AD_17_ −− RC_AD_5_	
−20−	8.7	STORED_AD_19_ −− RC_AD_7_	
−21−	8.8	STORED_AD_23_ −− RC_AD_11_	
−22−	8.8	STORED_AD_16_ −− RC_AD_4_	
−23−	8.8	STORED_AD_21_ −− RC_AD_9_	
−24−	8.9	COL_AD_9_ −− RC_AD_9_	

Summary

This application note is an introduction to FPGAs and high-level design tools. Specifics of the Cypress 380 FPGA family of devices were presented along with the *Warp3* design tool set. A design example was presented to illustrate the global flow of a complete development. This was not meant to be a design tutorial for the design tools, schematic entry, VHDL encoding or design optimization, but rather it is intended as an overall map as to how a designer would proceed and the options available. The 380 family FPGAs and the *Warp3* tool set is a powerful combination of device architecture and development tool that can help a designer achieve design success in a very short period of time.

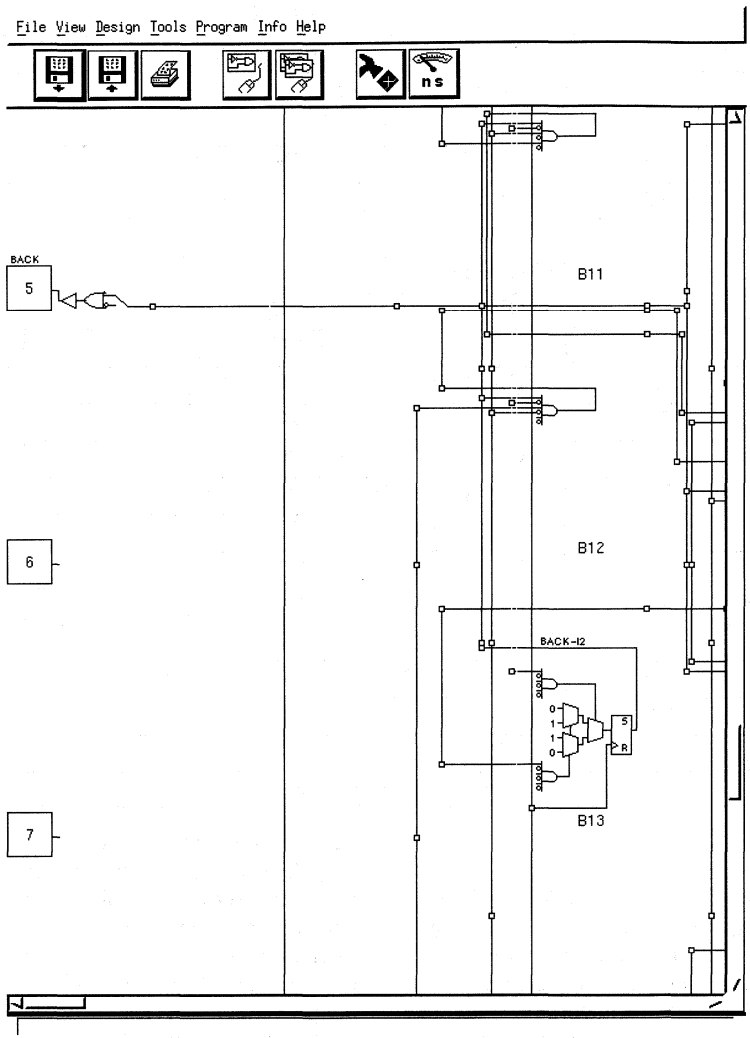


Figure 10. Physical View of Design Re−Placed and Routed with Constraint

```
| controller command file
logfile controller.log
stepsize 50ns
defaults -bignet -cmdfile -time
wave
vector address address[31:0]
radix hex address

vector dram_ad RC_AD[11:0]
radix hex dram_ad

vector r_c RAS_CAS[3:0]
radix hex r_c

wave controller.wfm clk as burst address dram_ad r_c ras_en cas_en back
       reset

clock clk 0 1

wfm reset @0=1 100ns=0
wfm as @0=1 200ns=0 100ns=1
wfm burst @0=1
wfm address @0=a0f0\h
cycle 200
log
```

Figure 11. Simulation Command File

Appendix A. Complete Design Behavioral Description

```
entity controller is
  port(clk,burst,as,reset: in bit;
    address: in bit_vector(31 downto 0);
    back,ras_en,cas_en: out bit;
    ras_cas: out bit_vector(3 downto 0);
    rc_ad: out bit_vector(11 downto 0));
attribute part_name of controller:entity is "C384";
end controller;

use work.bv_math.all;
use work.rtlpkg.all;

architecture behavior of controller is

  type name is (idle,asdet,rasa,casa,w1,w2,w3,nocas,refad,wr1,wr2);
  ATTRIBUTE state_encoding OF name: type IS ONE_HOT_ONE

  signal state: name;
  signal bst_cnt: bit_vector(1 downto 0);
  signal col_sel, ref_sel, as_flag, burst_flag, burst_stored, match,
      clr_as, ref_req, clk_in, ck_x, ck_y, as_x, as_in, rs_x, rs_in: bit;
  signal stored_ad: bit_vector(31 downto 0);
  signal re_ad, col_ad: bit_vector(11 downto 0);

  alias top_ad: bit_vector(3 downto 0) is stored_ad(31 downto 28);
  alias row_ad: bit_vector(11 downto 0) is stored_ad(23 downto 12);
  alias bank: bit_vector(3 downto 0) is stored_ad(27 downto 24);

begin

-- Special IO ports for device

  ck1:PAckcell
    PORT MAP(clk, ck_x, ck_y, clk_in);

  hd1:PAincell
    PORT MAP(as, as_x, as_in);

  hd2:PAincell
    PORT MAP(reset, rs_x, rs_in);

-- Address register process

  adreg:process
  begin
    wait until clk_in = '1';
```

Appendix A. Complete Design Behavioral Description (continued)

```
    if(rs_in = '1') then
      as_flag <= '0';
    elsif(as_in = '0') then
      as_flag <= '1';
    elsif(clr_as = '1') then
      as_flag <= '0';
    end if;

    if(as_in = '0') then
      stored_ad <= address;
      burst_stored <= burst;
    end if;

  end process;

-- Match Comparator

  match <= '1' when top_ad = "0000" else '0';

-- DRAM address multiplexer
  re_ad <= "000000000000";
  ref_req <= '0';

  mux:process(col_ad,col_sel,row_ad,re_ad,state)
  begin
    if(state = refad or state = wr1 or state = wr2) then
      rc_ad <= re_ad;
    elsif(col_sel = '1') then
      rc_ad <= col_ad;
    else
      rc_ad <= row_ad;
    end if;
  end process;

-- Encoded RAS / CAS select

  ras_cas <= bank;

-- Column Address, Intel Order

  col_ad(11 downto 2) <= stored_ad(11 downto 2);
  col_ad(1) <= stored_ad(1) xor bst_cnt(1);
  col_ad(0) <= stored_ad(0) xor bst_cnt(0);

-- State Machine process

  control:process
  begin
    wait until clk_in = '1';
```

Appendix A. Complete Design Behavioral Description (continued)

```
if(rs_in = '1') then
  state <= idle;
  cas_en <= '1';
  ras_en <= '1';
  back <= '1';
  col_sel <= '0';
  ref_sel <= '0';
else
case state is
  when idle =>
    cas_en <= '1';
    ras_en <= '1';
    back <= '1';
    if (ref_req = '1') then
      state <= refad;
      ref_sel <= '1';
    elsif (as_flag = '1') then
      state <= asdet;
      clr_as <= '1';
    end if;
  when asdet =>
    clr_as <= '0';
    if (match = '1') then
      state <= rasa;
      ras_en <= '0';
      col_sel <= '1';
      burst_flag <= burst_stored;
    else
      state <= idle;
    end if;
  when rasa =>
      state <= casa;
      cas_en <= '0';
      --back <= '0';

  when casa =>
      state <= w1;
      --back <= '1';

  when w1 =>
      state <= w2;

  when w2 =>
      state <= w3;
      back <= '0';
```

Appendix A. Complete Design Behavioral Description (continued)

```
        when w3 =>
          cas_en <= '1';
          back <= '1';
          if(burst_flag = '1' and bst_cnt /= "11") then
            state <= nocas;
            col_sel <= '1';
          else
            state <= idle;
            ras_en <= '1';
            col_sel <= '0';
          end if;

        when nocas =>
            state <= casa;
            cas_en <= '0';
        when refad =>
            state <= wr1;
            ras_en <= '0';

        when wr1 =>
            state <= wr2;
        when wr2 =>
            state <= idle;
            ref_sel <= '0';
            ras_en <= '1';

      end case;
      end if;
    end process;

-- Burst counter

    burst_count:process
    begin
      wait until clk_in = '1';
      if(state = idle) then
        bst_cnt <= "00";
      elsif(state = w3) then
        bst_cnt <= inc_bv(bst_cnt);
      end if;
    end process;

end behavior;
```

Document #: PLD0002–0194
Warp is a trademark of Cypress Semiconductor Corporation.
pASIC is a trademark of QuickLogic Corporation.

CY7C380 Family Quick Power Calculator

This brief is intended to provide a rapid method of calculating the approximate power consumed by a CY7C380 family device. Because the intent is a first-estimate calculation, some details are neglected. The quiescent power of about 20 mW is not included. There is no estimate of the power for the number of columns and number of loads per column of the clock distribution tree. Wiring capacitance is neglected. High drive cell power is taken to be the same as a normal input cell. I/O cell power is averaged over input and output. The power calculation does not include the power of an output driving an external load. This approach was taken to simplify the calculation. The average toggle rate and per cent of the device used are assumed to be rough estimates, thus there is no need to strive for great accuracy. For detailed considerations refer to the application note "Power Characteristics of Cypress Products."

The equations used to create the curves are:

P(I/O) = (number of I/Os)*Fav*(0.3)

P(cells) = (% used)*F_{AV}*(0.38) for 7C381/C382 (*Figure 1*)

P(cells) = (% used)*F_{AV}*(0.77) for 7C383/C384 (*Figure 2*)

P(cells) = (% used)*F_{AV}*(1.54) for 7C385/C386 (*Figure 3*)

Where F_{AV} is the average toggle rate frequency

Quick Power Calculation Process

1. Estimate the toggle rate (frequency in MHz) for each of the major blocks of the design.
2. Select a 7C380 family device.
3. Estimate the percent of the device that will be utilized to implement each block.
4. For each block, use the power vs. toggle rate curves for the selected device and read the power for the estimated toggle rate and percent utilization. Enter the power in the work sheet.
5. Sum the individual powers for an estimate of the total power.

Work Sheet

Toggle Rate (MHz)	Percent of Device	Number of I/Os	Factor	Power (mW)
	X		0.3	
	X		0.3	
		X	X	
		X	X	
		X	X	
		X	X	
X	X	X	X	

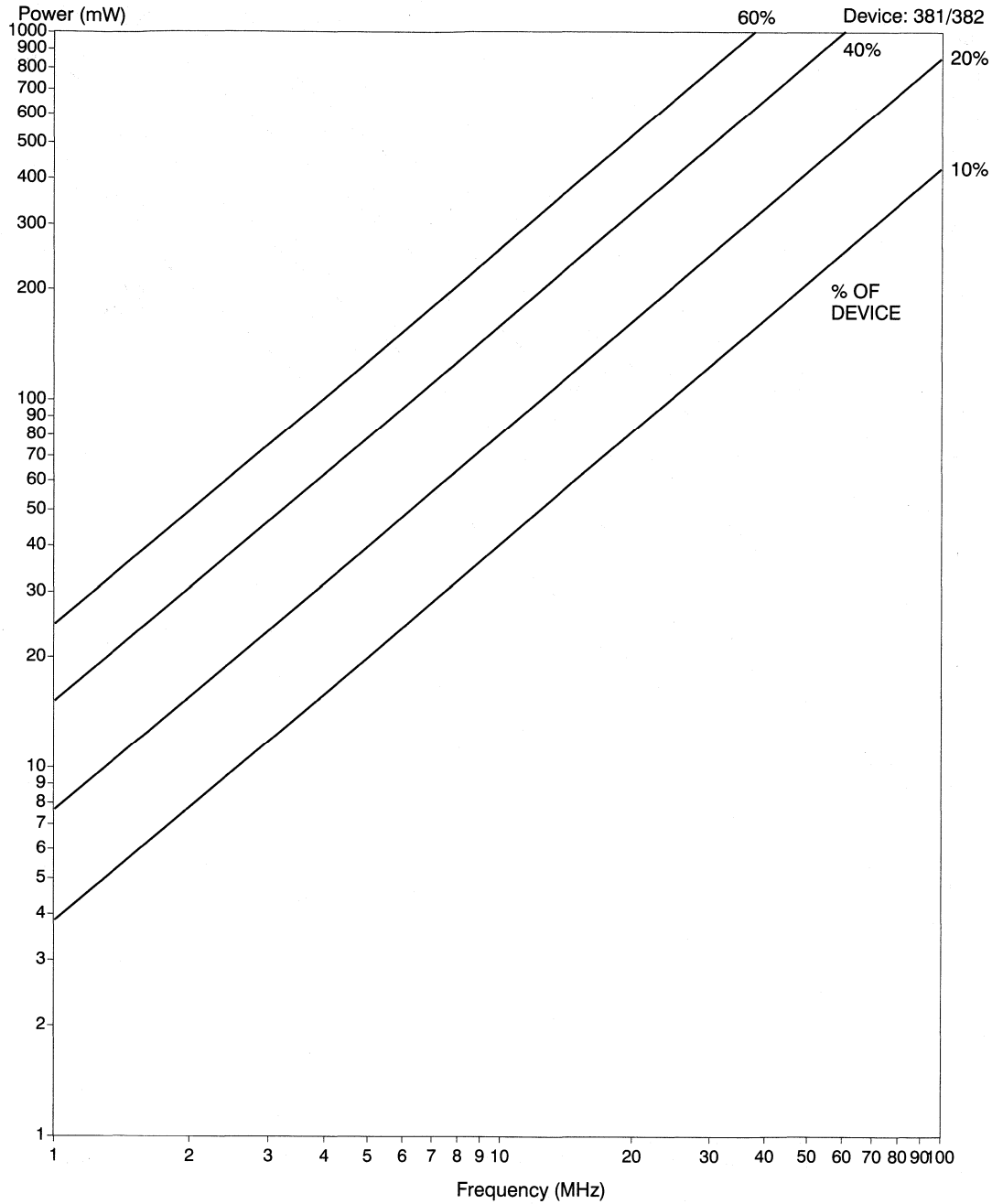


Figure 1. Average Toggle Range for CY7C381/2

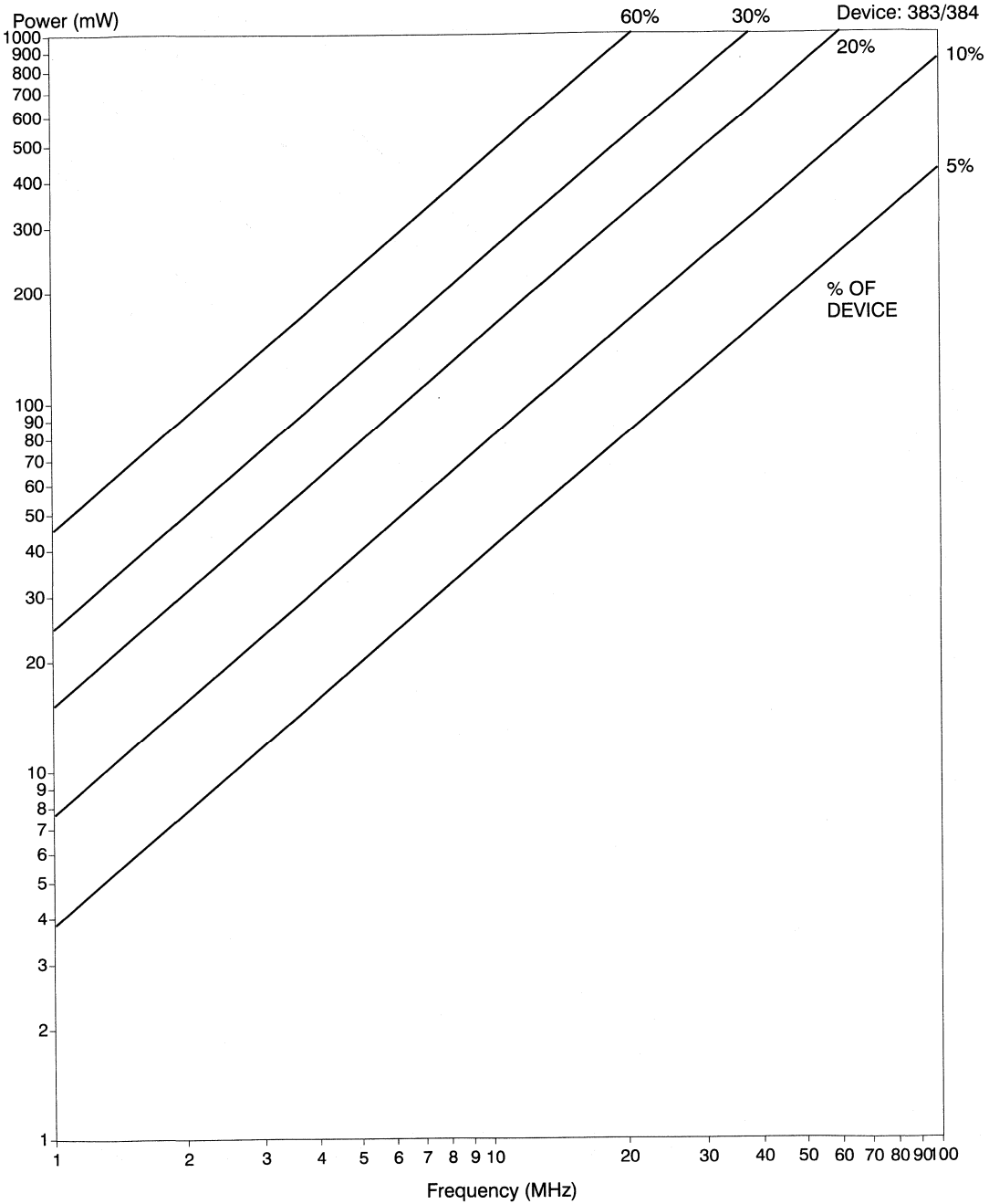


Figure 2. Average Toggle Range for CY7C383/4

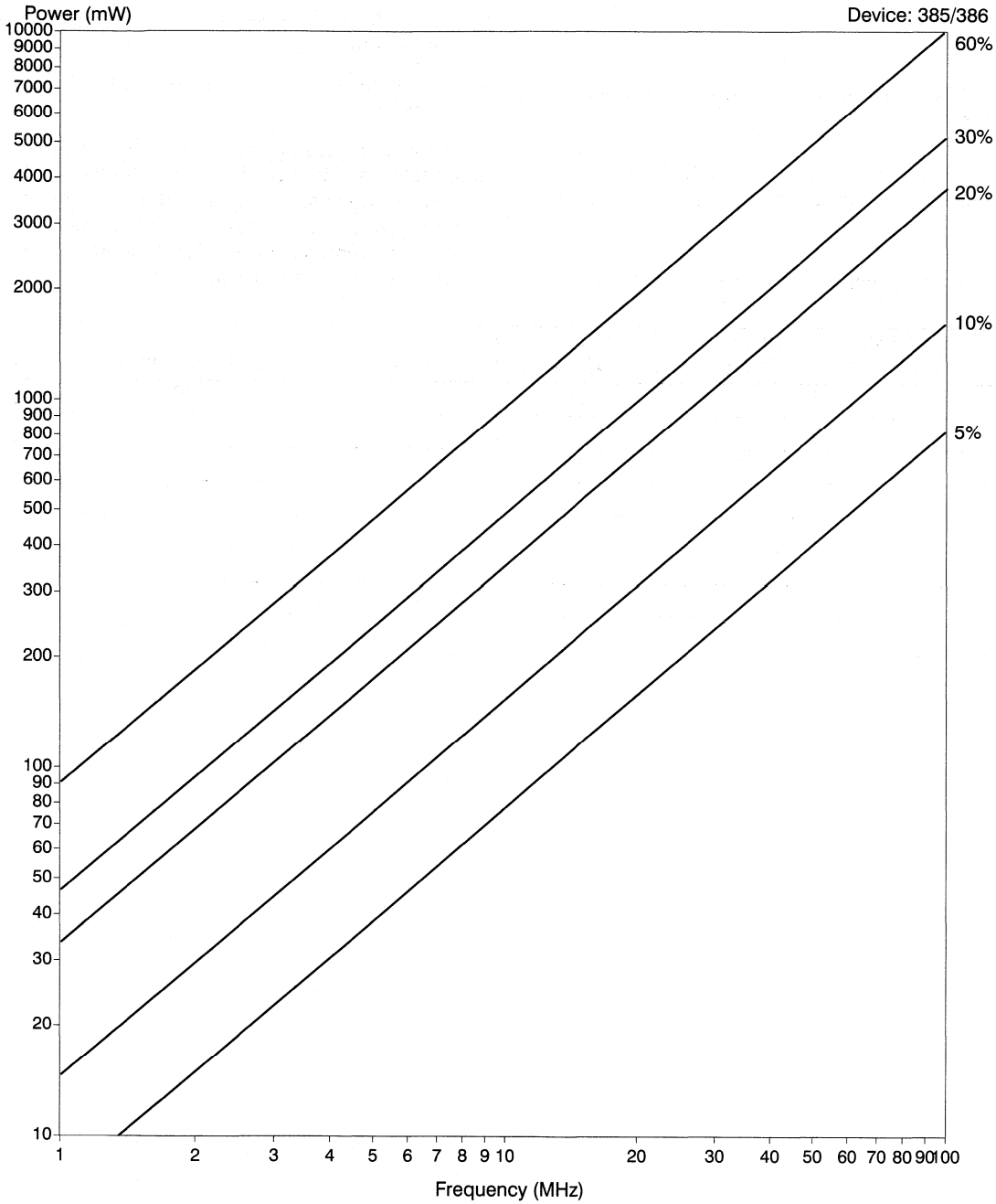


Figure 3. Average Toggle Range for CY7C385/6

Example

A CY7C382 FPGA is to be used with the following estimates:

- 32 I/Os are connected to a 40-MHz bus (half are assumed to be changing at this rate on the average).
- The remaining I/Os have a low duty cycle.
- 10% of the device is going to be toggling at the 40-MHz rate.
- 60% of the device is estimated to be toggling at 10 MHz.

The work sheet is filled in as shown below. The toggle rate of the I/Os is taken to be 20 because half are assumed to change in any clock (on the average). The next two entries are taken from the graph for the 7C382 and entered into the Power column. Total power is summed at the bottom.

Toggle Rate (MHz)	Percent of Device	Number of I/Os	Factor	Power (mW)
20	X	32	0.3	192
	X		0.3	
40	10	X	X	150
10	60	X	X	220
		X	X	
		X	X	
X	X	X	X	562

Document #: PLD0009–0194

Using Scan Mode on pASIC380 For In-Circuit Testing

All pASIC380 family devices have a built-in serial scan path linking the logic cell register functions. This is provided to guarantee factory test coverage and to permit testing by the user following programming. Test vectors for use in this way are automatically generated from the *Warp3* design tool. The user can also use this scan path to test the device either in production or when the device is mounted in the application. This application note describes the use of the scan path.

In the unprogrammed device, all programmable inputs to internal gates are open, leaving these gates floating. Applying V_{CC} to the unprogrammed device can cause large currents to flow, which will cause irreparable damage to the device. Under no circumstances should the scan path procedure described herein be applied to an unprogrammed device.

The scan path is invoked by applying a super voltage to a specific pin. This connects the D input and Q output of the logic cell registers to the scan path and the clocks of the logic cell registers to the scan path clock. The scan path has three functional interface pins: SI, the input to the scan path; SO, the output of the scan path; and SCLK, the clock that shifts the scan path data from SI through the linked registers and out through SO. Data is shifted on the rising edge of the scan clock. Using the scan path, the present state of the registers can be read and a new state of the registers can be set. Invoking the scan path can be used in place of normal operation of the device so proper functioning can be verified. The scan path can be invoked first, if desired, to set the states of the internal registers. The scan path mode can be exited, the part run as in normal operation, and stopped. After stopping, the scan path mode can be invoked again to read the states of the internal registers. The state of any device outputs must be examined externally. The user must provide for disconnecting application connections from the pins that serve secondary purposes for SO, SI, SCLK, and V_{SP}. Then the scan path signals must be connected.

The pin information for the scan path is given in *Table 1*. For pin number translation to other package forms, refer to the device datasheet. Also included in this table is the number of clocks required to shift through the scan path. *Table 2* lists the electrical characteristics and *Figure 1* shows the timing for invoking and operating the scan path. Assuming V_{CC} is applied to the normal V_{CC} pins of the device, all user clocking is stopped and the scan path clock input pin is held stable at V_{IH} prior to invoking the scan path. The V_{SP} pin is then brought to the V_{SP} value as given in the electrical characteristics. Shifting of the scan path is done as shown in *Figure 1*. After V_{SP} is lowered to V_{CC} or below, normal operation of the device can commence.

The scan path can be related to the user's signals and flip-flop locations through the Physical View in the *Warp3* tool. In the place-and-route process, a view of the device layout is given and user signals are labeled in this view. Each logic cell is given a position designator consisting of a letter and a number as in a matrix. Letters designate the columns beginning with "A" on the left and numbers represent the rows beginning with "1" at the top. Locating a signal name on the output of a flip-flop in the physical view allows association of this signal with the logic cell at that location.

Table 1. Pin Definition and Number of Clocks for the Scan Path

Device	V_{SP} Pin	SI Pin	SO Pin	SCLK Pin	Clocks
C381–PLCC	11	32	36	10	96
C382–PLCC	17	50	54	16	96
C383–PLCC	17	50	54	16	192
C384–PLCC	22	63	66	21	192
C385–PLCC	22	63	66	21	384
C385–TQPF	12	61	65	11	384
C386–TQFP	18	89	93	17	384

Table 2. Scan Path Electrical Characteristics[1, 2]

Parameter	Symbol	Min.	Typ.	Max.	Unit
Supply Voltage[2]	V_{CC}	4.5	5.0	5.5	V
Scan Mode Super Voltage[2]	V_{SP}	11.75	12.0	12.35	V
Scan Clock Frequency ($t_C = 1/f_{SCLK}$)	f_{SCLK}			5.0	MHz
Data Set-Up to Scan Clock	t_S	100			ns
Data Hold After Scan Clock	t_H	100			ns
Data Valid After Scan Clock	t_{CD}			100	ns
V_{SP} Ramp Rate	t_r	1.0		100	µs

The scan path input is at the cell in the upper right of the device. This cell has different designations depending upon the device: H1 for the CY7C381 and CY7C382, L1 for the CY7C383 and CY7C384, and P1 for the CY7C385 and CY7C386. Shifting moves form SI (scan input) to this cell and down the right side column. At the bottom of this column, the scan moves up the next column on the immediate left. The scan follows this pattern moving down one column and up the next one on the immediate left. The last logic–cell in the scan path is cell "A1" whose output appears on the pin SO (scan out).

Table 3 shows the last column and last row for the C380 family devices. This information can be used to locate the logic cell register value that appears in a particular position in the scan–in or scan–out serial bit stream.

As an example, the scan path for the CY7C384 begins at logic cell L1, shifts toward L16; L16 shifts into K16 and the shift continues up column K. K1 shifts into J1. Lastly, A1 is presented at the output, SO.

This is illustrated in *Figure 2*. Using this association, the desired flip-flop state can be located in the serial output stream. All flip-flops are shifted out (or in to) whether or not they are employed in the user's design.

Table 3. Scan Path Row/Column Range

Device	Last Column	Last Row
CY7C381	H	12
CY7C382	H	12
CY7C383	L	16
CY7C384	L	16
CY7C385	P	24
CY7C386	P	24

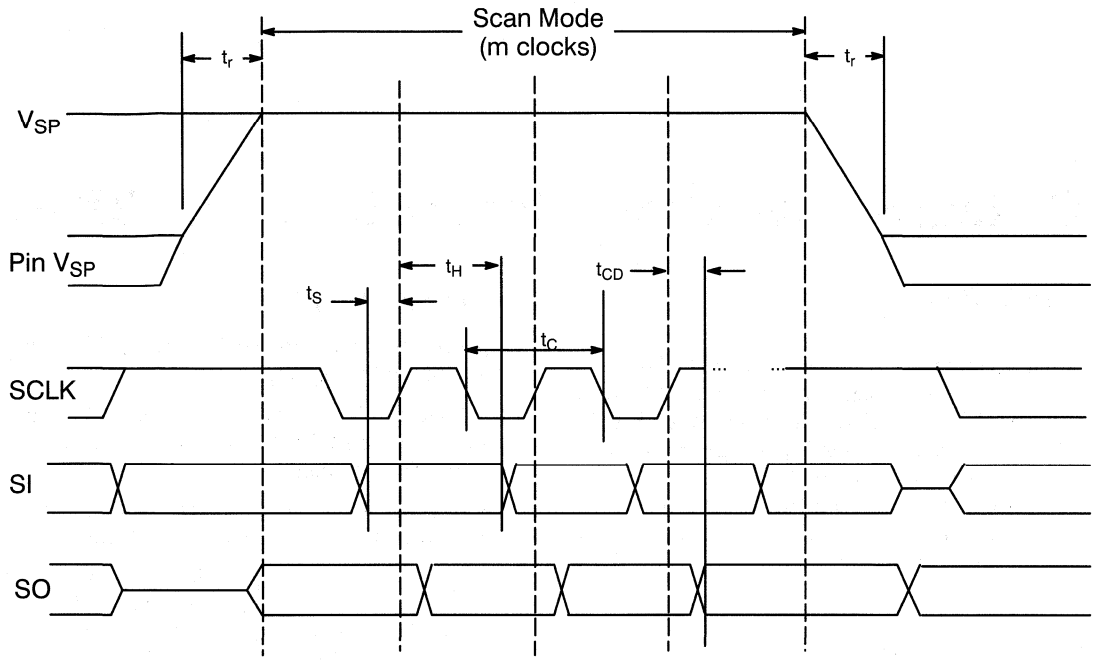


Figure 1. Scan Path Timing Diagram

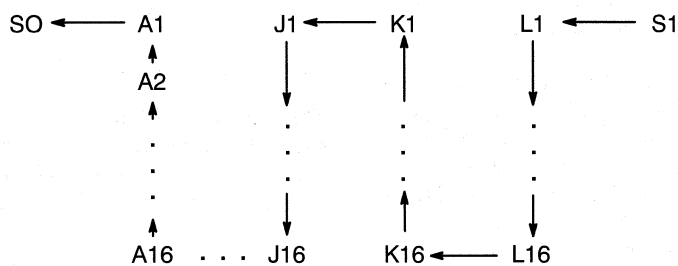


Figure 2. Scan Path Example for pASIC384

Notes

1. V_{IH}, V_{IL}, V_{OH}, and V_{OL} are the same as in the device datasheet.
2. Application of these voltages to an unprogrammed device can cause irreparable damage.

Document #: PLD0011–0394

Getting Started Converting .ABL Files to VHDL

Introduction

This application note is intended to assist *Warp*™ users in converting designs written in DATA I/O's ABEL hardware description language to IEEE 1076 VHDL. It contains several language cross reference tables and many helpful hints. It also includes two real-world designs that have been converted from MACH™ 210–ABEL descriptions to FLASH371–VHDL descriptions.

VHDL versus ABEL

VHDL is different from ABEL and virtually all other popular hardware description languages in one very significant way. It is an open language. It is based on IEEE standard number 1076.

VHDL is different in other ways, too. VHDL is a high-level language. As such, it is much more powerful than ABEL. For instance, it supports hierarchical design entry, structural (low-level instantiation of components) and behavioral (IF-THEN-ELSE) design entry. VHDL supports process and concurrent process statements. It also supports various types of signals such as integer, real, character, bit, Boolean, physical unit, and any others that a user can define. It supports sequential and concurrent statements, variables, and signals. VHDL supports sub-programs, FOR loops, WHILE loops, arrays, concurrent procedure calls, and more.

Surprisingly, certain aspects of VHDL related to low-level behavioral logic description are very similar to ABEL. In fact, some key words and relational operators are identical or logically similar.

Conversion Preparation

Preparing to convert an ABEL (.ABL) file should include the following steps:

1. Locate and have at hand one good VHDL language reference book. See the *Warp* documentation for a bibliography.
2. Obtain copies of the *Warp* VHDL design examples titled Basic, Intermediate, and Advanced.
3. Locate two Cypress application notes, one titled "Designing State Machines with *Warp2*™ VHDL" and another titled "VHDL Techniques for Optimal Design Fitting." Both are contained in the *Cypress Applications Handbook* (1993).
4. Install the *Warp* VHDL compiler on your hardware platform.

Conversion Approach

There are many different ways to convert a given design. The same design may be expressed in a number of different ways, all yielding compiled designs with the same functionality. The general approach suggested for converting ABEL (.ABL) files to VHDL (.VHD) consists of five basic steps:

1. Analyze the design and determine:
   a. Which signals are registered and which are not. Group them into two categories.
   b. Which types of design entry the .ABL file includes: state machines, comparators, counters, decoders, multiplexers, adders, multipliers, shift registers, or state_tables.
   c. Whether or not group (set) declarations are used.

   d. Which signals are input, output, I/O, and/or active LOW.

2. Replace as many of the keywords and operators with the corresponding VHDL keywords and operators using your favorite SEARCH and REPLACE text editor and a backup copy of the .ABL or .DOC file.

3. Add the VHDL entity (black box inputs and outputs), architecture (description of the logic circuitry), and process (encapsulates a set of sequential-behavioral functions) statements.

4. Add the proper library USE statements to the file such as USE WORK.CYPRESS.ALL.

5. Iteratively compile the design revising incomplete or incorrectly converted syntax.

Some designs will be much easier to convert than others. The more regular the design the easier it will be to convert. The most efficient and highest level of conversion will be achieved by using the source (.ABL) file, the five steps above, and the cross reference information below.

The simpler approach is to use the .DOC file exclusively. Using the .DOC file works but does have one significant drawback. The .DOC file tends to be verbose. It is verbose because it describes the design at a low level. A converted ABEL (.DOC) design thus results in unnecessarily verbose VHDL. In other words, it results in inefficient code.

When converting using the .DOC file, place all of the registered signals into a process with a WAIT UNTIL CLOCK = '1' and place all of the combinatorial signals outside the process. This avoids the necessity of PROCESS sensitivity lists and IF-THEN-ELSE statements. The converted designs below and all of the *Warp* example designs attempt to describe functions behaviorally and at a higher level. For this reason no low-level design conversion examples are included.

Refer to the following sections and tables for helpful information when converting ABEL .ABL and .DOC descriptions.

Comments

Comments in ABEL are denoted by the quote symbol ("). Comments in VHDL are denoted by two consecutive dashes −−. For example:

ABEL	VHDL
"Inputs	−−Inputs
"Outputs	−−Outputs

VHDL-ABEL Special Constant Cross Reference

ABEL	VHDL	Description
.C.	requires user definition	Clocked input (0 −>1−> 0)
.D.	requires user definition	Clock down edge (1 −> 0)
.F.	requires user definition	Floating input or output
.K.	requires user definition	Clocked input (1−> 0−> 1)
.P.	requires user definition	Register preload
.SVn.	requires user definition	Super voltage ($2 \le n \le 9$)
.U.	requires user definition	Clock up edge (0 − > 1)
.X.	requires user definition	Don't care condition
.Z.	requires user definition	High impedance

In VHDL a constant is an object whose value may not be changed. The syntax for declaring a constant in VHDL is:

```
constant identifier_list : type [ :=expression];
```

An example of this is:

```
TYPE stvar is bit_vector (0 to 1);
constant State0 : stvar  := "00";
```

This example declares a constant that is identified by the name State0, is of type stvar, which has been previously defined as a bit_vector subtype of length 2. This constant is given the value 00. Defining a constant of user-defined type called state variable (stvar) is useful when designing state machines in VHDL. See the State Machine section of this application note.

Special constants in ABEL are used for simulation vectors that are included in the source file (.ABL). *Warp* does not provide simulation support directly within the source file. So, the conversion recommendation for files containing simulation vectors is to delete or comment them out of the .VHD file. *Warp* provides simulation separately from the design file (.VHD). Simulation can take one of two forms. The first, functional waveform based design verification using NOVA. Second, full AC timing verification via VIEWSIM and VIEWTRACE. VIEWSIM and VIEWTRACE support both pattern files and waveforms. Both forms of *Warp* VHDL simulation exceed the capabilities of ABEL simulation.

VHDL-ABEL Dot Extension Cross Reference

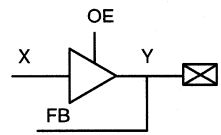
ABEL	VHDL	Function
.CLK	none	Clock input to flip-flop
.PIN	none	Pin feedback
.FB	none	Register feedback
.D	none	D-type flip-flop
.J	none	J input to JK flip-flop
.K	none	K input to JK flip-flop
.S	none	S input to SR flip-flop
.R	none	R input to SR flip-flop
.T	none	T-type flip-flop
.Q	none	Register feedback
.PR	none	Register preset
.RE	none	Register reset
.SP	none	Synchronous reg preset
.SR	none	Synchronous reg reset
.LE	none	Latch-enable input
.LH	none	Latch-enable input (HIGH)
.LD	none	Register load input
.CE	none	Clock enable input
.AP	none	Asynchronous preset
.AR	none	Asynchronous reset
.OE	none	Three-state output enable
.CLR	none	Synchronous clear
.ACLR	none	Asynchronous clear

VHDL-ABEL Dot Extension Cross Reference (continued)

ABEL	VHDL	Function
.SET	none	Synchronous set
.ASET	none	Asynchronous clear
.COM	none	Combinational feedback
.FC	none	Flip-flop mode control

Although VHDL is capable of supporting these constructs, it directly does not. Indirectly, through behavioral description, structural description, and an intelligent compiler, all of these constructs are supported. For example, *Warp* does provide predefined register transfer level (RTL) components (such as D- and T-type flip-flops). These RTL components can be structurally instantiated to model the ABEL extensions listed above. Specifically, to model the .OE ABEL extension, use an RTL component called *bufoe*. The syntax and port map (inputs and outputs) of a *bufoe* component is the following:

Label: BUFOE PORT MAP(X, OE, Y, FB)



X, OE, Y, and FB are sample signal names. The position each one occupies in the port map is the mechanism VHDL uses to correctly connect the signal names to the actual component in the architecture of the target device.

The behavioral equivalent of structurally instantiating a bidirectional buffer is called a behavioral three-state. Behavioral three-states are presently not supported, but will be in the future.

If the ABEL description equation is written in .T (T-type) flip-flop style, the recommended conversion method is to rewrite the equations as D-type (XOR the original equation with the flip-flop output signal name) and let *Warp* optimize the equation for either D- or T-type. See the real-world design conversion example in Appendix A called FLAGCTLR.

IS_TYPE Attribute Cross Reference

ABEL	VHDL	Description
'buffer'	none, may use RTL – buf	Macrocell has no inverter between reg and pin
'com'	none, may use RTL – buf	Signal is combinatorial
'invert'	none, NOT RHS of equation	Macrocell has an inverter between reg and pin
'neg'	none, NOT signal in equation	Complement Sum of Products
'pos'	none	Do not Complement Sum of Products
'reg'	none, place sig in process	Generic flip-flop
'reg_D'	none, may use RTL – dff	D-type flip-flop
'reg_T'	none, may use RTL – tff	T-type flip-flop
'reg_SR'	none, may use RTL – srff	SR-type flip-flop
'reg_JK'	none, may use RTL – jkff	JK-type flip-flop
'reg_G'	none	D-flip-flop w/gated clock
'xor'	none, may use RTL – xbuf	Target architecture has XOR

Operator Cross Reference

ABEL	Order of Precedence	VHDL	Order of Precedence	Operation
!	1	NOT	Context dependent	NOT (invert)
&	2	AND	1	AND
*	2	*	5	Multiplication
/	2	/	5	Division
%	2	mod	5	Modulus
<<	2	none		Shift left
>>	2	none		Shift right
+	3	+	3	Arithmetic addition
−	3	−	3	Arithmetic subtr.
$	3	XOR	1	XOR
!$	3	NOT XOR		XNOR
#	3	OR	1	OR
==	4	=	2	Equal
!=	4	/=	2	Not equal
<	4	<	2	Less than
<=	4	<=	2	Less than or equal
>	4	>	2	Greater than
>=	4	>=	2	Greater or equal
none		NAND	1	NAND
none		NOR	1	NOR
none		&	3	Concatenation
none		rem	5	Remainder
none		abs	6	Absolute Value
none		**	6	Exponentiation
?		+	4	Sign
?		−	4	Sign
= or :=		<=		Signal assignment
none		:=		Variable assignment
=		<=		Comb. assignment
:=		<=		Reg. assignment
none		=>		Association

The only ABEL operators without a direct VHDL counterpart are the >> and <<, (shift right and shift left). To directly (structurally) describe logic that performs an N-bit shift left or right function see the *Warp* design example titled advanced SHIFTN.VHD. To emulate (behaviorally describe) a shift function in VHDL use bit_vectors or arrays and index the elements using LOOPs. Another technique is to use *2 and /2 (multiply by 2 and divide by 2).

Keyword (Statement) Cross Reference

ABEL Keyword	VHDL Equivalent
CASE	CASE
DECLARATIONS	Note 1
DEVICE	ATTRIBUTE PART_NAME IS ...
ELSE	ELSE
ENABLE (Obsolete)	none
ENDCASE	END CASE
ENDWITH	Note 2
EQUATIONS	Note 3
FLAG (Obsolete)	none
FUSES	Note 4
GOTO	EXIT – Note 5
IF	IF
IN (Obsolete)	none
ISTYPE	Note 6
LIBRARY	USE
MACRO	FUNCTION – Note 7
MODULE	FUNCTION – Note 8
NODE	SIGNAL
OPTIONS	Note 9
PIN	Note 10
PROPERTY	none
STATE	Note 11
STATE_DIAGRAM	Note 11
TEST_VECTORS	Note 12
THEN	THEN
TITLE	Note 13
TRACE	Note 14
TRUTH_TABLE	Note 15
WHEN	WHEN
WITH	Note 16
ASYNC_RESET	Note 17
SYNC_RESET	Note 17
STATE_REGISTER	Note 18
XOR_FACTORS	Note 19

VHDL does not use the term keyword. Analogous to ABEL's use of the term keyword, VHDL uses the terms statement, reserved word, and identifier.

Notes

1. There is not a DECLARATIONS keyword in VHDL. However, the DECLARATIONS keyword is analogous to declaring an ENTITY in VHDL. Within the ENTITY construct inputs, outputs, and I/Os are declared with appropriate mode and type. Mode loosely refers to the pin drive direction, which can be IN, OUT, or INOUT. Refer to your language reference book for a more formal definition of the terms mode, IN, OUT, and INOUT.
2. ENDWITH is part of the WITH-ENDWITH transition structure used with IF-THEN-ELSE or CASE keywords. In VHDL conditional transition is handled via an IF-THEN-ELSE or CASE statement within a PROCESS. The process statement may or may not use a sensitivity list and instead use a WAIT UNTIL (condition) statement. See the application note titled "Designing State Machines with *Warp2* VHDL."
3. Equations in VHDL are listed within an architecture statement.
4. VHDL and *Warp* do not provide predefined fuse-level program specification.
5. VHDL does not have a GOTO keyword (statement). It provides an EXIT keyword for stopping execution of loops entirely.
6. The IS_TYPE keyword (statement) defines attributes and/or characteristics of pins and nodes. VHDL provides these attributes through behavioral specification. Additionally, *Warp* provides a set of predefined attributes and VHDL provides a mechanism for declaring new attributes. See the attribute table below.
7. VHDL provides function call and return capability. MACRO is more of a low-level substitution technique such that, wherever the MACRO_id occurs, the text associated with that macro will be substituted.
8. The MODULE ... END statement(s) are source file requirements in ABEL. In VHDL the ENTITY-ARCHITECTURE pair are the basic source file requirements. Both the ENTITY and ARCHITECTURE constructs require an END statement.
9. OPTION is a string of processing options that affect the way in which the ABEL source file is processed by the language processor. The analogous control in VHDL is not done in the source file. It in fact is not part of the VHDL language. It is simply a menu of compiler options that are set when using *Warp* to synthesize the design.
10. PIN is used to declare input and output signals that must be available on a device I/O pin. The analogous PIN specification is implied in VHDL via the port map list in the ENTITY construct. All signal names listed in the entity port map are input, output, and I/Os of the entity.
11. See application note titled "Describing State Machines with *Warp*2 VHDL."
12. Test vectors are not directly supported by VHDL. However, both behavioral simulation and full AC timing simulation are available for design verification.
13. The TITLE statement is used to give an ABEL MODULE a title that will appear as a header in both the programmer load file (the JEDEC file) and the documentation file. When compiling a VHDL design using *Warp*, the filename of the VHDL (.VHD) design file is passed through to the programmer load file (.JED) as well as the documentation file (.RPT).
14. The TRACE statement controls the display features of ABEL's simulator. There is not a similar keyword in VHDL because simulation is separate from the source file description.
15. The TRUTH_TABLE keyword is used in ABEL to specify outputs as functions of different input combinations in a tabular form. VHDL does not directly provide a TRUTH_TABLE keyword. However,

in the common library (directory) included in *Warp*, there is a file called LIBSTATE.VHD that contains a FUNCTION called TTF. TTF is a predefined truth table function that can be used for both combinatorial truth tables and for state transition tables. See the application note titled "Describing State Machines in *Warp*2 VHDL" regarding use of the TTF function.

16. WITH is part of the WITH-ENDWITH transition structure used with IF-THEN-ELSE or CASE statements. In VHDL, conditional transition is handled via an IF-THEN-ELSE or CASE statement within a PROCESS. The process statement may use a sensitivity list or may include a WAIT UNTIL (clock = '1') statement.
17. ASYNC_RESET and SYNC_RESET statements are used in Symbolic State descriptions. They symbolically specify what state the machine should asynchronously or synchronously reset to, based upon a signal or an expression. In VHDL, asynchronous and synchronous resets are best handled from a behavioral perspective the Resets and Presets section of this note for more detail.
18. STATE_REGISTER is a mechanism whereby specific states of a machine can be identified symbolically. See the State Machine section of this note for more detail.
19. XOR_FACTORS is a keyword that is useful for factoring logic designs that target a device which features XOR gates. There is not an analoguos keyword in VHDL. HOwever, the functional aspect of this keyword is part of the *Warp* Compiler Option menu. For more details see the *Warp* Compiler Options Documentation.

Predefined Attributes Supported by *Warp*

Value Attributes	'Left 'Right 'High 'Low 'Length
Function Attributes (types)	'Pos 'Val 'Leftof 'Succ 'Rightof 'Pred
Function Attributes (objects)	'Left 'Right 'High 'Low 'Length
Function Attributes (signals)	'Event
Type Attributes	'Base
Range Attributes	'Range 'Reverse_range

Other user-defined attributes include: Enum-encoding, Flip-flop-type, Order_code, Part_name, Pin_numbers, Polarity, State_encoding, and State_Variable. See *Warp* documentation for details.

Number Representations

ABEL	VHDL	Radix
^b	b" " or " " or ' '(default)[20]	Binary
^o	o" "	Octal
^d (default)	Note 21	Decimal
^h	x" "	Hexadecimal

Notes

20. The default number representation in ABEL is decimal. The default number representation in VHDL is binary.

21. The default number representation in VHDL is binary. Decimal representations of numbers in VHDL require the user to define a signal or variable with type integer or use an integer number and then type-convert it to bit_vector. This is easier than it sounds. In the common library directory within *Warp* there is a file called LIBINT.VHD that contains a predefined function called i2bv. This function takes an integer and returns a bit_vector. So, using a decimal number is not too difficult, but one must know that an integer must be used and then type-converted to bit_vector.

For example:

ABEL syntax	VHDL syntax	Description
^b1	'1'	binary 1
^b0	'0'	binary 0
^b101010000	"10101000"	binary 10101000
^hF	x"F"	hex F
^hF1	x"F1"	hex F1
^hAAA	x"AAA"	hex AAA
^oF0F0	o"F0F0"	octal F0F0
^d23	i2bv(23,5)	decimal 23
^d99	i2bv(99,7)	decimal 99

Polarity Conventions

VHDL does not know whether a signal name should be interpreted as an active HIGH or an active LOW. Therefore, a signal named SHIFT4 will be interpreted logically the same as one named L_SHIFT4, and as one named SHIFT4_NOT. In other words, the behavioral equations must test with the proper level and assert with the desired level.

During logic synthesis and optimization, the software may determine that by flipping the polarity of a function the logic required will be optimized.

Identifiers

VHDL is not case sensitive, so a signal named SHIFT0 is identical to one named sHIFT0.

Resets and Presets

Although there are a variety of ways to specify a reset or preset, the best method is behavioral specification. If the reset or preset is asynchronous, use the following:

Place an IF-THEN-ELSIF-ENDIF inside a process with a (CLK'EVENT and CLK='1') placed as the condition for the ELSIF. In the first IF, place your reset and preset condition test and your signal assignments. In other words, the first part of the IF contains the asynchronous or combinatorial logic description and the second part, the ELSIF, contains the clocked logic description. In the process statement use a sensitivity list that includes the clock, and reset/preset for the design. Don't forget that statements in a process are considered sequential and are only updated upon changes in signals listed in the sensitivity list. See the basic example called COUNTER2.VHD and the real-world converted design example called FLAGCTLR below.

If the reset or preset is synchronous, place the condition inside the clocked portion of the IF-THEN-ELSIF-ENDIF mentioned above and perform the appropriate signal assignments.

This methodology ensures that behavioral operation is preserved and no device specific attributes are required.

Groups

ABEL allows declaration of groups or sets. Sets are groupings of signals. For example a bus is a set of signals. To create a set of signals in VHDL use the bit_vector type declaration. To perform Boolean operations on these new sets use IF-THEN-ELSE and FOR LOOPs to index the individual elements. See the special type conversion function and the real-world examples below.

Special VHDL Type Conversion Function (Advanced)

VHDL is a strongly typed language. ABEL on the other hand is not a strongly typed language. ABEL allows a user to mix Boolean operations with relational operations on sets. To concisely convert ABEL equations that contain relational operations on sets (converted to VHDL type BIT_VECTOR) combined with Boolean operations on signals (converted to VHDL type BIT), use the following type-conversion function. All equations requiring this type-conversion function call can be modified easily with a SEARCH and REPLACE text editor.

```
------------------------------ cut here ------------------------------------

FUNCTION frbl_to_b (in1:Boolean)     RETURN bit IS
BEGIN
IF (in1=TRUE) THEN
    RETURN '1';
ELSE
    RETURN '0';END IF;
END frbl_to_b;
--  This type conversion function converts a signal or relational operation
--  result from type BOOLEAN to type BIT.  A Boolean can have a value of
--  either 'TRUE' or 'FALSE'.  A bit can have a value of either '0' or '1'.

------------------------------ cut here ------------------------------------
```

For example if you had an equation in ABEL such as:

```
ramwr = !addren & ba16 & !write & ((addr ==^h210)
    # (addr==^h212)
    # (addr==^h214)
    # (addr==^h216));
```

Where addr is a set of 16 address bits,

This equation could be converted to VHDL in at least two ways:

```
ramwr <= not addren and ba16 and not write and (fr_bl_to_b(addr =x"210")
     or fr_bl_to_b(addr=x"212")
     or fr_bl_to_b(addr=x"214")
     or fr_bl_to_b(addr=x"216"));
```

OR,

```
ramwr <= not addren and ba15 and not write and(
      (not addr(11) and not addr(10) and addr(9) and not addr(8) and not addr(7)
       and not addr(6) and not addr(5) and addr(4) and not addr(3) and not addr(2)
       not addr(1) and not addr(0))
   OR  (not addr(11) and not addr(10) and addr(9) and not addr(8) and not
       addr(7) and not addr(6) and not addr(5) and addr(4) and not addr(3) and not
       addr(2) and addr(1) and not addr(0))
   OR  (not addr(11) and not addr(10) and addr(9) and not addr(8) and not
      addr(7) and not addr(6) and not addr(5) and addr(4) and not addr(3) and
       addr(2) and not addr(1) and not addr(0))
   OR (not addr(11) and not addr(10) and addr(9) and not addr(8) and not addr(7)
      and not addr(6) and not addr(5) and addr(4) and not addr(3) and addr(2) and
      addr(1) and not addr(0)));
```

This example assumes all of the signals from the ABEL equations are converted to signals of type BIT except the set called 'addr', which is converted to type BIT_VECTOR.

This special type-conversion function has a obvious advantage and is well suited for use in converting descriptions to VHDL. By no means is it a requirement that descriptions use this function. It should be used for one reason only, to make a VHDL description concise. See the real-world design example in Appendix A called FLAGCTLR.

State Machines

See the *Warp* design examples titled intermediate TRAFFIC.VHD, intermediate DRINK.VHD and advanced TTF.VHD. See the application note titled "Describing State Machines in *Warp*2 VHDL." Also refer to pitfall numbers five and seven below.

Decoders

See the *Warp* design example titled basic DECODER.VHD and the special type-conversion function above.

Comparators

See the *Warp* design examples titled intermediate COMPARE.VHD and COMPARE2.VHD.

Counters

See the *Warp* design examples titled basic COUNTER.VHD, basic COUNTER2.VHD, intermediate COUNTER3.VHD, advanced COUNTER4.VHD, and advanced COUNTER5.VHD.

Multiplexers

Use the truth table function that is shown in the application note titled "Describing State Machines in *Warp*2 VHDL" or create a multiplexer using Boolean equations.

Shift Registers

See the *Warp* design example titled advanced SHIFTN.VHD. This example illustrates the use of the GENERATE statement.

Adders

See the *Warp* design examples titled basic ADDER1.VHD and basic ADDER2.VHD.

Repetitive Logic

The VHDL GENERATE statement lends itself to regular or repetitive logic structures. For example, n-bit registers, n-bit counters, n-bit shift registers, n-bit multiplexers, n-bit adders, and n-bit comparators may be concisely described by using the GENERATE statement. See the *Warp* design examples titled advanced SHIFTN.VHD and advanced COUNTER4.VHD.

Pitfalls

There are potential pitfalls. Some of the common mistakes made during conversion are:

1. Incorrect order of precedence of operators. For instance, all of the logical operators in VHDL have the same level of precedence. In other words, an equation that has both AND and OR operators requires parenthesis around the ANDed terms for proper logic synthesis. Refer to the cross reference and order of precedence table above.
2. Incomplete separation of clocked signals from combinatorial signals. Two simple ways to ensure proper logic synthesis of clocked signals and combinatorial signals are:
   a. Use a process for all signals, but use an IF-THEN-ELSIF-ENDIF within the process that groups all combinatorial signals under the IF, and groups all registered signals under the ELSIF. See the real-world design example in Appendix A called FLAGCTLR.
   b. Place all registered signals within a process (using a WAIT UNTIL CLOCK ='1') and place all combinatorial signals outside the process.
3. Using loops and variables outside of a process. VHDL requires that loops and variables be used inside a process. If there is more than one process, signals communicate between processes.
4. Using the incorrect mode for either output or bidirectional signals. Refer to your language reference book for a formal definition of mode.
5. Incomplete state specification for state machines. When designing a state machine, you **MUST** do one of the following:
   a. Specify all output values in each state of the machine.
   or
   b. Specify default values for all outputs at the beginning of the process.

The reason for this has to do with the way a process works. Each time a process is run (i.e., a clock event has occurred) the outputs that are specified in the particular pass through the process are updated. If a branch exists within the states of the machine that allows a pass through the process with one or more outputs not assigned a value, the logic synthesis engine either (a) assumes that the last statement for an unassigned output is valid and should be latched, or (b) that it is allowed to change with the clock. In other words, it is legal in VHDL to not specify all output values in each state of the machine, or not specify default values for all outputs at the beginning of the process, or not specify either one. If this subtle detail is overlooked, the design will compile and appear to synthesize successfully, but functional operation may not be correct. It is also possible that the logic synthesized will not be minimal. In other words, use defaults or specify the value of all outputs within each state of the machine.

6. Incorrect set or reset operation found in simulation. Polarity optimization settings used during logic synthesis and fitting can cause set and or reset operations to appear to operate inconsistently. During logic synthesis and fitting, the fitter can decide, by flipping the polarity of a function, the logic required will be minimized. This can have an adverse effect on the user selection of set or reset. (Note this pitfall only applies to 22V10s and FLASH370 where the polarity inversion is located between the output of the register and the pin.) See the polarity attribute in the *Warp* documentation for more details.
7. Failure to close, or complete, IF-THEN-ELSE-ENDIF and CASE statements. In other words, design descriptions that contain an IF must contain an ELSE, and descriptions containing a CASE-WHEN (condition), must contain a WHEN-OTHERS statement. This is required so that unnecessary implicit memory elements are not synthesized. See the application note titled "VHDL Techniques for Optimal Design Fitting" for more information.

Logic Synthesis

Proper logic synthesis is the goal of conversion. If the converted design compiles and synthesizes without errors, but the logic equations in the report file are not as expected (or simulation results are not as desired) consult the pitfalls section above. Also, consult your *Warp* - GALAXY compiler options documentation and *Warp* - NOVA user's guide. If all else fails, contact your local Cypress field application engineer.

Real-World Converted Designs

The designs in Appendix A originally were intended to fit into MACH 110s. However, due to product term and internal fanout requirements, MACH 210s were required. The designs were later converted to FLASH371s. Consult your Cypress data book for more information on the CY7C371's architecture.

Summary

Any design that has been described in Data I/O's ABEL language can be converted to VHDL. From an overall capability perspective, VHDL can be considered a superset of ABEL. Two designs documented in Appendix A were successfully converted using the cross reference tables and helpful hints contained within this application note.

Appendix A. Real-World Converted Designs

```
---------------------------------- cut here ----------------------------------

  Module FLAGCTLR
  Title 'Flag Controller 1 - Uxx_xx
  Revision 01'

  "ALGORITHM
  "
  "

  FLAGCTLR            device 'mach210a';

  "Inputs:

      R_40MHZ       pin ; "
      H_FEP_S0      pin ; "
      H_FEP_S1      pin ; "
      H_FEP_S2      pin ; "
      H_FEP_S3      pin ; "
      H_FEP_SET     pin ; "
      L_FEP_WE      pin ; "
      H_PPA_S0      pin ; "
      H_PPA_S1      pin ; "
      H_PPA_S2      pin ; "
      H_PPA_S3      pin ; "
      H_PPA_SET     pin ; "
      L_PPA_WE      pin ; "
      H_PPB_S0      pin ; "
      H_PPB_S1      pin ; "
      H_PPB_S2      pin ; "
      H_PPB_S3      pin ; "
      H_PPB_SET     pin ; "
      L_PPB_WE      pin ; "
      L_RESET       pin ; "

  "Outputs:

      H_FA0      pin  istype 'reg,buffer'; "
      H_FA1      pin  istype 'reg,buffer'; "
      H_FA2      pin  istype 'reg,buffer'; "
      H_FA3      pin  istype 'reg,buffer'; "
      H_FA4      pin  istype 'reg,buffer'; "
      H_FA5      pin  istype 'reg,buffer'; "
      H_FA6      pin  istype 'reg,buffer'; "
      H_FA7      pin  istype 'reg,buffer'; "

      H_FB0      pin  istype 'reg,buffer'; "
      H_FB1      pin  istype 'reg,buffer'; "
      H_FB2      pin  istype 'reg,buffer'; "
      H_FB3      pin  istype 'reg,buffer'; "
      H_FB4      pin  istype 'reg,buffer'; "

      H_AB0      pin  istype 'reg,buffer'; "
      H_AB1      pin  istype 'reg,buffer'; "
```

Appendix A. Real-World Converted Designs (continued)

```
    H_AB2      pin  istype 'reg,buffer'; "
    H_AB3      pin  istype 'reg,buffer'; "
    H_AB4      pin  istype 'reg,buffer'; "

Declarations

    X = .X.;
    C = .C.;
    Z = .Z.;

    FA =     [H_FA7,H_FA6,H_FA5,H_FA4,
              H_FA3,H_FA2,H_FA1,H_FA0];

    FB =     [H_FB4,H_FB3,H_FB2,H_FB1,H_FB0];
    AB =     [H_AB4,H_AB3,H_AB2,H_AB1,H_AB0];
    PPA_SEL  = [H_PPA_S3,H_PPA_S2,H_PPA_S1,H_PPA_S0];
    PPB_SEL  = [H_PPB_S3,H_PPB_S2,H_PPB_S1,H_PPB_S0];
    FEP_SEL  = [H_FEP_S3,H_FEP_S2,H_FEP_S1,H_FEP_S0];

Equations

  FA.CLK = R_40MHZ;
  FB.CLK = R_40MHZ;
  AB.CLK = R_40MHZ;

  FA.RE  = !L_RESET;
  FB.RE  = !L_RESET;
  AB.RE  = !L_RESET;


  H_FA0.T = (!H_FA0.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h0)
           #  H_FA0.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h0)
           # !H_FA0.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h0)
           #  H_FA0.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h0));

  H_FA1.T = (!H_FA1.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h1)
           #  H_FA1.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h1)
           # !H_FA1.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h1)
           #  H_FA1.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h1));


  H_FA2.T = (!H_FA2.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h2)
           #  H_FA2.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h2)
           # !H_FA2.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h2)
           #  H_FA2.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h2));

  H_FA3.T = (!H_FA3.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h3)
           #  H_FA3.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h3)
           # !H_FA3.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h3)
           #  H_FA3.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h3));


  H_FA4.T = (!H_FA4.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h4)
           #  H_FA4.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h4)
           # !H_FA4.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h4)
```

Appendix A. Real-World Converted Designs (continued)

```
          #  H_FA4.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h4));

H_FA5.T = (!H_FA5.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h5)
          #  H_FA5.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h5)
          # !H_FA5.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h5)
          #  H_FA5.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h5));


H_FA6.T = (!H_FA6.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h6)
          #  H_FA6.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h6)
          # !H_FA6.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h6)
          #  H_FA6.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h6));

H_FA7.T = (!H_FA7.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h7)
          #  H_FA7.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h7)
          # !H_FA7.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h7)
          #  H_FA7.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h7));


H_FB0.T = (!H_FB0.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h0)
          #  H_FB0.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h0)
          # !H_FB0.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h8)
          #  H_FB0.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h8));

H_FB1.T = (!H_FB1.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h1)
          #  H_FB1.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h1)
          # !H_FB1.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h9)
          #  H_FB1.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^h9));


H_FB2.T = (!H_FB2.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h2)
          #  H_FB2.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h2)
          # !H_FB2.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^ha)
          #  H_FB2.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^ha));

H_FB3.T = (!H_FB3.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h3)
          #  H_FB3.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h3)
          # !H_FB3.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^hb)
          #  H_FB3.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^hb));


H_AB0.T = (!H_AB0.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h8)
          #  H_AB0.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h8)
          # !H_AB0.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h8)
          #  H_AB0.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h8));

H_AB1.T = (!H_AB1.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h9)
          #  H_AB1.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h9)
          # !H_AB1.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h9)
          #  H_AB1.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^h9));


H_AB2.T = (!H_AB2.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^ha)
          #  H_AB2.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^ha)
          # !H_AB2.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^ha)
```

Appendix A. Real-World Converted Designs (continued)

```
            #  H_AB2.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^ha));

  H_AB3.T = (!H_AB3.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^hb)
            #  H_AB3.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^hb)
            # !H_AB3.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^hb)
            #  H_AB3.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^hb));


  H_FB4.T = (!H_FB4.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h4)
            #  H_FB4.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^h4)
            # !H_FB4.Q &  H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^hc)
            #  H_FB4.Q & !H_FEP_SET & !L_FEP_WE & (FEP_SEL == ^hc));


  H_AB4.T = (!H_AB4.Q &  H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^hc)
            #  H_AB4.Q & !H_PPB_SET & !L_PPB_WE & (PPB_SEL == ^hc)
            # !H_AB4.Q &  H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^hc)
            #  H_AB4.Q & !H_PPA_SET & !L_PPA_WE & (PPA_SEL == ^hc));

test_vectors ([R_40MHZ,L_RESET,
               L_FEP_WE, FEP_SEL, H_FEP_SET,
               L_PPA_WE, PPA_SEL, H_PPA_SET,
               L_PPB_WE, PPB_SEL, H_PPB_SET]
           -> [H_FA7, H_FA6, H_FA5, H_FA4, H_FA3, H_FA2, H_FA1, H_FA0,
               H_FB4, H_FB3, H_FB2, H_FB1, H_FB0,
               H_AB4, H_AB3, H_AB2, H_AB1, H_AB0])
[C,1,1,^h0,0,1,^h1,0,1,^h0,0]->[X,X,X,X,X,X,X,X, X,X,X,X,X, X,X,X,X,X];
[C,1,0,^h0,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h1,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h2,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h3,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h4,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h5,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h6,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h7,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h8,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h9,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^hA,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^hB,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^hC,0,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h0,1,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,0,1, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h1,1,1,^h1,0,1,^h0,0]->[0,0,0,0,0,0,1,1, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h2,1,1,^h1,0,1,^h0,0]->[0,0,0,0,0,1,1,1, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h3,1,1,^h1,0,1,^h0,0]->[0,0,0,0,1,1,1,1, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h4,1,1,^h1,0,1,^h0,0]->[0,0,0,1,1,1,1,1, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h5,1,1,^h1,0,1,^h0,0]->[0,0,1,1,1,1,1,1, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h6,1,1,^h1,0,1,^h0,0]->[0,1,1,1,1,1,1,1, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h7,1,1,^h1,0,1,^h0,0]->[1,1,1,1,1,1,1,1, 0,0,0,0,0, 0,0,0,0,0];
[C,1,0,^h8,1,1,^h1,0,1,^h0,0]->[1,1,1,1,1,1,1,1, 0,0,0,0,1, 0,0,0,0,0];
[C,1,0,^h9,1,1,^h1,0,1,^h0,0]->[1,1,1,1,1,1,1,1, 0,0,0,1,1, 0,0,0,0,0];
[C,1,0,^hA,1,1,^h1,0,1,^h0,0]->[1,1,1,1,1,1,1,1, 0,0,1,1,1, 0,0,0,0,0];
[C,1,0,^hB,1,1,^h1,0,1,^h0,0]->[1,1,1,1,1,1,1,1, 0,1,1,1,1, 0,0,0,0,0];
[C,1,0,^hC,1,1,^h1,0,1,^h0,0]->[1,1,1,1,1,1,1,1, 1,1,1,1,1, 0,0,0,0,0];
[C,1,1,^h7,1,0,^h0,0,1,^h0,0]->[1,1,1,1,1,1,1,0, 1,1,1,1,1, 0,0,0,0,0];
```

Appendix A. Real-World Converted Designs (continued)

```
  [C,1,1,^h7,1,0,^h1,0,1,^h0,0]->[1,1,1,1,1,1,0,0, 1,1,1,1,1, 0,0,0,0,0];
  [C,1,1,^h7,1,0,^h2,0,1,^h0,0]->[1,1,1,1,1,0,0,0, 1,1,1,1,1, 0,0,0,0,0];
  [C,1,1,^h7,1,0,^h3,0,1,^h0,0]->[1,1,1,1,0,0,0,0, 1,1,1,1,1, 0,0,0,0,0];
  [C,1,1,^h7,1,0,^h4,0,1,^h0,0]->[1,1,1,0,0,0,0,0, 1,1,1,1,1, 0,0,0,0,0];
  [C,1,1,^h7,1,0,^h5,0,1,^h0,0]->[1,1,0,0,0,0,0,0, 1,1,1,1,1, 0,0,0,0,0];
  [C,1,1,^h7,1,0,^h6,0,1,^h0,0]->[1,0,0,0,0,0,0,0, 1,1,1,1,1, 0,0,0,0,0];
  [C,1,1,^h7,1,0,^h7,0,1,^h0,0]->[0,0,0,0,0,0,0,0, 1,1,1,1,1, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h0,0]->[0,0,0,0,0,0,0,0, 1,1,1,1,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h1,0]->[0,0,0,0,0,0,0,0, 1,1,1,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h2,0]->[0,0,0,0,0,0,0,0, 1,1,0,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h3,0]->[0,0,0,0,0,0,0,0, 1,0,0,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h4,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h8,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h9,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^hA,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^hB,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^hC,0]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^hC,1]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 1,0,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^hB,1]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 1,1,0,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^hA,1]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 1,1,1,0,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h9,1]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 1,1,1,1,0];
  [C,1,1,^h7,1,1,^h7,0,0,^h8,1]->[0,0,0,0,0,0,0,0, 0,0,0,0,0, 1,1,1,1,1];
  [C,1,1,^h7,1,1,^h7,1,0,^h4,1]->[0,0,0,0,0,0,0,0, 1,0,0,0,0, 1,1,1,1,1];
  [C,1,1,^h7,1,1,^h7,1,0,^h3,1]->[0,0,0,0,0,0,0,0, 1,1,0,0,0, 1,1,1,1,1];
  [C,1,1,^h7,1,1,^h7,1,0,^h2,1]->[0,0,0,0,0,0,0,0, 1,1,1,0,0, 1,1,1,1,1];
  [C,1,1,^h7,1,1,^h7,1,0,^h1,1]->[0,0,0,0,0,0,0,0, 1,1,1,1,0, 1,1,1,1,1];
  [C,1,1,^h7,1,1,^h7,1,0,^h0,1]->[0,0,0,0,0,0,0,0, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h7,1,1,^h0,1]->[1,0,0,0,0,0,0,0, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h6,1,1,^h0,1]->[1,1,0,0,0,0,0,0, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h5,1,1,^h0,1]->[1,1,1,0,0,0,0,0, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h4,1,1,^h0,1]->[1,1,1,1,0,0,0,0, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h3,1,1,^h0,1]->[1,1,1,1,1,0,0,0, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h2,1,1,^h0,1]->[1,1,1,1,1,1,0,0, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h1,1,1,^h0,1]->[1,1,1,1,1,1,1,0, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h0,1,1,^h0,1]->[1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,0,^h8,0,1,^h0,1]->[1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,1,1,1,0];
  [C,1,1,^h7,1,0,^h9,0,1,^h0,1]->[1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,1,1,0,0];
  [C,1,1,^h7,1,0,^hA,0,1,^h0,1]->[1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,1,0,0,0];
  [C,1,1,^h7,1,0,^hB,0,1,^h0,1]->[1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,0,0,0,0];
  [C,1,1,^h7,1,0,^hC,0,1,^h0,1]->[1,1,1,1,1,1,1,1, 1,1,1,1,1, 0,0,0,0,0];
  [C,1,1,^h7,1,0,^hC,1,1,^h0,1]->[1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,0,0,0,0];
  [C,1,1,^h7,1,0,^hB,1,1,^h0,1]->[1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,1,0,0,0];
  [C,1,1,^h7,1,0,^hA,1,1,^h0,1]->[1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,1,1,0,0];
  [C,1,1,^h7,1,0,^h9,1,1,^h0,1]->[1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,1,1,1,0];
  [C,1,1,^h7,1,0,^h8,1,1,^h0,1]->[1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,1,1,1,1];
  [C,1,1,^h7,1,1,^h0,1,1,^h0,1]->[1,1,1,1,1,1,1,1, 1,1,1,1,1, 1,1,1,1,1];

END FLAGCTLR;

---------------------------------- cut here ----------------------------------

--  Converted to IEEE 1076 VHDL

--  Module FLAGCTLR
```

Appendix A. Real-World Converted Designs (continued)

```
--  Title 'Flag Controller 1 - Uxx_xx
--  Revision 01'

use work.cypress.all;
use work.rtlpkg.all;
use work.int_math.all;

ENTITY FLAGCTLR IS PORT(
  R_40MHZ,H_FEP_SET,L_FEP_WE,H_PPA_SET,
  L_PPA_WE,H_PPB_SET,L_PPB_WE,L_RESET    : IN BIT;
  PPA_SEL,PPB_SEL,FEP_SEL                : IN BIT_VECTOR (3 downto 0);
  FA                                     : INOUT BIT_VECTOR (7 downto 0);
  FB,AB                                  : INOUT BIT_VECTOR (4 downto 0));
attribute part_name of eventflg: entity is "c371";
END FLAGCTLR;

ARCHITECTURE CONVERTED_ABL OF FLAGCTLR IS

FUNCTION frbl_to_b(in1:Boolean)     RETURN bit IS
BEGIN
IF (in1=true) THEN
     RETURN '1';
ELSE
     RETURN '0';
END IF;
END frbl_to_b;
--  This type conversion function converts a signal or relational operation
--  result from type BOOLEAN to type BIT.  A Boolean can have a value of either
-- 'TRUE' or'FALSE'.  A bit can have a value of either '0' or '1'.

BEGIN
PROCESS (R_40MHZ, L_RESET)
BEGIN

IF (L_RESET ='0') THEN

      FOR i IN 0 TO 4 LOOP
           FA(i) <= '0'; FB(i) <= '0'; AB(i) <= '0';
      END LOOP;

      FOR i IN 5 TO 7 LOOP
           FA(i) <= '0';
      END LOOP;

ELSIF (R_40MHZ'EVENT AND R_40MHZ ='1') THEN

  FA(0) <= FA(0) XOR
         ((NOT FA(0) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"0"))
       OR (FA(0) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"0"))
       OR (NOT FA(0) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"0"))
       OR (FA(0) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"0")));

  FA(1) <= FA(1) XOR
         ((NOT FA(1) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"1"))
       OR (FA(1) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"1"))
```

Appendix A. Real-World Converted Designs (continued)

```
    OR (NOT FA(1) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"1"))
    OR (FA(1) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"1")));

FA(2) <= FA(2) XOR
        ((NOT FA(2) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"2"))
    OR (FA(2) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"2"))
    OR (NOT FA(2) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"2"))
    OR (FA(2) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"2")));

FA(3) <= FA(3) XOR
        ((NOT FA(3) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"3"))
    OR (FA(3) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"3"))
    OR (NOT FA(3) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"3"))
    OR (FA(3) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"3")));

FA(4) <= FA(4) XOR
        ((NOT FA(4) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"4"))
    OR (FA(4) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"4"))
    OR (NOT FA(4) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"4"))
    OR (FA(4) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"4")));

FA(5) <= FA(5) XOR
        ((NOT FA(5) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"5"))
    OR (FA(5) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"5"))
    OR (NOT FA(5) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"5"))
    OR (FA(5) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"5")));

FA(6) <= FA(6) XOR
        ((NOT FA(6) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"6"))
    OR (FA(6) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"6"))
    OR (NOT FA(6) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"6"))
    OR (FA(6) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"6")));

FA(7) <= FA(7) XOR
        ((NOT FA(7) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"7"))
    OR (FA(7) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"7"))
    OR (NOT FA(7) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"7"))
    OR (FA(7) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"7")));

FB(0) <= FB(0) XOR
        ((NOT FB(0) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"0"))
    OR (FB(0) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"0"))
    OR (NOT FB(0) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"8"))
    OR (FB(0) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"8")));

FB(1) <= FB(1) XOR
        ((NOT FB(1) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"1"))
    OR (FB(1) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"1"))
    OR (NOT FB(1) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"9"))
    OR (FB(1) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"9")));

FB(2) <= FB(2) XOR
        ((NOT FB(2) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"2"))
    OR (FB(2) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"2"))
    OR (NOT FB(2) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"A"))
```

Appendix A. Real-World Converted Designs (continued)

```
      OR (FB(2) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"A")));

  FB(3) <= FB(3) XOR
         ((NOT FB(3) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"3"))
      OR (FB(3) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"3"))
      OR (NOT FB(3) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"B"))
      OR (FB(3) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"B")));

  AB(0) <= AB(0) XOR
         ((NOT AB(0) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"8"))
      OR (AB(0) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"8"))
      OR (NOT AB(0) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"8"))
      OR (AB(0) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"8")));

  AB(1) <= AB(1) XOR
         ((NOT AB(1) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"9"))
      OR (AB(1) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"9"))
      OR (NOT AB(1) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"9"))
      OR (AB(1) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"9")));

  AB(2) <= AB(2) XOR
         ((NOT AB(2) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"A"))
      OR (AB(2) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"A"))
      OR (NOT AB(2) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"A"))
      OR (AB(2) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"A")));

  AB(3) <= AB(3) XOR
         ((NOT AB(3) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"B"))
      OR (AB(3) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"B"))
      OR (NOT AB(3) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"B"))
      OR (AB(3) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"B")));

  FB(4) <= FB(4) XOR
         ((NOT FB(4) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"4"))
      OR (FB(4) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"4"))
      OR (NOT FB(4) AND H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"C"))
      OR (FB(4) AND NOT H_FEP_SET AND NOT L_FEP_WE AND frbl_to_b(FEP_SEL=x"C")));

  AB(4) <= AB(4) XOR
         ((NOT AB(4) AND H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"C"))
      OR (AB(4) AND NOT H_PPB_SET AND NOT L_PPB_WE AND frbl_to_b(PPB_SEL=x"C"))
      OR (NOT AB(4) AND H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"C"))
      OR (AB(4) AND NOT H_PPA_SET AND NOT L_PPA_WE AND frbl_to_b(PPA_SEL=x"C")));

END IF;
END PROCESS;
END CONVERTED_ABL;

------------------------------------ cut here ----------------------------------

Module CONVERTER
Title 'Converter
Revision 01'

" This device converts a 32-bit floating point word from one format to another.
```

Appendix A. Real-World Converted Designs (continued)

```
CONVERTER                       device       'MACH210A';

"Control Inputs:

CLK             PIN 35;" Clock
OE              PIN 10;" Low Active Output Enable
WE              PIN 11;" Low Active Write Enable
MODE            PIN 13;" Shift Mode

"Data I/O BITS:

D31             PIN 43    ISTYPE    'REG,BUFFER';
D30             PIN 42    ISTYPE    'REG,BUFFER';
D29             PIN 41    ISTYPE    'REG,BUFFER';
D28             PIN 40    ISTYPE    'REG,BUFFER';
D27             PIN 39    ISTYPE    'REG,BUFFER';
D26             PIN 38    ISTYPE    'REG,BUFFER';
D25             PIN 37    ISTYPE    'REG,BUFFER';
D24             PIN 36    ISTYPE    'REG,BUFFER';
D23             PIN 31    ISTYPE    'REG,BUFFER';
D22             PIN 30    ISTYPE    'REG,BUFFER';
D21             PIN 29    ISTYPE    'REG,BUFFER';
D20             PIN 28    ISTYPE    'REG,BUFFER';
D19             PIN 27    ISTYPE    'REG,BUFFER';
D18             PIN 26    ISTYPE    'REG,BUFFER';
D17             PIN 25    ISTYPE    'REG,BUFFER';
D16             PIN 24    ISTYPE    'REG,BUFFER';
D15             PIN 21    ISTYPE    'REG,BUFFER';
D14             PIN 20    ISTYPE    'REG,BUFFER';
D13             PIN 19    ISTYPE    'REG,BUFFER';
D12             PIN 18    ISTYPE    'REG,BUFFER';
D11             PIN 17    ISTYPE    'REG,BUFFER';
D10             PIN 16    ISTYPE    'REG,BUFFER';
D09             PIN 15    ISTYPE    'REG,BUFFER';
D08             PIN 14    ISTYPE    'REG,BUFFER';
D07             PIN  9    ISTYPE    'REG,BUFFER';
D06             PIN  8    ISTYPE    'REG,BUFFER';
D05             PIN  7    ISTYPE    'REG,BUFFER';
D04             PIN  6    ISTYPE    'REG,BUFFER';
D03             PIN  5    ISTYPE    'REG,BUFFER';
D02             PIN  4    ISTYPE    'REG,BUFFER';
D01             PIN  3    ISTYPE    'REG,BUFFER';
D00             PIN  2    ISTYPE    'REG,BUFFER';

H,L,C,Z,X   =   1,0,.C.,.Z.,.X.;

DIN         =   [D19.PIN,D18.PIN,D17.PIN,D16.PIN,
                 D15.PIN,D14.PIN,D13.PIN,D12.PIN,
                 D11.PIN,D10.PIN,D09.PIN,D08.PIN,
                 D07.PIN,D06.PIN,D05.PIN,D04.PIN,
                 D03.PIN,D02.PIN,D01.PIN,D00.PIN];

DOUT        =   [D31,D30,D29,D28,D27,D26,D25,D24,
                 D23,D22,D21,D20,D19,D18,D17,D16,
```

Appendix A. Real-World Converted Designs (continued)

```
                D15,D14,D13,D12,D11,D10,D09,D08,
                D07,D06,D05,D04,D03,D02,D01,D00];

DBIDI       =   [D19,D18,D17,D16,D15,D14,D13,D12,
                D11,D10,D09,D08,D07,D06,D05,D04,
                D03,D02,D01,D00];

DOUTFB      =   [L,L,L,L,L,L,L,L,L,L,L,L,
                D19.FB,D18.FB,D17.FB,D16.FB,
                D15.FB,D14.FB,D13.FB,D12.FB,
                D11.FB,D10.FB,D09.FB,D08.FB,
                D07.FB,D06.FB,D05.FB,D04.FB,
                D03.FB,D02.FB,D01.FB,D00.FB];

M0_SHIFT6   =   !WE & !MODE &
                ((D19.PIN == H) # (D18.PIN == H) # (D17.PIN == H));

M0_SHIFT3   =   !WE & !MODE &
                (((D19.PIN == L) & (D18.PIN == L) & (D17.PIN == L)) &
                ((D16.PIN == H) # (D15.PIN == H) # (D14.PIN == H)));

SHIFT0      =   !WE &
                (((D19.PIN == L) & (D18.PIN == L) & (D17.PIN == L)) &
                ((D16.PIN == L) & (D15.PIN == L) & (D14.PIN == L)));

M1_SHIFT6   =   !WE & MODE &
                ((D19.PIN == H) # (D18.PIN == H));

M1_SHIFT4   =   !WE & MODE &
                (((D19.PIN == L) & (D18.PIN == L)) &
                ((D17.PIN == H) # (D16.PIN == H)));

M1_SHIFT2   =   !WE & MODE &
                (((D19.PIN == L) & (D18.PIN == L) &
                (D17.PIN == L) & (D16.PIN == L)) &
                ((D15.PIN == H) # (D14.PIN == H)));

EQUATIONS

DOUT.CLK    =   CLK;

DOUT.OE     =   !OE;
DOUT        :=  M0_SHIFT6  &
                        [L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,H,L,D19.PIN,D18.PIN,
                             D17.PIN,D16.PIN,D15.PIN,D14.PIN,D13.PIN,D12.PIN,
                             D11.PIN,D10.PIN,D09.PIN,D08.PIN,D07.PIN,D06.PIN]

              #

                M0_SHIFT3  &
                        [L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,H,D16.PIN,D15.PIN,
                             D14.PIN,D13.PIN,D12.PIN,D11.PIN,D10.PIN,D09.PIN,
                             D08.PIN,D07.PIN,D06.PIN,D05.PIN,D04.PIN,D03.PIN]

              #

                SHIFT0      &
                        [L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,D13.PIN,D12.PIN,
```

Appendix A. Real-World Converted Designs (continued)

```
                        D11.PIN,D10.PIN,D09.PIN,D08.PIN,D07.PIN,D06.PIN,
                        D05.PIN,D04.PIN,D03.PIN,D02.PIN,D01.PIN,D00.PIN]

        #

          M1_SHIFT6  &
                    [L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,H,H,D19.PIN,D18.PIN,
                        D17.PIN,D16.PIN,D15.PIN,D14.PIN,D13.PIN,D12.PIN,
                        D11.PIN,D10.PIN,D09.PIN,D08.PIN,D07.PIN,D06.PIN]

        #

          M1_SHIFT4  &
                    [L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,H,L,D17.PIN,D16.PIN,
                        D15.PIN,D14.PIN,D13.PIN,D12.PIN,D11.PIN,D10.PIN,
                        D09.PIN,D08.PIN,D07.PIN,D06.PIN,D05.PIN,D04.PIN]

        #

          M1_SHIFT2  &
                    [L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,H,D15.PIN,D14.PIN,
                        D13.PIN,D12.PIN,D11.PIN,D10.PIN,D09.PIN,D08.PIN,
                        D07.PIN,D06.PIN,D05.PIN,D04.PIN,D03.PIN,D02.PIN]

        #

          WE & DOUTFB;

Test_Vectors

([CLK,OE,WE,MODE,                DBIDI]          ->       DOUT)
 [C,H,L,X,                      ^b00000000101101110110] -> Z;"Write
 [C,L,H,X,Z]     -> ^b00000000000000000000000101101110110;"Read,Shift0
 [C,H,L,L,                      ^b00000110101101011011] -> Z;"Write
 [C,L,H,X,Z]     -> ^b00000000000000000001001101101101011;"Read,Shift3/Mode0
 [C,H,L,L,                      ^b00001010101101011011] -> Z;"Write
 [C,L,H,X,Z]     -> ^b00000000000000000001010101101101011;"Read,Shift3/Mode0
 [C,H,L,L,                      ^b00010010101101011011] -> Z;"Write
 [C,L,H,X,Z]     -> ^b00000000000000000001100101101101011;"Read,Shift3/Mode0
 [C,H,L,L,                      ^b00011010101101011011] -> Z;"Write
 [C,L,H,X,Z]     -> ^b00000000000000000001110101101101011;"Read,Shift3/Mode0
 [C,H,L,L,                      ^b00010110101101011011] -> Z;"Write
 [C,L,H,X,Z]     -> ^b00000000000000000001101101101101011;"Read,Shift3/Mode0
 [C,H,L,L,                      ^b00001110101101011011] -> Z;"Write
 [C,L,H,X,Z]     -> ^b00000000000000000001011101101101011;"Read,Shift3/Mode0
 [C,H,L,L,                      ^b00011110101101011011] -> Z;"Write
 [C,L,H,X,Z]     -> ^b00000000000000000001111101101101011;"Read,Shift3/Mode0
 [C,H,L,L,                      ^b00111110101101011011] -> Z;"Write
 [C,L,H,X,Z]     -> ^b00000000000000000010001111101101101;"Read,Shift6/Mode0
 [C,H,L,L,                      ^b01011110101101011011] -> Z;"Write
 [C,L,H,X,Z]     -> ^b00000000000000000010010111101101101;"Read,Shift6/Mode0
 [C,H,L,L,                      ^b10011110101101011011] -> Z;"Write
 [C,L,H,X,Z]     -> ^b00000000000000000010100111101101101;"Read,Shift6/Mode0
 [C,H,L,L,                      ^b10100110101101011011] -> Z;"Write
 [C,L,H,X,Z]     -> ^b00000000000000000010101001101101101;"Read,Shift6/Mode0
```

Appendix A. Real-World Converted Designs (continued)

```
 [C,H,L,L,                    ^b01100110101101011011] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000100110011010110l;"Read,Shift6/Mode0
 [C,H,L,L,                    ^b11000110101101011011] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000101100011010110l;"Read,Shift6/Mode0
 [C,H,L,L,                    ^b11111110101101011011] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000101111111010110l;"Read,Shift6/Mode0
 [C,H,L,H,                    ^b00000100101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000101001011011101;"Read,Shift2/Mode1
 [C,H,L,H,                    ^b00001000101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000110001011011101;"Read,Shift2/Mode1
 [C,H,L,H,                    ^b00001100101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000111001011011101;"Read,Shift2/Mode1
 [C,H,L,H,                    ^b00010000101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000100100001011011l;"Read,Shift4/Mode1
 [C,H,L,H,                    ^b00100000101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000101000001011011l;"Read,Shift4/Mode1
 [C,H,L,H,                    ^b00110000101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000101100001011011l;"Read,Shift4/Mode1
 [C,H,L,H,                    ^b00111100101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000101111001011011l;"Read,Shift4/Mode1
 [C,H,L,H,                    ^b10000000101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000111000000010110l;"Read,Shift6/Mode1
 [C,H,L,H,                    ^b01000000101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000110100000010110l;"Read,Shift6/Mode1
 [C,H,L,H,                    ^b11000000101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000111100000010110l;"Read,Shift6/Mode1
 [C,H,L,H,                    ^b11010100101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000111101010010110l;"Read,Shift6/Mode1
 [C,H,L,H,                    ^b11100100101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000111110100010110l;"Read,Shift6/Mode1
 [C,H,L,H,                    ^b11111100101101110110] -> Z;"Write
 [C,L,H,X,Z]    -> ^b00000000000000000111111110010110l;"Read,Shift6/Mode1

End CONVERTER;

---------------------------------- cut here ----------------------------------

-- CONVERTED TO IEEE 1076 VHDL

-- Module CONVERTER
-- Title 'Converter
-- Revision 01'

-- This device converts a 32-bit floating point word from one format to another.

-- Control Inputs

use work.cypress.all;
use work.rtlpkg.all;
use work.int_math.all;

ENTITY CONVERTER IS PORT(
CLK,OE,WE,MODE                              : IN BIT;
D                                           : INOUT X01Z_VECTOR (0 TO 31));
```

Appendix A. Real-World Converted Designs (continued)

```
attribute part_name of CONVERTER: entity is "c371";
END CONVERTER;

ARCHITECTURE CONVERTED_ABL OF CONVERTER IS
SIGNAL SHIFT2_TMP, SHIFT1_TMP, SHIFT0_TMP, SHIFT2,
       SHIFT1, SHIFT0, M0_SHIFT6, M0_SHIFT3,
       M_SHIFT0, M1_SHIFT6, M1_SHIFT4, M1_SHIFT2      : BIT;
SIGNAL D_TMP, D_FB                                    : BIT_VECTOR (0 TO 31);
BEGIN

P1: PROCESS
 BEGIN
  WAIT UNTIL CLK ='1';

   FOR i IN 0 TO 13 LOOP
   D_TMP(i) <=(D_FB(i+6) AND     SHIFT2 AND     SHIFT1 AND NOT SHIFT0)
           OR (D_FB(i+4) AND     SHIFT2 AND NOT SHIFT1 AND NOT SHIFT0)
           OR (D_FB(i+3) AND NOT SHIFT2 AND     SHIFT1 AND     SHIFT0)
           OR (D_FB(i+2) AND NOT SHIFT2 AND     SHIFT1 AND NOT SHIFT0)
           OR (D_FB(i+0) AND NOT SHIFT2 AND NOT SHIFT1 AND NOT SHIFT0)
           OR (WE AND D_TMP(i));
   END LOOP;

   D_TMP(14) <= ('0' AND M0_SHIFT6) OR ('1' AND M0_SHIFT3)
           OR ('0' AND  M_SHIFT0) OR ('1' AND M1_SHIFT6)
           OR ('0' AND M1_SHIFT4) OR ('1' AND M1_SHIFT2)
           OR (WE AND D_TMP(14));

   D_TMP(15) <= ('1' AND M0_SHIFT6) OR ('0' AND M0_SHIFT3)
           OR ('0' AND  M_SHIFT0) OR ('1' AND M1_SHIFT6)
           OR ('1' AND M1_SHIFT4) OR ('0' AND M1_SHIFT2)
           OR (WE AND D_TMP(15));

   FOR i IN 16 TO 28 LOOP
   D_TMP(i) <='0';
   END LOOP;

END PROCESS P1;

M0_SHIFT6  <= (NOT WE AND MODE) AND
              (D_FB(19) OR D_FB(18) OR D_FB(17));

M0_SHIFT3  <= (NOT WE AND NOT MODE AND NOT D_FB(19) AND NOT D_FB(18)
               AND NOT D_FB(17)) AND (D_FB(16) OR D_FB(15) OR D_FB(14));

M_SHIFT0   <= NOT WE AND NOT D_FB(19) AND NOT D_FB(18) AND NOT
              D_FB(17) AND NOT D_FB(16) AND NOT D_FB(15) AND NOT D_FB(14);

M1_SHIFT6  <= (NOT WE AND MODE) AND (D_FB(19) OR D_FB(18));

M1_SHIFT4  <= (NOT WE AND MODE AND NOT D_FB(19) AND NOT D_FB(18))
               AND (D_FB(17) OR D_FB(16));

M1_SHIFT2  <= (NOT WE AND MODE AND NOT D_FB(19) AND NOT D_FB(18) AND
               NOT D_FB(17) AND NOT D_FB(16)) AND (D_FB(15) OR D_FB(14));
```

Appendix A. Real-World Converted Designs (continued)

```
SHIFT2_TMP <= M0_SHIFT6
              OR M1_SHIFT6
              OR M1_SHIFT4;

SHIFT1_TMP <= M0_SHIFT6
              OR M0_SHIFT3
              OR M1_SHIFT6
              OR M1_SHIFT2;

SHIFT0_TMP <= M0_SHIFT3;

-- Mapping for the Bidirectional buffers
-- D_TMP is the internal signal which drives the output buffer
-- OE    is the signal for output enable (active high)
-- D     is the pin name, matches name in port assignment
-- D_FB  is the signal from pin that feeds back and drives the internal
--       structure
G1: FOR i IN 0 TO 28 GENERATE
      B1:BUFOE PORT MAP(D_TMP(i), OE,D(i), D_FB(i));
END GENERATE;

B2: BUF PORT MAP(SHIFT0_TMP,SHIFT0); -- Forces logic synthesis to "split
B3: BUF PORT MAP(SHIFT1_TMP,SHIFT1); -- sums" into SHIFT codes that are
B4: BUF PORT MAP(SHIFT2_TMP,SHIFT2); -- encoded and placed on outputs D29-31

B5: BUFOE PORT MAP(SHIFT0, OE, D(29), open);
B6: BUFOE PORT MAP(SHIFT1, OE, D(30), open);
B7: BUFOE PORT MAP(SHIFT2, OE, D(31), open);

END CONVERTED_ABL;

---------------------------------- cut here ----------------------------------
```

Document #: PLD0004–0194

Warp, and *Warp2* are trademarks of Cypress Semiconductor Corporation.
MACH is a trademark of Advanced Micro Devices, Inc.

Top-Down Design Methodology With VHDL (Designing an Interrupt Controller)

Working from a known objective to a finished product is one of the many challenges facing engineers today. Goals such as designing a DMA controller, or interfacing a processor to a VMEbus, or designing an interrupt controller are very abstract and need to be broken down into manageable pieces. All of the internal functions necessary to accomplish the design are not yet known so this approach is known as top-down design and can be easily accomplished with VHDL.

Let's take the example of the interrupt controller and interface it to the Cypress CY7C611A embedded controller. The necessary features must be listed. This is the process of breaking down the design.

Interrupt Controller features:

1. Register the interrupt lines. This helps eliminate false interrupts from glitches on the lines and reduces the possibility of metastability occurrences.
2. Prioritize the interrupts. The interrupts with higher priority will appear on the encoded outputs before those of lower priority.
3. Ability to mask individual interrupts. This gives the processor the added ability to turn off individual interrupts to allow code to run undisturbed.
4. Ability to encode the interrupts. The CY7C611A has four encoded interrupt lines to reduce pin count. The incoming value of '0' is no interrupt, while 'F' (hexadecimal) is a non-maskable interrupt.
5. Processor interface for setting, clearing, and reading mask bits.
6. Processor interface for reading and writing interrupts. This can be used as a diagnostic tool.
7. A RESET signal that is equivalent to the power-on reset signal that the processor receives. This is used to initialize the interrupt controller. The initial state will be all interrupts disabled (except the non-maskable interrupt).

From this list we can draw a simple block diagram (*Figure 1*) and create an Entity statement for VHDL (*Figure 2*).

Now the task at hand is to take the blocks outlined in *Figure 1* and create the VHDL that will describe them. The four basic blocks are the Interrupt Register, Mask Register, Mask Logic, and Priority Encoder. The divide-by-two block on the clock is used to illustrate that the encoded interrupt signals need to be held for a minimum of two clock cycles to meet the CY7C611A specification.

The first block that will be completed is the Interrupt Register (*Figure 3*). The features that it must have are:

1. Register incoming interrupts (15 signals)
2. All interrupt registers can be written to or read from over the bus.

From this feature list we can create the ENTITY statement for the Interrupt Register block (*Figure 4*).

Note the mode and type of the DATA signals (INOUT, x01z_VECTOR). Unlike the other signals that have been defined, the DATA signals must be bidirectional (INOUT). When the Mask or Inter-

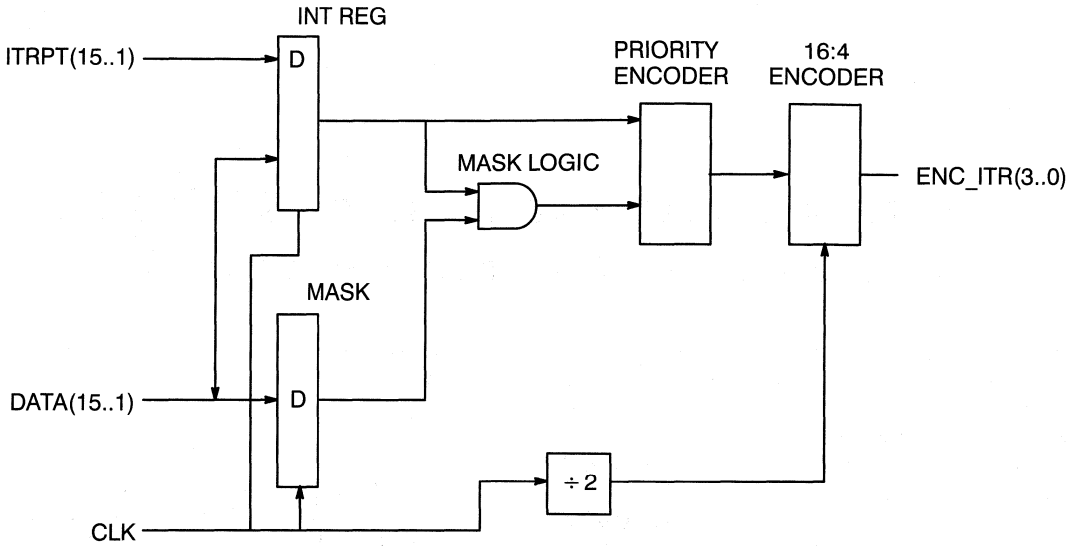


Figure 1. Block Diagram

```
ENTITY int_ctl IS
  PORT (
    clk,RESETn,CSn,RD_WRn:IN BIT;
    DATA:                 INOUT x01z_VECTOR(15 DOWNTO 1);
    msk_reg:              IN BIT;  -- H: mask registers; L: itrpt registers
    ITRPT:                IN BIT_VECTOR(15 DOWNTO 1);
    ENC_ITR:              OUT BIT_VECTOR(3 DOWNTO 0)
    );
END int_ctl;
```

Figure 2. VHDL Entity Statement

rupt Registers are being written, the DATA lines are inputs on the PLD. When these registers are read, the DATA lines become outputs. A bidirectional signal is achieved in a PLD by the use of a three-state buffer.

The two possibilities for RI(15..1) are simply the registered value of the corresponding ITRPT(15..1) or DATA(15..1) (when the chip select (CSn) is LOW, the write line is LOW (RD_WRn) and the interrupt registers are selected (MSK_REG is LOW)). RESETn does not enter into these equations because that function is handled by the Mask register (*Figure 5*).

The signals will not be captured, only registered. This will not be a problem for the debug mode of writing an interrupt to the controller because data will be valid for more than the required two clock cycles.

When the interrupt registers are read (CSn and MSK_REG -->L, RD_WRn --> H), the DATA lines should take the values of RI(15..1). This is accomplished by using a component from the pro-

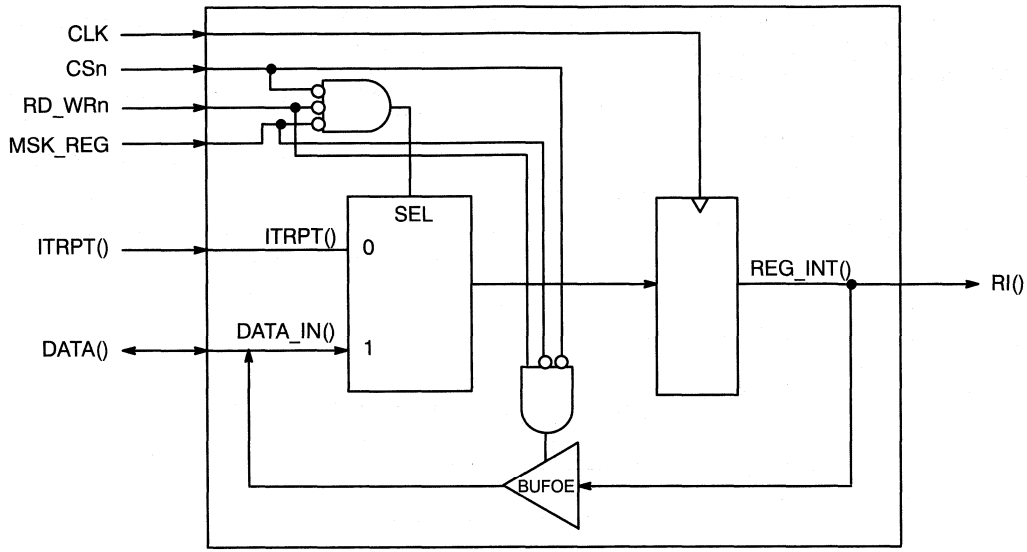


Figure 3. Interrupt Register

```
ENTITY int_reg IS
   PORT (
      clk,CSn,RD_WRn,msk_reg :      IN BIT;
      ITRPT:                        IN BIT_VECTOR(15..1);
      DATA:                         INOUT x01z_VECTOR(15 DOWNTO 1);
      RI:                           OUT BIT_VECTOR(15 DOWNTO 1)
      );
END int_reg;
```

Figure 4. Interrupt Register Entity Statement

```
RI(i) <= (ITRPT(i))
   OR     (DATAIN(i) AND NOT(CSn) AND NOT(RD_WRn) AND NOT(msk_reg));
```

Figure 5. Interrupt Register Definition

vided package *rtlpkg* called *bufoe* (*Figure 6*). As mentioned above, bidirectional signals are realized by using three-state buffers. The port map for the *bufoe* signal is as follows: (DATA_IN, ENABLE, DATA_OUT, FEEDBACK). All signals except DATA_OUT are of type bit. The DATA_IN and ENABLE signals are of mode INPUT, while the FEEDBACK is of mode OUTPUT. Because the DATA_OUT signal is used as an input and is tied to an output, it must be of mode INOUT and type x01z. Therefore the port map of our *bufoe* can be defined as shown in *Figure 7*.

REG_int(i) are the inputs to the buffer, READ_OE is the active HIGH three-state control signal, DATA(i) are the names of the signals that connect to the pins of the part, and DATAIN(i) are the signals that feed back into the part.

The next step is to define the code for the Mask register (*Figure 8*). The functions that it must perform are:

1. Store the mask bits (14 signals, interrupt 15 is non-maskable).
2. All Mask registers can be written to or read from over the bus.
3. RESET state is no interrupts (all mask bits set to zero)

From this feature list we can create the Entity statement for the Mask register block (*Figure 9*).

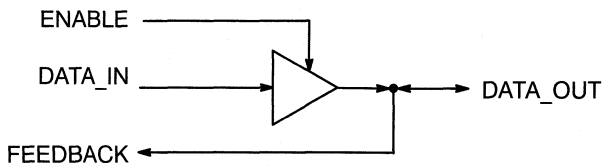


Figure 6. *bufoe* Component

```
Ux: bufoe PORT MAP(REG_int(i),read_oe,DATA(i),DATAIN(i));

read_oe <=   NOT(CSn) AND NOT(msk_reg) AND RD_WRn;
```

Figure 7. Port Map of *bufoe* for Interrupt Register

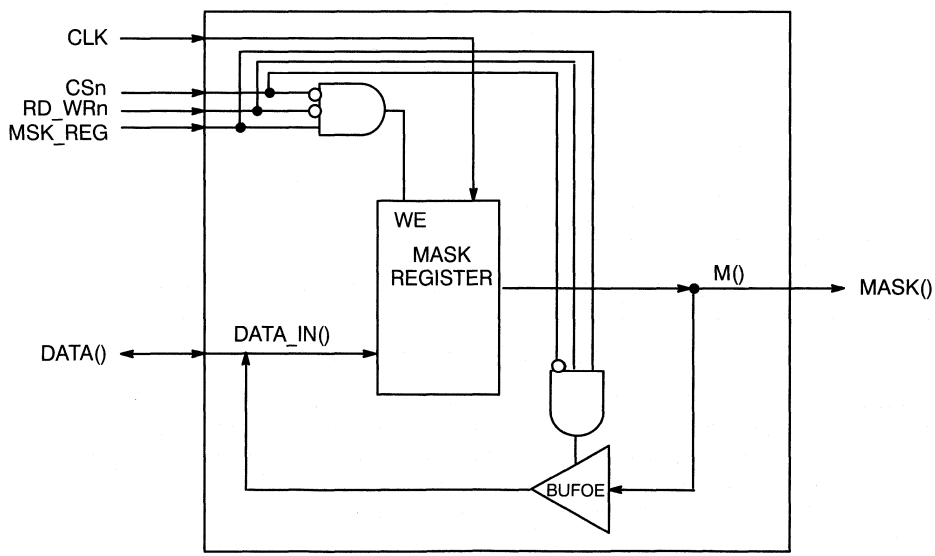


Figure 8. Mask Register

Once the mask bits are written to the Mask register, they need to be held until changed by another write or a reset (*Figure 10*).

When the mask registers are read (CSn -->L, RD_WRn, MSK_REG --> H), the DATA lines should take the values of MASK(14..1). Because there is no mask bit for interrupt 15, whenever the Mask register is read, bit 15 will return the value of the Interrupt register bit 15. Once again we will call on the *bufoe* component to handle the three-state busing (*Figure 11*).

M(i) are the inputs to the buffer, READ_OE is the active HIGH three-state control signal, DATA(i) are the names of the signals that connect to the pins of the part, and DATAIN(i) are the signals that feed back into the part. From this last set of equations it is easy to see that the Mask register and the Interrupt register blocks are very similar. The only difference in hooking to the buffer is the input signal. Let's modify the input signal to combine the signals into one buffer (*Figure 12*).

The new buffer enable signal is simply the old ones without the MSK_REG term (*Figure 13*).

Put altogether, the VHDL code that defines the Interrupt and Mask registers is as shown in *Figure 14*.

The next step is to implement the Mask Logic. This is simply the ANDing of the mask bits with the registered interrupts (*Figure 15*).

```
ENTITY mask_reg IS
  PORT (
    clk,CSn,RD_WRn,msk_reg :   IN BIT;
    DATA:                      INOUT x01z_VECTOR(14 DOWNTO 1);
    MASK:                      OUT BIT_VECTOR(15 DOWNTO 1)
    );
END mask_reg;
```

Figure 9. Mask Register Entity Statement

```
  M(i) <=  RESETn AND DATAIN(i) AND NOT(CSn) AND
NOT(RD_WRn) AND       msk_reg
    OR(RESETn AND M(i) AND (CSn OR RD_WRn OR
NOT(msk_reg)));
```

Figure 10. Mask Register Definition

```
Ux: bufoe PORT MAP(M(i),read_oe,DATA(i),DATAIN(i));

read_oe <=   RESETn AND NOT(CSn) AND msk_reg AND RD_WRn;
```

Figure 11. *bufoe* of the Mask Register

```
Buffer_in(i) <= (M(i) AND msk_reg) OR (REG_int(i) AND NOT(msk_reg));
```

Figure 12. Combining Mask and Interrupt Registers

```
read_oe <=   RESETn AND NOT(CSn) AND RD_WRn;
```

Figure 13. New Buffer Enable

```
ENTITY msk_reg IS PORT (
     clk,RD_WRn,CSn,RESETn,msk_reg: IN BIT;
     ITRPT: IN BIT_VECTOR(15 DOWNTO 1);
     DATA: INOUT x01z_VECTOR(15 DOWNTO 1);
     RI: OUT BIT_VECTOR(15 DOWNTO 1);
     MASK: OUT BIT_VECTOR(14 DOWNTO 1)
);
END msk_reg;

USE work.rtlpkg.all;

ARCHITECTURE ARCHmsk_reg OF msk_reg IS
SIGNAL DATAIN : BIT_VECTOR(15 DOWNTO 1);
SIGNAL REG_int: BIT_VECTOR(15 DOWNTO 1);
SIGNAL Buffer_in: BIT_VECTOR(15 DOWNTO 1);
SIGNAL M: BIT_VECTOR(14 DOWNTO 1);
SIGNAL read_oe: BIT;

  BEGIN
    PROCESS BEGIN
      WAIT UNTIL clk = '1';
L1:
      FOR i IN 1 TO 15 LOOP
        REG_int(i) <=   (ITRPT(i))
                     OR (DATAIN(i) AND NOT(CSn) AND
                         NOT(RD_WRn) AND NOT(msk_reg));
      END LOOP L1;
L2:
      FOR i IN 1 TO 14 LOOP
        M(i) <= ((RESETn AND DATAIN(i) AND NOT(CSn) AND NOT(RD_WRn) AND
msk_reg)
             OR  (RESETn AND M(i) AND (CSn OR RD_WRn OR NOT(msk_reg))));
      END LOOP L2;

    END process;
L3:
    FOR i IN 1 TO 14 GENERATE
      Buffer_in(i) <= (M(i) AND msk_reg) OR (REG_int(i) AND NOT(msk_reg));
    END GENERATE L3;

    Buffer_in(15) <= REG_int(15);

    read_oe      <=  RESETn AND NOT(CSn) AND RD_WRn;
```

Figure 14. Interrupt and Mask Register

```
    FOR i in 1 TO 15 GENERATE
      Ux: bufoe PORT MAP(Buffer_in(i),read_oe,DATA(i),DATAIN(i));
    END GENERATE;

    RI <= REG_INT;
    MASK <= M;

END ARCHmsk_reg;
```

Figure 14. Interrupt and Mask Register (continued)

```
  MI(i)    <= MASK(i) AND ITRPT(i);
  MI(15) <= ITRPT(15);     --(Remember, interrupt 15 is nonmaskable)
```

Figure 15. Mask Logic

If an active LOW interrupt is desired, the equation for that bit is:

```
MI(i) <= MASK(i) AND NOT(ITRPT(i));
```

From here we must design the Priority Encoder and Encoder. If a higher priority interrupt is present, It will override the interrupt currently being encoded. This is easily accomplished by using a state-table description, commonly known as a truth table. The ttf() function is available by including the statement

```
USE work.table_bv.all;
```

in the design file. Combining the mask logic and the priority encoder into one file produces the VHDL file shown in *Figure 16*.

The divide-by-two clock block shown in *Figure 1* actually uses the same clock as the rest of the design. The SLOWDOWN signal is used to only allow the values of ENC_ITR() to change every other clock.

Now that the lower-level pieces of the design are complete, it is time to put them together. First a package must be written for both sections of VHDL. This is easily accomplished by copying and then slightly modifying the Entity statements of both designs (*Figure 17*).

Each of these packages is placed at the top of their individual files and re-compiled. Now that the packages are in place, a file can be constructed that will connect the outputs of the Mask and Interrupt registers to the inputs of the Mask Logic and Priority Encoder (*Figure 18*).

The final task remaining is fitting the device into a PLD. From the entity of the design, we can see that the part needs 15 bidirectional pins (DATA[15..1]), one clock pin (CLK), 19 input pins (RESETn, RD_WRn, CSn, MSK_REG, ITRPT[15..1]), and four output pins (ENC_ITR[3..0]) for a total of 39 pins. The design also calls for 49 registers. The part that would give the maximum performance with minimum unused macrocells would be the CY7C373 (64 macrocells and I/O). This would still leave plenty of room to add features such as a counter or timer. The list of part families from Cypress that would also fit this design include the MAX™ family (CY7C342 or larger), the FLASH370 family (CY7C373 or larger), and the pASIC380 family (CY7C382 or larger).

The process of breaking a design into smaller, more manageable pieces is done quite easily in VHDL. The connection of the individual design segments by using packages makes it possible to have several people working on a design at once. In this example, one engineer could have been working on the Mask and Interrupt Register package, while another worked on the Priority Encoder and Mask Logic. This also simplifies the debug process. Each individual package can be tested by itself. After all packages are debugged, they can be hooked together for a first-pass error-free design.

```
ENTITY int_enc IS PORT(
  clk: IN BIT;
  MASK: IN BIT_VECTOR(14 DOWNTO 1);
  ITRPT: IN BIT_VECTOR(15 DOWNTO 1);
  ENC_ITR: OUT BIT_VECTOR(3 DOWNTO 0)
);
END int_enc;

USE work.table_bv.all;

ARCHITECTURE ARCHint_enc OF int_enc IS
  SIGNAL slowdown :      BIT;
  SIGNAL MI:                  BIT_VECTOR(15 DOWNTO 1);
  SIGNAL ENC_TMP1:  BIT_VECTOR(3 DOWNTO 0);
  SIGNAL ENC_TMP2:  BIT_VECTOR(3 DOWNTO 0);

  CONSTANT encoder: x01_table(0 TO 15,0 TO 18) := (
    "1xxxxxxxxxxxxxx" & "1111",
    "01xxxxxxxxxxxxx" & "1110",
    "001xxxxxxxxxxxx" & "1101",
    "0001xxxxxxxxxxx" & "1100",
    "00001xxxxxxxxxx" & "1011",
    "000001xxxxxxxxx" & "1010",
    "0000001xxxxxxxx" & "1001",
    "00000001xxxxxxx" & "1000",
    "000000001xxxxxx" & "0111",
    "0000000001xxxxx" & "0110",
    "00000000001xxxx" & "0101",
    "000000000001xxx" & "0100",
    "0000000000001xx" & "0011",
    "00000000000001x" & "0010",
    "000000000000001" & "0001",
    "000000000000000" & "0000");

BEGIN
  PROCESS BEGIN
    WAIT UNTIL clk = '1';

    FOR i IN 1 to 14 LOOP
      MI(i) <= (MASK(i) AND ITRPT(i));
    END LOOP;
    MI(15) <= ITRPT(15);   --(Remember, interrupt 15 is nonmaskable)

-- use divide by two clock for encoding
    slowdown <= NOT(slowdown);
    FOR i IN 0 to 3 LOOP
      ENC_TMP1(i) <= ((ENC_TMP2(i) AND slowdown)
                OR (ENC_TMP1(i) AND NOT(slowdown)));

    END LOOP;
  END PROCESS;
    ENC_TMP2 <= ttf(encoder,MI(15 DOWNTO 1));
    ENC_ITR <= ENC_TMP1;
END ARCHint_enc;
```

Figure 16. Interrupt Encoder

```
PACKAGE enc_pkg IS
  COMPONENT int_enc PORT(
      clk: IN BIT;
      MASK: IN BIT_VECTOR(14 DOWNTO 1);
      ITRPT: IN BIT_VECTOR(15 DOWNTO 1);
      ENC_ITR: OUT BIT_VECTOR(3 DOWNTO 0)
    );
  END COMPONENT;
END enc_pkg;

PACKAGE mskreg_pkg IS
  COMPONENT msk_itr PORT (
    clk,RD_WRn,CSn,RESETn,msk_reg: IN BIT;
    ITRPT: IN BIT_VECTOR(15 DOWNTO 1);
    DATA: INOUT x01z_VECTOR(15 DOWNTO 1);
    RI: OUT BIT_VECTOR(15 DOWNTO 1);
    MASK: OUT BIT_VECTOR(14 DOWNTO 1)
    );
  END COMPONENT;
END mskreg_pkg;
```

Figure 17. Package Statements

```
ENTITY int_ctl IS PORT (
     clk,RD_WRn,CSn,RESETn,msk_reg: IN BIT;
     ITRPT: IN BIT_VECTOR(15 DOWNTO 1);
     DATA: INOUT x01z_VECTOR(15 DOWNTO 1);
     ENC_ITR: OUT BIT_VECTOR(3 DOWNTO 0)
);
END int_ctl;

--This package contains the Mask Register and Interrupt Register
USE work.mskreg_pkg.all;
--

--This package contains the Mask logic and Priority Encoder
USE work.enc_pkg.all;
--

ARCHITECTURE ARCHint_ctl OF int_ctl IS
 COMPONENT msk_itr PORT (
    clk,RD_WRn,CSn,RESETn,msk_reg: IN BIT;
    ITRPT: IN BIT_VECTOR(15 DOWNTO 1);
    DATA: INOUT x01z_VECTOR(15 DOWNTO 1);
    RI: OUT BIT_VECTOR(15 DOWNTO 1);
    MASK: OUT BIT_VECTOR(14 DOWNTO 1)
    );
  END COMPONENT;
  COMPONENT int_enc PORT(
      clk: IN BIT;
      MASK: IN BIT_VECTOR(14 DOWNTO 1);
      ITRPT: IN BIT_VECTOR(15 DOWNTO 1);
      ENC_ITR: OUT BIT_VECTOR(3 DOWNTO 0)
    );
  END COMPONENT;

-- The  following signals are used to hook the two components together.
SIGNAL mbits : BIT_VECTOR(14 DOWNTO 1);
SIGNAL regbits : BIT_VECTOR(15 DOWNTO 1);

  BEGIN
    U1: msk_itr PORT MAP(
        clk,RD_WRn,CSn,RESETn,msk_reg,
        ITRPT,DATA,regbits,mbits);

    U2: int_enc PORT MAP(
        clk,mbits,regbits,ENC_ITR);

END ARCHint_ctl;
```

Figure 18. Combining the Packages

Document #: PLD0005−0194

MAX is a trademark of Altera Corporation.

Abel™ – HDL vs. IEEE – 1076 VHDL

Abstract

Currently there exist several popular Hardware Description Languages (HDLs) that allow designers to describe the function of complex logic circuits textually, as opposed to schematically. One of the most widely used of these languages is Data I/O's Abel–HDL. Abel–HDL, as a language, can be used to describe the behavior of logic circuits that can be fitted to a wide variety of PALs, PLDs, PROMs, and FPGAs from a variety of programmable logic IC manufacturers. IEEE–1076 VHDL (VHSIC Hardware Description Language) has recently been gaining widespread support. VHDL is an open, portable language defined and standardized by the IEEE that can be used to describe the behavior of an entire system from the highest levels of functionality all the way down to the logic-gate level. A majority of CAE vendors, programmable IC manufacturers, and third-party software vendors already have or are planning tools that support VHDL logic synthesis, logic modeling, and/or VHDL simulation.

The purpose of this application note is to compare and contrast the complexity and basic features of Abel–HDL with those of IEEE–1076 VHDL. Both of these languages are very robust in their support of different types of constructs that can be used to describe the same functionality at different levels of abstraction. It is beyond the scope of this document to exhaustively describe these possibilities or to present a complete tutorial for writing code in either language because of the great variety of constructs and syntax available with which to describe the functionality of a given circuit. Rather, a simple example design that contains a mixture of synchronous and asynchronous logic circuits will be shown. Sample code is written in both Abel–HDL and VHDL that describes the example's functionality and synthesizes to create functionally identical hardware. The code written here represents a typical level of abstraction that balances readability with compactness. With experience, designers can develop their own preferences for style. For instance, state machines can be described in a number of ways: state tables, IF-THEN-ELSE statements, CASE-WHEN statements, or explicitly using a combination of Register-Transfer-Level (RTL) code (individually describe each gate/register as a component with its inputs and outputs) and/or Boolean equations.

Example Description

The following example is a circuit that creates a 50% duty cycle clock with programmable frequency. *Figure 1* shows the block diagram of this Programmable Clock Generator. The output of the circuit is CLK_OUT, whose period is equal to clock(ns)*incnt*2. To program the device, LD_CNT is used to latch the value present at the INCNT(3:0) inputs into the 4-Bit Input Register. The output of this register is ENDCNT(3:0). The clock input is used to clock the 4-bit Up/Down Counter, whose output is COUNT(3:0). The 4-bit Comparator is an asynchronous comparator that compares the values of COUNT and ENDCNT. Its outputs are endeqcnt (ENDCNT = COUNT), endltcnt (ENDCNT < COUNT), endeq0 (ENDCNT = 0), and cnteq0 (COUNT = 0). Note that it is possible for ENDCNT to be less than COUNT if a new value for ENDCNT is loaded into the input registers that is less than the current value of COUNT. The cnt_state State Machine controls the CLK_OUT signal and is clocked and enabled by the clock and en inputs, respectively.

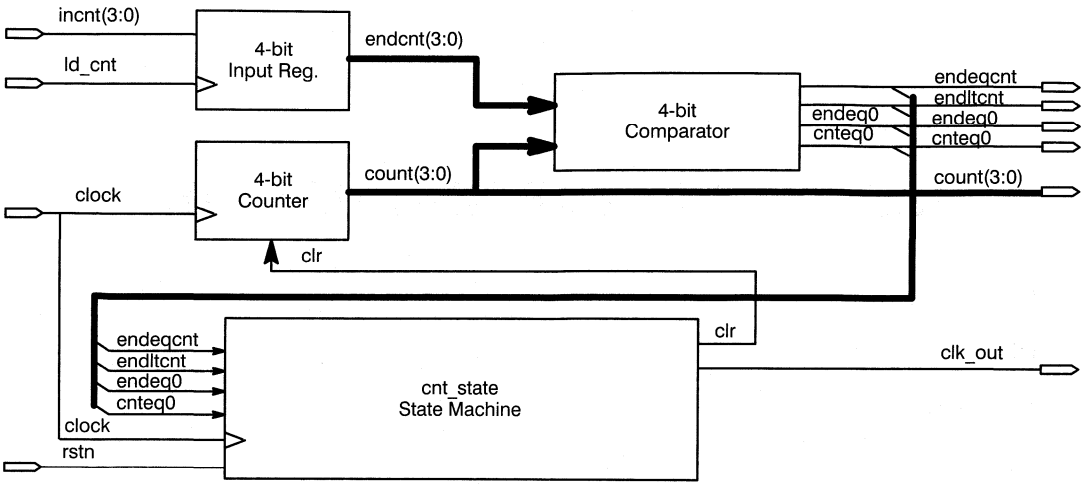


Figure 1. Block Diagram

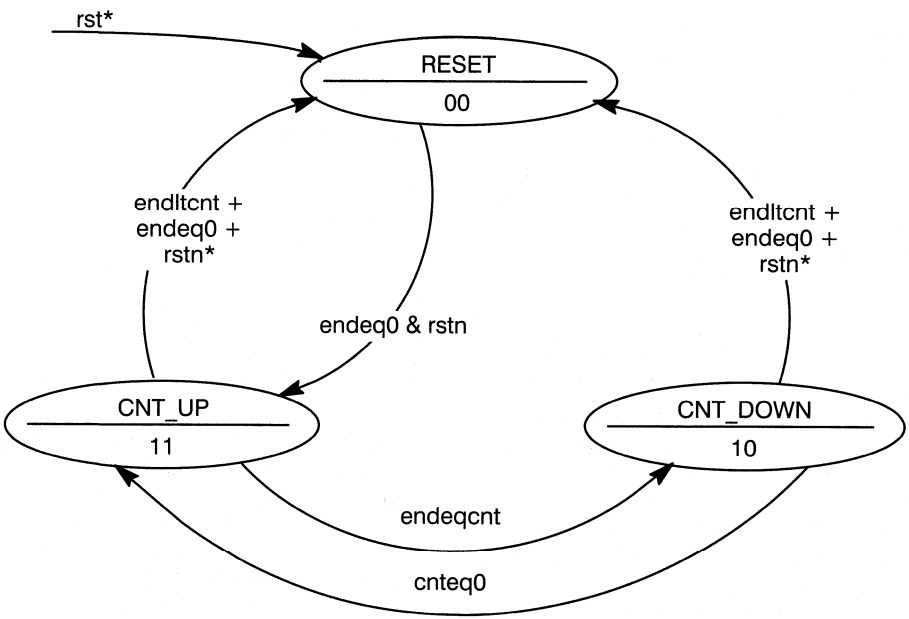


Figure 2. State Diagram

Figure 2 shows the state diagram for cnt_state. The state machine consists of three states: *reset*, *cnt_up*, and *cnt_down*. Two state bits are used to describe this Mealy-type state machine. The state machine powers-up to the reset state. It also synchronously enters the *reset* state from any state if RSTN=0. Once RSTN=1, the state machine will count up until COUNT equals ENDCNT, at which point it will begin to count down until COUNT equals 0 and then repeat the cycle. If at any time ENDCNT = 0, cnt_state will return to *reset*. Also, if ENDCNT is ever less than COUNT, thus signifying an invalid condition, cnt_state will go to *reset*.

The PLD targeted is the CY7C335. It was chosen because it can be used to illustrate a variety of features contained in some PLDs such as input registers, multiple clocks, buried registers, and synchronous reset/preset. However, any PLD with sufficient resources could be targeted. This is one of the main advantages of using a HDL. High-level languages, by design, allow a designer to write generic code that can be targeted to different devices/architectures with little or no modification. Going one step further, VHDL allows simulation and debugging of the logic from the source code. This can greatly reduce the overall design cycle time by allowing functional verification of the logic prior to targeting a specific physical device. Once the logic has been verified, the designer can then compile and fit the same design into a variety of devices. From here, the designer can decide which implementation best suits his requirements.

Abel−HDL vs. VHDL

In general, the constructs used to describe logic function in both ABEL−HDL an IEEE−1076 VHDL are very similar. Each can accept Boolean equations, truth tables, state descriptions (IF-THEN and CASE-WHEN), and signal assignments that are quite similar in appearance. However, differences do exist in syntax and in the overhead sections. These are the declarative sections which define hierarchical organization, design libraries, etc. As we shall see, some of these statements found in the VHDL code have no direct counterpart in Abel−HDL. This is because Abel−HDL was created with a single purpose in mind, logic synthesis for PALs and PLDs. Therefore, with Abel−HDL, assumptions can be made by the compiler which can simplify the overhead syntax needed. Because VHDL is a one-language-fits-all standard which applies to synthesis, modeling, and system definition at all levels, some syntax overhead is necessary to fully describe a design in the proper context. For example, the use of standard and user-definable packages and libraries allows many designs to share commonly used definitions, components, macros, functions, etc. Fortunately for the logic designer, these statements are used in most designs so that familiarity with them comes quickly and easily. This commonality also allows ample "cutting-and-pasting" from design to design.

The following sections compare and contrast the source code files for Abel−HDL and VHDL on a logical section-by-section basis. Both of these files, when compiled, create functionally identical logic. Copies of the source code files in their entirety can be found in the Appendices.

Design and I/O Declarations

The basic structure of both Abel−HDL and VHDL source files allow for one or more design units to be defined within it. Each design unit is a complete logic description. Multiple design units may be combined hierarchically in a single top-level source file which binds them together. In the example, we have used a single design unit for simplicity. The first section of code chosen for comparison contains the design declaration and device I/O declarations. In this example, the pin numbers have been fixed. This optional in either language. The declaration of the target device is also optional in the source code itself. The targeted device need not be declared until it's time to compile and fit the logic.

Figure 3 contains the Abel−HDL code that declares the module name (line 01), the device (line 03), and the input and output pin numbers and types (lines 04−09). Line 45 is the end statement that completes the design module.

The corresponding VHDL code is shown in *Figure 4*. VHDL requires a slightly different structure. A design unit consists of an Entity section and an Ar-

```
01: module clk_gena;

02: declarations
03: device 'p335';

    "Inputs
04:  clock, ld_cnt, rstn                pin 3,1,7;
05:  incnt3, incnt2, incnt1, incnt0     pin 6,5,4,2;

    "Outputs
06:  endeqcnt, endltcnt                 pin 25,23 istype 'com';
07:  endeq0, cnteq0                     pin 24,27 istype 'com';
08:  clk_out                            pin 17 istype 'reg_d';
09:  count3, count2, count1, count0     pin 19,15,28,26;
.
.
.
45: end clk_gena;
```

Figure 3. Abel−HDL Design and I/O Declarations

chitecture section. By themselves, each of these is considered a separate design component. The Entity section defines the component name (line 01). The port statement (lines 02−06) declares the I/O of the entity. For each signal, a mode (in, out, buffer) defines the direction of the pin (buffer signifies output with feedback). The signal type is also defined here. The type of a signal defines size and possible values which that signal can take on. Type bit defines a one-bit signal that can have the values of "0" and "1". Type bit_vector(0 to 3) declares that the signal is a 4-bit vector, each bit of which can take on the values of "0" or "1". Line 07 declares the target device and is optional. Lines 08−12 declares the fixed pin assignments and is also optional. Note that this line could be written as a single line terminated with a ";". The "&" in this context signifies a continuation from the previous line. Line 13 is the end statement that terminates the *clk_genv* entity.

Lines 14 and 15 are statements that call out other libraries and packages making them visible to this design. These libraries and packages may be predefined in the VHDL language or may be user defined. They may contain components, functions, procedures, declarations, etc., which may then be used by the current design. For instance, *work.int_math.all* enables all functions contained in the package *int_math* (integer math), which is found in the work library. These functions describe the operation of the "+" and "−" operators used in the up and down counter logic descriptions. The package *rtlpkg* contains the definition of the Global Synchronous Set (gss) statement used on line 27.

The second part of a complete design unit is the Architecture section. In this section is where we find the description of the behavior of the black box defined in the Entity section. Associated with the Architecture statement are begin and end statements. Line 16 declares the Architecture name, behave, for the following statements which describe the functionality of the Entity *clk_genv*. The Entity and Architecture sections are separated because VHDL allows multiple architectures to be defined for a given entity. Only one architecture can be associated with an entity in a given design. This feature allows multiple versions of an architecture to be saved in a library. The Configuration statement is used to select a specific architecture (see Reference 3).

```
01: entity clk_genv is
02: port(clock, ld_cnt, rstn      :in bit;
03:      incnt                    :in bit_vector(3 downto 0);
04:      count                    :buffer bit_vector(3 downto 0);
05:      endeqcnt, endltcnt, clk_out :buffer bit;
06:      endeq0, cnteq0           :buffer bit);

07: attribute part_name of clk_genv : entity is "c335";
08: attribute pin_numbers of clk_genv : entity is
09:  "clock:3 ld_cnt:1 rstn:7 clk_out:17 "
10:& "incnt(3):6 incnt(2):5 incnt(1):4 incnt(0):2 "
11:& "count(3):19 count(2):15 count(1):28 count(0):26 "
12:& "endeqcnt:25 endltcnt:23 endeq0:24 cnteq0:27";
13: end clk_genv;

14: use work.int_math.all;
15: use work.rtlpkg.all;

16: architecture behave of clk_genv is
.
.
.
22: begin
.
.
.
60: end behave;
```

Figure 4. VHDL Design and I/O Declarations

Internal Signal Declarations

In both Abel−HDL and VHDL, internal signals (nodes) may be defined. These signals do not connect directly to an input or an output pin and may result from buried logic or may simply represent a wire to transfer data. Both languages are similar in that the signal name and type are declared.

As can be seen in *Figure 5*, lines 10 and 11 of the Abel−HDL code declare the signals RST_CTR and ENDCNT3...0. Each is defined as type node, as opposed to type pin, which would mean that the signal

```
10:  rst_ctr                          node istype 'reg_d';
11:  endcnt3,endcnt2,endcnt1,endcnt0  node;
12:  incnt    = [incnt3,incnt2,incnt1,incnt0];
13:  count    = [count3,count2,count1,count0];
14:  endcnt   = [endcnt3,endcnt2,endcnt1,endcnt0];
15:  outputs = [count,endeqcnt,endltcnt,endeq0,cnteq0,clk_out];
16:  cnt_state = [rst_ctr,clk_out];
```

Figure 5. Abel−HDL Signal Declarations

```
17:  signal endcnt : bit_vector(3 downto 0);
18:  signal cnt_state : bit_vector(0 to 1);
```

Figure 6. VHDL Signal Declarations

would be connected to an I/O pin. An optional signal attribute may be used with the keyword istype to define the signal's characteristics more explicitly. Lines 12 – 16 show the groupings of signals into sets. Defining sets allows a group of signals to be referenced by one name. Any operation performed on the set name will be performed on each member of the set.

VHDL allows signal names that represent, among others, bit vectors such as the one shown in line 17 in *Figure 6*. Here ENDCNT is equivalent to [ENDCNT(3), ENDCNT(2), ENDCNT(1), ENDCNT(0)]. As seen earlier in the entity declaration, INCNT has been defined as a port (I/O pins) and is a 4-bit vector similar to ENDCNT. Individual signals cannot be declared and grouped into sets as with Abel−HDL. Rather, groups are declared initially as bit vectors. The individual members of the set can then be operated upon separately or as a group (line 58 of the VHDL source code shows cnt_state(1) being assigned to the output CLK_OUT).

State-Machine State Definitions

The section where state register assignments are declared is very similar for Abel−HDL and VHDL. Both languages require assignment of a constant value to a name which gets compared to the current value of the state bits (cnt_state) in the actual state machine implementation (IF-THEN-ELSE, CASE-WHEN). Shown in this document is one method of designing a state machine. Both Abel−HDL and VHDL allow a variety of ways in which to create a

```
17:  reset      = [0,0];
18:  cnt_up     = [1,1];
19:  cnt_down   = [1,0];
```

Figure 7. Abel−HDL State Machine Definition

state machine. For large state machines, a more compact implementation might be with a Truth-Table in which inputs and outputs are described in a tabular form. This method is more compact but some may find it less "readable" than other methods. Both languages also support Mealy, Moore, and one-hot (one register per state) state machine implementations. *Figure 7* shows the Abel−HDL state assignment code.

Figure 8 shows the VHDL code needed to make state assignments. Note here the increased verbosity relative to Abel−HDL. This, again, is due to the fact that VHDL is a more general-purpose language and that statements must be more explicit. Here, a constant is defined to be a certain type (bit_vector) and then is assigned a initial value using the ":=" operator. Note that VHDL's usage of the ":=" operator is different than its meaning as a registered signal assignment operator in Abel−HDL.

Combinatorial Logic Equations

Both Abel−HDL and VHDL allow combinatorial Boolean) logic equations. As *Figures 9* and *10* show, the syntax is quite similar. Combinatorial statements in Abel−HDL are signified by a "=" operator. Use of the *istype reg* attribute in the signal declaration section and/or use of the appropriate explicit

```
19:  constant reset : bit_vector(0 to 1) := "00";
20:  constant cnt_up : bit_vector(0 to 1) := "11";
21:  constant cnt_down : bit_vector(0 to 1) := "10";
```

Figure 8. VHDL State Machine Definition

```
20: equations
21:    endeqcnt = ((endcnt.fb - 1) == count.fb);
22:    endltcnt = (endcnt.fb < count.fb);
23:    endeq0 = (endcnt.fb == 0);
24:  cnteq0 = (count.fb == 1);
25:    outputs.sp = !rstn;
26:    count.clk = clock;
```

Figure 9. Abel−HDL Combinatorial Logic Equations

```
23: endeqcnt <= '1' when (count = (endcnt-1)) else '0';
24: endltcnt <= '1' when (endcnt < count) else '0';
25: endeq0   <= '1' when (endcnt = "0000") else '0';
26: cnteq0   <= '1' when (count = "0001") else '0';
27: gss <= NOT(rstn);
```

Figure 10. VHDL Combinatorial Logic Equations

signal extensions (.c, .q, .d, etc...) and the ":=" operator signifies registered logic. In VHDL, the syntax for both a combinatorial and registered signal assignment is the same, "<=". The difference being determined by where in the code the statement appears. It is treated as a registered signal only if the signal assignment statement occurs inside of a clocked process. (See References 1 and 3 for full explanation of processes.)

Since the CY7C335 used in this example, like many other devices, contains special features, such as global or individual resets, presets, or OEs, there needs to be a means to expressly access them. Lines 25 and 27 of the Abel−HDL and VHDL, respectively, show how to access the available global synchronous preset of the device. In the Abel−HDL code we have defined a set to be all of the output signals (see Line 14). By simply using the .sp extension, !rstn is assigned to the global synchronous preset signal. Note that it is not necessary to define a set. Since in this device the preset is global, by assigning !rstn to the preset of one output register, all are automatically connected. Line 26 assigns the signal clock to the counter registers.

In VHDL, since extensions are not allowed in signal assignments, special functions are created and placed in a standard package which access these specific device features. In this case, the gss (global synchronous set) function is found in the package *rtlpkg*. When using a function (or other statement) found in a package, a use statement must be added (Line 15) so that the contents of the package are visible to this design. This requirement may seem cumbersome on the surface, it in fact represents one of the most powerful advantages of using VHDL. It gives the ability to save and reuse commonly used statements in standard or user defined packages which can then be accessed by any design. These packages can in turn be placed in libraries which can be organized by function, project, etc.

Input Register Logic Definition

The CY7C335 was chosen for this example because it can be used to illustrate a variety of features that may be found in other programmable logic devices. In this section, we are making use of the registered inputs. The inputs are defined as INCNT(3:0). Once registered, these signals become ENDCNT(3:0) and are assigned to internal nodes in the device.

In the Abel_HDL code shown in *Figure 11*, Line 27 assigns the signal LD_CNT to be the clock for the ENDCNT registers. Line 28 uses the registered assignment operator, ":=", to create ENDCNT from INCNT.

```
27:  endcnt.clk = ld_cnt;
28:  endcnt := incnt;
```

Figure 11. Abel−HDL Input Register Definition

In VHDL, to create registered logic, a process must be used. This highlights a key concept in VHDL, the notion of concurrent vs. sequential statements. All concurrent statements are continuously and simultaneously evaluated, creating combinatorial logic. Sequential statements, as the name implies, are evaluated in order. The IF-THEN-ELSE construct is a classic example of a sequential statement. In VHDL, only statements found within a process are sequential. The processes themselves are concurrent and are continuously evaluated at the same time as all other statements between the begin and end of an Architecture section. A process is awakened (i.e., evaluated) when a change occurs in the value of a signal that is sensitive to that process. Sensitive signals are defined by the use of a sensitivity list or a wait statement at the beginning of the process. In our example we have used the wait statement to awaken a process when the clock signal for the associated logic sees a rising edge (Lines 28−31 of *Figure 12*). Here, the *in_reg* process is evaluated when a rising edge occurs on LD_CNT. Inside the process statements are evaluated sequentially. In this process there is a single statement which, when evaluated, causes the value of INCNT to be transferred to ENDCNT. This effectively creates a register clocked by LD_CNT.

State-Machine Description

The description of the simple three-state state machine (see *Figure 2*) is shown next for both Abel−HDL and VHDL in *Figures 13* and *14*, respectively. In general, both languages allow a variety of state machine definition methods. Included are truth tables, IF-THEN-ELSE statements, and CASE-WHEN statements. Both implementations require a state machine name declaration, clock declaration, and state descriptions.

```
28: in_reg : process begin
29:      wait until (ld_cnt = '1');
30:        endcnt <= incnt;
31: end process;
```

Figure 12. VHDL Input Register Definition

```
29:  cnt_state.clk = clock;
30:  state_diagram cnt_state
31:       state reset:
32:                 count := 0000;
33:            if (endeq0) then reset;
34:            else cnt_up;

35:       state cnt_up:
36:                 count := (count.fb + 1);
37:            if (endltcnt # endeq0) then reset;
38:            else if (endeqcnt) then cnt_down;
39:            else cnt_up;

40:       state cnt_down:
41:                 count := (count.fb - 1);
42:            if (endltcnt # endeq0) then reset;
43:            else if (cnteq0) then cnt_up;
44:            else cnt_down;
```

Figure 13. Abel−HDL State Machine Equations

```
32: counter : process begin
33:  wait until (clock = '1');
34:       case cnt_state is
35:       when reset =>
36:         count <= "0000";
37:       if (endeq0 = '0') then cnt_state <= cnt_up;
38:       else cnt_state <= reset;
39:       end if;

40:       when cnt_up =>
41:      count <= count +1;
42:       if (endltcnt='1' OR endeq0='1') then
43:      cnt_state <= reset;
44:       elsif (endeqcnt = '1') then cnt_state <= cnt_down;
45:       else cnt_state <= cnt_up;
46:       end if;

47:       when cnt_down =>
48:      count <= count - 1;
49:       if (endltcnt='1' OR endeq0='1') then
50:      cnt_state <= reset;
51:       elsif (cnteq0 = '1') then cnt_state <= cnt_up;
52:       else cnt_state <= cnt_down;
53:       end if;

54:       when others =>
55:      count <= "0000";
56:      cnt_state <= reset;

57:       end case;
58: end process;
59:  clk_out <= cnt_state(1);
```

Figure 14. VHDL State Machine Equations

In the Abel−HDL code of *Figure 13*, line 29 declares the signal clock to be the clock source for the state registers. Line 30 declares the following state descriptions to be for the state machine cnt_state. Lines 31−44 are the descriptions for each of the three states. Within each state description can be found signal assignments and IF-THEN-ELSE statements defining the conditional next-state assignments.

Similar to Abel−HDL, the VHDL code of *Figure 14* contains a clock declaration on line 33 (the wait until statement implies clock is the register clock in this process), and a state machine declaration on line 34 (the case statement defines cnt_state as the state machine under evaluation). Lines 35−57 contain the individual state descriptions. Lines 32 and 58 declare the beginning and end of the process called counter. This explicit declaration of the beginning and end of processes is necessary because of the VHDL distinction between concurrent statements and sequential (within a process) statements. Note the addition of the "when others" statement. This is added to insure that the state machine can recover from an invalid (undefined) state. Lastly, line 59 as-

signs the value of the state bit cnt_state(1) to the output signal CLK_OUT. Had this statement been placed inside the process it would have been treated as a sequential statement and, therefore, would be registered. This would have caused a registered, or pipelined, delay to be added to CLK_OUT.

Summary

In summary, as design languages, Abel−HDL and IEEE−VHDL are really quite similar in complexity. Many experienced Abel−HDL users may perceive VHDL to be unnecessarily complicated. This may be true if one is limited to the smaller playing field covered by Abel−HDL. VHDL, on the other hand, covers a broader set of applications, such as full system-level description and simulation. The extra verbosity is minimal when compared to the extra functionality provided. For instance, VHDL allows true source code simulation, an easy migration path to ASICs (standard, portable language), and design with different types such as integers, enumerated types, records, etc. It also is truly device independent. For instance, falling edge clocks and XORs can be described behaviorally in VHDL whereas in Abel−HDL, the target device must be declared and specific fuses programmed to make use of these special features.

References

1. Cypress Semiconductor, Warp2 *User's Manual*, 1993.
2. Data I/O, *Abel Design Software User Manual*, 1990.
3. S. Mazor and P. Langstraat, *A Guide To VHDL*, Kluwer Academic Publishers, Norwell, MA, 1992.
4. Douglas L. Perry, VHDL Second Edition, McGraw-Hill Series, Computer Engineering, 1994.

Appendix A. VHDL Design File for Prog. Clock Generator

```
01: entity clk_genv is
02: port(clock, ld_cnt, rstn   :in bit;
03:      incnt                 :in bit_vector(3 downto 0);
04:      count                 :buffer bit_vector(3 downto 0);
05:      endeqcnt, endltcnt, clk_out  :buffer bit;
06:      endeq0, cnteq0    :buffer bit);
07: attribute part_name of clk_genv : entity is "c335";
08: attribute pin_numbers of clk_genv : entity is
09:    "clock:3 ld_cnt:1 rstn:7 clk_out:17 "
10: & "incnt(3):6 incnt(2):5 incnt(1):4 incnt(0):2 "
11: & "count(3):19 count(2):15 count(1):28 count(0):26 "
12: & "endeqcnt:25 endltcnt:23 endeq0:24 cnteq0:27";
13: end clk_genv;

14: use work.int_math.all;
15: use work.rtlpkg.all;

16: architecture behave of clk_genv is

17: signal endcnt : bit_vector(3 downto 0);
18: signal cnt_state : bit_vector(0 to 1);
19: constant reset : bit_vector(0 to 1) := "00";
20: constant cnt_up : bit_vector(0 to 1) := "11";
21: constant cnt_down : bit_vector(0 to 1) := "10";

22: begin

23: endeqcnt <= '1' when (count = (endcnt-1)) else '0';
24: endltcnt <= '1' when (endcnt < count) else '0';
25: endeq0   <= '1' when (endcnt = "0000") else '0';
26: cnteq0   <= '1' when (count = "0001") else '0';
27: gss <= NOT(rstn);

28: in_reg : process begin
29:     wait until (ld_cnt = '1');
30:        endcnt <= incnt;
31: end process;

32: counter : process begin
33:    wait until (clock = '1');
34:    case cnt_state is
35:    when reset =>
36:        count <= "0000";
37:      if (endeq0 = '0') then cnt_state <= cnt_up;
38:      else cnt_state <= reset;
39:      end if;
```

Appendix A. VHDL Design File for Prog. Clock Generator (continued)

```
40:     when cnt_up =>
41:       count <= count +1;
42:        if (endltcnt='1' OR endeq0='1') then
43:       cnt_state <= reset;
44:        elsif (endeqcnt = '1') then cnt_state <= cnt_down;
45:        else cnt_state <= cnt_up;
46:        end if;

47:  when cnt_down =>
48:       count <= count - 1;
49:        if (endltcnt='1' OR endeq0='1') then
50:       cnt_state <= reset;
51:        elsif (cnteq0 = '1') then cnt_state <= cnt_up;
52:        else cnt_state <= cnt_down;
53:        end if;

54:    when others =>
55:       count <= "0000";
56:       cnt_state <= reset;

57:     end case;
58:  end process;
59:  clk_out <= cnt_state(1);
60:  end behave;
```

Appendix B. Abel−HDL Design File for Prog. Clock Generator

```
01: module clk_gena;

02: declarations
03: device 'p335';

    "Inputs
04:  clock, ld_cnt, rstn        pin 3,1,7;
05:  incnt3, incnt2, incnt1, incnt     pin 6,5,4,2;

    "Outputs
06:  endeqcnt, endltcnt      pin 25,23 istype 'com';
07:  endeq0, cnteq0        pin 24,27 istype 'com';
08:  clk_out           pin 17 istype 'reg_d';
09:  count3, count2, count1, count0    pin 19,15,28,26;

10:  rst_ctr           node istype 'reg_d';
11:  endcnt3,endcnt2,endcnt1,endcnt    node;

12:  incnt   = [incnt3,incnt2,incnt1,incnt0];
13:  count   = [count3,count2,count1,count0];
14:  endcnt  = [endcnt3,endcnt2,endcnt1,endcnt0];
15:  outputs = [count,endeqcnt,endltcnt,endeq0,cnteq0,clk_out];

16:  cnt_state = [rst_ctr,clk_out];
17:  reset     = [0,0];
18:  cnt_up    = [1,1];
19:  cnt_down  = [1,0];

20: equations

21:  endeqcnt = ((endcnt.fb - 1) == count.fb);
22:  endltcnt = (endcnt.fb < count.fb);
23:  endeq0 = (endcnt.fb == 0);
24:  cnteq0 = (count.fb == 1);
25:  outputs.sp = !rstn;
26:  count.clk = clock;
27:  endcnt.clk = ld_cnt;
28:  endcnt := incnt;

29:  cnt_state.clk = clock;
30:  state_diagram cnt_state
31:       state reset:
32:                 count := 0000;
33:            if (endeq0) then reset;
34:            else cnt_up;
```

Appendix B. Abel−HDL Design File for Prog. Clock Generator (continued)

```
35:       state cnt_up:
36:                 count := (count.fb + 1);
37:            if (endltcnt # endeq0) then reset;
38:            else if (endeqcnt) then cnt_down;
39:            else cnt_up;

40:       state cnt_down:
41:                 count := (count.fb - 1);
42:            if (endltcnt # endeq0) then reset;
43:            else if (cnteq0) then cnt_up;
44:            else cnt_down;
45: end clk_gena;
```

Document #: PLD0006−0194

Abel is a trademark of Data I/O Corporation.

VHDL Techniques for Optimal Design Fitting

This application note describes three simple rules that will help you when your design does not fit into the target device using *Warp*™. In other words, when you have coded your design, and the report file tells you there aren't sufficient resources in the device for your design, following these three rules should be your next step. These rules allow you to describe your design in a way that produces the most efficient, compact, logical description for your behavioral design.

The rules are as follows:

1. Complete IF−THEN−ELSE chains with an ELSE statement.
2. Complete a CASE statement with a WHEN OTHERS construct.
3. Assign "default" values to all output signals at the beginning of a process.

VHDL is a very powerful language that offers you the ability to create designs at different levels of abstraction. Its powerful descriptive capability and its independence from technology make it particularly well suited for logic synthesis. *Warp* supports a rich subset of VHDL for synthesis and is fully VHDL compliant for the subset of VHDL that it supports.

Warp's function is to produce optimal results for your design using the syntax and semantics of the VHDL language, as defined by the IEEE−1076 standard. As part of the semantic rules defined in this standard, it is possible under certain conditions for synthesis to produce a circuit that is functionally 100% accurate but not very efficient in terms of the logic generated. This may result in a design not "fitting" into the target device when it should. Following some simple rules when coding your design, however, will go a long way toward producing efficient results.

Signals and Implicit Memory

You will be dealing with signals of various types when describing your designs. Signals are objects that not only have a current value, but also have a past history of values; they "remember" their values from "last time." Thus, if signals are preceded by a conditional statement, they are never evaluated unless that conditional statement is true; if it is not true, they continue to hold their previous value. This makes perfect sense for a simulator, since simulators need to track where something was during the last "tick" of the simulation. *Warp* fulfills this VHDL semantic requirement by creating a latch to preserve the state of every signal in the circuit.

Implication and Solution

Also, there is no explicit concept of a register in VHDL. Instead, registers are inferred from a "wait until" synchronization statement at the beginning of a process. For example,

```
sync:process
  begin
    wait until clock = '1';
      foo <= fred;
  end process;
```

places a register after some combinatorial logic, where the register synchronizes the combinatorial logic with respect to another signal ("clock" in this case). In the example, "foo" is a signal that takes whatever value is in "fred" every time "clock" is on a rising edge. In this case, "foo" retains its past value since we have inferred a register from the "wait until" statement. The circuit produced by the *Warp*

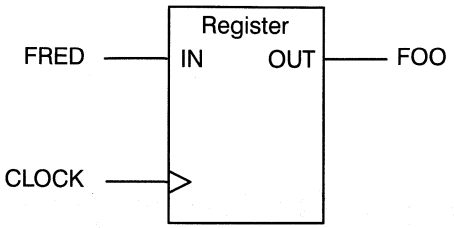


Figure 1. Register Circuit

compiler from this behavioral description is shown in *Figure 1*.

Observe what happens, though, when a conditional assignment is added to this example:

```
sync:process
  begin
  wait until clock = '1';
  if (jack = '1') then
    foo <= fred;
  end if;
end process
```

Some PLD languages assume that the value of "foo" is a don't care if "jack" is not asserted. This is not true of VHDL: "foo" only takes the value in "fred" if the signal "jack" is asserted. However, since "foo" is a signal, it has implicit memory: it remembers its past value. And because there is no concept of a register in VHDL (the register storing "foo" was inferred from the "wait until" statement), the *Warp* compiler creates a latch that stores the value of "foo" and only allows "foo" to get the value "fred" when "jack" is asserted. The resulting circuit would look like *Figure 2*.

Further along in the synthesis process, the above incomplete conditional assignment will produce extra product terms to build latches in the PLD due to the implicit memory of the signals. For example,

```
sync:process
  begin
  wait until clock = '1';
  if (jack='1' and jill='1') then
    foo <= fred;
  end if;
end process
```

results in the circuit in *Figure 3*.

As you can see, the incomplete conditional statement, especially ones that are complex, can very rapidly "blow-up" to a large circuit. It may turn out that all the extra latches result in more product terms or macrocells for a given signal than are available in a device, so the design does not fit.

The same applies to incompletely specified CASE statements. Consider the following example:

```
architecture BEHV of TEST is
signal chances: bit_vector(0 to 1);

  async: process
  begin
  case chances is
    when b"01"
      foo <= fred;
    end case;
  end process;
end BEHV;
```

The result is shown in *Figure 4*.

What this means is that any time you have an incomplete specification, you will generate a memory element. There is a very easy solution to this problem,

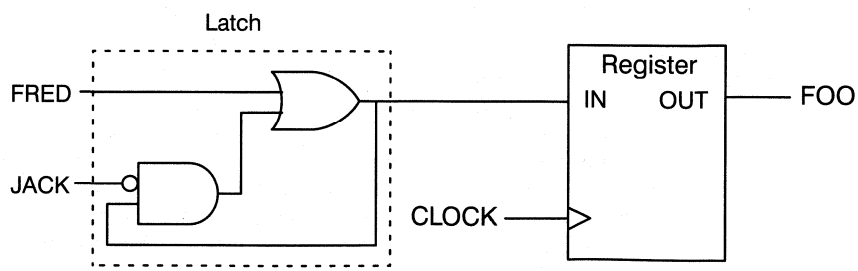


Figure 2. Implicit Memory Due to Incomplete IF Statement

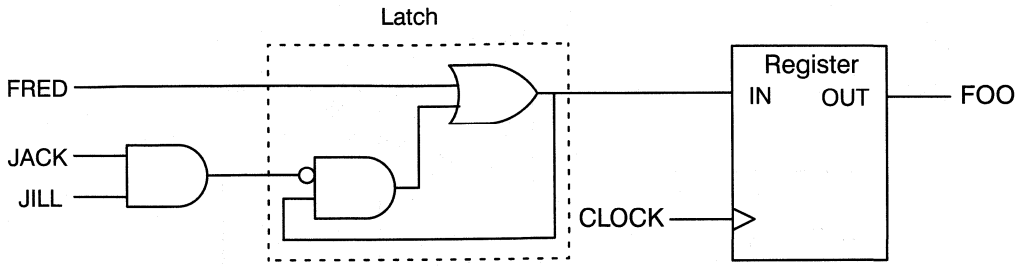


Figure 3. Implicit Memory Persists Through Complex Conditional Assignment

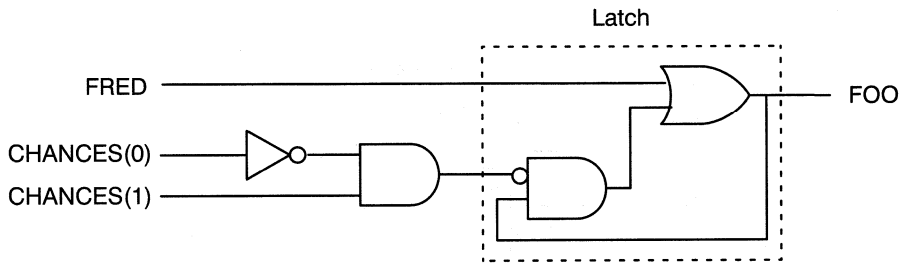


Figure 4. Implicit Memory with a CASE Statement

however: completely specify all conditional assignments in your design.

With a small amount of care, this produces excellent results. Let us consider the incomplete IF statement from an earlier example:

```
sync:process
begin
wait until clock = '1';
  if (jack='1' and jill='1') then
    foo <= fred;
  end if;
end process
```

By simply completing the conditional assignment as follows,

```
sync:process
begin
wait until clock = '1';
  if (jack='1' and jill='1') then
    foo <= fred;
  else
    foo <= '0';
  end if;
end process;
```

the circuit in *Figure 3* is reduced to the circuit shown in *Figure 5*.

We are now rid of the implicit latch, replacing it with a multiplexer. This is usually optimized out during the optimization stage of design process. We now utilize fewer product terms and/or macrocells because the latch (cross-coupled NAND gates) has been eliminated.

Similarly, completing the CASE statement from another of the above examples:

```
architecture BEHAV of MY_ENTITY is
signal chances: bit_vector(0 to 1);

  async: process
  begin
    case chances is
    when b"01"
      foo <= fred;
    when others =>
      foo <= '0';
    end case;
```

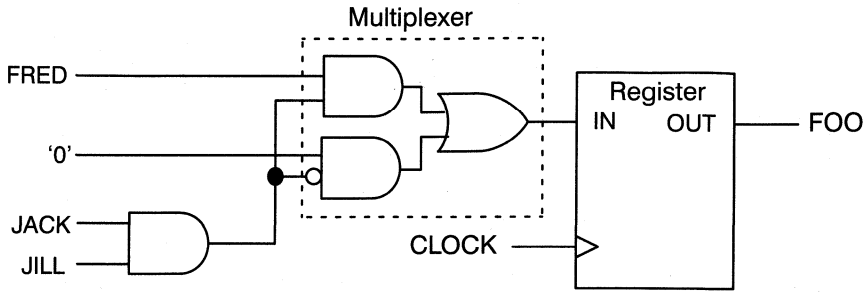


Figure 5. Implicit Memory Eliminated When IF Statement is Completed

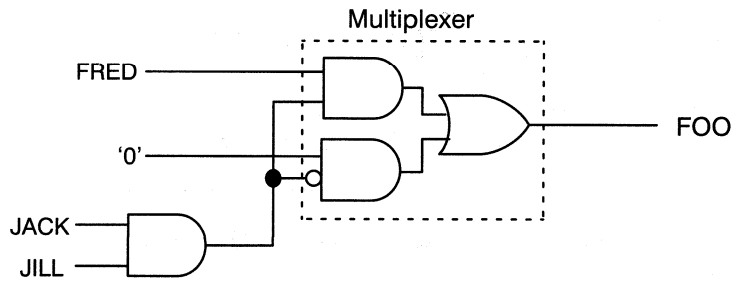


Figure 6. Implicit Memory Eliminated When CASE Statement is Completed

```
  end process;
end BEHAV;
```

results in the elimination of the latched memory element (as shown in *Figure 6*), and the optimizer can easily produce an efficient description of this circuit.

Another way in which you can "complete" the specification for a given signal is by creating a set of defaults at the beginning of your process. This will force the signals to be assigned this default value if the conditional expression that precedes it is not evaluated. So, we would modify our examples as follows:

```
sync:process
begin
wait until clock = '1';
  foo <= '0'; -- Assign default
              -- value;
  if (jack = '1') then
    foo <= fred;
  end if;
end process

architecture BEHAV of MY_ENTITY is
signal chances: bit_vector(0 to 1);

  async: process
  begin
    foo <= '0'; -- Assign default
                -- value;
  case chances is
    when b"01"
    foo <= fred;
    end case;
  end process;
end BEHAV;
```

This produces the same efficient circuitry that we saw earlier when we completed the conditional assignments in the IF statement and the CASE statement.

How should you choose the default value? Starting off with 0 usually produces good results. If you are not satisfied (or if you are superstitious), try 1. The

report file produced by *Warp* will let you quickly check the resource usage, so you should be able to make a determination quickly and easily.

In summary, signals are never evaluated unless there is a true evaluation of a conditional expression. Because they have implicit memory, they retain their past value when the conditional expression that precedes them is not evaluated. Incomplete conditional assignments will result in logically and semantically correct code, but inefficient circuitry. If you wish to describe the most efficient circuit possible, remember to follow the three rules described in this application note.

Document #: PLD0014–1092

Warp is a trademark of Cypress Semiconductor Corporation.

Describing State Machines with *Warp2*™ VHDL

Overview

VHDL (VHSIC Hardware Description Language) is a very rich, powerful programming language that offers designers the ability to describe designs at different levels of abstraction. *Warp2*™ utilizes a subset of VHDL as its design-entry language for PLDs and PROMs. This subset includes the ability to describe state machines using flexible but intuitive constructs. One result of this flexibility is that there is no single, correct way to describe a given design; the same design may be expressed in a number of ways, all yielding compiled designs with the same functionality.

The purpose of this application note is to introduce the reader to the methodology used to design state machines in *Warp2*. It first defines the basic types of state machines. A bubble diagram of a sample design is then presented and the procedure for translating it into *Warp2* is discussed. This same design is later modified to show alternate implementations for the same design as well as for the various types of state machines described. The intent is to present the reader with sample designs of each of the basic types of state machines using the most straight-forward and intuitive *Warp2* syntax.

It is assumed that the reader has read the *Warp2* User's Manual and is familiar with basic *Warp2* syntax, commands, and file structure. When studying the sample design files, keep in mind that the files presented are by no means the only possible implementations of the designs.

Device-Independent Design

Because *Warp2* allows device-independent design, the target device chosen for the sample designs is not relevant to this application note. The Cypress CY7C335 Universal Synchronous EPLD was arbitrarily chosen as the target device, but the discussion and the Design Entity files could also apply to any PLD or PROM used to implement a state machine. Because the same device is used for all of the examples (and all of the examples have the same set of inputs and outputs), the Binding Architecture files are identical except for the names of the package and the component. The Binding Architecture files for each of the designs are included in the appendices.

Moore Machine

A state machine in which all outputs are synchronous is called a Moore machine. As shown in *Figure 1*, it is characterized by the fact that each output of the circuit is a register whose input is a function of the current state of the machine and the inputs. The outputs are synchronously clocked with the state registers.

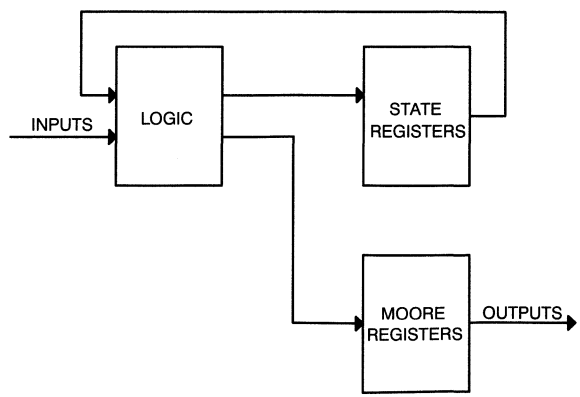


Figure 1. Moore Machine Block Diagram

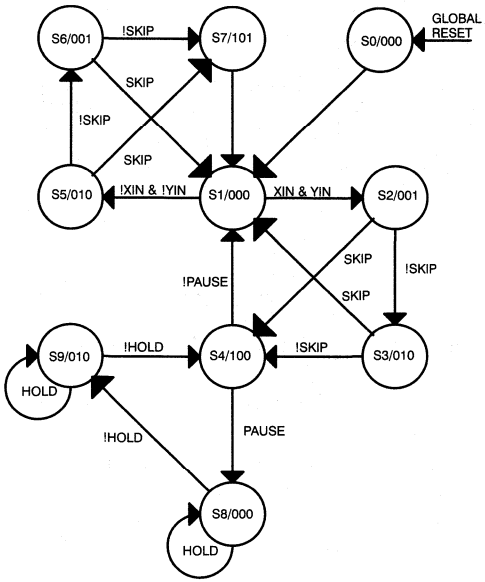


Figure 2. State Machine Bubble Diagram

Figure 2 is a bubble diagram of a sample state machine containing 5 inputs (plus a global reset), 3 Moore outputs, and 10 states. The outputs are shown inside the bubbles of the corresponding states and are, from left to right, OUT2, OUT1 and OUT0. Although omitted from the diagram to simplify reading, all combinations of inputs not shown on transitions are assumed to cause the machine to stay in the same state. For example, when in state S1, if X_{IN} and Y_{IN} are not either both 0 or both 1, the machine remains in state S1. All other inputs are considered to be don't cares for this state.

The Design Entity file for an implementation of this design is shown in Appendix A. The package declaration creates a package and places the components listed into it. The component and entity declarations simply list the port map, and they are the same for all designs in this application note. We will discuss how the architecture is described.

A CONSTANT is assigned for each of the 10 states. This allows the operation of the machine to be described using state names, and allows the state bit assignments to be modified easily if necessary. The assignment of the 4 state bits is up to the designer. In this note it is done in binary order. The only restriction is that since the Global Synchronous Reset is defined in the Binding Architecture file (gsr <= pin2), the reset state S0 must be defined as state 0000.

The remainder of the architecture is a process that is sensitive to a clock containing a Case structure. For each state, the transitions out of that state and the new values of the outputs are specified. Essentially, there is an IF clause for each outgoing transition (represented by an arrow) from a state in which the new value of the signal STATE is assigned as well as the new values of the outputs. Note that an ELSE clause is always included to handle the remaining cases mentioned above when the machine remains in the same state. This ELSE must be included to make the IF statement exhaustive, because the compiler will not make this assumption. If a given transition is always made (a single arrow with no conditions), the IF is not needed and the next state and outputs are simply assigned.

Since all of these assignments are within a process that is sensitive to the clock, they will all occur on the rising edge of that clock, making this implementation a Moore machine. Notice that just as the ELSE is required, a WHEN OTHERS statement is also required to make the Case statement exhaustive. In this example, an illegal state will transition back to S0.

Compact Form

While the above example is very readable, it is also somewhat verbose. The new values of the outputs must be specified for each state transition. For a machine of this type with only Moore-type outputs, a more compact method is available. An implementation of this method using the same bubble diagram is shown in Appendix C.

In this implementation, we've taken advantage of the fact that whenever the state machine is in a given state, the outputs always have the same values regardless of which transition was made to get there. The state CONSTANTs are now 7 bits wide; the 4 left-most bits are the state bits, and the 3 right bits are the Moore outputs. The process is the same as

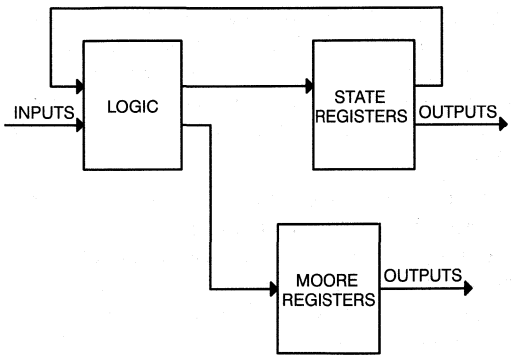


Figure 3. Moore Machine with States Used as Outputs

in the previous example, except that the assignments of the outputs are deleted because they are contained in the next-state assignments. These assignments are all still within a synchronous process and will occur on the rising edge of the clock.

The only remaining requirement is the three concurrent assignments at the end which assign the appropriate bits to the output pins. Since these are concurrent assignments outside of the process, they simply cause the state bits to become outputs of the device. This design produces exactly the same state machine operation as the previous one.

This method of implementation is not only more compact, it also allows registers to act as both state registers and Moore output registers, creating the type of Moore machine shown in *Figure 3*. By modifying the CONSTANT assignments, some of the state bits could be removed as long as the bit vectors are still unique. This would yield a state machine that has the same functionality but uses fewer macrocell registers. However, because fewer registers are used, each register might require more product terms, and this type of design would be more likely to run out of product terms in the targeted device.

Mealy Machine

Figure 4 is a block diagram of a Mealy state machine. The only difference between this and the basic Moore machine is that the outputs are now asynchronous rather than registered. In this type of a machine, an input can cause an output to transition immediately rather than waiting for the next clock edge. An example where this is useful is a cache controller state machine. Often times such a state machine will be in a state waiting for a HIT or MISS signal from a cache tag RAM. A Mealy machine is able to respond to a MISS signal and assert a WAIT signal to a microprocessor within the same clock cycle.

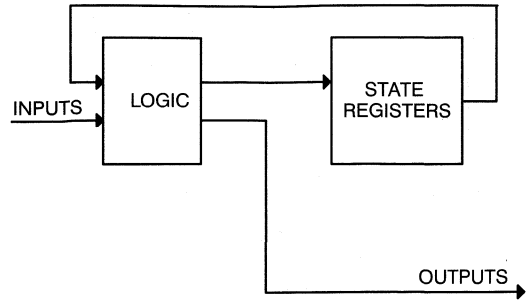


Figure 4. Mealy Machine Block Diagram

The same bubble diagram will again be used as the example, but here as a Mealy machine. Although it is not shown, it is customary to label the Mealy outputs along the transition arrows rather than in the state bubbles since Mealy outputs are a function of the present state and the inputs, but not of the next state.

The Design Entity file is shown in Appendix E. The CONSTANT assignments are the same as in the first example. The primary difference in this example is that two processes are required—one that is sensitive to a clock and the other sensitive to the input signals. The Case structure derived from the bubble diagram is contained in *proc2*. Notice that a new signal, NEXTSTATE, is used to hold the asynchronous version of the next-state assignment. In *proc1* the next-state assignment is made synchronously. In this way the state register transitions occur on the rising edge of the clock, while the output assignments remain combinatorial or asynchronous.

This example could also be made more compact using the procedure described in Appendix C.

Combining Moore and Mealy Outputs

Many real-world designs require that a single state machine contain both Moore and Mealy type state outputs. To demonstrate how this is accomplished, assume in our example that outputs OUT0 and OUT1 must be Mealy-type while OUT2 is Moore-type. This Design Entity is shown in Appendix G.

As in the Mealy machine, two processes are required—one sensitive to the clock and one not. Note that in the asynchronous process, a new signal NEXTOUT2 is used to hold the combinatorial version of OUT2, the Moore output. In *proc1* the output is assigned to OUT2 synchronously with the state assignment.

This allows OUT2 to be clocked synchronously with the state registers, while the other two outputs remain Mealy-type.

State Table Descriptions

All of the sample design architectures described thus far have had Case statements using what is called the explicit statement method. Another popular method of describing state machines uses state tables to specify the next-state and output transitions for a state machine. *Warp2* also supports this method. This section uses the same example and describes how to implement a Moore machine, Mealy machine, and a combination of both using the state table method.

The Moore machine Design Entity file is shown in Appendix I. The statement

```
use work.table_bv.all;
```

is added to the file. This is analogous to an Include statement in the C programming language and it is necessary to allow use of the package *table_bv*, which includes the truth table function, ttf().

The architecture contains a new signal called TABLE_OUT which is 7 bits wide in order to hold the 4 next-state bits and 3 outputs read from the truth table described later. A CONSTANT is assigned for each of the states as before. There is also a new CONSTANT called TABLE, which contains the state transition table. This table is constructed directly from the bubble diagram with one line for each transition arrow in the state machine. Each line contains, from left to right, present state, inputs, next state, and outputs. Note that for state S1, entries are required to handle the input combinations that cause the machine to remain in S1. This serves the same purpose as the ELSE in the Case statement example. Although each line is simply a 16-bit string literal, they are broken into columns separated by the concatenation operator to improve readability.

As in the other Moore machine (described in Appendix A), only a single process is needed. The process contains a call to the truth table function ttf() to look up a line in the table. The function is passed the name of the table (TABLE), the current state of the machine (TABLE_OUT(0 TO 3)), and the inputs. The truth table function finds the line that begins with those values and assigns TABLE_OUT to the next-state and outputs at the end of that line.

Since the process is sensitive to a clock, each of these state registers and outputs is updated at its rising edge.

Finally, three concurrent assignments are used to assign the appropriate bits of the table to the output pins. Since these are concurrent assignments outside of the process, they simply cause the table bits to become outputs of the device. This design produces exactly the same state machine operation as the one shown in Appendices A and C.

Appendix K shows the Mealy machine example using the truth table method. The key difference is that the truth table function look-up takes place concurrently outside of the process, and the next state and outputs are assigned to the new signal NEXTTABLE_OUT. These values are synchronously assigned to TABLE_OUT in a process that is sensitive to a clock. This allows the state registers to be clocked while the outputs are Mealy-type since they are driven by the asynchronous signal NEXTTABLE_OUT. This design produces exactly the same state machine operation as the one shown in Appendix E.

Finally, Appendix M shows the Design Entity file for a combination of output types using the state table method. This example is identical to the pre-

vious one except that the Moore output, OUT2, is assigned synchronously within the process rather than asynchronously as a concurrent assignment. This design produces exactly the same state machine operation as the one shown in Appendix G.

Any state machine can be expressed in either the Case statement form or the state table form and the choice is primarily one of personal preference. The state table method tends to be more compact, while the Case statement method is often more readable.

Summary

This application note describes the methodology used to create a *Warp2* Design Entity file for each of the types of state machines using either explicit statements or state tables. Using these examples, it should be simple matter to translate any state machine bubble diagram into a *Warp2* design. This design can then be targeted to any Cypress registered PLD or state machine PROM.

Warp2 is a trademark of Cypress Semiconductor Corporation.

Appendix A. Design Entity File for Moore Machine

```
package moore_pkg is
        component moore
        port (clock, xin, yin, skip, pause, hold : in bit;
                out0, out1, out2 : out bit);
        end component;
end moore_pkg;
entity moore is
        port (clock, xin, yin, skip, pause, hold : in bit;
                out0, out1, out2 : out bit);
end moore;
architecture fsm of moore is
        signal state:bit_vector(0 to 3);
        constant s0: bit_vector(0 to 3) := "0000";
        constant s1: bit_vector(0 to 3) := "0001";
        constant s2: bit_vector(0 to 3) := "0010";
        constant s3: bit_vector(0 to 3) := "0011";
        constant s4: bit_vector(0 to 3) := "0100";
        constant s5: bit_vector(0 to 3) := "0101";
        constant s6: bit_vector(0 to 3) := "0110";
        constant s7: bit_vector(0 to 3) := "0111";
        constant s8: bit_vector(0 to 3) := "1000";
        constant s9: bit_vector(0 to 3) := "1001";

        begin
                process
                        begin
                        wait until clock = '1';
                        case state is
                                when s0 =>
                                        state <= s1;
                                        out0 <= '0';
                                        out1 <= '0';
                                        out2 <= '0';
                                when s1 =>
                                        if (xin = '1' and yin = '1') then
                                                state <= s2;
                                                out0 <= '1';
```

Appendix A. Design Entity File for Moore Machine (continued)

```
                                out1 <= '0';
                                out2 <= '0';
                        elsif (xin = '0' and yin = '0')
then
                                state <= s5;
                                out0 <= '0';
                                out1 <= '1';
                                out2 <= '0';
                        else
                                state <= s1;
                                out0 <= '0';
                                out1 <= '0';
                                out2 <= '0';
                        end if;
                when s2 =>
                        if (skip = '1') then
                                state <= s4;
                                out0 <= '0';
                                out1 <= '0';
                                out2 <= '1';
                        else -- skip = '0'
                                state <= s3;
                                out0 <= '0';
                                out1 <= '1';
                                out2 <= '0';
                        end if;
                when s3 =>
                        if (skip = '1') then
                                state <= s1;
                                out0 <= '0';
                                out1 <= '0';
                                out2 <= '0';
                        else -- skip = '0'
                                state <= s4;
                                out0 <= '0';
                                out1 <= '0';
                                out2 <= '1';
```

Appendix A. Design Entity File for Moore Machine (continued)

```
                end if;
        when s4 =>
                if (pause = '1') then
                        state <= s8;
                        out0 <= '0';
                        out1 <= '0';
                        out2 <= '0';
                else -- pause = '0'
                        state <= s1;
                        out0 <= '0';
                        out1 <= '0';
                        out2 <= '0';
                end if;
        when s5 =>
                if (skip = '1') then
                        state <= s7;
                        out0 <= '1';
                        out1 <= '0';
                        out2 <= '1';
                else -- skip = '0'
                        state <= s6;
                        out0 <= '1';
                        out1 <= '0';
                        out2 <= '0';
                end if;
        when s6 =>
                if (skip = '1') then
                        state <= s1;
                        out0 <= '0';
                        out1 <= '0';
                        out2 <= '0';
                else -- skip = '0'
                        state <= s7;
                        out0 <= '1';
                        out1 <= '0';
                        out2 <= '1';
                end if;
```

Appendix A. Design Entity File for Moore Machine (continued)

```
                        when s7 =>
                                state <= s1;
                                out0 <= '0';
                                out1 <= '0';
                                out2 <= '0';
                        when s8 =>
                                if (hold = '1') then
                                        state <= s8;
                                        out0 <= '0';
                                        out1 <= '0';
                                        out2 <= '0';
                                else -- hold = '0'
                                        state <= s9;
                                        out0 <= '0';
                                        out1 <= '1';
                                        out2 <= '0';
                                end if;
                        when s9 =>
                                if (hold = '1') then
                                        state <= s9;
                                        out0 <= '0';
                                        out1 <= '1';
                                        out2 <= '0';
                                else -- hold = '0'
                                        state <= s4;
                                        out0 <= '0';
                                        out1 <= '0';
                                        out2 <= '1';
                                end if;
                        when others =>
                                state <= s0;
                                out0 <= '0';
                                out1 <= '0';
                                out2 <= '0';
                end case;

        end process;
end fsm;
```

Appendix B. Binding Architecture File for Moore Machine

```
--Binding Architecture
use work.cypress.all;
use work.rtlpkg.all;
use work.moore_pkg.all;

ARCHITECTURE appnote OF c335 IS

BEGIN
        gsr <= pin2;
U1 : moore PORT MAP (
        xin => pin3,
        yin => pin4,
        skip => pin5,
        pause => pin6,
        hold => pin7,
        clock => pin1,
        fbx(out0) => pin28,
        fbx(out1) => pin27,
        fbx(out2) => pin26
     );

END appnote;
```

Appendix C. Design Entity File for Compact Moore Machine

```
package moore2_pkg is
        component moore2
        port (clock, xin, yin, skip, pause, hold : in bit;
                out0, out1, out2 : out bit);
        end component;
end moore2_pkg;
entity moore2 is
        port (clock, xin, yin, skip, pause, hold : in bit;
                out0, out1, out2 : out bit);
end moore2;
architecture fsm of moore2 is
        signal state:bit_vector(6 downto 0);
        constant s0: bit_vector(6 downto 0) := "0000" & "000";
        constant s1: bit_vector(6 downto 0) := "0001" & "000";
        constant s2: bit_vector(6 downto 0) := "0010" & "001";
        constant s3: bit_vector(6 downto 0) := "0011" & "010";
        constant s4: bit_vector(6 downto 0) := "0100" & "100";
        constant s5: bit_vector(6 downto 0) := "0101" & "010";
        constant s6: bit_vector(6 downto 0) := "0110" & "001";
        constant s7: bit_vector(6 downto 0) := "0111" & "101";
        constant s8: bit_vector(6 downto 0) := "1000" & "000";
        constant s9: bit_vector(6 downto 0) := "1001" & "010";

        begin
                process
                        begin
                        wait until clock = '1';
                        case state is
                                when s0 =>
                                        state <= s1;
                                when s1 =>
                                        if (xin = '1' and yin = '1') then
                                                state <= s2;
                                        elsif (xin = '0' and yin = '0')
then
                                                state <= s5;
                                        else
                                                state <= s1;
                                        end if;
                                when s2 =>
```

Appendix C. Design Entity File for Compact Moore Machine (continued)

```
                if (skip = '1') then
                        state <= s4;
                else -- skip = '0'
                        state <= s3;
                end if;
        when s3 =>
                if (skip = '1') then
                        state <= s1;
                else -- skip = '0'
                        state <= s4;
                end if;
        when s4 =>
                if (pause = '1') then
                        state <= s8;
                else -- pause = '0'
                        state <= s1;
                end if;
        when s5 =>
                if (skip = '1') then
                        state <= s7;
                else -- skip = '0'
                        state <= s6;
                end if;
        when s6 =>
                if (skip = '1') then
                        state <= s1;
                else -- skip = '0'
                        state <= s7;
                end if;
        when s7 =>
                state <= s1;
        when s8 =>
                if (hold = '1') then
                        state <= s8;
                else -- hold = '0'
                        state <= s9;
                end if;
        when s9 =>
                if (hold = '1') then
                        state <= s9;
```

Appendix C. Design Entity File for Compact Moore Machine (continued)

```
                                    else -- hold = '0'
                                            state <= s4;
                                    end if;
                            when others =>
                                    state <= s0;
                    end case;

        end process;
        out0 <= state(0);
        out1 <= state(1);
        out2 <= state(2);
end fsm;
```

Appendix D. Binding Architecture File for Compact Moore Machine

```
--Binding Architecture
use work.cypress.all;
use work.rtlpkg.all;
use work.moore2_pkg.all;

ARCHITECTURE appnote OF c335 IS

BEGIN
        gsr <= pin2;
U1 : moore2 PORT MAP (
        xin => pin3,
        yin => pin4,
        skip => pin5,
        pause => pin6,
        hold => pin7,
        clock => pin1,
        fbx(out0) => pin28,
        fbx(out1) => pin27,
        fbx(out2) => pin26
     );

END appnote;
```

Appendix E. Design Entity File for Mealy Machine

```
package mealy_pkg is
        component mealy
        port (clock, xin, yin, skip, pause, hold : in bit;
                out0, out1, out2 : out bit);
        end component;
end mealy_pkg;
entity mealy is
        port (clock, xin, yin, skip, pause, hold : in bit;
                out0, out1, out2 : out bit);
end mealy;
architecture fsm of mealy is
        signal state:bit_vector(0 to 3);
        signal nextstate:bit_vector(0 to 3);
        constant s0: bit_vector(0 to 3) := "0000";
        constant s1: bit_vector(0 to 3) := "0001";
        constant s2: bit_vector(0 to 3) := "0010";
        constant s3: bit_vector(0 to 3) := "0011";
        constant s4: bit_vector(0 to 3) := "0100";
        constant s5: bit_vector(0 to 3) := "0101";
        constant s6: bit_vector(0 to 3) := "0110";
        constant s7: bit_vector(0 to 3) := "0111";
        constant s8: bit_vector(0 to 3) := "1000";
        constant s9: bit_vector(0 to 3) := "1001";

        begin
                proc1:process
                        begin
                        wait until clock = '1';
                        state <= nextstate;
                end process;

                proc2:process (xin, yin, skip, pause, hold)
                        begin
                        case state is
                                when s0 =>
                                        nextstate <= s1;
                                        out0 <= '0';
                                        out1 <= '0';
                                        out2 <= '0';
                                when s1 =>
```

Appendix E. Design Entity File for Mealy Machine (continued)

```
                        if (xin = '1' and yin = '1') then
                                nextstate <= s2;
                                out0 <= '1';
                                out1 <= '0';
                                out2 <= '0';
                        elsif (xin = '0' and yin = '0')
then
                                nextstate <= s5;
                                out0 <= '0';
                                out1 <= '1';
                                out2 <= '0';
                        else
                                nextstate <= s1;
                                out0 <= '0';
                                out1 <= '0';
                                out2 <= '0';
                        end if;
                when s2 =>
                        if (skip = '1') then
                                nextstate <= s4;
                                out0 <= '0';
                                out1 <= '0';
                                out2 <= '1';
                        else -- skip = '0'
                                nextstate <= s3;
                                out0 <= '0';
                                out1 <= '1';
                                out2 <= '0';
                        end if;
                when s3 =>
                        if (skip = '1') then
                                nextstate <= s1;
                                out0 <= '0';
                                out1 <= '0';
                                out2 <= '0';
                        else -- skip = '0'
                                nextstate <= s4;
                                out0 <= '0';
                                out1 <= '0';
                                out2 <= '1';
```

Appendix E. Design Entity File for Mealy Machine (continued)

```
                end if;
        when s4 =>
                if (pause = '1') then
                        nextstate <= s8;
                        out0 <= '0';
                        out1 <= '0';
                        out2 <= '0';
                else -- pause = '0'
                        nextstate <= s1;
                        out0 <= '0';
                        out1 <= '0';
                        out2 <= '0';
                end if;
        when s5 =>
                if (skip = '1') then
                        nextstate <= s7;
                        out0 <= '1';
                        out1 <= '0';
                        out2 <= '1';
                else -- skip = '0'
                        nextstate <= s6;
                        out0 <= '1';
                        out1 <= '0';
                        out2 <= '0';
                end if;
        when s6 =>
                if (skip = '1') then
                        nextstate <= s1;
                        out0 <= '0';
                        out1 <= '0';
                        out2 <= '0';
                else -- skip = '0'
                        nextstate <= s7;
                        out0 <= '1';
                        out1 <= '0';
                        out2 <= '1';
                end if;
        when s7 =>
                nextstate <= s1;
                out0 <= '0';
```

Appendix E. Design Entity File for Mealy Machine (continued)

```
                        out1 <= '0';
                        out2 <= '0';
                when s8 =>
                        if (hold = '1') then
                                nextstate <= s8;
                                out0 <= '0';
                                out1 <= '0';
                                out2 <= '0';
                        else -- hold = '0'
                                nextstate <= s9;
                                out0 <= '0';
                                out1 <= '1';
                                out2 <= '0';
                        end if;
                when s9 =>
                        if (hold = '1') then
                                nextstate <= s9;
                                out0 <= '0';
                                out1 <= '1';
                                out2 <= '0';
                        else -- hold = '0'
                                nextstate <= s4;
                                out0 <= '0';
                                out1 <= '0';
                                out2 <= '1';
                        end if;
                when others =>
                        nextstate <= s0;
                        out0 <= '0';
                        out1 <= '0';
                        out2 <= '0';
        end case;

    end process;
end fsm;
```

Appendix F. Binding Architecture File for Mealy Machine

```
--Binding Architecture
use work.cypress.all;
use work.rtlpkg.all;
use work.mealy_pkg.all;

ARCHITECTURE appnote OF c335 IS

BEGIN
        gsr <= pin2;
U1 : mealy PORT MAP (
        xin => pin3,
        yin => pin4,
        skip => pin5,
        pause => pin6,
        hold => pin7,
        clock => pin1,
        fbx(out0) => pin28,
        fbx(out1) => pin27,
        fbx(out2) => pin26
     );

END appnote;
```

Appendix G. Design Entity File for Machine with Moore and Mealy Outputs

```
package mix1_pkg is
        component mix1
        port (clock, xin, yin, skip, pause, hold : in bit;
                out0, out1, out2 : out bit);
        end component;
end mix1_pkg;
entity mix1 is
        port (clock, xin, yin, skip, pause, hold : in bit;
                out0, out1, out2 : out bit);
end mix1;
architecture fsm of mix1 is
        signal state:bit_vector(0 to 3);
        signal nextstate:bit_vector(0 to 3);
        signal nextout2:bit;
        constant s0: bit_vector(0 to 3) := "0000";
        constant s1: bit_vector(0 to 3) := "0001";
        constant s2: bit_vector(0 to 3) := "0010";
        constant s3: bit_vector(0 to 3) := "0011";
        constant s4: bit_vector(0 to 3) := "0100";
        constant s5: bit_vector(0 to 3) := "0101";
        constant s6: bit_vector(0 to 3) := "0110";
        constant s7: bit_vector(0 to 3) := "0111";
        constant s8: bit_vector(0 to 3) := "1000";
        constant s9: bit_vector(0 to 3) := "1001";

        begin
                proc1:process
                        begin
                        wait until clock = '1';
                        state <= nextstate;
                        out2 <= nextout2;
                end process;

                proc2:process (xin, yin, skip, pause, hold)
                        begin
                        case state is
                                when s0 =>
                                        nextstate <= s1;
                                        out0 <= '0';
                                        out1 <= '0';
```

Appendix G. Design Entity File for Machine with Moore and Mealy Outputs (continued)

```
                nextout2 <= '0';
        when s1 =>
                if (xin = '1' and yin = '1') then
                        nextstate <= s2;
                        out0 <= '1';
                        out1 <= '0';
                        nextout2 <= '0';
                elsif (xin = '0' and yin = '0')
then
                        nextstate <= s5;
                        out0 <= '0';
                        out1 <= '1';
                        nextout2 <= '0';
                else
                        nextstate <= s1;
                        out0 <= '0';
                        out1 <= '0';
                        nextout2 <= '0';
                end if;
        when s2 =>
                if (skip = '1') then
                        nextstate <= s4;
                        out0 <= '0';
                        out1 <= '0';
                        nextout2 <= '1';
                else -- skip = '0'
                        nextstate <= s3;
                        out0 <= '0';
                        out1 <= '1';
                        nextout2 <= '0';
                end if;
        when s3 =>
                if (skip = '1') then
                        nextstate <= s1;
                        out0 <= '0';
                        out1 <= '0';
                        nextout2 <= '0';
                else -- skip = '0'
                        nextstate <= s4;
                        out0 <= '0';
```

Appendix G. Design Entity File for Machine with Moore and Mealy Outputs (continued)

```
                        out1 <= '0';
                        nextout2 <= '1';
                end if;
        when s4 =>
                if (pause = '1') then
                        nextstate <= s8;
                        out0 <= '0';
                        out1 <= '0';
                        nextout2 <= '0';
                else -- pause = '0'
                        nextstate <= s1;
                        out0 <= '0';
                        out1 <= '0';
                        nextout2 <= '0';
                end if;
        when s5 =>
                if (skip = '1') then
                        nextstate <= s7;
                        out0 <= '1';
                        out1 <= '0';
                        nextout2 <= '1';
                else -- skip = '0'
                        nextstate <= s6;
                        out0 <= '1';
                        out1 <= '0';
                        nextout2 <= '0';
                end if;
        when s6 =>
                if (skip = '1') then
                        nextstate <= s1;
                        out0 <= '0';
                        out1 <= '0';
                        nextout2 <= '0';
                else -- skip = '0'
                        nextstate <= s7;
                        out0 <= '1';
                        out1 <= '0';
                        nextout2 <= '1';
                end if;
        when s7 =>
```

Appendix G. Design Entity File for Machine with Moore and Mealy Outputs (continued)

```
                                nextstate <= s1;
                                out0 <= '0';
                                out1 <= '0';
                                nextout2 <= '0';
                        when s8 =>
                                if (hold = '1') then
                                        nextstate <= s8;
                                        out0 <= '0';
                                        out1 <= '0';
                                        nextout2 <= '0';
                                else -- hold = '0'
                                        nextstate <= s9;
                                        out0 <= '0';
                                        out1 <= '1';
                                        nextout2 <= '0';
                                end if;
                        when s9 =>
                                if (hold = '1') then
                                        nextstate <= s9;
                                        out0 <= '0';
                                        out1 <= '1';
                                        nextout2 <= '0';
                                else -- hold = '0'
                                        nextstate <= s4;
                                        out0 <= '0';
                                        out1 <= '0';
                                        nextout2 <= '1';
                                end if;
                        when others =>
                                nextstate <= s0;
                                out0 <= '0';
                                out1 <= '0';
                                nextout2 <= '0';
                end case;

        end process;
end fsm;
```

Appendix H. Binding Architecture File for Machine with Moore and Mealy Outputs

```
--Binding Architecture
use work.cypress.all;
use work.rtlpkg.all;
use work.mix1_pkg.all;

ARCHITECTURE appnote OF c335 IS

BEGIN
        gsr <= pin2;
U1 : mix1 PORT MAP (
        xin => pin3,
        yin => pin4,
        skip => pin5,
        pause => pin6,
        hold => pin7,
        clock => pin1,
        fbx(out0) => pin28,
        fbx(out1) => pin27,
        fbx(out2) => pin26
     );

END appnote;
```

Appendix I. Design Entity File for Moore Machine Using State Table Method

```
package moore3_pkg is
        component moore3
        port (clock, xin, yin, skip, pause, hold : in bit;
                out0, out1, out2 : out bit);
        end component;
end moore3_pkg;
use work.table_bv.all;
entity moore3 is
        port (clock, xin, yin, skip, pause, hold : in bit;
                out0, out1, out2 : out bit);
end moore3;
architecture state_table of moore3 is
        signal table_out:bit_vector(0 to 6);
        constant s0:x01_vector(0 to 3) := "0000";
        constant s1:x01_vector(0 to 3) := "0001";
        constant s2:x01_vector(0 to 3) := "0010";
        constant s3:x01_vector(0 to 3) := "0011";
        constant s4:x01_vector(0 to 3) := "0100";
        constant s5:x01_vector(0 to 3) := "0101";
        constant s6:x01_vector(0 to 3) := "0110";
        constant s7:x01_vector(0 to 3) := "0111";
        constant s8:x01_vector(0 to 3) := "1000";
        constant s9:x01_vector(0 to 3) := "1001";
        constant table:x01_table(0 to 19, 0 to 15) := (
-- q3, q2, q1, q0 & xin, yin, skip, pause, hold & q3', q2', q1', q0' &
-- out2, out1, out0
                s0 & "xxxxx" & s1 & "000",
                s1 & "11xxx" & s2 & "001",
                s1 & "00xxx" & s5 & "010",
                s1 & "01xxx" & s1 & "000",
                s1 & "10xxx" & s1 & "000",
                s2 & "xx1xx" & s4 & "100",
                s2 & "xx0xx" & s3 & "010",
                s3 & "xx1xx" & s1 & "000",
                s3 & "xx0xx" & s4 & "100",
                s4 & "xxx1x" & s8 & "000",
                s4 & "xxx0x" & s1 & "000",
                s5 & "xx1xx" & s7 & "101",
                s5 & "xx0xx" & s6 & "001",
                s6 & "xx1xx" & s1 & "000",
```

Appendix I. Design Entity File for Moore Machine Using State Table Method (continued)

```
                s6 & "xx0xx" & s7 & "101",
                s7 & "xxxxx" & s1 & "000",
                s8 & "xxxx1" & s8 & "000",
                s8 & "xxxx0" & s9 & "010",
                s9 & "xxxx1" & s9 & "010",
                s9 & "xxxx0" & s4 & "100");
        begin
                process
                begin
                        wait until clock='1';
                        table_out <=
                                ttf(table,(table_out(0 to 3) & xin & yin &
skip & pause & hold));
                end process;
        out0 <= table_out(6);
        out1 <= table_out(5);
        out2 <= table_out(4);
end state_table;
```

Appendix J. Binding Architecture File for Moore Machine Using State Table Method

```
--Binding Architecture
use work.cypress.all;
use work.rtlpkg.all;
use work.moore3_pkg.all;

ARCHITECTURE appnote OF c335 IS

BEGIN
        gsr <= pin2;
U1 : moore3 PORT MAP (
        xin => pin3,
        yin => pin4,
        skip => pin5,
        pause => pin6,
        hold => pin7,
        clock => pin1,
        fbx(out0) => pin28,
        fbx(out1) => pin27,
        fbx(out2) => pin26
    );

END appnote;
```

Appendix K. Design Entity File for Mealy Machine Using State Table Method

```
package mealy2_pkg is
        component mealy2
        port (clock, xin, yin, skip, pause, hold : in bit;
                out0, out1, out2 : out bit);
        end component;
end mealy2_pkg;
use work.table_bv.all;
entity mealy2 is
        port (clock, xin, yin, skip, pause, hold : in bit;
                out0, out1, out2 : out bit);
end mealy2;
architecture state_table of mealy2 is
        signal table_out:bit_vector(0 to 6);
        signal next_table_out:bit_vector(0 to 6);
        constant s0:x01_vector(0 to 3) := "0000";
        constant s1:x01_vector(0 to 3) := "0001";
        constant s2:x01_vector(0 to 3) := "0010";
        constant s3:x01_vector(0 to 3) := "0011";
        constant s4:x01_vector(0 to 3) := "0100";
        constant s5:x01_vector(0 to 3) := "0101";
        constant s6:x01_vector(0 to 3) := "0110";
        constant s7:x01_vector(0 to 3) := "0111";
        constant s8:x01_vector(0 to 3) := "1000";
        constant s9:x01_vector(0 to 3) := "1001";
        constant table:x01_table(0 to 19, 0 to 15) := (
-- q3, q2, q1, q0 & xin, yin, skip, pause, hold & q3', q2', q1', q0' &
-- out2, out1, out0
                s0 & "xxxxx" & s1 & "000",
                s1 & "11xxx" & s2 & "001",
                s1 & "00xxx" & s5 & "010",
                s1 & "01xxx" & s1 & "000",
                s1 & "10xxx" & s1 & "000",
                s2 & "xx1xx" & s4 & "100",
                s2 & "xx0xx" & s3 & "010",
                s3 & "xx1xx" & s1 & "000",
                s3 & "xx0xx" & s4 & "100",
                s4 & "xxx1x" & s8 & "000",
                s4 & "xxx0x" & s1 & "000",
                s5 & "xx1xx" & s7 & "101",
                s5 & "xx0xx" & s6 & "001",
```

Appendix K. Design Entity File for Mealy Machine Using State Table Method (continued)

```
                s6 & "xx1xx" & s1 & "000",
                s6 & "xx0xx" & s7 & "101",
                s7 & "xxxxx" & s1 & "000",
                s8 & "xxxx1" & s8 & "000",
                s8 & "xxxx0" & s9 & "010",
                s9 & "xxxx1" & s9 & "010",
                s9 & "xxxx0" & s4 & "100");
        begin
                process
                begin
                        wait until clock = '1';
                        table_out <= next_table_out;
                end process;
        next_table_out <= ttf(table,(table_out(0 to 3) & xin & yin & skip &
pause & hold));
        out0 <= next_table_out(6);
        out1 <= next_table_out(5);
        out2 <= next_table_out(4);
end state_table;
```

Appendix L. Binding Architecture File for Mealy Machine Using State Table Method

```
--Binding Architecture
use work.cypress.all;
use work.rtlpkg.all;
use work.mealy2_pkg.all;

ARCHITECTURE appnote OF c335 IS

BEGIN
        gsr <= pin2;
U1 : mealy2 PORT MAP (
        xin => pin3,
        yin => pin4,
        skip => pin5,
        pause => pin6,
        hold => pin7,
        clock => pin1,
        fbx(out0) => pin28,
        fbx(out1) => pin27,
        fbx(out2) => pin26
     );

END appnote;
```

Appendix M. Design Entity File for Machine with Moore and Mealy Outputs Using State Table Method

```
package mix2_pkg is
        component mix2
        port (clock, xin, yin, skip, pause, hold : in bit;
                out0, out1, out2 : out bit);
        end component;
end mix2_pkg;
use work.table_bv.all;
entity mix2 is
        port (clock, xin, yin, skip, pause, hold : in bit;
                out0, out1, out2 : out bit);
end mix2;
architecture state_table of mix2 is
        signal table_out:bit_vector(0 to 6);
        signal next_table_out:bit_vector(0 to 6);
        constant s0:x01_vector(0 to 3) := "0000";
        constant s1:x01_vector(0 to 3) := "0001";
        constant s2:x01_vector(0 to 3) := "0010";
        constant s3:x01_vector(0 to 3) := "0011";
        constant s4:x01_vector(0 to 3) := "0100";
        constant s5:x01_vector(0 to 3) := "0101";
        constant s6:x01_vector(0 to 3) := "0110";
        constant s7:x01_vector(0 to 3) := "0111";
        constant s8:x01_vector(0 to 3) := "1000";
        constant s9:x01_vector(0 to 3) := "1001";
        constant table:x01_table(0 to 19, 0 to 15) := (
-- q3, q2, q1, q0 & xin, yin, skip, pause, hold & q3', q2', q1', q0' &
-- out2, out1, out0
                s0 & "xxxxx" & s1 & "000",
                s1 & "11xxx" & s2 & "001",
                s1 & "00xxx" & s5 & "010",
                s1 & "01xxx" & s1 & "000",
                s1 & "10xxx" & s1 & "000",
                s2 & "xx1xx" & s4 & "100",
                s2 & "xx0xx" & s3 & "010",
                s3 & "xx1xx" & s1 & "000",
                s3 & "xx0xx" & s4 & "100",
                s4 & "xxx1x" & s8 & "000",
                s4 & "xxx0x" & s1 & "000",
                s5 & "xx1xx" & s7 & "101",
```

Appendix M. Design Entity File for Machine with Moore and Mealy Outputs Using State Table Method (continued)

```
                s5 & "xx0xx" & s6 & "001",
                s6 & "xx1xx" & s1 & "000",
                s6 & "xx0xx" & s7 & "101",
                s7 & "xxxxx" & s1 & "000",
                s8 & "xxxx1" & s8 & "000",
                s8 & "xxxx0" & s9 & "010",
                s9 & "xxxx1" & s9 & "010",
                s9 & "xxxx0" & s4 & "100");
        begin
                process
                begin
                        wait until clock = '1';
                        table_out <= next_table_out;
                        out2 <= next_table_out(4);
                end process;
        next_table_out <= ttf(table,(table_out(0 to 3) & xin & yin & skip &
pause & hold));
        out0 <= next_table_out(6);
        out1 <= next_table_out(5);
end state_table;
```

Appendix N. Binding Architecture File for Machine with Moore and Mealy Outputs Using State Table Method

```
--Binding Architecture
use work.cypress.all;
use work.rtlpkg.all;
use work.mix2_pkg.all;

ARCHITECTURE appnote OF c335 IS

BEGIN
        gsr <= pin2;
U1 : mix2 PORT MAP (
        xin => pin3,
        yin => pin4,
        skip => pin5,
        pause => pin6,
        hold => pin7,
        clock => pin1,
        fbx(out0) => pin28,
        fbx(out1) => pin27,
        fbx(out2) => pin26
    );

END appnote;
```

Document #: PLD0015−1292

Using Hierarchical VHDL Design

Introduction

Hierarchical design methodology has been commonly used for quite some time by system designers and software developers. There are two primary advantages to using this methodology. First, it allows commonly-used building blocks to be created separately and saved for later use without having to redesign or reverify them. Second, it allows for more readable design files by keeping the top-level design file as a simple integration of smaller building blocks, either user-defined or from a vendor-supplied library. In system design, these building blocks normally take the form of schematic symbols instantiated into a schematic drawing, while in software they are functions or procedures that are called from the main program.

Warp2 and *Warp3* VHDL includes a set of features specifically designed to make hierarchical design both simple and powerful. This note will first describe these features and then walk through a simple example of how they might be used. It assumes that the reader has read the *Warp2* User's Manual and is familiar with how to create a VHDL design unit consisting of an entity-architecture pair.

Key Concepts

In order to construct a hierarchical design in VHDL, the designer must understand the concepts of components, packages and libraries.

Component – A component is a VHDL design unit that may be instantiated in other VHDL design units. Before it can be instantiated, it must be declared using the COMPONENT declaration which specifies the name of the component and lists its local signal names.

Package – A package is a collection of VHDL declarations that can be used by other VHDL descriptions. For the purpose of creating hierarchical designs, a package consists of one or more components. However, a package may also include other types of declarations.

Library – A library is a logical storage facility for design units. Before a component can be instantiated in a higher-level design unit, its package must be compiled into a library that is visible to that design unit, usually the current work library.

Simple Example

Consider the following example. A designer discovers that for a specific type of circuit design he commonly needs an unusual type of counter. ("Commonly," in this reference, means that this counter is likely to be used either multiple times in a particular design or across multiple designs. Both are cases where hierarchical design simplifies things.) This counter is a simple four-bit counter, but it must output a terminal count indication (tc) and roll over to zero when it reaches 1110 rather than 1111.

A design file that would accomplish this is shown in Appendix A. (The reader should understand the contents of the entity-architecture pair—they will not be discussed further.) In order to use this counter in other VHDL design units, it is declared as a component within a package at the top of the file. The component declaration simply names the design unit and lists its signal names. When this file is compiled, the package is placed into the current library and the component it contains may then be instantiated into other designs compiled into that library. If this were a standalone design, the entire package declaration could be omitted.

Now, suppose this design consists of two of these counters with their outputs multiplexed as in *Figure 1*. We can then instantiate our counter twice as in the design in Appendix B. All that is necessary is the statement:

```
use work.cnt_pkg.all;
```

at the top of the file, which makes any components in the cnt_pkg visible within the current design unit, as long as the package was compiled into the work library. The counters are then instantiated by giving them unique labels and listing the signals connected to the port map in the same order as the component declaration.

We could also have created the mux as a separate component and instantiated it, but it is simpler to use the if-then-else structure.

Multiple Components

For further illustration, assume our complete design includes two types of counters, one that rolls over at 1110 and one that rolls over at 1011, as shown in *Figure 2*. We could simply repeat the above procedure and create another design file with another component and package and then use both of these packages in our top-level design file.

However, it may be easier to keep track of things if we keep similar counter designs together in a single package as in Appendix C. This file contains both entity-architecture pairs and two components in a single package. As before, when this file is compiled, the package is added to the current library and its components are made visible with a single-use clause as in Appendix D.

Configurable Components using Generics

When multiple components that have the same basic architecture but differ in one or more parameters are needed (such as the two counters in the previous example) VHDL generics allow a more compact approach. Generics are a means by which parameters may be passed to a component when it is instantiated allowing a configurable component.

In Appendix E a component is created that is the same basic counter, but allows the terminal count to be configured using a generic. Instead of hard-cod-

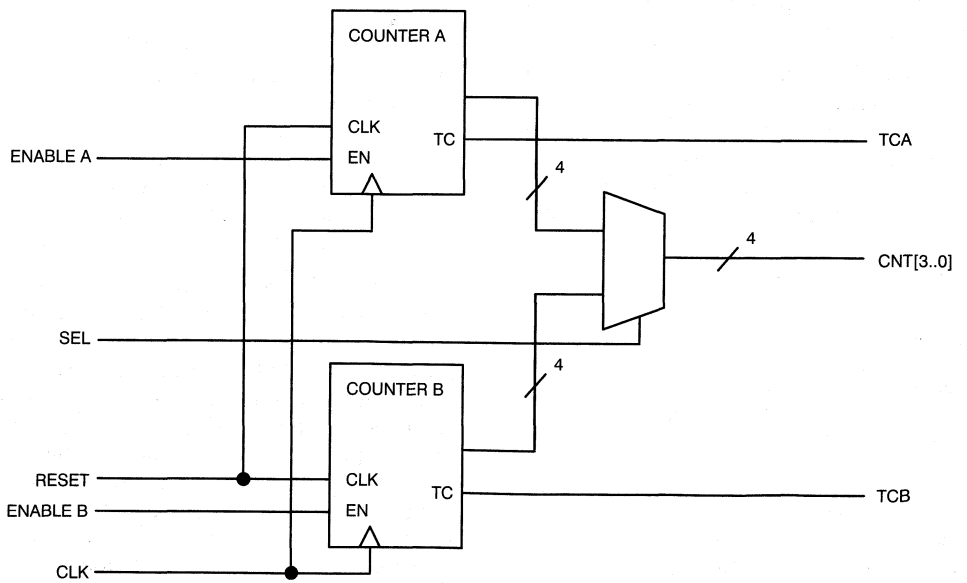


Figure 1. Multiplexed Dual Counter Design

ing this value, a bit_vector is used in the architecture. This bit_vector is then declared in the entity and component declarations. Generics may also be of other types such as integers and a component may contain multiple generics (although our example contains only one).

Appendix F is the top-level design unit of the same design from *Figure 2*, but this time it is using the component with the generic rather than two different components. When the component is instantiated, it is configured by passing it the specific bit_vector in the generic map.

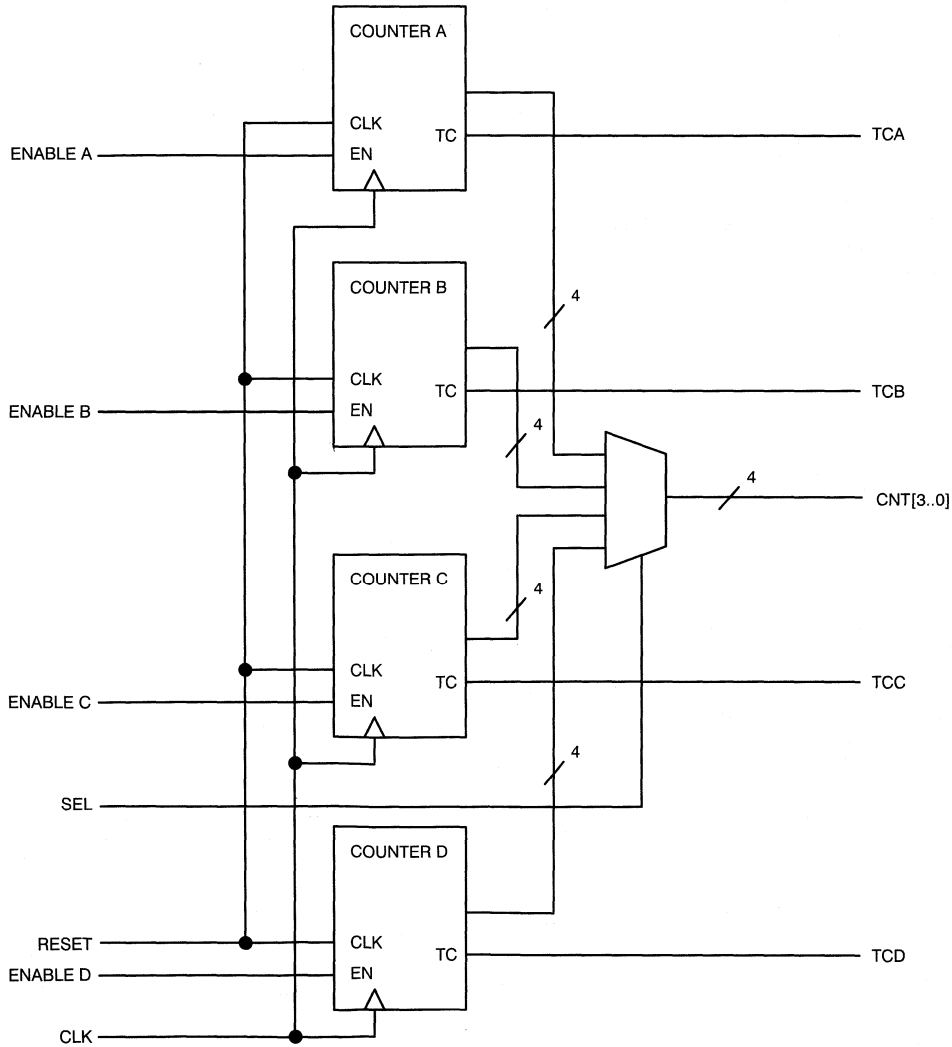


Figure 2. Multiplexed Quad Counter Design

Appendix A. Counter with Terminal Count and Rollover Selection

```
use work.cypress.all;
use work.rtlpkg.all;

package cnt_pkg is
  component count15 port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
  end component;
end cnt_pkg;

use work.bv_math.all;

entity count15 is port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
end count15;

architecture one of count15 is
begin

process begin
if cnt="1110" then
    tc<='1';
else
    tc<='0';
end if;

end process;

process(clk,reset) begin
    if reset='1' then
        cnt<="0000";
    elsif (clk'event and clk='1') then
        if cnt="1110" and enable='1' then
            cnt<="0000";
        elsif enable='1' then
            cnt<=inc_bv(cnt);
        else
            cnt<=cnt;
        end if;
    end if;

end process;

end one;
```

Appendix B. Instantiation of Counter from Appendix A

```
use work.cypress.all;
use work.rtlpkg.all;
use work.cnt_pkg.all;

entity muxcntr is port(
     clk, enablea, enableb, reset, sel:in bit;
     cnt:out bit_vector (3 downto 0);
     tca, tcb:out bit);
end muxcntr;

architecture one of muxcntr is

signal muxina, muxinb:bit_vector(3 downto 0);

begin

cntra:count15 port map(clk, enablea, reset, muxina, tca);
cntrb:count15 port map(clk, enableb, reset, muxinb, tcb);

process begin
if sel='1' then
     cnt<=muxina;
else
     cnt<=muxinb;
end if;

end process;
end one;
```

Appendix C. Multiple Counters in a Single Package

```
use work.cypress.all;
use work.rtlpkg.all;

package cnt_pkg is
  component count15 port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
  end component;
  component count12 port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
  end component;
end cnt_pkg;

use work.bv_math.all;

entity count15 is port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
end count15;

architecture one of count15 is
begin

process begin
if cnt="1110" then
    tc<='1';
else
    tc<='0';
end if;
end process;

process(clk,reset) begin
    if reset='1' then
        cnt<="0000";
    elsif (clk'event and clk='1') then
        if cnt="1110" and enable='1' then
            cnt<="0000";
        elsif enable='1' then
            cnt<=inc_bv(cnt);
        else
            cnt<=cnt;
        end if;
    end if;
```

Appendix C. Multiple Counters in a Single Package (continued)

```
end process;
end one;

use work.bv_math.all;

entity count12 is port(
     clk, enable, reset:in bit;
     cnt:inout bit_vector (3 downto 0);
     tc:out bit);
end count12;

architecture one of count12 is
begin

process begin
if cnt="1011" then
     tc<='1';
else
     tc<='0';
end if;
end process;

process(clk,reset) begin
     if reset='1' then
          cnt<="0000";
     elsif (clk'event and clk='1') then
          if cnt="1011" and enable='1' then
               cnt<="0000";
          elsif enable='1' then
               cnt<=inc_bv(cnt);
          else
               cnt<=cnt;
          end if;
     end if;

end process;
end one;
```

Appendix D. Instantiation of Counters in Appendix C

```
use work.cypress.all;
use work.rtlpkg.all;
use work.cnt_pkg.all;

entity muxcntr is port(
     clk, enablea, enableb, enablec, enabled, reset:in bit;
     sel:in bit_vector (1 downto 0);
     cnt:out bit_vector (3 downto 0);
     tca, tcb, tcc, tcd:out bit);
end muxcntr;

architecture one of muxcntr is

signal muxina, muxinb, muxinc, muxind:bit_vector(3 downto 0);

begin

cntra:count15 port map(clk, enablea, reset, muxina, tca);
cntrb:count15 port map(clk, enableb, reset, muxinb, tcb);
cntrc:count12 port map(clk, enablec, reset, muxinc, tcc);
cntrd:count12 port map(clk, enabled, reset, muxind, tcd);

process begin
if sel="11" then
     cnt<=muxina;
elsif sel="10" then
     cnt<=muxinb;
elsif sel="01" then
     cnt<=muxinc;
else
     cnt<=muxind;
end if;

end process;
end one;
```

Appendix E. Parametrizable Counters Using Generics

```
use work.cypress.all;
use work.rtlpkg.all;

package cnt_pkg is
  component countg
  generic (stop:bit_vector(3 downto 0):="1111");
    port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
  end component;
end cnt_pkg;

use work.bv_math.all;

entity countg is
  generic (stop:bit_vector(3 downto 0):="1111");
    port(
    clk, enable, reset:in bit;
    cnt:inout bit_vector (3 downto 0);
    tc:out bit);
end countg;

architecture one of countg is
begin

process begin
if cnt=stop then
    tc<='1';
else
    tc<='0';
end if;
end process;

process(clk,reset) begin
    if reset='1' then
        cnt<="0000";
    elsif (clk'event and clk='1') then
        if cnt=stop and enable='1' then
            cnt<="0000";
        elsif enable='1' then
            cnt<=inc_bv(cnt);
        else
            cnt<=cnt;
        end if;
    end if;

end process;
end one;
```

Appendix F. Multiplexed Quad Counter Design

```
use work.cypress.all;
use work.rtlpkg.all;
use work.cnt_pkg.all;

entity muxcntr is port(
     clk, enablea, enableb, enablec, enabled, reset:in bit;
     sel:in bit_vector (1 downto 0);
     cnt:out bit_vector (3 downto 0);
     tca, tcb, tcc, tcd:out bit);
end muxcntr;

architecture one of muxcntr is

signal muxina, muxinb, muxinc, muxind:bit_vector(3 downto 0);

begin
cntra:countg generic map("1110") port map(clk, enablea, reset, muxina, tca);
cntrb:countg generic map("1110") port map(clk, enableb, reset, muxinb, tcb);
cntrc:countg generic map("1011") port map(clk, enablec, reset, muxinc, tcc);
cntrd:countg generic map("1011") port map(clk, enabled, reset, muxind, tcd);

process begin
if sel="11" then
     cnt<=muxina;
elsif sel="10" then
     cnt<=muxinb;
elsif sel="01" then
     cnt<=muxinc;
else
     cnt<=muxind;
end if;

end process;
end one;
```

Document #: PLD0008–0194

Specialty Memories – 4





Specialty Memories Section Contents and Abstracts

Understanding Dual-Port RAMs **4–1**

This application note reviews the history of multi-port memories and explains the operation of Cypress's Dual-Port RAMs. Features discussed range from basic dual-port fundamentals to more advanced issues (like the "deadly embrace," for example) and ends with a design example. This application note is intended for designers of all experience levels and addresses most of the common issues that arise when using dual-port RAMs.

Understanding Large FIFOs **4–19**

This application note explains the operation, architecture, and design considerations of Cypress's CY7C42X, 7C43X, and 7C46X families of FIFOs. These FIFOs feature industry-standard operation and pinout and are available in depths up to 32K x 9. Basic logic and timing operations such as reading and writing to the FIFO memory array, are covered in detail. Timing waveforms are included to help illustrate these operations. Common FIFO configurations such as Standalone Mode, Depth Expansion and Width Expansion are explained in detail. These sections explain how to properly use the flags in these modes and cover the operation of the expansion-in/expansion-out (XI/XO) pins. The final sections of this application note cover common problems and solutions encountered when using large FIFOs. These common problems include corrupted or repetitive data, missing or disappearing data, and FIFO lock up. Boundary flag operation is also discussed in relation to these problems. The final section covers V_{CC} noise related failures and recommends specific power bypassing techniques.

Understanding Clocked FIFOs **4–29**

This application note explains the basic operations and features of Cypress Clocked FIFOs (CY7C44X and CY7C45X Clocked FIFOs). The first few sections explain the Clocked FIFO architecture in detail. Reading and writing to the FIFO memory array are discussed and timing waveforms are included to illustrate these operations. A large portion of the application note is devoted to explaining the synchronous flag architecture. Gate-level logic diagrams are provided to help explain flag operation. This section also explains commonly misunderstood concepts such as flag encoding and flag latency cycles. A section on programming and resetting Clocked FIFOs explains how to properly perform these operations and covers the common design pitfalls to avoid. Two sections are devoted to configuring Clocked FIFOs for depth or width expansion modes. These sections include discussions on proper flag decoding and expansion-in/expansion-out (XI/XO) pin operation. The final section discusses how to use a Clocked FIFO like an industry standard asynchronous FIFO.

FIFO Dipstick Using *Warp2*™ VHDL and the CY7C371 ..... **4–39**

Programmable FIFO flags can often simplify the design of a digital system by generating status which will prevent overrun or underrun conditions for an elastic FIFO buffer. Although many FIFOs are available with programmable flag functions on-chip, these features are not available on industry-standard asynchronous FIFOs. Of those FIFOs that do have programmable flags, some do not allow the almost-empty and almost-full values to be programmed independently, or in some cases, for these values to be programmed at any specific word boundary. This application note will present a method by which FIFOs of any size may be monitored by an external Programmable Logic Device that will then generate all of the flags necessary for most FIFO applications. The FIFO Dipstick PLD behaves like a measuring device that can observe the level of data within a FIFO.

CY7C42X/46X FIFO Interface to the CY7C923 (HOTLink™) 4–46

This application note discusses the parallel interface between industry standard FIFOs (CY7C42X/46X) and a Cypress HOTLink Transmitter (CY7B923). A simple design example is provided. The bulk of this application note focuses on explaining the impact of datasheet timing parameters on the maximum interface frequency. Six timing relationships are derived from the provided design example. Datasheet timing parameters from different speed grade FIFOs are inserted into these equations. The results are summarized in table form showing maximum FIFO-HOTLink Transmitter interface operating frequency as a function of FIFO speed. This application note is useful as a guide when performing timing analysis on similar HOTLink-FIFO interface configurations.

Interfacing the CY7B923 and CY7B933 (HOTLink) to a Clocked FIFO 4–49

This application note considers the interface issues between the Cypress CY7B923/933 (HOTLink) transmitter/receiver and Cypress Clocked FIFOs. This note is divided into two sections: HOTLink Transmitter-Clocked FIFO interfaces, and HOTLink Receiver-Clocked FIFO interfaces. The transmitter interface section provides a simple design example that uses a state machine to control the HOTLink-FIFO interface. A state transition diagram for the controller is provided. Critical path timing analysis is then discussed for this design example. The derived critical path equations and their critical datasheet parameters are provided and explained. A timing diagram is shown to help illustrate these critical timing relationships.

The HOTLink Receiver-FIFO interface section also includes a simple design example. A simple state machine controls this interface. The state machine addresses design issues such as reframing the serial data, BIST (Built-In Self-Test), and programming clocked FIFOs. These issues are discussed in detail. A state transition diagram is included. Critical path timing equations are derived and the advantages of pipelining the interface are discussed. Timing waveforms are shown to help illustrate the critical timing relationships.

Interfacing the CY7B923 and CY7B933 (HOTLink) to a Wide Data Clocked FIFO 4–57

This application note considers the interface issues between the Cypress CY7B923/933 (HOTLink) transmitter/receiver and Cypress Clocked FIFOs. The focus of this application note is on applications that use wide data, e.g., 32 bits. This note is divided into two sections: HOTLink Transmitter-Clocked FIFO interfaces, and HOTLink Receiver-Clocked FIFO interfaces. The transmitter interface section provides a simple design example that uses a state machine to control the HOTLink-FIFO interface. The data word size is chosen to be 32 bits. A simple 4:1 mux is used to funnel the data out of the FIFOs and into the HOTLink Transmitter. The state machine controls the sequencing of the data through the muxes. A state transition diagram for the controller is provided. Critical path timing analysis is then discussed for this design example. The derived critical path equations and their critical datasheet parameters are provided and explained. A timing diagram is shown to help illustrate these critical timing relationships.

Understanding Dual-Port RAMs

This application note examines the evolution of multi-port memories and explains the operation and benefits of Cypress's dual-port RAMs.

A dual-port RAM is a random-access memory that can be accessed simultaneously by two independent entities. In digital ICs, this implies a dual-port memory cell that can be accessed at the same time using two independent sets of address, data, and control lines.

A Brief History of Multi-Port Memories

The first multi-port memories were probably used in the CPU of the first computers. Many two-operand instructions are efficiently implemented using dual-port registers for the operands and the result.

For example, consider *Equation 1*, which describes a typical two-operand operation in the ALU (arithmetic logic unit) of a CPU:

(C) = (A) [OPERATOR] (B) Eq. 1

A and B could be either the operands (i.e., the data) or the addresses of the operands, in which case the data could be either in memory or in registers. In any case, *Equation 1* describes two pieces of data, A and B, being operated upon by the OPERATOR and the results designated as C. C could also be the data, a register, or a memory location. OPERATOR could be arithmetic or logical.

The Combinatorial ALU

The 74181 was the first integrated circuit ALU. In this IC, the 4-bit operands, A and B, are operated upon according to a 4-bit command; the result, C, is output. The chip also provides a carry-in input, a carry-out output, and A = B outputs. A mode-control pin selects either logical or arithmetic operations. The 74181 is combinatorial; no storage is provided.

Early computers used the contents of a memory location as one operand and an accumulator in the CPU as the second operand. The results were usually stored in the accumulator.

Bringing the Registers On Chip

The 67901 was the first 4-bit slice that brought 16 4-bit registers onto the chip. The MMI 67901 was second-sourced by AMD and became the 2901. At one time, five vendors offered this industry-standard bipolar ALU. The Cypress CMOS CY7C901 is the highest-performance, TTL-compatible, 4-bit slice that is form, fit, and functionally equivalent to the original 901.

The 16-word-deep, 4-bit-wide register array is functionally equivalent to a 16 x 4 dual-port memory. Four A address lines and four B address lines select the contents of two of the 16 registers, whose outputs are applied to transparent latches. The latch outputs are then applied to 3:1 multiplexers, whose outputs drive the ALU inputs. The ALU outputs can be sent off chip, entered into a temporary register (Q), or written back into the register file, thus replacing one of the operands. This architecture is shown in the CY7C901 block diagram in Cypress's *Data Book*.

CY7C901 Dual-Port Memory Operation

A simplified CY7C901 block diagram appears in *Figure 1*. The device's A and B addresses select the contents of two registers, whose outputs are applied to two 4-bit latches. When the clock (CP) is HIGH, the latch outputs follow their data inputs (i.e., are

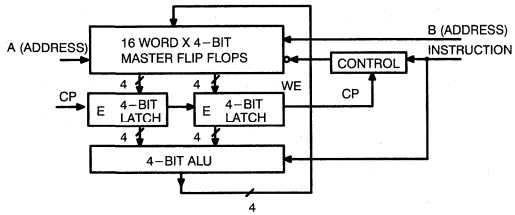


Figure 1. CY7C901 Dual-Port Memory (Simplified)

transparent). When the clock is LOW, the ALU outputs are written ($\overline{WE}$) into the register array at the location specified by the A or B addresses, depending upon the instruction being executed. A LOW on the clock causes the data in the latches not to change, so that the ALU outputs are stable when they are written back into the register array.

Note that the CY7C901 does not perform the three-port function described by *Equation 1*. In the CY7C901, the C operand equals either the A or B operand, depending upon the instruction being executed. In fact, the A and B addresses can be the same. An old programming trick is to Exclusive-OR the contents of a register with itself, which clears the register.

Additionally, the CY7C901's dual-port memory does not use a dual-port memory cell. This type of cell is not required because the CY7C901 does not need the ability to simultaneously write independently to two separate memory locations.

Dual-Port Memory Using Single-Port RAM

Before the dual-port memory cell existed, designers created dual-port RAMs from single-port RAMs by adding a multiplexer between the RAM and the two entities that shared the RAM. *Figure 2* illustrates a block diagram of such an arrangement. Two processors, MP1 and MP2, share the RAM. If each processor has access to the RAM half the time, the resource is shared equally and is said to be allocated according to a fairness doctrine.

This time division multiplexing assures that there is no contention for the RAM. However, performance

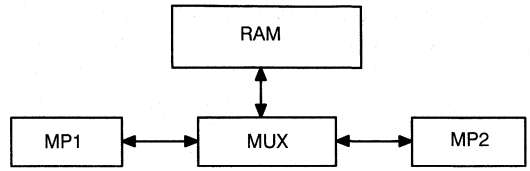


Figure 2. Dual-Port Memory Using Single-Port RAM

suffers if the RAM's access time does not equal 1/2 or less of the processors' clock period, assuming that the processors are clocked from the same source.

For example, consider two processors clocked from the same 25-MHz source, for a period of 40 ns. Because the processors are closely coupled, only one operating system is in memory. In this case, the maximum access time of the dual port has to be 20 ns or less. The highest-speed dual-port RAM available has a 25-ns access time. Therefore, each processor suffers a worst-case 20% performance degradation.

Dual-Port RAM Applications

The first applications for dual-port memories were for CPU register files. Dual-port RAMs can also serve as data or instruction cache memories. However, the largest usage of dual-port RAMs is in communications, which includes the exchange of data between processors, processes, and systems.

Virtual Dual-Port RAM

Communication between systems does not require physical dual-port RAMs. Instead, a conventional RAM memory is partitioned into virtual data-storage areas (buffers), usually to store at least two data packets. These buffers are shared between the communications controller and the intelligent element that assembles the packets and stores them (usually a microprocessor). The communications controller can also be a microprocessor. It reads the data from memory, converts the data from parallel to serial form, encodes the data, converts the data to analog form, and sends the data out over the communications channel on the transmit side. If the system contains only one processor, the data buffers are not shared, and the system needs neither a virtual nor a physical dual-port RAM.

Control information associated with each data buffer tells the communications controller the number of words in the buffer and the starting address of the data in the buffer. The control information resides in one or more memory locations whose addresses have been previously agreed upon by the two processors.

This simple software-based buffer example requires a second level of control—a mechanism or procedure that prevents the two microprocessors from getting in each other's way. In other words, the system needs a procedure control mechanism.

Another way of analyzing this requirement introduces the concept of data ownership. Say, for example, that processor A assembles and stores messages and thus owns the data while performing these tasks. Likewise, the communications processor B owns the data while performing its tasks. The procedure control mechanism amounts to a technique for transferring data ownership between processor A and B.

In large systems, where many processors perform many different operations, the processing of the information is called a job or a procedure. The procedure is divided into many tasks, which can be performed by different processors. The tasks can either be scheduled and assigned by a processor dedicated to that task or be performed by any available processor. These alternatives are referred to as autocratic and egalitarian systems, respectively. The term egalitarian implies that the processors are treated equally. In either case, the processors must have access to a shared-memory location used for message passing.

Synchronizing sequential processes is the cornerstone of concurrent programming, which applies to multi-tasking, single-processor systems; distributed-processor networks; and tightly coupled multiprocessor systems.

Message Passing

In the two-processor system under consideration, synchronization can be achieved by using a lockword or lockvariable. The lockvariable can apply either to data (as in this example) or to executable instructions.

The lockvariable is a location in shared memory that is operated upon using two synchronization primitives: LOCK (v) and UNLOCK (v), where (v) is the location operated upon. These are simple binary switch operations. If a processor wishes to lock or own a critical section of code or data, the processor indivisibly sets the lockvariable if testing shows the lockvariable to be zero. If the lockvariable is not zero, then the operation is repeated until the lockvariable is zero. To unlock the critical section, a processor sets the lockvariable to zero and continues.

Most modern processors have indivisible read/modify/write instructions, also called test and set (TAS) instructions. In Reference 1, however, E. W. Dijkstra shows that lockvariables can be implemented without using a read/modify/write instruction. And in Reference 2 he develops the semaphore, a technique for managing a queue of tasks waiting for a resource. Lockvariables surround or bracket semaphores and thus provide entry and exit control on a mutual-exclusion basis.

Typical TAS Instruction

The current example assumes that the processors have a TAS instruction. A typical TAS instruction operates as follows: read, test, and set to X. The addressed memory location is read, and if its contents are zero, the value X is written into that location. If the contents are not zero, the contents are returned to the processor, and the value in the memory location is not disturbed.

The usual convention is that a value of zero in the lockvariable means that the resource associated with it is available. A non-zero value means that another processor temporarily owns the resource and that the resource is not available. After performing the task associated with the lockvariable, the processor sets the lockvariable's value to zero. The system is initialized with all lockvariables set to zero.

In the current example, processor A performs a TAS operation on the lockvariable and, finding the lockvariable to be zero, sets the lockvariable to a one. This tells processor B that the message is in the pro-

cess of being assembled in the memory buffer area and is not ready to be transmitted. Processor A then assembles the message. After the message is assembled, processor A clears the lockvariable, sends a message to processor B saying that the message is ready to be transmitted, and gives the data's location and the number of bytes to be sent. Processor B reads the message from processor A and performs a TAS operation on the lockvariable; finding the lockvariable to be zero, processor B sets it to a two. This tells processor A that the message is in the process of being transmitted. Processor B then transmits the message and clears the lockvariable. Processor B sends processor A a message that the transmission task has been completed. After receiving the message from processor B, processor A performs a TAS operation on the lockvariable; finding the lockvariable to be zero, processor A concludes that the message has been successfully transmitted.

Note that this procedure does not require the use of a dual-port RAM. The procedure does require each processor to perform a TAS instruction, clear the lockvariable, and send a message to the other processor. Sending a message implies writing to a location in shared memory. To know that a message is waiting, the processor receiving the message must either read the memory location periodically (referred to as polling a mailbox) or the act of writing to the mailbox must generate an interrupt to the receiving processor. The interrupt-driven alternative is usually preferred because the receiving processor does not have to waste time in a polling sequence.

Dual-Port RAM Cell History

The first dual-port RAM ICs to use a dual-port RAM cell were the Synertek SY2130 and SY2131, introduced in 1983. These products are organized as 1024 words of 8 bits and use n-channel, double-polysilicon technology to achieve 100-ns access times. The SY2130 has an automatic power down feature controlled by the chip enables, and the SY2131 does not. The smaller (512 x 8) SY2132 and SY2133 were similar but unsuccessful.

The original dual-port RAMs include two mailboxes for message passing. When written to from one port, a mailbox generates an interrupt to the opposite port. Additionally, on-chip arbitration logic generates a busy signal to the loser when both left and right ports address the same memory location. If the loser was attempting to write, the write is suppressed.

Most of the dual-port RAMs on the market today are functionally equivalent to the original Synertek products. The "new features" added to several dual-port RAM products by Cypress, Motorola, and Integrated Device Technology (IDT) include dedicated semaphore registers. Hardware semaphores provide efficient means of allocating exclusive priority accesses to blocks of shared memory locations in dual-port RAMs.

The SY2130 was second-sourced by IDT in 1984 and Advanced Micro Devices (AMD) in 1985. IDT also doubled the density to 2K x 8 and called the new part the IDT7132. Due to pin limitations (48 pins), the interrupt functions were deleted.

In 1985 IDT added slave companion parts to the company's dual-port family. The IDT7140 (1024 x 8) is the slave to the IDT7130, and the IDT7142 (2K x 8) is the slave to the IDT7132. The slave device provides word-width expansion. BUSY is an input to the slave from the master, and the slave contains no arbitration logic. One master can drive many slaves. This arrangement avoids the classic deadly embrace problem described in the next section.

The Deadly Embrace

The deadly embrace can occur when two masters are connected in parallel to make a wider word. If the left and right port addresses match, and the left and right port chip enables then become active to both chips at approximately the same time, it is possible to have one port of one master lose and the opposite port of the other master also lose. In other words, if an address match occurs and both ports are enabled during a small time window or an aperture of uncertainty, the dual-port RAM cannot determined which port wins or loses.

Under these conditions, if the corresponding left and right port busy pins are connected together, both ports of both masters are active (LOW). This condition occurs because the busy outputs are open drain, and the loser pulls the node LOW.

This condition is the simplest example of the deadly embrace. As far as the external world is concerned, both ports are busy, and the system remains locked up indefinitely, with each port waiting to be released by the other. Each master's arbiter section thinks it has lost the arbitration and is waiting to be released by the other.

In general, the deadly embrace occurs under two conditions: a processor requires one or more resources to perform a task, and one or more of the required resources is temporarily owned by another processor, which requires one or more of the same resources to perform its task.

For example, if processor A owns resource X and processor B owns resource Y, and both resources are required to accomplish the task, a stalemate occurs in which each processor waits for the other to relinquish the required resource. This is the simplest example. The concept extends to *n* processors and *m* resources.

The solution to the deadly embrace depends upon whether the system is autocratic or egalitarian, the tasks' priorities, etc., and is beyond the scope of this discussion. In the case of dual-port RAMs, however, the solution is simple: Do not cascade two masters in width; use a master and a slave.

The Cypress Dual-Port RAM Family

Table 1 lists the members of the Cypress dual-port RAM family. The package designator D26 stands for 600-mil ceramic DIP, and P25 stands for 600-mil plastic DIP. The 48-pin ceramic leadless chip carrier (LCC) is designated as L68. The 52-pin packages are designated as L69 for ceramic LCC and J69 for plastic LCC (PLCC). The 68-pin packages are designated by L81 for ceramic LCC, J81 for plastic LCC (PLCC), and G68 for ceramic pin grid array.

Note that the interrupt function is not available at the 2048 x 8 level in a 48-pin package. This is due to pin limitations. At the 2-Kbyte level, each port requires an additional address pin for the address's most significant bit.

The M/S column in *Table 1* indicates whether the device is a master or slave. The difference between these devices is that the masters have arbitration logic and the slaves do not. The busy signals are outputs from the master and inputs to the slave. (The ramifications of this are examined later.)

Table 1. The Cypress Dual-Port RAM Family

Configuration	Part #	M/S	Semaphores	Interrupt	Busy	Package Options							
						48-Pin Dual In-Line Pkg		48-Pin Square	52-Pin Square		68-Pin Square		
						Ceramic	Plastic	LCC	LCC	PLCC	LCC	PLCC	PGA
1Kx8	CY7C130	M	N	Y	Y	D26	P25	L68					
	CY7C131	M	N	Y	Y				L69	J69			
	CY7C140	S	N	Y	Y	D26	P25	L68					
	CY7C141	S	N	Y	Y				L69	J69			
2Kx8	CY7C132	M	N	N	Y	D26	P25	L68					
	CY7C136	M	N	Y	Y				L69	J69			
	CY7C142	S	N	N	Y	D26	P25	L68					
	CY7C146	S	N	Y	Y				L69	J69			

Table 1. The Cypress Dual-Port RAM Family (continued)

Configuration	Part #	M/S	Semaphores	Interrupt	Busy	Package Options							
						48-Pin Dual In-Line Pkg		48-Pin Square	52-Pin Square		68-Pin Square		
						Ceramic	Plastic	LCC	LCC	PLCC	LCC	PLCC	PGA
4Kx8	CY7B134	M/S	N	N	N	D26	P25	L68					
	CY7B135	M/S	N	N	N				L69	J69			
	CY7B1342	M/S	Y	N	N				L69	J69			
	CY7B138	M/S	Y	Y	Y						L81	J81	G68
2Kx9	CY7B139	M/S	Y	Y	Y						L81	J81	G68
8Kx8	CY7B144	M/S	Y	Y	Y						L81	J81	G68
8Kx9	CY7B145	M/S	Y	Y	Y						L81	J81	G68

Cypress Dual-Port RAM Operation

A simplified block diagram of the Cypress dual-port RAM appears in *Figure 3*. The device interface includes three types of signals: address, data, and control. There are two sets of these signals: those of the left port and those of the right port. Each signal has either the subscript L or R to designate left or right, respectively.

The address pins are designated A0 through A9 (1024 x 8) and A0 through A10 (2048 x 8), where A0 is the least significant bit (LSB) and A9 or A10 is the most significant bit (MSB). The address pins are unidirectional inputs to the device; their states specify the memory location to be read from or written into.

The data pins are designated I/O0 through I/O7, where I/O0 is the LSB and I/O7 is the MSB. The data pins are bidirectional; their states represent either the data to be written or the data to be read.

The control pins are chip enable ($\overline{CE}$), read/write (R/$\overline{W}$), and output enable ($\overline{OE}$). A semaphore enable control pin ($\overline{SEM}$) is included on dual-port RAMs with semaphores. Two flags are also provided, $\overline{INT}$ and $\overline{BUSY}$; both have open-drain outputs and require external pull-up resistors. A LOW on the chip enable input allows that port to become functional. Data is either read from the internal

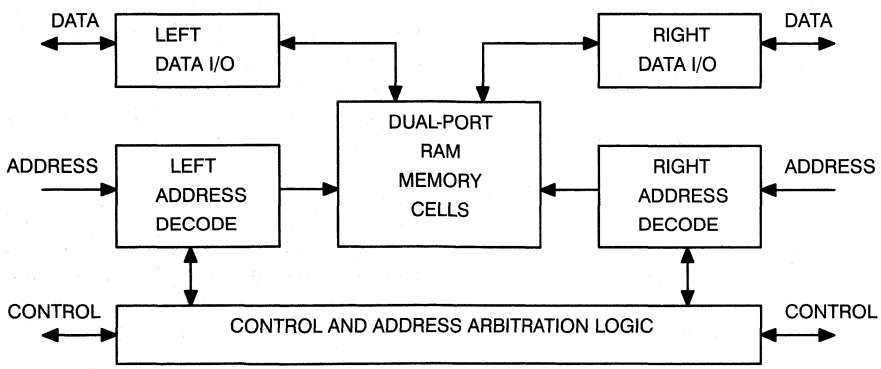


Figure 3. Dual-Port RAM Block Diagram

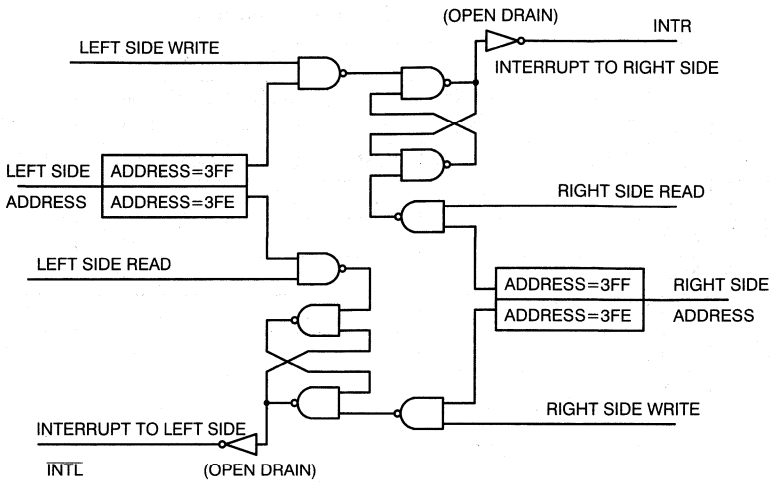


Figure 4. Interrupt Logic

dual-port RAM array or written into it, depending upon the state of the read/write signal; a LOW initiates a write operation. The three-state data output drivers are enabled by a LOW output enable.

When one port writes to a pre-determined mailbox, an interrupt to the other port is generated. When the interrupted port reads that memory location, the interrupt is reset.

When both ports address the same memory location and both chip enables are active (LOW), contention occurs for that address. An arbitration is then performed, and ownership of the memory location is assigned to the winner. An active (LOW) busy signal notifies the loser of the arbitration.

Dual-Port RAM Functional Description

An important aspect of the Cypress dual-port RAMs is their interrupt logic. A simplified logic diagram of this logic appears in *Figure 4*, with the chip enables deleted. A port's chip enable must be asserted for the port to either read from or write to any location, including the mailboxes. Note that you can use the mailbox locations as conventional memory by not connecting the interrupt line to the appropriate processor.

The upper two memory locations (7FF and 7FE for 2K x 8; 3FF and 3FE for 1K x 8) can be used for message passing. The highest memory location serves as the mailbox for the right processor. When the left processor writes to this mailbox, the interrupt (request) to the right processor, $\overline{\text{INTR}}$, goes LOW. When the right processor reads its mailbox, the flip-flop is reset, and $\overline{\text{INTR}}$ goes HIGH.

The second highest memory location serves as the mailbox for the left processor. When the right processor writes to this mailbox, the interrupt (request) to the left processor, $\overline{\text{INTL}}$, goes LOW. When the left processor reads its mailbox, the flip-flop is reset, and $\overline{\text{INTL}}$ goes HIGH.

Note that each port can read the other port's mailbox without resetting the associated flip-flop. If your application does not require message passing, leave the appropriate pin open. Do not connect a pull-up resistor to the pin, and do not connect the pin to the processor's interrupt request pin.

Table 2. Functional Operation of Dual-Port Masters

Operation			Result of Operation after Arbitration (Master)
Case	Left Port	Right Port	
1	Read	Read	Both ports read.
2	Read	Write	Loser is prevented from writing. If loser is reading and ports are asynchronous, data read might not be valid.
3	Write	Read	
4	Write	Write	

Note that the active state of the busy signal prevents a port from setting the interrupt to the winning port. Additionally, an active busy signal to a port prevents that port from reading its own mailbox and thus resetting the interrupt. These operations are ramifications of the data-ownership concept.

If both ports address the same memory location at the same time, the master performs an arbitration, so that one port wins and the other loses. Because each of the two ports can be in either the reading or writing state, there are four possible combinations of ports and states (*Table 2*).

Both Ports Reading

If both ports of a dual-port IC read the same location at the same time, you can assume that both ports read the same data. When arbitration occurs as a result of contention in a Cypress dual-port RAM, the port that wins the arbitration gets temporary ownership of the memory location. The losing port can read the memory location but the busy signal tells it that it lost the arbitration.

To guarantee data integrity in a multiprocessor system, it is standard practice to apply the concept of data ownership. This ownership can apply to executable code, data, or control locations in memory. The control locations in memory can be associated with a resource, such as a printer, tape drive, disk drive, or communications port.

One Port Reading, the Other Writing

The result of arbitration will allocate priority to either the reading or the writing port. In Cypress dual-port RAMs, if the losing port is attempting to write data, the write is inhibited so that the data in memory is not corrupted. The $\overline{\text{BUSY}}$ flag to the losing port signals that the write was not performed.

If the losing port is attempting to read data, it is possible for the data to be old data, new data, or some random combination of the two. The $\overline{\text{BUSY}}$ flag to the losing port signals that the old data is still being read on the losing port's data lines. The old data will remain undisturbed for an access time after either $\overline{\text{BUSY}}$ on the losing port goes HIGH, the losing port's address is toggled, $\overline{\text{CE}}$ for the losing port is toggled, or R/$\overline{\text{W}}$ for the losing port is toggled during a valid read.

If the new data is needed, the $\overline{\text{BUSY}}$ flag can be used to generate a delay until the new data is present or can signal a processor to attempt the read again after $\overline{\text{BUSY}}$ is cleared.

Both Ports Writing

The losing port is prevented from writing so that the data cannot be corrupted. $\overline{\text{BUSY}}$ is asserted to the losing port, indicating that the write operation was unsuccessful.

Arbitration Logic

Figure 5 shows the arbitration logic used in Cypress dual-port RAM masters. The arbitration logic has three functions: to decide which port wins and which loses if the addresses are equal simultaneously, to prevent the losing port from writing, and to provide a busy signal to the losing port.

The arbitration logic consists of left and right address equality comparators with their associated delay buffers; the arbitration latch formed by the cross-coupled, three-input NAND gates labeled L and R; and the gates that generate the busy signals.

Operation With Unequal Addresses

When the addresses of the right and left ports are not equal, the outputs of the address comparators (nodes A and B) are both LOW, and the outputs of the gates labeled L and R (nodes C and D) are both HIGH. This condition forces both $\overline{\text{BUSY}}$ signals HIGH and both $\overline{\text{Write Inhibit}}$ signals HIGH. The arbitration latch does not function as a latch.

Left Port Camped on an Address

Next, consider the condition where the left-port address and chip enable are quiescent, and the right-port address changes to an address equal to that of the left port. Nodes A and B are initially LOW.

Because the right-port address does not go through the delay buffer, the output of the right-address comparator (node B) goes HIGH before node A goes HIGH by a delay interval, d. The delay must be greater than the delay through the R gate, so that when node B goes HIGH, node D goes LOW, causing node C to remain HIGH. CE(R) and CE(L) are both HIGH; they are the inverse of the chip enable inputs. Node D going LOW causes the output of the BR gate to go LOW, which tells the right port that the memory location it just addressed belongs to the left port. A write inhibit signal is also generated that prevents the right port from writing into the addressed memory location.

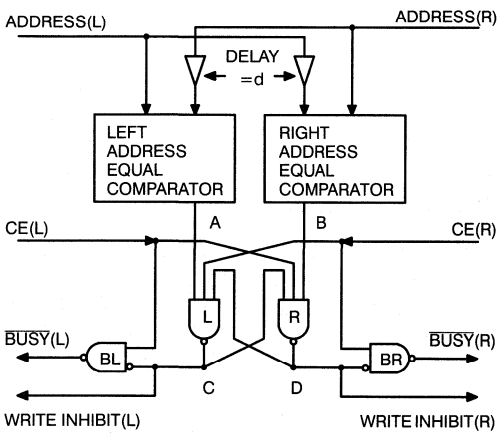


Figure 5. Arbitration Logic

In summary, when the right port addresses a memory location that is already being addressed by the left port, a delay occurs that equals the sum of the propagation delays of the right-address comparator, the R gate, the BR gate, and the output driver (not shown in the diagram). Then the busy signal to the right port is asserted. Nodes A, B, and C are now HIGH, and node D is LOW. $\overline{\text{BUSY}}$ is asserted to the right port.

Due to the symmetry of the arbitration logic, the device operates the same when either the right or left ports are camped on an address.

Right and Left Addresses Equal Simultaneously

In the general case, it is possible to have both ports access the same memory location simultaneously, unless this is guaranteed not to occur by the design of the system. When nodes A and B go from LOW to HIGH at exactly the same instant, the arbitration latch settles into one of two states and determines which port wins and which port loses. The latch is designed such that its two outputs are never LOW at the same time. It also has a very fast switching time.

The dual-port RAM imposes a minimum time difference between either of two events: the two chip enables going from inactive to active and the two sets of addresses going from mismatch to equal. If the events are close together in time, the probability of each port either winning or losing the arbitration is approximately equal. This parameter is called port set-up time for priority and is abbreviated as t_{PS} on the datasheets. The specified value is 5 ns. (Note, though, that Cypress product engineers have measured t_{PS} at room temperature and nominal V_{CC} (5V) and found a value of approximately 200 ps.) In other words, if one port addresses a memory location 5 ns before the other port, the first port is guaranteed to win. If not, the result of the subsequent arbitration is unpredictable.

Other Key $\overline{\text{BUSY}}$ Parameters

Several other key parameters are specified with respect to the busy signal. For example, $\overline{\text{BUSY}}$ LOW from address match, t_{BLA}, is the maximum time it takes busy to go LOW, as measured from the time the two port addresses are the same. This is the time from an address match until the losing port is notified that it has lost the arbitration. Obviously, the sooner this occurs the better. If the value of t_{BLA} is greater than the memory cycle time, another cycle must be added to detect the condition, which can severely reduce performance. This time is less than the minimum cycle time for all speed grades of all Cypress dual-port RAMs.

Another parameter, $\overline{\text{BUSY}}$ HIGH from address mismatch, t_{BHA}, is the maximum time it takes $\overline{\text{BUSY}}$ to go from LOW to HIGH, as measured from the time the two port addresses do not match until the $\overline{\text{BUSY}}$ signal goes HIGH. The comments of the preceding paragraph also apply here.

The next two parameters are similar to the preceding two. The difference is that the chip enable controls the busy signal. The parameters are $\overline{\text{BUSY}}$ LOW from $\overline{\text{CE}}$ LOW, t_{BLC}, and $\overline{\text{BUSY}}$ HIGH from $\overline{\text{CE}}$ HIGH, t_{BHC}. Both of these parameters are less than the minimum cycle time for all speed grades of all Cypress dual-port RAMs.

$\overline{\text{BUSY}}$ HIGH to valid data, t_{BDD}, is the maximum time it takes the data to become valid to the losing port after $\overline{\text{BUSY}}$ goes away. This parameter's value equals the address access time, t_{AA}, because a read cycle is initiated to the losing port when its $\overline{\text{BUSY}}$ signal transitions from LOW to HIGH. An action by either port can cause the busy transition. The winning port can either change its address or deassert its chip enable.

To illustrate the last two parameters, *Figure 6* shows the timing for the right port performing a write operation and the left port asynchronously moving to the same address and attempting to perform a read operation. The first parameter of interest is t_{DDD}, which is the maximum time between the stabilization of the data to be written by the winning port and that same data becoming valid at the outputs of the port that received the $\overline{\text{BUSY}}$. The second parameter of interest is t_{WDD}, which is the maximum time between the HIGH-to-LOW transition of the winning port's write strobe and the data becoming valid at the outputs of the port that received the $\overline{\text{BUSY}}$.

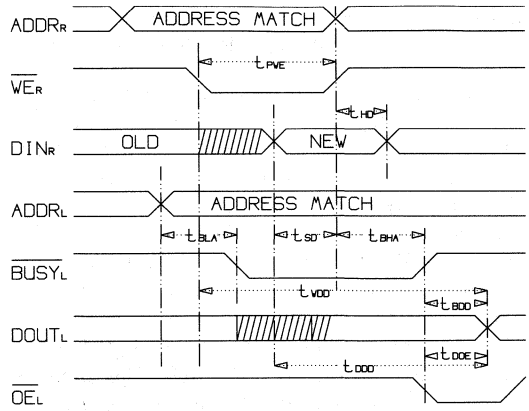


Figure 6. $\overline{\text{BUSY}}$ Timing

It is possible for the losing port to read either the old data, the new data, or some random combination of the two under these circumstances: the two ports are operating asynchronously (i.e., with independent clocks), and the conditions illustrated in *Figure 6* occur (winning port writing and losing port reading). If the read occurs early with respect to the write, old data is read. If the read occurs late with respect to the write, new data is read. And, if the read occurs at the same time the data is changing from old to new, the data read is not predictable. However, all is not lost. There are two general solutions. Both use the fact that the busy signal is asserted to the losing port, telling the port in this instance that the data it is reading might not be valid.

One solution is to use the HIGH-to-LOW transition of the busy signal to the losing port to generate an interrupt to the processor (or state machine) so that operation can be repeated. The drawback of this technique is that a snapshot of the states of the losing port's address lines and read/write line must be taken, so that the processor can tell what load/store operation caused the interrupt. Taking this snapshot requires latches or flip-flops for the data and control logic for doing the sampling, and the tech-

nique uses up an interrupt line. The processor must also be able to read the sampled data later.

A second solution is to use the LOW level of the $\overline{BUSY}$ signal to the losing port to prompt one of three types of delays: delay the reading of data until the data becomes valid, which occurs an access time after the LOW-to-HIGH transition of $\overline{BUSY}$; insert wait states until $\overline{BUSY}$ goes HIGH; or stretch the clock until $\overline{BUSY}$ goes HIGH. Any of these methods probably require less hardware and control logic than the preceding approach. Use of these methods does mean that the $\overline{BUSY}$ signal must eventually go from LOW to HIGH. This happens when the winning port either changes its address or deasserts its chip enable. For this reason, as well as for system noise immunity and power-saving considerations, it is recommended that blocks of addresses be decoded to generate chip enables for the dual-port RAMs.

Because the losing port has no control over the winning port in the general case, however, a question arises: What can the losing port do to successfully read the data just written, assuming the winning port does not change its address, write, or chip enable signals? There are two possible operations:

1. Change an address line to a different address, then change back to the original address. This toggles the $\overline{BUSY}$ signal to the losing port.
2. Change the state of the chip enable. This also toggles the $\overline{BUSY}$ signal to the losing port.

Hardware Semaphores

Cypress offers dual-port RAMs with eight on-chip hardware semaphore latches that are independent from RAM memory locations. Semaphore signaling is a popular method of allocating mutually exclusive accesses to blocks of memory that are shared among several processors. Exclusive processor control guarantees data integrity in sensitive applications such as shared I/O buffers. Semaphore signaling can also improve the efficiency of block memory accesses by preventing delays and processor stalls due to a memory location being busy from another processor access.

Traditional semaphore signaling has been implemented in software using dedicated memory locations to hold the semaphore signals. A processor could attempt to gain control of a semaphore by using an indivisible test and set instruction to test if the semaphore was set by another processor. If the semaphore is free, the processor sets the semaphore and gains exclusive control of a block of memory.

Cypress dual-port RAMs have on-chip hardware semaphores that are independent from RAM memory locations. Hardware semaphores eliminate the need to use a processor with an indivisible test and set instruction. Semaphore control requests are handled using a standard write to the semaphore latch followed by a read instruction. There is no requirement to lockout other processor accesses to the semaphore between the write and read.

The hardware semaphores provide flexible software configuration of shared memory. The semaphores operate independent of any memory in the RAM allowing software to allocate block addresses and block sizes.

Cypress hardware semaphores implement a "token passing" scheme allowing the port in possession of the token to have exclusive access to a block of shared memory. Possession of the token can only be relinquished by the port with possession. A port's request for possession of the token will be denied if the token is held by the other port.

Possession of a token is indicated by the state of a semaphore latch formed from two cross-coupled NOR gates (see *Figure 7*). The latch can be set so that only one port controls the semaphore at a time. Additional input latches on the semaphore ports are used to hold requests to set or clear the latch. An output latch on each port is used to prevent the output from changing during a read from the port.

The semaphore latches are accessed through the data and address ports the same way as a RAM cell access. The semaphore enable line ($\overline{SEM}$ = LOW) initiates a semaphore access cycle. The A0–2 lines select which semaphore latch is accessed. Only the data on D_0 is latched into the semaphore during a write. The other data lines are ignored. During a

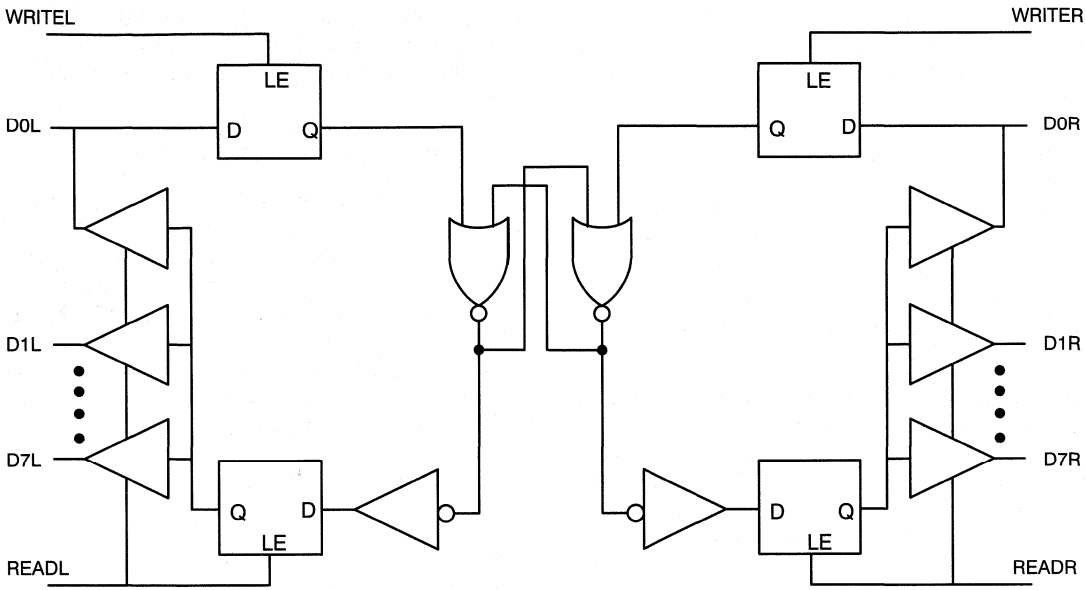


Figure 7. Semaphore Latch Cell

read, the semaphore drives all the data lines (D0 through D7, D8) with the semaphore signal.

A processor requests control of a semaphore by writing a 0 to the D0 port of the semaphore addressed by A0–2. The 0 is latched into the port's input register and held until another write attempts to set it to 1. If the semaphore is free at the time of the request, the port will immediately be granted control of the semaphore. If the semaphore is controlled by the other port, the request for control will be denied. If control of the semaphore is relinquished by the other port while the 0 is still pending, then the requesting port will gain control of the semaphore. Control of the semaphore can only be relinquished by the controlling port by writing a 1 to the semaphore.

To see if a request for control of the semaphore was successful, a read of the semaphore is performed. A port controls the semaphore if 0 is read out on D0. The port does not control the semaphore if a 1 is read. The semaphore outputs drive all of the data lines with the state of the semaphore, so D0–7 will be "00000000" when control is granted and will be "11111111" when control is denied. The state of the internal semaphore latches may change during a read, but the output latch prevents the changes from propagating to the data lines. A new read cycle must be performed in order to update the port's output lines.

If both ports attempt to write a 0 within t_{SPS} of each other while the semaphore is free, semaphore arbitration logic will guarantee that only one side gains control of the semaphore.

Address Transition Detection

Why does changing the address or chip enable allow a losing port to read data successfully? All Cypress dual-port RAMs, both masters and slaves, use a circuit design technique called Address Transition Detection (ATD) to improve performance and reduce power dissipation.

ATD improves performance by equilibrating differential paths, pre-charging critical nodes, and forcing the outputs to a high-impedance state. Equilibration and pre-charging will bias critical nodes to voltage levels approximately in the mid-point of the small-signal operating range; when the data is sensed, it takes a shorter amount of time to transition to the 0 or 1 level. Forcing the outputs to their high-impedance states improves speed slightly, but more importantly, the technique reduces output switching noise by eliminating crowbar current and separating the output current into two pulses instead of one.

ATD minimizes power consumption because it turns on power-hungry circuits only when they are required. Slightly over 50 percent of a RAM's circuits are linear, and approximately 70 percent of the power is dissipated in the sense amplifiers during a read operation. When the RAM is operating at its maximum frequency, the ATD circuits are constantly triggered, so the power savings are minimal. At lower speeds or smaller duty cycles, however, the power savings are significant.

A diagram representing a typical ATD sequence is illustrated in *Figure 8*. The event that triggers the ATD sequence for either port is the transition of any address, chip-enable, or read/write signal. Equilibration and pre-charging are performed next, followed by either turning on the sense amplifiers and latching the data (read operation) or pulling the BIT and $\overline{BIT}$ lines to the required levels (write operation) at the addressed location. The master clock pulse lasts from 7 to 11 ns, depending upon temperature, supply voltage, and the distribution of IC processing parameters. At the end of the pulse, the data is latched and the appropriate circuits are turned off.

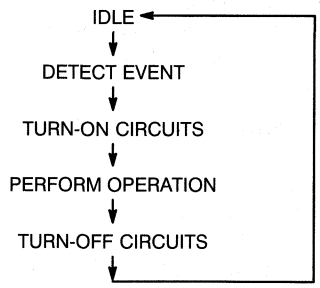


Figure 8. Simplified ATD Sequence

Master Standalone Operation

Figure 9 presents a block diagram of a system using two 8-bit microprocessors, the Cypress CY7C132 dual-port RAM, static RAM, and EPROM. The address lines of each microprocessor are decoded to generate the chip enables to the dual-port RAM, the SRAM, and the EPROM. Note that pull-up resistors are required on the interrupt requests to the microprocessors and the busy signals, which go to the microprocessors' wait inputs.

Slave Word-Width Expansion

The block diagram in *Figure 10* shows how to interconnect a CY7C132 (2K x 8) master and a CY7C142 (2K x 8) slave to form a 16-bit-wide word. The diagram does not show the interfaces to the processors or the connections for the interrupt signals. As previously explained, the interrupt outputs are not available at the 2K x 8 level in the 48-pin DIP due to pin limitations. In the LCC and PLCC packages, the interrupt outputs are available from both the master and the slave devices. You can use either one. You do not have to tie the corresponding interrupt pins of the master and the slave together.

Delaying the Write Strobe

In width expansion, the write signals to the slave devices must be delayed by an interval at least equal to t_{BLA}, which is the time required for the master to assert the busy signal to the slave after an address match. The delay prevents the slave data at the address in contention from being overwritten. Both the write and read cycle times must be increased by this amount of time. In equation form:

$$t_{WC} = t_{PWE} + t_{BLA} \qquad \text{Eq. 2}$$

where the delay must be at least equal to t_{BLA}.

Note that if you add more slaves to make a wider word, (e.g., 24 or 32 bits) the delay elements' outputs can connect directly to the write strobe inputs. Additional delay elements are not required.

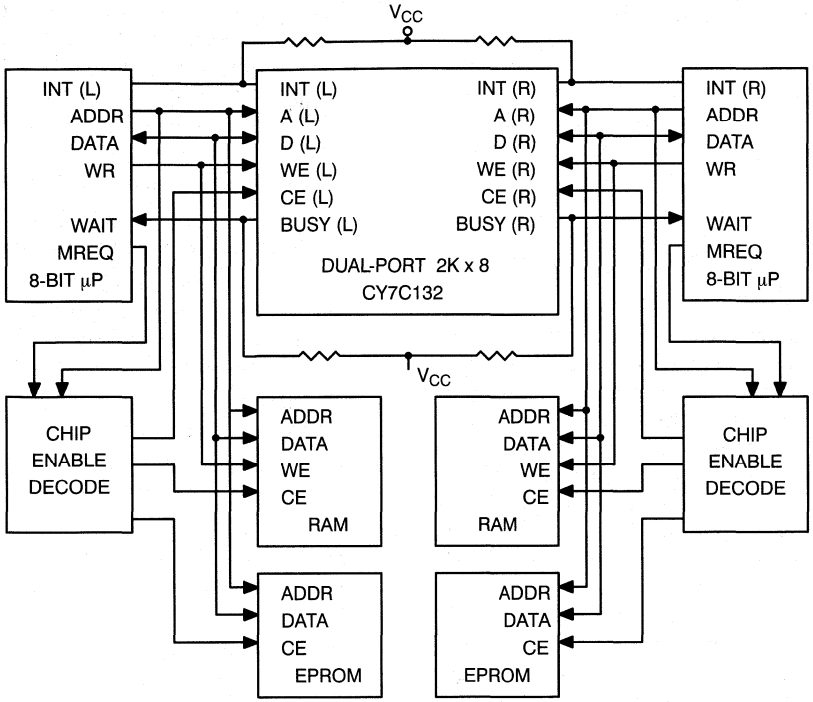


Figure 9. Typical 8-Bit Microprocessor

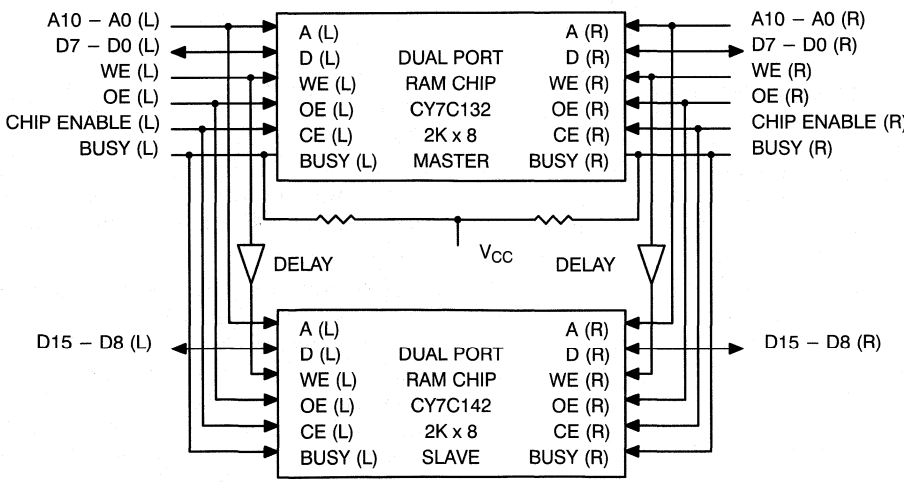


Figure 10. Expansion (2K x 16) with Slave

Slave Standalone Operation

Some applications might require that you give one port permanent and absolute priority over the other. You can easily do this by implementing the memory using only slave dual-port RAMs. The $\overline{BUSY}$ input to the priority port must be tied HIGH by either connecting it directly to V_{CC} or to V_{CC} through a 10-KΩ pull-up resistor. You can connect the low-priority port's $\overline{BUSY}$ input to the high-priority port's read/write input.

In this configuration, the busy (read/write) signal to the lower-priority port always prevents the port from writing when the high-priority port is writing to any location. The data of the lower-priority port is overwritten when the two ports operate asynchronously, the lower-priority port is writing, and the higher-priority port simultaneously writes. This is not a very elegant solution because the $\overline{BUSY}$ input to the low-priority port is not qualified by comparing the addresses of the two ports or their chip enables. However, this approach suggests how the slave dual-port RAMs can be used with external arbitration logic. The busy inputs can be used by control logic or under program control to dynamically change the port priorities.

If the lower-priority port is read only, you can tie its $\overline{BUSY}$ input HIGH by either connecting it directly to V_{CC} or to V_{CC} through a pull-up resistor.

Dual-Port Design Example

The following design example illustrates the methodology to follow when designing with Cypress dual-port RAMs. In this example, a dual-port memory is used for message passing and bus snooping for many bus masters on a 32-bit-wide system bus. The dual-port RAMs interface to a 32-bit system bus on the right side and a 16-bit processor on the left side. From the right port, the memory appears as 8K 32-bit words, and from the left port the memory appears as 16K 16-bit words.

The memory has the following characteristics:

1. The memory location corresponding to address 0 for both ports is the same.
2. The data read from and written to the memory from both ports is in the same order. Thus, D0 of the right port corresponds to D0 of the left port. Additionally, D16 of the right port appears as D0 of the left port in address location 2048.
3. The minimum cycle time is 35 ns.
4. To conserve power, blocks of addresses are decoded to generate the required chip selects.
5. The CY7C132 and CY7C142 dual-port RAMs are used. Part of the design task is to specify the number of masters and slaves required and the way they must be interconnected.
6. The appropriate $\overline{BUSY}$ signals must be generated to the correct port when contention occurs.
7. All possible mailbox locations that can be used for message passing are used.
8. The right port signals are AR0...AR12, DR0...DR31, $\overline{CER}$, and $\overline{BUSYR}$. The left port signals are AL0...AL13, DL0...DL15, $\overline{CEL}$, and $\overline{BUSYL}$.

A simplified logic diagram of the memory appears in *Figure 11*. A total of 16 2K x 8 dual-port RAMs are required. The devices labeled MA (master, bank A) through MD (master, bank D) are CY7C132 masters. The devices labeled SU (slave, upper half-word) and SL (slave, lower half-word) are CY7C142 slaves. The memory consists of four masters and twelve slaves, along with the required control logic.

From the right port the memory is configured as 8K 32-bit words, with a master controlling three slaves. The one-of-four decoder labeled RB (right bank) generates chip enable signals for each bank of 2K 32-bit words. Data is written (sampled) on the bus side, and the only reads performed are from the mailbox locations.

A general-purpose, right-port, control-logic block generates control signals that conform to the timing diagram shown in *Figure 12*. The diagram does not show the generation of the output enable control signals, but they are similar to the RB decoder signals. If your application does not require message passing to the right port, you can tie the right-port output enable pins of all of the dual-port RAMs directly to V_{CC}.

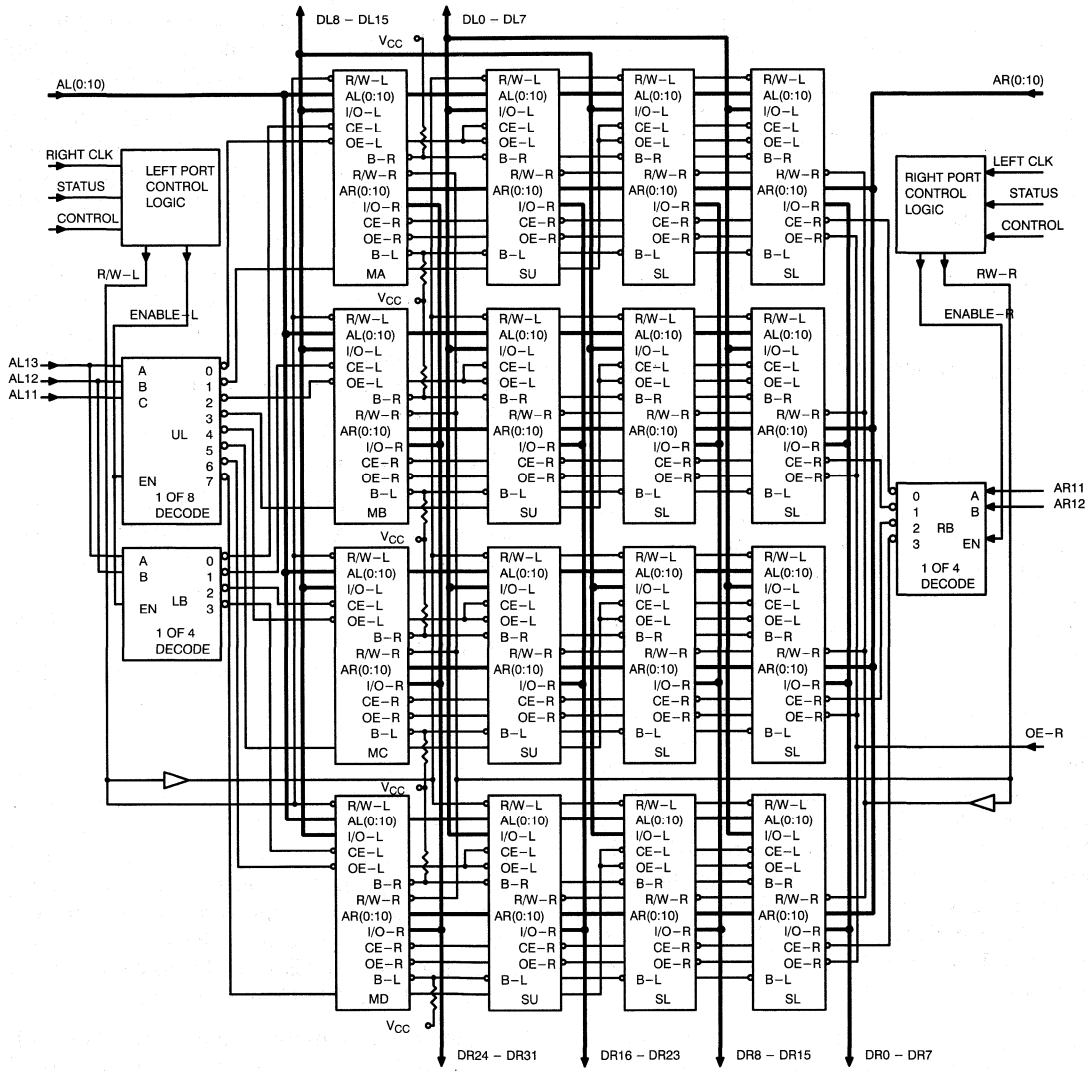


Figure 11. Logic Diagram for Dual-Port Example

From the left port, the memory is configured as 16K 16-bit words. For this organization, you might think that the slave dual-port RAMs in the second column from the right in *Figure 11* should be masters. If this were the case, however, you would have to defeat the arbitration logic in them when the right port addressed the same address; this would add logic, reduce the speed, and complicate the design. Therefore, this design uses a combination of left-bank decoding (LB, 1-of-4 decoder) and upper-lower 16-bit word decoding (UL, 1-of-8 decoder) to cause the bank master to arbitrate when the right port is

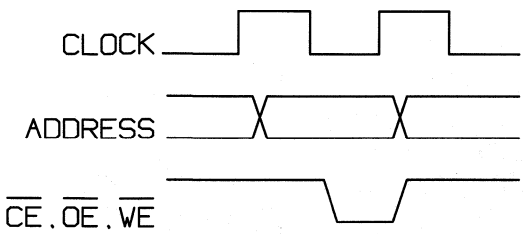


Figure 12. Timing for Dual-Port Example

addressing the same bank as the left port (more on this later).

Right-Port Operation

For purposes of this discussion, "word" refers to the 32-bit word at the right-port system-bus interface. At the 16-bit processor interface, the 32-bit word is referred to as either the lower half word (right-port bits 0 through 15) or the upper half-word (right-port bits 16 through 31).

The bank-selection process employs the chip enables. Specifically, the 1-of-4 RB decoder decodes the four combinations of the upper two right-port address-bus signals and generates four active-LOW chip enables to each bank of four dual-port RAMs. Bank A contains addresses 0 through 2047, bank B contains addresses 2048 through 4095, bank C contains addresses 4096 through 6143, and bank D contains addresses 6144 through 8191. In other words, bank A addresses 0 to 2K, bank B 2K to 4K, bank C 4K to 6K, and bank D 6K to 8K.

The lower 11 right-port address lines, AR(0:10), are connected to the A0 through A10 right-port address pins of all the dual-port RAMs.

Figure 12 does not show the generation of the write strobe, but does show the signal's timing. The write enable is applied directly to all the masters in parallel, then buffered, and then applied to all the slaves. The minimum propagation delay of the buffer must be at least as large as t_{BLA}, which is the time required for the master to assert the busy signal to the slaves after an address match occurs.

Note that all the right-port output-enable pins are connected together. These pins should be driven if reading is required; otherwise connect them to V_{CC}.

The open-drain busy outputs of the right port masters must be pulled up to V_{CC} using resistors. A value of 330Ω is recommended. The master busy outputs connect to all the right-port slave busy inputs for each bank.

For the data bus interface, the I/O pins of each RAM column connect to their respective I/O pins on each bank. This OR-tie connection is allowed because the bank-selection chip enable causes the output buffers of the unselected banks to go to the high-impedance state.

Left-Port Operation

The 1-of-4 decoder labeled LB performs bank selection for the left port. The upper two left-port address lines, AL13 and AL12, decode bank-select chip enable signals for the four masters only. Bank A corresponds to addresses 0 through 4095, bank B corresponds to addresses 4095 through 8191, bank C corresponds to addresses 8192 through 12,287, and bank D corresponds to addresses 12,288 through 16,383.

To perform upper and lower half-word selection, the 1-of-8 decoder labeled UL decodes the upper three right-port address signals. The decoder then generates eight chip enable signals with a resolution of 2048. The chip enables connect to the slaves' chip-enable and output enable pins (2048 resolution) and to the masters' output enable. Because the master chip enable resolution is 4096, the master arbitrates for two blocks of 2048 16-bit half words.

The lower eleven left-port address lines, AL(0:10), connect to left-port address pins A0 through A10 of all the dual-port RAMs.

At the 16-bit interface, writing is only required if the left port wishes to send a message to the right port. Otherwise, you can connect the left-port write pins of all the dual-port RAMs to V_{CC}.

To implement the left-port data bus interface, the left port's data I/O pins are connected together in the same manner as those of the right port for all RAMs in the same column. In addition, to multiplex

a 32-bit data word to a 16-bit half word, the least-significant bytes and the most-significant bytes of each 2048-word group are connected together. The UL decoder that controls the left-port output enable performs the selection.

If you use the masters' interrupt pins, pull them up to V_{CC} through a 330Ω resistor and connect them to the processor interrupt-request input. You can leave the slaves' interrupt pins unconnected.

If the control signal connections from their source to the dual-port memory constitute electrically long lines, they might require proper termination to avoid voltage reflections due to impedance mismatches. Refer to Cypress's application note titled "Systems Design Considerations When Using Cypress CMOS Circuits."

References

1. Dijkstra, E.W., "Solution of a Problem in Concurrent Programming Control." *CACM*, Vol 8, no.9, Sept. 1965, p 569.

2. Dijkstra, E.W., "Co-operating Sequential Processes." *Programming Languages*, F. Genyus (Ed.) Academic Press, New York, 1968, pp 43 – 112.

Notes

1. The Interrupt function is not available at the 2K x 8 level in a 48-pin package.

Document #: SRA0001–0492

Understanding Large FIFOs

This application note explains the internal operation of the large FIFOs manufactured by Cypress and shows how to use the devices to accomplish depth and width expansion. Other topics covered here include FIFO interfacing, the writing and reading process, failure modes, and typical problem symptoms and solutions. This information applies to the following Cypress FIFOs: CY7C420, CY7C421, CY7C424, CY7C425, CY7C428, CY7C429, CY7C432, CY7C433, CY7C439, CYM4210, and CYM4220.

Timing parameters given in this application note are taken from Cypress Semiconductor's *High Performance Data Book*.

Large FIFO Overview

The Cypress line of large FIFOs provide densities from 512 x 9 to 4K x 9 in monolithic devices; 8K and 16K x 9 in high-density modules; and a 2K x 9 bidirectional FIFO. Access times are as fast as 20 ns, and all the FIFOs feature identical, industry-standard pinouts. The monolithic devices are available in space-saving 300-mil-wide DIPs (odd-numbered devices) as well as industry-standard 600-mil-wide DIPs (even-numbered devices), and various surface-mount packages.

Although the first FIFOs utilized a shift-register type of architecture, today's large FIFOs employ an SRAM type of interface. Data is written into and read out of the devices, as with SRAM write and read operations. These operations can occur independently of one another and are made possible by a specially designed six-transistor, dual-ported SRAM cell. This cell makes use of separate read and write transistors to allow independent R/W operation.

Operating these FIFOs at their maximum throughput rates demands the generation of narrow write and read pulses. To facilitate significantly higher throughput rates, Cypress has developed the CY7C440 and CY7C450 families of clocked, or self-timed FIFOs.

These FIFOs feature 70-MHz operation and are characterized by self-timed interfaces. You generate the read and write enables, which are combined internally with the appropriate clocks. Thus, you do not need to generate narrow read and write pulses. These FIFOs also feature totally independent, asynchronous, read and write operations.

Each FIFO is organized such that data is read out in the same sequential order in which it was written. Full, half-full and empty flags facilitate writing and reading. Additional pins are provided to facilitate unlimited expansion in width and depth, with no performance penalty.

Writing to and Reading from the FIFO

Figure 1 shows the large FIFOs' read and write timing. Reads and writes are asynchronous to each other. The read process begins with $\overline{R}$'s falling edge. The output data bus, Q0 – Q8, leaves the high-impedance state t_{LZR} ns after $\overline{R}$'s falling edge. The output data becomes valid t_A ns after that same falling edge. This t_A period is referred to as the FIFO's read access time. $\overline{R}$'s rising edge ends the read process.

The data on the Q0 – Q8 bus remains valid for t_{DVR} ns following the $\overline{R}$ rising edge. This is the output data hold time at the end of the read cycle. The internal circuitry then readies itself for the next read operation. This period is referred to as the t_{RR}, or read recovery time, and must be observed between

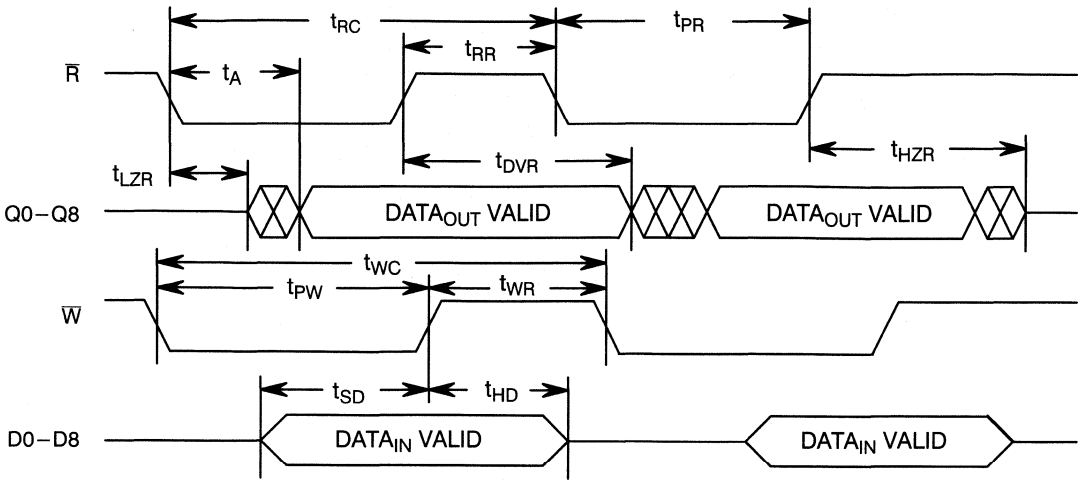


Figure 1. Asynchronous Read and Write Timing

consecutive read operations. The read signal's minimum pulse width is denoted by t_{PR} and is identical to the read access time, t_A.

You can determine the read cycle time (t_{RC}) by adding the access time (t_A) and the read recovery time (t_{RR}), which you can find in the FIFO data sheet. The maximum read frequency is the reciprocal of $t_A + t_{RR}$. For example, a Cypress FIFO with a 20-ns access time and a 10-ns read recovery time results in a 30-ns read cycle time, or 33.3-MHz maximum read cycle frequency.

The write process is similar to the read process. A write begins with the falling edge of the write line, $\overline{W}$, and terminates with $\overline{W}$'s rising edge. For a valid write to occur, the input data bus, D0 – D8, must be stable for t_{SD} ns prior to $\overline{W}$'s rising edge and for t_{HD} ns after this edge. These specifications are referred to as the data set-up and hold times, respectively. The write strobe also has a minimum negative pulse width, denoted as t_{PW}. A minimum recovery time, t_{WR}, is required between write cycles.

The maximum write frequency is the reciprocal of $t_{PW} + t_{WR}$. As an example, a device with a 20-ns write strobe width and a 10-ns write recovery time yields a 30-ns write cycle time, or a 33.3-MHz maximum write cycle frequency.

The FIFOs include separate write and read counters (pointers). Each write or read operation increments the appropriate counter one position. When the FIFO is empty, both counters point to the same location. The relative position of these counters determines the device's status, which is indicated externally via empty, half-full, and full flags.

Applications

FIFOs are asynchronous devices that are ideal for interfacing between two asynchronous processes. A FIFO allows two systems running at different data rates to communicate by providing a temporary data or control buffer.

Typical FIFO applications include

- Interprocessor communications, in which bidirectional devices are especially useful
- Communications systems, including local area networks
- Digital-signal-processing-based systems for buffering real-time data

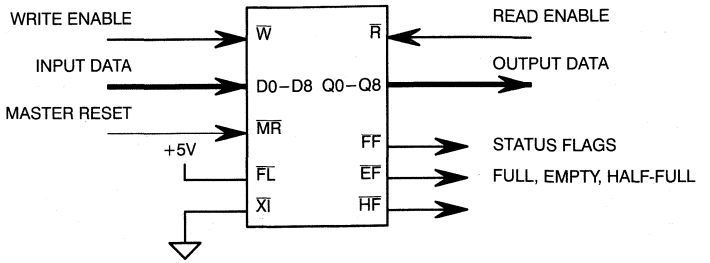


Figure 2. Standalone Operation

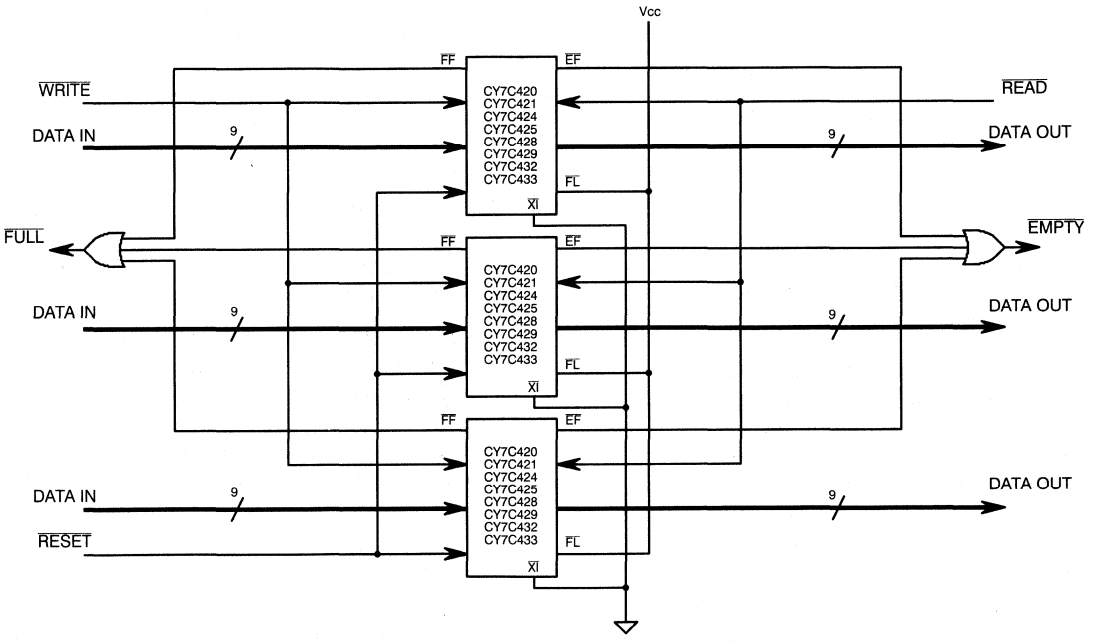


Figure 3. Width Expansion

- Electronic data processing, CPU, and peripheral equipment, including high-performance disk controllers

Common FIFO Configurations

Every Cypress FIFO, from the 512 x 9 CY7C420/21 to the 4K x 9 CY7C432/3, is fully cascadable. Width expansion allows you to create word widths of any multiple of nine bits. Cascading in depth creates FIFOs of various depths. Width and depth expansion modes are described here, along with design considerations.

Figure 2 illustrates standalone mode, and *Figure 3* shows width expansion mode. In both these modes, the $\overline{XI}$ (expansion in) pin is grounded and the FL (first load) pin is tied HIGH.

The OR gates in the width-expansion design generate composite full, half-full, and empty flags (F, H-F, E). Composite flags are necessary because variations in propagation delays might prevent the indi-

vidual FIFOs in the design from entering the F, H-F, or E states simultaneously. A composite flag properly reflects the instantaneous status of the entire word.

Figure 4 illustrates depth expansion. The $\overline{FL}$ (first load) pin on one device must be grounded to define that FIFO as the first FIFO to be written to. The FIFOs are then daisy-chained together by connecting one device's $\overline{XO}$ (expansion out) output pin to the next device's $\overline{XI}$ (expansion in) input. The $\overline{XO}$ of the last device in the chain is connected to the $\overline{XI}$ of the first device, thus forming a token-passing ring.

Token passing allows the writing and reading processes to stay consistent. That is, the passing and holding of a read or write token tells an individual FIFO whether it is actively being read from or written to. In the token-passing procedure for write operations, the first FIFO is written to until it is filled. An internal write pointer determines the location written to, and after every write, the pointer is incremented. When the pointer reaches the last physical location, no more writes can occur to that device. At that point, the first FIFO passes the write token to the next FIFO in the chain via the $\overline{XO}-\overline{XI}$ interface. The second device, now in possession of the write token, receives all future written data until this device also fills up and passes the write token onto the next device in the chain.

If enough writes occur to fill up the FIFO chain, the last device fails in its attempt to pass the write token back to the first device. This is because the full FIFO cannot accept a write token. No further writes to the FIFO chain are allowed until a read operation occurs, which frees up an internal location. The relative positions of the internal write and read counters determine a device's status and whether it can accept data though a write operation. *Figure 5* shows the timing for write operations.

As with the procedure for writes, the first FIFO in the chain holds the read token. When the FIFO chain is read from, the device holding the read token

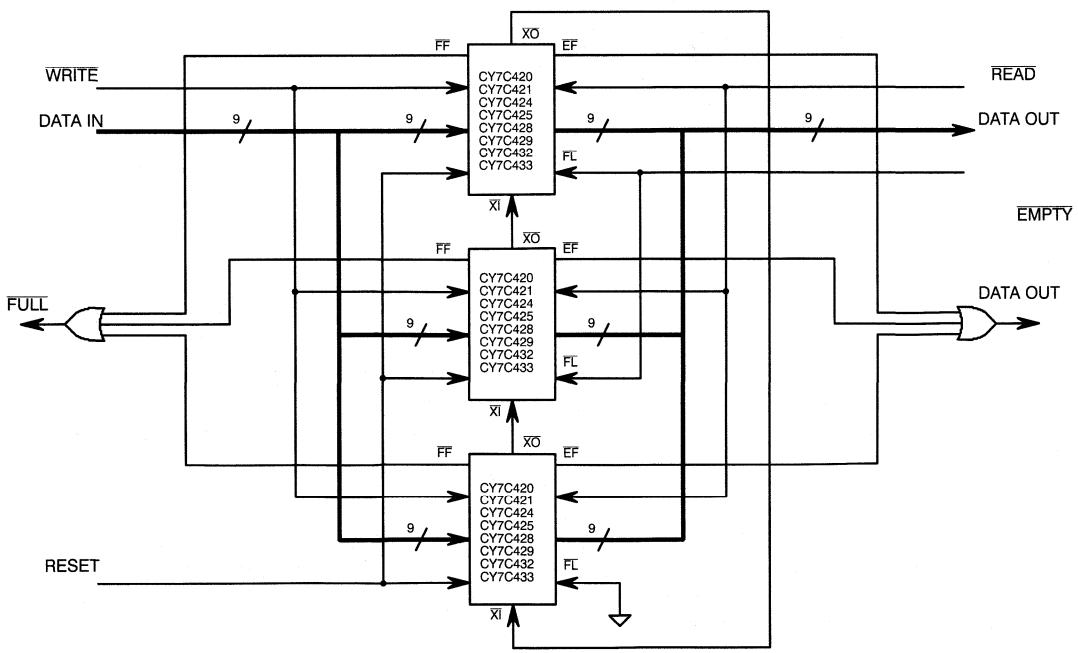


Figure 4. Depth Expansion

supplies the data from the address specified by the device's read pointer. The read pointer is then incremented. The incrementing continues until the FIFO is empty, and the read token is passed to the next device in the chain. The passing of the read token is done via the $\overline{XO}-\overline{XI}$ interface. *Figure 6* shows the timing for read operations.

A depth-expansion design must generate composite status flags to adequately reflect the instantaneous state of the FIFO chain, as is done for width expansion.

Retransmit

The retransmit feature is useful in communications for retransmitting packets of data and in disk drives for rewriting sectors. It is especially useful in applications where a single block of data in the FIFO must be sent out multiple times, as in a word or pattern generator.

Data can be retransmitted any number of times, and with Cypress FIFOs, the retransmit feature can be used at any time, no matter how much data the FIFO contains. This is in contrast to some compet-

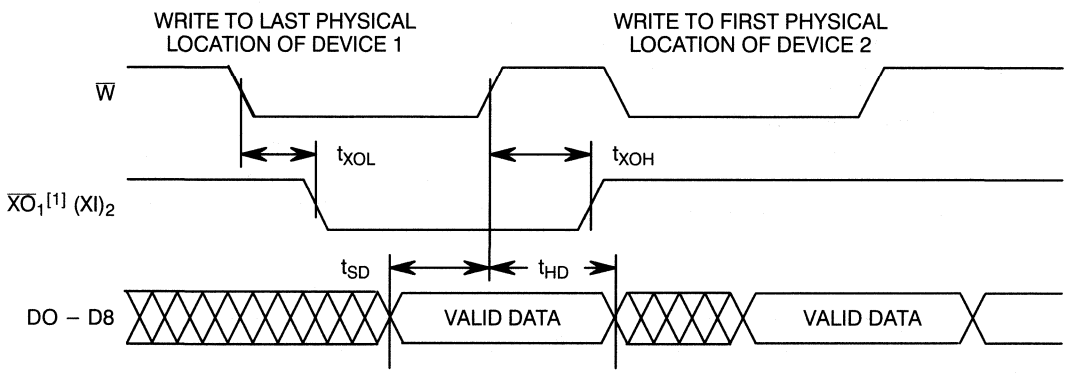


Figure 5. Write Expansion Timing

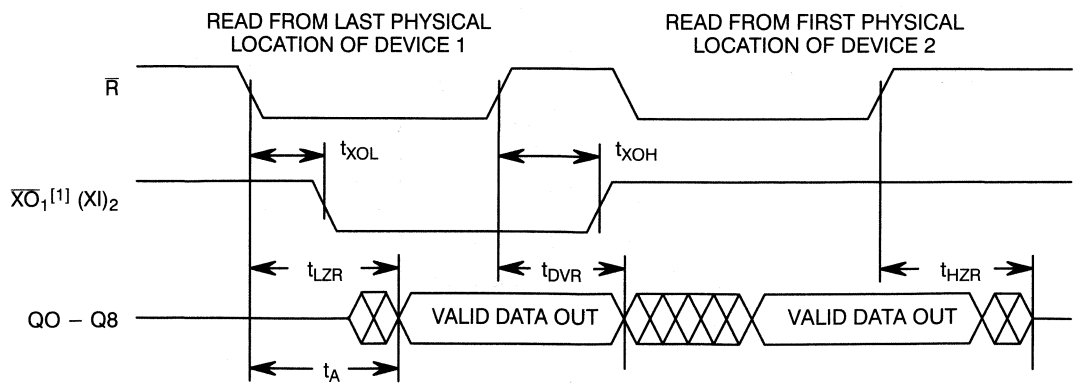


Figure 6. Read Expansion Timing

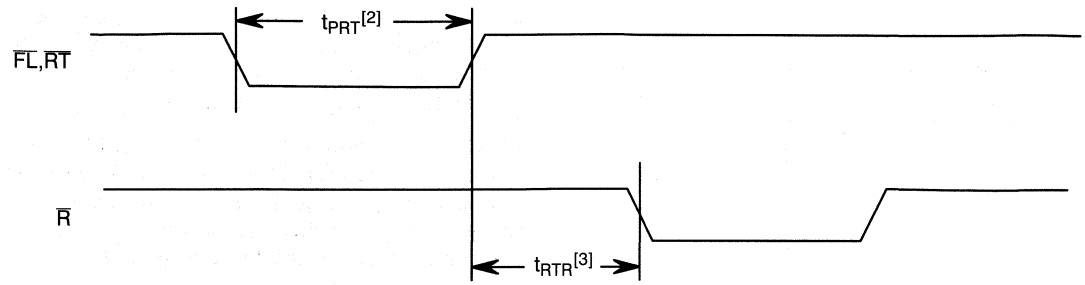


Figure 7. Retransmit Timing

ing FIFOs, such those from IDT, which do not allow use of the retransmit function when the FIFO is full.

In the retransmit operation, the read pointer is reset to its initial location and the $\overline{R}$ pin is pulsed until the read pointer advances to the same memory location addressed by the write pointer. The retransmit ($\overline{RT}$) pin is available in the single-device and width-expansion modes, but not in depth expansion because this pin designates the FIFO to be loaded first.

The retransmit function is initiated by asserting an active-LOW pulse to the retransmit input, which resets the internal read counter to zero. Keep the $\overline{R}$ input inactive during this time; otherwise, the conflicting requirements on the read counter might cause it to become corrupted. The retransmit process does not affect the state of the write counter or the write process, though the retransmit timing constraints shown in *Figure 7* must not be violated.

Note that the architectural description in the 1990 and previous Cypress data books incorrectly stated that the $\overline{W}$ input must be inactive during a retransmit cycle. No design or usage rules are violated if retransmit and write cycles overlap or occur simultaneously; the device does not lock up, and data is neither lost nor corrupted.

The reasons for the data book's retransmit/write restriction are more historical and application-oriented than functional. Specifically, the first large FIFOs did not permit writes during a retransmit cycle. This set a documentation precedent that all future devices had to match.

Additionally, keeping track of what data is currently in the FIFO and what data is being read out can become complicated. For example, if a FIFO is half full and the retransmit function is activated and writes continue, filling the FIFO to three quarters full before the read pointer catches up with the write pointer, the FIFO outputs all of the data.

Common Problems and Solutions

To help prevent problems and correct them when they occur, this section describes the causes and solutions to some common FIFO problems. The first problem to consider is corrupted or repetitive data in a FIFO.

Corrupted or Repetitive Data

The most common cause of corrupted and repetitive data being present in a FIFO is a spurious active signal (glitch) on the FIFO's $\overline{W}$ input. Because Cypress devices are extremely fast, a write pulse as short as 3 ns initiates a write. Write glitches cause whatever logic levels are present at the data inputs to be written into the FIFO, which can put false data into the device. If valid data is present at the data inputs, a write glitch causes this data to be written a second time, resulting in duplicated data.

Write glitches are often the result of voltage reflections due to impedance mismatches, which you can eliminate using impedance-matching termination networks. Termination networks are recommended on the $\overline{W}$ and $\overline{R}$ traces on printed circuit boards (PCBs) when the lines exceed approximately 4 in-

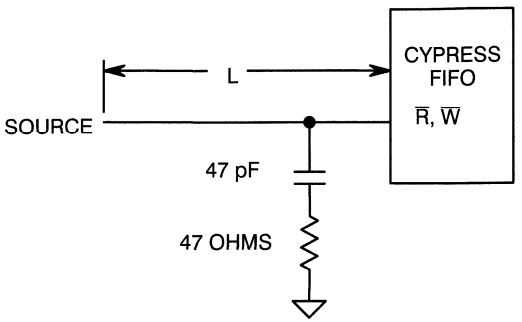


Figure 8. Recommended Termination Network

ches from source to a single load. This line length assumes a 2-ns rise/fall time for the read and write strobes. For $\overline{R}$ and $\overline{W}$ signals with sub-2-ns rise/fall times, line lengths as short as 1 inch might require termination.

A termination network matches the load impedance to the PCB trace's characteristic impedance, which is typically 50Ω or less for microstrip or stripline construction on G–10 glass epoxy material. To minimize voltage reflections, a slightly overdamped termination is preferred. Cypress recommends a 47-pF (max.) series capacitor and a 47-ohm resistor be connected from the read or write pin to ground (*Figure 8*). This termination network acts as a high-pass filter to short, high-frequency pulses and dissipates no DC power. Read or write lines that drive more than one FIFO require only one termination network. Put the network at the input that is electrically farthest from the source. For multiple loads, see the "Systems Design Considerations When Using Cypress CMOS Circuits" application note for help in determining the maximum line length.

FIFO data corruption can also be caused by violation of master-reset timing constraints. As shown in the timing diagram in *Figure 9*, the read and write signals must be inactive around the rising edge of $\overline{MR}$ (master reset) to satisfy the t_{RMR}, or master-reset recovery-time specification. This constraint is necessary because the FIFO goes through an internal initialization process during reset and requires a settling period after the reset terminates.

FIFO Locks Up

Short noise pulses on the FIFO's master reset pin can cause the FIFO to not respond because it is "partially reset." If this problem occurs, you might need to terminate the master reset line.

Missing or Disappearing Data

Glitches on the $\overline{R}$ input can cause data to disappear because of an unintended read operation. The read increments the internal read counter, resulting in the loss of the current data word. Here again, a termination network eliminates the unwanted glitches.

Repetitive or Out-of-Sequence Data, False Full or Empty

A misaligned internal read or write pointer can cause a variety of symptoms, including repetitive or out-of-sequence data and false full and/or empty

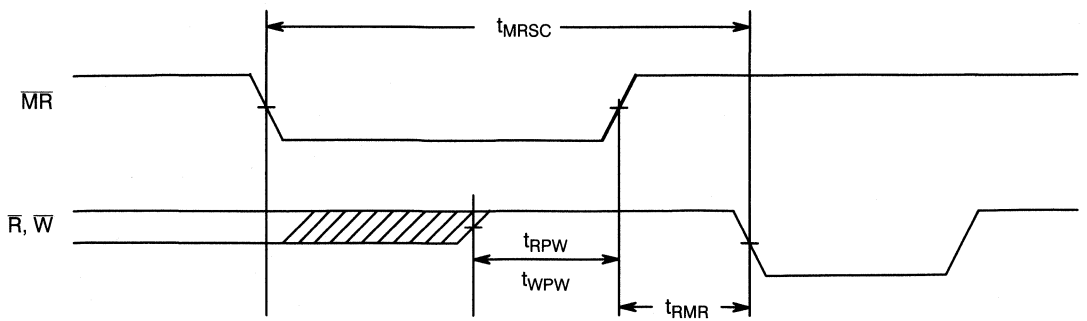


Figure 9. Master Reset Timing

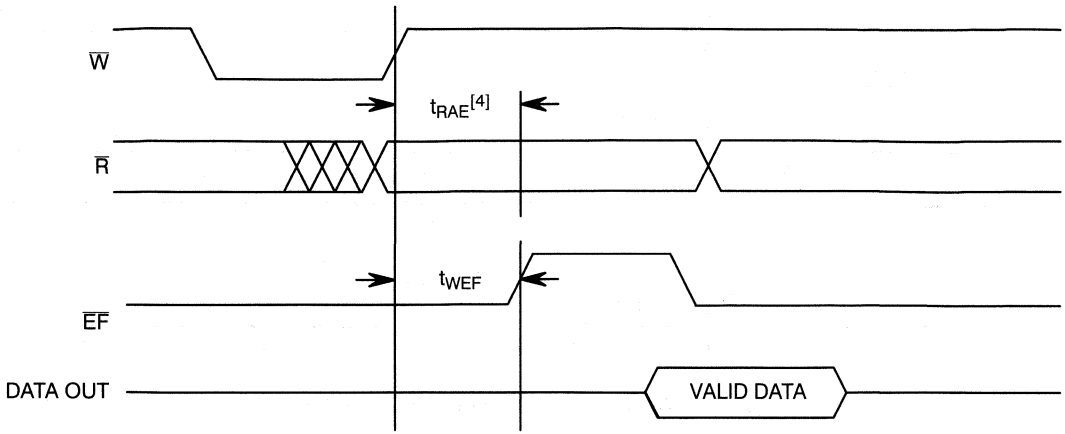


Figure 10. Read Fall-Through Timing Violation

conditions. The two most common causes of misaligned pointers are master-reset violations and boundary-condition violations.

Boundary conditions are defined as the FIFO being either full or empty. When high-density FIFOs are connected in parallel to make a wider word, certain conditions can cause the FIFOs to choose individually to either ignore or act upon a read or write request. The system-level symptom of individual FIFOs making different decisions is word misalignment. The problem occurs in the empty condition when a read immediately follows a write and in the full condition when a write immediately follows a read.

Operation at the Empty Boundary

Consider a FIFO that has been reset and is empty. The empty flag is active (LOW), and internal logic inhibits read operations. In the general case, the read and write signals are asynchronous. Upon completion of the write operation the internal state of the FIFO goes from empty to empty + 1. During this interval, a read operation might or might not be recognized. A read preceding the write is ignored; a read following the write is not. In between these conditions, the FIFO decides whether to recognize the read. During this aperture of uncertainty, you cannot determine whether the read will be ignored or not. With one FIFO, this uncertainty is acceptable. However, if two or more FIFOs are connected in parallel to make a wider word, some might ignore the read, and others might not.

Operation at the Full Boundary

A similar condition occurs when a single FIFO becomes full. The full flag is active (LOW), and internal logic inhibits write operations. A read operation immediately followed by a write operation causes the FIFO to go from full to full − 1 and back to full. During the time the FIFO is going from full to full − 1, a write operation might or might not be recognized. The aperture of uncertainty applies here because the FIFO takes a finite amount of time to change states, and a write command arriving at this instant might be ignored.

Waiting at the Empty Boundary

Figure 10 shows the timing that prevents problems with reads at the empty boundary. Any device reading from the FIFO must wait an amount of time, t_{RAE}, after the termination of the write operation before causing a HIGH-to-LOW transition of the $\overline{R}$ signal. The $\overline{W}$ signal's rising edge indicates the termination of the write operation.

One way to satisfy this timing is to gate read operations with the composite empty flag ($\overline{EF}$) such that

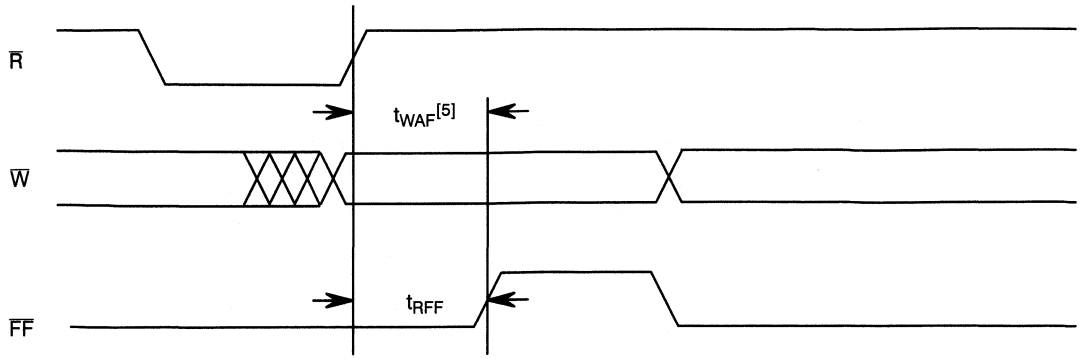


Figure 11. Write Bubble-Through Timing Violation

the read operation is prevented when the empty flag is active. Note, however, that the $\overline{R}$ signal can be LOW either before or during the first write to the empty FIFO and the data still propagates to the outputs correctly.

Waiting at the Full Boundary

Figure 11 shows the timing that prevents problems with writes at the full boundary. Any device writing to the FIFO must wait an amount of time, t_{WAF}, after the termination of the read operation before causing a HIGH-to-LOW transition of the $\overline{W}$ signal. The $\overline{R}$ signal's rising edge indicates the end of the read operation.

You can meet this timing by gating write operations with the composite full flag ($\overline{FF}$) such that the write operation is prevented when the full flag is active. However, the $\overline{W}$ signal can be LOW either before or during the first read from a full FIFO and the data is still properly written.

Empty Reads and Full Writes

When Cypress FIFOs are empty, their data outputs go to the high-impedance state. Therefore, attempting to read from an empty FIFO yields unpredictable data. Internal logic inhibits the read, and the read pointer is not incremented.

Internal logic also inhibits attempts to write to a full FIFO, and the write pointer is not incremented.

Intermittent Malfunctions

If all the timing requirements appear to be met and data in the FIFO is still corrupted, the cause is likely to be noise on the power supply. Random spikes on either the V_{CC} or ground pins of the FIFO are likely culprits when non-repeatable failures occur.

The cure for this problem is to add a high-pass filter capacitor between the device's power and ground pins. This practice is recommended whenever the read or write frequency exceeds 5 MHz. Use a very small (100 – 500 pF) ceramic or mica capacitor. Surface-mounted capacitors are recommended because they have at least an order of magnitude less lead inductance than radial or axial leaded capacitors.

The filter capacitor is in addition to the 0.1- or 0.01-μF decoupling capacitor that should always be present with any high-speed digital chip. Although decoupling capacitors are often referred to as bypass capacitors—implying filtering properties—their true function is to supply the instantaneous current required when many or all device outputs simultaneously switch from LOW to HIGH. This larger capacitor thus decouples or isolates the IC from the power distribution system.

Notes

1. Expansion out of device 1 (XO_1) is connected to expansion in of device 2 (XI_2).
2. t_{PRT} is the minimum retransmit pulse width.
3. t_{RTR} is the retransmit recovery time. It is a timing window that must not be violated.
4. t_{RAE} is an invalid read window. A read operation should never be initiated inside this window.
5. t_{WAF} is an invalid write window. A write operation should never be initiated inside this window.

Document #: LOG0001–0592

Understanding Clocked FIFOs

Introduction

This application note explains the basic operations and features of Cypress clocked FIFO memories. Cypress clocked FIFOs are ideally suited for applications requiring high data throughput and asynchronous data buffering. The clocked FIFO interface simplifies high-speed design and provides greater noise immunity over industry-standard asynchronous FIFOs.

Design considerations of the clocked FIFO architecture are examined, including proper flag operation and decoding, FIFO boundary operation, and resetting and programming the FIFO. FIFO depth and width expansion are also covered.

The Cypress family of clocked FIFOs are available in several densities with a variety of features. *Table 1* outlines the features of Cypress's clocked FIFOs. The entire clocked FIFO family feature fully asynchronous operation at clock rates of up to 70 MHz in non-depth expansion mode. Clocked FIFOs cascaded for depth expansion can operate at frequencies of up to 50 MHz.

The CY7C441 and CY7C443 feature 512 and 2K word by 9 bit memory arrays, respectively. These FIFOs feature high-speed operation and $\overline{\text{Empty}}$, $\overline{\text{AlmostEmpty}}$, and $\overline{\text{AlmostFull}}$ flags, center power and ground pins, and width expandability. Both FIFOs are available in either a 32-pin PLCC/LCC package or a 28-pin DIP package.

The CY7C451 and CY7C453 clocked FIFOs have all of the features of the 7C44X FIFOs plus $\overline{\text{Full}}$ and $\overline{\text{HalfFull}}$ flags, programmable $\overline{\text{AlmostEmpty}}$ and $\overline{\text{AlmostFull}}$ flags, parity generation and parity checking, output enable ($\overline{\text{OE}}$), and depth expandability. The 7C451 features a 512 word by 9 bit memory array and the 7C453 features a 2K word by 9 bit memory array. Both FIFOs are available in either a 32-pin PLCC/LCC package or a 32-pin DIP package.

Table 1. Features of Cypress Clocked FIFOs

FIFO	Density	Speed	Flag Architecture	Parity	Output Enable	Depth Expandable	Width Expandable
7C441	512 x 9	71.4 MHz	Synchronous	No	No	No	Yes
7C443	2048 x 9	71.4 MHz	Synchronous	No	No	No	Yes
7C451	512 x 9	71.4 MHz	Synchronous, Programmable	Programmable	Yes	Yes*	Yes
7C453	2048 x 9	71.4 MHz	Synchronous, Programmable	Programmable	Yes	Yes*	Yes

* 50 MHz in this mode

Clocked Architecture

The clocked FIFO architecture is designed to achieve maximum performance from FIFO memories while simplifying their use in a system. Timing pulses for the memory array are generated internally from the read and write clocks thus eliminating the need for generating very narrow external read and write pulses.

The read and write ports have separate clock inputs (CKR, CKW), and read and write operations are enabled through separate clock-enable pins ($\overline{ENR}$, $\overline{ENW}$). The read and write clocks can be fully asynchronous. *Figure 1* demonstrates asynchronous reading and writing to a clocked FIFO.

The clocked FIFO interface is ideally suited for state machine control. A state machine can perform reads or writes by simply asserting the respective enable lines LOW. It is not necessary to toggle the enable lines to perform consecutive operations.

FIFO Writes

Figure 2 shows a simplified block diagram of the clocked FIFO data path. The internal write control logic circuitry controls the input register, the write pointer, and the write port of the dual-ported memory array.

This write operation is similar to writing to a standard 377 register. The FIFO input register is clocked by CKW and enabled by $\overline{ENW}$. Data is clocked into the FIFO on the enabled rising edge of CKW. The data is then written into the memory location pointed to by the write pointer, provided the FIFO is not full ($\overline{Full}$ = 1). The write pointer is then incremented. A full FIFO will ignore any attempted write without upsetting the memory array or the flags. The 70-MHz clocked FIFOs have a data and enable set-up time (t_{SD} and t_{SEN}) of 7 ns.

FIFO Reads

The internal read control logic circuitry controls the output register, the read pointer, and the read port of the dual-ported memory array. The output register holds the word that was last read from the FIFO memory array. This register is loaded from the memory array in a manner similar to loading a standard 377 register. The output register is clocked by CKR and enabled by $\overline{ENR}$. Note that the CY7C45X family of clocked FIFOs feature a three-state output register controlled by $\overline{OE}$.

The read pointer points to a word in the memory array. That word is loaded into the output register

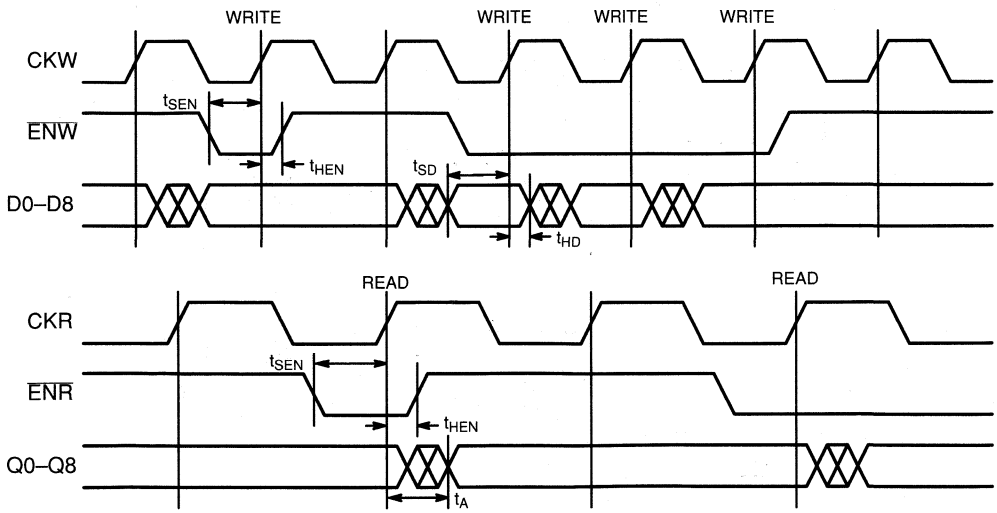


Figure 1. Asynchronous Writing and Reading to a Clocked FIFO

on the enabled rising edge of CKR, provided the FIFO is not empty ($\overline{Empty}$ = 1), and the read pointer is then incremented. The word is available at the output pins t_A after the clock edge. An empty FIFO will ignore the attempted read and continue to hold the last word in its output register. The set-up time for $\overline{ENR}$ (t_{SEN}) is 7 ns and the data access time (t_A) is 10 ns for a 70-MHz clocked FIFO.

Flag Architecture

Cypress clocked FIFOs feature a synchronous encoded flag architecture that simplifies FIFO integration into a synchronous system. Synchronous flags guarantee that a flag update is only triggered by a rising clock edge. The state of a flag is guaranteed to be valid t_{FD} after the rising clock edge.

Unclocked asynchronous FIFOs can generate narrow flag pulses with indeterminate timing based on the timing relationship of read and write pulses. External flag synchronization logic is required in synchronous designs using unclocked FIFOs. The Clocked FIFO architecture eliminates these short flag pulses and avoids the need for external flag synchronization logic.

A small package footprint is maintained by encoding the state of the flags. Pin count and package size are reduced and fewer PCB board signals require routing. Only two signals are needed to encode four states of the 7C44X FIFOs and three signals encode six states of the 7C45X FIFOs.

The FIFO flags are easily decoded inside a programmable control unit or a state machine controller. Decoding the signals properly produces flags synchronized to a single clock. *Figures 3* and *4* show a block diagram of the flag architecture for both the 7C44X and 7C45X FIFOs. The diagrams also show the external logic needed to decode and synchronize the flags.

The decoded Empty-type flags are synchronized to the read clock (CKR) and decoded Full-type flags are synchronized to the write clock (CKW). The CY7C45X family of Clocked FIFOs features a Programmable Almost Full/Empty flag ($\overline{PAFE}$) that is synchronized to the read and write clocks.

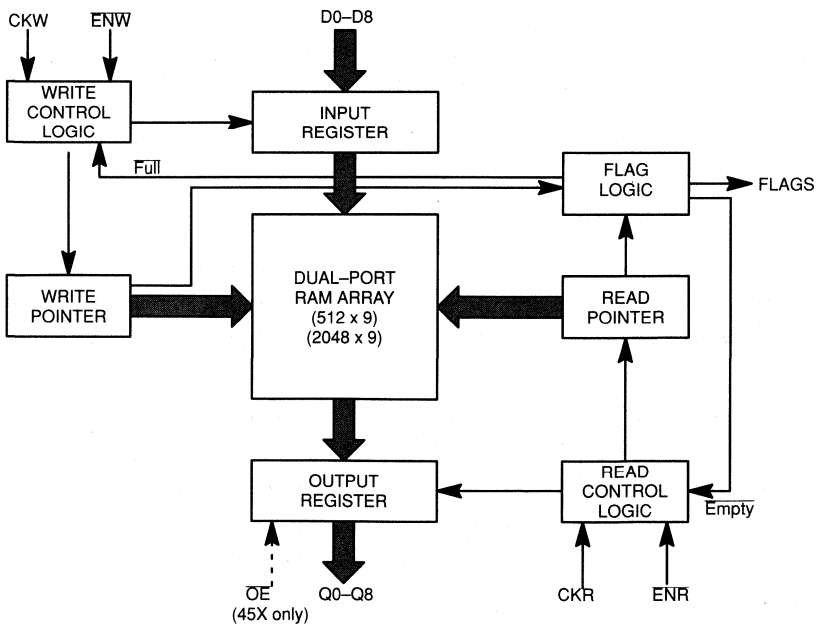


Figure 2. Clocked FIFO Data Path

Reads and Writes with Boundary Flags

The Empty and Full flags are considered Boundary flags because they indicate that the FIFO has reached its boundary of operation. Attention must be paid to the status of these flags when operating the FIFO at or near the boundaries. The internal FIFO write and read control logic uses the Boundary flags to determine if an access to the memory array is possible. The internal write control logic will not attempt to write to the memory array or increment the write pointer if the FIFO is Full, as indicated by the registered $\overline{\text{Full}}$ flag. Similarly, the read control logic will not load the output register or increment the Read Pointer if the FIFO is empty, as indicated by the registered $\overline{\text{Empty}}$ flag (see *Figures 2, 3,* and *4*).

The boundary flags determine the state of the read and write logic control circuits inside the Clocked FIFO. Design considerations with boundary flags are explored in the next two sections.

Boundary Latency Cycles

A write or a read can cause the FIFO memory array to exit from an empty- or full-boundary condition. At the empty boundary, the FIFO write control logic will allow an enabled write clock to store a word in the memory array. However, the $\overline{\text{Empty}}$ flag synchronization register will not reflect the current state of the FIFO memory array until it is clocked by the read clock. Similarly, at the full boundary, the FIFO read control logic will allow an enabled read clock to remove a word from the memory array, but the $\overline{\text{Full}}$ flag synchronization register will not reflect the current state of the FIFO until it is clocked by write clock.

A FIFO latency cycle (update cycle) refers to the clock cycle that causes a boundary flag register to be updated with the current status of the memory array. During this cycle, only a boundary flag regis-

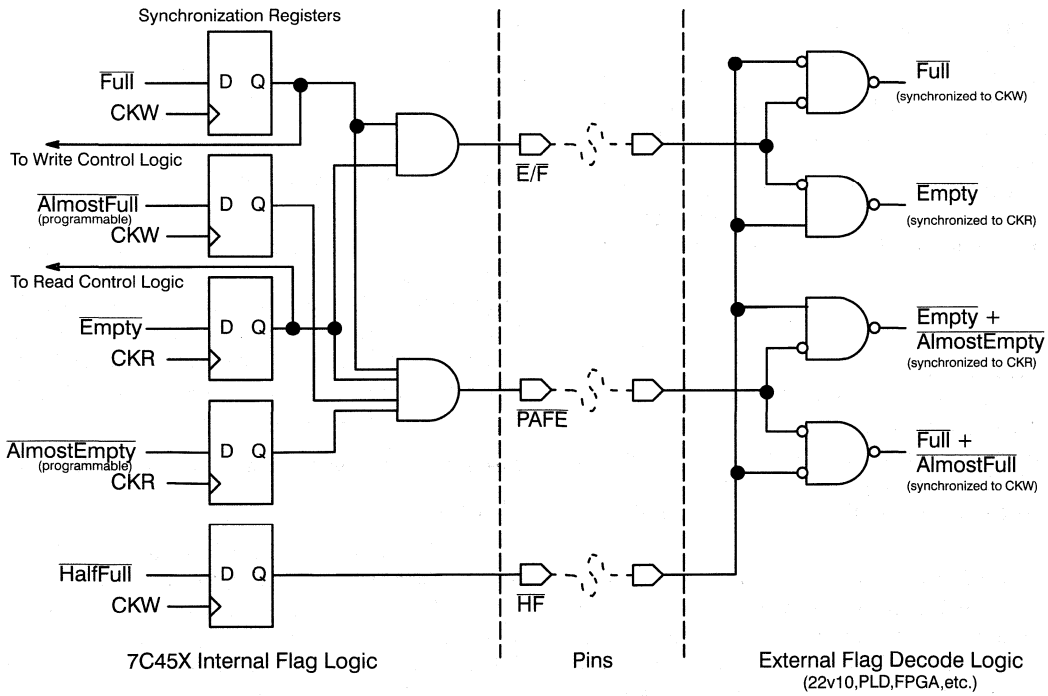


Figure 3. 7C45X Flag Architecture

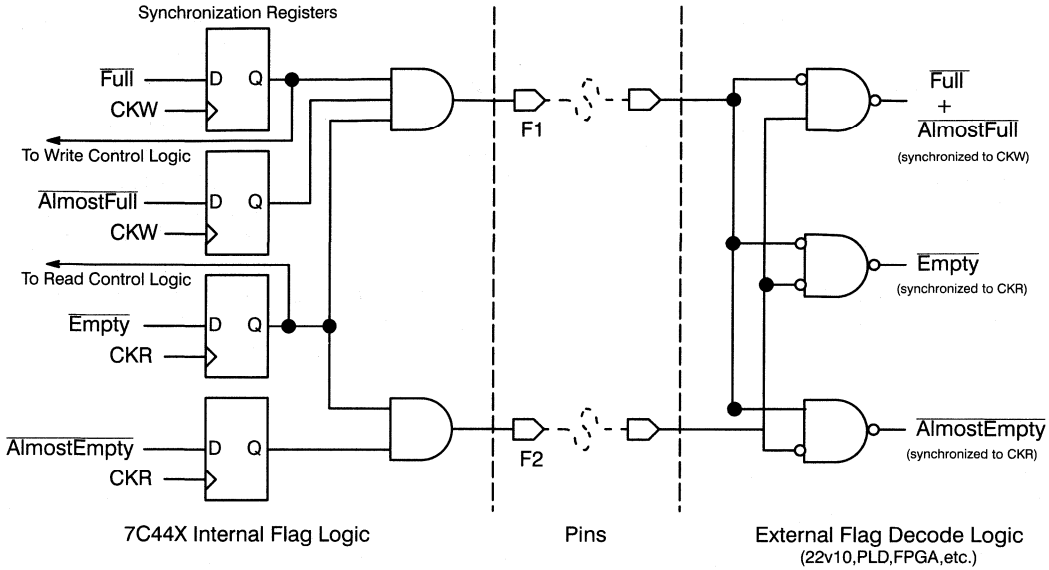


Figure 4. 7C44X Flag Architecture

ter is updated regardless of the state of $\overline{ENR}$ or $\overline{ENW}$. A read-clock latency cycle updates the $\overline{Empty}$ flag register from LOW to HIGH regardless of the state of $\overline{ENR}$. When the $\overline{Empty}$ flag register is in the HIGH state, an enabled read clock can retrieve data from the memory array. The overall effect is that after the FIFO memory becomes non-empty, it takes two read cycles to get the first word from the FIFO—one to update the flag and one to read the data.

Similarly, a write clock latency cycle updates the $\overline{Full}$ flag register from LOW to HIGH regardless of the state of $\overline{ENW}$. When the $\overline{Full}$ flag register is in the HIGH state, an enabled write clock can store data in the memory array. The overall result is that after the FIFO memory becomes non-full, it takes two write cycles to put the first word in the FIFO.

This type of flag operation is desirable because it guarantees that flags in the inactive (HIGH) state will be valid and usable for at least one clock cycle. This architecture eliminates indeterminate short flag pulses characteristic of asynchronous flag architectures.

Free-Running CKR and CKW Clocks

Boundary-operation timing and latency cycles should pose no problem in designs that employ free-running read and write clocks. Free-running clocks insure that flag update cycles will be performed automatically. The flag registers will be constantly updated with the current FIFO status.

Designs that do not use free-running clocks must explicitly issue a clock cycle near the FIFO boundaries in order to update the flag registers. Absence of free-running clocks may decrease system performance by causing the external control circuitry to wait for one clock cycle during the flag update cycle before performing an operation.

Resetting and Programming Clocked FIFOs

Master Reset

Clocked FIFOs are reset by pulsing the $\overline{MR}$ (Master Reset) pin LOW. Resetting the FIFO clears the read and write pointers so that they both point to location zero of the memory array, causing the FIFO to be Empty. The data output register will contain all 0s after the reset pulse occurs. Master Reset also resets the internal read and write control

logic circuits. The 7C45X family of clocked FIFOs can also be programmed during Master Reset. Programming the FIFO causes the program word to be stored in the FIFO program register.

Clocked FIFOs generate internal timing pulses off of the falling edge of $\overline{MR}$ in order to reset and program the internal FIFO control logic. For this reason, it is very important that the assertion of $\overline{MR}$ be glitch free. A narrow glitch of only a few nanoseconds while $\overline{MR}$ is LOW can be interpreted as a false edge and interrupt the reset timing sequence. As a result, the FIFO will not be fully reset or programmed.

To insure that Master Reset is glitch free, it is recommended that $\overline{MR}$ be driven by a flip-flop. In applications requiring a single Master Reset signal to reset or program multiple FIFOs, the FIFO pin farthest way from the flip-flop may need to be terminated in order to reduce glitches caused by voltage reflections. The need for terminations is a function of trace length, rise time, and PCB characteristics (see "System Design Considerations When Using Cypress CMOS Circuits," in the *Cypress Semiconductor Applications Handbook*).

The probability of improperly resetting a clocked FIFO due to glitches induced by ground bounce or other sources of noise can be reduced by using a Master Reset pulse that is as short as possible but is greater than t_{PMR}. Long reset pulses increase the chance that noise from somewhere in the system will be coupled to the $\overline{MR}$ pin through the ground plane. *Figure 5* shows a circuit for creating a short $\overline{MR}$ pulse from a long reset pulse. The duration of the $\overline{MR}$ pulse can be increased by adding more delay registers before the AND gate.

The proper reset sequence requires that enabled read and write cycles not be performed during or near the Master Reset pulse. Clock cycles that are not enabled by $\overline{ENR}$ or $\overline{ENW}$ are allowed during Master Reset. To insure that the clocks are disabled, $\overline{ENR}$ and $\overline{ENW}$ should not glitch LOW. Exact timing parameters are given in the data sheet. An easy way to insure that timing restrictions are met with a state machine is to insert pad states (clock enables HIGH) between the last read and write before Master Reset and between the first read and write after Master Reset.

Programming the 7C45X

The 7C45X family of clocked FIFOs can be programmed during the Master Reset cycle. Programming affects the $\overline{AlmostEmpty}$ and $\overline{AlmostFull}$ flags and sets the Parity. Programming is accomplished by writing data to the FIFO while asserting $\overline{MR}$ LOW. The program word is stored in the program register. The programming information may be ver-

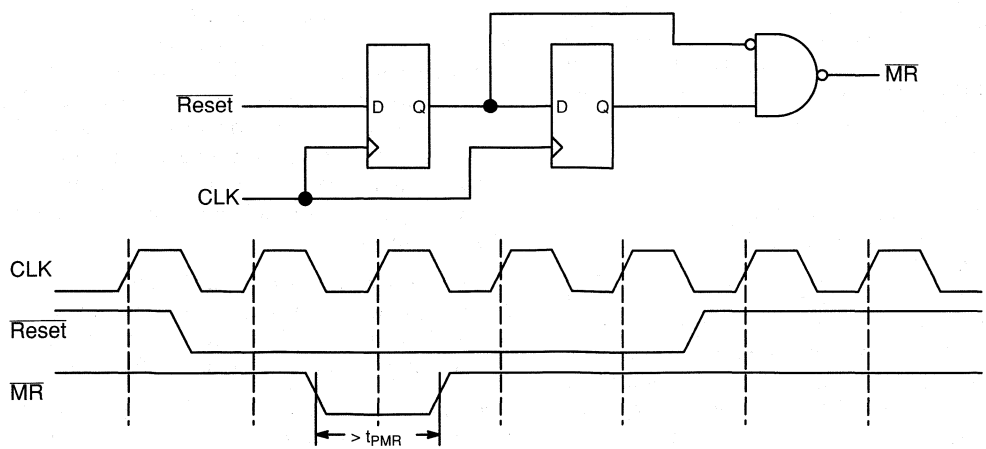


Figure 5. $\overline{MR}$ Pulse Generation

ified my reading the FIFO while $\overline{MR}$ is still asserted LOW. The FIFO program register is programmed to its default value if no write is performed during a Master Reset.

Data lines D0–D5 are are used to program the $\overline{AlmostEmpty}$ and $\overline{AlmostFull}$ flags. The value of D0–D5, which is written into the program register, determines the distance from the FIFO boundary flags ($\overline{Empty}$ and $\overline{Full}$) that these flags become active. The distance is programmable in 16-word increments and is determined by 16•P where P is the value of D0–D5. The $\overline{PAFE}$ pin encodes the programmable flag states.

Data lines D6–D8 program the FIFO parity option. D8 enables the Parity feature when set HIGH. D7 selects between Parity Generation and Parity Checking. Parity Generation is selected when D7 is LOW. D6 selects even parity when set LOW and odd parity when set HIGH.

Parity generation provides a simple means for systems to detect data bit errors. When enabled, the FIFO parity checker will examine bits D0–D7 being written into the FIFO before writing them into the memory array. The ninth bit (D8) will be set according to the parity mode set in the program register. Even-parity mode will set D8 such that the sum off all the bits including D8 is even. Odd-parity mode will set D8 such that the sum is odd. D8 is available on output line Q8/PG/$\overline{PE}$ during a read from the FIFO. Parity checkers down stream in the system can use D8 to determine when data has been corrupted.

The 7C45X can be configured as a parity error checker. During a write, data bits D0–D8 are examined before being stored in the memory array. D8 is set LOW if a parity error is detected. When set for for even parity checking, a parity error occurs if bits D0–D7 add to an odd number. Odd-parity checking will detect an error if D0–D8 add to an even number. D8 is written into the memory array with the rest of bits D0–D8. The parity-error bit (D8) is then available on Q8/PG/$\overline{PE}$ during a read from the FIFO.

Depth Expansion

The 7C45X Family of Clocked FIFOs feature depth expandability. Two or more 7C45Xs may be cascaded to achieve a single, large FIFO memory array. Depth expansion may be used in applications requiring buffering of large data packets, using extremely disparate read and write rates, or having long read latencies.

Depth expansion is achieved by cascading several FIFOs using the expansion pins. Data is automatically multiplexed from the FIFOs onto a single output bus using the FIFO's three-state output drivers. The flags must be combined to form composite flags. *Figure 6* shows two FIFOs cascaded for depth expansion.

The cascaded devices act as a single FIFO memory array. Read and write control is passed from one FIFO to another using the expansion pins. When a single FIFO has had all of its memory locations written to, it asserts the Expansion Out pin ($\overline{XO}$) signaling the next FIFO to begin writing to its array. Similarly, when the FIFO has had all of its memory locations read from, it deasserts the Expansion Out pin to signal the next FIFO to read data from its array. The FIFOs' expansion pins form a simple token ring.

The token-passing architecture necessitates the use of composite flags in order to detect when composite FIFO is in a boundary state (Full or Empty). In a long series of reads and writes, it is difficult to track which of the individual FIFOs possess the read and write tokens. The state of the composite FIFO could be determined by looking at the flags of the FIFOs in possession of these tokens, but this is difficult and unnecessary. Composite flags, shown in *Figure 6*, bypass this problem by looking at all the flags in parallel.

The First Load pin ($\overline{FL}$) indicates which device possesses the read and write tokens following a Master Reset. Only one device should have its $\overline{FL}$ pin tied to V_{SS}. All other devices should tie $\overline{FL}$ to V_{CC}.

The Almost Empty and Almost Full flags are not usable in depth expansion. The cascaded devices, however, can be programmed for parity. All cascaded devices will be programmed the same since

all control and data pins are common. Program read occurs automatically on the First Load device only to avoid bus contention.

Width Expansion

Both the 7C44X and 7C45X family of Clocked FIFOs can be width expanded for applications requiring data wider than 9 bits.

Width expansion is achieved by wiring the FIFOs in parallel. *Figure 7* shows two FIFOs wired for width expansion. Composite flags should be used to provide proper read and write signaling near the FIFO Empty and Full boundaries. Process variations between FIFOs can result in differences in t_{SKEW1} and t_{SKEW2}. This can cause the update cycles to occur on staggered clock cycles in different FIFOs. Data misalignment can occur at the boundary condition if an operation is performed before all FIFO flag registers are in the same state. Composite flag signaling insures that all FIFOs are in the same state so that an operation at the boundary is performed concurrently by all FIFOs.

The $\overline{\text{PAFE}}$ flag from either FIFO may be monitored and will give the correct status, or each FIFO may be programmed differently to give different $\overline{\text{PAFE}}$ flags. Parity generation/checking is performed in each device independently according to how they are individually programmed.

Using a Clocked FIFO Like a Standard FIFO

Applications that require high-speed unclocked asynchronous FIFOs memory may use clocked FIFOs. Unclocked asynchronous FIFOs operate at much lower frequencies than clocked FIFOs but feature read and write interfaces driven by single read and write strobes.

Applications can use clocked FIFOs to emulate this operation at high speeds by tying the clock to the appropriate enable line. The enable lines should not be tied straight to ground. Grounding the enable lines directly increases the probability of violating enable set-up times in a noisy environment. Tying the enables to the clocks closes the timing window (when noise can affect the enable pins) and filters out unwanted ground noise. The zero hold-time

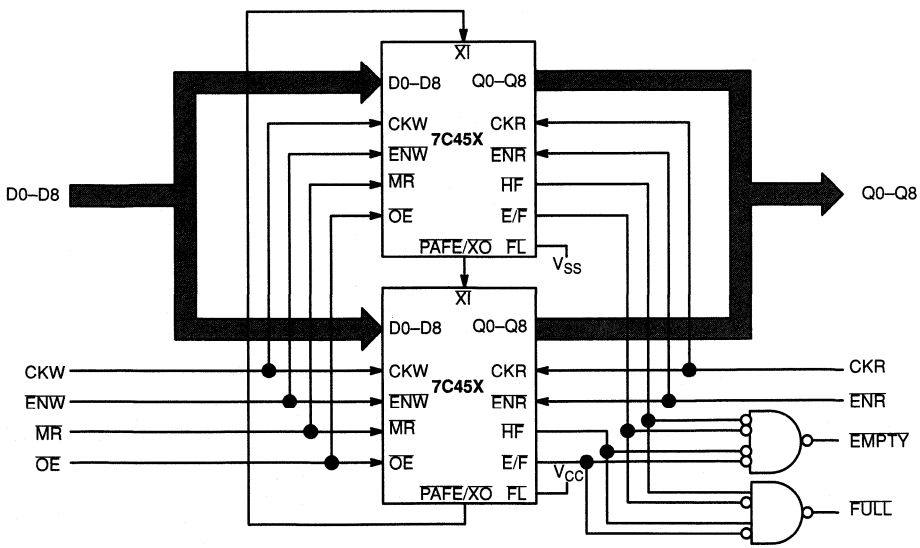


Figure 6. Depth Expansion with CY7C45X

feature of the enable makes this configuration possible. *Figure 8* shows a 7C45X configured as a standard FIFO.

A caveat occurs at the boundary condition flag timing. Absence of a free running clock will prevent the flags from being updated. As a consequence, the internal FIFO control logic will inhibit read or write operations if the respective flag is not updated. To avoid this problem, the FIFO in a boundary state must be strobed in order to force a flag update cycle. Data is not destroyed during the update cycle. The desired operation may proceed once the the flag is updated correctly.

For example, an empty FIFO with its empty flag asserted is written to by strobing the write port. The empty flag, however, is only updated by the rising

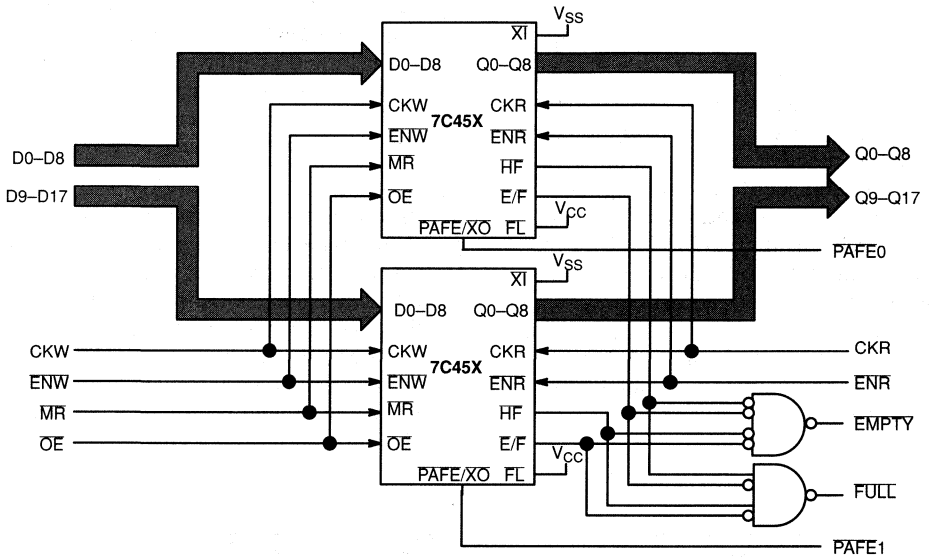


Figure 7. Width Expansion with CY7C45X

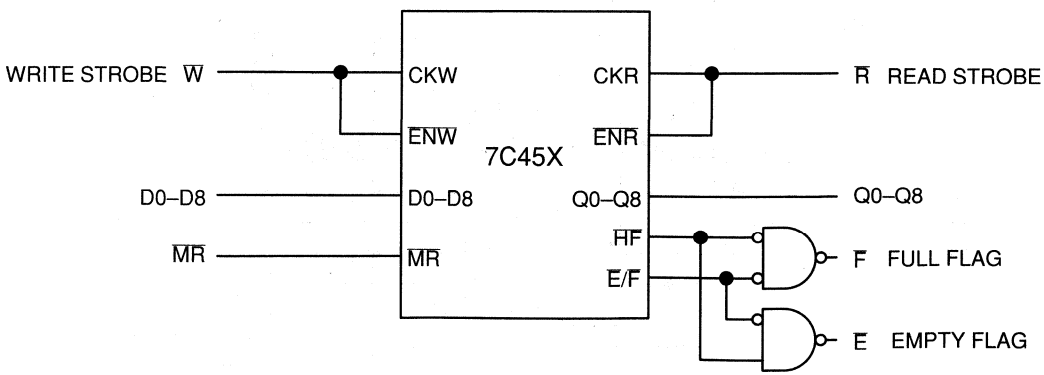


Figure 8. Using a Clocked FIFO Like a Standard FIFO

edge of CKR. Consequently, the read port must be strobed in order to force the flag to be updated. While the empty flag is asserted, the attempted reads are ignored (data remains in the FIFO) and only serve to update the empty flag. Once the empty flag is deasserted, the data can be read from the FIFO in the normal manner.

It also possible to build a controller that forces an update cycle at the FIFO boundary without checking the state of the flags. When a read or a write strobe occurs affecting the state of the memory array, the controller forces an update automatically by toggle the other strobe line.

Conclusion

Cypress 7C44X and 7C45X Clocked FIFOs solve a wide variety of data buffering and storage needs for telecommunications, interprocessor, and data gathering applications. The clocked FIFO architecture offers 70-MHz performance and avoids the timing and noise problems inherent in unclocked asynchronous FIFOs.

Document #: LOG0003–0293

FIFO Dipstick Using *Warp2* ™ VHDL and the CY7C371

Introduction

Programmable FIFO flags can often simplify the design of a digital system by automatically indicating a status that can prevent overrun or underrun in an elastic FIFO buffer. Although many FIFOs are available with on-chip programmable flag functions, these features are not available on industry-standard asynchronous FIFOs. Of those FIFOs that do have programmable flags, some do not allow the almost-empty and almost-full values to be programmed independently, or in some cases, for these values to be programmed to any specific word boundary. This application note presents a method by which FIFOs of any size may be monitored by an external Programmable Logic Device which will then generate all of the flags necessary for most FIFO applications. The FIFO Dipstick PLD behaves like a measuring device that can observe the level of data within a FIFO.

Application Description

A variable-length up-down counter is implemented with VHDL to measure the exact level of data within a FIFO. The number of bits required for the dipstick counter is dependent on the size of the FIFO and must satisfy the following equation:

2^n = FIFO Depth; Where: n = number of counter bits required

For example, a 2K FIFO would require an 11-bit counter. The n^{th} bit is necessary to prevent the dipstick counter from rolling over to zero when the last byte is written into the FIFO. In other words, the n^{th} bit will only be set when the FIFO is completely full.

Due to the truly asynchronous nature of the read and write ports of a FIFO, a state machine must be implemented to control the operation of the dipstick counter. This state machine must resolve the overlapping and nesting conditions that may occur with the FIFO_READ_L and FIFO_WRITE_L signals to the FIFO. For instance, multiple read pulses may occur within a single write pulse, read and write pulses may occur simultaneously, or read and write pulses may overlap by any amount of time.

The status of the almost-full and almost-empty flags is determined by simply comparing the dipstick counter value to pre-programmed levels and generating the appropriate combinatorial outputs. This method allows for the generation of any flag outputs required for a given application. The almost-full and almost-empty flags are the most typical levels required and are used to determine greater-than-or-equal-to and less-than-or-equal-to specified levels, respectively. Many possibilities exist, however, such as an approx-half-full flag, which could be used to add hysteresis to the half-full value of a FIFO.

Synchronous FIFO Ports

The VHDL/FLASH370™ implementation in this application note is based upon the following assumption. Both the read and write ports of the FIFO are controlled by clocked circuitry and the clocks for each port are synchronous to each other. This assumption allows a single clock to be used for the state machine and the counter. It also provides for the read, write, and reset inputs to be used without any chance of a metastable event occurring. As a result of this synchronous implementation, the almost-flags will change state combinatorially within

three clock cycles after the clock cycle that initiated the read or write. For instance, if a FIFO read is held active for two clock cycles followed by one clock cycle for read-recovery time, the updated almost-empty flag will be available during the read-recovery cycle.

Asynchronous FIFO Ports

The read and write ports of a FIFO may be controlled by clocked circuitry with clocks that are asynchronous to each other. In this case, the state machine and counter should be controlled by the one clock that best suits the application. If it is imperative that the write port of the FIFO receives the almost-full flag immediately, the write port clock should be used. If it is imperative that the read port of the FIFO receives the almost-empty flag immediately, the read port clock should be used. In either case, the read or write input from the opposite port needs to be synchronized to the dipstick's clock before it is used as a state machine input. The CY7C371 is ideally suited for this because of its dedicated inputs, which can be configured as single- or double-registered which will achieve a guaranteed 10-year MTBF. In addition, the port that is asynchronous to the dipstick's clock must also synchronize the almost-flags before use to prevent metastability problems. A negative aspect of using the FIFO dipstick in this application is that additional delays are introduced between a FIFO access and the almost-flags status change. These additional delays may or may not be tolerable, depending on the application.

State Machine Design

The finite state machine observes the FIFO_READ_L and FIFO_WRITE_L inputs in order to control the operation of the dipstick counter (see *Figure 1*). There are eight states required: an idle state, four counter-enabled states, and three counter-disabled states. The counter-enabled states are further categorized into count-up states (write and rd_hold_wr) and count-down states (read and wr_hold_rd). The counter-disabled states (rd_hold, wr_hold, rd_hold_wr_hold) are required for the FIFO_READ_L and FIFO_WRITE_L pulses that are active for greater than one clock cycle.

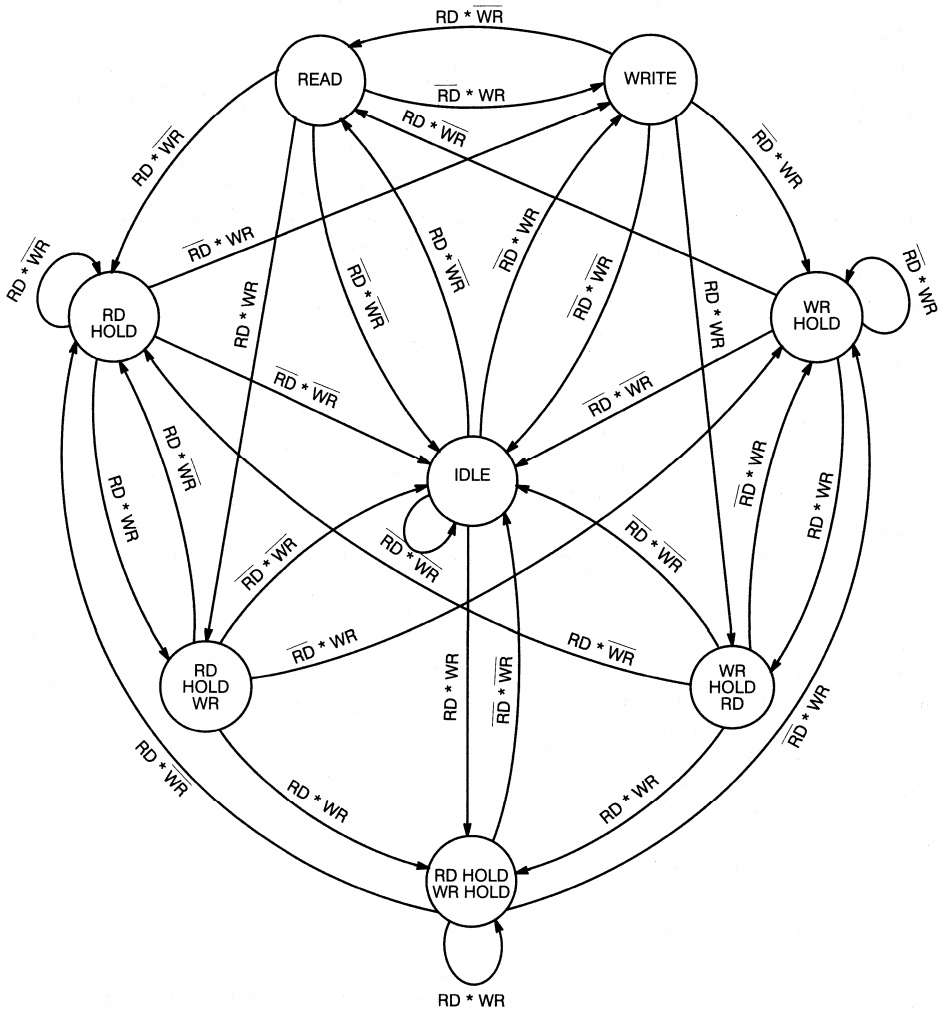
Within each state, all four permutations of FIFO_READ_L and FIFO_WRITE_L are evaluated to determine the next state. If neither signal is active, the state machine always returns to the idle state. If a single signal goes active or stays active, the state machine will progress to the appropriate state such that the counter-enabled states are active for a single clock cycle only, during each FIFO_READ_L and FIFO_WRITE_L pulse to the FIFO. If both FIFO_READ_L and FIFO_WRITE_L are observed going active on the same clock cycle, the counter-enabled states are avoided completely, allowing the dipstick counter to remain constant. The FIFO_RESET_L signal is not required as an input to the state machine because the dipstick counter will remain cleared if FIFO_RESET_L is active.

Warp2 VHDL Implementation

The VHDL design used for the FIFO Dipstick is completely behavioral. This high-level design methodology eliminates any need to describe device specific implementations and it also allows for the most readability. The *Warp2* VHDL Compiler will synthesize the design into low-level components necessary for a CY7C371 automatically.

The design entity defines all the inputs and outputs of the design and assigns a type to these signals. The architecture describes the behavior of the circuit. See Appendix A for a listing of the code.

The entity declaration is comprised of a port map, a generic statement, and an attribute statement. The port map defines the FIFO Dipstick inputs, outputs, and the bidirectional counter bits for a variable-length up-down counter. The FIFO_READ_L, FIFO_WRITE_L, and FIFO_RESET_L inputs are written with a _l suffix to indicate that they are active LOW signals, all others are active HIGH. The generic statement is used as a convenient way to define the actual size of the dipstick counter. Simply defining the counter size once in this statement allows the entire design to be modified accordingly, including the number of bidirectional pins defined for the dipstick counter. The attribute statement has been included to define the CY7C371 as the PLD for which


READ
WRITE
RD HOLD
WR HOLD
IDLE
RD HOLD WR
WR HOLD RD
RD HOLD WR HOLD
RD * WR
RD * WR
RD * WR

The process titled *counter* controls the operation of the FIFO dipstick counter. If the FIFO_RESET_L input is asserted, then the counter is cleared on the next clock edge. This is accomplished by using a *Warp2* function-call titled i2bv, which converts the integer constant "zero_count" to a bit_vector of the appropriate length. The result is then assigned as the new counter value which is a bit_vector of all zeros. If the FIFO_RESET_L input is not active, the counter operation is then determined by the state of the dipstick state machine and the current counter value. The count-up states will increment the counter unless the counter's MSB is set indicating that the maximum count has been reached. The count-down states will decrement the counter unless it is currently equal to zero. If none of the conditions described above exist, then the counter will maintain its current value as indicated in the else statement. The inc_bv and dec_bv functions, used to increment and decrement the counter bit_vector respectively, are provided by *Warp2* as part of the bv_math package in the work library.

The process titled *state_machine* implements the finite state machine, as represented in the bubble chart of *Figure 1*. This behavioral description makes use of the enumerated-type form. The major advantage of this form is that the state encoding can be easily changed by the user. The current encoding options available are sequential, gray, one-hot, and user-defined and are determined by the attribute state_encoding. The enumerated type is defined at the beginning of the architecture body and is comprised of the eight state names. The case statement within the process defines the state machine transitions based on the current state and the FIFO_READ_L and FIFO_WRITE_L inputs.

Differences From Programmable FIFOs

The following two differences between the FIFO Dipstick design and the use of a FIFO with programmable flags must be understood. First, the latency incurred between a FIFO access and the update of flag status may be prohibitive; refer to the synchronous and asynchronous FIFO ports sections above. Second, the flag outputs of a FIFO will always go inactive based on a FIFO strobe going inactive, whereas the FIFO Dipstick solution will always change flag states based on the strobes going active.

Summary

This application note provides the information required to implement programmable flags for any size FIFO by simply changing the values in the VHDL statements of Appendix A, which are noted as application specific in the source code. For applications that require dynamically alterable flags, a microprocessor port is easily adaptable. The design in Appendix A is also easily adaptable to different FIFO applications, i.e., clocked FIFOs, BiFIFOs, FIFOs with asynchronously clocked ports, etc.

Appendix A. FIFO Dipstick *Warp2* VHDL Source Code

```
USE work.bv_math.all;
USE work.int_math.all;

ENTITY dipstick IS
GENERIC (counter_size: INTEGER := 16);-- APPLICATION SPECIFIC
   PORT (clock, fifo_reset_l, fifo_rd_l, fifo_wr_l: IN BIT;
      afull, aempty: OUT BIT := '0';
      counter: INOUT BIT_VECTOR(counter_size DOWNTO 0));
   ATTRIBUTE part_name OF dipstick: ENTITY IS "C371";
END dipstick;

ARCHITECTURE behavior OF dipstick IS
   CONSTANT afull_value: INTEGER := 32000;-- APPLICATION SPECIFIC
   CONSTANT aempty_value: INTEGER := 07;-- APPLICATION SPECIFIC

   TYPE fifostate IS
      (idle, read, rd_hold, rd_hold_wr, rd_hold_wr_hold,
       write, wr_hold, wr_hold_rd);
   SIGNAL nextstate: fifostate;
   ATTRIBUTE state_encoding OF fifostate: TYPE IS SEQUENTIAL;

BEGIN
   outputs: PROCESS BEGIN
      IF (counter >= afull_value) THEN
         afull <= '1';
         aempty <= '0';
      ELSIF (counter <= aempty_value) THEN
         afull <= '0';
         aempty <= '1';
      ELSE
         afull <= '0';
         aempty <= '0';
      END IF;
   END PROCESS;

   counter: PROCESS
      CONSTANT zero_count: INTEGER := 0;
   BEGIN
   WAIT UNTIL (clock = '1');
      IF (fifo_reset_l = '0') THEN
                counter <= I2BV(zero_count, counter_size);
      ELSIF ((nextstate = write) OR (nextstate = rd_hold_wr)) AND
            (counter(counter_size-1) = '0') THEN
                counter <= inc_bv(counter);
      ELSIF ((nextstate = read) OR (nextstate = wr_hold_rd)) AND
            (counter /= zero_count) THEN
                counter <= dec_bv(counter);
      ELSE      counter <= counter;
      END IF;
   END PROCESS;
```

Appendix A. FIFO Dipstick *Warp2* VHDL Source Code (continued)

```
state_machine: PROCESS
BEGIN
WAIT UNTIL (clock = '1');

CASE nextstate IS
   WHEN idle =>
      IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
         nextstate <= idle;
      ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
         nextstate <= read;
      ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= write;
      ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= rd_hold_wr_hold;
      END IF;
   WHEN read =>
      IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
         nextstate <= idle;
      ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
         nextstate <= rd_hold;
      ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= write;
      ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= rd_hold_wr;
      END IF;
   WHEN rd_hold =>
      IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
         nextstate <= idle;
      ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
         nextstate <= rd_hold;
      ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= write;
      ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= rd_hold_wr;
      END IF;
   WHEN rd_hold_wr =>
      IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
         nextstate <= idle;
      ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
         nextstate <= rd_hold;
      ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= wr_hold;
      ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
         nextstate <= rd_hold_wr_hold;
      END IF;
   WHEN rd_hold_wr_hold =>
      IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
         nextstate <= idle;
```

Appendix A. FIFO Dipstick *Warp2* VHDL Source Code (continued)

```
        ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
           nextstate <= rd_hold;
        ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
           nextstate <= wr_hold;
        ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
           nextstate <= rd_hold_wr_hold;
        END IF;
      WHEN write =>
        IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
           nextstate <= idle;
        ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
           nextstate <= read;
        ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
           nextstate <= wr_hold;
        ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
           nextstate <= wr_hold_rd;
        END IF;
      WHEN wr_hold =>
        IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
           nextstate <= idle;
        ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
           nextstate <= read;
        ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
           nextstate <= wr_hold;
        ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
           nextstate <= wr_hold_rd;
        END IF;
      WHEN wr_hold_rd =>
        IF ((fifo_rd_l AND fifo_wr_l) = '1') THEN
           nextstate <= idle;
        ELSIF (((NOT fifo_rd_l) AND fifo_wr_l) = '1') THEN
           nextstate <= rd_hold;
        ELSIF ((fifo_rd_l AND (NOT fifo_wr_l)) = '1') THEN
           nextstate <= wr_hold;
        ELSIF (((NOT fifo_rd_l) AND (NOT fifo_wr_l)) = '1') THEN
           nextstate <= rd_hold_wr_hold;
        END IF;
      WHEN OTHERS =>
          nextstate <= idle;
   END CASE;
   END PROCESS;
END behavior;
```

Document #: LOG0004–0294

Warp2 and FLASH370 are trademarks of Cypress Semiconductor Corporation.

CY7C42X/46X FIFO Interface to the CY7B923 (HOTLink™)

Transmitter Interface Description

This application note considers the interface between a Cypress CY7B923 (HOTLink™) Transmitter and generic FIFOs. Minimal interface logic is required to achieve a high-performance interface. A block diagram of the HOTLink Transmitter and generic FIFO interface is shown in *Figure 1.*

The FIFO operates as an asynchronous data rate buffer between the HOTLink Transmitter and the data source. The data is continually read from the FIFO into the transmitter when the Transmit signal is asserted. Reading continues until the FIFO is empty.

Critical Timing Analysis

The following equations describe the critical timing relationships. They have been solved for the minimum bit time t_B. The clock period time is $10t_B$. A timing diagram is provided in *Figure 2.* The critical timing equations are shown at the bottom of the diagram.

Read Pulse Width

$$t_{PR(min)} \leq 6t_B - 3\text{ ns} - 2\text{ns} \qquad \text{Eq. 1}$$
$$t_B \geq (t_{PR(min)} + 5\text{ ns}) / 6$$

The read pulse width for the FIFO is t_{PR}.

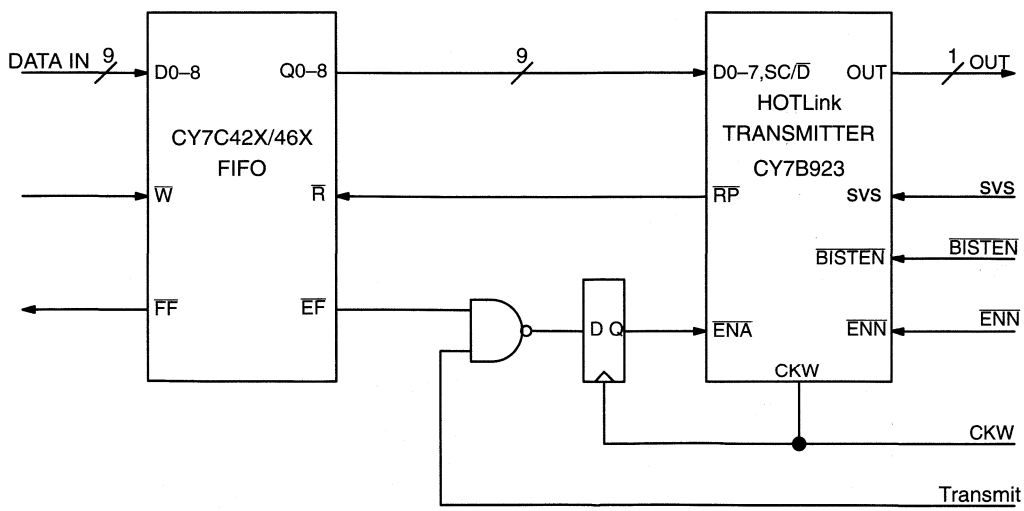
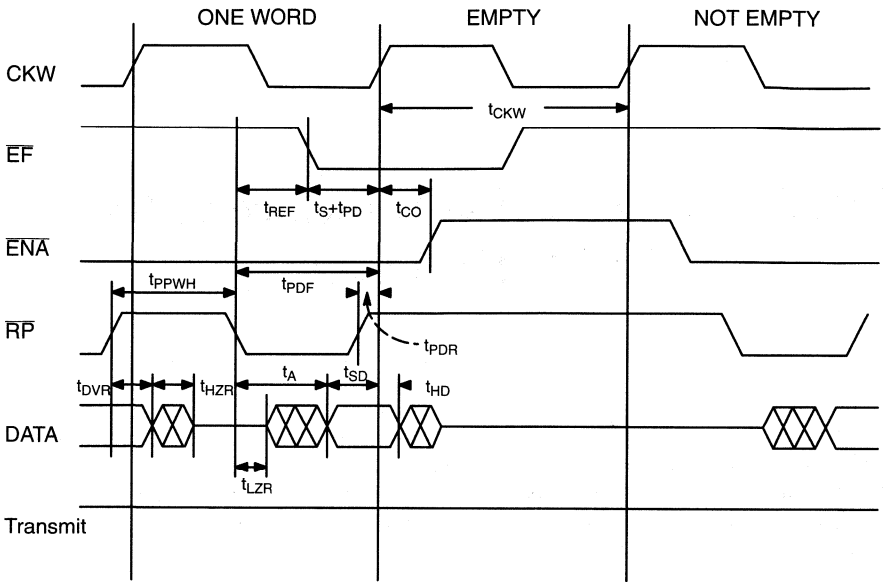


Figure 1. Transmitter Interface Diagram



Critical Timing Analysis

1. Read pulse width:
$$t_{PR(min.)} \leq t_{PDF(min.)} - t_{PDR(max.)}$$
2. Read recovery time:
$$t_{RR(min.)} \leq t_{PPWH(max.)}$$
3. Data set-up time:
$$t_{A(max.)} + t_{SD(min.)} \leq t_{PDF(min.)}$$
4. Empty flag to register set-up time:
$$t_{REF(max.)} + t_{PD(max.)} + t_{S(min.)} \leq t_{PDF(min.)}$$
5. Transmit enable to HOTLink set-up time:
$$t_{CO(max.)} \leq 10\, t_B - t_{SENP(min.)}$$
6. Data hold time:
$$t_{PDR(max.)} + t_{HD(max.)} \leq t_{DVR(min.)}$$

Figure 2. Interface Timing Diagram

Read Recovery Time

$$t_{RR(min.)} \leq 4t_B - 3\text{ ns} \qquad \text{Eq. 2}$$
$$t_B \geq (t_{RR(min.)} + 3\text{ ns}) / 4$$

The read recovery time for the FIFO is t_{RR}.

Data Set-Up Time

$$t_{A(max.)} + 5\text{ ns} \leq 6t_B - 3\text{ ns} \qquad \text{Eq. 3}$$
$$t_B \geq (t_{A(max.)} + 8\text{ ns}) / 6$$

The data access time for the FIFO is t_A and it is the basis of FIFO speed ratings.

Empty Flag to Register Set-Up Time

$$t_{REF(max.)} + t_{PD(max.)} + t_{S(min.)} \leq 6t_B - 3\ n \qquad \text{Eq. 4}$$
$$t_B \geq (t_{REF(max.)} + t_{PD(max.)} + t_{S(min.)} + 3\ ns) / 6$$

The Empty flag delay from the FIFO is t_{REF}. The register set-up time for the external register is t_S.

Transmit Enable to HOTLink Set-Up Time

$$t_{CO(max.)} + \leq 4t_B - 8\ ns \qquad \text{Eq. 5}$$
$$t_B \geq (t_{CO(max.)} + 8\ ns) / 4$$

The register clock to output delay is t_{CO}. The propagation delay of the external control logic is t_{PD}.

Data Hold Time

$$t_{DVR(min.)} \geq 2\ ns \qquad \text{Eq. 6}$$

The valid data hold time from a FIFO read is t_{DVR}. HOTLink has a zero data hold time.

Table 1. Critical FIFO Timing Parameters

Parameter	FIFO Speed Rating		
	−10	−15	−20
$t_{PR(min)}$	10 ns	15 ns	20 ns
$t_{RR(min)}$	10 ns	10 ns	10 ns
$t_{A(max)}$	10 ns	15 ns	20 ns
$t_{REF(max)}$	10 ns	15 ns	20 ns
$t_{DVR(min)}$	3 ns	3 ns	3 ns

Table 1 shows the critical timing parameters for various speed grades of generic FIFOs. The FIFO timing parameters are taken from a hypothetical CY7C42X−10, a CY7C46X−15, and a CY7C46X−20.

Table 2 shows the maximum frequency of CKW associated with each of the timing equations for the different speed grades of generic FIFOs. The maximum interface operating frequency is shown in italics. A PAL20−5 (t_{PD} = 5 ns, t_{CO} = 5 ns, t_S = 2.5 ns) is used for the flag register and enable control logic.

Equation 6 is independent of the clock frequency and is satisfied by all of the considered FIFOs.

Equation 4 is the critical timing relationship for all of the FIFO speed grades. Timing margins can be increased by using faster control logic (PAL20−4).

Table 2. Maximum Transmitter Interface Frequency with Asynchronous FIFOs

Eqn. #	−10	−15	−20	Units
1	40.0	30.0	24.0	MHz
2	30.7	30.7	30.7	MHz
3	33.3	26.1	21.4	MHz
4	29.2	23.5	19.7	MHz
5	30.8	30.8	30.8	MHz
bit rate	292	235	197	Mbits/s

Summary

With available CY7C46X−15 FIFOs, the HOTLink-FIFO interface can operate at a frequency of 23.5 MHz with minimal interface logic. This corresponds to a serial bit rate of 235 Mbits/s.

When −10 FIFOs become available, the maximum interface frequency will increase to 29.2 MHz (292 Mbits/s).

Document #: LOG0005−0293

HOTLink is a trademark of Cypress Semiconductor Corporation.

Interfacing the CY7B923 and CY7B933 (HOTLink™) to a Clocked FIFO

Introduction

This application note considers the interfacing issues between the Cypress CY7B923/CY7B933 (HOTLink™) Transmitter/Receiver and Cypress clocked FIFOs. The HOTLink-FIFO interface is capable of performing parallel bus transactions at rates of up to 70 Mbytes/s and serial transfers at rates of up to 330 Mbits/s. The FIFO serves as an asynchronous storage buffer between the data bus and the serial link.

Transmitter Interface

This section describes the design considerations of a high-speed serial transmitter with FIFO (First In First Out) data buffers. The interface design supports basic data transmission control and serial link testing. The transmitter design is intended to interface to a higher-level system controller responsible for handling bus transactions and the serial link protocol. The interface is a primitive building block that is easily modified to meet system requirements.

Data Path and Controller

The transmitter interface consists of a single CY7C441/3−14 clocked FIFO interfacing directly to the HOTLink Transmitter. A transmitter controller supplies the control signals to both the FIFO and HOTLink. The architecture of the controller is left unspecified, but it can be implemented in a PLD or an FPGA. State diagrams and generic timing diagrams are provided. A block diagram of the transmitter interface is shown in *Figure 1*.

Built-In Self-Test

The transmitter is capable of checking the functionality of the transmitter serial connection by exercising the Built-In Self-Test (BIST) mode of HOTLink. To initiate BIST, the $\overline{\text{BISTEN}}$ pin is held LOW, resulting in the transmission of the sequence ...1 0 1 0.... The HOTLink $\overline{\text{ENA}}$ (Enable Parallel Data) pin is then pulled LOW to enable transmission of the BIST test pattern. HOTLink will assert the $\overline{\text{R}}_{\text{P}}$ (Read Pulse) pin HIGH at the beginning of BIST and will pulse it LOW once per BIST loop. During BIST, HOTLink ignores data at its parallel port and the FIFO does not perform any reads.

Resetting the FIFO

The higher-level controller should reset or clear the FIFO at power-up, before a new block of data is transmitted, or if an error occurs. Resetting the FIFO is accomplished by asserting the $\overline{\text{MR}}$ (Master Reset) pin on the FIFO LOW. Neither a read nor a write can occur on the cycles immediately preceding, during, or following the assertion of $\overline{\text{MR}}$. To insure that this condition is met, the interface controller must be in the IDLE state (*Figure 2*). Proper FIFO reset also requires that $\overline{\text{MR}}$ be glitch free. The higher-level controller is responsible for coordinating the read and write ports and insuring that the reset conditions are met.

Controller State Description

For applications requiring high-speed asynchronous data buffering, the FIFO read and write ports should be controlled by separate control circuitry synchronized to the FIFO ports. The FIFO write

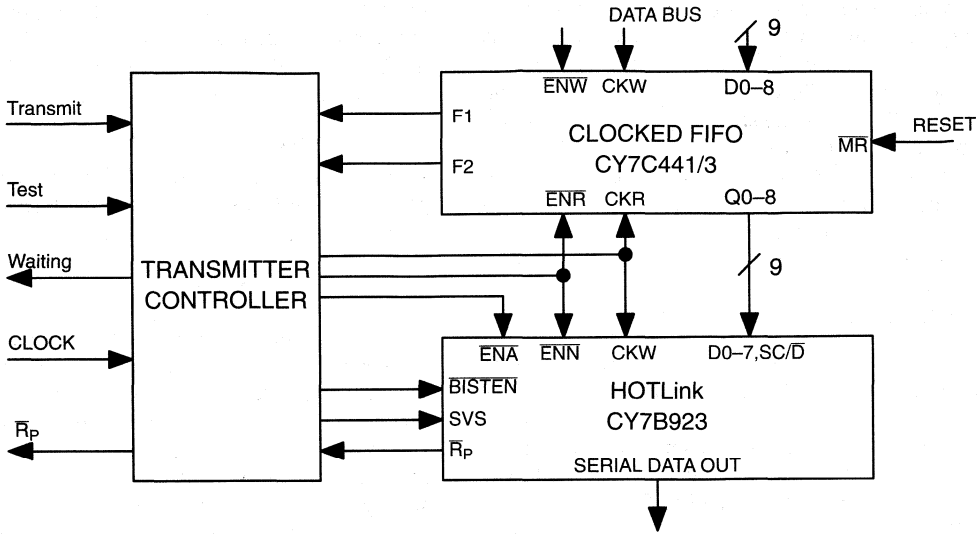


Figure 1. Transmitter Interface Block Diagram

port interfaces directly to a 9 bit data bus. Data is written into the FIFO by asserting $\overline{ENW}$ to enable the write clock (CKW). Data may be written at any time as long as the FIFO is not full as indicated by the FIFO full flag.

The FIFO read port interfaces to the HOTLink transmitter serial port. Control of this interface is the focus of this section. The transmitter interface state machine controls FIFO-HOTLink data transactions and initiates the HOTLink Built-In Self-Test. The interface state machine is under the control of a higher-level controller responsible for both the serial protocol and the data bus–FIFO transactions.

The interface controller is a simple state machine as shown in *Figure 2*. While the state machine waits in the IDLE state, HOTLink will transmit Idle fill characters (K28.5). When the Transmit signal is asserted by the higher-level controller, the transmitter state machine transitions to the TX state. The TX

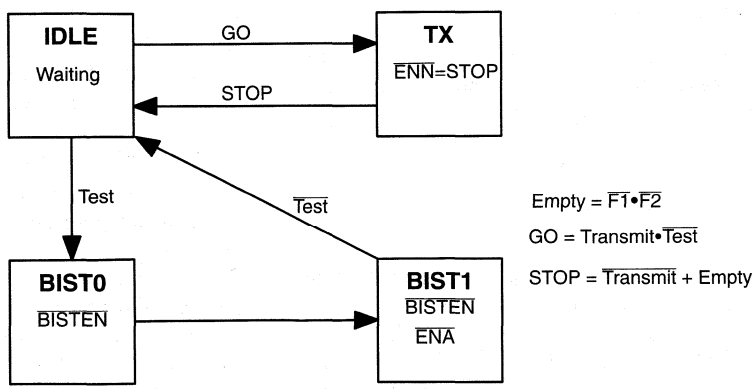


Figure 2. Transmitter Controller State Diagram

state reads 9 bit words out of the FIFO into HOTLink until a Stop condition is detected (the FIFO is empty or the Transmit signal is deasserted). Reading data from the FIFO is accomplished by asserting $\overline{ENR}$ LOW. The same signal is connected to $\overline{ENN}$ (Enable Next Parallel Data) pin of HOTLink. Assertion of $\overline{ENN}$ causes data on the next rising edge of the clock to be transmitted. $\overline{ENN}$ functionality is specifically designed to operate with the pipelined architecture of clocked FIFOs. After a Stop condition is detected, the state machine returns to the IDLE state and asserts the Waiting signal.

The state diagram includes test states for exercising the Built-In Self-Test (BIST) capabilities of HOTLink. The Built-In Self-Test loop is entered when the higher-level controller asserts the Test signal while the transmitter state machine is in the IDLE state. The BIST0 state asserts $\overline{BISTEN}$ to initiate to the transmission of the ...1 0 1 0... sequence. The BIST1 state then asserts $\overline{ENA}$ to start the BIST pattern generation. The higher-level controller should monitor $\overline{R}_P$ in order count the number of BIST patters sent. Built-In Self-Test will conclude when the higher-level controller deasserts Test after the desired number of BIST patterns have been sent. Control then returns back to the IDLE state.

Critical Timing Analysis

Timing diagrams are provided for the transmitter interface. The analysis assumes that the state machine state state bits are accessible sooner than any data or input control signal.

FIFO-HOTLink Data timing is governed by the FIFO access time (t_A= 10 ns) and the data set-up time for HOTLink (t_{SD} = 5 ns).

$$t_A + t_{SD} \leq t_{CKW}$$

With clock periods greater than 30 ns, the data has no trouble meeting these timing constraints.

The critical timing path of the FIFO-HOTLink interface is due to the delay associated with decoding the flags and generating the enable for the clocked FIFO ($\overline{ENR}$) and HOTLink ($\overline{ENN}$). Note that these are the same signals, but $\overline{ENR}$ requires a longer set-up time than $\overline{ENN}$. The delay due to the state machine decoding the flags and generating the enable is represented as t_{PD}. The FIFO flag delay, t_{FD}, is 10 ns. The read enable set-up time for the FIFO, t_{SEN}, is 7 ns.

$$t_{PD} \leq t_{CKW} - t_{SEN} - t_{FD}$$

A 30-ns clock period leaves the controller 13 ns to generate the $\overline{ENN}$ signal. A timing diagram is provided in *Figure 3*.

Receiver Interface

The receiver interface uses a single CY7C451/3−14 clocked FIFO to buffer the incoming HOTLink data stream. The 45X features programmable flags and three-state output drivers for bus applications. The HOTLink receiver interface is capable of receiving serial data at rates of up to 330 Mbits/s and then storing 9 bit words in the FIFO. Words in the FIFO can be read to the data bus at rates of up to 70 MBytes/s. A higher-level controller is responsible for coordinating the receiver interface and bus transactions according to the serial link protocol. *Figure 4* shows a block diagram of the HOTLink-FIFO interface.

Reframe

The HOTLink serial receiver must synchronize itself with the proper word alignment of the incoming data. Assertion of the HOTLink RF (Reframe) input forces HOTLink to synchronize its internal bit counter with the boundary of a received K28.5 character. HOTLink will respond by asserting $\overline{RDY}$ LOW when the first K28.5 is received. The receiver state machine controller should be instructed to synchronize HOTLink at the beginning of data reception or after excessive errors have been received.

Data Path and Controller

The receiver state machine responds to control signals from a higher-level controller. The higher-level controller initiates data reception by asserting the Receive signal to the receiver state machine. Nine bit words from the HOTLink parallel port are stored into the 7C45X FIFO each time $\overline{RDY}$ is asserted LOW. $\overline{RDY}$ will pulse LOW when new data is available at the HOTLink parallel port and will be HIGH when a pad character is received (K28.5 fol-

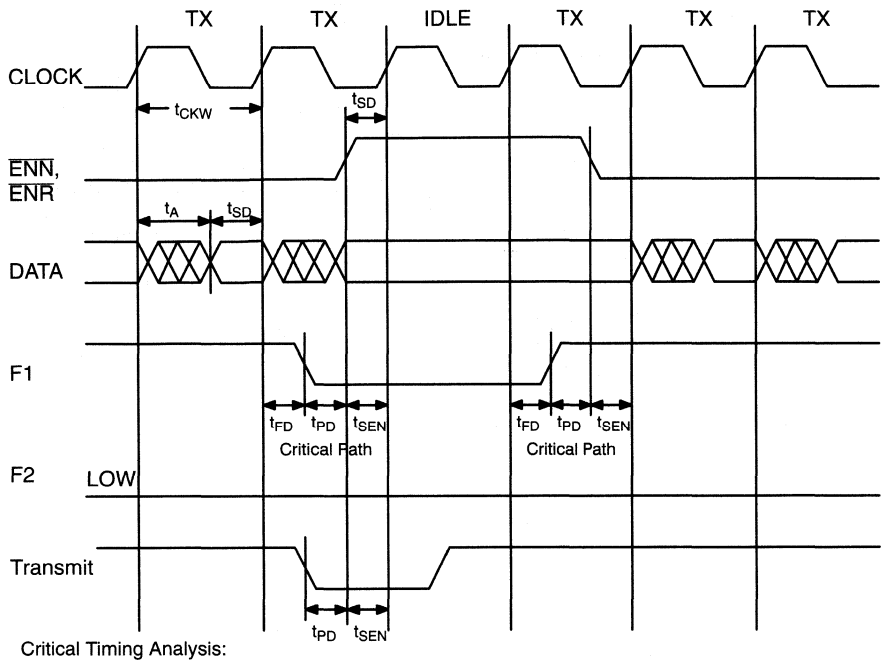


Figure 3. Transmitter Timing Diagram

lowed by K28.5). $\overline{RDY}$ is used to prevent the FIFO from filling with pad characters. Data storage will stop immediately when Receive is deasserted.

If the FIFO becomes full, it will ignore attempted writes. Full and Empty flags are decoded so that the higher-level controller can detect when the FIFO contains data or is completely full.

The 7C45X features programmable Almost Full and Almost Empty flags. The distance that these flags become active from the Empty and Full FIFO boundary is programmed during the FIFO Master Reset cycle. The distance can be set such that a flag is asserted when a fixed length packet of data has been received. The higher level controller responds to the flag by reading the data packet out of the FIFO. The Almost Full flag is useful for preventing data from being lost. This flag can be programmed to compensate for the response latency of the higher level controller so that data can be read from the FIFO before it becomes full resulting in lost data. The decoding of the programmable flag signals is left out of the controller design for clarity.

Optional Pipeline Register

The optional pipeline register increases interface speed by capturing the $\overline{RDY}$ pulse and easing the control signal timing margins. $\overline{RDY}$ is a delayed 60% LOW duty cycle signal shaped for asynchronous FIFOs. Without the pipeline register, the LOW phase of $\overline{RDY}$ leaves less than $\frac{1}{2}t_{CKR} - 10$ ns

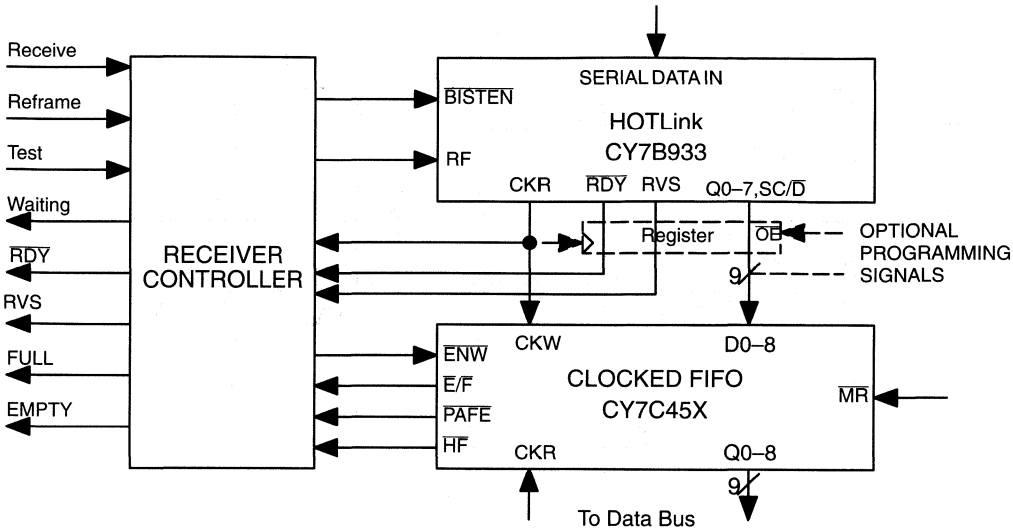


Figure 4. Receiver Interface Block Diagram

to generate the FIFO write enable and meet the set-up time. A clock period of 40 ns (250 Mbit/s) leaves a manageable 10 ns for the receiver state machine to generate the FIFO write enable, but as the clock period decreases to 30 ns (330 Mbit/s), the enable generation time shrinks to only 5 ns. This timing difficulty is overcome by pipelining the interface. The data and status signals must be pipelined to insure the proper word is written into the FIFO. The timing implications are considered in the section on critical timing analysis.

A data pipeline register with three-state output drivers can also be used to isolate the HOTLink parallel port from the FIFO write port while programming the 45X FIFO flags. A 9-bit program word from an external source can be written into the FIFO during a Master Reset Cycle. The program word sets the Almost Empty and Almost Full flags and sets the FIFO parity option.

Resetting and Programming the FIFO

The higher-level controller should perform a FIFO Master Reset cycle after power-up, before new data is received, if an error occurs, or in order to program the FIFO flags. A Master Reset cycle is accomplished by asserting the $\overline{MR}$ pin on the FIFO LOW. Proper resetting or programming requires that $\overline{MR}$ be glitch free. In addition, neither a read nor a write can occur on the cycles immediately preceding, during, or following the assertion of $\overline{MR}$ unless the FIFO is being programmed. If FIFO is not being programmed, the receiver state machine should remain in the WAIT state during the Master Reset cycle.

In order to program the FIFO, the higher level controller should put the data pipeline register in the high impedance state. The program word is then supplied to the FIFO by an external source (data bus, controller, etc.). The word is written into the FIFO internal program register during the Master Reset cycle on the rising edge of the clock that is enabled by $\overline{ENW}$ asserted LOW.

Built-In Self-Test

The Built-In Self-Test mode is exercised by asserting the $\overline{BISTEN}$ pin on HOTLink. Upon entering BIST, HOTLink will wait for the BIST initialization code and then assert $\overline{RDY}$ LOW when the code has been received. $\overline{RDY}$ will pulse HIGH once per received BIST loop. RVS will pulse HIGH if a byte pattern mismatch occurs. $\overline{RDY}$ and RVS can be monitored

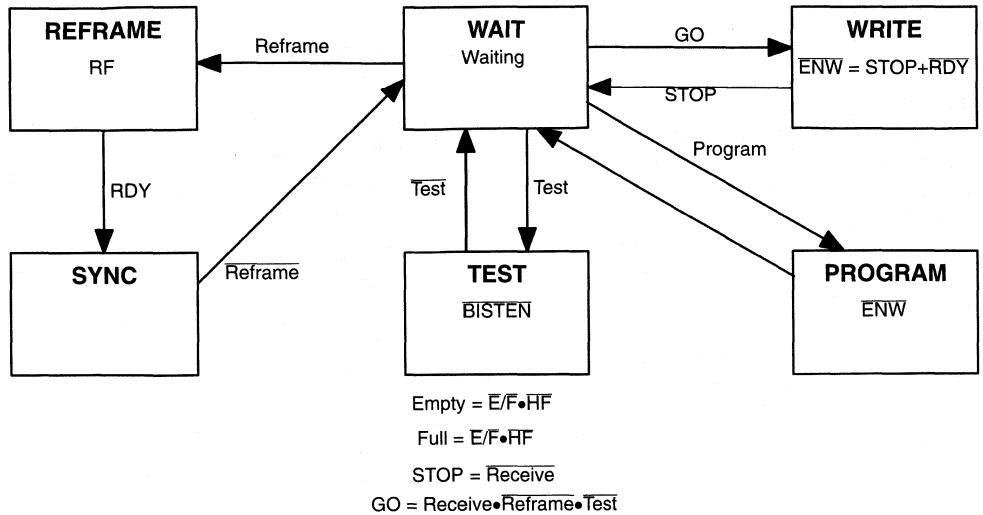


Figure 5. Receiver Controller State Diagram

by the higher-level controller to characterize the integrity of the link.

Controller State Description

A state diagram for a receiver state machine is shown in *Figure 5*. Five simple signals control the interface. The Receive signal instructs the state machine to store words into the FIFO when RDY pulses LOW. Deassertion of Receive ends data reception abruptly. The Reframe signal tells the state machine synchronize HOTLink to the serial data. The Test signal forces HOTLink to perform BIST and the Program signal causes the state machine to write a word into the FIFO internal program register. The Waiting output signal is asserted when the state machine is in the WAIT state.

Full and Empty signals are decoded for the convenience of the higher-level controller to assist in reading data out of the FIFO. The programmable flags may also be decoded if they have been programmed. It is important that the flags be monitored because a full FIFO will ignore attempted writes. The higher-level controller is responsible for insuring that the FIFO does not become full.

The REFRAME state is entered by the assertion of Reframe from the WAIT state. The REFRAME state is used to synchronize the receiver to the incoming data stream. When the state machine asserts RF, the HOTLink receiver synchronizes its internal bit counter with received K28.5 characters. RDY will pulse LOW when the first synchronized K28.5 character is available. The state machine will return to the WAIT state when the serial data has been resynchronized and Reframe is deasserted.

Data reception is initiated by asserting the Receive signal while the state machine is in the WAIT state. The controller will immediately transition to the WRITE state and store data when RDY is asserted LOW. The WRITE state continually writes valid characters into the FIFO until Receive is deasserted. Control then returns to the WAIT state and Waiting is asserted.

The BIST state is included for handling the Built-In Self-Test. During BIST, writing to the FIFO is disabled. Assertion of RVS will signal a character reception error. RDY will pulse once per BIST loop

and should be used to count the number of BIST loops received. The higher-level controller should monitor these signals in order to characterize the link.

The PROGRAM state writes the program word into the FIFO internal program register. This state is entered from the WAIT state at the command of the higher-level controller. Programming should only be performed during a Master Reset cycle ($\overline{MR}$ LOW). In order to meet the FIFO programming timing requirements, it is recommended that at least one clock cycle occur on each side of the program cycle while $\overline{MR}$ is LOW. The higher-level controller is responsible for meeting the specific programming timing requirements discussed in the Resetting and Programming the FIFO section of the CY7C45X datasheet.

Critical Timing Analysis

Timing analysis for both the pipelined and unpipelined interface are presented in this section. A Timing diagram is provided for the receiver interface that does not include the optional register. Critical timing relationships are provided at the bottom of *Figure 6*. This diagram highlights the critical timing of the $\overline{RDY}$ pulse. The interface timing with pipeline registers is straight forward and the results are presented below.

The timing analysis assumes that the state machine state bits are stable and valid before any critical signal is available to the state machine and that state bit set-up time is not an issue. This assumption allows the state machine timing to modeled by its combinatorial t_{PD}.

Unregistered Timing

The delayed $\overline{RDY}$ pulse tightens the timing margins on the receiver controller. The state machine combinatorial delay for generating output control signals from valid inputs is modeled as t_{PD}. The FIFO enable set-up time is t_{SEN}=7 ns. Assuming t_{CKR} is 30 ns, the constraint on t_{PD} is

Write enable generation time from $\overline{RDY}$ LOW:

$t_{PD} \leq 1/2\ t_{CKR} - t_{SEN} - 3\ ns = 5\ ns.$

A 40 ns clock period eases the timing constraint to a more reasonable 10 ns.

The parallel data have no problem meeting the timing constraints imposed by a 30-ns clock period. The HOTLink access time, t_A, is 9 ns and the FIFO data set-up time, t_{SD}, is 7 ns:

Critical data timing:

$t_A + t_{SD} \leq t_{CKR}$

This assumes no trace delays or clock skew.

Registered Timing

With the optional pipeline register inserted, the timing constraint on the controller is eased. A register access time, t_{AR}, of 10 ns and set-up time, t_{SU}, of 5 ns are assumed. Using a 30-ns clock, the HOTLink access time is $t_A = t_{CKR}/5 + 3\ ns = 9\ ns$.

The constraint on the combinatorial delay through the controller is

Write enable generation time from $\overline{RDY}$ LOW:

$t_{PD} \leq t_{CKR} - t_{AR} - t_{SEN} = 13\ ns$

The HOTLink data and $\overline{RDY}$ pulse timing constraints to the pipeline register are

Data set-up time:

$t_A + t_{SU} \leq t_{CKR}$

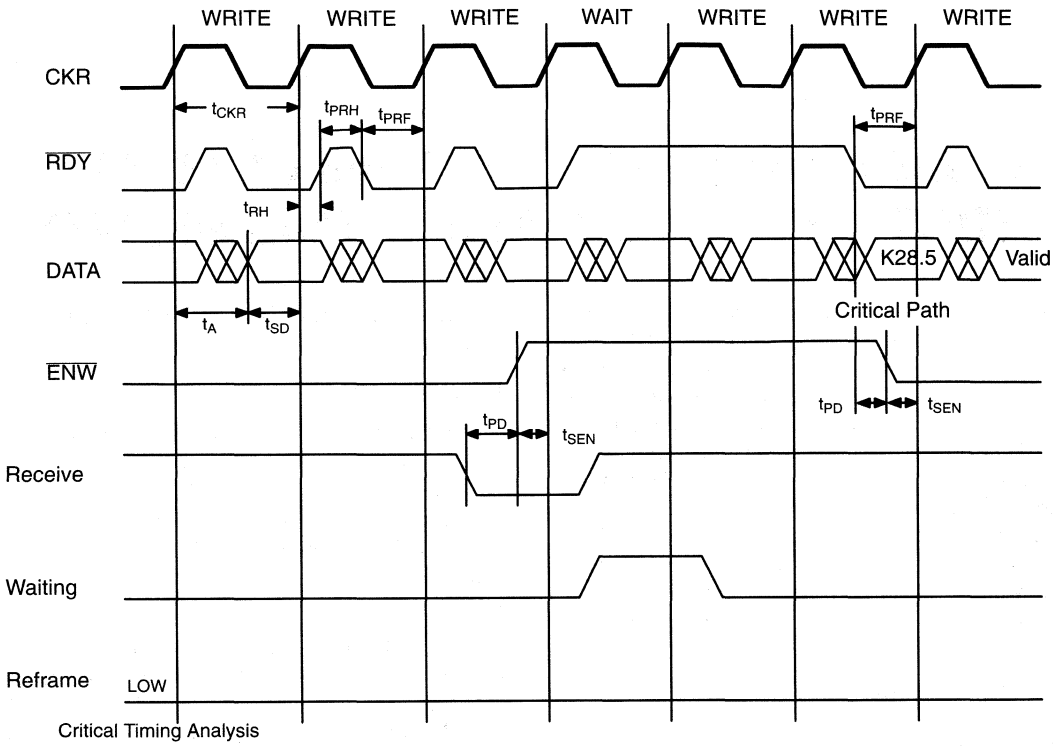
$\overline{RDY}$ set-up time:

$t_{SU} \leq 1/2 t_{CKR} - 3ns$

These constraints are easily met.

Conclusion

The HOTLink transmitter/receiver interfaces to clocked FIFOs can operate at speeds up to 330 Mbits/s with no external logic. Simple state machine controllers enable the transmission and reception of serial data and enable the HOTLink Built-In Self-Test capability.



Critical Timing Analysis

1. Data set-up time:
$$t_A + t_{SD} \leq t_{CKR}$$

2. Write enable set-up time from $\overline{RDY}$ going LOW:
$$t_{PD} + t_{SEN} \leq t_{PRF}$$

Figure 6. Receiver Timing Diagram

Document #: LOG0006–0293

HOTLink is a trademark of Cypress Semiconductor Corporation.

Interfacing the CY7B923 and CY7B933 (HOTLink™) to a Wide Data Clocked FIFO

This application note considers general interfacing issues between the Cypress CY7B923/CY7B933 (HOTLink™) Transmitter/Receiver and Cypress clocked FIFOs. The focus is on applications with a 36-bit data bus requiring high data transfer rates. A parallel FIFO solution is recommended for applications requiring large data bandwidth. Four FIFOs can achieve parallel data transfers on and off a 36-bit bus at rates of up to 280 Mbytes/s. The HOTLink serial link can transfer data at a serial rate of 330 Mbits/s. The FIFOs act as asynchronous storage buffers between the data bus and the serial link.

Transmitter Interface

This section describes the design considerations of a high-speed transmitter interface with FIFO (First In First Out) data buffers. The design implements basic data transmission and serial link testing capabilities. The transmitter is intended to interface to a higher-level controller responsible for coordinating bus transactions and handling the various protocol layers. The design considerations are easily extended to handle specific design requirements.

The transmitter interface consists of four Cypress CY7C441/3−14 clocked FIFOs buffering data between a 36-bit data bus and a Cypress HOTLink Transmitter. A 4:1 multiplexer (9 bits wide) funnels the wide FIFO data into the HOTLink parallel port. A local state machine controller coordinates the flow of data between the FIFOs and HOTLink. The FIFO−data bus interface and local controller architecture are left unspecified for generality. A block diagram of the FIFO-HOTLink interface is shown in *Figure 1*.

Data Multiplexers

The 4:1 multiplexers are part of the critical data path timing. These multiplexers can be implemented in several ways. Standard high-speed 153 dual 4:1 multiplexers can be used. Five of these devices are needed to accommodate 9-bit data. 74ACT153s with a maximum t_{SZ} of 11.5 ns and t_{DZ} of 9.5 ns are sufficient.

The 4:1 multiplexers can also be implemented with three Cypress 16L8−10s. Each 16L8 can accommodate three 4:1 multiplexers. This solution provides a smaller footprint and improves the critical timing margins. Critical timing margins are discussed in the Critical Timing Analysis section of this application note.

Built-In Self-Test

The transmitter interface is capable of checking the functionality of the serial link by exercising the Built-In Self-Test (BIST) mode of HOTLink. To initiate BIST, the $\overline{\text{BISTEN}}$ pin is held LOW, resulting in the transmission of the sequence ...1 0 1 0... . The $\overline{\text{ENN}}$ (Enable Next Parallel Data) pin is then pulled LOW to enable transmission of the BIST test pattern. HOTLink will assert the $\overline{\text{RP}}$ (Read Pulse) pin LOW at the beginning of BIST and will pulse it HIGH once per BIST loop. $\overline{\text{RP}}$ can be used to count the number of BIST loops sent. During BIST, HOTLink ignores data at its parallel port and the FIFOs do not perform any reads.

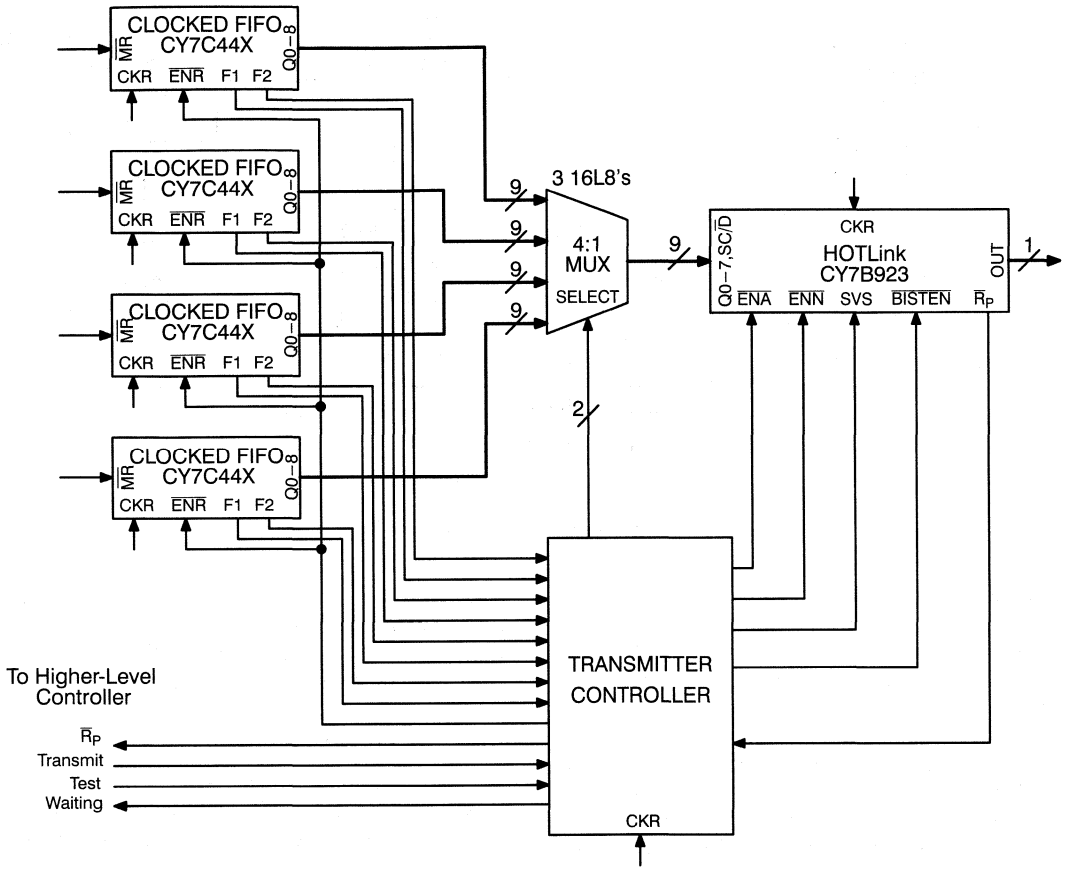


Figure 1. Transmitter Interface Block Diagram

Resetting the FIFOs

The higher-level controller should reset the FIFOs at power-up, before a new block of data is transmitted, or if an error is detected. Resetting or clearing the FIFOs is accomplished by pulsing the $\overline{MR}$ (Master Reset) pin on the FIFOs LOW. Neither a read nor a write can occur on the cycles immediately preceding, during, or following the assertion of $\overline{MR}$. $\overline{MR}$ must be glitch free. During the FIFO Master Reset cycle, the local transmitter controller should be in the WAIT state (see *Figure 2*). The higher-level controller is responsible for insuring that these conditions are met while performing the Master Reset cycle.

Transmitter Controller State Description

The local transmitter controller is responsible for reading data from the parallel FIFOs via the mux select lines and initiating the HOTLink BIST feature. The controller can be synthesized into a PLD or FPGA. Timing requirements of the controller are considered in the next section.

The local controller waits in the WAIT state while data is loaded into the FIFO. Meanwhile, HOT-

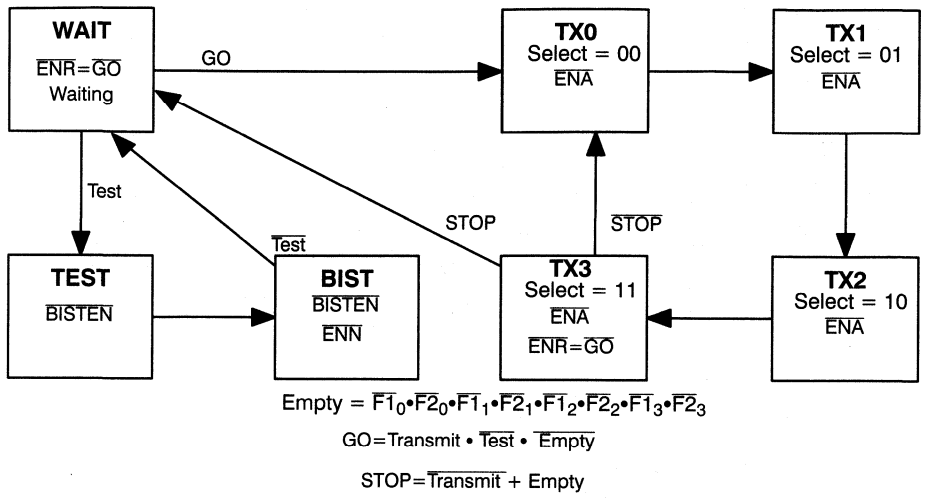


Figure 2. Transmitter Controller State Diagram

Link will transmit Idle special characters (K28.5). When the higher-level controller asserts the Transmit signal, the local transmitter controller issues a read ($\overline{ENR}$ LOW) to all the FIFOs and transitions to the TX0 state.

The transmit states (TX0–3) select data from the FIFOs in an ordered sequence. The TX0 state selects the byte out of FIFO0 for transmission and then transitions to the TX1 state. The TX1 state selects a byte out of FIFO1 and then transitions to the TX2 state. The TX3 state is responsible for checking the flags to determine if all of the FIFOs are empty, and then asserts $\overline{ENR}$ if they are not. (The controller can be designed to report an error if not all FIFOs are empty at the same time.) The transmit loop continues until all the FIFOs are empty or until Transmit is deasserted. Control then returns to the WAIT state. The Waiting signal should be monitored to determine when data transmission has ceased.

The state diagram of the local transmitter controller includes states for exercising the Built-In Self-Test capabilities of HOTLink. The local state machine enters the BIST state from the WAIT state when the higher-level controller asserts the Test signal. BIST is exited when Test is deasserted. The higher-level controller monitors $\overline{RP}$ for BIST loop counting. $\overline{RP}$ will pulse LOW one time per BIST loop. *Figure 2* illustrates the controller state diagram.

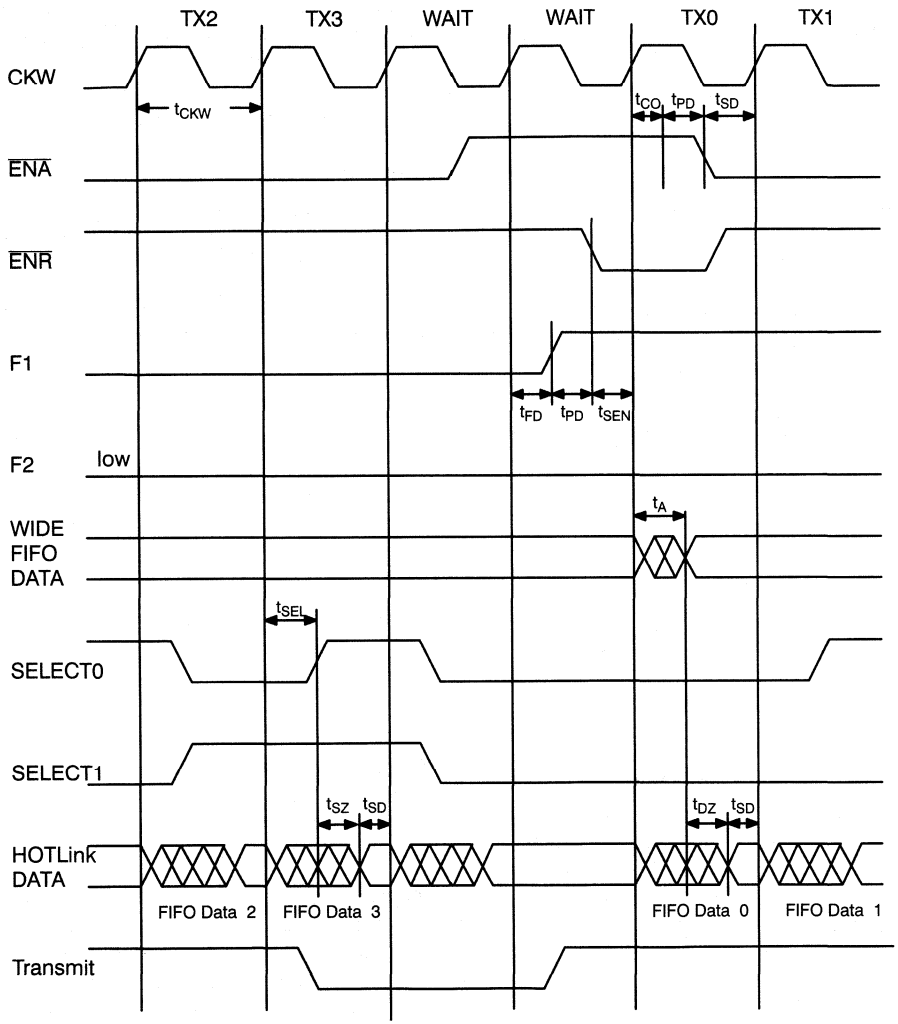
Critical Timing Analysis

The timing analysis in *Figure 3* highlights three critical data timing paths. The first critical path arises in the WAIT or TX3 states from the delay associated with decoding the flags and generating the read enable for the clocked FIFOs. The FIFO delay for generating the flags, t_{FD}, is 10 ns. The delay due to the controller decoding the flags and generating the enable is represented as t_{PD}. The read enable set-up time for FIFOs, t_{SEN}, is 7 ns ($t_{SEN} > t_{SD}$). The constraint imposed upon the controller is

$$t_{PD} \leq t_{CKW} - t_{FD} - t_{SEN}$$

With a-30 ns clock period, the signal propagation delay through the controller must be $t_{PD} \leq 13$ ns excluding trace delays and clock skew. This timing analysis assumes that the state register outputs are fed back to the controller before the flags signals are valid ($t_{CO} < t_{FD}$).

The second critical timing case assumes that data is available at the mux before the data selector signals



Critical Timing Analysis

1. Read enable set-up time:
$$t_{FD} + t_{PD} + t_{SEN} \leq t_{CKW}$$

2. HOTLink data set-up time from MUX data select:
$$t_{SEL} + t_{SZ} + t_{SD} \leq t_{CKW}$$

3. HOTLink data set-up time from FIFO data access:
$$t_A + t_{DZ} + t_{SD} \leq t_{CKW}$$

Figure 3. Transmitter Timing Diagram

($t_{SEL} > t_A$, where the delay from a clock edge to the arrival of the data selectors at the muxes is t_{SEL}). The delay from the selector pins to valid output data is t_{SZ}. The data set-up time to HOTLink, t_{SD}, is 5 ns. The critical timing associated with this path is

$$t_{SEL} + t_{SZ} + t_{SD} \leq t_{CKW}$$

The time to generate the data selectors from the controller is minimized by using the low-order bits of the state machine as the selectors and assigning TX0–3 to these states. This decreases the hardware required for the controller and reduces the selector signal-generation time to the clock-to-output time (t_{CO}) of the state registers. Assuming a 30-ns clock and t_{CO}=10 ns, the mux delay must be $t_{SZ} \leq 15$ ns.

The delay through the mux from valid input data to valid output is t_{DZ}. Assuming that the data selectors arrive before the data ($t_A > t_{SEL}$), the critical timing of this path is given by

$$t_A + t_{DZ} + t_{SD} \leq t_{CKW}$$

The data access time of the FIFOs, t_A, is 10 ns. With a 30-ns clock period, the constraint imposed upon the mux is $t_{DZ} \leq 15$ ns, assuming no trace delays or clock skew.

Receiver Interface

In this section a solution is presented for interfacing a HOTLink receiver to a 36-bit data bus. Control of the interface is simple and is easily adapted to system requirements. The four parallel CY7C451/3–14 FIFOs provide a high-speed interface to the data bus, allowing parallel transfer at rates up to 280 Mbytes/s. The serial link can receive data at serial rates up to 330 Mbits/s. The receiver interface is designed to provide proper word alignment in the FIFOs after synchronization to the data stream has been achieved. *Figure 4* shows a block diagram of the HOTLink-FIFO receiver interface.

Reframe

The receiver interface must synchronize itself to the incoming data and then store the data in the FIFOs with proper word alignment. The HOTLink RF (Reframe) input is used to synchronize the receiver to the transmitted data. Assertion of RF forces HOTLink to synchronize its internal bit counter with the boundary of a K28.5 character. HOTLink will respond by asserting $\overline{RDY}$ LOW when the first K28.5 is received. Reframing may be performed before data storage in order to synchronize HOTLink to the incoming serial data stream.

Idle Decoder

The Idle Decoder decodes the three types of idle characters: K28.5 (C5.0), −K28.5 (C1.7), +K28.5 (C2.7). These idle characters are used to signal the boundary of data words to be read into the FIFOs. A logic equation for the Idle Decoder is contained in *Figure 5*. A–H refer to HOTLink output pins Q0–Q7. When the Receive1 signal is asserted by the higher-level controller to the local controller, reception of any of these idle characters will trigger received data to be continually stored in the FIFOs starting with FIFO0 (*Figure 5*). The combinatorial delay through the decoder is modeled as t_{ID}.

Data Path and Controller

The HOTLink receiver parallel port interfaces directly to the FIFOs' write ports. A pipeline register may be inserted to improve timing margins or allow the FIFOs to be programmed. A local receiver controller coordinates the data flow and enables the HOTLink receiver BIST feature. The local receiver controller interfaces to a higher-level controller that coordinates all of the protocol layers of the link and the data bus transactions.

The higher-level controller instructs the local controller when to start data reception. A K28.5 character delimits the start of a data transmission. When this character is detected by either HOTLink or the Idle Decoder, the local controller writes the incoming data into the 45X FIFOs. The writing process continues until the higher-level controller signals the local receiver controller to stop.

The FIFO flags are decoded to signal when the FIFOs are empty or are full. A full FIFO will ignore attempted writes. The 45X FIFO features programmable Almost Full and Almost Empty flags that can assist in signaling when the FIFO is becoming too full. Programmable flag signals are left out of the design for clarity.

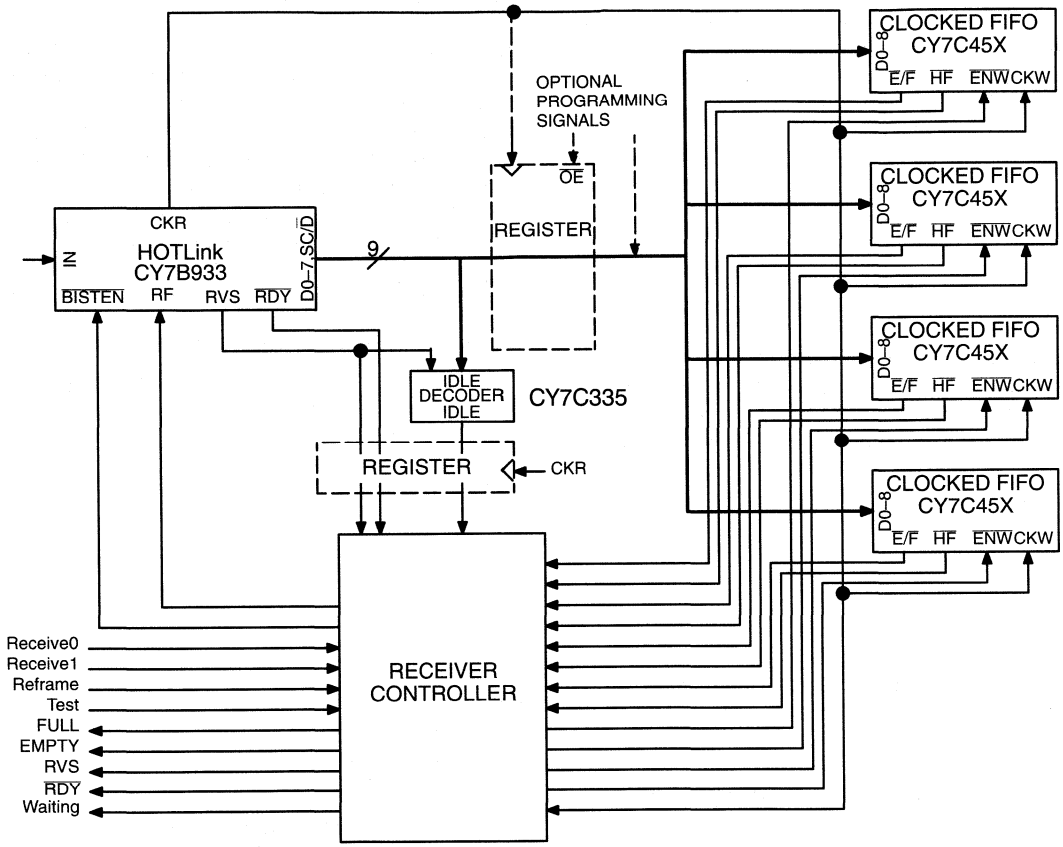


Figure 4. Receiver Interface Block Diagram

The Cypress 45X family of clocked FIFOs feature three-state data output drivers for direct interfacing to a data bus. The higher-level controller is responsible for reading words from the FIFOs' read port to the data bus.

The architecture of the local receiver controller is unspecified, but can be implemented with a PLD or FPGA. State machine descriptions and a timing analysis of the data path and local receiver controller are provided in the next sections.

Optional Pipeline Registers

The optional pipeline registers increase the interface speed by capturing the $\overline{RDY}$ pulse and easing timing constraints on the controller. $\overline{RDY}$ is a 60% LOW duty cycle signal shaped for interfacing to generic asynchronous FIFOs. The LOW phase of $\overline{RDY}$ leaves less than $\frac{1}{2}t_{CKR} - 10$ ns to generate the FIFO write enable and meet the FIFOs' set up time. A 40-ns clock period (250 Mbit/s) allows 10 ns for the local controller to generate a FIFO enable. This time shrinks to 5 ns when a clock period of 30 ns (330

Mbit/s) is used. The optional pipeline register captures the delayed $\overline{RDY}$ pulse and allows it to be processed earlier during the next clock cycle. The data and control signals must also be delayed by one clock cycle to ensure proper data alignment. A single CY7C335 PLD can be used to accommodate the data pipeline registers, the Idle Decoder, and the control signal delay registers. The timing implications of the registers are considered in the section on critical timing analysis.

The pipeline registers also isolate the HOTLink parallel port from the FIFO write ports while programming the FIFOs. A data pipeline register with three-state output drivers should be used so that data from an external source can be used to program the FIFOs. Additional states and control signals must be added to the controller. Programming is performed during the FIFO master reset cycle.

Built-In Self-Test

The Built-In Self-Test mode is exercised by asserting $\overline{BISTEN}$. Upon entering BIST, HOTLink will await the BIST initialization code and then assert $\overline{RDY}$ LOW when the code has been received. $\overline{RDY}$ will pulse HIGH once per received BIST loop. RVS will pulse HIGH if a byte pattern mismatch occurs. $\overline{RDY}$ and RVS can be monitored by the high-level controller to characterize the error rate.

Resetting and Programming the FIFOs

The higher-level controller should reset the FIFOs after power-up, before a new block of data is received, if an error occurs or in order to program the FIFOs. Resetting or programming the FIFOs is accomplished by pulsing the $\overline{MR}$ pin on the FIFOs LOW. Neither a read nor a write can occur on the cycles immediately preceding, during, or following the assertion of $\overline{MR}$ unless the FIFOs are being programmed. FIFO programming information is contained in the CY7C451/3 data sheet. $\overline{MR}$ must be glitch free. The receiver controller should only be in the WAIT or PROGRAM states during a master reset. The higher-level controller is responsible for insuring that these conditions are met.

Controller State Description

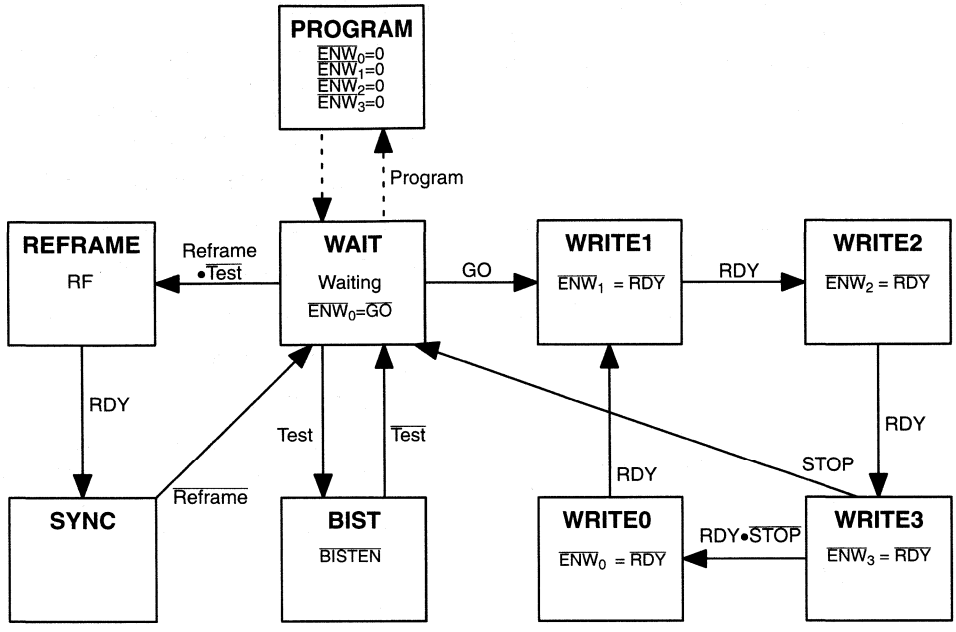
A state diagram for the receiver interface controller is shown in *Figure 5*. Five simple signals control the interface. The Receive0 and Receive1 signals are used to initiate and stop the reception of data. Reframe is used to synchronize the receiver to the serial data stream. Test causes HOTLink to perform BIST. Waiting is an output signal that indicates that the receiver is in the WAIT state.

Full and Empty signals are decoded for use by the higher-level controller to assist in managing data out of the FIFOs. The programmable flags may also be decoded but are not shown. A full FIFO ignores attempted writes resulting in lost data. Monitoring the state of the FIFOs is the responsibility of the higher-level controller. Resetting the FIFOs by pulsing $\overline{MR}$ LOW is also the responsibility of the higher-level controller.

The REFRAME state is used to synchronize the receiver to the incoming serial data stream. The REFRAME state asserts RF to the HOTLink receiver, signaling it to synchronize its internal bit counter with the first-received K28.5 character. $\overline{RDY}$ will pulse LOW when a synchronized K28.5 character is available. The controller will transition back to the WAIT state when synchronization is achieved and the Reframe signal is deasserted.

Receive0 and Receive1 initiate the storing of data in the FIFOs from the WAIT state. The assertion of Receive0 causes the controller to look for the assertion of $\overline{RDY}$ in order to begin data storage. The assertion of Receive1 causes the controller to look for the assertion of IDLE in order to begin data storage. The received K28.5 is written into FIFO0 and then the write loop is entered. The choice of which receive mode to use depends on the serial link protocol.

The write loop continually writes valid characters into the FIFOs. $\overline{ENW0-3}$ are cycled in order as the data is received. The fullness of the FIFOs is ignored by the controller. The higher-level controller monitors the Full flag signal and takes corrective action if the FIFOs become too full. The deassertion of both receive signals will end the writing process and return control back to the WAIT state on the



$$IDLE = \overline{H}\bullet\overline{G}\bullet\overline{F}\bullet\overline{E}\bullet\overline{D}\bullet C\bullet\overline{B}\bullet A\bullet SC/\overline{D} + H\bullet G\bullet F\bullet\overline{E}\bullet\overline{D}\bullet\overline{C}\bullet\overline{B}\bullet A\bullet SC/\overline{D} + H\bullet G\bullet F\bullet\overline{E}\bullet\overline{D}\bullet\overline{C}\bullet B\bullet\overline{A}\bullet SC/\overline{D}$$

$$GO = (Receive0\bullet RDY + Receive1\bullet IDLE)\bullet\overline{Reframe}\bullet\overline{Test}$$

$$STOP = \overline{Receive0 + Receive1}$$

$$Empty = \overline{E}/\overline{F}_0\bullet\overline{HF}_0\bullet\overline{E}/\overline{F}_1\bullet\overline{HF}_1\bullet\overline{E}/\overline{F}_2\bullet\overline{HF}_2\bullet\overline{E}/\overline{F}_3\bullet\overline{HF}_3$$

$$Full = \overline{E}/\overline{F}_0\bullet\overline{HF}_0\bullet\overline{E}/\overline{F}_1\bullet\overline{HF}_1\bullet\overline{E}/\overline{F}_2\bullet\overline{HF}_2\bullet\overline{E}/\overline{F}_3\bullet\overline{HF}_3$$

Figure 5. Receiver Controller State Diagram

next word boundary. The higher-level controller should monitor the Waiting signal to determine when receiver controller has returned to the WAIT state.

The BIST state is included for handling the Built-In Self-Test. During BIST, writing to the FIFOs is disabled. HOTLink signals are passed on to the higher-level controller for error analysis. RVS will signal character reception errors. $\overline{RDY}$ will pulse HIGH once per BIST loop and should be used to count the number of completed BIST loops.

A single PROGRAM state that writes an external program word to all of the FIFOs in parallel can be added to the state machine. This state is entered and exited during a FIFO master reset cycle. The higher-level controller should assert $\overline{MR}$ LOW, put the data pipeline register in the high-impedance state, and then drive the external program word to the FIFO write ports. The higher-level controller then puts the local controller in the PROGRAM state. The program word is written into the FIFOs' internal program registers when the local controller exits the PROGRAM state.

Critical Timing Analysis

A critical timing analysis of both the pipelined and unpipelined receiver interfaces is presented in this section. A timing diagram with critical timing equa-

tions is provided in *Figure 6* for the receiver interface that does not include the optional pipeline registers. Timing for the pipelined case is very similar. The analysis assumes that the state register bits are valid before any critical signals are available to the controller.

The critical timing path constrains the propagation delays associated with the local receiver controller and Idle Decoder. The combinatorial timing delay through the controller is modeled as t_{PD}. The combinatorial delay through the Idle Decoder is modeled as t_{ID}.

Unpipelined Timing

The timing for the unpipelined configuration is as follows. Assuming t_{CKR}=30 ns and t_{SEN}=7 ns, the propagation delays are

Write enable generation time from $\overline{RDY}$ LOW:

$$t_{PD} \leq 1/2\ t_{CKR} - t_{SEN} - 3ns = 5\ ns$$

IDLE generation time from data:

$$t_{ID} \leq 4/5\ t_{CKR} - t_{SEN} - t_{PD} - 3\ ns = 9\ ns$$

These constraints require (approximately) $t_{PD} \leq 5$ ns and $t_{ID} \leq 9$ ns. With a 40 ns clock cycle, these timing constraints are relaxed to $t_{PD} \leq 10$ ns and $t_{ID} \leq 12$ ns.

Pipelined Timing

With the optional pipeline registers inserted the timing margins of the control logic are eased. Assuming the register access time is t_{AR}=10 ns and the register set up time is t_{SU}=5 ns

Write enable generation time from clock:

$$t_{PD} \leq t_{CKR} - t_{SEN} - t_{AR} = 13\ ns$$

IDLE generation time from data:

$$t_{ID} \leq 4/5\ t_{CKR} - t_{SU} - 3\ ns = 14ns$$

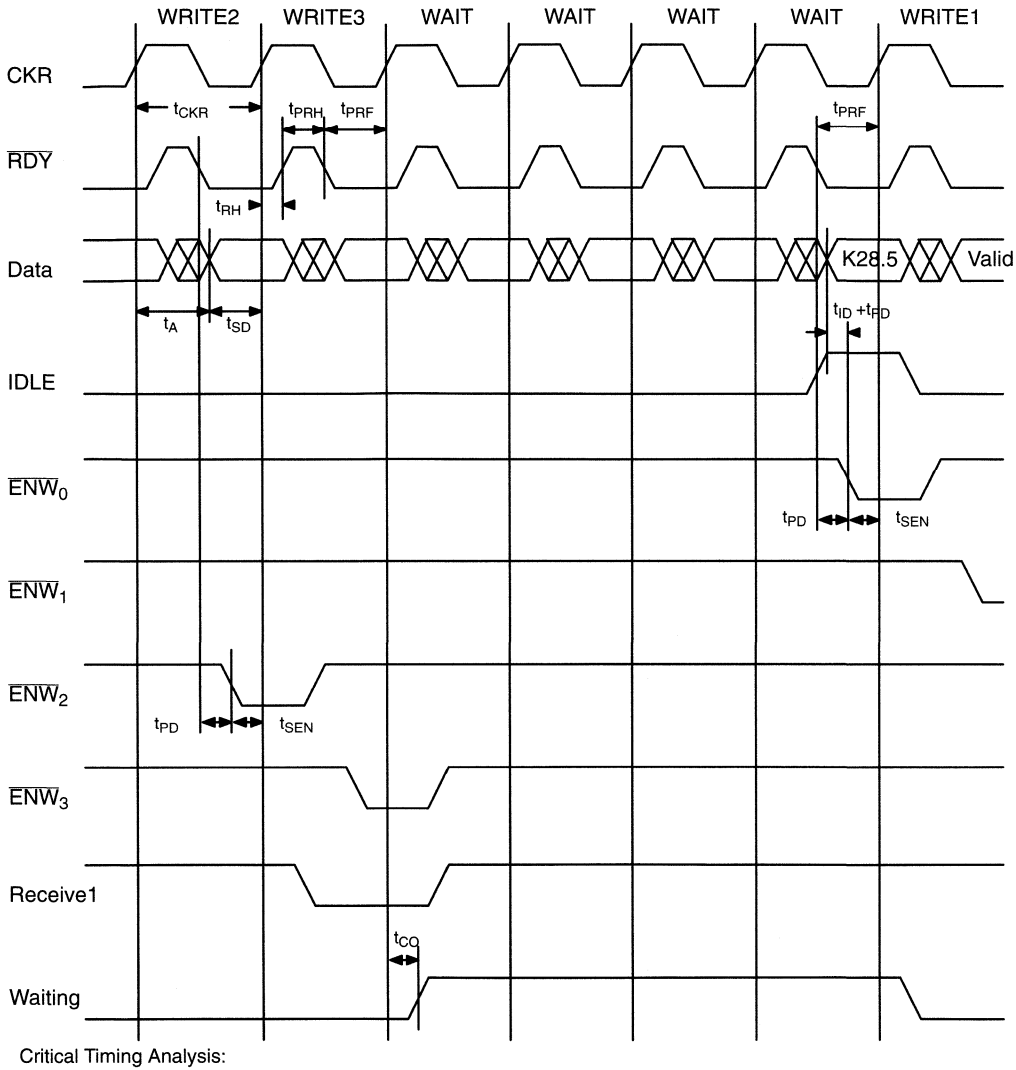
$\overline{RDY}$ capture timing:

$$t_{SU} \leq 1/2 t_{CKR} - 3ns = 12\ ns$$

The pipeline registers ease the receiver control logic timing margins to (approximately) 13 ns. The entire pipeline circuitry, including the Idle Decoder, can be synthesized into a single CY7C335–83 PLD while meeting these timing constraints.

Conclusion

The HOTLink Transmitter/Receiver interfaces to wide data FIFOs can operate at speeds of up to 330 Mbits/s with minimal interface logic. State machine controllers ensure proper word alignment during data transfers over the HOTLink serial link and provide Built-In Self-Test capability. Critical timing equations are provided. The interface designs are easily modified to meet specific demands.



Critical Timing Analysis:

1. Data set-up time
   $t_A + t_{SD} \leq t_{CKR}$
2. Write enable set-up time from $\overline{RDY}$ LOW
   $t_{PD} + t_{SEN} \leq t_{PRF}$
3. Write enable set-up time from idle HIGH:
   $t_A + t_{ID} + t_{PD} + t_{SEN} \leq t_{CKR}$

Figure 6. Receiver Timing Diagram

Document #: LOG0007–0293

HOTLink is a trademark of Cypress Semiconductor Corporation.

Data Communications – 5



Data Communications Section Contents and Abstracts

HOTLink™ Design Considerations **5–1**

This application note describes how to implement and characterize high-speed serial links made using the CY7B923 and CY7B933 HOTLink parts. Primary topics are an overview of how both HOTLink parts operate internally, how to work with ECL signals, and how to interface to optical fiber and electrical (copper) cables.

HOTLink Built-In Self-Test **5–62**

This application note describes some important features included in the HOTLink Transmitter and Receiver. It describes the Built-In Self-Test (BIST) function in detail, and describes several ways in which BIST can assist in the evaluation of HOTLink products and the evaluation of various transmission link-interconnect components. This detailed description is intended to expand upon the cursory information provided in the HOTLink datasheet.

Upgrade Your TAXI–275 with HOTLink **5–79**

This application note will explain how to upgrade TAXI–275™ (Am79168/Am79169) devices with the HOTLink (CY7B923/CY7B933) devices from Cypress Semiconductor. It will aid in the migration of TAXI–275 designs to the HOTLink architecture. This note begins with an introduction to HOTLink and then gives advantages of HOTLink and replacement suggestions for the TAXI–275 devices.

Replace Your TAXI–125™ and TAXI–175™ **5–92**

This application note will explain how to replace TAXIchip™ devices with the HOTLink devices from Cypress Semiconductor. This note begins with an introduction to HOTLink and then gives advantages and replacement suggestions for the TAXI–125 and TAXI–175 devices.

Parallel Cyclic Redundancy Check (CRC) for HOTLink **5–105**

This note discusses using CRC codes to insure data integrity over high-speed serial links, such as Fibre Channel, ESCON™ and other standards supported by Cypress's CY7B923 and CY7B933 HOTLink devices. It also shows why parity is not useful and then describes the most common CRC codes (CRC–16 and CRC–32) used in high-speed communications systems. Finally, an algorithm for calculating CRC–16 one byte at a time and for calculating CRC–32 two bytes at a time is presented. These algorithms are necessary because of the impracticality of running programmable devices at the 155 to 330 MHz required to calculate the CRCs in a bit-serial fashion.

Fiber Channel Level 2 Design Considerations **5–111**

Fibre Channel is emerging as an industry standard for high-speed point-to-point data communications. This application note explores the control, protocol, and hardware interface requirements between level 1 and level 2 of the Fibre Channel standard. Design tradeoffs are considered for performance and cost. A design example is provided to illustrate a PLD based implementation.

Using High-Speed Serial Links to Supplement Parallel Data Buses **5–128**

Today's designers face a multitude of problems when trying to move data within their systems. These problems range from overtaxed parallel-bus bandwidth to a lack of pins at the card edge connector. Even routing paral-

lel buses around today's dense circuit boards is very difficult. This application note discusses using high-performance serial links as a solution to some of these bottlenecks. A serial approach provides three immediate benefits: first, bandwidth may be offloaded from the backplane bus; second, connector pins are saved; and, third, circuit board routing is made much easier since only two traces have to be routed for the data path (versus one for each data bus bit).

Using the CY7B923 as an ECL Clock Source **5–135**

This application note details the use of an inexpensive data communications transmitter device as a high-precision, flexible, and programmable Emitter-Coupled-Logic (ECL) or Positive-Emitter-Coupled Logic (PECL) clock source. Issues concerning clock characteristics, stability, distribution and design techniques are discussed in detail. Information is provided to allow the user to configure the device for a variety of applications.

CY9266 HOTLink Demo Board User's Guide **5–141**

This document describes the construction, interfaces, and operation of the CY9266–F (optical), CY9266–T (shielded twisted-pair/twinax), and CY9266–C (coaxial cable) HOTLink demo boards. These boards implement a bidirectional parallel-to-serial and serial-to-parallel communications link, capable of operation at serial rates of 160 to 330 Mbits/second (16–33 Mbytes/second). Complete schematics, parts lists, and artwork are included.

HOTLink CY7B933 $\overline{\text{RDY}}$ Pin Description **5–208**

This application note describes the behavior of the $\overline{\text{RDY}}$ (Ready) pin of the CY7B993 HOTLink Receiver in several modes of operation: Encoded, Bypass, and BIST (Built-In Self-Test). The $\overline{\text{RDY}}$ pin indicates the status of the HOTLink Receiver control logic and output pins. Its function and timing are dependent on the state of the Mode, $\overline{\text{BISTEN}}$ (Built-In Self-Test Enable), and RF (Reframe) pins. The detailed information contained in this application note should serve as a guide when integrating the $\overline{\text{RDY}}$ pin into the interface logic.

HOTLink CY7B9331 OLC Receiver **5–214**

This application note describes the special characteristics of the HOTLink CY7B991 OLC Receiver. The characteristics described in this document are intended to make the HOTLink OLC Receiver exactly compatible with the functionality and timing of the IBM™ Optical Link Card (OLC-266). The behavior of OLC specific pins is described in detail. The information presented in this application note should be used as a guide when integrating the CY7B9331 into a system.

Interfacing with the SST **5–221**

This application note describes how to interface the CY7B951 SONET/SDH Serial Transceiver (SST) with other physical-layer devices. The SST performs clock and data recovery from a SONET (SDH) 51.84 or 155.52 Mb/sec interfaces and can be used in a variety of SONET and ATM applications. The application note will begin with a brief introduction to the SST. Next, interface examples will be given that illustrate how to connect the SST to two different ATM controller devices; the first from PMC–Sierra called the PM5345 SUNI and the second called the WAC–013 from Integrated Telecom Technologies (IgT).

HOTLink™ Design Considerations

Application Note Overview

The HOTLink family of data communications products provide a simple and low-cost solution to high-speed data transmission. While these products are easy to use, the methods used to connect them to high-speed serial interfaces are often not intuitive. This document provides a basic level of explanation of the parallel and serial interface characteristics, and provides some cookbook solutions for interfacing them to different types of parts and media.

Primary Topics

The primary topics covered in this application note are

- HOTLink Overview
- HOTLink Serial Signal Characteristics
- Terminating HOTLink Serial Signals
- Interfacing to HOTLink
- Serial Link Support Components

HOTLink Overview

HOTLink Features

- Fibre Channel compliant
- IBM ESCON™ compliant
- ATM Compatible
- 8B/10B-coded or 10-bit unencoded
- 160- to 330-Mbps data rate
- TTL synchronous I/O
- No external PLL components
- Triple ECL 100K serial outputs
- Dual ECL 100K serial inputs
- Low power: 350 mW (Tx), 650 mW (Rx)
- Compatible with fiberoptic modules, coaxial cable, and twisted-pair media
- Built-In Self-Test
- Single +5V supply
- 28-pin SOIC/PLCC/LCC
- 0.8µ BiCMOS

Functional Description

The CY7B923 HOTLink Transmitter and CY7B933 HOTLink Receiver are point-to-point communications building blocks that transfer data over high-speed serial links (fiberoptic, coax, and twisted/parallel pair) at 160- to 330-Mbits/second. *Figure 1* illustrates typical connections to host systems or controllers.

Eight bits of user data or protocol information are loaded into the HOTLink Transmitter and are encoded. Serial data is shifted out of the three differential Pseudo ECL (PECL) serial ports at the bit-rate (which is 10 times the byte-rate).

The HOTLink Receiver accepts the serial bit stream at its differential line receiver inputs, and using a completely integrated Phase-Locked-Loop (PLL) clock synchronizer recovers the timing information necessary for data reconstruction. The bit stream is deserialized, decoded, and checked for transmission errors. The recovered byte is presented in parallel to the receiving host along with the synchronized byte-rate clock.

The 8B/10B encoder/decoder (Reference 1, 2) can be disabled in systems that already encode or

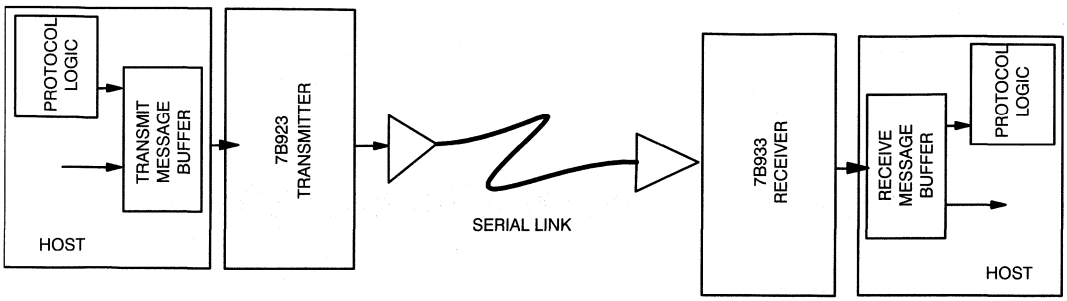


Figure 1. HOTLink System Connections

scramble the transmitted data. Signals are available to create a seamless interface with both asynchronous FIFOs (i.e., Cypress's CY7C42X) and clocked FIFOs (i.e., Cypress's CY7C44X). A Built-In Self-Test pattern generator and checker allows testing of the transmitter, receiver, and the connecting link as a part of a system diagnostic check.

HOTLink devices are ideal for a variety of applications where a parallel interface can be replaced with a high-speed point-to-point serial link. Applications include interconnecting workstations, servers, mass storage, and video transmission equipment.

CY7B923 HOTLink Transmitter Description

The function of the HOTLink Transmitter is to convert byte-rate parallel data into a high speed serial data stream. A logic block diagram of the transmitter is shown in *Figure 2*.

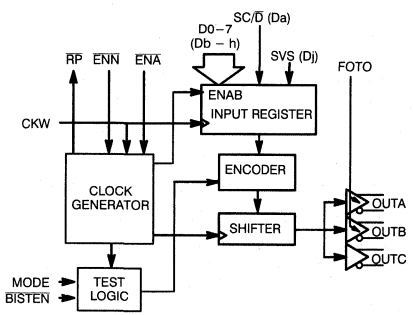


Figure 2. CY7B923 Transmitter Logic Block Diagram

Input Register

The Input register holds the data to be processed by the HOTLink transmitter and allows the input timing to be made consistent with standard FIFOs. The Input register is clocked by CKW (Clock Write) and loaded with information on the D0–D7, SC/$\overline{D}$ (Special Character/Data select), and SVS (Send Violation Symbol) pins. Two enable inputs ($\overline{ENA}$ and $\overline{ENN}$) allow the user to choose when data is to be sent. Asserting $\overline{ENA}$ (Enable, active LOW) causes the inputs to be loaded on the rising edge of CKW. If $\overline{ENN}$ (Enable Next, active LOW) is asserted when CKW rises, the data present on the inputs on the *next* rising edge of CKW will be loaded into the input register. These two inputs allow proper timing and function for compatibility with either asynchronous FIFOs or clocked FIFOs without external logic.

In BIST mode, the Input register becomes the signature pattern generator by logically converting the parallel input register into a Linear Feedback Shift Register (LFSR). When enabled, this LFSR generates a 511-byte sequence that includes all Data and Special Character codes, including the explicit violation symbols. This pattern provides a predictable but pseudo-random sequence that can be matched to an identical LFSR in the HOTLink Receiver. For additional information see the Cypress Semiconductor application note "HOTLink Built-In Self-Test."

Encoder

The Encoder transforms the input data held by the Input register into a form more suitable for transmission on a serial interface link. The code used is

specified by ANSI X3T11 (Fibre Channel) (Reference 3) and the IBM ESCON channel (Reference 4) (code tables are available in the CY7B923/CY7B933 datasheet). The eight D0–D7 data inputs are converted to either a Data symbol or a Special Character, depending upon the state of the SC/$\overline{D}$ input. If SC/$\overline{D}$ is HIGH, the data inputs represent a control code and are encoded using the Special Character code tables. If SC/$\overline{D}$ is LOW, the data inputs are converted using the Data code table. If a byte-time passes with the inputs disabled, the Encoder will output a Special Character Comma (K28.5 or SYNC) to maintain link synchronization. The SVS input forces the transmission of a specified Violation symbol to allow the user to check error handling logic in the system controller.

The 8B/10B coding function of the Encoder can be bypassed for systems that include an external coder or scrambler function as part of the controller. This bypass capability is controlled by setting the MODE select pin high. When in bypass mode, Da–Dj (note that bit order is specified by Fibre Channel 8B/10B code) become the ten inputs to the Shifter, with Da being the first bit to be shifted out.

Shifter

The Shifter accepts parallel data from the Encoder once each byte-time and shifts it to the serial interface output buffers using a PLL multiplied bit-clock that runs at ten (10) times the byte-clock (CKW) rate. Timing for the parallel transfer is controlled by the counter included in the Clock Generator, and is not affected by signal levels or timing at the input pins.

OutA, OutB, OutC

The serial interface ECL output buffers (100K signal levels referenced to +5V) are the drivers for the serial media. They are all connected to the Shifter and contain the same serial data. Two of the output pairs (OUTA± and OUTB±) are controlled by the FOTO input and can be disabled by the system controller to force a logical zero (i.e., "light off") at the outputs. The third output pair (OUTC±) is not affected by FOTO and will supply a continuous data stream suitable for loop-back testing of the subsystem.

OUTA± and OUTB± will respond to FOTO input changes within a few bit times. However, since FOTO is not synchronized with the transmitter data stream, the outputs will be forced off or turned on at arbitrary points in a transmitted byte. This function is intended to augment an external laser safety controller and as an aid for Receiver PLL testing.

In wire-based systems, control of the outputs may not be required, and FOTO can be strapped LOW. The three output pairs are intended to add system and architectural flexibility by offering identical serial bit streams with separate interfaces for redundant connections or for multiple destinations. Unneeded outputs can be left open or wired to V_{CC} to disable and power down the unused output circuitry.

Clock Generator

The clock generator is an embedded phase-locked loop (PLL) that takes a byte-rate reference clock (CKW) and multiplies it by ten (10) to create a bit-rate clock for driving the serial shifter. The byte-rate reference comes from CKW, the rising edge of which clocks data into the Input register. This clock must be a crystal-referenced pulse stream that has a frequency between the minimum and maximum specified for the HOTLink Transmitter/Receiver pair. Signals controlled by this block form the bit-clock and the timing signals that control internal data transfers between the Input register and the Shifter.

The read pulse ($\overline{RP}$) is derived from the feedback counter used in the PLL multiplier. It is a byte-rate pulse stream with the proper phase and pulse widths to allow transfer of data from an asynchronous FIFO. Pulse width is independent of CKW duty cycle, since proper phase and duty cycle is maintained by the PLL. The $\overline{RP}$ pulse stream will insure correct data transfers between asynchronous FIFOs and the transmitter input latch with no external logic.

Test Logic

Test logic includes the initialization and control for the Built-In Self-Test (BIST) generator, the multiplexer for Test mode clock distribution, and control logic to properly select the data encoding. Test logic is discussed in more detail in the CY7B923/CY7B933 HOTLink datasheet.

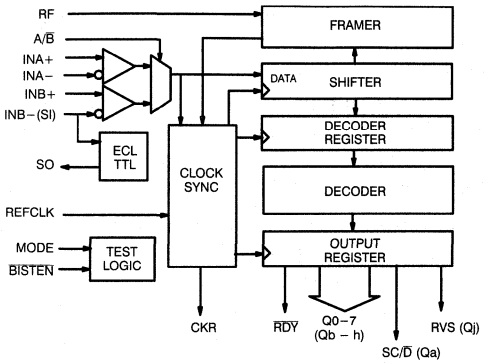


Figure 3. CY7B933 Receiver Logic Block Diagram

CY7B933 HOTLink Receiver Description

The function of the HOTLink Receiver is to convert a high-speed serial data stream into byte-rate parallel data. A logic block diagram of the receiver is shown in *Figure 3*.

Serial Data Inputs

The HOTLink Receiver has two differential line receivers (INA± and INB±) that can be selected as inputs for the serial data stream. INA± or INB± is selected with the A/$\overline{B}$ input. INA± is selected when A/$\overline{B}$ HIGH and INB± is selected when A/$\overline{B}$ LOW. The threshold of A/$\overline{B}$ is compatible with ECL 100K signals. TTL logic elements can be used to select the INA± or INB± inputs by adding a resistor voltage divider to a TTL driver connected to A/$\overline{B}$ (see *Figure 35*). The differential sensitivity of INA± and INB± will accommodate wire interconnect with filtering losses or transmission line attenuation greater than 20 dB ($V_{DIF} \geqq 50$ mV). These inputs can alternatively be directly connected to fiberoptic interface modules (any ECL logic family, not limited to ECL 100K) with up to 1.2V of differential signal. The common-mode tolerance accommodates a wide range of signal termination voltages. The highest HIGH input that can be tolerated is $V_{IN} = V_{CC}$, and the lowest LOW input that can be interpreted correctly is V_{IN} = GND+2.0V.

ECL-TTL Translator

The function of the INB(INB+) input and the SI(INB−) input is determined by the connection on the SO output pin. If the ECL/TTL translator function is not required, the SO output is wired to V_{CC}. A sensor circuit detects this connection and causes the inputs to become INB± (a differential line-receiver for serial-data input). If the ECL/TTL translator function is required, the SO output is connected to a normal TTL load (typically one or more TTL inputs, but no pull-up resistor) and the inputs become INB (single-ended ECL 100K-level serial-data input) and SI (single-ended ECL 100K-level status input).

This positive-referenced ECL-to-TTL translator is provided to eliminate external logic between an ECL carrier-detect or link status signal and a TTL input in the control logic. The input threshold is compatible with ECL 100K-levels (+5V referenced).

Clock Sync

The Clock Synchronizer function is performed by an embedded phase-locked loop (PLL) that tracks the frequency of the incoming serial bit stream and aligns the phase of its internal bit-rate clock to the serial data transitions. This block contains the logic to transfer the data from the Shifter to the Decode register once every byte. The counter that controls this transfer is initialized by the Framer logic. CKR is a buffered output derived from the bit counter used to control Decode register and Output register transfers.

Clock output logic is designed such that when reframing causes the counter sequence to be interrupted, the period and pulse width of CKR will never be less than normal. Reframing may stretch the period of CKR by up to 90%, and either CKR Pulse Width HIGH or Pulse Width LOW may be stretched, depending on when reframe occurs.

The REFCLK input provides a byte-rate reference frequency to improve PLL acquisition time and limit unlocked frequency excursions of CKR when no data is present at the serial inputs. The frequency of REFCLK is required to be within ±0.1% of the

frequency of the clock that drives the transmitter CKW pin.

Framer

Framer logic checks the incoming bit stream for the pattern that determines the byte boundaries. This combinatorial logic filter looks for the ANSI Fiber Channel symbol defined as a Special Character Comma (K28.5) (Reference 3). When it is found, the free-running bit counter in the Clock Sync block is synchronously reset to its initial state, thus framing the data on the correct byte boundaries.

Random errors that occur in the serial data can corrupt some data patterns into a bit pattern identical to a K28.5, and thus cause an erroneous data-framing error. The RF input prevents this by inhibiting reframing during times when normal message data is present. When RF is held LOW, the HOTLink Receiver deserializes the incoming data without trying to reframe the data to incoming patterns. When RF rises, $\overline{RDY}$ is inhibited until a K28.5 has been detected, after which $\overline{RDY}$ resumes its normal function. While RF is HIGH, it is possible that an error could cause misframing, after which all data will be corrupted. Likewise, a K28.7 followed by D11.x, D20.x, or an SVS (C0.7) followed by D11.x will cause erroneous framing. These sequences must be avoided while RF is HIGH.

If RF remains HIGH for greater than 2048 bytes, the framer switches to double-byte framing, requiring two K28.5 Special Characters within five bytes.

Shifter

The Shifter accepts serial data from one of the Serial Data input pairs one bit at a time, as clocked by the Clock Sync logic. Data is examined by the Framer on each bit, and is transferred to the Decode register once per byte.

Decode Register

The Decode register accepts data from the Shifter once per byte as determined by the logic in the Clock Sync block. It is presented to the Decoder and held until it is transferred to the output latch.

Decoder

Parallel data is transformed from ANSI Fiber Channel 8B/10B codes (Reference 3) back to "raw data" in the Decoder. This block uses the standard decoder patterns found in the Valid Data Characters and Valid Special Character Codes and Sequences (code tables are available in the CY7B923/CY7B933 datasheet). Data patterns are signaled by a LOW on the SC/$\overline{D}$ output and Special Character patterns are signaled by a HIGH on the SC/$\overline{D}$ output. Unused patterns or disparity errors are signaled as errors by a HIGH on the RVS (Received Violation Symbol) output and by specific Special Character codes.

Output Register

The Output register holds the recovered data (Q0–Q7, SC/$\overline{D}$, and RVS) and aligns it with the recovered byte clock (CKR). This synchronization insures proper timing to match a FIFO interface or other logic that requires glitch free and specified output behavior. Outputs change synchronously with the rising edge of CKR.

In BIST mode, this register becomes the signature pattern generator and checker by logically converting itself into a Linear-Feedback Shift-Register (LFSR) pattern generator. When enabled, this LFSR generates a 511-byte sequence that includes all Data and Special Character codes, including the explicit violation symbols. This pattern provides a predictable but pseudo-random sequence that can be matched to an identical LFSR in the transmitter. When synchronized, it checks each byte in the Decoder with each byte generated by the LFSR and indicates errors using RVS. Patterns generated by the LFSR are compared after being buffered to the output pins and then fed back to the comparators, allowing test of the entire receive function.

In BIST mode, the LFSR is initialized by the first occurrence of the transmitter BIST loop start code D0.0 (D0.0 is sent only once per BIST loop). Once the BIST loop has been started, RVS will be HIGH for pattern mismatches between the received sequence and the internally generated sequence. Code rule violations or running disparity errors that occur as part of the BIST loop do not cause an error indication. $\overline{RDY}$ pulses high once per BIST loop

and can be used to check test pattern progress. The receiver BIST generator can be reinitialized by leaving and re-entering BIST mode.

Test Logic

Test logic includes the initialization and control for the Built-In Self-Test (BIST) generator, the multiplexer for Test mode clock distribution, and control logic for the decoder. Test logic is discussed in more detail in the CY7B923/CY7B933 HOTLink datasheet.

HOTLink Serial Signal Characteristics

The serial interfaces on the HOTLink Transmitter and Receiver are based on the standard for high-speed digital logic called Emitter-Coupled-Logic or ECL. This form of logic has been used commercially in integrated circuits since the early 1960s, and prior to that it was implemented in discrete form.

ECL is a non-saturating form of digital logic. ECL gets its name from how the emitters of a differential amplifier in the circuit are connected. The main features of this logic family are very high speed, low noise, and the ability to drive low-impedance transmission lines.

In the past, many engineers have avoided ECL as a logic family because it was different from the TLL-compatible families with which they were more familiar. Proper use of ECL requires the understanding and application of transmission lines, line termination, and power supply bypassing. Because of the faster speeds present in the newer TTL compatible families, these same disciplines are now required for TTL circuits as well.

ECL Signal Level Reference

The primary differences between ECL and other logic families are the signal levels used to represent the HIGH and LOW logic levels.

In the TTL and CMOS logic families, a LOW is usually some level close to V_{SS}, and a HIGH is usually some level close to V_{CC}. The ground or reference point for these measurements is usually the V_{SS} point, with V_{CC} set to +5V from that ground reference.

In standard ECL this changes significantly. Instead of having the ground reference at V_{SS}, it is placed at V_{CC}. This means that both HIGH and LOW logic levels exist at potentials that are negative with respect to ground. Standard ECL is specified as operating with a negative supply (−4.5V to −5.2V for V_{EE}). Since ground is only a reference point, it is also possible to operate ECL with a positive supply. When used in this mode ECL is usually referred to as PECL which means either Positive or Pseudo ECL.

ECL Basic Switch

Internally, ECL gates (or switches) operate using a current source whose current is directed through one of two paths back to V_{CC}. A schematic of this basic ECL switch is shown in *Figure 4* (Reference 5).

In this ECL switch, the state of the switch is determined by the voltage drop across R1 and R2. The output signal swing is set by the size of these resistors and the magnitude of the current passed through them.

The base of Q2 is biased at a fixed voltage called V_{BB}. This voltage determines at what level of V_{IN} on Q1 that the majority of the current flowing in the switch changes from R1 to R2. If V_{IN} is set to the same voltage as V_{BB}, the current divides equally between R1 and R2. Increasing V_{IN} by 125 mV above V_{BB} causes essentially all the current to be run through Q1 (and hence R1). Lowering V_{IN} to 125 mV below V_{BB} causes essentially all the current to flow through Q2. This means that an input swing

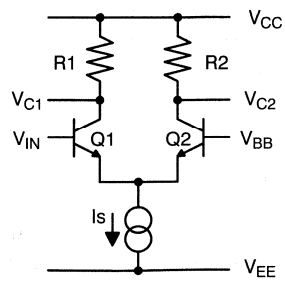


Figure 4. Basic ECL Switch

of as little as 250 mV can cause the ECL gate to switch completely from a 0 to a 1. To provide noise immunity and allow operation over a wide variety of conditions, the actual signal swing specified for ECL signals is around 800 mV.

Emitter-Follower

The switch shown in *Figure 4* can react very quickly but, because of its high-value resistor pull-ups (R1 and R2), its switching delay varies directly with load capacitance. To allow larger loads to be driven, and to make the output voltages compatible with the input of subsequent gates, additional transistors are added in an emitter-follower configuration as illustrated in *Figure 5*.

These emitter-follower transistors have a very low on impedance (5–7Ω). This allows ECL gates to drive transmission lines having impedances as low as 50Ω and can supply load currents of up to 50 mA.

The emitter-follower transistors have an uncommitted emitter as their output. This allows the transistor to source, but not sink, current. This is effectively the opposite of an open-collector output in a TTL part. To allow the output to function correctly, it requires a load that operates as a pull-down.

ECL Signal Levels

ECL signals operate over a very narrow and tightly controlled range. All of these signals are referenced from the V_{CC} pins of the parts. *Figure 6* shows the relationships of the different output and input levels

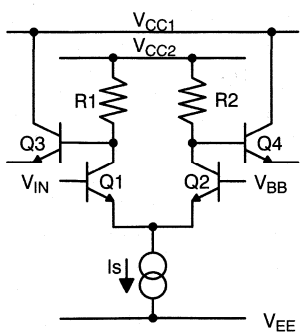


Figure 5. Buffered ECL Switch

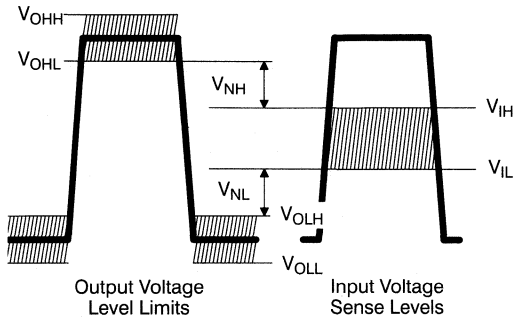


Figure 6. ECL Signal Levels

for ECL gates. The names of these levels are detailed in *Table 1*.

Table 1. ECL Signal Level Names

Name	Description
V_{OHH}	Highest Output High Voltage
V_{OHL}	Lowest Output High Voltage
V_{OLH}	Highest Output Low Voltage
V_{OLL}	Lowest Output Low Voltage
V_{IH}	Lowest Input High Voltage Threshold
V_{IL}	Highest Input Low Voltage Threshold
V_{NH}	High Input Noise Margin ($V_{OHL}-V_{IH}$)
V_{NL}	Low Input Noise Margin ($V_{OLH}-V_{IL}$)

ECL Output Signal Levels

ECL outputs are all referenced from V_{CC}. A typical ECL driver has an output-high level (V_{OH}) of V_{CC} – 0.85V and an output-low level (V_{OL}) of V_{CC} – 1.7V. These typical values are seldom specified for parts because a good design must be done using the range limits for these signals as listed in *Table 1*. Actual values for these levels vary by individual part type and ECL family.

ECL Input Signal Levels

ECL Inputs are also referenced from V_{CC}. A typical ECL receiver has an input high (V_{IH}) threshold of V_{CC} – 1.1V and an input low (V_{IL}) threshold of V_{CC} – 1.47V. These differences between the output and input high and low values translate directly into the usable noise margin (V_{NH} and V_{NL}) of a system.

Viewing ECL Signals

Proper viewing of ECL signals requires use an oscilloscope and probes with sufficient bandwidth to see the important features of the waveforms. Depending on the speed of the signals being viewed, different scope and probe characteristics are required.

Oscilloscope Bandwidth

Oscilloscope bandwidth is not a simple number; it is based on the combined bandwidths of multiple pieces of the measurement system. These can include the oscilloscope, the scope probe amplifier, the probe itself, and possibly other components.

The calculation for bandwidth is based on an inverse sum-of-squares as shown in *Equation 7.*

$$bw = \frac{1}{\sqrt{(\frac{1}{bw_1})^2 + (\frac{1}{bw_2})^2}} \qquad \text{Eq. 7}$$

Thus a scope with a 1-GHz bandwidth probe using a 1-GHz bandwidth amplifier would only have a usable bandwidth of 700 MHz.

The current ANSI Fibre Channel standard specifies the minimum system bandwidth for testing as 1.8 times the baud rate. For testing with the HOTLink parts (330 Mbaud), this translates to a minimum system bandwidth of 600 MHz. This is translated into a viewable rise time using *Equation 8* (Reference 6).

$$t_r = \frac{.35}{bw} \qquad \text{Eq. 8}$$

This means that the oscilloscope and probes, having a 600 MHz bandwidth, can display signals with rise-times no faster than 600 ps, without having more than 3 dB of attenuation.

Note: Various scope manufacturers use different conventions to specify bandwidth for their equipment; i.e., specified bandwidth is not necessarily where the displayed waveforms are 3 dB down in amplitude.

Scope Probes

Scope probes are available with many different characteristics. The three main types are referred to as passive high-impedance, active high-impedance, and passive low-impedance.

Passive high-impedance probes usually range from as low as 10-kΩ to 10-MΩ load impedance. This number identifies the loading effect of the probe when attached to a circuit. The best feature of high-impedance probes is that their impedance is usually much larger that those of the circuit under test and thus do not present any appreciable DC load to the measured signal when present.

Passive high-impedance probes do suffer one major drawback: significant capacitive loading. Most high-impedance probes present from 5 pF to 20 pF of capacitance at the probe tip. This capacitance affects measurements in two ways; it slows down the circuit being measured, and it degrades the rise-time of the probe. The upper bandwidth limit for passive high-impedance probes is around 400 MHz.

Active high-impedance probes combine a high bandwidth amplifier with the probe to improve the overall bandwidth of the system. These probes usually exhibit load impedances of 10 kΩ to 10 MΩ but have load capacitances of less than 3 pF. This type of probe has a typical upper bandwidth limit of around 1 GHz.

Care should be taken when using active probes as the manufacturers specified bandwidth may not be where the signal measured is 3 dB down. To achieve the higher bandwidths some active probes have non-linear responses to equalize the probe response. When presented with edge rates or frequency components beyond the specified probe bandwidth, the probe and scope may actually display a distorted waveform having more high-frequency components present than are actually in the measured signal.

Passive low-impedance (resistive divider) probes are used for the highest frequency work. These probes are available in load impedances from 50Ω to 5 kΩ and present load capacitances of 1 pF or less. A typical upper bandwidth limit for these probes is around 3 GHz. Unlike the high-impedance probes, low-impedance probes are designed to connect to a 50Ω transmission line system and do not require compensation. The probe itself is an extension of the 50Ω transmission line present in the

scope, and contains a precision resistive-divider at the probe tip.

The main drawback of passive low-impedance probes is the load impedance they present to the circuit. The rule of thumb for probes is that the probe impedance needs to be an order of magnitude greater than the impedances present around it to avoid any appreciable distortion. To get around this the probe is often designed as part of the system under test, such that its impedance is factored into the design. When the probe is not present it may be necessary to change component values or configurations to compensate for the absence of the probe (Reference 7).

Table 2 shows a summary of typical oscilloscope probe characteristics. For proper viewing of HOTLink ECL signals, an oscilloscope should have a minimum system bandwidth of 600 MHz. In most cases this will require use of low-impedance probes.

Table 2. Typical Probe Characteristics

Probe Type	Z	C_{load}	BW (MHz)
Passive High-Z	10k–10MΩ	5–20 pF	400
Active High-Z	10k–10MΩ	3 pF	1000
Passive Low-Z	50–5kΩ	1 pF	3000

Probe Grounding

As with any measurement, a good ground is mandatory. What is often misunderstood is just what *is* a good ground. At the frequencies used with HOTLink, a long looping ground lead is about as good as no ground at all. Three factors come into play: the reflections caused by the scope probe, and the ground inductance and parasitic capacitance limiting the probe's bandwidth. A simple rule of thumb for ground leads is that they exhibit about 1 nH of inductance for each millimeter of length. As the length of the probe's ground lead increases, the probe's resonance point decreases.

To view a signal with minimal distortion, the probe's resonant frequency must remain above the highest frequency signal component of interest. The graph in *Figure 7* shows how a scope probe's resonant frequency varies for different lengths of ground loop inductance and tip capacitance. This graph is based on *Equation 9* with the diagram of a low-impedance probe shown in *Figure 8*.

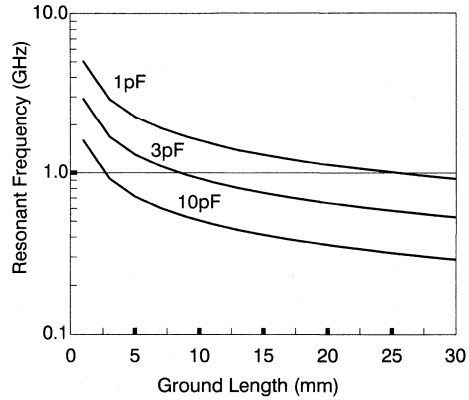


Figure 7. Scope Probe Resonant Frequency

$$\omega = 2\pi f = \frac{1}{\sqrt{LC}} \qquad \text{Eq. 9}$$

From this graph it is quite apparent that a ground lead of only 10 mm cuts the resonant frequency of the probe by 75%. For signal viewing at HOTLink serial data rates it is usually necessary to use coaxial scope-tip sockets soldered directly to a circuit board, or some other probe type that probes for signal and ground without a loose ground lead (Reference 8).

Probing From V_{CC}

The normal mode for probing ECL is to use V_{CC} as the ground reference. In this mode the signal being viewed is below ground and is relatively close to the ground reference. If the overall circuit design uses TTL parts in a mix with the negative referenced ECL, the TTL signals will all exist above ground. If

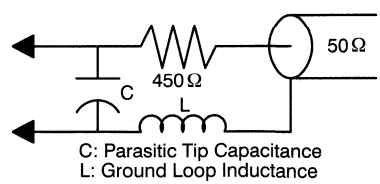


Figure 8. Scope Probe Tip Schematic

the ECL parts are operated in a PECL mode where they share a common V_{CC} supply with other TTL or CMOS parts, all probing should be done from TTL ground, which is the V_{EE} side of the ECL parts.

Probing From V_{EE}

When V_{EE} is used as the scope ground, other issues may come into play. In this mode the ECL signal is now positioned almost 4V above the reference point. While many scopes are able to perform a DC offset to make the ECL signal viewable, some do this at the expense of sensitivity. In other words, a signal that is viewable at 100 mV/div when offset less than 2V, may only be viewable at 500 mV/div when offset by 4V. Since the total signal swing for ECL signals is only 800 mV, it may be difficult to see a detailed representation of the waveform at this resolution.

Another problem with measuring from V_{EE} is that all the references in the ECL part are regulated from V_{CC}, not V_{EE}. This means that any amplitude changes or ripple in the power supply are now added into the displayed waveform.

One way around the offset problem is to AC couple the signal into the scope. Some scopes offer this as a front panel set-up selection, while others require the addition of a wide-bandwidth DC blocking capacitor in line with the scope probe. Either of these will remove all DC components from the signal under test, and allow the signal to be displayed at the maximum resolution of the oscilloscope.

Wide-bandwidth capacitors designed for this function are available from most test equipment manufacturers for use with existing probes and scope amplifiers. Some common capacitor types for SMA connector probes are the Tektronix 015-1-13-00 and Hewlett-Packard 11742A. For BNC connectored probes the Hewlett-Packard 10240B is also available.

Sample ECL Waveforms

ECL signals, when properly biased, terminated, and bypassed, are very clean and stable. Any noticeable overshoot on signals is usually caused by reflections from improperly terminated transmission lines or improper probing. *Figure 9* shows what a pristine single-ended ECL waveform should resemble when viewed on a scope.

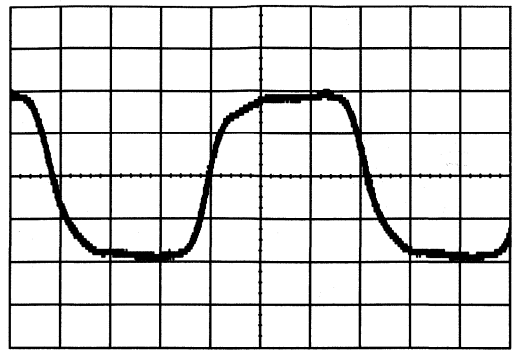


Figure 9. Good ECL Waveform, Single-Ended vs. V_{CC} Ground

Both the rising and falling edges are quite symmetrical and approximate an RC charge/discharge curve. The peak-to-peak range of the transition covers approximately 800 mV and is centered around V_{CC} − 1.3V. This signal was measured using a 500Ω, 1.5-GHz bandwidth low-impedance probe, on a scope having 1-GHz bandwidth. This signal was measured with V_{CC} as the probe ground. The probe load impedance (500Ω) was combined with other bias resistors to present a 50Ω to V_{CC} − 2V load on the signal.

With incorrect termination, a waveform such as that illustrated in *Figure 10* can result. Here the spike in the middle of a low area may cross the receiver V_{IH} threshold and cause the receiver to start to switch.

ECL Logic Families

Just as the TTL compatible world has its 7400, 74LS, 74H, 74S, 74AS, 74ALS, etc. logic families that have evolved over time, so does ECL. The most common families still in use are referred to as 10K (e.g., SL10104), 10KH (e.g., MC10H116), and 100K (e.g., F100150). These ECL families differ in terms of speed, signal levels, noise margins, and temperature and voltage stability.

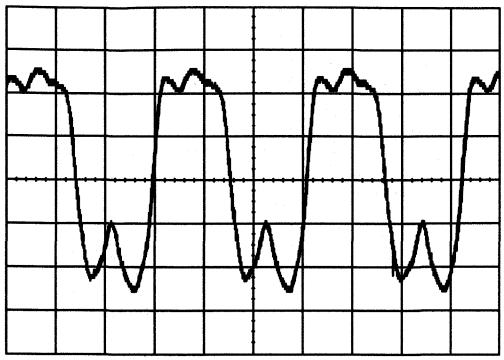


Figure 10. Bad ECL Waveform

10K ECL

The 10K ECL family has been around since 1971. It provides propagation delays of 2 ns with slow 3.5-ns edge rates (10%−90%). The voltage swings and switching thresholds of this logic family are relatively insensitive to variations in the power supply voltage but are affected by operating temperature (−30°C to +85°C). The V_{BB} bias network is fixed at V_{CC} − 1.29V, and is compensated for voltage and temperature. In the basic 10K ECL switch the current source is unregulated and consists of a single resistor between V_{EE} and the tied emitters of the differential amplifier. The transfer curves of a simple 10K gate are illustrated in *Figure 11* and detail how this family is sensitive to temperature variations in both inputs and outputs (Reference 19).

10KH ECL

To improve system speeds, the 10KH ECL family was introduced in 1981. It reduced propagation delays to 1 ns while edge rates were set to 1.8 ns. Because the thresholds and voltage swings remain the same in 10KH as in 10K, these two ECL families are fully compatible with each other. The temperature and voltage compensated V_{BB} reference network from 10K parts was replaced with a fully compensated and regulated supply. To improve the V_{OL} levels the resistor current source was replaced with a regulated current source. This allowed the collector resistors in the ECL switch to be matched and have similar switching characteristics. The transfer curves of a simple 10KH gate (see *Figure 12*) illustrate how this family improves noise margins over 10K ECL, yet remains sensitive to temperature variations. The 10KH family also is specified to operate over a narrower temperature range (0°C to 75°C) than 10K ECL (Reference 19).

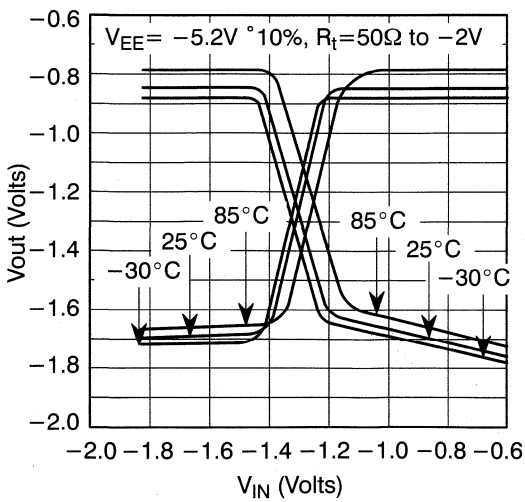


Figure 11. 10K ECL Transfer Functions

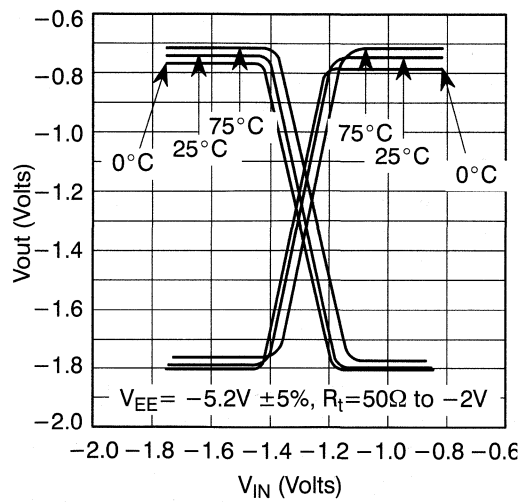


Figure 12. 10KH ECL Transfer Functions

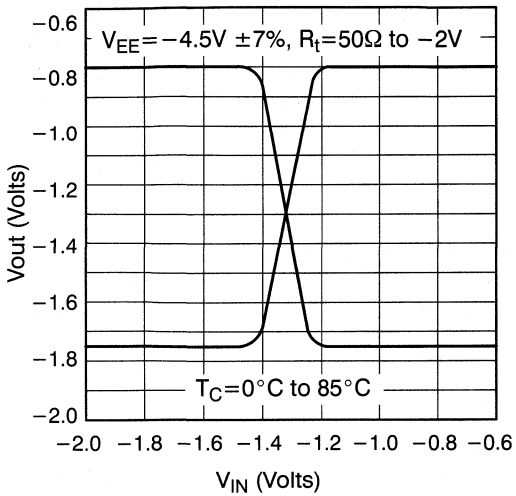


Figure 13. 100K ECL Transfer Functions

100K ECL

The 100K ECL family is a faster and easier to use ECL logic family. Introduced in 1973, this family improved on the internal structures to provide 750-ps propagation delays and 1-ns edge rates. In addition to speed improvements, the 100K ECL family was the first to introduce full compensation. This means that all the critical structures in the parts are now compensated for variations in voltage and temperature. This minimizes differences in propagation delays from one stage to the next that limit the maximum operating rate of a system. This stability is illustrated in the transfer curves in *Figure 13* (Reference 5).

In the 100K ECL family the operating temperature range is expanded to 0°C to 85°C but the nominal operating voltage changes from −5.2V to −4.5V.

HOTLink ECL Outputs

All ECL outputs of the HOTLink Transmitter are ECL 100K-level compatible. This means that these outputs meet or exceed all voltage, current, and edge rates specifications of 100K ECL and will interoperate with other 100K ECL parts. This signal level compatibility is required by the ANSI Fibre Channel standard (Reference 3).

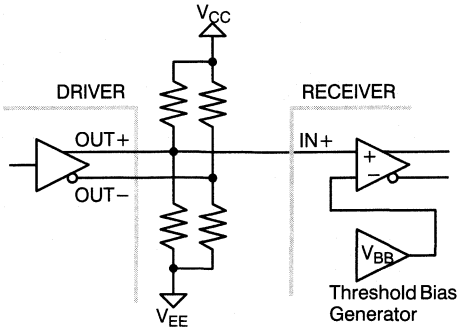


Figure 14. Single-Ended Connection

The HOTLink ECL outputs actually are substantially better than the 100K ECL specification, allowing operation with 5V±10% supplies over the full −55°C to +125°C temperature range. This allows the HOTLink parts to be used in a TTL, PECL, or ECL environment.

The HOTLink Transmitter has six ECL outputs configured as three differential pairs: OUTA±, OUTB±, and OUTC± (see *Figure 2*). These differential outputs may be used to communicate with ECL compatible receivers in either single-ended (strongly discouraged) or differential (preferred) modes.

HOTLink Transmitter Single-Ended Connections

A single-ended connection is used most often for logic functions. In this type of a connection, a single output of a driver is attached to a single input of a receiver. The receiving element is thus dependent on the driver and interconnect for maintaining the input signal in the narrow voltage bands specified for a valid logic 1 or 0.

Figure 14 illustrates the basic components of a single-ended connection. The driver differential pair outputs are biased to allow them to switch. The receiver, as with all ECL gates, is based on a differential amplifier. In the case of a single-ended receiver, the second input into the differential amplifier is not present at an external pin on the chip, but is instead connected internally to a V_{BB} reference voltage. As the signal present on IN+ goes either above or below the internal threshold set by V_{BB}, the receiver will switch.

While connections of this type are perfectly fine for logic functions, they should be avoided for a communications link. In a single-ended environment, any signal level differences (caused by temperature, logic family, transients, power supply noise, etc.) directly affect the received signal timing. In a logic function this timing variation limits a design both in determining how fast the system may operate, and in how much noise margin is present.

In a communications link these variations in timing translate directly into jitter in the serial data stream. Jitter affects a serial link by limiting not only how fast the link can operate (data rate) but also how far the data can be sent. Jitter is discussed in detail later in this document.

The only expected single-ended connection on a HOTLink Transmitter is for a local loopback function to a HOTLink Receiver (when the INB− input is not available for a differential connection because it has been used as an ECL-to-TTL translator). In this connection it is expected that the transmitter and receiver are in relatively close proximity, such that the connection between them is more on the order of a logic connection than a communications link. The small amount of jitter caused by the single-ended connection will be far below the jitter susceptibility of the HOTLink Receiver.

HOTLink Transmitter Differential Connections

A differential connection is the preferred attachment for HOTLink Transmitter serial outputs. In a differential connection both outputs a of a driver are connected to the true and complement inputs of an ECL-compatible receiver. When connected in this fashion the majority of the interconnect dependencies are removed. The main advantages of a differential connection are insensitivity to the logic family, operating temperature, and power supply variations. In addition, the connection is now immune to most common-mode noise.

Figure 15 illustrates the basic components of a differential connection. The driver differential pair outputs are biased to allow them to switch. Now both true and complement inputs of the the receiver differential amplifier are available at external pins

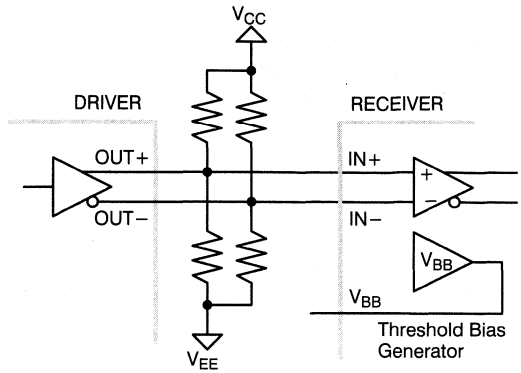


Figure 15. Differential Connection

and are connected to the complementary outputs of the driver.

Some ECL differential receivers may also provide an external V_{BB} reference. This reference is provided for those cases where a driver is connected single-ended to one of the differential receiver inputs. The other receiver input must then be connected to the V_{BB} reference to allow the receiver to switch. With a true differential connection this V_{BB} output should remain open.

The main concerns in a differential connection are signal skew and crosstalk. Skew is the difference in arrival time of the OUT+ and OUT− signals at the receiver. Crosstalk is the coupling of energy between these same two signals.

As the amount of signal skew present in a differential connection is increased, the effective signal rise and fall times at the differential receiver also increase. In systems with large amounts of signal skew, it is possible for short pulses to never be detected by the receiver.

The main cause for signal skew is asymmetric routing of the true and complement signals between the driver and the receiver. A 1-inch difference in routing length is equal to about 150 ps of signal skew. This problem is corrected by maintaining matched signal runs between the HOTLink Transmitter and the ECL differential receiver.

The main cause for crosstalk is long parallel signal runs. The adjacent lines act as coupling transform-

ers and transfer energy from one to another. One cure for this is to limit the length of the connection by placing the ECL differential receiver as close to the HOTLink Transmitter as possible. Other possibilities are to route the two signals on opposite sides of a circuit board with an interposed power plane to act as a shield. If routing is to remain on the same plane, the crosstalk effects can be minimized by horizontally separating the two signals as far as possible or by routing a ground trace (with many vias to attach the ground trace to the ground plane) between the two signals.

HOTLink ECL Inputs

The ECL inputs on the HOTLink Receiver are also ECL 100K-level compatible. Similar to the transmitter, these inputs have also been enhanced to operate over a wider range than standard 100K ECL.

The differential INA± and INB± inputs offer improved minimum sensitivity of 50 mV, compared to 150 mV for the few 100K ECL differential receivers available. These inputs may be connected directly to either power rail without damage to the part, or changing the internal thresholds of other sections of the receiver. These same differential inputs also operate with a 3V common-mode rejection range (V_{CC} down to V_{CC} – 3V) that is twice the 1.5V range of standard 100K ECL differential receivers (V_{CC} – 0.5V down to V_{CC} – 2V).

Note: While differential outputs are quite common on ECL parts, true differential inputs are rare. The most common usage for differential inputs is on line receivers and clock drivers. The common-mode range on some parts with differential inputs is quite limited and should not be expected to operate over even a narrow range unless explicitly stated in the manufactures datasheet.

The INA± inputs of the HOTLink Receiver should always be connected to a differential signal source. Since there is no V_{BB} reference output on the receiver there is no way to properly bias the second input of the differential receiver.

The INB± inputs may be configured to operate either as a differential receiver (in which case it should be connected to a differential signal source) or as two single-ended receivers. When operated as two single-ended receivers (as configured using the SO pin) the INB+ input operates as a 100K ECL single-ended receiver for serial data, while the INB–(SI) input operates as a 100K ECL single-ended receiver for an ECL-to-TTL level translator. The V_{BB} reference for these signals is available only inside the HOTLink Receiver and is not brought to an external pin. Signals connected to these single-ended inputs must now ensure operation within the 100K threshold levels.

Mixing ECL Logic Families

It is often desirable to use ECL parts of different families together in the same design. This can be done if certain rules are observed. The main reasons for these rules are the variability in signaling levels in ECL 10K family parts. *Figure 16* shows a DC-level comparison for 100K ECL outputs driving single-ended 10K ECL inputs.

In this configuration there is only 20 mV of margin between the 100K V_{OHL} and the 10K V_{IH} at the upper end of the temperature range. With 10K parts driving other 10K parts (assuming a common operating temperature) this is not a problem as the internal V_{BB} reference in each part follows a similar temperature shift. If the case temperature of the receiving 10K part can be kept below 35°C (100-mV margin), it can safely be used with 100K ECL parts for logic functions.

While the V_{OLH} specification appears to also have a noise margin problem, it does not. What occurs here is a condition where the receiver may be operated outside its linear region; i.e., 1s and 0s will be detected properly but the timing response may not match the manufacturer's data sheet.

Figure 17 shows the opposite configuration with 10K ECL logic driving either a single-ended 100K ECL receiver or a HOTLink Receiver. Here there are no tight margin areas between input and output thresholds. This means that 10K ECL parts can safely be used to drive 100K ECL inputs over their full temperature range.

Figure 17 also highlights the enhanced input range for the HOTLink Receiver. Unlike the narrow input range present on standard ECL families, the

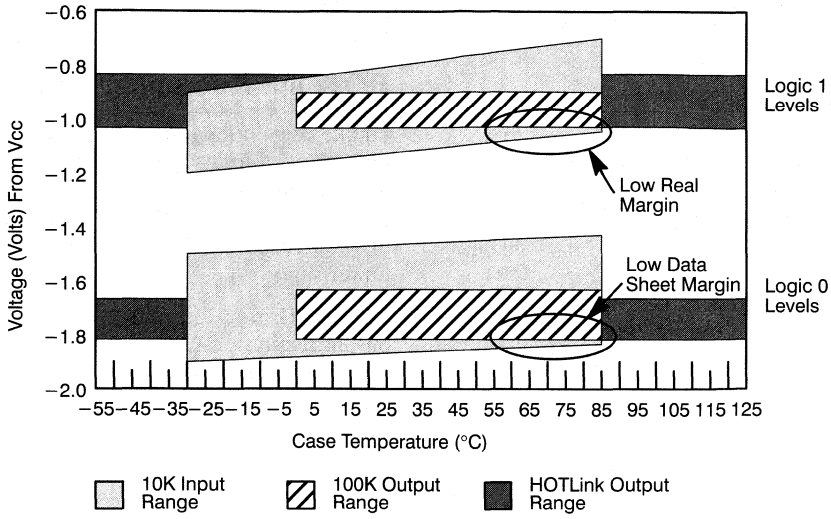


Figure 16. 100K ECL Driving 10K ECL

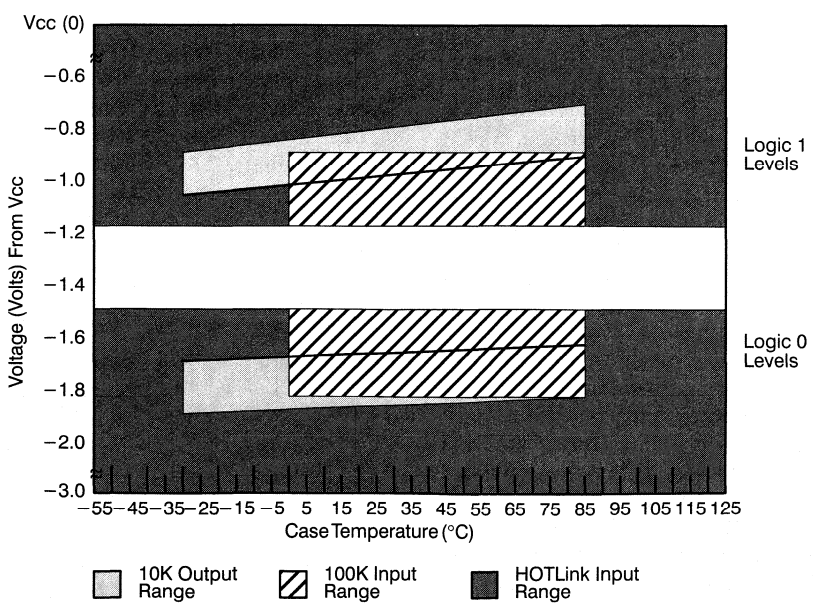


Figure 17. 10K ECL Driving 100K ECL

ECL inputs on the HOTLink Receiver maintain normal operation over the entire V_{CC} to V_{CC} – 3V range.

Single-Ended Connections

Both of these comparisons are based on single-ended connections, where only a single ECL output is used to drive the receiving internally referenced single-ended gate. In these cases, the other input to the receiving differential amplifier is connected internally to a V_{BB} reference. This type of connection should not be used to drive the INA± or INB± differential inputs of the HOTLink Receiver.

Differential Connections

One of the biggest advantages of ECL is the ability to communicate in a differential mode. This mode is relatively rare on logic parts (most commonly used for clock drivers and line receivers), as it requires both the driving and receiving parts to have both true and complement outputs and inputs respectively. When connected in this manner, the receiving part is no longer comparing the input signal to its V_{BB} reference, but instead compares the true and complement inputs to each other.

When used in this mode there is no problem using 100K ECL with 10K ECL at any temperature. Because an ECL receiver only requires around 250 mV of *difference* to fully switch, and the difference between the outputs of a differential driver remains near 800 mV, any differential connection has a minimum of twice the noise margin of a single-ended connection.

This type of connection is also immune to minor differences in the references voltages between parts. Because the connection is differential, any common-mode voltages present on the received signals (due to power supply differences, AC coupling, ground shift, etc.) within the common-mode range are canceled out in the receiving differential amplifier. Some ECL parts with differential inputs can accept up to 1V of common-mode offset on the received signal without degradation of performance. The enhanced 100K ECL compatible inputs of the HOTLink receiver can accept inputs between V_{CC} and V_{CC} – 3V, offering a common-mode range of 3V.

HOTLink Transmitter Connections

Unlike conventional negative-referenced ECL, the high-speed outputs on the HOTLink Transmitter are implemented in 100K Positive-referenced ECL (PECL). This allows the TTL and ECL interfaces on the transmitter to operate from a common +5V power supply.

The HOTLink Transmitter has three differential output sections: OUTA±, OUTB±, and OUTC±. In addition to operating as 100K ECL-compatible signals, these outputs have been enhanced with additional features.

Power Saving Mode

A standard ECL output structure uses a constant current source at the base of a differential amplifier (see *Figure 5*). In these standard parts, this current source is enabled and dissipating power even when the outputs are not used.

The HOTLink Transmitter ECL outputs, while still operating as true 100K ECL outputs, incorporate some additional structures (see *Figure 18*) to save power when the outputs are not used. The differential amplifier (D1) under normal conditions will direct the I_S current from the current source through its internal transistors. As this current is switched, the output driver transistors (Q1 and Q2) change their operation point and the amount of current

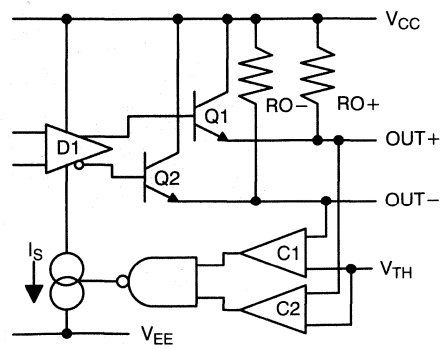


Figure 18. HOTLink Transmitter ECL Output

they source (a properly biased ECL output sources current in both 1 and 0 states; i.e., it does not turn off). Each output driver (Q1 and Q2) contains a high value pull-up resistor (RO+ and RO−) and a voltage comparator (C1 and C2). When both voltage comparators of a HOTLink differential output detect a voltage above a 100K ECL output-high level (V_{TH}), the current source (I_S) for that differential output pair is disabled. This results in a current savings of around 5 mA (25 mW) for each unused output pair.

FOTO Control of OUTA± and OUTB±

The HOTLink Transmitter OUTA± and OUTB± differential outputs have an additional control input not present in the OUTC± output pair. While the OUTC± outputs are always enabled to follow the serial data stream generated in the HOTLink Transmitter shifter, the OUTA± and OUTB± outputs are not. These outputs are also controlled by a TTL-level input called FOTO (FiberOptic Transmitter-Off). While OUTA± and OUTB± are disabled, the OUTC± pair remains active and can be used for a local loopback source.

This FOTO signal is used to force the differential outputs of the OUTA± and OUTB± drivers to a state where a logical 0 is being driven. This state corresponds to a condition on optical modules where no light is transmitted. While not required for LED-based optical modules, this capability is required for laser-based links (see ANSI Z136.1 and Z136.2, F.D.A regulation 21 CFR subchapter J, and IEC 825) (References 9, 10, 11, 12, 13).

ECL Output Biasing

ECL outputs have specific loading requirements to insure proper operation. Because of the open-emitter structure of an ECL output, it can source current but cannot sink current. To allow the output to switch, some form of pull-down is required on the output. This pull-down usually takes the form of a resistive load; either to V_{EE} or $V_{CC} - 2V$.

Most ECL outputs are specified for driving load impedances as low as 50Ω. Because an ECL output does not swing rail-to-rail, this load is usually specified at $V_{CC} - 2V$, a point slightly below the ECL V_{OL}. At this point, when the ECL gate is driving a logic-0 level signal, a small current is running through the load resistor to keep the output transistor in the active region. Typical currents sourced when driving a logic-1 (I_{OH}) and logic-0 (I_{OL}) are calculated using *Equations 10* and *11* respectively, where R_T is the effective load impedance and V_{TT} is the effective bias voltage.

$$I_{OH} = \frac{V_{OH} - V_{TT}}{R_t} = \frac{(-0.9) - (-2.0)}{50\Omega} = 22mA \quad \text{Eq. 10}$$

$$I_{OL} = \frac{V_{OL} - V_{TT}}{R_t} = \frac{(-1.7) - (-2.0)}{50\Omega} = 6mA \quad \text{Eq. 11}$$

These I_{OH} and I_{OL} values are the basis for the timing and signal levels in the HOTLink datasheet. For other values of I_{OH} and I_{OL}, the transmitter will exhibit slightly different characteristics. These current flows can be achieved in many ways. The four most common methods are

- Shunt bias to V_{TT} bias voltage
- Shunt bias to V_{EE} bias voltage
- Thevenin bias to V_{TT} bias voltage
- Y-bias to V_{TT} bias voltage

Shunt Bias to V_{TT}

In shunt bias, as illustrated in *Figure 19*, a single resistor is used as a pull-down load on an ECL output to some bias voltage. When biased to V_{TT}, a single 50Ω resistor (R_T) from the ECL output to V_{TT} is all that is necessary. This requires an additional power supply to provide the ($V_{CC} - 2V$) V_{TT}-level. This termination type dissipates the least average-power (13 mW) of any output load type. It is often used in large ECL systems, in systems where overall power dissipation is a major concern, or where there is enough ECL present to warrant its design and implementation.

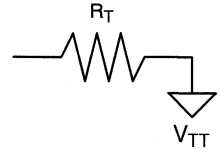


Figure 19. Shunt Bias to V_{TT}

Shunt Bias to V_{EE}

ECL outputs may also be biased to the V_{EE} supply as illustrated in *Figure 20*. Here a load resistance (R_T) of near 270Ω is connected to the V_{EE} supply to provide a similar current load for the ECL output driver. This value is determined by taking the average current flow for both a 1 and a 0 at the midway point (V_{BB}) in the output swing. The calculation for this is shown in *Equation 12*.

$$R = \frac{V_{EE} - V_{BB}}{\frac{I_H + I_L}{2}} = \frac{5 - 1.3}{\frac{22+6}{2}} = 264\Omega \qquad \text{Eq. 12}$$

Unlike the shunt bias to V_{TT}, this bias arrangement dissipates a significant amount of power in both the 1 and 0 states (47 mW average). This bias type (due to mismatched RC charge and discharge rates) exhibits a faster falling edge than rising edge. Because of this, its use is usually limited to logic functions, and is discouraged for serial links and for biasing differential output pairs. This is discussed in detail later in this document.

Thevenin Bias to V_{TT}

In a Thevenin bias network, a pair of resistors (R1 and R2) is used to create a load whose Thevenin equivalent matches that of a single resistor attached to a specific bias voltage (V_{TT}). For ECL this voltage is usually V_{CC} – 2V. These resistors are connected as illustrated in *Figure 21*. The values of R1 and R2 are solved using *Equations 13* and *14*.

$$R1 = \frac{V_{EE} \times R_T}{V_{EE} - V_{TT}} \qquad \text{Eq. 13}$$

$$R2 = \frac{V_{EE} \times R_T}{V_{TT}} \qquad \text{Eq. 14}$$

Solving for 50Ω and V_{CC} – 2V yields values of 82Ω and 120Ω for a 5V system. While this combination does provide a similar dynamic load to the shunt bias to V_{TT}, it dissipates nearly an order of magnitude more power (138 mW) than its shunt to V_{TT} equivalent.

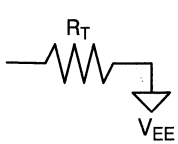


Figure 20. Shunt Bias to V_{EE}

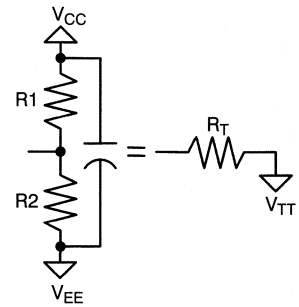


Figure 21. Thevenin Bias Equivalent

The capacitor shown in *Figure 21* is needed to allow R1 and R2 to provide the proper load for AC signals. In a Thevenin equivalent circuit, the power supply is assumed to be a short circuit. While this may be accurate for DC or very low frequency AC signals, the power supply appears as a near infinite impedance at RF frequencies. The bypass capacitor across R1 and R2 is used to create an AC short. This capacitor must be sized to operate as a short near the frequencies in use. For HOTLink-based systems this capacitor should probably be in the range of 300 pF to 0.01 µF. For best results two capacitors should be placed in parallel to provide a passband for the desired frequencies.

Y-Bias to V_{TT}

Unlike the three previously described terminations, the, Y-bias can only be used with differential outputs. In this configuration the active ECL output (logic 1) is used to source current for a voltage divider, while the inactive ECL output (logic 0) is pulled down to the bias voltage created by this divider. A schematic of this bias network is illustrated in *Figure 22*.

Here R_T is the desired load impedance, usually 50Ω to V_{CC} – 2V for ECL systems. R_L is determined by summing the currents of a logic 1 and a logic 0 (as shown in *Equations 10* and *11*), and calculating the resistance necessary to drop the remaining voltage. This calculation is shown in *Equation 15* and solved for a 50Ω R_T.

$$R_L = \frac{V_{CC} - V_{EE} + V_{TT}}{I_{HI} + I_{LOW}} = \frac{3V}{28mA} = 107\Omega \qquad \text{Eq. 15}$$

This type of bias provides a significant power savings over a Thevenin bias because only a single pull-down resistor is used to dissipate power for two outputs. For a 50Ω equivalent load the power dissipation is only 110 mW for two outputs (55 mW for one). Just as with the Thevenin bias, a capacitor is necessary to create an AC short.

Matched Loading

Just as the differential amplifier in an ECL switch directs current flow, so do the emitter-follower output transistors. As these transistors are turned on an off, large amounts of current are switched through the driver's V_{CC} package pins. Because of the inductance present in these pins, transients can be induced in the internal V_{CC} supply.

Fortunately the effects of this lead-inductance only manifest themselves when the current through the V_{CC} supply pin *changes*. If the current is kept stable, no transients are induced. Due to the differential configuration of many ECL outputs, it is possible to keep this current stable by having matched loads on the true and complement outputs of the differential driver. This means that if a design uses one or both outputs of a differential driver, they both should drive loads of the same magnitude.

Figure 23 shows a differential output driver connected to a load including the package inductance present on the V_{CC} power pin. As the differential driver changes state, the overall current through L1

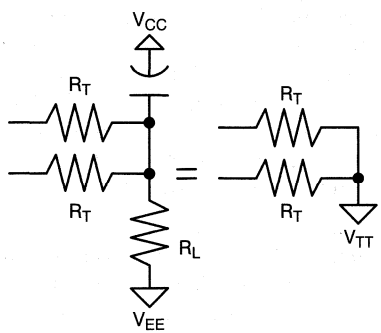


Figure 22. Y-Bias Network

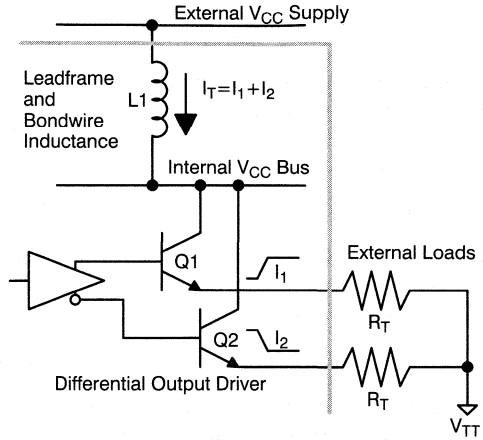


Figure 23. Loaded Differential Driver

remains the same (assuming that both R_T loads are the same value).

If one of the two R_T load resistors is removed, some very undesirable things start to happen. The first is that the external power supply must now react to a dynamic rather than a static need for current. This increases the amount of power-supply bypassing that is needed next to the ECL driver V_{CC} pin. The second is a variation in the internal and external V_{CC} supplies caused by the dynamic current flow. This effect is illustrated in the following approximation.

For a single ECL output the current difference from a logic 1 to a logic 0 (into a 50Ω to V_{CC} – 2V load) is 16 mA (see *Equations 10* and *11*). From the ECL 100K family datasheets we know that signal transition times may be under 500 ps. By assuming the rise and fall portions of the signal are related to a triangular waveform, this transition may be roughly converted to a fundamental frequency using *Equation 16*.

$$F = \frac{1}{2 \times T_r} = \frac{1}{2 \times 500E^{-12}} = 1GHz \qquad \text{Eq. 16}$$

The Fourier series for a triangular waveform is listed in *Equation 17*. This illustrates that most of the energy content is present at the fundamental frequency with much smaller components present at the higher odd harmonics. To simplify the following calculations only the fundamental frequency is assumed to be present (Reference 24).

$$\frac{8V}{\pi^2}(\cos\omega_o t + \frac{1}{9}\cos 3\omega_o t + \frac{1}{25}\cos\omega_o t + ...) \quad \text{Eq. 17}$$

If a package pin inductance of 4 nH is assumed (typical for many surface mount components), *Equation 18* can be used to determine the impedance of the package at this frequency.

$$X_L = 2\pi FL = 2\pi \times 1E^9 \times 4E^{-9} = 25\Omega \quad \text{Eq. 18}$$

Using Ohm's Law we can now convert this change in current into an internal voltage change, as illustrated in *Equation 19*.

$$V = I \times X_L = 16mA \times 25\Omega = 400mV \quad \text{Eq. 19}$$

This temporary difference between the internal V_{CC} and the external V_{CC} supply is the same phenomenon known in a TTL environment as ground bounce.

All of this, of course, is based on the assumption that the output will be able to switch at this speed (500 ps) and provide the specified current (16 mA) when presented with a high-impedance source. What actually occurs is that the output edge slows down to match the current transfer permitted by the on-resistance of the output driver transistor and the package reactance.

Most ECL parts use a couple of different techniques to combat this problem. Both are quite simple to implement. The first is to use a separate package pin to provide power to the emitter-follower output transistors. This prevents any V_{CC} shift caused by the output drivers from affecting the sensitive differential amplifiers and voltage references present in other parts of the device.

The second method is to maintain a balanced load on the differential output drivers. Since the rising and falling edge rates of ECL are very symmetrical, $\Delta I_1 = \Delta I_2$. Because these changes in output current are symmetrical, $\Delta I_T \cong 0$. From *Equation 19* we know that any induced ΔV is directly proportional to ΔI; thus as ΔI goes to 0, so does ΔV.

AC Characteristics of Output Drivers

In an ECL driver, the time it takes for the signal to rise is largely determined by its internal resistors and parasitic capacitors (C_{int} and R_{int} in *Figure 24*),

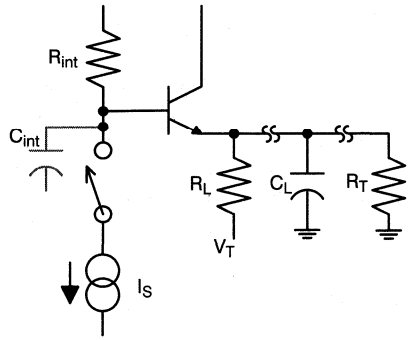


Figure 24. ECL Output Driver With Loading

since the emitter-follower can supply large currents to charge the load capacitance. The DC voltage to which the output rises is determined by the emitter-follower transistor characteristics and the internal driver resistor (R_{int}) value. However, the AC voltage (overshoot, ringing, etc.) is determined primarily by the load characteristics. A capacitive load (along with the inductance found in the package, printed circuit traces, and other load components) causes the output to rise significantly beyond the anticipated DC output level, since the emitter-follower cannot supply any compensating current at the top of its transition.

Unlike the output rise time, the fall time is primarily determined by the time constants of the load capacitance and pull-down circuit. The output low voltage (V_{OL}), is determined by R_{int}, I_S, and the characteristics of the emitter-follower transistor. In a properly designed system the load circuit has time constants comparable to (or shorter than) the internal fall time, such that the emitter-follower can source a small amount of current during the entire time it is switching from HIGH to LOW. If this is not true, the emitter-follower transistor will be shut off for part of the transition time, and the output will follow the time constant of the load.

Figure 25 illustrates the effects of two different load or bias circuits. The assumption in both of these examples is that the load circuit controls the fall time of the signal, and that the pull-down current is being supplied by a resistor to a V_T of either V_{CC} – 2V or V_{EE} (+3V or ground for a PECL environment). In

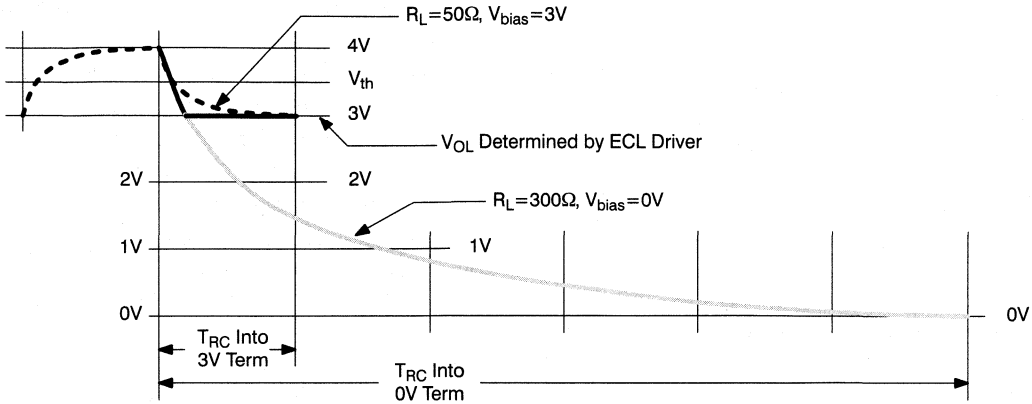


Figure 25. Falling Edge Rate Comparison For Bias to V_{TT} and V_{EE}

the dashed curve, the standard ECL load of 50Ω to V_{CC} – 2V is used, causing an output current of approximately 20 mA when the output is HIGH, and 5 mA when the output is LOW. This load (or its equivalent) can be created using all of the previously described bias networks except shunt bias to V_{EE} (shown in the solid curve).

The same amount of pull-down current can be realized with a single resistor (R_L in *Figure 24*) in a shunt bias to V_{EE} configuration. To get a comparable output current (and assure comparable voltages at the output) the pull-down resistor would be chosen to sink approximately the average of I_{OH} and I_{OL} when connected to a voltage midway between V_{OH} and V_{OL} (see *Equation 12*). The I_{OH} and I_{OL} currents listed here thus would yield a pull-down resistor of around 300Ω. This type of bias is perfectly correct and adequate for ECL logic circuits where the mismatch between rise and fall times is absorbed into the normal logic delays and set-up times. In a data transmission system the effects of this type of output bias can be unpredictable and will often degrade performance.

Figures 25 and *26* illustrate the difference in output fall time assuming a constant load capacitance, with the only variation being the bias resistor and voltage. The 50Ω load resistor (dashed line) follows an RC discharge curve which ends at V_{CC} – 2V. For normal loading this *soft* edge rate more closely matches the rise time of the output as controlled by the emitter-follower, and is less affected by variations in load capacitance and reflection currents.

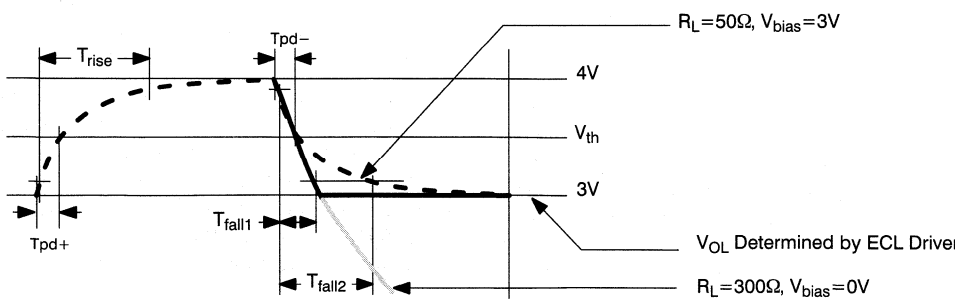


Figure 26. Expanded Detail of Falling Edge Rate Comparison

The 300Ω load resistor (solid line) follows an RC discharge curve which would normally end at V_{EE} (ground). While this appears to have a crisper edge rate, it will be more severely affected by load capacitance variation and transmission line reflection currents that must be accommodated.

Figure 26 shows that with either pull-down the total voltage swing is the same and is determined by the internal voltage swing of the driver, as buffered by the emitter-follower transistor. While the RC curve for the 300Ω pull-down continues to V_{EE}, the emitter-follower is turned on and sourcing current at the V_{OL} point and does not allow the output to continue farther down the curve.

In either configuration the signal delays match, since both falling edges cross the mid-swing line at the same time, but the rise and fall times are different. These rise and fall times determine the higher frequency spectral components of the waveform. Differences in these spectral components affect the termination efficiency and waveform distortion caused by cable attenuation (Reference 14).

Transmission Line Termination

While often confused with ECL output biasing, termination of transmission lines is something quite different. Because of the impedance characteristics of transmission line termination, the resistors used for termination may often be used as part of the output bias network, but they perform different functions.

Due to the high switching speeds of ECL, most of the interconnect between parts cannot be treated as simple connections. They must instead be treated as transmission lines. The distance between parts, in conjunction with the signal loading and rise and fall times, is used to determine at what point the interconnect must be treated as a transmission line. The general assumption is that short lines do not require termination, while long ones do. The determination of what is a long line is made using *Equation 20* (Reference 5).

$$\ell \max = \frac{1}{2}\sqrt{\left(\frac{C_L}{C_o}\right)^2 + \left(\frac{Tr}{\delta}\right)^2} - \frac{C_L}{2C_o} \qquad \text{Eq. 20}$$

The values for this equation for microstrip construction on G10/FR4 type board would be

- l max – maximum unterminated line length
- Tr – source 20% to 80% rise time
- C_L – load capacitance (2 pF assumed for a load)
- δ – delay per unit length (0.148-ns/inch)
- C_O – capacitance per inch

Running this calculation for various impedance and rise-time combinations yields the lengths listed in *Table 3.* Lengths beyond those listed here require termination.

Table 3. 100k ECL Maximum Unterminated Line Length (in inches), Microstrip Construction

Z_O	Line Length (in inches)		
	0.5 ns	1 ns	1.5 ns
50Ω	1.38	3.06	4.74
62Ω	1.32	2.99	4.67
75Ω	1.25	2.91	4.59
90Ω	1.18	2.82	4.50
100Ω	1.14	2.76	4.44

The lengths listed in *Table 3* assume digital switching characteristics. The HOTLink ECL serial signals are, for the most part, analog in nature. This effectively shortens the maximum unterminated length. For HOTLink serial signals, any ECL trace greater than one inch in length should be terminated.

The objective of transmission line termination is to prevent reflection of power from the the destination back to the source. This is accomplished by terminating a transmission line in its characteristic impedance (Z_O). The two basic types of line termination are referred to as series and parallel termination.

The actual amount of the source signal reflected is based on how well the line impedance matches the destination impedance. This determines how much voltage is reflected back into the transmission line. This ratio of reflected voltage to incident voltage is called the reflection coefficient ϱ (rho) and is shown in *Equation 21* (Reference 5).

$$\frac{V_r}{V_i} = \varrho = \frac{R_T - Z_O}{R_T + Z_O} \qquad \text{Eq. 21}$$

Series Termination

Series termination (sometimes referred to as source termination) requires that the load be high-impedance to properly operate. This type of line termination is **not** recommended for use with HOTLink because of the reactive nature of all parts at the high frequencies present on the HOTLink ECL signals.

Parallel Termination

In parallel termination the desired characteristic is to terminate the *end* of the line (rather than the source) in its characteristic impedance. This results in a reflection coefficient of zero; i.e., no signal is reflected. This type of termination is implemented the same as shunt bias networks. *Figures 27* and *28* show the two equivalent forms of parallel termination.

Parallel termination offers the advantages of allowing distributed loads on the transmission line, and of having the termination network also operate as the bias network.

In the single-resistor form of parallel termination illustrated in *Figure 27*, the R_T resistor is sized to match the Z_O impedance of the transmission line. This termination form has the same advantage as the single resistor shunt bias because it dissipates less overall power than the Thevenin equivalent termination. It also has the same drawback of requiring a separate power supply.

In a Thevenin equivalent termination (illustrated in *Figure 28*) two resistors (R1 and R2) are used to form an equivalent circuit to that in *Figure 27*. *Table 4* lists the R1 and R2 resistor values for a number of common transmission line impedances. This table assumes operation with a 5V source and a termination voltage of $V_{CC} - 2V$, and selects the nearest standard 1% resistor value when an exact match is not available. These values are calculated using the same *Equations 13* and *14* as used for calculating a Thevenin bias network (Reference 15).

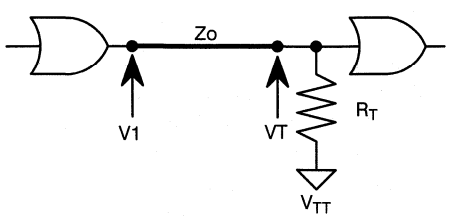


Figure 27. Parallel Termination to V_{TT}

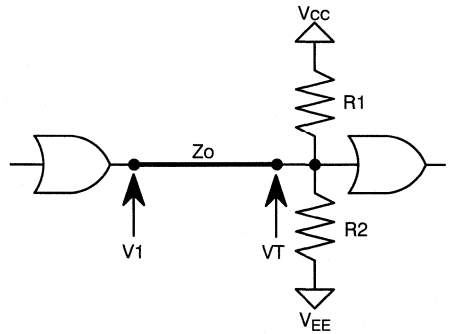


Figure 28. Thevenin Equivalent Parallel Termination

Table 4. Thevenin Bias Resistor Values

Z_O	R1	R2
50Ω	82.5	124
70Ω	118	174
75Ω	124	187
80Ω	133	200
90Ω	150	226
100Ω	165	249
120Ω	200	301
150Ω	249	374

Terminating HOTLink Transmitter ECL Signals

The HOTLink CY7B923 transmitter has three different ECL differential output pairs named OUTA±, OUTB± and OUTC± (see *Figure 2*). How (or if) these outputs are terminated is dependent on what the output is used for.

OUTC±

The OUTC± outputs of the HOTLink transmitter are not controlled by the transmitter FOTO signal and are thus always enabled to drive serial data.

While fully capable of driving either optical modules or copper cables, it is expected that the most common usage of this differential output will be as a local loopback to a HOTLink CY7B933 Receiver INB± inputs.

This signal may be connected to the HOTLink Receiver either differentially or single-ended. When connected differentially, the OUTC+ output is connected to the INB+ input, and the OUTC− output is connected to the INB− input. When connected single-ended, the OUTC+output is connected to the INB+ input.

Note: For the INB+ input to be used differentially, the SI/SO ECL-to-TTL translator (mapped through the INB− input) must be disabled. This is done by connecting the SO output directly to V_{CC}.

Once the connection is made, the type of termination required is determined by the distance between the HOTLink Transmitter and the HOTLink Receiver. If the distance is kept short enough (under one inch) (Reference 5) no termination is required and the output only needs to be biased. This can be done with a single pull-down resistor to V_{EE}. While this type of termination does induce some jitter into the serial data stream (due to mismatched rise and fall times), the amount is well within the receiver limits.

If the distance is greater than one inch, the line should be terminated (Reference 5). To do this correctly requires determination of the characteristic impedance of the board traces used to connect the source and destination. Please see the Cypress Semiconductor application note "System Design Considerations When Using Cypress CMOS Circuits" for information on how to determine the characteristic impedance of various types of transmission lines (Reference 16).

For local connections that do not travel through external transmission media (i.e., coax, twisted-pair, optical fiber, etc.) parallel termination may be used. The important consideration here is that both the OUTC+ and OUTC− outputs must be terminated/biased into the same type of load to maintain a current balance inside the HOTLink Transmitter.

If neither of the OUTC± outputs are used, both outputs should be left open or pulled up to V_{CC} to disable the current source for the differential driver (see *Figure 18*).

OUTA± and OUTB±

The OUTA± and OUTB± outputs of the HOTLink Transmitter are both controlled by the FOTO signal which is required to meet laser safety regulations for communications links (References 9, 10, 11, 12, 13). Other than this special enable signal, these outputs operate the same as the OUTC± outputs.

Driving Optical Modules

When connecting to optical modules, it is best to drive the optical module data inputs differentially. This provides the highest noise immunity for the system, and the lowest signal jitter. When used with *de facto* standard optical modules this becomes mandatory because the optical modules have a differential data input, yet do not provide a V_{BB} supply to bias the other input of the differential amplifier of the optical transmitter. Because this interface is intended for driving some external segment of optical cable, series termination (which uses shunt bias to V_{EE} and increases jitter) should not be used. Since the HOTLink parts will most probably be the only ECL parts in the system, the recommended termination is a Thevenin or Y-termination.

Both the Thevenin and Y-terminations provide the bias necessary for the ECL signal to switch, and the impedance necessary to terminate a transmission line. One of these types of termination/bias should be used even when the distance from the HOTLink Transmitter to the optical transmitter is short. This is necessary to maintain symmetrical rise and fall times for the OUTx± differential outputs.

PECL Optical Modules

Interfacing to optical modules in PECL mode is quite simple, requiring only a few passive parts. The schematic in *Figure 29* illustrates the connections and parts necessary for this type of connection.

One of the key items often missed in this type of connection is proper bypassing of the termination/bias networks. The theory behind a Thevenin network is

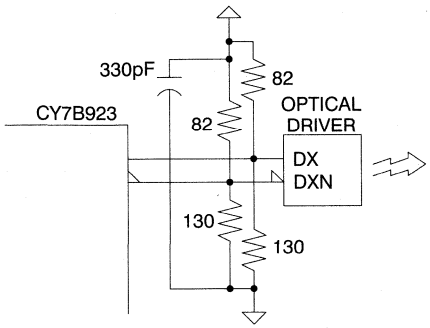


Figure 29. HOTLink Transmitter-to-PECL Optical Module

that the power supply is considered as a short for AC. While this may be true for near DC applications, the base frequencies and harmonics present in the HOTLink Transmitter output are far beyond any frequency the power supply itself could pass.

To make the power supply a short, a capacitor must be placed across the Thevenin pair. The size of the capacitor is determined by the frequency of operation of the serial link. A good rule-of-thumb is to pick the largest value capacitor whose series resonant frequency is 30% above the highest baseband frequency of the baud rate of the serial data (Reference 17). Since the data is sent using an NRZ modulation (Non-Return-to-Zero), the highest baseband frequency is one half the serial bit-rate (Reference 18).

Another important characteristic is the dielectric type of the capacitor. For this type of analog operation, a good high frequency RF type capacitor must be specified. This means specifying either NP0 or C0G type capacitors.

Standard ECL Optical Modules

Those optical modules with the case connected to V_{CC} are designed for use in a negative DC supply system. These types of modules may also be driven by a HOTLink Transmitter.

By far the simplest method is to connect the module the same as a PECL module, with the exception of the Case pins. Here, instead of attaching the Case pins to ground (V_{EE}), they are attached to V_{CC}. If the case is metallic in nature, care must then be exercised such that it does not come into direct contact with ground.

If the optical module is to be used below ground, it must be AC coupled to the HOTLink Transmitter. This type of connection is illustrated in *Figure 30*.

The HOTLink Transmitter outputs are biased the same as for a PECL optical module. AC coupling capacitors are used to connect the HOTLink Transmitter positive referenced ECL outputs to the negative referenced ECL inputs of the optical module. These coupling capacitors actually operate as a bandpass filter, centered around their series resonant frequency. To pass additional low- or high-frequency components, additional capacitors should be placed in parallel with the coupling capacitors.

Capacitively coupled signals require DC restoration and, if the connection length warrants, transmission line termination. DC restoration is necessary to place the signal swings in the input range of the ECL receiver. Unlike ECL outputs, which are biased to a level slightly below their V_{OL}(min)-level (V_{CC} – 2V), AC coupled ECL inputs need to be biased to the center of the receiver input range. This is the same as the V_{BB} reference point of V_{CC} – 1.3V. In *Figure 30*, this reference point is created from a resistive divider network, and bypassed with a 0.01-µF capacitor to provide the dynamic current response needed for the differential inputs.

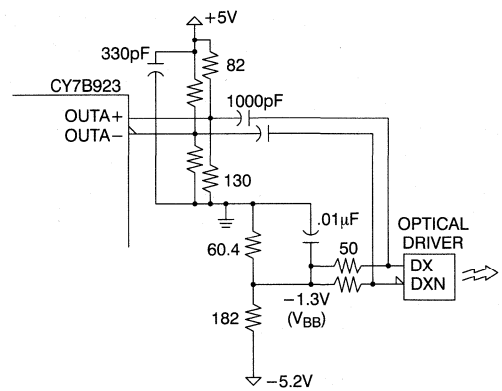


Figure 30. HOTLink Transmitter-to-Negative Referenced ECL Optical Module

While many optical modules or ECL gates generate a V_{BB}-level, this output *must not* be used to bias this reference point because it cannot provide sufficient dynamic current. The V_{BB} output of an optical module, or other ECL gate, is an unbuffered tap of the internal V_{BB} reference. While fully capable of delivering the few µA of current necessary to drive an input, it cannot tolerate the transient currents present at the end of a low-impedance transmission line. Because the V_{BB} source is unbuffered, this also means that any external transients applied to it will move the V_{BB} reference *inside* the receiver, with unpredictable consequences.

While it is possible to create a V_{BB} power amplifier (by using multiple ECL buffers in parallel) to create a buffered form of V_{BB}, such amplifiers should not be used with HOTLink. They are prone to oscillation and ringing. Such amplifiers should also not be used for DC restoration (as needed here) because the V_{BB} amplifier is not quite DC stable; i.e. its output usually contains a low-level (10–50 mV) oscillation whose frequency is set by the delay through the part. This low-level noise is not a problem for logic applications, but for analog applications causes increased jitter on the biased signals.

In this example, the V_{EE} for the optical module is set to −5.2V. This is a common supply voltage for ECL circuits. If a different supply voltage is used, the values in the resistive divider must be changed to maintain the V_{BB} reference point at V_{CC} − 1.3V.

One drawback of this circuit is the inability to react to a DC state in the data stream. If the HOTLink Transmitter is set to transmit all 1s or all 0s (e.g., FOTO is set to disable transmitting), the optical module inputs will both return to a V_{BB}-level. At this level the optical module's output will probably oscillate due to the high gain present in the optical module's ECL-to-optical translator. In this AC coupled configuration (when operated with laser-based optical drivers) it is necessary to use some method other than FOTO to meet the laser safety restrictions (References 9, 10, 11, 12, 13).CC

Driving Copper Media

The ANSI Fibre Channel Standard currently identifies both coaxial cable and shielded twisted pair as copper media types that are supported. The HOTLink Transmitter easily interfaces to these and many other types of copper media, and allows communicating on them at distances well beyond the lengths called out in the ANSI Standard (Reference 3).

Numerous characteristics determine how far a signal can be transmitted on copper media. The most important of these are:

- Voltage amplitude of the signal fed into the cable
- Jitter and ringing on the source signal
- The attenuation characteristics of the cable
- The length of the cable
- What (if any) equalization is used in the system
- Receiver loading and sensitivity

Coupling to the cable (transmission line if on a backplane) may be done in multiple ways, depending on the media type and distances involved.

Direct Coupled

For those instances where the signal never leaves the same chassis (or even the same board) it is possible to directly couple to the media. Here the media is effectively the circuit board traces, runs of twisted-pair, twinax, or dual coax. The main criteria here is that there must be no chance for a significant V_{CC} reference difference (transient or DC) between the HOTLink Transmitter and HOTLink Receiver, including any common-mode induced noise. For the HOTLink Receiver, this maximum difference is around 1V. Under these conditions the HOTLink Transmitter and Receiver may be connected as illustrated in *Figure 31*.

While *Figure 31* shows a 50Ω transmission line, the actual impedance can be higher than this. For other impedance values it is necessary to change the Thevenin termination networks.

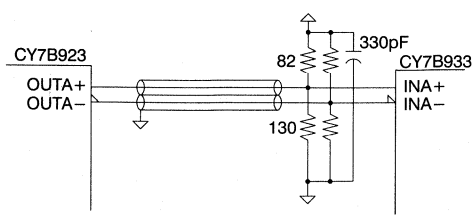


Figure 31. Direct Coupled Copper Interface

When sent through twin coaxial cables (as shown in *Figure 31*) or two separate transmission lines, care must be taken to make sure that both lines are electrically the same length. Any difference in length causes one of the two transmitted signals to arrive at the receiver input either leading or lagging the other. This difference manifests itself as jitter in the receiver. If twisted-pair or twinax is used instead, both the OUTA+ and OUTA− signals combine to form a single signal sent down a balanced transmission line.

Capacitor Coupled

For configurations where it is possible to have significant ground or reference differences, some form of AC coupling becomes necessary. If the signals remain in a well protected environment (minimal EMI/ESD exposure) this AC coupling can be performed with capacitors. When this is done, bias/termination networks are required at both ends of the cable. A schematic detailing this type of connection is illustrated in *Figure 32*.

Good low-loss RF-grade capacitors should be used for this application. These parts are available in many different case types and voltage ratings. The capacitors used must be able to withstand not just the voltage of the signals sent, but of any DC difference between the transmitter and receiver and the maximum ESD expected. A typical 1000-pF 50-WV C0G capacitor would be available in an 0805 surface mount case size (0.08″L x 0.05″W x 0.02″H). For on-board applications a 50-WV rating should be sufficient. While capacitors with much higher breakdown voltages are available, both cost and space make their use prohibitive. This same 1000-pF C0G capacitor at 5-kV breakdown is almost a half cubic inch in size (Reference 15).

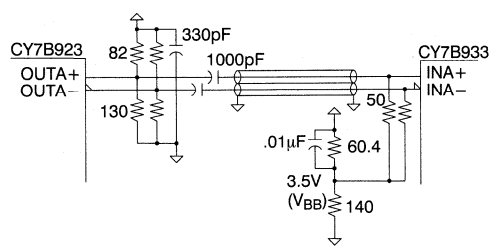


Figure 32. Capacitive Coupled Copper Interface

This type of coupling is very similar to that used to drive an optical module that is not at the same reference as the HOTLink Transmitter. Since the HOTLink Receiver and an optical module both operate with ECL 100K-level compatible inputs, this should be expected.

In this configuration, the receiver reference point is set slightly different from that for a standard ECL receiver. Part of this is due to the HOTLink Receiver being designed for operation at +5V rather than −5.2V or −4.5V. The other is that the HOTLink Receiver has a wider common-mode range than standard 100K ECL parts. To allow operation over the widest range of signal conditions the V_{BB} bias network on the receive end of the transmission line is set to the center of the HOTLink Receiver 3V common-mode range at V_{CC} − 1.5V.

This capacitively coupled interface is not recommended for cabling systems that leave a cabinet or extend for more than a few feet. This is primarily due to

- Limited voltage breakdown under ESD situations of the coupling capacitors
- ESD susceptibility of the receiver due to transients induced in the cable
- Limited common-mode rejection at the receiver end

Addition of a second set of coupling capacitors at the receive end may improve some of these characteristics, but it will not remove them.

Transformer Coupled

The preferred copper attachment method is to transformer couple to the media. Transformers have multiple advantages in copper-based interfaces. They provide

- High primary-to-secondary isolation
- Common-mode cancelation
- Balanced-to-unbalanced conversion

The transformer is similar to a capacitor in that it also has passband characteristics, limiting both low

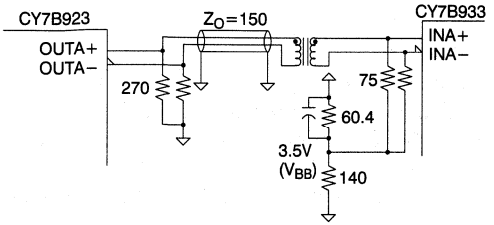


Figure 33. Transformer Coupled Copper Interface

and high frequency operation. Proper selection of a coupling transformer allows passing of the frequencies necessary for HOTLink serial communications. A schematic detailing a transformer coupled interface is shown in *Figure 33*.

This transformer coupled configuration has many similarities to the capacitively coupled interface. It still provides DC isolation between the HOTLink Transmitter and Receiver, and requires the V_{BB} bias and termination network at the receiver.

The connection at the HOTLink Transmitter is quite different now. The output bias network is now a simple pull-down to V_{EE}. While this causes the transmitter outputs to have asymmetric rise and fall times, it does not add to the system jitter. Instead, the true and complement outputs combine in the transformer to provide a single signal with symmetrical rise and fall times. This bias arrangement also the has the advantage of delivering the entire transmitter output voltage swing into the transformer, rather than part into the transformer and part into the bias network.

The configuration shown in *Figure 33* uses only a single transformer and either 150Ω twinax or twisted-pair as the transmission line. This can be done because the transmission system remains balanced from end to end. Here the primary functions of the transformer are to provide isolation and common-mode cancelation. Because most common-mode signal pickup will occur along the transmission line, the transformer should be placed at the receiving end of the cable.

In *Figure 34* a second transformer is added to the transmission system at the source end of the cable. This configuration allows use of either balanced or unbalanced (coaxial) transmission lines. The configuration shown here is a 75Ω coaxial cable system. Here the first transformer is used for balanced-to-unbalanced conversion, while the second transformer provides unbalanced-to-balanced conversion.

HOTLink Receiver ECL Inputs

The HOTLink Receiver has five 100K ECL (PECL) compatible inputs: INA+, INA−, INB+, INB−(SI), and A/$\overline{B}$ (see *Figure 3*). The A/$\overline{B}$ input is used to select which serial data input (INA± or INB±) is fed to the receiver PLL and shifter.

The INA± differential input is normally used for the primary received data input. This input is only functional as a differential receiver. To use it as a single-ended receiver, a V_{BB} reference must be attached to one of the INA± inputs. Since the HOTLink Receiver does not provide a V_{BB} output, this must come from either an external ECL gate or a resistive divider. Because neither of these sources can be guaranteed to be at the exact internal V_{BB} reference of the HOTLink Receiver (and will thus introduce jitter into the system), operation of INA+ in single-ended mode is not recommended. Also, operation in single-ended mode generally takes twice the signal swing (100 mV for HOTLink) for a receiver to properly detect data.

The INB± differential input is expected to be used as the local loopback receiver. It is capable of being operated as a differential receiver, or as two single-ended receivers.

To operate the INB± inputs as a differential receiver it is necessary to have the SO output either

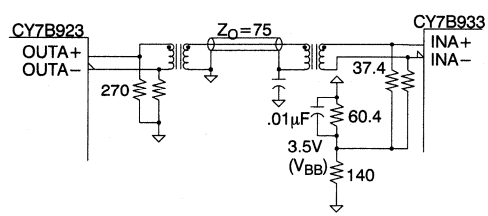


Figure 34. Dual Transformer Coupled Copper Interface

directly connected to V_{CC} or pulled up to V_{CC} through a resistor (minimum of V_{CC} – 250 mV). This pin, while normally used as an output, has a voltage comparator on the output to both disable it and to operate the INB± inputs as a differential pair. When used as a differential receiver the INB± inputs operate the same as the INA± inputs.

If the SO pin is instead allowed to remain in the standard TTL output range (below V_{CC} – 850 mV), it is enabled as a TTL-level driver, and is the output end of an ECL-to-TTL level translator. In this mode the HOTLink Receiver INB+ input is a single-ended ECL receiver for serial data, while the INB– input becomes the input end of the ECL-to-TTL translator. The expected use of this translator is for converting an ECL carrier-detect signal to TTL levels.

ECL Input Levels

Unlike standard 100K ECL logic, the HOTLink ECL inputs are designed to operate, not only over the full 100K ECL voltage and temperature range, but substantially beyond as well.

Normally 100K ECL inputs should never be raised above V_{CC} – 700 mV. If this occurs, the input transistor saturates and can damage other internal structures in the gate. Because the HOTLink Receiver is designed for use in a communications environment, its input structures are more robust and can be taken all the way up to V_{CC} with no degradation in performance. This provides a common-mode operating range more than twice that of standard ECL.

The HOTLink ECL Receivers also provide higher gain than that available from standard 100K ECL. The receiver is able to fully detect 1s and 0s with as little as 50 mV of differential signal at the inputs. Those few 100K ECL parts capable of differential operation usually specify this at 150–200 mV.

The HOTLink ECL inputs are also robust on the V_{IL}(min) side. When operated in differential mode these inputs provide full functionality down to V_{CC} – 3V, yielding a full 3V common-mode operating range. For single-ended operations these same

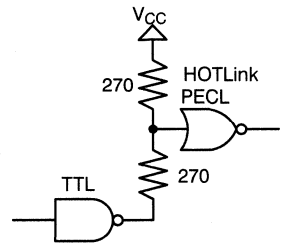


Figure 35. TTL-To-HOTLink PECL Interface

inputs can be taken all the way to V_{EE} (ground or 0V).

Controlling A/B From TTL

While the A/B path select on the HOTLink Receiver is a PECL input, it can be controlled from a TTL driver with as few as two resistors. Controlling a traditional PECL input from TTL normally requires a third resistor to limit the high state to the specified V_{IH}(max). Only a two resistor divider is needed with the HOTLink Receiver (as illustrated in *Figure 35*) because it can tolerate a full V_{CC}-level on its ECL inputs.

HOTLink Receiver Biasing

Unlike ECL outputs, which always require an output bias to create the output-low level, ECL inputs instead require levels within their input range to allow them to switch. When the HOTLink Receiver is directly connected to the output of either a 10K, 10KH, or 100K ECL driver (see *Figures 16* and *17*), these conditions are satisfied.

PECL Optical Modules

Connecting a PECL optical module to the HOTLink Receiver is the same as connecting two ECL parts together. This is connection is illustrated in *Figure 36*.

A bias network is required on the output of the optical module to allow it to switch. A Thevenin or Y-bias network should be used on the high-speed serial lines (RO and NRO as illustrated in *Figure 36*) to keep induced jitter to a minimum. The signal- or carrier-detect output (SIGO) of the module is con-

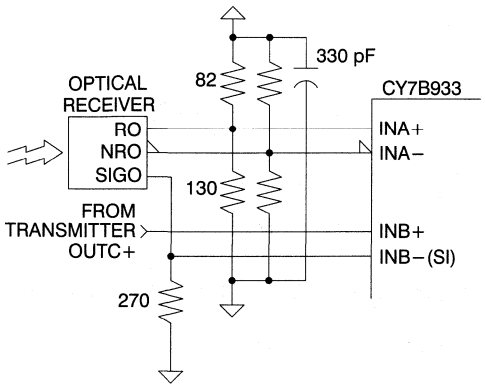


Figure 36. PECL Optical Module-to-HOTLink Receiver

sidered as a logic level signal and only requires a pull-down type of biasing to allow the output to switch.

If the distance between the optical module and the HOTLink Receiver is short (see *Table 3*) then the bias network may be placed anywhere between the optical module and the HOTLink Receiver. If this distance is long, then the interconnect traces must be treated as a transmission line and the bias network must be moved to the receiver to also act as line termination. If the transmission line impedance is other than 50Ω, then different values of resistors are necessary (see *Equations 13* and *14,* and *Table 4*).

Standard ECL Optical Modules

Optical modules with the Case pins connected to V_{CC} are designed for use in a negative DC supply system. These types of modules may also drive a HOTLink Receiver.

By far the simplest method is to connect the module the same as a PECL module, with the exception of the Case pins. Here, instead of attaching the Case pins to ground (V_{EE}), they are attached to V_{CC}. If the case is metallic in nature, care must then be exercised such that it does not come into direct contact with ground.

If the optical module is used below ground it must be AC coupled to the HOTLink Receiver. A schematic detailing this type of connection is illustrated in *Figure 37.*

From a parts count standpoint this type of connection should be avoided if at all possible. Just as with the HOTLink transmitter-to-negative referenced ECL optical modules, this interface requires biasing on both sides of the AC coupling capacitors.

Because the signal detect output of the optical module is not an AC signal, capacitive coupling cannot be used to feed this signal into the HOTLink Receiver INB− input. The simplest thing to do here is to use an external ECL-to-TTL translator (as illustrated in *Figure 37*) to convert the signal-detect output to a positive referenced TTL environment.

The INA± differential inputs must be biased to near the midpoint of the common-mode range of the HOTLink Receiver. The two 50Ω resistors tied to this synthesized reference point are sized to properly terminate the transmission line impedance of the interconnect.

Receiving From Copper Media

The direct-coupled, capacitor-coupled, and transformer-coupled configurations for copper interconnect are covered in the HOTLink transmitter-to-copper interface section of this document, with schematics of same illustrated in *Figures 31* through *34.*

Signal-Detect For Copper Interface

When interfacing to optical modules, the generation of a carrier- or signal-detect function is a simple connection to an ECL output. With a copper interface, this signal-detect function must be built from other components.

The key to a good signal-detect implementation is to create one that accurately detects the presence or absence of a valid data stream, yet does not load or distort the received signal. A sample carrier-detect circuit is shown in *Figure 38.*

This circuit uses a reference divider-network similar to that in *Figure 37,* except that an additional voltage reference point is created. This new reference point sets a threshold for received amplitude at which the signal detect circuit will start to respond. For this

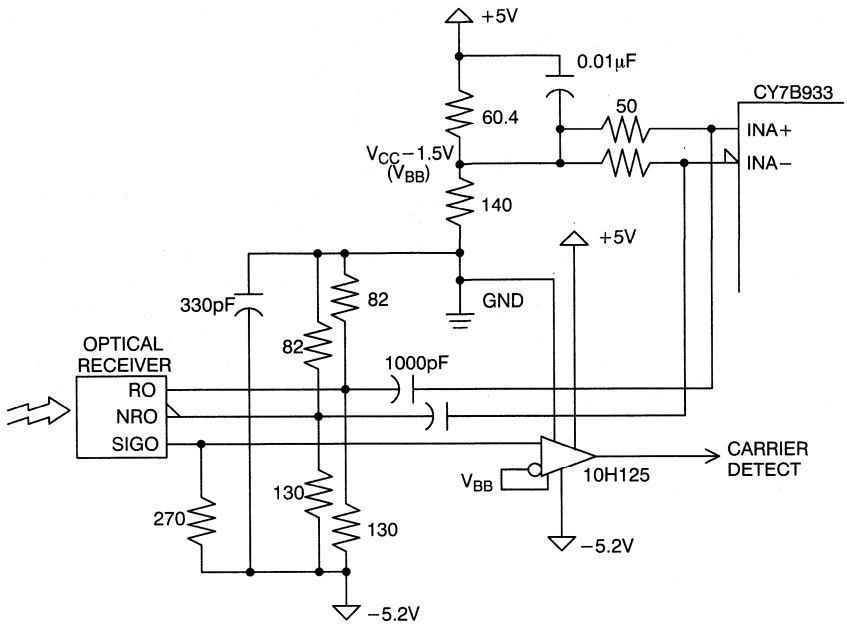


Figure 37. Negative Referenced Optical Module-to-HOTLink Receiver

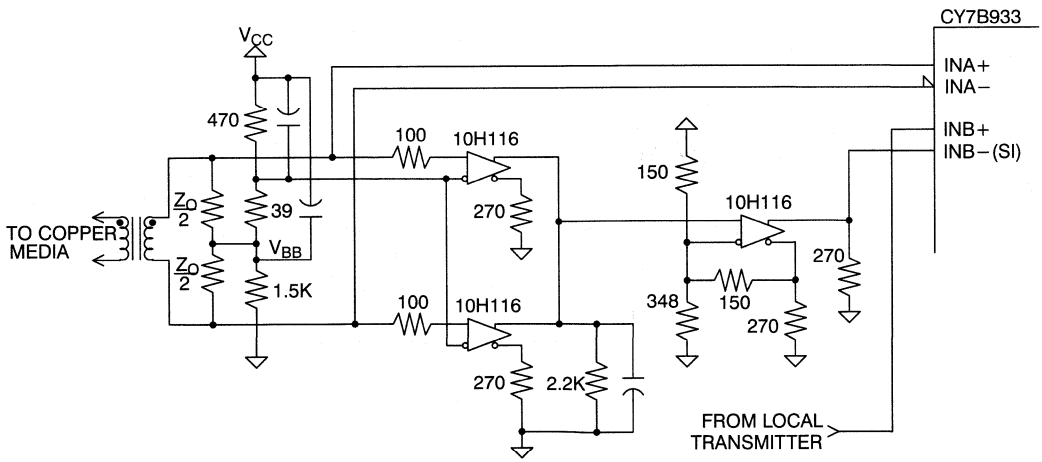


Figure 38. Copper Interface Signal Detect Circuit

example, this reference point is set to 100 mV above the carrier detect receiver V_{BB} reference point. This 100-mV offset is also necessary to prevent the 10H116 amplifiers from oscillating when no signal is present.

A 10H116 was selected here for numerous reasons. It is small (20-pin PLCC), fast (1 ns), and does not have 50-kΩ pull-down resistors built into its input structures. While these pull-down resistors (present on most ECL parts) are very handy for logic design, they have a significant impact when used for fast analog applications as done here.

Two sections of the 10H116 are used as received signal level comparators. One looks for logic-1 levels while the other look for logic-0 levels. The output of these two comparators are wire-ORed together and feed an RC network. The capacitor in this network is charged when either of the comparators is turned on, and discharges through a bleeder resistor when neither comparator is on.

The third section of the 10H116 also operates as a comparator, evaluating the voltage level on the RC network. Because the level on this capacitor changes so slowly, and ECL operates as an analog amplifier, positive feedback was added to cause the comparator to switch faster and to full ECL levels. The amount of hysteresis is set by the feedback resistor. For slow changing signals of this type, a minimum of 150-mV of hysteresis is recommended.

Copper Signal Characteristics

Communication on copper-based media is very similar to communication on optical fiber. Both suffer from increasing signal degradation with increasing media length.

The transmitted signal is composed of multiple frequency components, and requires a fairly wide bandwidth media to propagate those signal components. A large part of the bandwidth requirement is determined by the 8B/10B code and NRZ modulation used in HOTLink for communication.

NRZ Modulation

NRZ is an acronym for Non-Return-to-Zero. This is one of the most basic types of data encoding whereby a signal is HIGH for a 1 and LOW for a 0. The upper waveform in *Figure 39* illustrates an NRZ data stream. Other forms of modulation (Manchester, Biphase, etc.) are used in data communications that encode clock information as part of the 1s and 0s. With an NRZ data stream, a phase-locked-loop is necessary to recover the bit clock to allow data to be captured (Reference 18).

8B/10B Code Dependencies

A phase-locked-loop (PLL) requires transitions meeting specific criteria to allow it to recover a clock. If binary data were sent serially using only an NRZ modulation, long periods could exist where no transitions are sent. During these periods (if they are long enough) the receiving PLL can drift such that it is no longer able to properly recover the data sent. 8B/10B encoding is used to ensure that sufficient transitions are present in the NRZ data stream such that the receiving PLL remains synchronized to the data.

The 8B/10B code is a run-length limited code. This means that there are limits to the maximum and minimum length of a continuous sequence of 1s or 0s in the data stream. The code operates by converting an 8-bit data byte (with uncontrolled transitions) into a 10-bit transmission character (with controlled

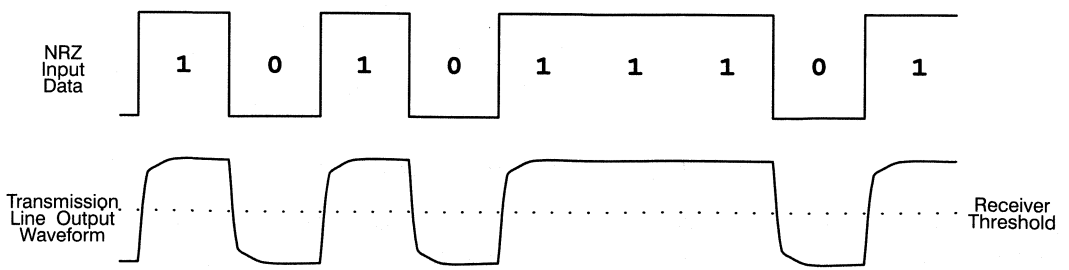


Figure 39. Short Time Constant Transmission Line Response

transitions). The 8B/10B code is referred to as a 1:5 code because the minimum number of consecutive 1s or 0s is one, while the maximum number is five (References 1, 2).

Translating these code limits into frequencies gives the baseband limits of the code. For example, with a serial bit-rate of 300 MHz, a pattern sent with the maximum number of consecutive 1s and 0s (five high, five low) would be equivalent to a 30-MHz square wave. Using the highest rate of alternating bits of 0 and 1 gives a frequency of 150 MHz.

As far as signal propagation goes, these numbers only refer to a sinusoidal frequency. Since square waves are used at the source, there are many additional higher frequency harmonics present. To propagate a reasonable signal it is recommended that the system bandwidth also include at a minimum the 3rd harmonic of the highest baseband frequency, and preferably through the 9th harmonic.

For our previously described example operating at a bit-rate of 300 MHz, the necessary system bandwidth would be

$$BW = (3 \times F_{MAX}) - F_{MIN} \quad \text{Eq. 22}$$

$$BW = (3 \times 150MHz) - 30MHz = 420MHz \quad \text{Eq. 23}$$

Transmission Line Effects On Serial Data

In a perfect world we could launch a perfect square wave down a perfect transmission line and it would come out the end looking the same as it went in. Unfortunately, the laws of physics make such a transmission line impossible.

Instead, our transmission lines have significant amounts of parasitic capacitance, inductance, resistance, and the terminations are reactive in nature. This means that a lossy system exists. The cable attenuation characteristics of copper cables are such that the higher frequencies have greater losses than the lower frequencies (see *Figure 79* for some sample cable attenuation curves).

When data is sent through such a lossy medium, distortion occurs. The higher frequency components are significantly reduced in amplitude, while the lower frequency components are reduced by a lesser amount. The square waves fed into the cable come out looking like RC charge/discharge curves.

These frequency-selective losses are equivalent to a time constant. For very short transmission lines (or very slow data rates) this time constant is short enough that transmitted 1s and 0s can completely charge or discharge the transmission line for each bit sent. The input and output signal waveforms for a transmission line of this type are illustrated in *Figure 39*.

Because the line can fully charge or discharge on even the fastest possible transition, the time to reach the receiver threshold is always the same. This allows the data out of the receiver to look just like the data sent into the transmission line.

As this transmission line is lengthened, its time constant increases. When the time constant is large enough that the line can no longer be fully charged and discharged in a single bit time, the received data edges become time displaced from their desired positions. Since coding theory refers to each transmitted 0 and 1 as a symbol, this type of distortion is called *intersymbol interference* or ISI. For communications systems, distortion of this type is called Data Dependent Jitter (DDJ).

Input and output waveforms for a long time constant transmission line are shown in *Figure 40*. The receiver output is added to illustrate the edge displacement. As the transmission line becomes increasingly longer it is even possible for some single-bit transitions to not be detected at all by the receiver (based on the data pattern sent) because they fail to cross the receiver threshold. This may be corrected through use of frequency compensation circuits at either the source (precompensation) or destination (equalization) ends of the transmission line.

8B/10B Code Running Disparity

The 8B/10B code attempts to limit the maximum distance (voltage) from the receiver threshold that a transmitted signal can reach, by controlling the DC signal content of the characters sent and the maximum separations between 1s and 0s used to represent each character. To do this the 8B/10B code provides two 10-bit sequences to represent

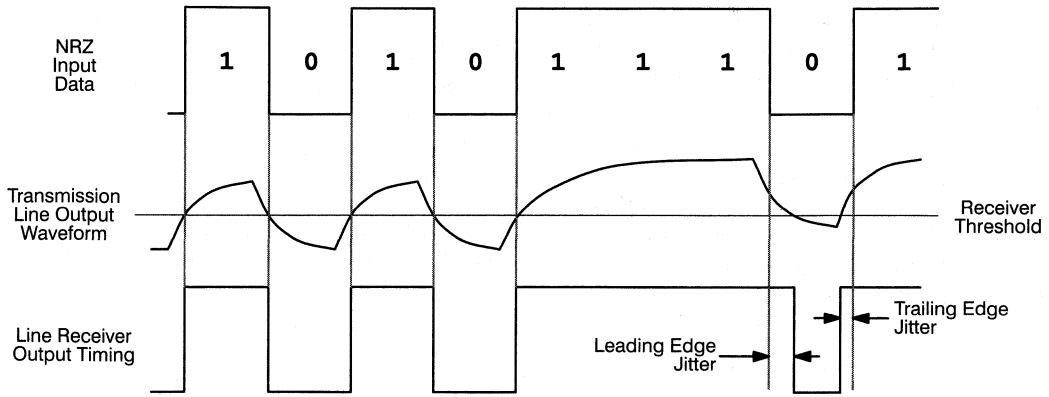


Figure 40. Long Time Constant Transmission Line Response

each 8-bit data value. The difference between these patterns is the ratio of 1s to 0s. To determine which of the two values to send, the HOTLink Transmitter counts the number of 1s and 0s used to send each 10-bit transmission character (when operated with the encoder enabled). If the net result is more 1s than 0s (referred to as positive running disparity), the following data byte is encoded using the form with more 0s than 1s. If the net result is more 0s than 1s (referred to as negative running disparity), the following data byte is encoded using the form with more 1s than 0s. The goal of this is to maintain as near as possible a net value of DC over time for the serial data sent to minimize baseline wander.

Baseline Wander

Methods of data encoding that are not DC balanced (i.e., 4B/5B as used with FDDI) suffer from a characteristic known as baseline wander. This is a side effect of an AC coupled system attempting to propagate a signal that contains a DC component.

Baseline wander is a (relatively) long-term, low-frequency effect, generated when the average DC-level of a transmitted signal varies with the data sent. This DC component is lost because the transmission system is AC coupled. At the receiving end of the cable this appears as data that does not remain centered around the receiver threshold. This effect is illustrated in *Figure 41*.

If the receiver was actually presented with perfectly square pulses (with transitions that always crossed the receiver threshold) then baseline wander would not be a problem. Unfortunately, what are actually sent and received are more in the form of trapezoids with measurable rise and fall times. The farther a signal drifts from being centered around the receiver threshold, the more that the threshold crossings are time displaced. This time displacement is also known as jitter.

Jitter

Jitter is a high-frequency deviation from the ideal timing of an event. Many different aspects of a serial link can affect the total jitter present in the link. Those based on real and repeatable direct measurements are referred to as *deterministic jitter*. Other effects, which are not directly repeatable and are more probabilistic in nature, are called *random jitter*.

Deterministic jitter itself may be broken into two major components: those based on the accuracy of the duty cycle of the information, and those based on the interaction of the 1s and 0s due to the limited

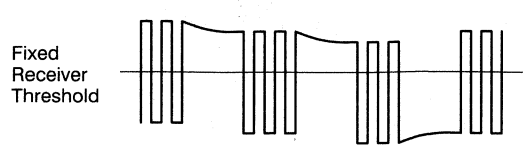


Figure 41. Baseline Wander Example

bandwidth of the transmission system. The jitter that affects adjacent edges and duty cycle is called Duty Cycle Distortion (DCD). The jitter based on the data patterns sent is called Data Dependent Jitter (DDJ).

Data Dependent Jitter Characteristics (DDJ)

Data Dependent Jitter (DDJ) is a measurement of intersymbol interference based on the maximum timing deviations caused by a worst-case data pattern. DDJ is affected by many environmental characteristics, in addition to the code used. These include the length of the cable, the attenuation characteristics of the cable, the integrity of the signal launched into the cable, and how well the cable is terminated. Because of the frequency selective attenuation present in copper cables, DDJ is one of the main limiting factors on how far a recoverable signal may be sent.

To measure DDJ for a specific configuration, data patterns having specific characteristics need to be repeatedly launched into the cable. These patterns must present the worst-case transition characteristics based on the code used for sending data. This is usually described in terms of sequential combinations of long and short 0s and 1s.

A long 0 or 1 is specified as the longest continuous LOW or HIGH that can be sent. For the 8B/10B code this is five bits in length. The short 0 or 1 is the shortest LOW or HIGH that can be sent. For the 8B/10B code this is one bit in length. The sequences used for testing are diagrammed in *Figure 42*.

A design feature of the HOTLink Transmitter is that, when operated in encoded data mode (transmitter MODE pin tied LOW), if no data is clocked into the part it repeatedly sends out the K28.5 SYNC code. The 10-bit pattern of this code is 0011111010. Since in encoded mode the transmitter also tracks disparity, this pattern is inverted on every other byte. This alternating pattern contains the necessary combinations of long and short 0s and 1s for performing a proper eye pattern test.

The opening of the "eye" (see *Figure 43*) relative to the width of a bit cell is a good measure of link integrity. As this window gets smaller, it becomes more

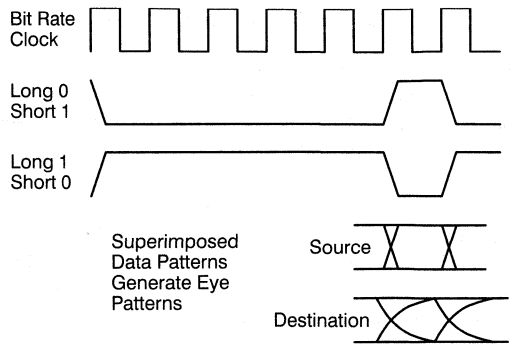


Figure 42. Eye Pattern Generation Waveforms

difficult for the HOTLink Receiver PLL to determine where to sample each bit cell (Reference 5).

The maximum variation, from early to late, of when the received signal crosses the receiver threshold is equal to the amount of jitter present. This jitter is usually expressed as a percentage relative to the width of a bit cell window. This relationship is shown in *Equation 24*.

$$Jitter = \frac{Bit_{TIME} - Th_{VAR}}{Bit_{TIME}} \times 100\% \qquad \text{Eq. 24}$$

The oscilloscope illustration in *Figure 44* is an actual DDJ measurement based on a 100 foot (30.4 m) segment of RG59 cable. The jitter present in this configuration is approximately 600 ps.

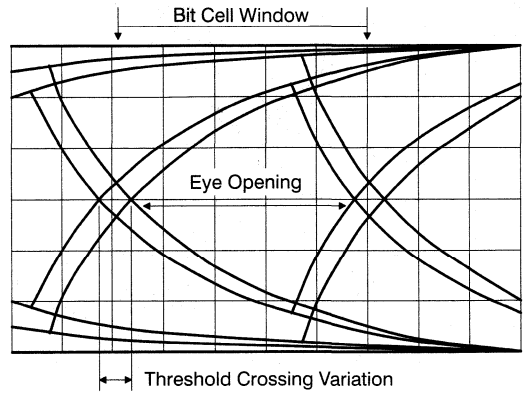


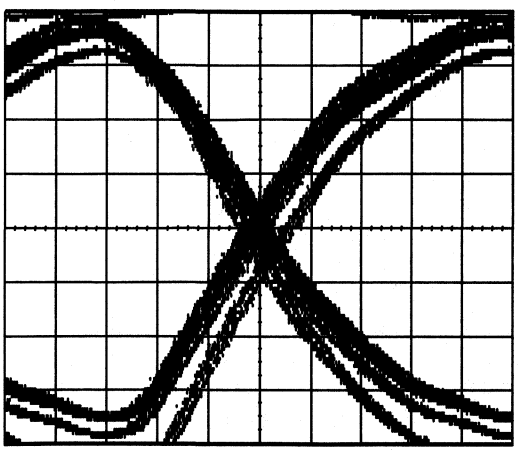
Figure 43. Eye Diagram

Duty Cycle Distortion (DCD) Jitter Characteristics

In most cases Duty Cycle Distortion is caused by the components used to make a link, rather than the data sent across the link. It manifests itself as either differences in the rise and fall times or differences in period for bits sent as a 0 compared to bits sent as a 1. This is measured by sending a pattern down a communications link that does not exhibit data dependent jitter (DDJ) and using an averaging mode on the oscilloscope to filter out any random jitter (RJ) that may be present.

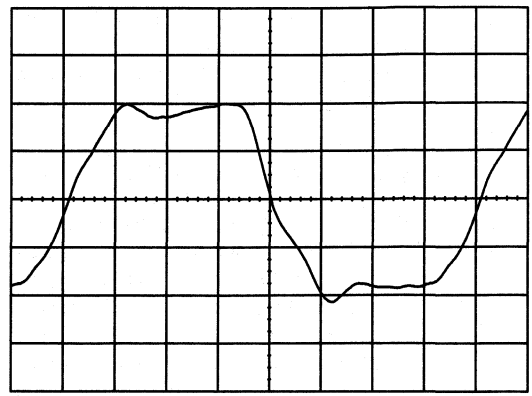
The HOTLink Transmitter has a built-in DCD pattern generator that is activated by placing the transmitter in BIST mode ($\overline{\text{BISTEN}}$ LOW) while both $\overline{\text{ENA}}$ and $\overline{\text{ENN}}$ remain HIGH. In this mode the transmitter sends out an alternating 1−0 pattern (D10.2 or D21.5). As all pulses in a square wave are the same, this pattern does not generate any DDJ. An example measurement of DCD for an optical link is illustrated in *Figure 45*.

When viewed from the receiver threshold (center horizontal line in *Figure 45*, the timing for a logic 1 is seen to be slightly shorter than that of a logic 0. This difference in time is the DCD jitter present in the link.



Ch. 2 = 100.0 mvolts/div Timebase = 500 psec/div

Figure 44. DDJ Measurement

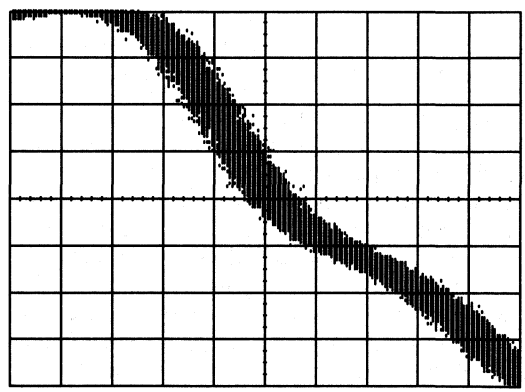


Timebase = 1.00 ns/div Ch. 1 = 200.0 mvolts/div

Figure 45. DCD Measurement

Random Jitter (RJ) Characteristics

Random jitter is that portion of jitter that is not repetitive in nature and is caused by external or internal noise in a system (thermal noise, EMI, etc.). It is measured by using a data pattern free of DDJ (i.e., the same pattern used to measure DCD) relative to the transmitter clock. Now, averaging is turned off but infinite persistence is enabled. This captures the maximum variation of a transition relative to the clock. An example measurement of RJ for an optical link is illustrated in *Figure 46*.



Timebase = 200 ps/div Ch. 1 = 100.0 mvolts/div

Figure 46. RJ Measurement

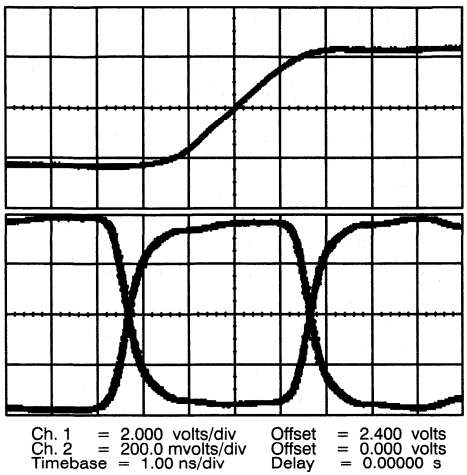


Figure 47. HOTLink Transmitter Serial Data

In this measurement the amount of random jitter present is measured by how wide the trace is as it crosses the threshold. This particular optical link has approximately 200 ps of random jitter present. This measurement was made using a 250-Mbit/second data pattern (4-ns/bit). *Equation 24* yields an RJ of 5% for this link example.

When making measurements of this kind, the tolerances of the signal sources and accuracy of the test equipment must also be taken into account. If the trigger source contains 50 ps of jitter, and the scope trigger accuracy is ±50 ps, then the actual jitter present may be substantially less than that measured.

Frequency Characteristics of 8B/10B Data

Most digital design engineers are used to viewing signals in the time domain using an oscilloscope. This instrument provides information about how a signal looks referenced to the passage of time. The waveforms in *Figure 47* illustrate the HOTlink Transmitter CKW clock on the upper trace and one of the ECL data output signals on the lower trace. The individual bit cells may be seen as the eye between the rising and falling output edges.

In the 8B/10B code, data is sent as a Non-Return-to-Zero (NRZ) waveform. In this waveform the clocking information is contained in the edges, while the data is contained in the interval between the edges. While an oscilloscope type display allows us to see what the output looks like in terms of voltage, rise time, period, etc., it does not present any frequency specific information. To properly design filters, couplers, or transmission systems, it is necessary to know the frequency characteristics of the signals. This information can only be examined through use of a spectrum analyzer.

A spectrum analyzer could easily be called a frequency domain oscilloscope. A conventional spectrum analyzer operates as a swept frequency, superheterodyne receiver that displays a signal's amplitude versus its frequency. It operates by sweeping a narrow-band tuned filter across a specified section of the electromagnetic spectrum, and measuring (and displaying) the rms voltage of the signal at each frequency. This swept filter technique shows the specific frequency components that make up a complex signal, but does not provide any phase related information (Reference 22).

The spectrum analyzer output in *Figure 48* illustrates the spectral characteristics of the HOTLink Transmitter serial outputs when sending the 511-byte BIST pattern. The data patterns sent in the BIST loop are similar to those sent during normal communications traffic. This figure was made using a 30-MHz byte-rate clock (300-MHz bit-rate data). The envelope shows a relatively even distribution of

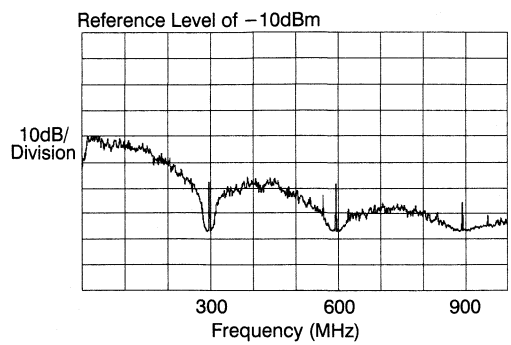


Figure 48. BIST Pattern Spectral Characteristics

power below the bit-rate of the data, and significant amounts of energy present in the information out to 1 GHz. This illustrates how necessary it is to have a true wideband transmission system to propagate the signals.

Figure 48 also shows a large dip in the energy distribution below 30 MHz. This confirms that the 8B/10B code used has no true DC component.

Figure 49 illustrates the spectral characteristics for the highest frequency data pattern that can be sent, a continuous 0101 (D21.5 character) pattern. With the 30-MHz byte-clock used here this pattern is equivalent to a 150-MHz square wave. Unlike *Figure 48*, most of the energy here is located at the fundamental base frequency of 150 MHz, and at odd harmonics of that frequency. Other frequency components present in the signal are at least 30 dB down from the data being sent. These components are either generated by other parts of the HOTLink circuitry as it clocks, encodes, shifts, etc., the users data, or from external sources such as power-supply switching noise.

Figure 50 shows the spectral characteristics for the lowest legal frequency pattern that can be sent, a continuous 0000011111 (K28.7) pattern. This pattern ends up being an exact match in period to the source clock (30 MHz) with a fixed 50% duty cycle. Here, the largest amounts of energy are present at 30 MHz and all odd harmonics above that. The smaller frequency components present at the even harmonics are again due to the internal operation of the HOTLink Transmitter and external system noise. If this figure is compared to *Figure 49*, many of these even harmonic components can be seen to have almost exactly the same level in both figures.

To verify that these spectral characteristics have some resemblance to theory, these same two source waveforms were generated mathematically and analyzed using an FFT (Fast Fourier Transform) algorithm. This transform analyzes a source waveform and computes its frequency components.

Because the input waveforms are not true square waves, time constant curves based on a natural logarithm were used to synthesize the the rising and falling edges. These rising and falling edge equations are listed in *Equations 25* and *26* respectively.

$$T_R = 1 - e^{(-t/T)} \quad \text{Eq. 25}$$

$$T_F = e^{(-t/T)} \quad \text{Eq. 26}$$

In these equations, T represents the time constant for rise and fall time. For the waveforms generated here, a T of 400 ps was used. *Figure 51* illustrates the signal generated with these equations for a 150-MHz clock rate (300-Mbit/second bit-rate). This is equivalent to the data pattern generated by a D21.5 character.

Running a 4096 point FFT on this waveform yields the spectral components illustrated in *Figure 52*. The vertical axis here is plotted on a log scale to match up with the spectrum analyzer outputs. This plot illustrates that the energy of a square wave having a symmetrical rise and fall is located at the odd harmonics.

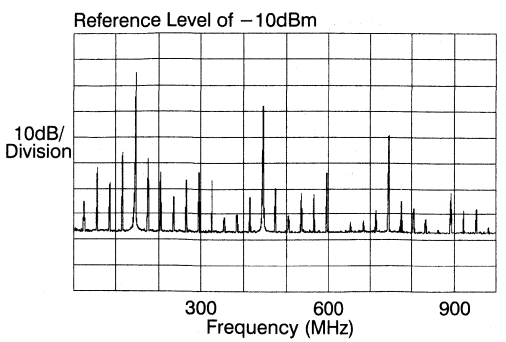


Figure 49. 0101 (D21.5) Pattern Spectral Characteristics

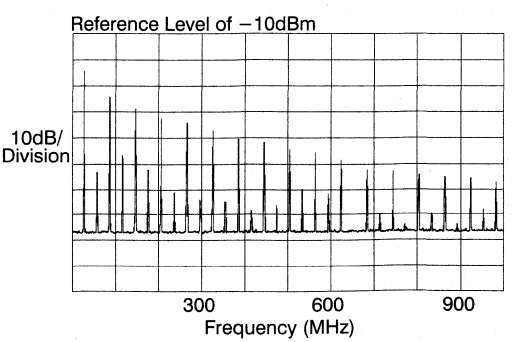


Figure 50. 0000011111 (K28.7) Pattern Spectral Characteristics

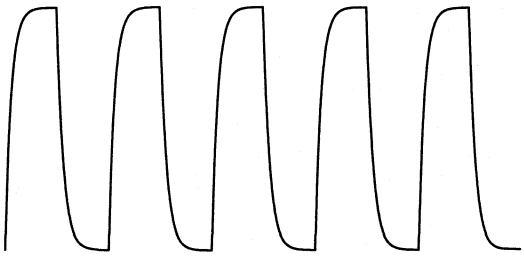


Figure 51. Synthesized D21.5 Waveform

An FFT is based on numeric analysis rather than a physical measurement and will calculate signal components with an amplitude of zero. Because Log(0) is equal to $-\infty$, a calculated FFT does not have a noise floor. To plot its results in a usable form requires the addition of an artificial noise floor to presents the points of interest on a reasonable scale. To allow a better comparison, *Figures 52* and *54* use the same noise floor as that measured in the spectrum analyzer charts.

Unlike a spectrum analyzer which only displays the magnitude of the spectral components, an FFT of a waveform yields both magnitude and phase in rectangular form as a complex number. To plot this information for comparison with a spectrum analyzer plot requires conversion to polar notation of magnitude and phase. This calculation of the magnitude portion is done using *Equation 27* (Reference 24).

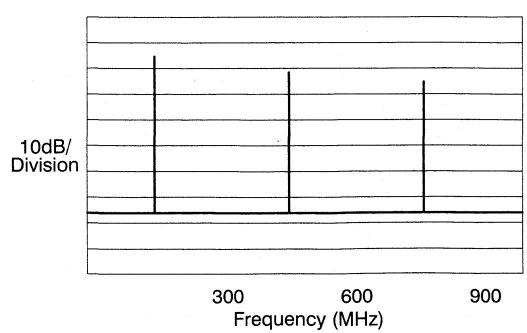


Figure 52. FFT Spectrum of Synthesized D21.5 Pattern

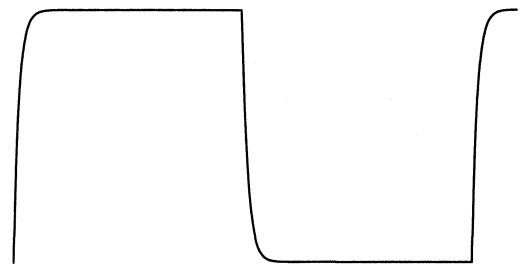


Figure 53. Synthesized K28.7 Waveform

$$Magnitude = \sqrt{Re^2 + Im^2} \qquad \text{Eq. 27}$$

This same FFT analysis was performed on the synthesized K28.7 pattern illustrated in *Figure 53*. This waveform uses the same 400-ps time constant as *Figure 51*. The FFT based spectral plot for this waveform is illustrated in *Figure 54*. Because it uses the same value for a time constant, this waveform has the same rise and fall times as the D21.5 pattern in *Figure 51*. As with the plot for the D21.5 pattern, all of the energy is contained in the odd harmonics.

The spectral plots for both the D21.5 and K28.7 synthesized patterns contain slightly more energy in the higher frequency harmonics than the actual measured signals. This is primarily due to the sharp knee present when the synthesized waveform changes between rising and falling. This knee is much rounder in the actual signal.

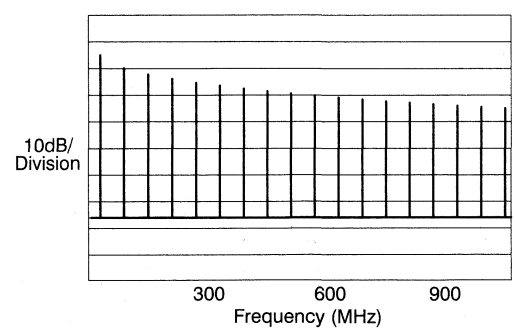


Figure 54. FFT Spectrum of Synthesized K28.7 Pattern

Components

The selection of support components for a HOTLink communications environment should not be taken lightly. The correct parts allow construction of a high-bandwidth, low error-rate system.

Several parts can be measured as key in a HOTLink system. These parts are

- Clock Oscillator
- Bypass/Coupling Capacitors
- Fiberoptic Emitters
- Fiberoptic Detectors
- Pulse Transformers
- Fiberoptic Cable
- Copper Cables
- Circuit Board

Clock Oscillators

The HOTLink Transmitter and Receiver are designed to operate from a very stable clock source. To achieve the necessary frequency accuracy and stability it is necessary for this clock source to be based on a quartz crystal.

The current ANSI Fibre Channel standard calls out a frequency accuracy of ±100-ppm for both source and destination (ANSI FC-PH 4.1 Section 6.1.2 Table 8, and Section 8 Table 9) to allow reliable communications. Clock oscillators with this initial accuracy are available from multiple sources (Reference 3).

What must also be considered is lifetime stability. Most oscillator manufacturers can easily deliver product that meets the ±100-ppm rating right out of the box, but this limit must be met over the life of the product, and is affected by the operating environment. The two most critical parameters are referred to as *aging* and *temperature* stability.

Aging refers to how an oscillator's output frequency varies over time (assuming other environmental factors remain constant). This is usually expressed in ppm/year. For most common "AT" cut crystals, the typical aging is 5 ppm/year for the first year and 3 ppm/year thereafter (5 ppm=.0005%).

A crystal's resonant frequency also varies with temperature. How much it varies is based both on how the crystal is cut, and over how wide a temperature range it is used. The stability over temperature is a non-linear function and is usually expressed as some peak-to-peak frequency change over a temperature range. The process for measuring and specifying temperature stability is called out in MIL-O-55310. Temperature stability may easily exceed the initial accuracy specification. Ratings of ±100-ppm for temperature alone are not uncommon. *Figure 55* shows a typical transfer curve of crystal frequency vs. temperature.

This curve can be rotated on the +25°C axis point by cutting the crystal differently. This can be used to create an oscillator that is more stable over a narrow temperature range (say 0°C to +50°C), yet is much more unstable outside of this range.

Temperature stability and initial accuracy are often combined in a vendor's specification; i.e., ±100-ppm @0°C to 70°C. These numbers do not take into account the aging characteristic of stability.

Modified oscillators are available that allow for a wider operating environment while maintaining a high stability. These are referred to as either TCXO (temperature compensated crystal oscillator) or OCXO (oven controlled crystal oscillator).

The TCXO is usually built by adding a varactor diode in series with the crystal. A special thermistor network across the diode causes the oscillator to maintain a very stable operating frequency. Because of the desired stability of a TCXO (±2 ppm), a better grade of crystal is used to provide better aging characteristics (±1 ppm/year). Oscillators of this type are usually larger in size (and

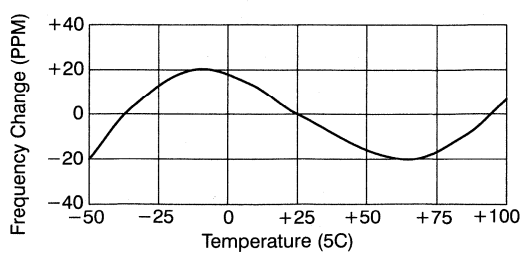


Figure 55. Oscillator Temperature Stability

higher in cost) than the standard 4/14-pin DIP footprint of standard clock oscillators.

The OCXO provides the highest-accuracy oscillators. These are built by placing a standard oscillator into a temperature-controlled environment. Rather than have to both heat and cool the crystal, the operating temperature is set to the upper end of the oscillator's range. Crystals are also cut such that a nearly flat area of temperature response is located at the operating temperature of the oven. The normal operating temperature of crystal ovens is in the 60°C to 100°C range.

Oven-controlled oscillators are generally quite large, expensive, and dissipate large amounts of power. They also have a significant warm-up period, requiring from 15 to 30 minutes after power on to achieve their specified stability (Reference 23).

HOTLink Oscillator Requirements

Unlike the ANSI requirement for ±100-ppm stability for end-to-end communication, the HOTLink family of parts will operate with a substantially wider range of reference frequencies between the HOTLink Transmitter and Receiver. The specification of 0.1% end-to-end frequency tolerance allows operation with oscillator sources operating at up to ±500-ppm tolerance. This allows even the lowest cost oscillators to be used with HOTLink.

Bypass Capacitors

At the frequencies that the HOTLink Transmitter and Receiver operate, the proper usage of power-supply bypassing becomes quite critical. Strategically sized and placed capacitors are used both to provide an AC path between V_{CC} and ground (V_{EE}), and to source current when the power supply cannot respond quickly enough due to the parasitics of the power distribution system.

The base of any power distributing system is the circuit board. Due to the very high frequencies developed in a HOTLink-based communications link, it is strongly advised to use full power and ground planes, rather than attempting to distribute power and ground on the same layers used for signal distribution. These power layers should be made with a minimum of 1-ounce copper.

To properly bypass the HOTLink Transmitter and Receiver it is necessary to know which V_{CC} pins are assigned to which portions of the logic inside the part.

HOTLink Transmitter Power Pins

The pin configuration for the HOTLink Transmitter is illustrated in *Figure 56*. The transmitter has three pins assigned as V_{CC} and two assigned as ground. All three of these V_{CC} power pins are connected internally and *must* be connected externally to the same power rail. The current flow from the slight voltage variations that would exist if different external V_{CC} supplies were used could damage the part.

Pin 4 of the HOTLink Transmitter is named V_{CCN} or Noisy V_{CC}. This pin provides power to the ECL emitter-follower output transistors. This pin is not usually a noise source if the ECL outputs are loaded in a balanced fashion. If these same outputs are operated single-ended with unbalanced loads, then a varying amount of current will flow through this pin as the outputs switch. To keep board noise to a minimum it is advised that, if an output is used, both outputs of the differential driver be loaded the same.

Pin 9 of the transmitter is named V_{CCQ} or Quiet V_{CC}. This pin provides power to the CMOS logic core of the part and the TTL compatible input buffers. This includes the 8B/10B encoder and the

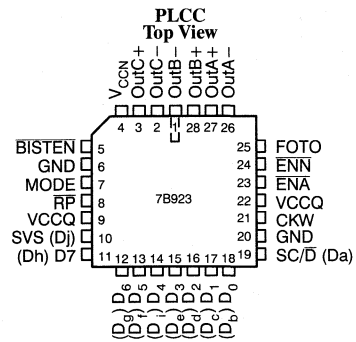


Figure 56. CY7B923 HOTLink Transmitter Pin Configuration

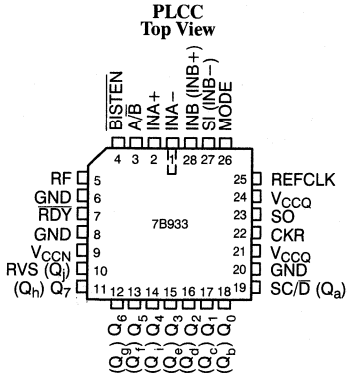


Figure 57. CY7B933 HOTLink Receiver Pin Configuration

counters and state machines used to control the flow of data through the part. Because the dynamic current draw through this pin should not be very large, the primary bypassing concern should be for higher frequency signal components present in the internal logic.

Pin 22 of the transmitter is also named V_{CCQ} or Quiet V_{CC}. This pin is probably the most critical of all the pins on the transmitter as it provides power to the analog core. This includes the charge pumps and comparators used with the PLL clock multiplier.

HOTLink Receiver Power Pins

The pin configuration for the HOTLink Receiver is illustrated in *Figure 57*. The receiver has three pins assigned as V_{CC} and three assigned as ground. All three of these power pins are connected internally and *must* be connected externally to the same power rail. The current flow from the slight voltage variations that would exist if different external V_{CC} supplies were used could damage the part.

Pin 9 of the HOTLink Receiver is named V_{CCN} or Noisy V_{CC}. This pin provides power to the TTL-compatible output buffers. Because there is no way to maintain a constant current load on these outputs (as can be done with the HOTLink Transmitter ECL outputs) there will always be significant dynamic current flow through this pin as the part operates.

Pin 21 of the receiver is named V_{CCQ} or Quiet V_{CC}. This pin provides power to the core CMOS logic in the receiver. This includes the 10B/8B decoder and the counters and state machines used to control the flow of data through the part. Because the dynamic current draw through this pin should not be very large, the primary bypassing concern should be for higher frequency signal components present in the internal logic.

Pin 24 of the receiver is also named V_{CCQ} or Quiet V_{CC}. This pin is probably the most critical of all the pins on the receiver as it provides power to the analog core. This includes the charge pumps and comparators used with the PLL and the input differential amplifiers for the high-speed serial data streams.

Bypass Capacitor Types

For the purposes of power supply bypassing, capacitors are used to store charge, and deliver that charge to a nearby device when necessary. While many still believe that charge is stored on the plates of a capacitor, it is not. Charge is stored in the dielectric (Reference 21).

There are two primary types of chip capacitors used for power supply bypassing; they are identified as either high-K or low-K capacitors. These capacitor types differ primarily in their dielectric material.

The K referred to here is the dielectric constant for the material used as a dielectric in the capacitor. High-K dielectrics for bypass type capacitors are usually based on titanates of barium, calcium, strontium or magnesium. This material provides dielectric constants in the range of 1200 to 12,000. These high-K dielectrics allow construction of physically small capacitors that provide a large amount of capacitance per unit area. The generally available range of high-K capacitors is from 200 pF to 0.1 μF. These high-K capacitors have temperature characteristics of type X7R, Z5U, or Y5V.

Both high-K and low-K dielectrics are used for power supply bypassing. High-K dielectrics are usually not used for temperature critical or high frequency operations because of their thermal and frequency dependent characteristics.

One of the biggest problems with using these high-K dielectric capacitors is sensitivity to temperature.

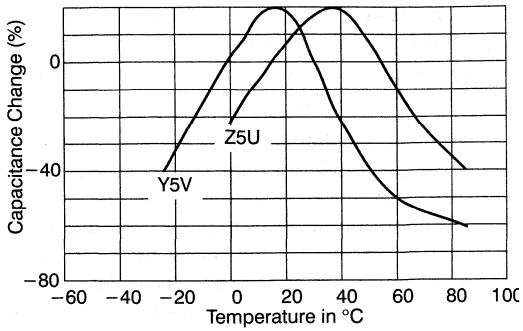


Figure 58. Capacitance vs Temperature For Y5V and Z5U Dielectrics

Per the graphs in *Figures 58* and *59*, these types of parts can change their capacitance values by over 80% over the operating temperature range of most commercial or industrial applications. (The temperature characteristics for Y5V are similar to Z5U except that the peak capacitance occurs around 20°C lower in temperature.)

A second problem is that these titanate-based dielectrics exhibit ferroelectric properties; i.e., they do not respond linearly to an AC signal. The effect is similar to a hysteresis loop in magnetics. This makes these dielectrics a poor choice when a distortion-free analog response is required.

When used for high-frequency (RF) or communications-link type applications, the high-K dielectrics

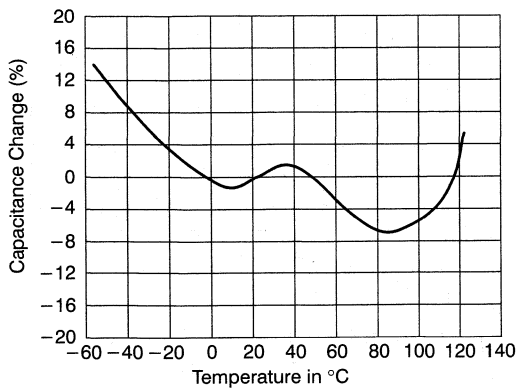


Figure 59. Capacitance vs Temperature For X7R Dielectrics

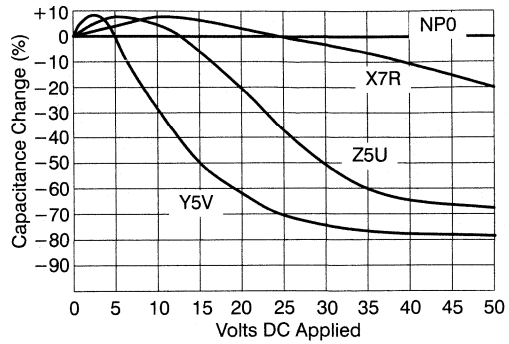


Figure 60. Capacitance vs DC Voltage

have other drawbacks. Capacitors based on these dielectric types are also very sensitive to operating voltage and frequency.

Figure 60 illustrates the voltage sensitivity of high-K dielectrics. Here the capacitance loss can exceed 70% with as little as 25V applied to the part. This parameter may become critical if capacitors are used as part of a DC block in a communications link.

Figure 61 illustrates one of the effects of operating frequency on capacitance. As the operating frequency increases, the high-K dielectrics exhibit less and less capacitance. If these high-K dielectrics are to be used at an RF frequency, a capacitance correction factor must be applied to determine the actual capacitance present in the circuit (Reference 26).

Low-K dielectrics are generally based on either titanium-dioxide ceramic, alumina, or porcelain. These materials provide dielectric constants in the

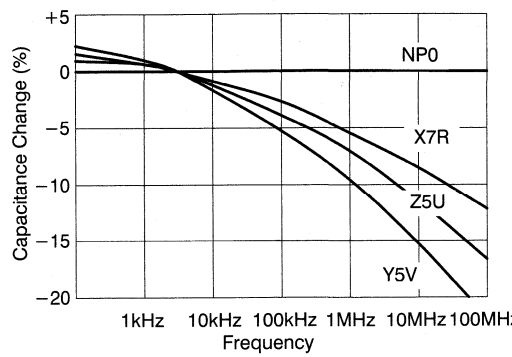


Figure 61. Capacitance vs Frequency

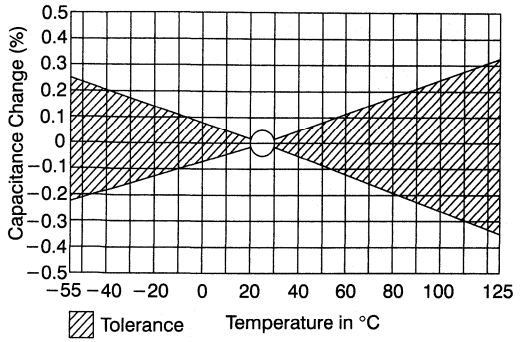


Figure 62. Capacitance vs Temperature for NP0/C0G Dielectrics

range of 9 to 30. Because of the low-K, these materials are only used for making small-valued capacitors in the range of 1 pF to 10,000 pF. These low-K capacitors are usually identified as having NP0 or C0G type temperature characteristics, and are often referred to as RF-grade capacitors because of their high-Q and low dissipation factors.

Low-K dielectric capacitors are very stable over temperature. Per *Figure 62*, these parts change in capacitance less than 0.5% over the full military temperature range of −55°C to 125°C. Because of this temperature stability, low-K capacitors are preferred for many analog applications where fixed time constants and resonant frequencies are necessary.

No capacitor provides a *pure* capacitance; i.e., there are other parasitic resistive and inductive components present in the complex impedance of a capacitor over frequency as illustrated in *Figure 63* (References 15, 16, 17, 21). These parasitic components of a capacitor are due to the materials used in, and mechanical construction of, the physical capacitor. Because of these parasitics, a capacitor cannot be

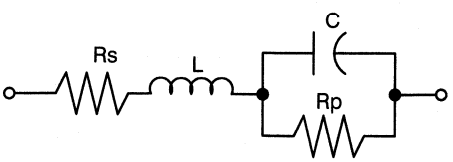


Figure 63. Capacitor Equivalent Model

treated as having ever-decreasing impedance with increasing frequency. At some frequency the capacitor passes through its series resonant point and must then be treated as an inductor.

A general rule of thumb is that as the capacitance decreases, the series resonant frequency increases. This relationship is illustrated in *Figure 64*. At this series resonant point, the capacitive and inductive reactance components cancel each other out, leaving only the effective series resistance (ESR). For most common bypass capacitors, the ESR is well under 1Ω When selecting parts for high-frequency operation, the smaller case sizes (0805 or 0603) are preferred as they have smaller inductive parasitics.

Bypassing Guidelines – Best Configuration

When possible, each V_{CC} power pin of the HOTLink Transmitter and Receiver should be individually bypassed by a combination of high-K and low-K capacitors. While a good ground and power plane does reduce the need for bypassing it does not remove it. The recommended values for these parts are 0.01 µF and 300 pF for the pins supporting output driver structures. The core logic pins require higher-frequency response but less overall current and can be bypassed with a 4700-pF and 100-pF pair.

The analog power pins are the most sensitive to disturbance and may be affected by high frequency noise on the incoming power. The recommended bypass configuration is to provide power through a

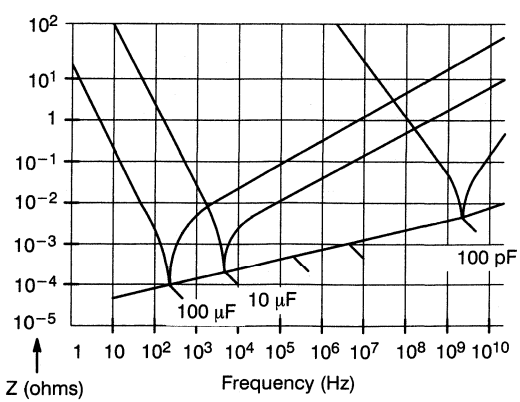


Figure 64. Capacitor Impedance Versus Frequency

small inductor, with a filter cap at the end of this inductor, directly adjacent to the analog V_{CCQ} pin. This inductor should be a 15- to 20-mil-wide foil trace approximately 10 mm in length. It is advised that this trace be on the surface of the board to allow a controlled capacitance environment and easier access for rework or probing. The bypass caps for this pin would be a 0.01-µF and 300-pF pair.

For surface mount versions of HOTLink, the preferred position of these bypass capacitors is on the same surface as the transmitter and receiver. This removes any additional inductance picked up through the board-to-power plane vias used to bring the filtered power through the board. Where multiple capacitors are used with a power pin, the smaller capacitor should always be placed closest to the package pin.

The foil traces connecting power (and ground) to the HOTLink Transmitter, Receiver, and its bypass networks, should be a minimum of 20-mil-wide traces for 1−ounce copper. The vias (preferably multiple to lower the inductance) used to connect to power should also be larger than those used for normal signals to lower the inductance in these areas. Vias for power and ground connections should be a minimum of 10 mil in diameter and preferably plated closed with copper or solder.

Because the ground connection also affects the operation of the design, it also should be enhanced. This generally consists of bringing the buried ground plane to the surface of the board, through multiple vias and area fills, to provide a low inductance path to ground.

Figures 65 and *66* illustrate possible placements of bypass capacitors and power routing for the HOTLink Transmitter and Receiver respectively. Both layouts use a six-capacitor bypass network. All signals are routed to vias and all vias are on a 0.1-inch grid for a low cost test fixture. All capacitors are 0805 surface mount case size. Routing is limited to 45° angles for all traces. Signal traces are 8-mil line width. A HOTLink package outline is superimposed to illustrate package placement and pin 1 orientation.

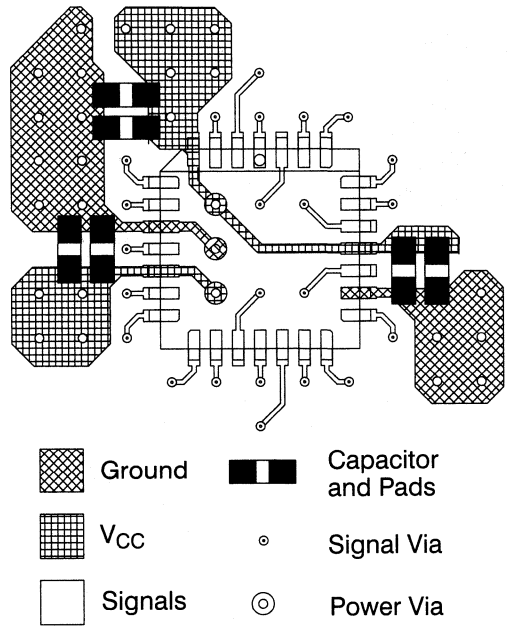


Figure 65. CY7B923 HOTLink Transmitter Sample Layout

Bypassing Guidelines − Minimal Configuration

While the *best configuration* for a design is always desirable, it may not always be possible or cost effective for a given implementation. Different criteria for board layout, manufacturability, inspection, cleaning, cooling, etc., will all affect what can be done in a real world environment.

When surface mount sockets are required to hold the HOTLink Transmitter and Receiver (discouraged, but commonly used), the sockets themselves often limit how close capacitors may be mounted on the top side of the circuit board. In this configuration, placement of the bypass networks on the bottom of the board can often get them closer to the power pins. When placed on the bottom, via diameter and quantity, as well as trace width and quantity, should be maximized to reduce the inductance through the board.

For those implementations where space or cost guidelines do not permit full six-capacitor set-ups

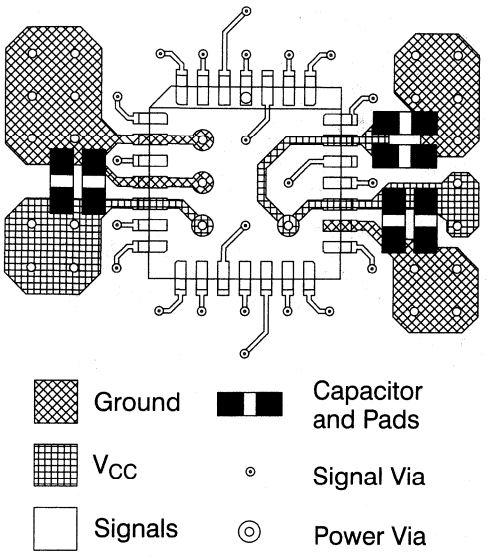


Figure 66. CY7B933 HOTLink Receiver Sample Layout

for each HOTLink part, a subset can be used with good results. The biggest impact may be in slightly higher jitter rates out of the transmitter and a similar decrease in jitter tolerance on the receiver. Here a minimum of one high-K (0.022 µF) and one low-K (330 pF) capacitor should be used per part and be connected as close to the V_{CCN} pins as possible. The connections to the power and ground planes for all power and ground pins should be made as short and wide as possible.

Where circumstances permit, the traces and vias used to connect to the V_{CC} and ground planes for the transmitter and receiver should be kept separate. This limits the switching noise generated in one part from being coupled into the other. Above all else, the use of continuous power and ground planes are the most important factors.

Resistors

Figure 67 shows a first order model of a real world resistor. Because of the parasitic L and C present, a resistor does not have a constant impedance over frequency. The actual amount of change in impedance from a pure resistance is based primarily on the construction of, materials in, and DC resistance value of the component.

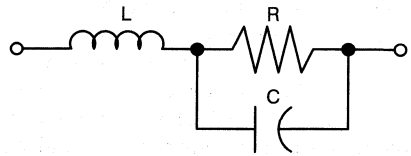


Figure 67. First Order Resistor Model

For high frequency or RF designs, most low value (<1 kΩ) composite (non wire-wound) resistors may be assumed to operate at or near their DC resistance. As the DC resistance of the part increases, its impedance at higher frequencies decreases. *Figure 68* shows this relationship for typical carbon film resistors. This change in impedance is referred to as the Boella Effect and is caused by the distributed shunt capacitance present in the conducting carbon particles (Reference 21).

This shows that low-value carbon film resistors have reasonable impedance characteristics for RF applications, but for higher values a different type of resistor must be used.

For higher resistance values at RF frequencies, metal film resistors should be used. Because these types of resistors are not formed from particulate material, the distributed capacitance is reduced. These types of resistors are manufactured by vacuum sputtering of thin films of mixed metals onto a ceramic substrate. Because there are no individual particles of metal, the capacitance is much lower.

Care must also be used when selecting metal film resistors as some of these have significant inductive

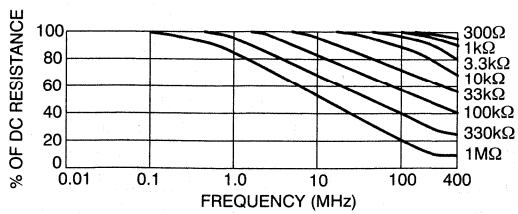


Figure 68. Carbon Film Resistor Frequency Characteristics

parasitics. These inductive parasitics are often caused by the method of laser trim used to adjust the value of the resistor. Those resistors created using two straight cuts, one from either side, are generally more inductive than those trimmed using a single straight or L shaped cut.

Metal film resistors should be used for resistors in the analog data path. This includes the transmission line termination and line bias resistors at both the source and destination ends of the serial link.

Fiberoptic Emitters (Drivers)

A fiberoptic emitter is an electro-optical converter that changes an electrical stimulus into light. A simplified block diagram of a fiberoptic emitter is illustrated in *Figure 69*. The input buffer is an ECL differential line receiver. While some emitters do provide a V_{BB} output to allow single-ended operation, its use is strongly discouraged. The ECL receiver controls a high-current amplifier. The amplifier drives its current through an LED or semiconductor laser to generate a shaped optical output in response to the ECL signal input. A micro-lens assembly (usually a small sphere of glass) is used to couple and direct the light into a port for an optical fiber. Because of the small core size of the optical fiber, the lens and fiber receptacle are aligned by the fiberoptic emitter manufacturer (Reference 27).

Fiberoptic emitters are available in may different case styles, wavelengths, launch modes, data rates, etc. When selecting an emitter, the main concerns are

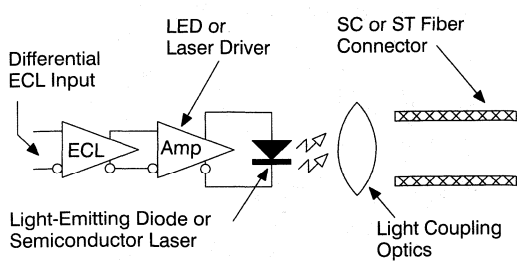


Figure 69. Fiberoptic Emitter Module Block Diagram

- Optical Receiver characteristics
- Operating data rate
- Cable plant characteristics

Most of these areas deal with interoperability of data communications links. If a shortwave laser is used as an emitter, the optical receiver must be designed to operate with the specific data rates and spectral properties of that shortwave laser. While it would be nice if a more mix-and-match combination of LED, shortwave laser, and longwave laser emitters could be used, existing receivers do not allow this. If a 1300-nm LED-driver is used, an optical receiver designed for 1300-nm LED reception must be used to properly detect the signals. In addition the optical receiver must be designed to support the data rate used in the link.

Optical emitter assemblies are available from multiple sources, including AMP/Lytel, Siemens Optical, Hewlett-Packard, Sumitomo Electric, AT&T, and others.

ANSI Fibre Channel Requirements

The current ANSI Fibre Channel standard calls out four optical interface technology options for use at the 25-MByte/second data rate supported by HOTLink. The ANSI designators for these technology options are (Reference 3)

- 25-SM-LL-L
- 25-SM-LL-I
- 25-M5-SL-I
- 25-M6-LE-I

These designators are interpreted as four fields. The first field identifies the data rate used (25 MBytes/second).

The second field identifies the media used. SM specifies single-mode fiber, M5 specifies 50-μm core multimode fiber, and M6 specifies 62.5-μm core multimode fiber.

The third field identifies the transmitter type. LL specifies a 1300-nm longwave laser, SL specifies a 780-nm shortwave laser, and LE specifies a 1300-nm LED-driver.

The last field identifies the distance class of the link. L specifies long distance (2m–10 km), and I specifies intermediate distance (2m–1.5 km).

HOTLink will correctly operate with all these different link types. However, it is up to the user to select the proper combination of emitter and detector for each class.

For those users intending to implement laser-based optical links, there are a number of federal and international safety certifications required before any such link can be put into public use. These safety requirements (ANSI Z136.1 and Z136.2, F.D.A. regulation 21 CFR subchapter J, and IEC 825) are called out in the ANSI Fibre Channel standard (References 9, 10, 11, 12, 13). No such certification requirements are necessary for LED based links.

Power Distribution Requirements for Optical Drivers

The LED or laser used to drive the optical link is probably the largest noise generating item in an optical link. When the optical driver is turned on (sending 1s), currents of 50 mA to 100 mA are forced through the LED or laser. While current steering is often used to minimize dynamic current requirements, significant high-frequency noise is still generated. Most optical modules attempt to remedy part of this situation by providing multiple V_{CC} and V_{EE} pins on their package, and including some power supply bypass capacitance inside the optical module. This does take care of some of the problem, but does not correct all of it.

While bypass capacitors are still necessary to provide dynamic current, additional power isolation and filtering is required to separate the high noise of the optical transmitter from the highly sensitive optical receiver, and from the serializer/deserializer operations of the HOTLink Transmitter and Receiver. Vendors recommendations for this include a 10-µF solid Tantalum capacitor located near the optical transmitter, and a 0.1-µF decoupling capacitor directly connected to the optical transmitter V_{CC} pins (Reference 27).

Isolation is provided by separating the V_{CC} or power plane for the transmitter from the rest of the surrounding power plane, through an inductive

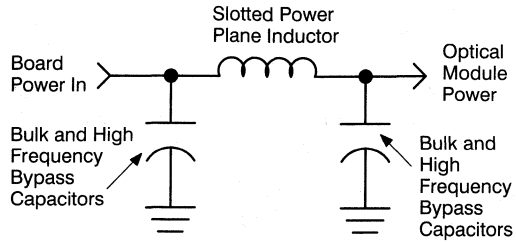


Figure 70. Optical Module Power π-Filter

path. This is done by placing a gap in the V_{CC} plane around most of the the transmitter V_{CC} pins with a single limited connecting point. If the transmitter package only has one or two V_{CC} pins, these may be treated individually by bringing in power through a small inductor or surface trace. For a low noise environment this inductor may be constructed as part of the circuit board using a 15-mil-wide trace approximately 10 mm in length (approximately 5 nH). The specified bypassing should occur after this inductive trace, right next to the optical transmitter. The net result is to implement a π-filter using the circuit board and capacitors for the different filter elements. This is illustrated schematically in *Figure 70.*

An example slotted power plane used to implement the inductive element in the π-filter is illustrated in *Figure 71.* This illustration details an actual power plane layout for an optical module. The black areas indicate the absence of copper. The slot in the center of the figure is used to separate the power for the optical transmitter from the optical receiver. The shaded line on the right hand side indicates a surface

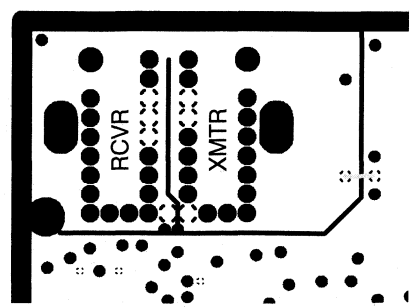


Figure 71. Fiberoptic Module Slotted Power Plane

layer trace (inductor) used to separate power for the optical module from the remainder of the design.

Fiberoptic Detectors (Receivers)

A fiberoptic detector is an opto-electric converter that changes a light stimulus into an electrical signal. A simplified block diagram of a fiberoptic detector is illustrated in *Figure 72*. Light enters the module through an optical fiber and is guided by the connector housing. A coupling lens focuses all available optical energy onto the active region of a light sensitive diode. The presence or absence of light affects the amount of current flow through the diode. This small current flow is then amplified by a transimpedance amplifier which then feeds an ECL differential driver. Many fiberoptic detectors also contain additional circuitry such as signal-detect (Reference 27).

Fiberoptic receivers are generally available from the same vendors as fiberoptic emitters. As with fiberoptic emitters, the optical receiver must match the characteristics of the light driven into the optical fiber.

Unlike the optical emitter where there are multiple technologies used for light generation, all optical receivers are based on the response of a PIN (Positive-Intrinsic-Negative) photodiode. These photodiodes are based on either Silicon or Gallium Arsinide technology. The output of the PIN photodiode is a small (<1 µA) change in current in response to received light. A fiberoptic detector module feeds the output of this PIN photodiode into a transimpedance amplifier. The function of this amplifier is to convert this small change in current into a large change (ECL 100K-level) in voltage.

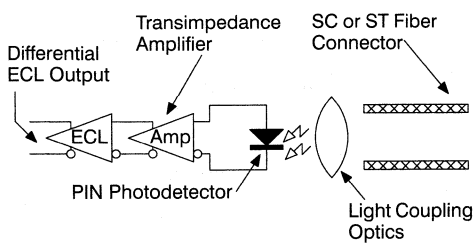


Figure 72. Fiberoptic Detector Module Block Diagram

For many optical receivers, it is possible to operate them above their stated maximum data rate. What is given up is receiver sensitivity; i.e., many 200-Mbit/second optical modules will operate at the ANSI Fibre Channel data 266-Mbit/second data rate, but with a 3-dB or greater loss of sensitivity. This loss may be converted directly into a shorter usable distance on the fiberoptic media.

Because the optical receiver has ECL outputs, care should be taken to maintain a balanced load on any differential outputs to minimize current transients. While some optical receiver outputs (i.e., signal-detect on endfire modules) may be single-ended, they usually do not change very often and should not affect data integrity when they do.

Power Distribution Requirements for Optical Receivers

The power filtering of the optical receiver is quite critical as the transimpedance amplifier must responding to very low current variations. This filtering problem is usually compounded by the placement of the high-noise generating optical transmitter, directly adjacent to the optical receiver.

Depending on the type of receiver, it may be implemented with only one, or many V_{CC} pins. For those made with a single V_{CC} connection, this pin should be isolated through a π-network or other network that implements an inductive leg to block RF on the power lead.

For those optical receiver modules that use multiple V_{CC} pins, these pins are usually kept separate internal to the module, and feed different sections of the logic. For those V_{CC} pins that supply power to the ECL output emitter-followers and the ECL differential amplifiers, all that is necessary is a good 0.1-µF decoupling capacitor next to the V_{CC} power pins. An inductive-based filter is recommended for the V_{CC} pin that provides power to bias the PIN photodiode and the transimpedance amplifier to limit the external noise input from the system supply.

Just as with the transmitter this inductive filter can be implemented either as a notched or slotted power plane, or by using a surface trace to act as an inductor. When implemented in this fashion the capacitor placed at the optical receiver end of the inductor should be 0.1 µF.

Optical Modules

Thanks to the efforts of a group of optical component manufacturers (AMP/Lytel, Siemens Optical, Hewlett-Packard, and Sumitomo Electric), a *de facto* standard footprint has been developed for optical modules. While originally developed for the FDDI market, optical modules with speeds suitable for Fibre Channel and ATM are also available. This footprint specifies the mechanical dimensions and signal names of two different package styles, yet allows a common board layout to accept both. The dimensions and pin numbering of this footprint are illustrated in *Figure 73*.

The two module types supported by this footprint are called DIP and endfire. The DIP modules utilize pins 1–32, while the endfire modules only use pins 33–41 (for signals) and pins 1 and 32 for package mounting. These two mounting pins are also larger in diameter than the other pins on the package.

These optical modules (DIP and endfire) share several signals. For compatibility with both module types, only the smaller set of signals present on the endfire module type should be used. A complete listing of the signals present in the standard footprint is found in *Table 5*. The signals present on the optical module are

- SD — Signal Detect
- TD — Transmit Data
- RD — Receive Data
- Case — Outer Case of Module
- V_{CC} — Positive Supply Voltage
- V_{EE} — Negative Supply Voltage
- V_{BB} — ECL Base Threshold Voltage

The V_{BB} and SD− signals are only present on the DIP footprint package and thus should not be used in designs that wish to support interchangeable module types.

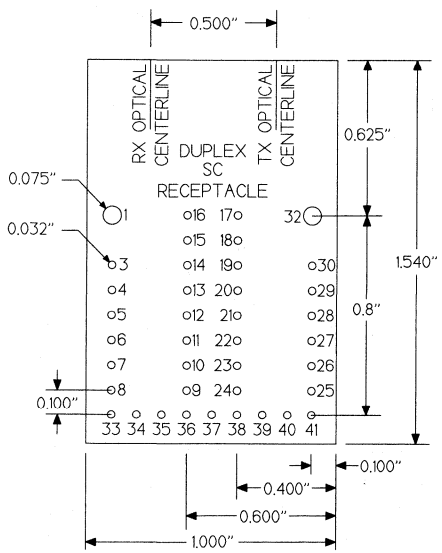


Figure 73. Standard Optical Module Footprint

Table 5. Standard Optical Module Pinout

DIP Pin Assignments			
Pin	**Signal**	**Pin**	**Signal**
1	Case	2	No Pin
3	Case	4	V_{EE}
5	V_{EE}	6	+SD
7	−SD	8	Case
9	Case	10	RD
11	+RD	12	V_{CC}
13	V_{CC}	14	V_{CC}
15	Case	16	Case
17	Case	18	Case
19	V_{CC}	20	V_{CC}
21	Case	22	+TD
23	−TD	24	Case
25	Case	26	V_{BB}
27	Case	28	Case
29	V_{EE}	30	V_{EE}
31	No Pin	32	Case
Endfire Pin Assignments			
Pin	**Signal**	**Pin**	**Signal**
33	V_{EE}	34	+RD
35	−RD	36	+SD
37	V_{CC}	38	V_{CC}
39	−TD	40	+TD
41	V_{EE}		

Care must be used when connecting to the pins marked as Case. These pins are not specified as being isolated, tied to V_{EE}, or tied to V_{CC}. As such, each manufacturer is allowed to connect them as they wish.

Isolated Case pins may be connected either to V_{CC} or V_{EE}. Usually this connection is made to whichever power rail is identified as ground in the system. When used with the HOTLink Transmitter, these types of modules are usually operated in PECL mode with the Case pins connected to V_{EE}.

When the case is connected to the V_{CC} pins, the part is designed for operation in a standard ECL (negative referenced) system. Modules of this type may still be used with HOTLink, but some care must be taken in how they are interfaced.

Pulse Transformers

A pulse transformer is a magnetic device used to couple electrical energy from one stage to another with minimal distortion. This coupling occurs through magnetic induction. How well this coupling occurs is based on the construction of the transformer and the materials used for the core and windings.

Core Materials

There are three basic types of core materials used for transformers: metal, powdered iron, and ferrites. Metal cores consist of pieces of low conductivity metal having some magnetic properties; usually soft iron or steel. This metal core is usually made from multiple strips or laminations of material to limit eddy currents in the core. Metal cores have a practical upper frequency limit of about 50 kHz.

Powdered iron cores use metal powder fused together by an insulating binder. Because of the smaller size of the magnetic particles, the upper frequency for powdered iron cores extends to near 1 MHz.

Ferrites are a magnetic form of ceramic. Depending on the type of ferrite and construction of the core, transformers with ferrite-based cores are available with operating frequencies of near 1 GHz. This is the core material that must be used for transformers used with HOTLink.

ANSI Fibre Channel Specifications

The current ANSI Fibre Channel standard, section 7.1, states that the recommended interface to all types of copper media is via transformer coupling. The primary benefits of transformer coupling are ground isolation, common-mode rejection, and the ability to drive both balanced and unbalanced transmission lines with the same interface (Reference 3).

Just as with optical interfaces, the ANSI standard calls out multiple copper technology options for use at the 25-MByte/second data rate supported by HOTLink. The ANSI designators for these technology options are

- 25-TV-EL-S
- 25-MI-EL-S
- 25-TP-EL-S

These designators are interpreted as four fields. The first field identifies the data rate used (25-MBytes/second).

The second field identifies the media used. TV specifies 75Ω video grade coaxial cable, MI specifies a 75Ω miniature coaxial cable, and TP specifies shielded twisted pair.

The third field identifies the transmitter type. The EL identifier is used for all electrical classes.

The last field identifies the distance class of the link. S specifies short distances (<75m).

While these are the only electrical classes that ANSI supports for Fibre Channel, many other impedances and distances will function with the HOTLink Transmitter and Receiver.

The typical transformer electrical characteristics to support these interface combinations are called out in the ANSI Fiber Channel standard in Section 7.1, Table 10 (Reference 3).

Pulse transformers suitable for coupling HOTLink to copper based cables are available from Pulse Engineering, Mini-Circuits, Premier Magnetics Inc., Valor, and others.

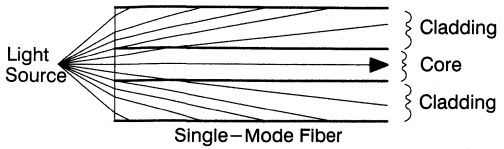


Figure 74. Single-Mode Fiber Propagation

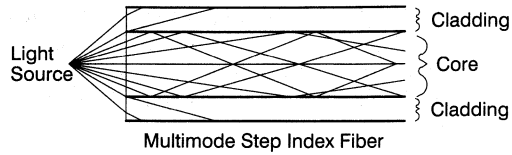


Figure 75. Multimode Step-Index Fiber Propagation

Fiberoptic Cables

Optical media generally falls into two categories: multimode and single-mode. The usage of each type is dictated by the spectral characteristics and launch mode of the light into the fiber.

Single-Mode Fiber

Single-mode fiber is most often used with optical drivers that are both spectrally pure (i.e., a laser) and coherent in their output (well collimated, long-wave laser). Fibers of this type have a very small core section to limit the modes of propagation of the transmitted light, and an index of refraction designed to only allow light to remain in the core that strikes the cladding at a very low critical angle. Its main propagation of light is by refraction (bending) of light that travels down the center of the core. In addition, a small number of tight turns of the fiber are usually placed near the optical transmitter to act as a filter for any of the higher-order modes of propagation that may be launched into the fiber. These turns change the incidence angle of the higher-order modes between the core and the cladding of the fiber, causing light at these modes to leave the core. A diagram of a single-mode fiber is shown in *Figure 74* (Reference 18).

Single-mode fibers are available in different core diameters for use with different optical sources. The fiber type called out for single-mode propagation in the ANSI Fibre Channel standard is 125-µm fiber diameter with a 9-µm core. With this core diameter, the fiber is limited to use with 1300-nm sources (Reference 3).

Multimode Fiber

Multimode fiber is usually used with optical drivers that that are not spectrally pure (i.e., LED) or not coherent in their output (i.e., shortwave lasers). The lensing system used to couple the optical driver's light output to the fiber is not designed for collimation, but to couple the maximum amount of light. This type of fiber allows propagation of light both by refraction and by reflection.

Two distinct classes of multimode fiber are in use today: step-index and graded-index. In a step-index fiber, the primary mode of light propagation is through total internal reflection. Light that enters the core on one end is continuously reflected at the core/cladding interface until it exits the cable at the other end. A diagram of multimode step-index fiber is shown in *Figure 75*.

In a graded-index fiber, light is propagated through refraction rather than reflection. The fiber core is constructed of multiple concentric layers of glass. The index of refraction in each layer is slightly different, getting lower as you move out from the center of the core. Because light travels faster in a lower index of refraction, the higher-order modes of propagation that travel the farthest arrive in phase with the low-order modes that remain near the center of the core. A diagram of a multimode graded-index fiber is shown in *Figure 76* (Reference 18).

The step-index form of multimode fiber is not normally used for data communications because its propagation characteristics limit the usable distance of a link. The ANSI Fibre Channel standard currently only supports graded-index fibers with

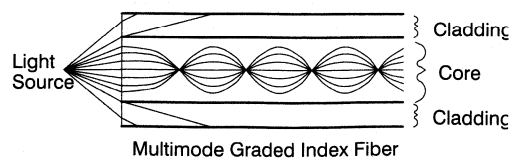


Figure 76. Multimode Graded-Index Fiber Propagation

core diameters of 50 µm or 62.5 µm, both with a cladding diameter of 125 µm (Reference 3).

Optical Pulse Dispersion

In a step-index fiber, light that travels straight through the core covers a shorter distance and arrives at the end of the fiber before light that repeatedly bounces off the core/cladding interface. This difference in delay through the fiber causes a narrow pulse launched into the fiber to widen as it travels down the fiber. Because this pulse widening or dispersion is caused by the different modes of propagation, this phenomena is known as modal dispersion.

When used with an LED driver, an additional source of dispersion comes into play. Unlike free space where all wavelengths of light propagate at the same rate, an optical fiber propagates different wavelengths at different rates. This causes any light pulse that is not spectrally pure (i.e., all the same wavelength) to widen as it travels down the fiber. Pulse widening caused by wavelength is called chromatic dispersion.

With multimode fiber one of the main limits to usable distance is the pulse spreading caused by light dispersion within the fiber. As the transmitted 1s (pulses of light) get wider through dispersion, they interact with adjacent transmitted 0s (absence of light). The effect of dispersion is illustrated in *Figure 77* (Reference 18).

With single-mode fiber, dispersion is usually not a limiting factor. Here the amount of attenuation over distance is the main limiting factor.

ANSI Fibre Channel Optical Fibre Requirements

Fiberoptic cables are available with many different optical and mechanical characteristics. International organizations have set standards for optical cable plants to allow manufactures to standardize on some cable types.

The standards body that created the standards used for optical cable plants is called EIA/TIA (Electronic Industry Association/Telecommunications Industry Association). The governing document for all optical fiber types is EIA/TIA 492BAAA. This includes single-mode and both core diameters of multimode fiber.

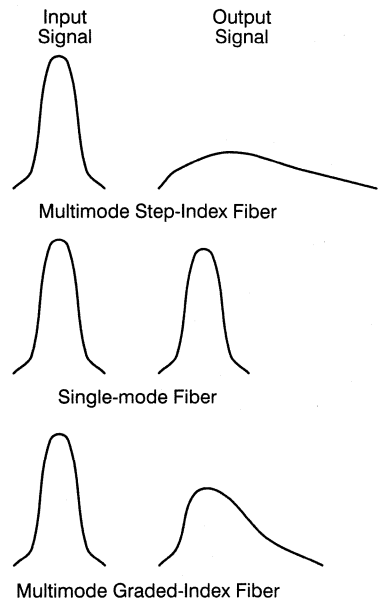


Figure 77. Pulse Dispersion

The ANSI Fibre Channel standard has also selected a common fiberoptic connector type for use with all types of optical fiber media. This connector type was developed by NTT in Japan and is known as an SC-type optical fiber connector. A diagram of a simplex SC connector is shown in *Figure 78*.

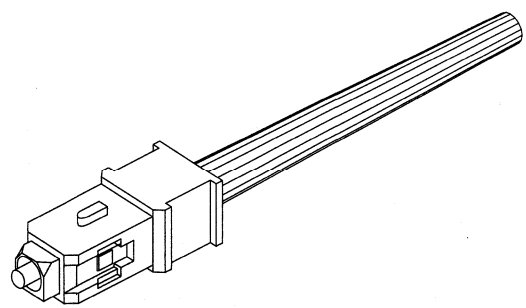


Figure 78. SC Simplex Fiberoptic Connector

These simplex connectors may be joined together using a plastic clip to form a duplex connector. In the duplex configuration the center-line spacing of the optical fibers is 0.5 inch.

Simplex and duplex cable assemblies are available from AMP, FOCS Inc., Alcoa Fujikura Ltd., Belden, and many others.

Copper Cables

There are three primary types of copper media available for distance data transmission: shielded twisted pair (STP), twinaxial cable, and coaxial cable. Each of these cable types has specific advantages and characteristics.

Shielded Twisted Pair

Shielded twisted pair (STP) cables are used for many low cost LAN installations. One of the most common of these is the IBM Type-1 and Type-6 cables used for IEEE 802.5 token ring networks. For use with the ANSI Fibre Channel, the standard calls out Type-1 and Type-2 150Ω STP cables as defined in EIA/TIA 568 (References 3, 11, 20).

STP cables are constructed of two insulated conductors twisted together at a specific number of twists per foot, with an overall shield and jacket. They are available with characteristic impedances of from 78Ω to 200Ω With this type of cable the transmission remains fully differential from source to destination. The shield is only used to prevent radiation and control susceptibility. Cables of this type are effective for long distances at low data rates, and short distances for high data rates. The main limiting factor for cables of this type is their attenuation at high frequencies. In many cases, cables of this type are so poor above 50 MHz that attenuation is not even specified at these frequencies. In some vendors' data, shielded twisted pair cables are also referred to as twinax (Reference 20).

Twinaxial Cable

Twinaxial cable is a shielded form of twin-lead. Twinaxial cables consist of two parallel insulated conductors, maintained at a fixed spacing with an overall shield. Cables of this construction are often used for television reception lead-in cable. As with STP cables, twinaxial cables maintain a fully differential transmission system from transmitter to receiver. Twinaxial cables have lower attenuation of high frequency signals than STP cables and can be used for longer distances.

Unshielded twin-lead, while having excellent high-frequency characteristics, is not generally usable for data communications due to the radiated emissions of the cable, and the impedance changes that occur as the unshielded cable is routed near metallic objects.

Twinax cables are available in impedances from 125Ω to 300Ω and velocities of 70% to 80% (Reference 20).

Coaxial Cable

Coaxial cable is used for the longest distances. They consist of a single center conductor surrounded by a dielectric spacer, surrounded by a concentric shield. Unlike either STP or twinax, coaxial cables are an unbalanced transmission line; i.e., the signal is transmitted and received as a signal relative to a ground or shield, rather than a signal relative to another signal.

In a coaxial cable the outer conductor acts both as part of the transmission line to propagate the signal, and as a shield to prevent radiation of the transmitted signal and susceptibility from outside signals.

Coaxial cables are available in impedances from 50Ω to 125Ω and velocities of 66% to 90%. The main element that affects the velocity of propagation is the dielectric type used between the center conductor and the shield. Solid polyethylene is a common dielectric at the 66% velocity. The fastest speeds usually resort to foamed Teflon or partial air core. *Table 6* lists some common coaxial cable types and characteristics (Reference 20).

One thing that cannot be seen from this table are the cable's attenuation characteristics versus frequency. This is one of the characteristics that determines just how far a usable signal can be sent. The cables listed in *Table 6* are plotted for attenuation in *Figure 79*.

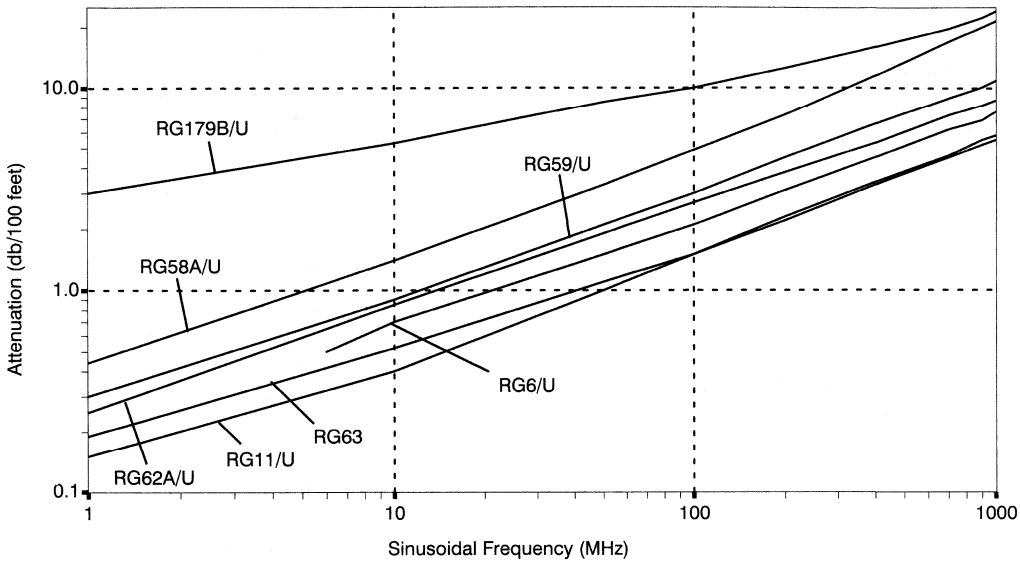


Figure 79. Coaxial Cable Attenuation Characteristics

Table 6. Common Coaxial Cable Types

RG/U Type	Belden Type	Z_O	Nominal O.D.	V_P
RG58A/U	8259	50	.193″	66%
RG179B/U	83264	75	.1″	70%
RG6/U	1223A	75	.290″	83%
RG59/U	9259	75	.242″	78%
RG11/U	87292	75	.348″	82%
RG62A/U	9268	93	.242″	84%
RG63	9857	125	.405″	84%

ANSI Fibre Channel Copper Cable Requirements

The ANSI cable plant requires copper cables with specific operating characteristics. These characteristics are called out in Section 9 and Annex F of the Fibre Channel PC-PH Rev 4.1 document (Reference 3).

Realizing these requirements means that the cable must be made with specific construction. For coaxial cables the V_P of 70% to 82% requires a foam dielectric.

The minimum necessary shield coverage for braid is 95%. This is necessary because of the high frequencies carried by the cables. With shield coverage lower than this, the signal leakage through the braid can allow not only significant signal radiation, but an impedance mismatch due to signal propagation down the outer surface of the braid. For best effectiveness, a 100% foil shield should be used in addition to the braid shield.

To meet flammability requirements, the National Electrical Code now requires that almost all installations use either CL2 or CL2P (plenum rated) jacket material (Reference 25).

Cables meeting all of these requirements are available from multiple vendors.

The ANSI standard also allows use of shielded twisted pair or twinaxial type cables. These cables all require a shield to meet EMI/EMC requirements. Unshielded twisted pair (used for many net-

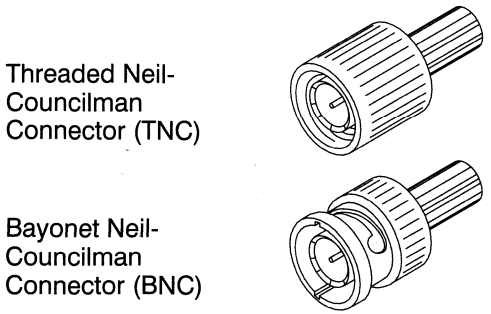


Figure 80. TNC/BNC Cable Connectors

works) should not be used. This is primarily due to radiated emissions rather than susceptibility.

Copper Cable Connectors

There are three primary connector types called out for use with copper cables: BNC and TNC for coaxial cables (illustrated in *Figure 80*) and a 9-pin D-sub (illustrated in *Figure 81*) for twisted−pair/twinax cables.

For coaxial cables, the BNC connectors are used on the transmitting end of the cable while the TNC connectors are used on the receiver end of the cable. This dual connector configuration allows a duplex cable to be connected without having to identify one cable from the other. With these connectors the

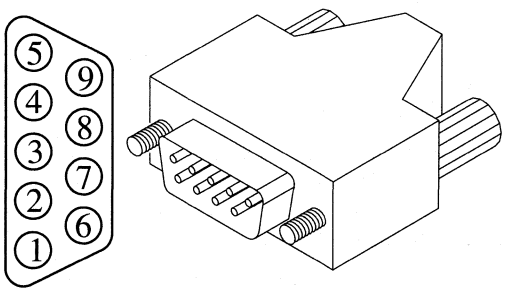


Figure 81. STP Cable Connector and Connector Pinout

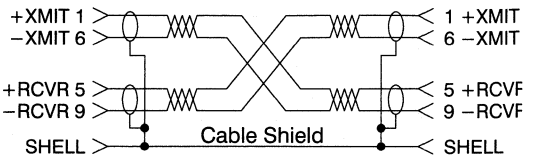


Figure 82. STP Cable Connections

male end is always on the cable while the female end is used at the board bulkhead.

For twisted-pair or twinaxial type cables a 9-pin D-sub connector is used. This connector is required to have a metal shell because the shields of both the transmit and receive pairs are terminated to the shell of the connector. As with the coaxial connectors the cable gets the male connector while the board or bulkhead gets the female connector.

The STP cable is wired in a crossover fashion where the transmit pins at one end of the cable (as illustrated in *Figure 82*) are connected to the receive pins at the other end of the cable. The cable shields for both pairs are tied together and connected to the D-sub connector shell at each end.

Because of the low current used in these cables, the connections are considered to be dry circuits. To prevent contact oxidation from degrading the link over time the contacts are required to be Gold or Palladium plated (Reference 28).

Conclusion

The HOTLink family of communications products provide designers with a simple yet elegant method of reliably moving large quantities of data at very high speeds from one place to another. These parts are capable of communicating over copper or optical media at distances well in excess of industry standards. Their BiCMOS implementation, along with their integrated power saving features, combine to offer one of the lowest-power high-speed serial communications link standards available.

IBM is a registered trademark of International Business Machines
ESCON is a trademark of International Business Machines
HOTLink is a trademark of Cypress Semiconductor

References

1. A. X. Widmer and P. A. Franazek, *A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code*, IBM Journal of Research and Development, 27, No. 5: 440-451, September 1983
2. U.S. Patent 4,486,739, Peter A. Franaszek and Albert X Widmer, *Byte Oriented DC Balanced (0,4) 8B/10B Partitioned Block Transmission Code*, December 4, 1984
3. *Fibre Channel Draft Standard, dpANS X3.230-1993*, American National Standards Instituite, 1993
4. *Enterprise System Architecture/390 ESCON I/O*, SA22-7202, IBM Corporation, 1990
5. *F100K ECL Logic Databook and Design Guide*, National Semiconductor, 1990
6. Lawrence B. Levit and Marco L. Vincelli, *Characterize High-speed Digital Circuits: A Job For Wideband Scopes*, Lecroy Corp., EDN June 10, 1993
7. *The ABC's of Probes*, Tektronix Pub 60W-6053-3
8. *Product Overview*, Cascade Microtech, Product Overview, 1992
9. *Safe Use of Lasers ANS Z136.1-1993*, American National Standards Institute, 1993
10. *Laser Safety in Optical Communication Systems ANS Z136.2*, American National Standards Institute
11. *Commercial Building Telecommunications Wiring Standard EIA/TIA-568*, Electronics Industries Association/Telecommunications Industries Association
12. F.D.A. Regulation 21, Code of Federal Regulations
13. *IEC825*, International Electrotechnical Committee
14. Scott, Paul, Cypress Semiconductor, *Draft Paper — HOTLink On Wire*, May 1992
15. *1990–91 Resistor/Capacitor Data Book*, Philips Components, 1990
16. *System Design Considerations When Using Cypress CMOS Circuits*, Cypress Semiconductor Applications Handbook, 1993
17. White, Donald R.J., *Electrical Filters, Synthesis, Design and Applications*, Second Edition, 1980
18. Sterling, Donald J.Jr., *Technician's Guide To Fiber Optics*, Second Edition, 1993
19. Blood Jr., William R., *MECL System Design HandBook*, Fourth Edition, 1988
20. *Belden Wire and Cable*, Cooper Industries, 1990
21. Botos, Bob, Hewlett Packard, *Designers Guide to RCL Measurements*, June 1979
22. *Hewlett-Packard Test and Measurement Catalog*, Hewlett-Packard Corp., 1992
23. *Crystal Oscillator Handbook and Catalog*, Vectron Laboratories, Inc., 1992
24. Ramierez, Robert W., *The FFT, Fundamentals and Concepts*, Tektronix, Inc. 1985
25. *National Electrical Code*, National Fire Protection Association
26. *1990–91 Resistor/Capacitor Data Book*, Philips Components, 1990
27. *Application Note 65074, Fiber-Optic Transmitter and Receiver*, AMP Incorporated, 1992
28. J. H. Whitley, AMP Inc., *Contacts and Dry Circuits*, AMP Symposium paper, October 1963

Support Component Vendors

ECL Logic

Motorola
Logic Integrated Circuits Division

National Semiconductor
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, CA 95052–8090
408–721–5000

Oscillators

Connor Winfield Corp.
1865 Selmarten Road
Aurora, IL 60505
708–851–4722

CTS Corporation
400 Reimann Avenue
Sandwich, IL 60548
815–786–8411

Fox Electronics
5570 Enterprise Parkway
Ft. Meyers, FL 33905
813–693–0099

MF Electronics
10 Commerce Drive
New Rochelle, NY 10801
914–576–6570

Murata Erie North America
2200 Lake Park Drive
Smyrna, GA 30080
800–831–9172

Vectron Laboratories Inc.
166 Glover Avenue
P.O. Box 5160
Norwalk, CT 06856–5160
203–853–4433

Wideband Transformers

Mini–Circuits
P.O. Box 350166
Brooklyn, NY 11235–0003
718–934–4500

Premier Magnetics
27111 Aliso Creek Road, STE175
Aliso Viejo, CA 92656
714–362–4211

Pulse Engineering
12220 World Trade Drive
San Diego, CA 92118
619–674–8100

Toko America
1250 Feehanville Drive
Mt. Prospect, IL 60056
708–297–0070

Valor Electronics, Inc.
9715 Business Park Ave.
San Diego, CA 92131–1642
619–537–2619

Optical Modules

AMP/Lytel
Branchburg Corporate Center
61 Chubb Way
P.O. Box 1300
Somerville, NJ 08876
908–685–2000

AT&T Microelectronics
Dept. 52AL040420
555 Union Boulevard
Allentown, PA 18103
800–372–2447

Hewlett Packard
Components Division
370 West Trimble Road
San Jose, CA 95131

Siemens Fiber Optic Components
20F Commerce Way
Totowa, NJ 07512
201–890–1606

Sumitomo Electric
Fiber Optics Corporation
777 Old Sawmill River Road
Tarrytown, NY 10591–6725
914–347–3770

Test Equipment

Hewlett Packard
Test & Measurement Division
5301 Stevens Creek Blvd.
P.O. Box 58059
Santa Clara, CA 95052–8059
800–452–4844

LeCroy Corporation
700 Chestnut Ridge Road
Chestnut Ridge, NY 10977–6499
914–425–2000

Tektronix, Inc.
P.O. Box 500
Beaverton, OR 97077–0001
800–835–9433

Copper Cables

Cooper Industries
Belden Division
P.O. Box 1980
Richmond, IN 47375
317–983–5200

Madison Cable Corporation
125 Goddard Memorial Drive
Box 3000
Worcester, MA 01603
508–752–7320

W.L. Gore & Associates, Inc.
1901 Barksdale Road
Newark, DE 19711
800–356–4622

Optical Cables

3M Fiber Optic Products
610 Industrial Way
Eatontown, NJ 07724
908–544–9119

Alcoa Fujikura Ltd.
Telecommunications Division
P.O. Box 5631
Spartanburg, SC 29304
800–866–3953

Amphenol Fiber Optic Products
1925 A Ohio Street
Lisle, IL 60532

Fiber Specialists, Inc.
485 West Armory Dr., STE C
South Holland, IL 60473
708–339–2500

FOCS
495 Technology Center West
50 D'Angelo Drive
Marlborough, MA 01752
508–480–9600

Gould, Inc.
Fiber Optics Division
6740 Baymeadow Drive
Glen Burnie, MD 21060
800–544–6853

IBM Corporation
Optical Link Products
1701 North Street – Dept. V06, Bldg. 032–2
Endicott, NY 13760
607–755–9935

M/A–COM LCS
6 Wentworth Drive
Hudson, NH 03051
800–669–1769

Methode Electronics, Inc.
Fiber Optic Products Division
7444 West Wilson Avenue
Chicago, IL 60656
800–323–6858

Optical Cable Corporation
P.O. Box 11967
Roanoke, VA 24022–1967
703–265–0690

Siecor Corporation
489 Siecor Park
Hickory, NC 28603–0489
800–743–2674

Sieko Instruments
Electronic Components Division
2990 West Lomita Blvd.
Torrence, CA 90505
213–517–7786

Siemens Fiber Optic Components
20F Commerce Way
Totowa, NJ 07512
201–890–1606

Sumitomo Electric
Fiber Optics Corp.
78 Alexander Drive
Research Triangle Park, NC 27709
800–358–7378

Resistors

AVX Corporation
801 17th Avenue South
Myrtle Beach, SC, 29577
803–448–9411

Dale Electronics, Inc.
East Highway 50
P.O. Box 180
Yankton, SD 57078–0180
605–665–9301

IRC
P.O. Box 1860
Boone, NC 28607
800–472–6467

Philips Components
6071 St. Andrews Road
Columbia, SC 29212–3198
803–772–2500

SMEC
12710 Research Blvd., STE285
Austin, TX 78759
800–336–7632

State of the Art
2470 Fox Hill Road
State College, PA 16803–1797
800–458–3401

Capacitors

AVX Corporation
801 17th Avenue South
Myrtle Beach, SC, 29577
803–448–9411

Murata Erie North America
2200 Lake Park Drive
Smyrna, GA 30080
800–831–9172

Philips Components
6071 St. Andrews Road
Columbia, SC 29212–3198
803–772–2500

SMEC
12710 Research Blvd., STE285
Austin, TX 78759
800–336–7632

TDK Corporation of America
1600 Freehanville Drive
Mount Prospect, IL 60056
708–803–6100

Tokin America Inc.
155 Nicholson Lane
San Jose, CA 95134
408–432–8020

Inductors

AVX Corporation
P.O. Box 867
801 17th Avenue South
Myrtle Beach, SC 29577
803–448–9411

Coilcraft
1102 Silver Lake Road
Cary, IL 60013
800–332–2645

Dale Electronics, Inc.
East Highway 50
P.O. Box 180
Yankton, SD 57078–0180
605–665–9301

Murata Erie North America
2200 Lake Park Drive
Smyrna, GA 30080
800–831–9172

SMEC
12710 Research Blvd., STE285
Austin, TX 78759
800–336–7632

Sprague–Goodman Electronics, Inc.
134 Fulton Avenue
Garden City Park, NY 11040–5395
516–746–1385

TDK Corporation of America
1600 Freehanville Drive
Mount Prospect, IL 60056
708–803–6100

Toko America
1250 Feehanville Drive
Mt. Prospect, IL 60056
708–297–0070

RF Connectors

AMP Inc.
Harrisburg, PA 17105–3608
800–522–6752

Document #: COM0003–1193

Amphenol
RF/Microwave Operations
One Kennedy Avenue
Danbury, CT 06810
800–625–7100

E.F. Johnson Company
299 Johnson Avenue
P.O. Box 1249
Waseca, MN 56093–0514
800–247–8256

Pasternack Enterprises
Coaxial Products Division
P.O. Box 16759
Irvine, CA 92713
714–261–1920

Trompeter Electronics, Inc.
31186 La Baya Drive
Westlake Village, CA 91362
800–982–2629

HOTLink™ Built-In Self-Test (BIST)

The Cypress CY7B923 and CY7B933 HOTLink™ Transmitter and Receiver offer the system integrator a tool to send data from place to place over a high-speed serial transmission link. They have been designed for easy integration into any data communication subsystem and have a data-interface equivalent to that of a data register or FIFO memory. *Figure 1* shows schematically where HOTLink might fit into a typical system. In this example, data is being sent from one data bus to another over a serial link. The controllers used at each end of this representative data link show the generic functions of all data-link controllers. The protocol controller might be simply a PLD state machine, or a VLSI protocol-specific subsystem. The Bus Interface might be an octal register, or a standard CPU bus controller. The data buffer might be a register in the bus interface or a packet-sized FIFO memory. HOTLink connects these typical logical building-blocks with a high-performance serial data link.

Serial data links are notoriously difficult to design and test. HOTLink simplifies the design problem, and offers the system integrator and end user a simple method to test the finished link. The Built-In Self-Test feature described in this application note

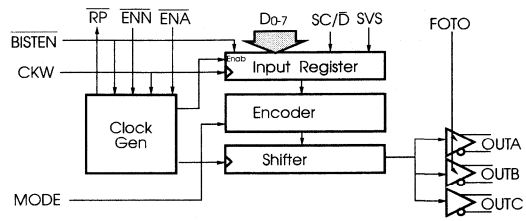


Figure 2. HOTLink Transmitter Block Diagram

gives an unambiguous, real-time, offline test of the entire link. Several ways to test serial links, and serial link components are described using BIST as the grading tool.

The Cypress CY7B923 and CY7B933 HOTLink Transmitter and Receiver block diagrams in *Figures 2* and *3* show the built-in functionality included in these parts. The transmitter uses a fully integrated PLL to multiply the user's byte-rate clock for use as a serial data rate clock. The receiver has a fully integrated PLL that tracks the serial data stream and recovers the bit-rate clock. The bit rate clock is used for internal data decoding and generating the parallel byte-rate clock aligned to the recovered data.

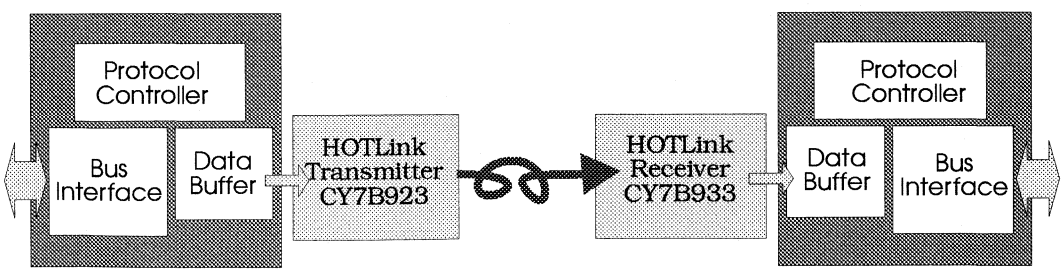


Figure 1. Typical Data-Communication Link using HOTLink

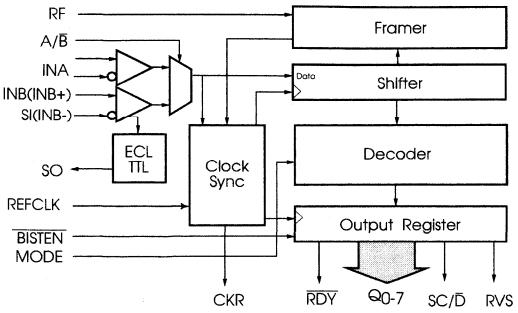


Figure 3. HOTLink Receiver Block Diagram

Both parts have built-in 8B/10B code converters that conform to ANSI X3T9.3 Fibre Channel and ESCON™ specifications. The encoder and decoder transform 8-bit user data to 10-bit characters more suitable for transmission systems built with fiber-optic cables or any common wire transmission line. Both parts have parallel I/O registers designed to interface with standard FIFO memories without any extra logic. These parts also offer a full-function Built-In Self-Test feature that tests all the circuitry in both ICs and all of the interconnect components that make up the link.

In most traditional serial interface links, the testing of the serial link components and interconnections was done off-line using high-speed analog test equipment and specialized parameter analyzers. If any in-line testing was done, it was done as part of some particular communication protocol or higher-level software functionality. In-line testing usually consisted of only a simple loop-back handshake test that often failed to isolate the problem (if any) to any specific component or connection. And, these simple tests were so superficial that they typically failed to indicate ANY problem at all.

HOTLink products offer the capability to do in-line testing of the entire serial link using without adding or removing components from an operational system. The overhead associated with this capability is minimal, and doesn't affect any of the critical analog or high speed interfaces.

Figure 4 shows a link interconnect that is typical for HOTLink systems and illustrates the interface signals used for the BIST operation. This configuration is identical to that of a normally operating data transmission link, except for the simple control features that have been added to facilitate the test function. While the form of the transmission link shown in *Figure 4* illustrates a test being run across an operational communication link, similar testing can be done in either Local-Loop-Back or Long-Loop-Back configurations as shown in *Figures 5* and *6*.

Since there is no required feedback connections between the receiver and transmitter, the BIST function can be used in any configuration that the system designer might find desirable. The previous example (*Figure 4*) illustrates a cooperative test between the two ends of a point-to-point transmission system. The same function can be accomplished as part of the loop-back test of a single communicating node. *Figure 5* shows a typical Local Loop-Back connection wherein the test is run without any involvement of the remote station.

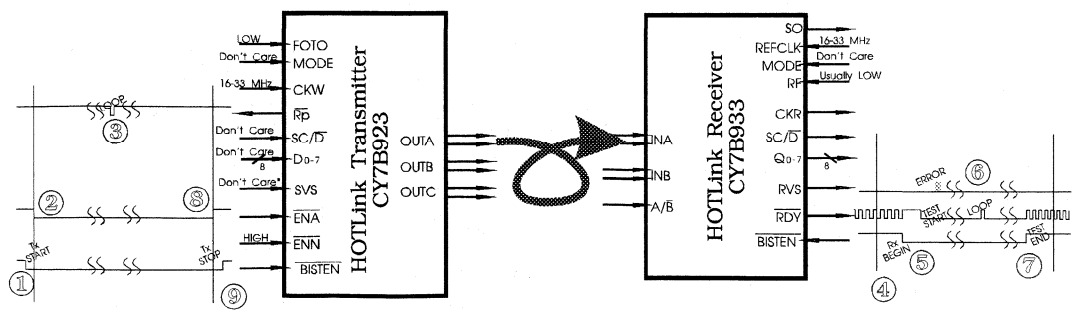


Figure 4. HOTLink BIST Connections

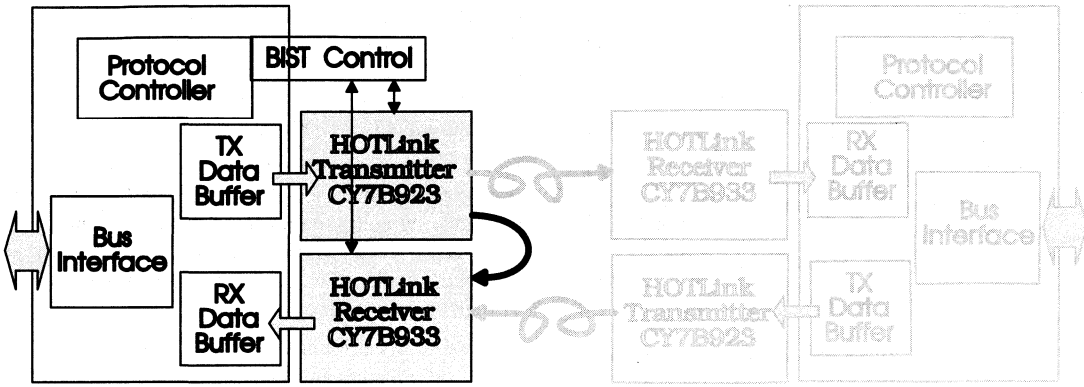


Figure 5. Local Loop Back Testing

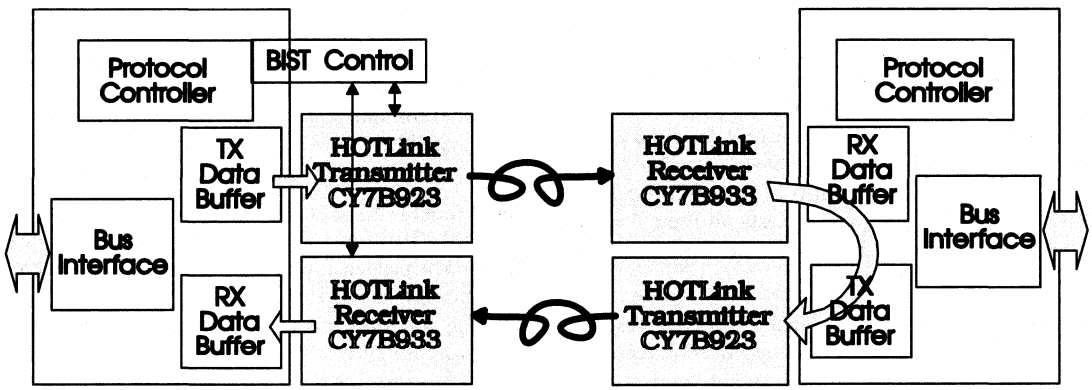


Figure 6. Long Loop Back Testing

A more complete test can be made if it is possible to gain the cooperation of the station at the remote end of the communication link. In the example shown in *Figure 6*, the remote station becomes a simple flow-through repeater. This connection is typical of many network systems, and can implemented in many simple and obvious ways. This test takes advantage of the orthogonality built into HOTLink products. The data patterns that emerge from the parallel data outputs of the receiver will send equivalent codes when put into the parallel data inputs of the transmitter. This is true in either Bypass Mode or Encoded Mode. The only restriction is that the receiver and transmitter, which are connected via the parallel data path, must both be in the same mode. In the illustration shown in *Figure 6*, the pair of HOTLink chips on the left should be in the same mode, and the pair on the right should be in the same mode. Each pair could be in either Bypass or Encoded mode.

These BIST specific features do not affect the normal message transaction function of the HOTLink's host system. *Figure 7* shows the logical functions that would be added to the protocol controller to support BIST. The timer shown in both the transmit and receiver BIST control functions might be an analog timer (one-shot) or digital logic that counts BIST-Loops. It could also be simply a user-controlled time-out that performs BIST while a TEST button is pressed.

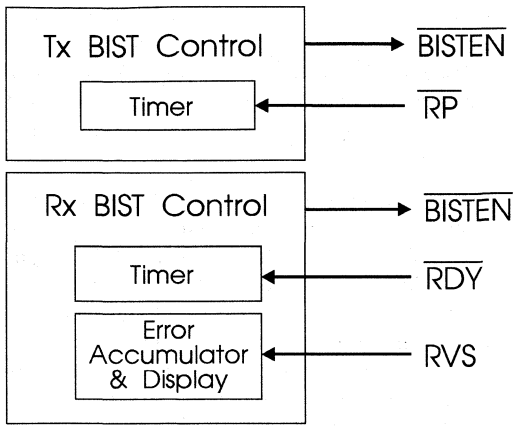


Figure 7. Control Functions for Built-In Self-Test

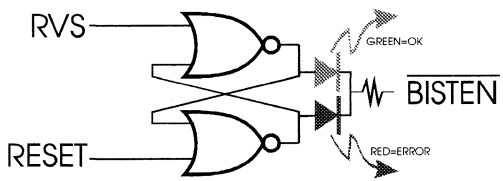


Figure 8. A Simple BIST Error Indicator

The error accumulator and display shown in the example could be as simple or as complex as desired. For a software supported BIST function, the error-count accumulator would be machine readable. For example, the CY9266 HOTLink evaluation card provides a two-digit counter that displays the accumulated error count, but in general any indicator will work. *Figure 8* shows a simple RS latch connected to trap errors and display any rare and random RVS indication.

The simple circuit shown above provides an unambiguous indication of errors detected by the BIST comparator and indicated by the RVS output of the HOTLink Receiver. If the $\overline{\text{BISTEN}}$ control is LOW, the LED output indicates the state of the R-S latch that records any pulses on RVS. After $\overline{\text{BISTEN}}$ is set LOW, and the circuit is RESET, the GREEN-LED is illuminated indicating no detected errors. If RVS ever pulses HIGH, then the RED-LED will light and stay illuminated until RESET is pulsed HIGH (probably by a momentary push-button switch or a control signal from the controller). When $\overline{\text{BISTEN}}$ is set HIGH, the LEDs will be extinguished.

The four areas that must be addressed to build BIST functionality into a system involve control of $\overline{\text{BISTEN}}$ pins, selection of transmitter enable pins ($\overline{\text{ENA}}$ or $\overline{\text{ENN}}$), test-progress supervision and error reporting.

The $\overline{\text{BISTEN}}$ control pins enable only the BIST function logic and can be externally controlled without affecting any system resources or timing. The BIST function performs its test sequence automatically, regardless of the encoding mode selected for the operational data link. No consideration needs to be made for this aspect of a HOTLink system.

The HOTLink Transmitter includes two separate enable inputs. The $\overline{\text{ENA}}$ input is used for systems that simultaneously present data and its validation signal (i.e., traditional controllers, and asynchronous FIFOs). The $\overline{\text{ENN}}$ input is used for systems that have a request function and expect the data to be presented on the next clock cycle (i.e., typical of read-request controllers and clocked FIFOs). The system integrator selects the appropriate enable pin to accommodate system functionality and timing. The other enable can be used to control BIST functionality, thus reducing the burden of additional control logic. When $\overline{\text{BISTEN}}$ is enabled (LOW) and one of the enables (either $\overline{\text{ENA}}$ or $\overline{\text{ENN}}$) is asserted (LOW) the HOTLink Transmitter creates a continuous 511 byte (2^9-1 bytes) pseudo-random stream of 8B/10B-encoded data-patterns, which the HOTLink Receiver checks byte-by-byte. The 256 data patterns are sent once each and the 12 special characters and the 4 error codes are sent sixteen times each (except C0.0 which is sent only 15 times) for a total of another 255 data patterns. For a complete list of codes used in the 8B/10B encoder and the special character and error codes, see the CY7B923/933 HOTLink Transmitter/Receiver datasheet.

The system control logic can monitor the test duration at the transmitter by observing the $\overline{RP}$ pin output. This pin is normally used as the byte-rate read-pulse for asynchronous FIFOs, but during the BIST function $\overline{RP}$ will pulse once per loop. By using a counter or timer that observes $\overline{RP}$ while $\overline{BISTEN}$ is LOW, the number of transmitter test-loops can be monitored. (See timing illustration in *Figure 4*, times 1, 2, 3, 8 and 9.)

The $\overline{RDY}$ output of the receiver serves a similar function as $\overline{RP}$ in the transmitter. In normal modes this output is a status indicator used to control the data flow to an external controller or FIFO. $\overline{RDY}$ pulses once per byte in Encoded Mode, and once per SYNC in Bypass Mode. In BIST Mode, it will rest HIGH while the receiver awaits the start of the BIST sequence, and then rest LOW for all but one byte-time of the BIST loop. A counter or timer similar to that described for the transmitter can be used to count when $\overline{RDY}$ pulses HIGH, recording the number of times that the receiver has executed the BIST loop. (See timing illustration in *Figure 4*, times 4, 5, 6 and 7.)

If errors are discovered in the received sequence, received running disparity or received transmission codes, they are flagged by the RVS output. (See timing illustration in *Figure 4*, time 6.) In Encoded Mode this output is not used as part of the communication data path. In Bypass Mode this output serves as the Qj output, but in either case the external BIST control engine will use RVS as an input with a minimal impact on the rest of the operating system.

Transmitter BIST Generator

The BIST generator in the transmitter is the Input register reconfigured into a nine-bit Linear Feedback Shift register (LFSR). In this configuration it makes every possible combination of nine bits (minus 1). The 256 data codes are sent once each, and the 12 special characters and 4 error codes shown in the datasheet are sent multiple times to complete the 511 byte pattern. This is the pattern that is sent regardless of the HOTLink Transmitter and Receiver encoding mode being used by the system. If the system is using the Bypass Mode (MODE input = V_{CC}) the Tx-Encoder and the Rx-Decoder is enabled upon the beginning of BIST Mode ($\overline{BISTEN}$ = LOW and $\overline{ENA}$ or $\overline{ENN}$ = LOW). When BIST Mode ends, the encoder and decoder are bypassed (if MODE = V_{CC}) and user data is directly serialized and deserialized.

Table 1 is a complete list of the codes sent by the transmitter when it has been enabled (either $\overline{ENA}$ or $\overline{ENN}$ held LOW) in BIST Mode ($\overline{BISTEN}$ input LOW). This sequence is a continuously repeating data loop that may start at any point and then continue through all of the codes. There are several places in the sequence where running disparity is explicitly forced, and as a result, each pass through the sequence will repeat exactly. (It is possible to precondition the BIST generator so that, on the first time through, the initial few codes will be sent using the wrong running disparity.) Running disparity will be corrected when one of the force codes (i.e., C1.7 or C2.7) is encountered and will certainly be correct before the next "Start of BIST".

Once per loop (synchronous with the D0.0 at the beginning of the table), $\overline{RP}$ will pulse (see *Figure 4*, time 3). This BIST specific behavior allows an external system monitor state machine to count the number of times that the transmitter has sent its BIST data. In normal non-BIST operation, $\overline{RP}$ pulses once per byte when $\overline{ENA}$ is asserted, and doesn't pulse at all otherwise.

Table 1 shows the transmitter BIST sequence expressed in codes that would be commonly sent by a controller connected to the transmitter input pins. This table would be used if a controller was programmed to send the BIST pattern as an external system function or as a test of the BIST pattern checker in the receiver.

In normal operation it is unnecessary for the user to initialize the transmitter BIST sequence since the receiver automatically aligns its code generator to the pattern being sent by the transmitter. If there is some ambiguity in the exact start of the transmitter BIST loop, the receiver will patiently wait for a good start and begin checking from there. If an erroneous code causes the receiver to make a false start, it will soon detect the error and resume waiting for the proper starting code.

Table 1. HOTLink Transmitter BIST Sequence

Start here ---->															
D0.0	C0.0	C4.0	C1.7	C4.7	C0.7	D27.7	D23.3-->	D29.1	D30.0	C0.7	D15.4	D11.2	D3.1	D17.0	D4.0
C8.0	C6.0	C2.7	D14.7	C11.0	D19.5	D21.2	D12.1	C10.0	C3.0	D5.6	D8.3	C2.0	C1.0	D0.6	C0.0
C4.0	C8.0	C2.0	C5.0	D24.7	C6.0	C9.0	D18.6	C5.0	D28.5	C4.7	C11.0	D23.6	D13.3	D26.1	C7.0
D9.4	D2.2	C1.0	D20.4	C1.7	C4.7	C0.7	D11.7	D19.3	D21.1	D28.0	C4.7	C0.7	D15.6	D11.3	D19.1
D21.0	D12.0	C10.0	C7.0	D13.6	D10.3	C3.0	D17.4	D4.2	C8.0	C6.0	C9.0	D6.7	C9.0	D18.5	C5.0
D8.5	C2.0	C1.0	D20.6	C1.7	C4.7	C11.0	D3.7	D17.3	D20.1	C1.7	C10.0	C7.0	D13.7	D26.3	C7.0
D25.4	D6.2	C9.0	D22.4	C2.7	D14.5	C11.0	D3.5	D17.2	D4.1	C8.0	C2.0	C5.0	D12.7	C10.0	C3.0
D17.6	D4.3	C8.0	C2.0	C1.0	D4.7	C8.0	C2.0	C1.0	D0.7	C0.0	C0.0	C0.0	C0.0	C4.0	C8.0
C2.0	***C1.0***	D16.7	C4.0	C8.0	C2.0	C1.0	D20.7	C1.7	C10.0	C3.0	D1.7	D16.3	C4.0	C8.0	C2.0
C5.0	D28.7	C4.7	C11.0	D19.6	D5.3	D24.1	C6.0	C9.0	D6.6	C9.0	D22.5	C2.7	D10.5	C3.0	D1.5
D16.2	C4.0	C1.7	C10.0	C7.0	D29.7	D30.3	C0.7	D27.4	D7.2	D9.1	D18.0	C5.0	D12.4	C10.0	C7.0
D29.6	D14.3	C11.0	D19.4	D5.2	D8.1	C2.0	C1.0	D4.6	C8.0	C6.0	C9.0	D2.7	C1.0	D16.5	C4.0
C8.0	C6.0	C9.0	D22.7	C2.7	D26.5	C7.0	D9.5	D18.2	C5.0	D28.4	C4.7	C0.7	D31.6	D15.3	D27.1
D23.0	D13.0	D10.0	C3.0	D5.4	D8.2	C2.0	C5.0	D8.6	C2.0	C5.0	D24.6	C6.0	C2.7	D26.6	C7.0
D29.5	D30.2	C0.7	D31.4	D15.2	D11.1	D19.0	D5.0	D8.0	C2.0	C5.0	D12.6	C10.0	C7.0	D25.6	D6.3
C9.0	D18.4	C5.0	D12.5	C10.0	C3.0	D21.6	D12.3	C10.0	C3.0	D1.6	D0.3	C0.0	C0.0	C0.0	C4.0
C1.7	C10.0	C3.0	D17.7	D20.3	C1.7	C10.0	C3.0	D5.7	D24.3	C6.0	C9.0	D2.6	C1.0	D20.5	C1.7
C10.0	C7.0	D9.7	D18.3	C5.0	D24.4	C6.0	C2.7	D30.6	C0.7	D31.5	D31.2	D15.1	D27.0	D7.0	D9.0
D2.0	C1.0	D4.4	C8.0	C6.0	C2.7	D10.7	C3.0	D17.5	D20.2	C1.7	C4.7	C11.0	D7.7	D25.3	D22.1
C2.7	D10.4	C3.0	D5.5	D24.2	C6.0	C2.7	D10.6	C3.0	D21.5	D28.2	C4.7	C0.7	D11.6	D3.3	D17.1
D20.0	C1.7	C4.7	C0.7	D15.7	D27.3	D23.1	D29.0	D14.0	C11.0	D7.4	D9.2	D2.1	C1.0	D0.4	C0.0
C4.0	C1.7	C10.0	C7.0	D25.7	D22.3	C2.7	D26.4	C7.0	D13.5	D26.2	C7.0	D29.4	D14.2	C11.0	D23.4
D13.2	D10.1	C3.0	D1.4	D0.2	C0.0	C4.0	C8.0	C6.0	C2.7	D26.7	C7.0	D25.5	D22.2	C2.7	D30.4
C0.7	D15.5	D27.2	D7.1	D25.0	D6.0	C9.0	D6.4	C9.0	D6.5	C9.0	D2.5	C1.0	D0.5	C0.0	C0.0
C4.0	C8.0	C6.0	C9.0	D18.7	C5.0	D24.5	C6.0	C9.0	D22.6	C2.7	D30.5	C0.7	D11.5	D19.2	D5.1
D24.0	C6.0	C2.7	D14.6	C11.0	D23.5	D29.2	D14.1	C11.0	D3.4	D1.2	D0.1	C0.0	C0.0	C4.0	C1.7
C4.7	C11.0	D19.7	D21.3	D28.1	C4.7	C11.0	D7.6	D9.3	D18.1	C5.0	D8.4	C2.0	C5.0	D28.6	C4.7
C0.7	D27.6	D7.3	D25.1	D22.0	C2.7	D14.4	C11.0	D7.5	D25.2	D6.1	C9.0	D2.4	C1.0	D4.5	C8.0
C2.0	C5.0	D8.7	C2.0	C1.0	D16.6	C4.0	C1.7	C10.0	C3.0	D21.7	D28.3	C4.7	C11.0	D3.6	D1.3
D16.1	C4.0	C8.0	C6.0	C2.7	D30.7	C0.7	D27.5	D23.2	D13.1	D26.0	C7.0	D13.4	D10.2	C3.0	D21.4
D12.2	C10.0	C7.0	D9.6	D2.3	C1.0	D16.4	C4.0	C1.7	C4.7	C11.0	D23.7	D29.3	D30.1	C0.7	D11.4
D3.2	D1.1	D16.0	C4.0	C1.7	C4.7	C0.7	D31.7	D31.3	D31.1	D31.0	D15.0	D11.0	D3.0	D1.0	GOTO Start

If the user wants to initialize the BIST pattern, there are two methods available. In the Encoded Mode (MODE input=GND) the SVS input overrides the BIST sequence and forces the transmitter to send the code indicating a Code Rule Violation (see the CY7B923/933 HOTLink datasheet for a complete list of 8B/10B data codes and special characters). It also resets the BIST LFSR to its initial state (D0.0). (Running disparity is not explicitly set by SVS, and the first few bytes after its release may be sent with the wrong disparity. The fourth byte of the sequence, C1.7, will explicitly set running disparity for the rest of the patterns.)

Alternatively, the transmitter BIST LFSR can be forced to start its pattern from any other point in the sequence by noting that the BIST sequence proceeds from the code that was in the Input register when BIST was asserted. (Note that the BIST sequence generator state sequence is expressed in En-

coded-Mode terms. If the transmitter is in Bypass Mode, the Next State can be deciphered by converting the actual bit pattern on the inputs to the code that the transmitter would send if it were in the Encoded Mode and that were its input pattern, and then looking for that code in the table.) In most cases this will be sufficient to initialize the sequence, except for the state of the internal running disparity flip-flop. If running disparity must also be assured, the codes for +K28.5 (C2.7) or −K28.5 (C1.7) should be used to initialize the LFSR. The C2.7 and C1.7 starting locations are shown in *Table 1* (row 9, items 1 and 2 respectively). The user would put, for example, C2.7 on the transmitter inputs for one or more byte times, then start the BIST test. The pattern would start from row 9, item 1 in this example. This technique can be useful if the user wants only a portion of the transmitter BIST pattern for some oscilloscope test.

Receiver BIST Comparator

The BIST generator in the receiver is the Output Register reconfigured into a nine-bit Linear-Feedback-Shift-Register (LFSR) that exactly matches the one in the transmitter. In this configuration it puts all possible combinations of nine bits on the receiver outputs (Q0−7 and SC/$\overline{D}$). *Table 2* is a complete list of the codes that must appear on the receiver serial inputs during a BIST test loop if the receiver is to indicate "no-errors". These codes are slightly different than those shown in *Table 1* (e.g., the fourth element in *Table 1* = C1.7 = K28.5 with forced negative running disparity, the fourth element in *Table 2*, = C5.0 = correct K28.5) because of the way that the running disparity affects the interpretation made by the receiver when it decodes certain characters. This table shows the codes that would be sent by a controller sending the BIST sequence without depending upon the LFSR in the transmitter, or by a transmitter connected to an Encoded Mode receiver that was receiving a BIST sequence while not itself in BIST mode.

It should be noted that the first K28.5 (C5.0) (the fourth byte in the BIST loop at *1 in *Table 2*) may cause a false RVS indication on the first pass through the BIST loop, because the transmitter sends it as a C1.7 (−K28.5) to force the state of running disparity (RD). Depending upon the actual RD state in the receiver as BIST starts, this forced RD might appear to the receiver as a running disparity error on the first time through the BIST loop. All subsequent loops are interpreted correctly.

The SVS character (C0.7) combined with the adjacent D11.5 (*2 in *Table 2*, row 25 bytes 13 and 14) encoded as (011000 0111 110100 1010) creates an alias K28.5 (001111 1010) which will cause an erroneous reframe if RF is HIGH for short periods of time (less than 2048 bytes). This alias sync can be used to check the system response to clock stretching, a topic that will be covered later. This error (normal single-byte reframe behavior) will not occur when Multi-Byte-Sync is enabled (i.e., RF = HIGH for more than 2048 byte times).

Note that there are several intentional code rule violations and incorrect running disparity transitions included in the receiver sequence. RVS (during BIST) will indicate when an error in the expected code has been detected; it does not indicate that an illegal code is present. *Figure 9* illustrates this behavior and shows the timing of RVS when the receiver detects an error in the sequence. RVS will pulse HIGH for the byte time following detection of a mismatch between the received-decoded pattern and the internally generated code.

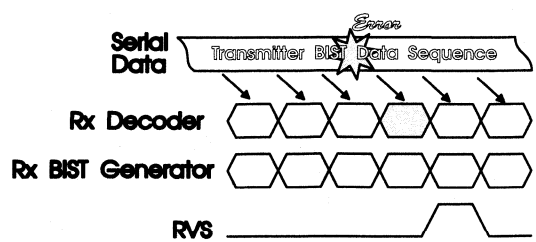


Figure 9. RVS Indicates Errors in Received Sequence

Table 2. HOTLink Receiver Input BIST Sequence

Start here ---->															
D0.0	C0.0	C4.0	**C5.0*1**	C4.7	C0.7	D27.7	D23.3 -->	D29.1	D30.0	C0.7	D15.4	D11.2	D3.1	D17.0	D4.0
C8.0	C6.0	C2.7	D14.7	C11.0	D19.5	D21.2	D12.1	C10.0	C3.0	D5.6	D8.3	C2.0	C1.0	D0.6	C0.0
C4.0	C8.0	C2.0	C5.0	D24.7	C6.0	C9.0	D18.6	C5.0	D28.5	C4.7	C11.0	D23.6	D13.3	D26.1	C7.0
D9.4	D2.2	C1.0	D20.4	C5.0	C4.7	C0.7	D11.7	D19.3	D21.1	D28.0	C4.7	C0.7	D15.6	D11.3	D19.1
D21.0	D12.0	C10.0	C7.0	D13.6	D10.3	C3.0	D17.4	D4.2	C8.0	C6.0	C9.0	D6.7	C9.0	D18.5	C5.0
D8.5	C2.0	C1.0	D20.6	C1.7	C4.7	C11.0	D3.7	D17.3	D20.1	C1.7	C10.0	C7.0	D13.7	D26.3	C7.0
D25.4	D6.2	C9.0	D22.4	C2.7	D14.5	C11.0	D3.5	D17.2	D4.1	C8.0	C2.0	C5.0	D12.7	C10.0	C3.0
D17.6	D4.3	C8.0	C2.0	C1.0	D4.7	C8.0	C2.0	C1.0	D0.7	C0.0	C0.0	C0.0	C0.0	C4.0	C8.0
C2.0	C1.0	D16.7	C4.0	C8.0	C2.0	C1.0	D20.7	C1.7	C10.0	C3.0	D1.7	D16.3	C4.0	C8.0	C2.0
C5.0	D28.7	C4.7	C11.0	D19.6	D5.3	D24.1	C6.0	C9.0	D6.6	C9.0	D22.5	C5.0	D10.5	C3.0	D1.5
D16.2	C4.0	C1.7	C10.0	C7.0	D29.7	D30.3	C0.7	D27.4	D7.2	D9.1	D18.0	C5.0	D12.4	C10.0	C7.0
D29.6	D14.3	C11.0	D19.4	D5.2	D8.1	C2.0	C1.0	D4.6	C8.0	C6.0	C9.0	D2.7	C1.0	D16.5	C4.0
C8.0	C6.0	C9.0	D22.7	C2.7	D26.5	C7.0	D9.5	D18.2	C5.0	D28.4	C4.7	C0.7	D31.6	D15.3	D27.1
D23.0	D13.0	D10.0	C3.0	D5.4	D8.2	C2.0	C5.0	D8.6	C2.0	C5.0	D24.6	C6.0	C2.7	D26.6	C7.0
D29.5	D30.2	C0.7	D31.4	D15.2	D11.1	D19.0	D5.0	D8.0	C2.0	C5.0	D12.6	C10.0	C7.0	D25.6	D6.3
C9.0	D18.4	C5.0	D12.5	C10.0	C3.0	D21.6	D12.3	C10.0	C3.0	D1.6	D0.3	C0.0	C0.0	C0.0	C4.0
C1.7	C10.0	C3.0	D17.7	D20.3	C1.7	C10.0	C3.0	D5.7	D24.3	C6.0	C9.0	D2.6	C1.0	D20.5	C1.7
C10.0	C7.0	D9.7	D18.3	C5.0	D24.4	C6.0	C2.7	D30.6	C0.7	D31.5	D31.2	D15.1	D27.0	D7.0	D9.0
D2.0	C1.0	D4.4	C8.0	C6.0	C2.7	D10.7	C3.0	D17.5	D20.2	C5.0	C4.7	C11.0	D7.7	D25.3	D22.1
C5.0	D10.4	C3.0	D5.5	D24.2	C6.0	C2.7	D10.6	C3.0	D21.5	D28.2	C4.7	C0.7	D11.6	D3.3	D17.1
D20.0	C1.7	C4.7	C0.7	D15.7	D27.3	D23.1	D29.0	D14.0	C11.0	D7.4	D9.2	D2.1	C1.0	D0.4	C0.0
C4.0	C5.0	C10.0	C7.0	D25.7	D22.3	C2.7	D26.4	C7.0	D13.5	D26.2	C7.0	D29.4	D14.2	C11.0	D23.4
D13.2	D10.1	C3.0	D1.4	D0.2	C0.0	C4.0	C8.0	C6.0	C2.7	D26.7	C7.0	D25.5	D22.2	C5.0	D30.4
C0.7	D15.5	D27.2	D7.1	D25.0	D6.0	C9.0	D6.4	C9.0	D6.5	C9.0	D2.5	C1.0	D0.5	C0.0	C0.0
C4.0	C8.0	C6.0	C9.0	D18.7	C5.0	D24.5	C6.0	C9.0	D22.6	C5.0	D30.5	**C0.7*2**	D11.5	D19.2	D5.1
D24.0	C6.0	C2.7	D14.6	C11.0	D23.5	D29.2	D14.1	C11.0	D3.4	D1.2	D0.1	C0.0	C0.0	C4.0	C1.7
C4.7	C11.0	D19.7	D21.3	D28.1	C4.7	C11.0	D7.6	D9.3	D18.1	C5.0	D8.4	C2.0	C5.0	D28.6	C4.7
C0.7	D27.6	D7.3	D25.1	D22.0	C2.7	D14.4	C11.0	D7.5	D25.2	D6.1	C9.0	D2.4	C1.0	D4.5	C8.0
C2.0	C5.0	D8.7	C2.0	C1.0	D16.6	C4.0	C5.0	C10.0	C3.0	D21.7	D28.3	C4.7	C11.0	D3.6	D1.3
D16.1	C4.0	C8.0	C6.0	C5.0	D30.7	C0.7	D27.5	D23.2	D13.1	D26.0	C7.0	D13.4	D10.2	C3.0	D21.4
D12.2	C10.0	C7.0	D9.6	D2.3	C1.0	D16.4	C4.0	C5.0	C4.7	C11.0	D23.7	D29.3	D30.1	C0.7	D11.4
D3.2	D1.1	D16.0	C4.0	C5.0	C4.7	C0.7	D31.7	D31.3	D31.1	D31.0	D15.0	D11.0	D3.0	D1.0	GOTO Start

When the receiver $\overline{\text{BISTEN}}$ input is set LOW, it initializes its BIST pattern generator and begins searching for the start of the transmitter BIST pattern (see *Figure 4*, time 5). While it is awaiting this start code, $\overline{\text{RDY}}$ and RVS will be HIGH. When it finds the beginning of the pattern (a D1.0 followed by a D0.0, with the proper running disparity), first RVS falls then $\overline{\text{RDY}}$ falls. $\overline{\text{RDY}}$ will remain LOW for the next 510 bytes of the BIST loop and then pulse HIGH for one byte time (see *Figure 4*, time 6). This BIST specific behavior of the $\overline{\text{RDY}}$ output allows an external system monitor state machine to count the number of times that the receiver has checked the BIST data. In non-BIST modes, $\overline{\text{RDY}}$ pulses once per byte in Encoded mode, or once per K28.5 in Bypass mode.

The actual bit pattern appearing at the receiver outputs (Q0–7, and SC/$\overline{\text{D}}$) will match the decoder output for all data patterns, but may not match the data-

sheet pattern for all of the Special Character codes being received. *Table 3* shows the patterns which will appear at the outputs of the receiver while BIST is running. Many of the codes shown do not appear in the datasheet and no correlation should be inferred between these output patterns and the reserved codes mentioned therein. Likewise, if these codes are presented to a transmitter, it will not send the codes necessary to create a good BIST pattern. These codes will typically be monitored by a logic analyzer, and might assist in debugging a particular serial link error phenomenon.

Table 3. HOTLink Receiver Output Patterns during a BIST Sequence

D0.0	C16.0	C20.4	C28.6	C30.7	C15.7	D27.7	D23.3	D29.1	D30.0	C31.0	D15.4	D11.2	D3.1	D17.0	D4.0
C24.0	C22.4	C29.6	D14.7	C11.3	D19.5	D21.2	D12.1	C10.0	C19.4	D5.6	D8.3	C2.1	C1.4	D0.6	C16.3
C4.5	C8.6	C18.7	C5.7	D24.7	C6.3	C9.5	D18.6	C21.3	D28.5	C14.2	C27.5	D23.6	D13.3	D26.1	C7.0
D9.4	D2.2	C17.1	D20.4	C28.2	C30.5	C15.6	D11.7	D19.3	D21.1	D28.0	C30.0	C31.4	D15.6	D11.3	D19.1
D21.0	D12.0	C26.0	C23.4	D13.6	D10.3	C3.1	D17.4	D4.2	C24.1	C6.4	C25.6	D6.7	C9.3	D18.5	C5.2
D8.5	C2.2	C17.5	D20.6	C28.3	C14.5	C11.6	D3.7	D17.3	D20.1	C12.0	C26.4	C23.6	D13.7	D26.3	C7.1
D25.4	D6.2	C25.1	D22.4	C29.2	D14.5	C11.2	D3.5	D17.2	D4.1	C8.0	C18.4	C21.6	D12.7	C10.3	C3.5
D17.6	D4.3	C8.1	C2.4	C17.6	D4.7	C8.3	C2.5	C1.6	D0.7	C0.3	C0.5	C0.6	C16.7	C4.7	C8.7
C2.7	C1.7	D16.7	C4.3	C8.5	C2.6	C17.7	D20.7	C12.3	C10.5	C3.6	D1.7	D16.3	C4.1	C8.4	C18.6
C21.7	D28.7	C14.3	C11.5	D19.6	D5.3	D24.1	C6.0	C25.4	D6.6	C25.3	D22.5	C13.2	D10.5	C3.2	D1.5
D16.2	C20.1	C12.4	C26.6	C23.7	D29.7	D30.3	C15.1	D27.4	D7.2	D9.1	D18.0	C21.0	D12.4	C26.2	C23.5
D29.6	D14.3	C11.1	D19.4	D5.2	D8.1	C2.0	C17.4	D4.6	C24.3	C6.5	C9.6	D2.7	C1.3	D16.5	C4.2
C24.5	C6.6	C25.7	D22.7	C13.3	D26.5	C7.2	D9.5	D18.2	C21.1	D28.4	C30.2	C31.5	D31.6	D15.3	D27.1
D23.0	D13.0	D10.0	C19.0	D5.4	D8.2	C18.1	C5.4	D8.6	C18.3	C5.5	D24.6	C22.3	C13.5	D26.6	C23.3
D29.5	D30.2	C31.1	D31.4	D15.2	D11.1	D19.0	D5.0	D8.0	C18.0	C21.4	D12.6	C26.3	C7.5	D25.6	D6.3
C9.1	D18.4	C21.2	D12.5	C10.2	C19.5	D21.6	D12.3	C10.1	C3.4	D1.6	D0.3	C0.1	C0.4	C16.6	C20.7
C12.7	C10.7	C3.7	D17.7	D20.3	C12.1	C10.4	C19.6	D5.7	D24.3	C6.1	C9.4	D2.6	C17.3	D20.5	C12.2
C26.5	C7.6	D9.7	D18.3	C5.1	D24.4	C22.2	C29.5	D30.6	C31.3	D31.5	D31.2	D15.1	D27.0	D7.0	D9.0
D2.0	C17.0	D4.4	C24.2	C22.5	C13.6	D10.7	C3.3	D17.5	D20.2	C28.1	C14.4	C27.6	D7.7	D25.3	D22.1
C13.0	D10.4	C19.2	D5.5	D24.2	C22.1	C13.4	D10.6	C19.3	D21.5	D28.2	C30.1	C15.4	D11.6	D3.3	D17.1
D20.0	C28.0	C30.4	C31.6	D15.7	D27.3	D23.1	D29.0	D14.0	C27.0	D7.4	D9.2	D2.1	C1.0	D0.4	C16.2
C20.5	C12.6	C26.7	C7.7	D25.7	D22.3	C13.1	D26.4	C23.2	D13.5	D26.2	C23.1	D29.4	D14.2	C27.1	D23.4
D13.2	D10.1	C3.0	D1.4	D0.2	C16.1	C4.4	C24.6	C22.7	C13.7	D26.7	C7.3	D25.5	D22.2	C29.1	D30.4
C31.2	D15.5	D27.2	D7.1	D25.0	D6.0	C25.0	D6.4	C25.2	D6.5	C9.2	D2.5	C1.2	D0.5	C0.2	C16.5
C4.6	C24.7	C6.7	C9.7	D18.7	C5.3	D24.5	C6.2	C25.5	D22.6	C29.3	D30.5	C15.2	D11.5	D19.2	D5.1
D24.0	C22.0	C29.4	D14.6	C27.3	D23.5	D29.2	D14.1	C11.0	D3.4	D1.2	D0.1	C0.0	C16.4	C20.6	C28.7
C14.7	C11.7	D19.7	D21.3	D28.1	C14.0	C27.4	D7.6	D9.3	D18.1	C5.0	D8.4	C18.2	C21.5	D28.6	C30.3
C15.5	D27.6	D7.3	D25.1	D22.0	C29.0	D14.4	C27.2	D7.5	D25.2	D6.1	C9.0	D2.4	C17.2	D4.5	C8.2
C18.5	C5.6	D8.7	C2.3	C1.5	D16.6	C20.3	C12.5	C10.6	C19.7	D21.7	D28.3	C14.1	C11.4	D3.6	D1.3
D16.1	C4.0	C24.4	C22.6	C29.7	D30.7	C15.3	D27.5	D23.2	D13.1	D26.0	C23.0	D13.4	D10.2	C19.1	D21.4
D12.2	C26.1	C7.4	D9.6	D2.3	C1.1	D16.4	C20.2	C28.5	C14.6	C27.7	D23.7	D29.3	D30.1	C15.0	D11.4
D3.2	D1.1	D16.0	C20.0	C28.4	C30.6	C31.7	D31.7	D31.3	D31.1	D31.0	D15.0	D11.0	D3.0	D1.0	GOTO Start

BIST Auto-Abort and Restart

When the receiver detects an error in the received expected sequence of transmission codes it will assert RVS during the byte-time following the error. A normally operating system will rarely experience one error per hour (a bit error rate of $1x10^{-12} \approx$ 1 error/hour @ 266 Mbaud), and systems doing some kind of design tolerance or performance limit testing will usually run with less than a few errors per second (BER of $1x10^{-8} \approx$ 3 error/second @ 266 Mbaud) even during link length testing. At these rates, it can be assumed that each error flagged by RVS was caused by an error that corrupts a single bit. It is impossible to distinguish between single bit errors and multiple bit errors within a single byte, since errors are only reported on a byte-by-byte basis. Further, since many kinds of errors change a legal data-byte into another legal byte many errors will be reported at times unrelated to when the error occurred. Single-bit errors can cause changes in the data stream running disparity, and will be detected as errors in the forced running disparity codes.

In extreme cases, where the errors cause PLL cycle slipping, or loss of framing, it is possible to create ambiguous error indications and seemingly endless running error sequences. Once the bit sequence has been corrupted, or after the PLL has bit-slipped, the BIST comparator will indicate a 100% error rate (except for the 32 expected violations that occur as part of the BIST pattern).

Since the BIST generator is a free-running counter that is only initialized while it awaits the start of the transmitter BIST sequence, errors of any kind don't affect the LFSR sequence. This feature can be used to advantage for several types of testing that generate long sequences of errors, since when the errors are removed, the receiver BIST generator predicted data will eventually match the received serial digital data without having to be realigned. Unfortunately, if the error causes the PLL to slip a bit, the received stream will never match.

To account for any loss of BIST sequence condition, the BIST logic included in the receiver will abort an extremely damaged sequence. It will abandon the current sequence and search for the start-of-BIST character and then resume comparisons from the beginning. When this auto-abort happens, $\overline{RDY}$ will go HIGH and remain there until the beginning of a new sequence is detected. While the receiver is waiting, RVS will also remain HIGH. The criteria for Auto-abort requires that there be $\geq$16 RVS indications within 32 contiguous bytes, and is checked every 64 bytes.

For system tests where the user wants to use the BIST comparator to check for longer running errors (and receiver PLL recovery without slipping) it is possible to disable the auto-abort function. The counter that is used to sample the error counter runs on REFCLK. By disconnecting the REFCLK input from the receiver after the PLL has reached the correct operating frequency, the internal counters that manage the error monitor are disabled. (There is a 50/50 probability that when REFCLK is disabled, the auto-abort counter will still be enabled, but by reconnecting, and then disconnecting the REFCLK the auto-abort function can be disabled. The function is controlled by an internal REFCLK divided-by-64 counter. For the first 32-byte times, auto-abort is enabled, for the other 32-byte times it is disabled.)

Tests Using BIST

Built-In Self-Test is a valuable and versatile tool for performing offline-test in any system. It also offers an unambiguous method to examine the performance of HOTLink products and other serial link components. The following short test descriptions are intended to introduce the reader to the capabilities of HOTLink and BIST as an evaluation tool. The tests described are typical of those required to evaluate most physical layer components.

Transmission Line Length

To check for the maximum transmission line length over which HOTLink can communicate, it is only necessary to connect the selected transmission line between a HOTLink Transmitter and Receiver. Most transmission line testing uses arbitrary data patterns that represent typical communication patterns. The HOTLink transmitter and receiver BIST function serves this purpose so the user can check

for an acceptable error rate without extra test equipment and without reconfiguration of an operational link just to perform this test. Transmission lines can be extended or modified until RVS indicates an unacceptable error rate. Tests that might use BIST to indicate system margin include;

- fiber-optic optical attenuation budget and optical or electrical margin testing;
- wire transmission-line attenuation, crosstalk, emissions and noise susceptibility testing;
- electrical interface connections and signal-margin testing;
- data sources for serial interconnect hardware testing.

Rx Jitter Tolerance

The ultimate performance of any serial link is determined by the performance of the receiver. The function of the receiver is to recover data from a (seemingly arbitrary) serial data stream. This data stream is translated several times, coupled to and though several non-linear devices and subjected to all manners of distortion. The receiver must accept this serial pulse train and recover a high-speed bit synchronous clock, de-jitter it, and then separate the DATA from the CLOCK. Jitter tolerance is the typical term used for this function. HOTLink receiver jitter tolerance can be measured by connecting a suitable transmission media between the transmitter and receiver, and inserting a jitter generation source similar to that shown in *Figure 10*. By inserting measured jitter amplitudes and watching the RVS output of the receiver, jitter tolerance can be measured.

There are two basic types of jitter that must be accommodated, deterministic jitter and random jitter. Deterministic jitter is comprised of data dependent jitter (DDJ) and duty cycle distortion (DCD). DDJ is caused by imperfections in the serial link that cause signal corruption that is proportional (or at least a strong function of) the particular data stream. DCD is caused by imperfection or imbalances in the serial link that cause signal corruption that is related to the timing of the rising or the falling edges. Random jitter is unrelated to the data stream, the edge rates, or the link quality. It is typically caused by external noise events or by thermal noise in the optical components. Random jitter is uncorrelated to the data stream and is difficult to reproduce experimentally.

DCD creates high-frequency jitter at about the bit rate of the serial data stream since bit placement errors are complemented within the pulse that is distorted. DDJ creates high-frequency jitter at about the bit rate of the serial data stream since bit-placement errors are usually complemented within a bit or two. These high-frequency jitter components should be filtered by the PLL filter, and should

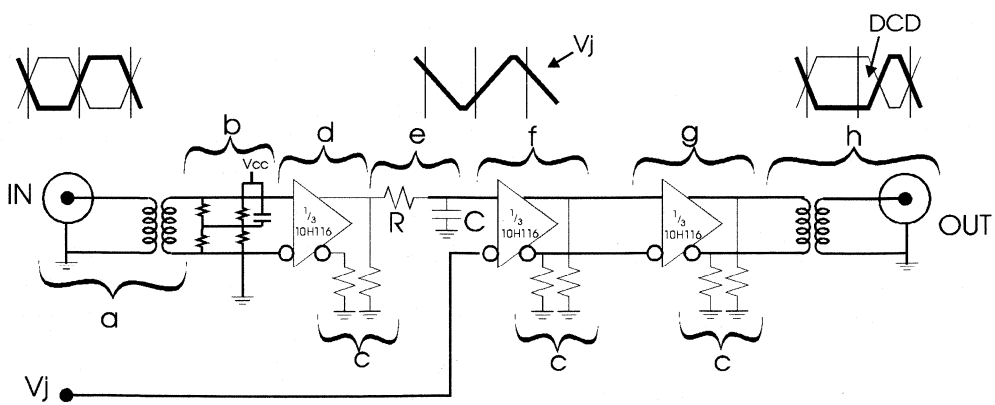


Figure 10. Jitter Generator Schematic

cause no significant jitter at the CKR output of the receiver. DDJ can cause baseline-wander at about the byte rate of the serial data stream, but since the 8B/10B code is balanced over multiple bytes, there should be little or no low-frequency components in the jitter. Random jitter has both high- and low-frequency components, and will cause output jitter as it causes the PLL to attempt to track a corrupted data stream. All three types of jitter must be accommodated by the receiver as it captures the data and aligns its serial clock.

Data dependent jitter can be generated by a suitable length of coaxial cable. If DDJ and input amplitude must be separately measured, an external line receiver and level restoration circuit might be needed. Duty cycle distortion can be generated by the circuit shown in *Figure 10*. This circuit uses the stages in a 10H116 (ECL triple-differential amplifier) to perform; 1) Differential-to-single-ended transformation; 2) Ramp generation; 3) Threshold shifting; 4) Level restoration; 5) Differential buffering. In this circuit the transmitter data stream is fed through the jitter generator while the receiver monitors and checks for correct operation. As the control voltage (Vj) input is varied between the 10KH Vil and VIh levels, the duty cycle of the data stream is corrupted in a repeatable and measurable manner.

Serial-data input to the jitter generator can use any appropriate connector, and coupling circuit. The connector and transformer shown at (a & h) will work with coaxial cable or STP cables. For fiber-optic interfaces, these could be eliminated by direct coupling to fiber-optic receiver/transmitter modules. Transmission-line termination and DC threshold adjustments are performed by the simple network shown at (b). The first differential stage of the '116 (d) is used as a differential-to-single-ended converter with a controlled output impedance and symmetrical rise and fall times. The ECL output termination resistors shown at the outputs of each differential stage (c) may be replaced with parallel termination resistors if better impedance control or closer edge-rate matching is required. The R-C ramp generator at (e) must be tuned to each data rate, to insure that 100% voltage swing is maintained for the narrowest pulses expected. If the Ramp is too long, it will be possible to raise Vj above the level of some data bits, thus losing data. The second differential stage of the '116 (f) serves as a voltage comparator between the control voltage (Vj) level and the level of the signal at the output of the ramp-generator. Additional DC-filtering may be required between the Vj input and its input to (f) to insure that high-frequency, single-ended noise does not corrupt the data flow. The third differential stage of the '116 (g) is used to restore crisp-edged, full-swing levels to the serial data, and to drive the subsequent transmission line. Further details on the fabrication of the jitter generator and the measurement techniques required for accurate measurement of this injected jitter is beyond the scope of this note.

Receiver Error-Free-Window Test

A normally operating receiver PLL will adjust its internal clock such that incoming data transitions are placed at the maximum distance from the data-separator flip-flop sampling window. This placement allows misplaced transitions (jittered edges) the maximum margin before data-misinterpretation occurs. The width of this error free zone is commonly called error-free-window. It is less than the actual bit width (expressed in nanoseconds) by the sum of maximum peak-to-peak receiver PLL jitter, data-separator flip-flop sampling window width, absolute misalignment of the internal PLL sampling clock. (Actual test results will be additionally affected by clock source jitter and test equipment trigger and measurement inaccuracies.)

To measure the error-free-window (EFW) in the HOTLink Receiver, it is only necessary to connect a HOTLink Transmitter and receiver in BIST mode, while controlling the serial data stream with the transmitter FOTO pin. The FOTO input to the transmitter causes the OUTA+ and OUTB+ outputs to be LOW, and the OUTA− and OUTB− outputs to be HIGH for the time that FOTO is HIGH. (For purposes of this example it will be assumed that Tx−OUTA+ is connected directly to Rx−INA+ and that Tx−OUTA− is connected directly to Rx−INA−) Since FOTO is an asynchronous TTL input, it is possible to use it as a Controlled Data Corrupter that can move an edge away from its nominal posi-

tion. The limits of the EFW will be signaled by an indication on RVS.

To set up the test, the user would connect a pulse generator to the transmitter FOTO pin. This generator would be triggered by the $\overline{RP}$ output and would be controllable in both delay and pulse width. Since $\overline{RP}$ pulses once each BIST loop, the generator would make pulses that were phase aligned with the serial data stream. By careful adjustment of pulse width (VERY narrow, adjustable-width pulses) and delay (alignment such that the forced LOW is placed in a position that is naturally LOW).

To perform the test, the user should first adjust the generator so that it causes no corruption in the actual data stream. Then, by carefully adjusting the delay and/or width of the generator a specific edge in the data stream can be realigned until the RVS output indicates a BIST error. By noting the position of the realigned edge relative to its nominal position, the early and late limits of the EFW can be measured.

The relationship between the FOTO pulse and its effect on the OUTA transition must be measured empirically. The OUTA+ expanded waveform shown in *Figure 11* illustrates the control that can be effected by FOTO. The vertical lines (Internal Rx Sampling Locations) indicate the location of ideal receiver sampling points, and the shaded regions around them indicate the built-in errors that limit EFW.

Since FOTO only forces the OUTA+ & B+ output to a LOW (and OUTA− & OUTB− to a HIGH), it

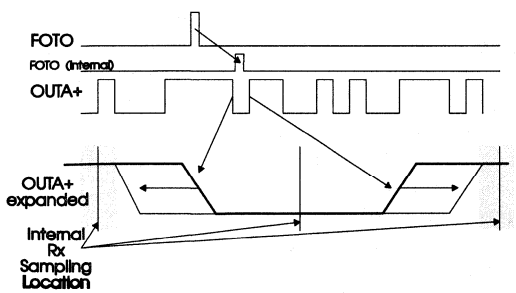


Figure 11. Example of Error Free Window Testing

is not possible to check for rising and falling edge symmetry with this test. A falling edge can only be forced to an earlier LOW, and a rising edge can only be forced to a later HIGH. By careful adjustment of the FOTO generator, it is possible to adjust the position of all of the various 1,2,3,4,5 bit length pulses found in the 8B/10B code.

Rx Run-Length Tolerance Test

An extension of the EFW test will allow the user to measure the receiver's tolerance to missing pulses. If the pulse width of the FOTO generator described above is increased beyond a few bits, the resulting data stream will have missing pulses beyond the 5-bit run-length of the 8B/10B code. These missing transitions allow the PLL control voltage to drift, causing an arbitrary phase change. When the transitions resume, the PLL realigns to the incoming bit stream. However, if the drift phase has gone beyond the jitter limit, the PLL may align to a different bit position than the one to which it was previously aligned. (See *Figure 12.*) This realignment is commonly called cycle slip and equates to the loss or addition of a bit to a serial data stream.

Obviously the RVS output will indicate an error, as shown in *Figure 13*, while the data is masked, but since the indicated error is bounded (i.e., recovers within a few bytes), the BIST detector shows that the receiver is able to continue finding good data within a few bits or bytes of resumption of the sequence.

As the FOTO pulse width increases from a few bits to a few bytes, the RVS indication widens proportionately. There may be positions where a minor change in width causes multiple byte errors and others where multi-bit width changes cause the RVS to show apparently good data. The former is an indication of a running disparity error which might run for several bytes before being terminated by a code in the BIST sequence. The latter is an indication that at that particular position in the sequence, BIST was already expecting a violation, so would not flag an error for this type of data corruption.

When the FOTO pulse width (or the RVS pulse width) approaches 16 bytes, the BIST-Auto-abort mechanism described in an earlier section will begin to obscure the real receiver tolerance. When the er-

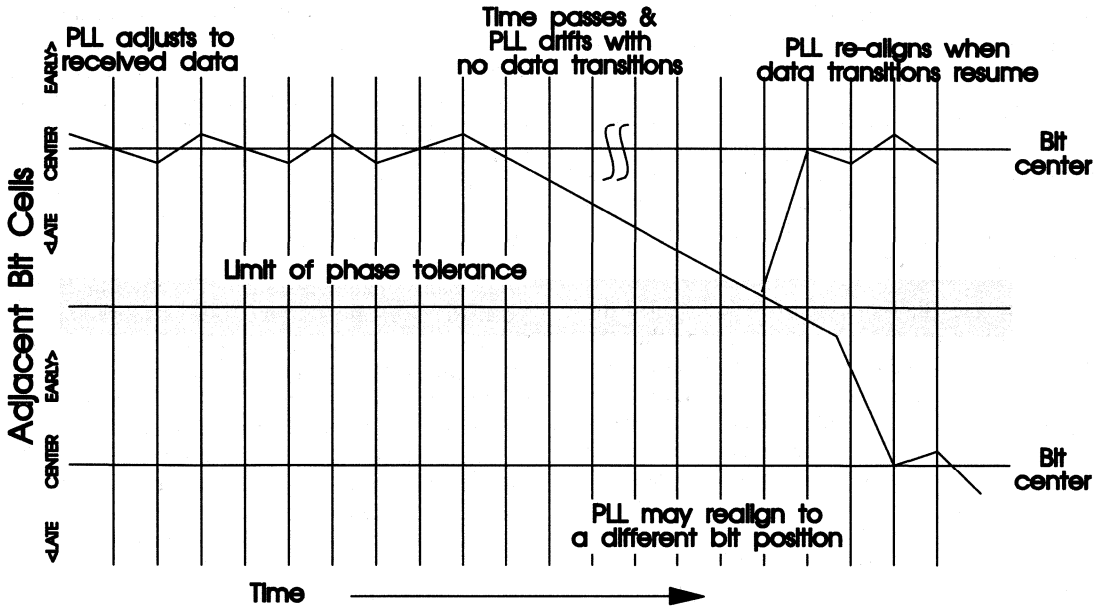


Figure 12. Long Spaces without Transitions May Cause Cycle Slip

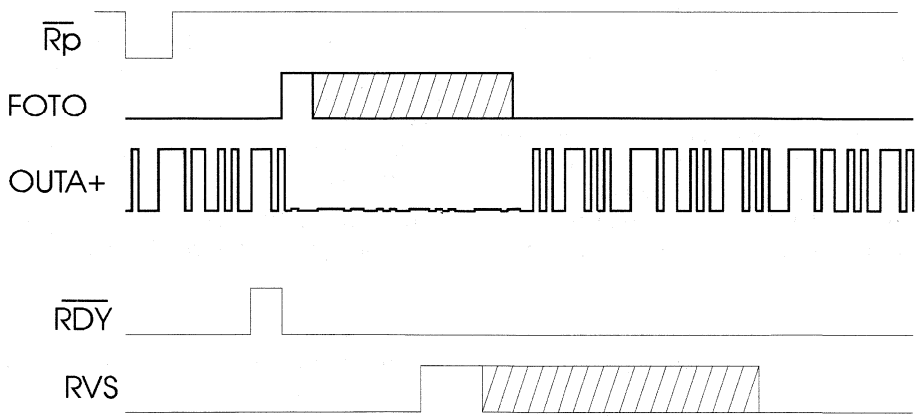


Figure 13. Missing-Transition Test Timing Diagram

ror run length approaches 16 bytes, the RVS width will become almost continuous. $\overline{RDY}$ will cease its normal pulse-once-per-loop operation and will rise during the RVS pulse, and stay HIGH for the remainder of the BIST loop as the receiver BIST checking circuit waits for a start-of-BIST pattern. If $\overline{RP}$, RVS and $\overline{RDY}$ are all simultaneously displayed on the oscilloscope screen, it will be noted that the BIST loop appears to start correctly, but that after the FOTO pulse, all the data is corrupted. This is

the automatic-restart behavior and is characteristic of the BIST-auto-abort logic, not an indication of a real corrupted data stream.

HOTLink Receiver can tolerate nearly 100 transitionless bytes without cycle-slipping so meaningful testing requires suppression of the auto-abort function. As described earlier, this BIST-auto-abort logic can be suppressed by removing the REFCLK input from the receiver.

Even with the BIST-auto-abort logic disabled the pulse width of the FOTO generator (and thus the number of missing data transitions) will ultimately become long enough that the receiver PLL will cycle slip before data resumes. Since the BIST comparator requires an absolutely perfect data stream and cannot realign without external assistance, the BIST checker will show that the all of the received data is incorrect. Once the RVS indication becomes continuous it will be necessary to either reconnect REFCLK (and allow the Auto-abort logic to reinitialize the receiver BIST generator) or to toggle the $\overline{\text{BISTEN}}$ input on the receiver (which forces the BIST generator to begin from the beginning). The FOTO-generator pulse width (expressed in bit times) that causes an unrecoverable error is the missing-transition limit of the HOTLink receiver (i.e., run length tolerance).

Figure 13 illustrates signals involved in the run length tolerance test. The actual time measurement will be affected by the timing of the FOTO pulse, and the jitter added by the interconnect link. Care should be taken to assure that the first missing pulse comes after a normally placed pulse (i.e., make sure that FOTO takes effect while the data stream is naturally LOW and that it does not disturb the position of the last transition before it kills the data stream). If the receiver PLL is recovering from a large phase correction at the time it is left to float, reduced run-length tolerances will result.

Reframe-CKR Stretch

In normal systems it is difficult to cause the HOTLink Receiver to reframe off established byte boundaries using normal transmission data. The HOTLink BIST sequence includes one occurrence of a bit pattern that mimics a K28.5 aligned to incorrect byte boundaries. To view this clock stretch behavior, it will be necessary to synchronize the oscilloscope with the $\overline{\text{RP}}$ of the transmitter, and delay the display to the area of the alias sync. *Figure 14* shows the effect of an alias sync (five bits misaligned). In this example, taken from the BIST sequence, the C0.7–D11.5 cause an alias-sync realignment. The next several bytes are corrupted because of this misframing. When the C2.7 (+K28.5) arrives, it realigns the data to the proper boundaries. In the six byte times between the C0.7 and the C2.7 there have been two clock-stretching events and only five bytes have come out of the receiver (all bad without any RVS indication). Please note that this illustration shows the function of the receiver and is not intended to show actual timing with respect to the serial data stream.

If the receiver RF input has been HIGH for less than 2048 bytes, this single alias-K28.5 (double-un-

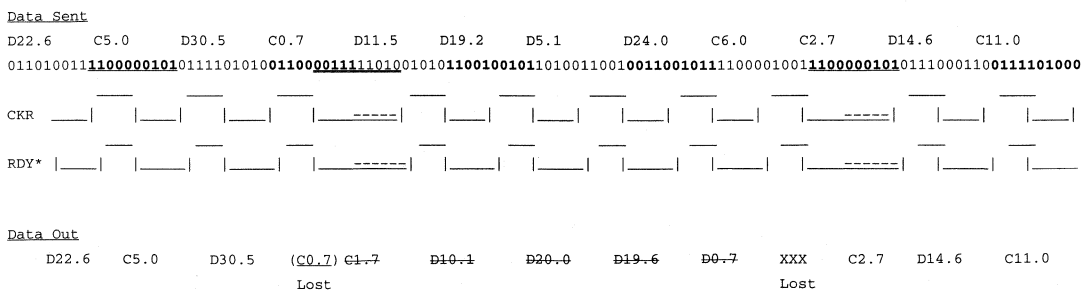


Figure 14. Illustration of Receiver Behavior during Reframe Clock Stretch

derlined in *Figure 8*) will cause a byte realignment (reframe) to the incorrect byte boundary (5 bits off of the real byte alignment) and thus a stretch of CKR, $\overline{RDY}$ and the position of Q0–7, SC/$\overline{D}$ and RVS until the next properly aligned K28.5 (approximately 5 bytes later). In the illustration, the C0.7 indication is lost because of the reframe caused by the alias sync and the adjacent clock edges are separated by fifteen bits. Similarly, when the real K28.5 arrives (C2.7 in the example), the Q0–7, SC/$\overline{D}$ outputs will change twice between adjacent clocks (i.e., internal bit 0) between the D0.7 and the C2.7 (i.e., once on the old bit 2 and again on the new bit 2). These adjacent clock edges are separated by fifteen bits and the specified set-up and hold times for subsequent logic will be assured.

After the receiver RF has been HIGH for more than 2048 bytes, the internal byte framer changes from requiring a single K28.5 to re-align the byte, to requiring two K28.5s to reframe. To keep the receiver in single byte-framing mode, and to perform this test it will be necessary to pulse the RF input at a rate less than once per 2048 bytes (maybe triggered by $\overline{RP}$/4).

An alternative method to show byte realignment and CKR stretching involves sending a string of data that includes a positive running disparity K28.7 (C7.0) followed by D11.x or D20.x or by sending a positive running disparity SVS (C0.7) followed by D11.x. (e.g., C0.7 = 0110000111 or 1001111000 and D11.x = 110100xxxx or 001011xxxx so if the correct running disparity SVS is followed by the correct running disparity D11.x, a five bit misaligned-alias-sync is created as follows; 01100**00111111010**0xxxx)

Receiver Offset Frequency

Differences in frequency between the transmitter crystal oscillator and receiver REFCLK crystal oscillator might limit performance of the data communication link. The HOTLink datasheet specifies that the receiver and transmitter frequencies can be different by ±0.1% (1000 ppm) without compromise to the reliability of the data link. This parameter is conveniently checked by operating a transmitter CKW on one generator or crystal oscillator, and the receiver REFCLK on another. If both HOTLink parts are operating in BIST mode the RVS output will indicate the quality of the link.

As the generator frequency is adjusted (slowly and smoothly) the RVS should stay LOW indicating correct operation. RVS may show errors when the generator frequency is adjusted, though it is unlikely. If this happens, it is probable that the frequency change is being made too abruptly. The test is still possible, if RVS is checked only after the generators stabilize at each new frequency.

Tolerance to Phase Changes in Received Data

Two transmitters operated from the same clock source will run at exactly the same data rate. If they are both in BIST mode and synchronized by simultaneous assertion of the SVS input, they will also be sending exactly the same serial data. If their respective clocks are phase adjusted over a narrow delay range, they can be used as a source of synchronized serial data with a known phase relationship.

The receiver has two equivalent serial inputs (INA± and INB±) which can be independently selected. If the two transmitters are each connected to one of the serial data inputs, and if a synchronized source alternately selects one, then the other (using A/B_Select), the receiver's phase adjustment behavior can be examined. (See *Figure 15*.) Synchronized switching is easily accomplished by using the $\overline{RP}$ output of one of the transmitters to trigger a long-pulse-width ECL generator (200–300 bytes pulse width, carefully aligned so that the change happens during a quiescent portion of the serial stream).

As the two transmitters are alternately selected and as the delay between them is increased, the receiver sees a continuous BIST data stream containing instantaneous phase changes equal to the difference in transmitter-to-transmitter, clock-to-clock skew. It must adjust to the new data phase and realign its internal clock to correctly recover the data. The theoretical maximum phase adjustment range is slightly less than ±0.5 bit time (i.e., ±0.5 bit less Rx PLL jitter, static alignment, and flip-flop set-up/hold times). When the phase difference reaches the limit, errors will be indicated by pulses on RVS that

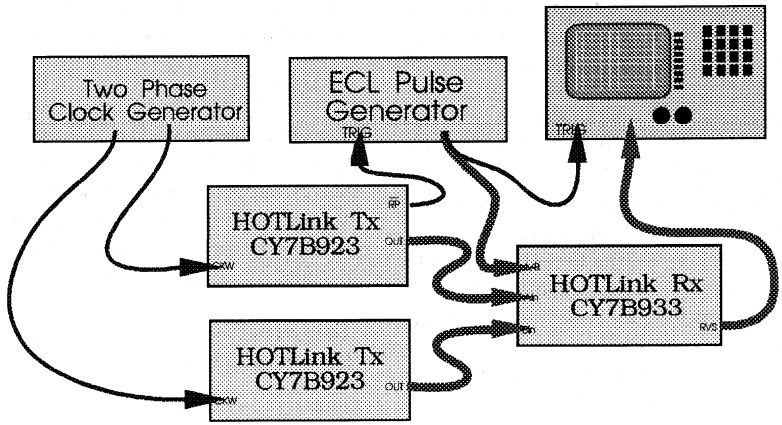


Figure 15. Receiver Phase Tolerance Test Setup

are one or more bytes wide. (Even though the actual error might involve only one bit, in one byte, the RVS indication may run for several byte times because of running disparity corruption.) As the data phase hop increases, the RVS pulses will increase in width proportional to the time taken to adjust the phase of the internal PLL.

Eventually RVS will stay high continually from the time of the A/B switch to the next $\overline{RDY}$ pulse (i.e., the start of the next BIST loop). As the magnitude of transmitter clock-to-clock phase difference approaches the point where the PLL phase alignment slips from one bit to the next (i.e., at approximately 180° phase difference) the BIST loop will become irreversibly corrupted and will auto-abort-restart after each phase hop.

Conclusion

HOTLink BIST capability should help system integrators add features to high-performance communication links. These features can be made to enhance usability and improve reliability of the link.

Test methods that use BIST will aid in evaluation of HOTLink products and other link support hardware. The HOTLink built-in test features allow an unambiguous indication of data quality, many of which require only inexpensive test equipment.

Document #: COM0002–1093

HOTLink is a trademark of Cypress Semiconductor Corporation.
ESCON is a trademark of International Business Machines.

Upgrade Your TAXI−275™ with HOTLink™

This application note will explain how to upgrade TAXI−275™ (Am79168/Am79169) devices with the HOTLink™ (CY7B923/CY7B933) devices from Cypress Semiconductor. It will aid in the migration of TAXI−275 designs to the HOTLink architecture. This note begins with an introduction to HOTLink and then gives advantages of HOTLink and replacement suggestions for the TAXI−275 devices.

HOTLink Introduction

The HOTLink family of devices transfers data from point to point over high-speed serial links at 160 to 330 Mbits/second (*Figure 1*). The CY7B923 Transmitter (*Figure 2*) takes an 8-bit parallel data stream and encodes it using the Fibre Channel and ESCON compliant 8B/10B code. This code maps all 8-bit data characters into a 10-bit transmission code that ensures that the transmission signal contains suitable transitions for recovery by the receiving device. The transmitter then takes this 10-bit data word and converts it to a serial bit stream and sends it at 10 times the byte rate over a serial transmission link.

The CY7B933 HOTLink Receiver (*Figure 3*) connects to the other end of a transmission link that may consist of anything from a few inches of printed circuit board trace to several kilometers of fiber optic cable. The receiver decodes the incoming bit stream and reconstructs the original parallel data character, which is presented at the outputs and aligned with the recovered clock. The receiver, in addition to these tasks, checks the incoming data stream for errors that may have occurred in the serial transmission.

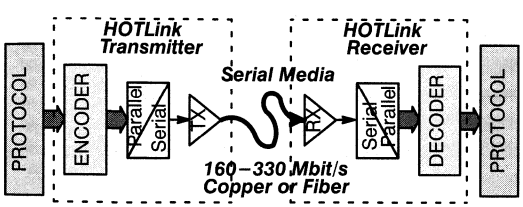


Figure 1. HOTLink System Diagram

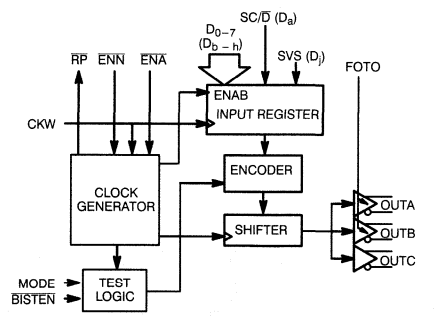


Figure 2. CY7B923 Transmitter Logic Diagram

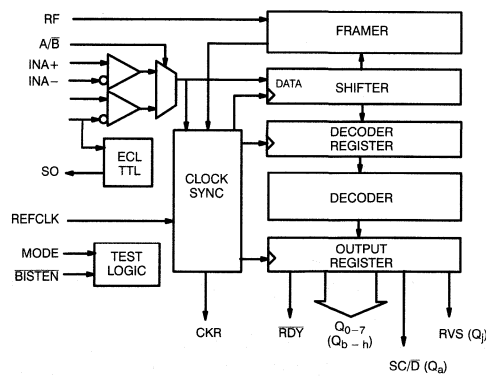


Figure 3. CY7B933 Receiver Logic Diagram

The SC/$\overline{D}$ (Special Character/$\overline{Data}$) pin provides the ability to transmit command codes in addition to sending data characters. These codes are mapped to 10-bit transmission characters defined in the 8B/10B codes of the Fibre Channel standard. This provides the ability to send commands as part of the transmission stream signaling events such as Idle, Start-of-frame, End-of-frame, etc.

Other features provide a complete solution for high-speed point-to-point communication in applications including interconnecting workstations, servers, mass storage, and video transmission equipment. These features include Built-In Self-Test (BIST) for in-system diagnostic testing, unencoded mode for sending 10-bit data in systems that use a different encoding method, and a seamless parallel interface for connection to both asynchronous and clocked FIFOs. A brief description of the various features of HOTLink are given below with a more detailed discussion found in the CY7B923/CY7B933 HOTLink Transmitter/Receiver datasheet. The PLCC pinouts for these devices are shown in *Figure 4*.

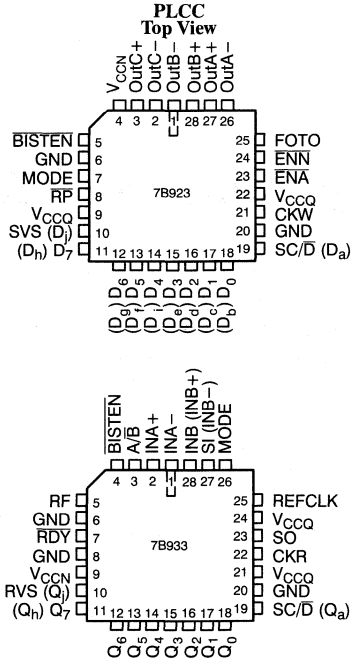


Figure 4. CY7B923 and CY7B933 Pin Configurations

Upgrade from TAXI−275

The following sections explain the architectural advantages of the Cypress CY7B923/CY7B933 HOTLink Transmitter and Receiver over the devices from AMD. This section begins with a brief explanation of the Am79168/Am79169 TAXI−275 devices. It then follows with a list of HOTLink features that make designing these high-speed point-to-point systems easier.

A Brief Explanation of TAXI−275

The Am79168/Am79169 TAXI−275 devices are similar to HOTLink. The Am79168 TAXI−275 Transmitter, shown in *Figure 5*, converts 8-bit or 10-bit parallel data into 10- or 12-bit transmission codes using either the 8B/10B code or the 10B/12B code. This data is encoded and shifted out serially over a transmission link operating at speeds of 175 to 275 MBaud. The Am79169 TAXI−275 Receiver, shown in *Figure 6*, converts the incoming serial data into parallel words, decodes, and presents the original words in 8-bit or 10-bit format to the outputs

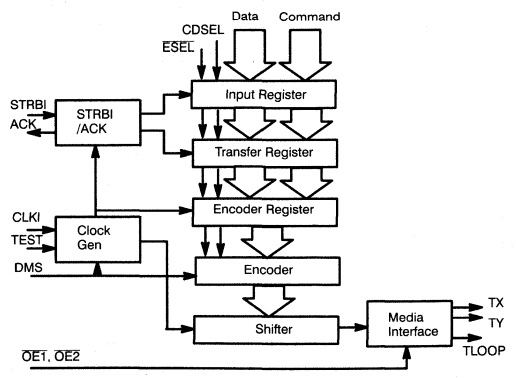


Figure 5. TAXI-275 Transmitter Block Diagram

along with the recovered clock. The pinouts of the Am79168 Transmitter and the Am79169 Receiver are shown in *Figure 7*.

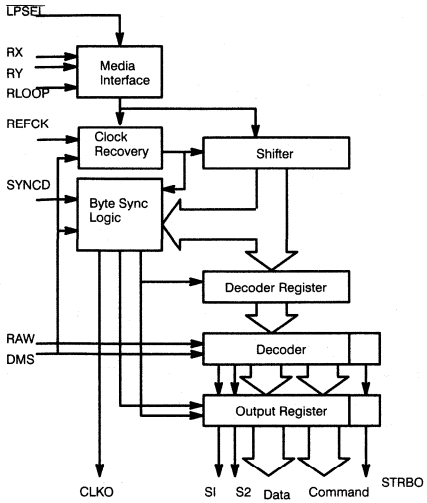


Figure 6. TAXI−275 Receiver Block Diagram

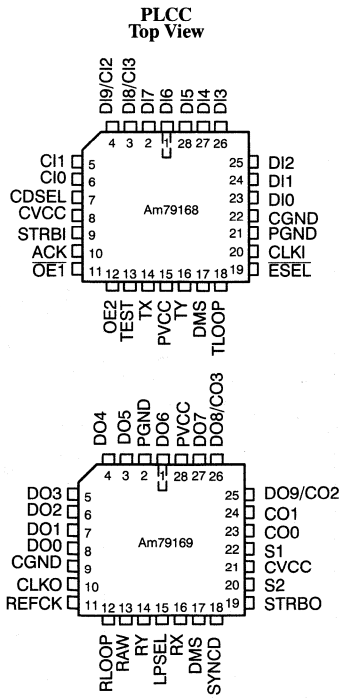


Figure 7. Am79168 and Am79169 Pin Configurations

Simplifying Your System with HOTLink

HOTLink offers additional features that will simplify system design. Below is a list of these features along with their benefits when designing high-speed point-to-point serial communications systems with Cypress HOTLink devices.

Multiplexed Command and Data

The TAXI−275 has separate inputs for command and data, while the HOTLink devices have an integrated command and data path. The status of SC/$\overline{D}$ pin (Special Character/$\overline{Data}$) determines if HOTLink sends a Special Character (Command) or data.

The integrated command and data paths of HOTLink allow simplification of the controller architecture. Instead of creating a separate command path, command codes can be integrated within the data stream with the addition of a ninth bit (the SC/$\overline{D}$) bit that indicates the status of the associated 8 bits of information.

More Outputs

The HOTLink transmitter has three identical differential Pseudo ECL (PECL) serial output ports. Two of these outputs can be turned off under control of FOTO (Fiberoptic Transmitter Off) pin. The TAXI−275 devices have only one differential PECL output pair and an additional single-ended TLOOP output intended for use in loop-back testing.

More Inputs

The HOTLink Receiver has two differential interfaces to the serial transmission medium (INA± and INB±) whereas the TAXI−275 devices have only a single input pair (RX,RY) and a single ended PECL input, RLOOP, used for loop-back testing. The media inputs of the HOTLink Receiver can be used to provide loop-back testing, redundant transmission paths, or more complex networks configurations.

Loop-back testing ensures that a node is sending and receiving data properly. In a typical network-style configuration, both the transmitter and receiver will exist for each node. In loop-back testing a redundant output from the transmitter is fed back to the additional input on the receiver. The TAXI−275 devices have an extra single-ended 100K

PECL output, TLOOP, and an extra single-ended 100K PECL input, RLOOP, that are used for loop-back testing. The HOTLink devices offer a more robust loop-back capability by offering redundant differential output pairs that can be connected to an additional differential input structure on the receiver, as shown in *Figure 8*. The additional single-ended input/output pair of the TAXI−275 does not provide a robust loop-back testing configuration.

In addition, the redundant outputs of the transmitter can be used in conjunction with the additional inputs of the receiver to build more complex network structures by allowing a single transmitter to communicate with multiple receivers, or a single receiver to be connected with multiple transmitters. The multiple outputs can also be used to to build redundant paths between two nodes.

More Flexible Command Codes

A coding system is necessary in serial communication systems to insure that the receiving device can determine the boundary between adjacent bits. The code makes sure that enough signal transitions exist on the transmission channel in order to track bit boundaries. In other words, the code must ensure that the clock used by the transmitter to transmit the data is imbedded within the data stream. The code maps each character into a code word that ensures that a minimum transition density and run length is maintained.

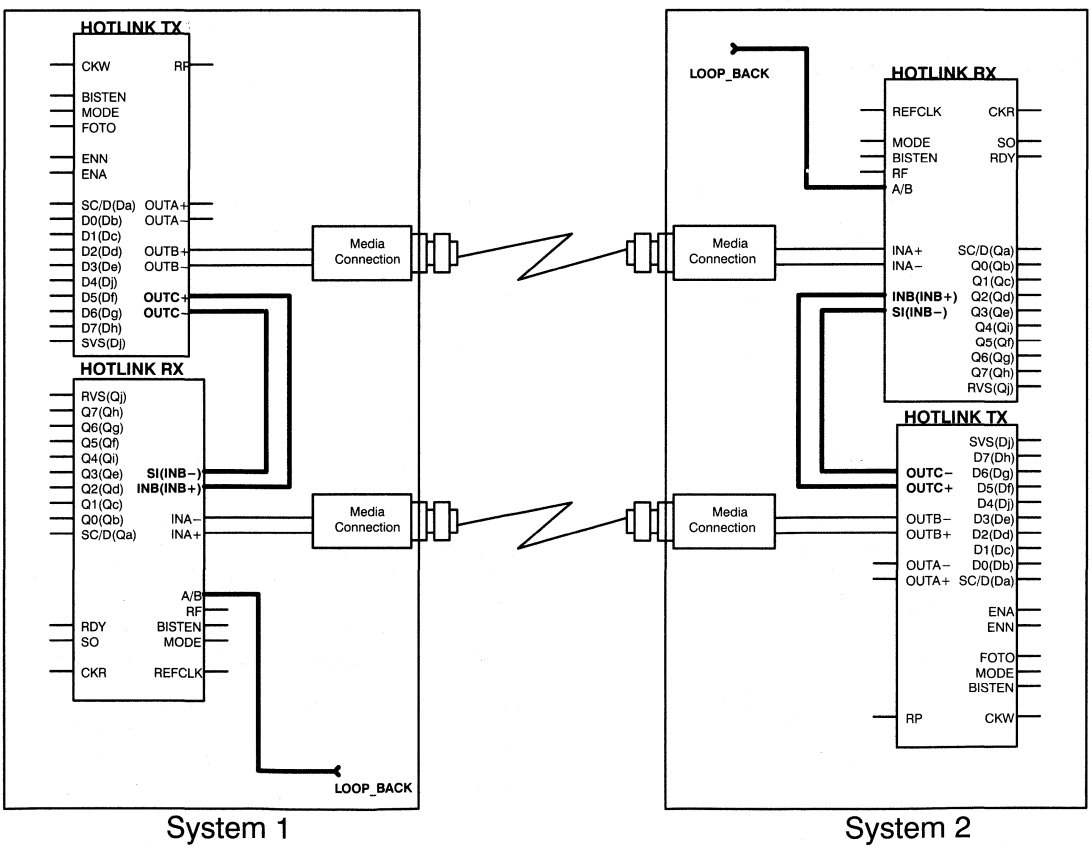


Figure 8. Example HOTLink Loop-Back System Connection

Both HOTLink and TAXI−275 use the transmission code specified by ANSI_X3T9.3 Fibre Channel and IBM ESCON standards. This code converts 8 bits into 10 transmission bits (8B/10B). This code generates NRZ (Non Return to Zero) transmission data where a logical 1 is represented by a HIGH level and a logic 0 is represented by a LOW level. The complete code tables are listed at the end of the CY7B923/CY7B933 HOTLink Transmitter/Receiver datasheet. This code ensures not only minimum transition density and run length, but also that the average number of 1s and 0s are equal. This feature of the code prevents the average DC level on the transmission link from "wandering" based on the data that is being sent.

In addition to specifying a mapping of every character into a transmission symbol, the code also specifies several command codes. These codes are useful for low level signaling without involving higher level protocols. They can be used to indicate information such as HALT, End Of Frame, or Start of Frame.

Table 1 shows the valid special characters and sequences that HOTLink can both encode on the transmitting end and decode on the receiving end. The first column in the table indicates the byte name of the special character. In the Fibre Channel and ESCON notation Special Characters are denoted with a 'K' prefix and Data Characters are denoted with a 'D' prefix. The first twelve special characters are defined in the Fibre Channel and ESCON specifications. The second column of the table gives the code name, both in decimal and hexadecimal notation, of the binary pattern on the I/O pins. The third column, bits, shows the pattern presented to the transmitter's data lines. This pattern, in combination with SC/$\overline{D}$ HIGH, will cause either the pattern in column four or column five to be sent. The pattern that the transmitter sends depends on the current Running Disparity.

In order to insure that the average number of 1s and 0s that are sent across the communications channel is equal, both the transmitter and receiver keep track of the Running Disparity of the data that was previously sent. Running Disparity (RD) can either be positive (+) or negative (−). In general, Running Disparity will be positive if, in the last transmission word, there were more 1s sent than 0s and it will be negative if there were more 0s sent than 1s. If RD is negative, the transmitter will send the code in column four and if RD is positive then the transmitter will send the code in column five.

Both HOTLink and TAXI−275 can send all codes labeled C0.0 through C11.0 in *Table 1*. Because of the different architectures of these two devices, the data presented to the inputs of the transmitter will be different, but the code sent across the transmission medium will be identical.

The next three codes represent sequences that the transmitter can send. For example, if the transmitter controller presents C0.1 (binary pattern 001 00000) to the data lines, then the transmitter will send −K28.5+, D21.4, D21.5, D21.5. In other words, the transmitter will send a negative K28.5 Special Character, a D21.4 (binary 100 10101) Data Character, and two D21.5 (binary 101 10101) Data Characters. It will continue to send this pattern as long as C0.1 is present at its inputs. The receiver will decode this pattern as a C1.7 or C5.0 depending on its current Running Disparity followed by D21.4 and two D21.5s. This pattern is defined in the Fibre Channel standard as the IDLE pattern. The ability of the transmitter to send this pattern as well as the R_RDY (Receiver Ready) pattern greatly simplifies controller design. The TAXI−275 devices have no ability to send complex data patterns with a single code as shown by the word **NONE** in *Table 1* under the TAXI−275 Code column.

In addition, if C2.1 is presented to the transmitter, it will send either a negative K28.5 or a positive K28.5, depending on the Running Disparity. It will then modify the Least Significant Bit (LSB) of the subsequent data word to be either a 0 if RD was (−) or a 1 if RD was (+). This simplifies controllers when building End of Frame (EOF) delimiters where the second byte is determined by the current RD. These packet structures are necessary to conform with the Fibre Channel specification. The TAXI−275 device only has the capability of modifying the LSB of two different Data Characters, limiting the possible EOF delimiters that can be constructed.

Table 1. HOTLink Valid Special Character Codes and Sequences (SC/$\overline{D}$ = HIGH)

HOTLink Special Code Byte Name	Special Code Code Name		Bits HGF	Bits EDCBA	Current RD− abcdei	Current RD− fghj	Current RD+ abcdei	Current RD+ fghj	Receiver Output Code Name	TAXI Code
K28.0	C0.0	(C00)	000	00000	001111	0100	110000	1011	C0.0	K28.0
K28.1	C1.0	(C01)	000	00001	001111	1001	110000	0110	C1.0	K28.1
K28.2	C2.0	(C02)	000	00010	001111	0101	110000	1010	C2.0	K28.2
K28.3	C3.0	(C03)	000	00011	001111	0011	110000	1100	C3.0	K28.3
K28.4	C4.0	(C04)	000	00100	001111	0010	110000	1101	C4.0	K28.4
K28.5	C5.0	(C05)	000	00101	001111	1010	110000	0101	C5.0	K28.5
K28.6	C6.0	(C06)	000	00110	001111	0110	110000	1001	C6.0	K28.6
K28.7	C7.0	(C07)	000	00111	001111	1000	110000	0111	C7.0	K28.7
K23.7	C8.0	(C08)	000	01000	111010	1000	000101	0111	C8.0	K23.7
K27.7	C9.0	(C09)	000	01001	110110	1000	001001	0111	C9.0	K27.7
K29.7	C10.0	(C0A)	000	01010	101110	1000	010001	0111	C10.0	K29.7
K30.7	C11.0	(C0B)	000	01011	011110	1000	100001	0111	C11.0	K30.7
Sequences										
Idle	C0.1	(C20)	001	00000	-K28.5+,D21.4,D21.5,D21.5, repeat				C5.0, D21.4, D21.5, D21.5	**NONE**
R_RDY	C1.1	(C21)	001	00001	-K28.5+,D21.4,D10.2,D10.2, repeat				C5.0, D21.4, D10.2, D10.2	**NONE**
EOFxx	C2.1	(C22)	001	00010	-K28.5,Dn.xxx0		+K28.5,Dn.xxx1		C5.0,Dn.xxx0 or C5.0,Dn.xxx1	**NONE**
Follows K28.1 for ESCON Connect-SOF (Rx indication only)										
C-SOF	C7.1	(C27)	001	00111	001111	1000	110000	0111	C7.1	**NONE**
Follows K28.5 for ESCON Passive-SOF (Rx indication only)										
P-SOF	C7.2	(C47)	010	00111	001111	1000	110000	0111	C7.2	**NONE**
Code Rule Violation and SVS Tx Pattern										
Exception	C0.7	(CE0)	111	00000	100111	1000	011000	0111	C0.7	**NONE**
-K28.5	C1.7	(CE1)	111	00001	001111	1010	001111	1010	C5.0 or C1.7	K28.5+
+K28.5	C2.7	(CE2)	111	00010	110000	0101	110000	0101	C5.0 or C2.7	**NONE**
Running Disparity Violation Pattern										
Exception	C4.7	(CE4)	111	00100	110111	0101	001000	1010	C4.7	**NONE**

C7.1 sends the ESCON Connect-Start of Frame (SOF) delimiter and C7.2 sends the Passive-SOF delimiter. C0.7 sends a deliberate code rule violation and has the same effect as having the SVS (Send Violation Symbol) pin HIGH during a character transmission. C1.7 sends a negative K28.5 regardless of the current running disparity. The receiver will decode this as either a C5.0 if its current RD was negative or as a C1.7 if its current RD was positive. C2.7 sends +K28.5 with the receiver decoding this as either a C5.0 or a C2.7 if its current RD was negative. Lastly, C4.7 sends a deliberate Running Disparity violation pattern. All of these codes simplify controller design as well as assist with in-system testing. TAXI−275 does not have the ability to send any of these codes.

Reframing

In a serial transmission system, the receiving device must have a method of determining byte boundaries. In many systems a unique special character is used for this purpose. When the byte framer is active, the receiver looks for (frames on) K28.5 SYNC characters present in the data stream. This character must be unique, such that any valid combination of other bits within the transmission stream will not erroneously create this synchronization (SYNC) symbol. Transmission line errors may cause some of

the bits within the information stream to become changed in such a way that the bits produce an erroneous (alias) SYNC. If the receiver has single-byte framing, this will cause the receiver to become misaligned, with all subsequent data being decoded incorrectly.

Both the HOTLink and TAXI−275 devices have the capability for double byte framing. Both devices Reframe on two occurrences of the Special Character K28.5 separated by 0, 1, 2, or 3 words (0, 10, 20, or 30 bits) as shown in *Figure 9*.

The HOTLink RF pin is used to activate and deactivate the reframing option. This is useful in systems that wish to prevent byte misalignment from alias SYNCs during data packets. Byte misalignment will cause all subsequent data in a packet to be corrupted instead of just the word or words that were corrupted due to transmission errors. Single-byte reframing is active for the first 2K bytes after the RF pin is asserted HIGH. This feature allows the receiver to SYNC to the first K28.5. After 2K bytes during RF HIGH, double-byte reframing will be activated. When activated, the single-byte frame saves 10 mA.

HOTLink pads the spaces between data packets with SYNC characters. When the "No Enable" ($\overline{ENN}$ and $\overline{ENA}$ = HIGH) condition exists, the transmitter fills the unused bandwidth with K28.5s. This pad string should be identified at the receiver so that the receiving system is not forced to process this information.

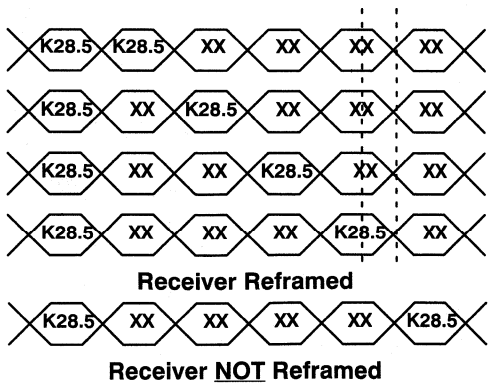


Figure 9. Double-Byte Reframing

TAXI−275 has no method of ignoring multiple SYNC characters and preventing them from being passed to the receiving system. The TAXI−275 STRBO pin pulses LOW in the presence of new Command or Data at the output register. It pulses LOW, therefore, every time a K28.5 character is received. If multiple SYNCs are passed to the outputs of the receiver, the receive FIFO will overflow with SYNC characters, which will require external decoder logic to discard this extraneous information.

HOTLink eliminates this problem by only pulsing the $\overline{RDY}$ pin LOW during the last SYNC character in a string of SYNC characters (the first SYNC character of a new packet of information). This is important in systems that have bursty data transmission or transmit data slower than the maximum data operating frequency. This prevents redundant information from being passed to the receive system, yet maintains packet boundaries for easy packet identification.

Higher Operating Frequency

HOTLink has a much broader frequency range than the TAXI−275 devices. TAXI−275 operates from 175 to 275 MBaud. This means that in 8B/10B mode, TAXI−275 can transmit and receive parallel data at rates from 17.5 to 27.5 MBytes/s. HOTLink, on the other hand, can transmit and receive parallel data at rates from 16 to 33 MBytes/s, allowing a much wider possible range of operating frequencies.

BIST

BIST (Built-In Self-Test) can be used to test the transmitter, receiver, and the link connecting them. During BIST (See *Figure 10*), the transmitter repeats a pattern representing all possible data and command characters, decodes them into transmission symbols and passes them to its outputs. The receiver, while in BIST, waits for the symbol that represents the beginning of the BIST pattern. It then decodes this and every following symbol and compares them with an internally generated pattern created by a pattern generator that matches the transmitter pattern generator. Detected errors at the receiver are indicated with pulses on the RVS (Received Violation Symbol) while completed BIST loops are indicated with pulses on the receiver $\overline{RDY}$ line. The BIST function checks the entire

function of the transmitter (except the transmitter input pins and the bypass function in the Encoder), the serial link, and the receiver.

These Built-In Self-Test functions are not implemented in the TAXI−275 devices. A substantial amount of additional circuitry is required in a system in order to integrate this function. This type of testing is necessary for many types of in-system diagnostic testing, including device functionality and link integrity.

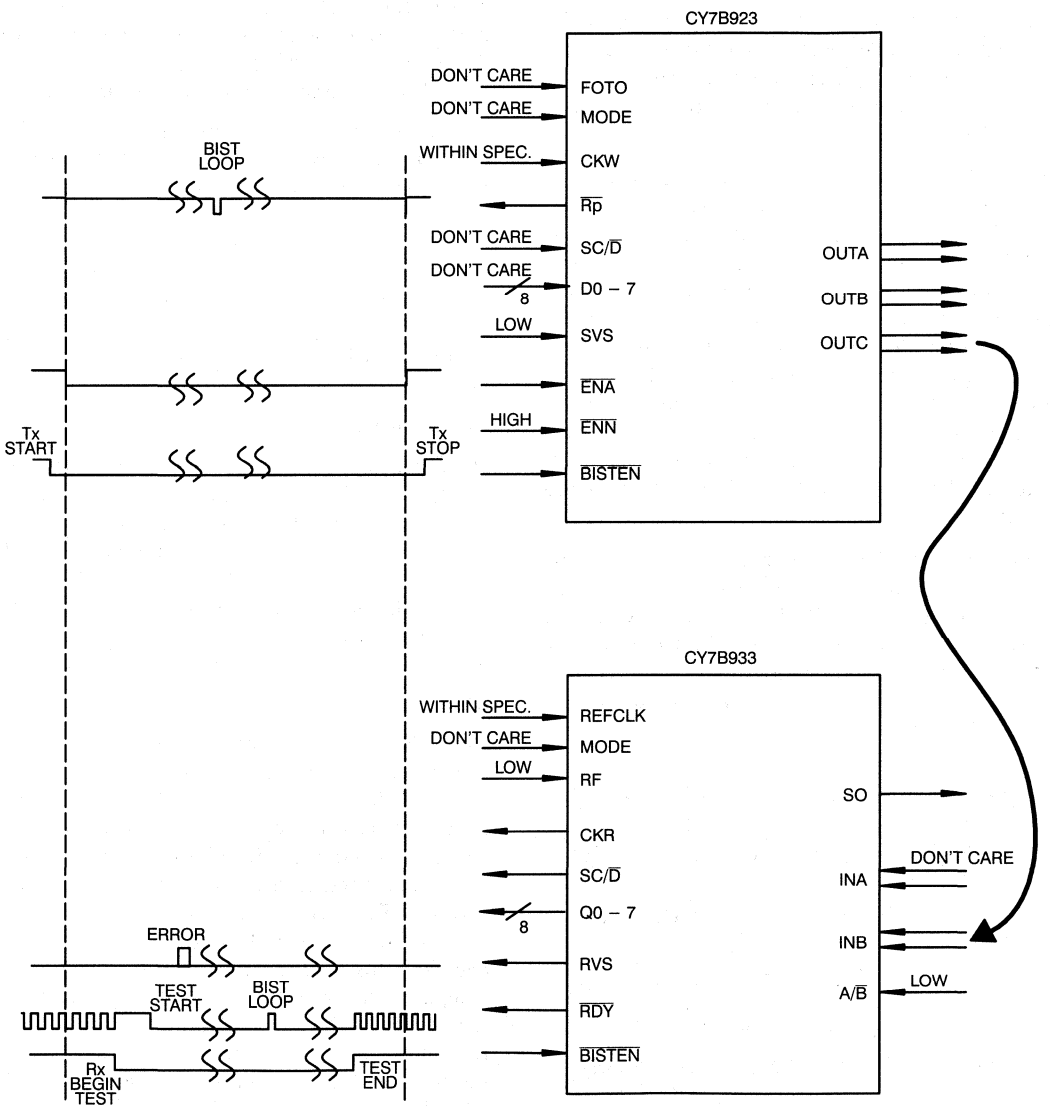


Figure 10. Built-In Self-Test

Parallel Interface

The TAXI−275 devices have two methods of strobing data into the device, synchronous and asynchronous. In the asynchronous mode of operation, a strobe line is used in conjunction with an acknowledge line to present data to the device. In this mode of operation the maximum operating frequency for the TAXI−275 devices under the most ideal of conditions is no faster than 20 MHz.

In the synchronous mode of operation, which is the most common method of device operation, the TAXI−275 device requires that the STRBI (Input Strobe) and the CLKI (Input Clock) be tied together. To enable or disable data in this mode requires external logic with slower than optimal (<275 Mbaud) operation. HOTLink has a very simple interface that allows seamless connection to both asynchronous and clocked FIFOs. On the transmitter, two enable inputs control when data is to be transmitted. When the $\overline{\text{ENA}}$ input is asserted, data on the data lines is serialized and transmitted. When the $\overline{\text{ENN}}$ line is asserted, data that is presented on the data lines during the next rising edge of the CLK input is transmitted. This allows efficient, synchronous state machines to control the flow of data over the serial link. In addition, the $\overline{\text{RP}}$ (read pulse) output can be connected to the $\overline{\text{R}}$ (read) input of asynchronous FIFOs, as shown in *Figure 11*, to provide a seamless asynchronous interface. The $\overline{\text{RP}}$ signal has timing that matches the timing required by asynchronous FIFOs. For clocked FIFO designs like that shown in *Figure 12*, the $\overline{\text{ENN}}$ input is used to not only read data from a Clocked FIFO like the Cypress CY7C443, but also to latch data into the Transmitter on the next rising edge of CKW.

The receiver has a $\overline{\text{RDY}}$ output that pulses LOW each time new data has been received. The $\overline{\text{RDY}}$ output has timing that allows the receiver to be seamlessly interfaced with both asynchronous and clocked FIFOs as shown in *Figures 11* and *12*. The TAXI−275 devices require a significant amount of additional circuitry to allow interfacing with FIFOs.

DC Specifications

The maximum current specification of the TAXI−275 Transmitter operating at 27.5 MB/s is 255 mA. The maximum current specification of the HOTLink Transmitter at 33 MB/s is only 80mA.

The TAXI−275 Receiver requires a maximum of 390 mA to operate at 27.5 MB/s whereas the HOTLink Receiver requires only 150 mA when operating at 33 MB/s.

Additionally, the TAXI−275 devices require 100 mV of differential input voltage at the receiver to accurately recover the clock and data from the input serial data stream. The HOTLink Receiver requires only 50 mV of differential input voltage. This translates into lower error rates, increased noise margins, higher jitter tolerance, and longer transmission distances when compared with the TAXI−275 devices.

Sending Violations

In many systems it is important to explicitly send violations. In normal system operation, a violation can be caused by either a received symbol having no corresponding decode value in the receiver, or a valid code received with the wrong Running Disparity. It is useful to send violation codes for testing, signaling, and interrupting the receiving system. The TAXI−275 devices have no method of code rule or Running Disparity violations. The HOTLink Transmitter, on the other hand, can send a pattern that will translate into a Code Rule Violation (C0.7) or Running Disparity Violation (C4.7) at the receiver. These Violations are indicated with a HIGH state on the RVS output with a Code Rule Violation indicated with command code C0.7 and a Running Disparity Violation indicated with command code C4.7. In addition, the SVS pin can be used to send a Code Rule Violation with the same indication at the Receiver.

ECL-to-TTL Translator

The TAXI−275 device does not include an ECL-to-TTL translator. The HOTLink Receiver has a built-in ECL-to-TTL translator where the SI input takes the single-ended ECL 100K (+5V referenced) signal in and the translated TTL signal is presented at

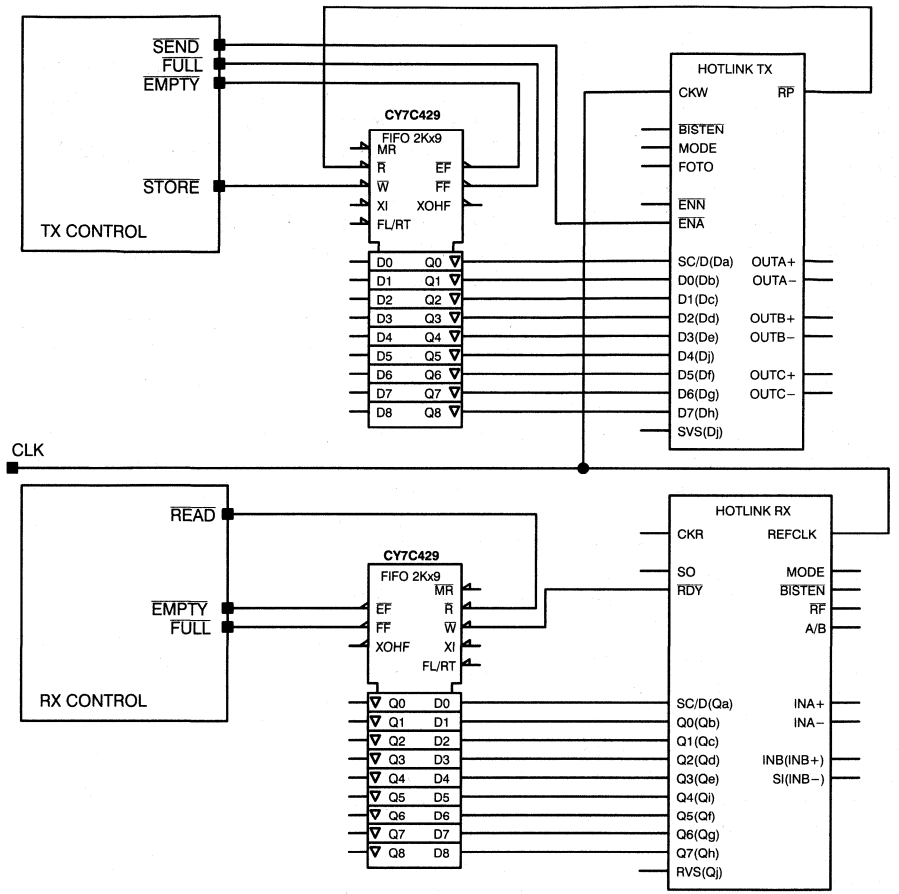


Figure 11. Asynchronous FIFO Interface

the SO output. The system can utilize this translator to convert an ECL carrier-detect signal from an optical module into its TTL equivalent for use by a controller.

Output Enable Considerations

The TAXI−275 devices use the $\overline{OE1}$ and $\overline{OE2}$ inputs to force the TX and TY outputs to their logic 0 state. A HIGH on $\overline{OE1}$ and a LOW on $\overline{OE2}$ will force TX LOW and TY HIGH. The analogous function on HOTLink is implemented with the FOTO (Fiberoptic Transmitter Off) pin. When the FOTO pin is held HIGH the OUTA+ and OUTB+ are forced LOW and the OUTA− and OUTB− outputs are forced HIGH. This causes a fiberoptic transmit module to extinguish its light output. The OUTC outputs are unaffected by the FOTO pin so that loop-back testing can be performed while the other outputs are turned off.

When the TAXI−275 $\overline{OE1}$ and $\overline{OE2}$ are both pulled HIGH, the TX and TY output drivers are turned off. This same result can be accomplished on HOTLink by either pulling both of the outputs of an output pair HIGH or simply leaving them unconnected. This will turn both outputs of an output pair off and save approximately 5 mA per output pair.

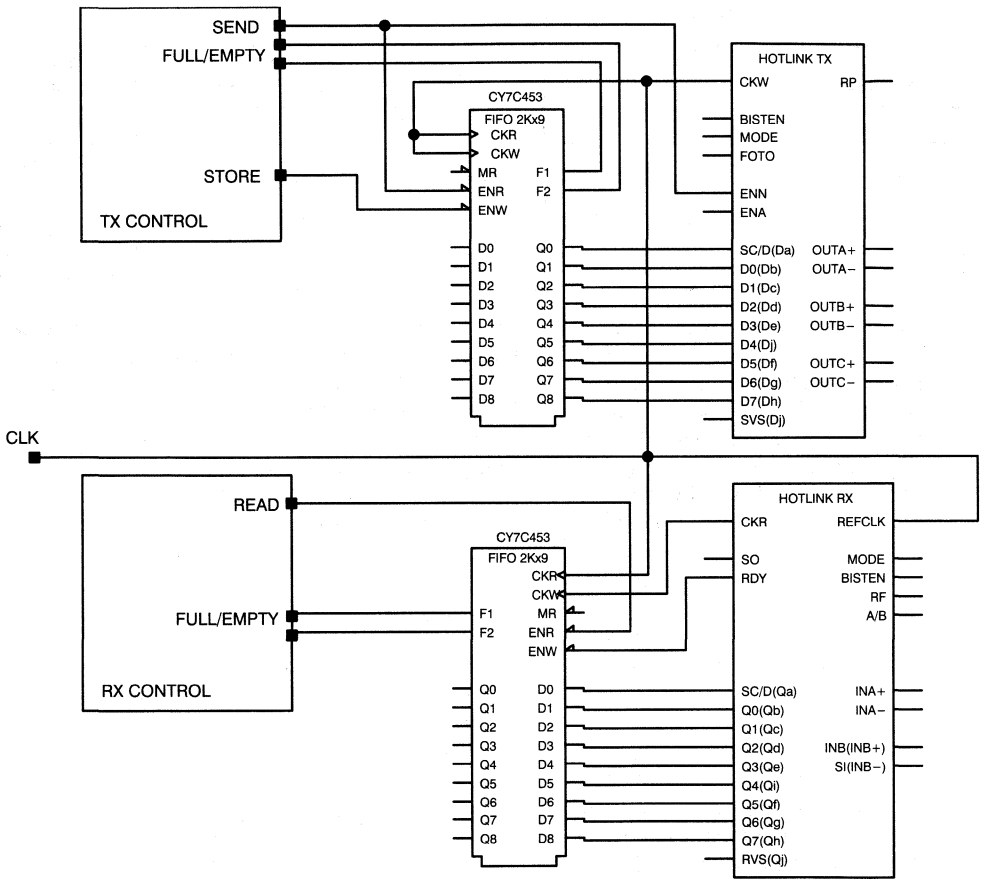


Figure 12. Clocked FIFO Interface

Status Indication

The TAXI−275 S1 and S2 status pins are used to indicate the status of the parallel output data as shown in *Table 2*.

Table 2. TAXI−275 Status Indication

Pin Status		Indication
S1	S2	
0	0	Data
1	0	Command
0	1	Violation
1	1	Re-Align

The SC/$\overline{D}$ (Special Character/$\overline{DATA}$), $\overline{RDY}$ ($\overline{Ready}$), and RVS (Receive Violation Symbol) outputs of HOTLink provide more status information than that provided by the TAXI−275 status pins as shown in *Table 3*. This table shows that Data and Command signalling on the HOTLink and the TAXI−275 devices are very similar. Violations, however, are indicated very differently between the two devices.

Table 3. HOTLink Status Indication

Function	SC/$\overline{D}$	$\overline{RDY}$	RVS	Q0−7	TAXI Indication
Data	0	0	0	Data	Data
Command	1	0	0	Command	Command
Code Rule Violation	1	0	1	C0.7	S1/S2=01
Running Disparity Violation	1	0	1	C4.7	S1/S2=01
Sync indication after reframe	1	0	1	C5.0	NONE

A Code Rule Violation is indicated with the SC/$\overline{D}$ pin HIGH, a LOW pulse on the $\overline{RDY}$ line, a HIGH on the RVS pin, and C0.7 on the data lines. A Code Rule Violation is a 10-bit transmission character that can not be decoded into an 8-bit symbol. Coding Violations are caused by errors during transmission across the link. A Running Disparity Violation is indicated in the same manner on the SC/$\overline{D}$, $\overline{RDY}$, and RVS pins as a Code Rule Violation, but the data output lines indicate the C4.7 command. A Running Disparity Violation is present when a transmission character is able to be decoded into an 8-bit symbol, but the transmission character had the wrong Running Disparity.

It is important that these two different types of violations are indicated separately to a controller. A Code Rule Violation indicates that the current symbol is corrupted. In this situation the controller would most probably throw away the erroneous word. A Running Disparity Violation, on the other hand, indicates that the current word probably is correct, but that at some point in the past the data became corrupted. In this situation, the controller would probably discard the entire packet.

Additionally, HOTLink provides a SYNC indication after entering reframing. When RF is brought high, the $\overline{RDY}$ line pulses low after the first SYNC character (K28.5) has been received. This feature assists the Reframe state machine in determining when the receiver has been reframed. The Reframe state machine could pull RF LOW after the SYNC indication. This would prevent alias SYNC characters from realigning the receiver. TAXI−275 devices do not have the ability to indicate when data has been framed to a K28.5.

The TAXI−275 devices only have an indication that the receiver was realigned (S1,S2=11). The status lines do not always indicate if the TAXI−275 Receiver has reframed when SYNCD is LOW. Only if the byte boundary has changed will the status pins change. A reframe controller, therefore, must monitor all of the command lines to determine if the receiver has correctly framed on the data stream. This complicates reframe state machine design.

Conclusion

HOTLink has many advantages when compared with the AMD Am79168 Transmitter and Am79169 Receiver (TAXI). These advantages include those listed below.

- Multiplexed command and data
- Three differential serial outputs
- Two differential serial inputs
- More flexible Command codes
- More flexible reframing
- Higher operating frequency
- Built-In Self-Test
- Simplified synchronous interface
- Reduced power consumption
- Ability to send violations
- Simplified output enable interface
- More complete receiver status indications
- ECL-to-TTL translator

These advantages of HOTLink provide greater system flexibility, simplified controller design, more reliable data communication, and lower power consumption.

References

1. Cypress Semiconductor, *CY7B923/CY7B933 HOTLink Transmitter/Receiver Preliminary Datasheet*, Cypress Semiconductor High Performance Data Book, August 1, 1993.

2. Cypress Semiconductor, *HOTLink Design Considerations Application Note*, October 1993.

3. Advanced Micro Devices, Am79168/Am79169−275 TAXI−275 Integrated Circuits Technical Manual, Rev. 1.0, 1993.

4. Advanced Micro Devices, Am79168/Am79169−275 TAXI−275 Transmitter/Receiver Transparent Asynchronous Transmitter/Receiver Interface Preliminary Data Sheet, March 1993 Rev B.

Document #: COM0001−1293

AMD, TAXI, and TAXIchip are trademarks of Advanced Micro Devices
HOTLink is a trademark of Cypress Semiconductor

Replace Your TAXI™-125 and TAXI-175

This application note will explain how to replace TAXIchip™ devices with the HOTLink™ devices from Cypress Semiconductor. This note begins with an introduction to HOTLink and then gives advantages and replacement suggestions for the TAXI-125 and TAXI-175 devices.

HOTLink Introduction

The HOTLink family of devices transfers data from point to point over high-speed serial links at 160 to 330 Mbits/second (*Figure 1*). The CY7B923 Transmitter (*Figure 2*) takes an 8-bit parallel data stream and encodes it using the Fibre Channel compliant and ESCON™-compliant 8B/10B code. This code maps all 8-bit data characters into a 10-bit transmission code that insures the transmission signal contains suitable transitions for recovery by the receiving device. The transmitter takes this 10-bit data word and converts it to a serial bit-stream and transmits it at 10 times the byte rate over a serial transmission link.

The CY7B933 HOTLink Receiver (*Figure 3*) lies on the other end of a transmission link that may consist of anything from a few inches of printed circuit-board trace to several kilometers of fiberoptic cable. The receiver decodes the incoming bit stream and reconstructs the original parallel data character, which is presented at the outputs aligned with the recovered clock. The receiver, in addition to these tasks, checks the incoming data stream for errors that may have occurred in the serial transmission.

The SC/$\overline{D}$ (Special Character/$\overline{Data}$) pin provides the ability to transmit command codes in addition to sending data characters. These codes are mapped

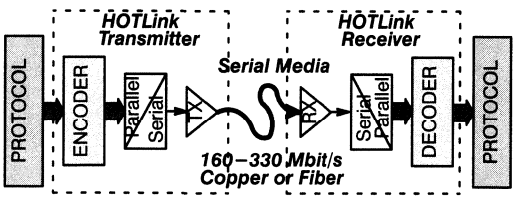


Figure 1. CY7B923 Transmitter Logic Diagram

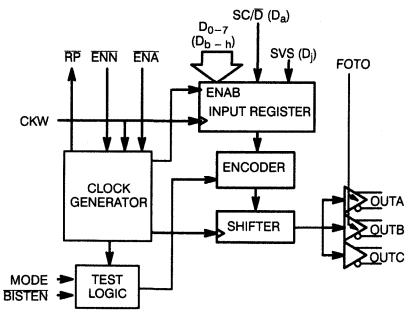


Figure 2. CY7B923 Transmitter Logic Diagram

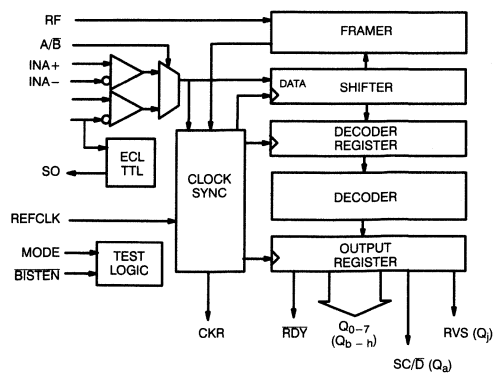


Figure 3. CY7B933 Receiver Logic Diagram

to 10-bit transmission characters defined in the 8B/10B codes of the Fibre Channel standard. This provides the ability to send commands as part of the transmission stream signaling events such as Idle, Start-of-frame, End-of-frame, etc.

Other features include Built-In Self-Test for in-system diagnostic testing, unencoded mode for sending 10-bit data in systems that use a different encoding method, and a seamless parallel interface for connection with both asynchronous and clocked FIFOs. A brief description of the various features of HOTLink is given below with a more detailed discussion found in the CY7B923/CY7B933 HOTLink Transmitter/Receiver datasheet. The PLCC pinouts for these devices are shown in *Figure 4*.

Replacement of TAXI−1 Devices

The following section shows how to upgrade a system using the TAXI−1 (Am7968/Am7969−125 or the Am7968/Am7969−175) with HOTLink. This section begins with a brief explanation of the TAXI−1 devices. It then shows how HOTLink simplifies systems that either use, or plan to use, these devices. It ends with a discussion on how to modify systems that use some of the features of these TAXI devices that are different from HOTLink.

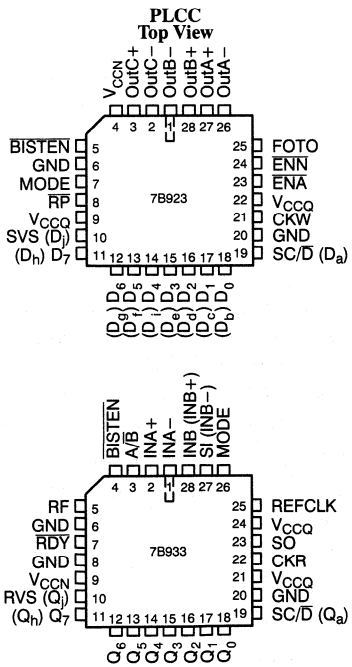


Figure 4. CY7B923 and CY7B933 Pin Configurations

Brief Description of TAXI−1

The Am7968/AM7969 provide a method of connecting systems over a serial link. These devices accept 8-, 9-, or 10-bit parallel data words on the transmitting side of the link (*Figure 5*) and convert the data to a serial bit stream using 4B/5B and 5B/6B NRZI (Non Return to Zero, Invert on 1s) codes. These codes convert 4 input bits into 5 transmission bits in the case of the 4B/5B code, or 5 input bits into 6 transmission bits in the case of the 5B/6B code. These codes insure that enough signal transitions occur on the link for the receiving device to recover the data. On the receiving end of the system (*Figure 6*), the serial data is decoded and presented to the outputs along with the recovered transmitted clock. The pinouts of the Am7968 TAXI Transmitter and Am7969 TAXI Receiver are shown in *Figure 7*.

Simplifying Your System with HOTLink

HOTLink offers an extensive feature list that provides a host of benefits when designing systems that perform point-to-point serial communication.

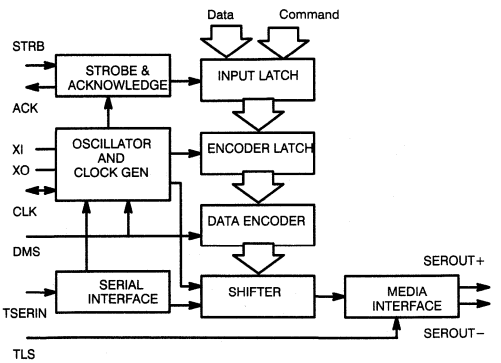


Figure 5. Am7968 Logic Diagram

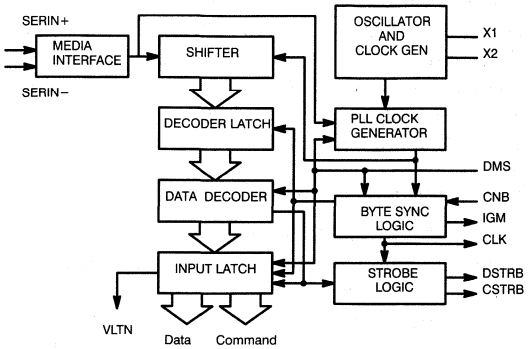


Figure 6. Am7969 Logic Diagram

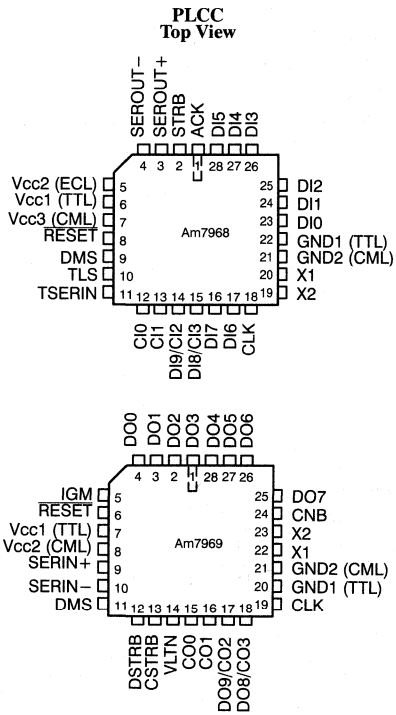


Figure 7. Am7968 and Am7969 Pin Configurations

While many of these features are offered in the TAXI−1 devices, they are difficult to use. These features include multiplexed command and data, multiple inputs and outputs, self-test operation, and many others. Below is a list of HOTLink features and advantage offered to the system designer when compared with the TAXI−1 devices.

Multiplexed Command and Data

One of the major differences between the HOTLink and the TAXI devices is the parallel data interface. The TAXI devices have separate inputs for command and data, while the HOTLink devices have an integrated command and data path. An external controller connected to the command inputs determines what command or data is to be sent. HOTLink determines if a Special Character (Command) should be sent by the status on SC/$\overline{D}$ pin (Special Character/$\overline{Data}$).

The integrated command and data paths of HOTLink allow a simpler, more conventional controller architecture. Instead of creating a separate command path, command codes can be integrated within the data stream and a ninth bit (the SC/$\overline{D}$ bit) can be added to indicate the status of the associated 8 bits of information.

More Serial Outputs

TAXI has one pair of differential ECL outputs. The HOTLink transmitter has three identical differential Pseudo ECL (PECL) serial output ports, any number of which can be disabled to conserve power. Additionally, two of these outputs may be switched off with the use of the FOTO (Fiberoptic Transmitter Off) pin. The AMD™ devices, on the other hand, have only one differential PECL output pair.

The additional HOTLink outputs can be used in a system to provide redundant data paths, for loopback testing, or for building complete networks where a single transmitter is received by multiple receivers.

More Serial Inputs

The TAXI Receiver has a single pair of differential inputs (SERIN±). The HOTLink Receiver has multiple interfaces to the serial transmission medium (INA± and INB±). As in the case of the

HOTLink Transmitter, the additional media inputs of the HOTLink Receiver can be used to provide loop-back testing, redundant transmission paths, or more complex network configurations. The TAXI SERIN− is used to control Test Mode function of the part. In addition to limiting the "in-circuit" test ability of TAXI, this limits the common mode range of the TAXI receiver.

Loop-back testing is easily accomplished with the multiple media interface features of the HOTLink devices. In a typical network-style configuration, both a transmitter and a receiver will exist for each node. By connecting one of the three output pairs of the transmitter to the the second input pair of the receiver as shown in *Figure 8*, the system gains the ability to perform a complete self-check upon system initialization or when an excessive amount of errors are received over the transmission link. HOTLink provides the ability to check the transmitting and receiving devices as well as the associated serial transmission link.

Neither complete system diagnostics nor integrated loop-back testing can be accomplished with the TAXI−1 devices. Many TAXI−1 based systems attempt this feature with single-ended ECL multi-

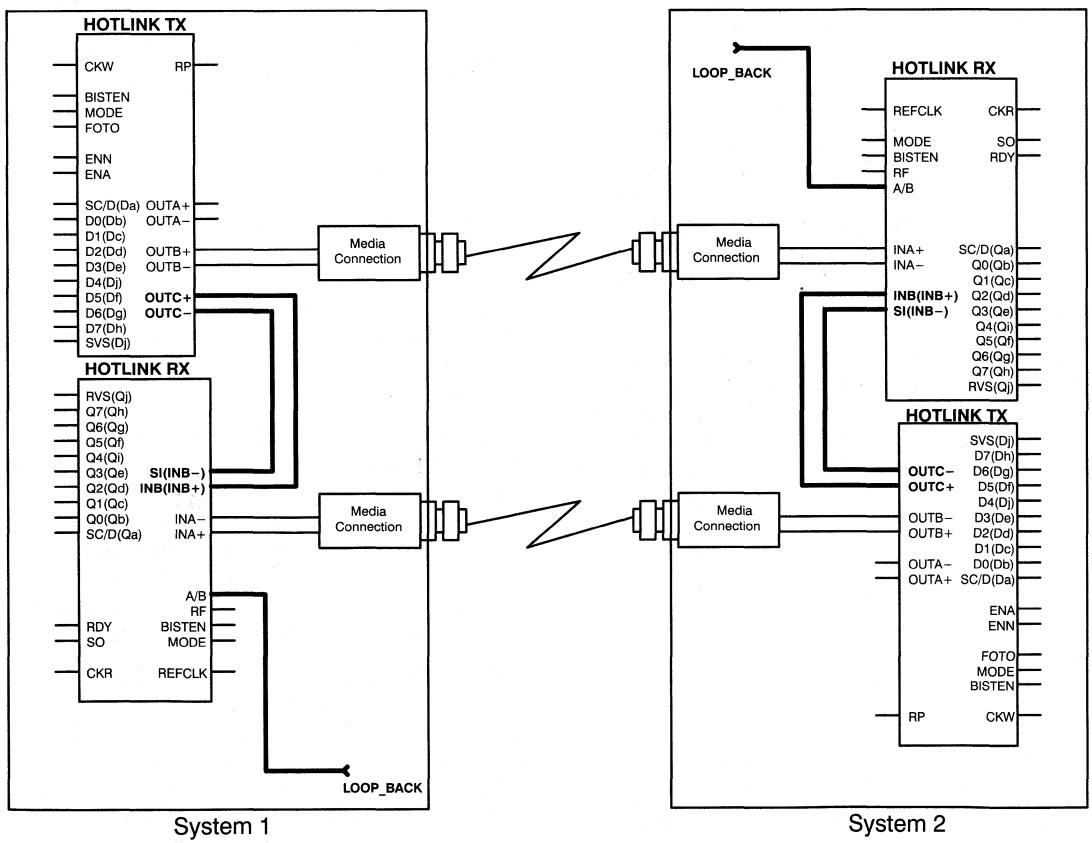


Figure 8. Example HOTLink Loop-Back System Connection

plexers, as shown in *Figure 9*. This solution compromises system reliability and performance.

Superior Data Encoding

Both the HOTLink and TAXI devices map the 8 bits of incoming transmission data into 10-bit transmission characters. The TAXI−1 devices accomplish this task by changing each pair of 4-bit nibbles of data into a pair of 5-bit transmission symbols according to the ANSI X3T9.5 (FDDI) standard. HOTLink, on the other hand, converts each 8 bits of data into a 10-bit transmission symbol according to the ANSI X3T9.3 Fibre Channel and ESCON (Enterprise System CONnection) specifications.

The primary purpose of converting the 8 data bits to 10 transmission bits is to include clock information in the data stream. A code is selected that maps each user character to a transmission character. This mapping insures that the data stream contains enough signal transitions to insure that the receiver PLL stays frequency and phase locked with the incoming data. By including the clock along with the data, the receiver is able to sample the incoming stream of data at the correct rate and position. For example, without this embedded clock information there would be no way of knowing if 1000, 999, or 1001 1s were sent in a row.

While the 4B/5B code used in TAXI−1 merely insures that the transition density of the serial bit stream is maintained, the 8B/10B code used in HOTLink also maintains the DC balance of the signal on the transmission line. This code maintains DC balance by insuring that, on the average, the number of 1s sent is equal to the number of 0s. This improves system performance by reducing the low-frequency "base-line" wander that causes jitter.

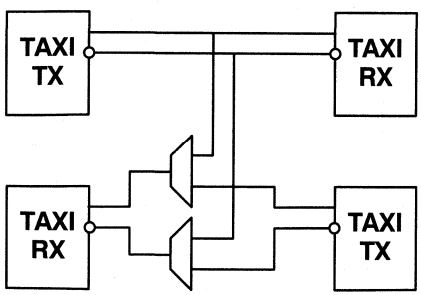


Figure 9. Loop-Back Testing with Multiplexers

More Robust Reframing Capabilities

To reassemble the incoming data stream into parallel data words, the receiver must know which bit location is the beginning of each byte. The transmitter must send SYNC characters to let the receiver know the location of byte boundaries. The TAXI−1 Transmitter sends a SYNC character when neither Data or Command is strobed into the part. At the receiver, this character is decoded as a command and the command strobe is pulsed.

The HOTLink Transmitter also sends a SYNC character when neither data nor command information is latched into the device. And again, at the receiver this character is decoded and the SC/$\overline{D}$ is held HIGH. HOTLink, however, differs in some very important ways from TAXI−1 devices.

TAXI−1 does not have a method for sending a SYNC character as part of the user character stream. HOTLink has a dedicated code that forces a SYNC character to be sent. This is important for controllers that wish to send a SYNC character at the beginning of each packet to insure that previous framing errors do not affect the current packet of data. This simplifies the controller and parallel data interface since the code can be embedded in a stream of other data.

Both TAXI−1 and HOTLink pad the spaces between data packets with SYNC characters. When the "No STRB" condition exists with TAXI−1 or the "No Enable" condition exists with HOTLink, the transmitter fills the unused bandwidth with JK (TAXI−1) or K28.5 (HOTLink). This pad string must be identified at the receiver so that the receiving system is not forced to process this information.

TAXI−1 has no method for ignoring multiple SYNC characters and preventing them from being passed to the receiving system. This is important in systems that have bursty data transmission or transmit data slower than the maximum TAXI−1 data operating frequency. If multiple SYNCs are passed to the outputs of the receiver, the receive FIFO will overflow with useless SYNC characters and this will

require external decoder logic to discard the extraneous information. HOTLink eliminates this problem by only presenting the last SYNC character in a string of SYNC characters (the first SYNC character of a new packet of information) to the outputs of the receiver. This prevents redundant information from being passed to the receive system, yet maintains packet boundaries for easy packet identification.

Occasionally transmission links will experience noise that transforms part of the information stream into a SYNC character (an alias SYNC). This may cause the receiver to incorrectly identify the byte boundary and cause all of the following information to be misframed. This will continue to occur until the transmitter sends an intentional SYNC symbol. The TAXI−1 devices have no method to prevent this unintended reframing. HOTLink can prevent this in two ways.

The first way HOTLink prevents misalignment is provided by its ability to disengage reframing under user control with the RF (reframe) pin. In systems that need reframing only between packets, or only during supervisory functions, the reframe option can be selectively activated or deactivated depending on the system needs.

The second way HOTLink prevents misalignment is provided by its multi-byte framing capability. After the initial start-up phase, approximately 2K bytes after reframe (RF=HIGH) has been activated, the receiver will no longer frame on just one SYNC character, but instead requires at least two SYNC characters separated by exactly 0, 10, 20, or 30 bits of valid data as shown in *Figure 10*.

The multi-byte reframe option is useful in systems that wish to keep the Reframe option activated continuously, but do not want to suffer the data corruption consequences of erroneous misalignment. Systems that stay connected for long communication sessions (e.g., point-to-point data recovery) rarely need to be Reframed since the receiver will rarely lose byte alignment. In these systems, the protocol, or an external timer, can control reframing and only enable framing when it is required. On the HOTLink Receiver this will save 50 mW of power.

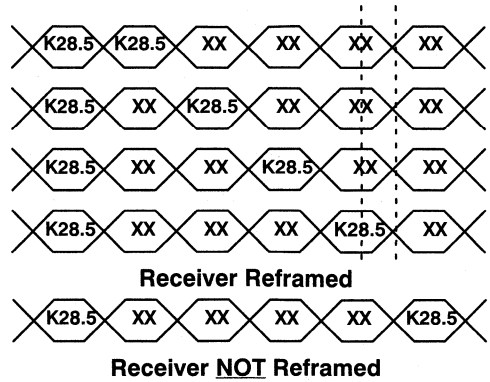


Figure 10. Double-Byte Reframing

For systems that are reconnected often (switched systems) the need to quickly reacquire byte synchronization requires that Reframe be continuously enabled. Multibyte framing available with HOTLink protects these systems from alias SYNC characters.

Higher Operating Frequency

HOTLink provides the biggest improvement in a system upgrade by allowing operation at nearly twice the rate of the TAXI−175 devices and nearly 2.5 times the rate of the TAXI−125s. The range of the TAXI−1 devices is 40 to 175 MBaud whereas HOTLink operates from 160 to 330 MBaud. This increased operating frequency range provides the ability to transfer data at over twice the rate of an equivalent TAXI system.

Built-In Self-Test Capabilities

BIST (Built-In Self-Test) can be used to test the transmitter, receiver, and the serial data link connecting them. During BIST (See *Figure 11*), the transmitter repeats a pattern representing all possible data and command characters, decodes them into transmission symbols and passes them to its outputs. The receiver, while in BIST, waits for the symbol that represents the beginning of the BIST pattern. It then decodes this symbol and every following symbol and compares it with an internally generated pattern that matches those produced by the transmitter's pattern generator. Error signals are indicated with pulses on the RVS (Received

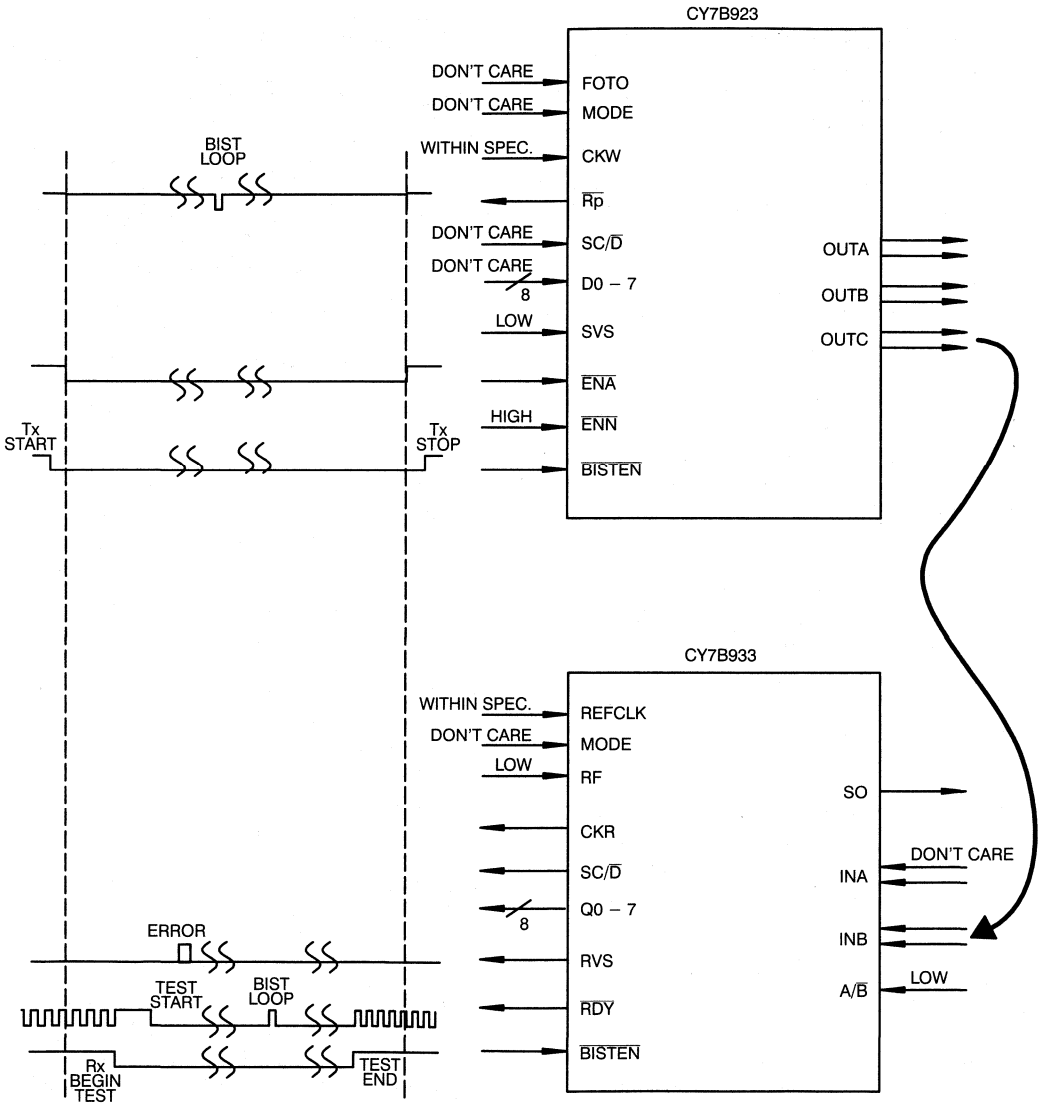


Figure 11. Built-In Self-Test

Violation Symbol) while completed BIST loops are indicated with pulses on the $\overline{RDY}$ line. The BIST function, therefore, checks the entire function of the transmitter (except the transmitter input pins and the bypass function in the Encoder), the serial link, and the receiver.

These functions can not be implemented with the TAXI devices. A substantial amount of additional circuitry would need to be added to a TAXI system to imitate this function. This type of testing is necessary for many types of diagnostics including device functionality and link integrity.

Simplified Synchronous Interface

The TAXI−1 devices have two methods of strobing data into the devices, synchronous and asynchronous. In the asynchronous mode of operation, a strobe line is used in conjunction with an acknowledge line to present data to the device. In this mode of operation the maximum byte-rate frequency for the TAXI−175 devices under the most ideal conditions is no faster than 14 MB/sec. The synchronous strobing feature of the TAXI−1 devices is also cumbersome. This method involves connecting the strobe to the clock line.

HOTLink has a very simple interface that allows seamless connection to both asynchronous and clocked FIFOs. On the transmitter, two enable inputs control when data is to be transmitted. When the $\overline{ENA}$ input is asserted, data on the data lines is serialized and transmitted. When the $\overline{ENN}$ line is asserted, data that is presented on the data lines during the next rising edge of the CLK input is transmitted. This allows efficient, synchronous state machines to control the flow of data over the serial link. In addition, the $\overline{RP}$ (read pulse) output can be connected to the $\overline{R}$ (read) input of asynchronous FIFOs, as shown in *Figure 12*, to provide a seamless asynchronous interface. The $\overline{RP}$ signal has timing that matches the timing required by asynchronous FIFOs. For clocked FIFO designs like that shown in *Figure 13*, the $\overline{ENN}$ input is used to read data from a Clocked FIFO like the Cypress CY7C453 as well as latch data into the transmitter on the next rising edge of CKW.

The receiver has an $\overline{RDY}$ output that pulses LOW each time new data has been received. The $\overline{RDY}$ output has timing that allows the receiver to be seamlessly interfaced with both asynchronous and clocked FIFOs as shown in *Figures 12* and *13*. The TAXI devices require a significant amount of additional circuitry to allow interfacing with FIFOs.

Better DC Specifications

The maximum current specifications of the TAXI−1 Transmitter operating at 17.5 MB/sec is 265 mA. The maximum current specification of the HOTLink Transmitter, on the other hand is 80 mA even when operating at 33 MB/sec.

The TAXI−1 Receiver requires a maximum of 350 mA to operate at 17.5 MB/sec whereas the HOTLink Receiver requires only 150 mA when operating at 33 MB/sec.

The TAXI−1 devices require 300 mV of differential input voltage at the receiver to accurately recover the clock and data from the input serial data stream. The HOTLink Receiver requires only 50 mV of differential input voltage. This translates into lower error rates, increased noise margins, higher jitter tolerance, and longer transmission distances when compared with the TAXI−1 devices.

Loop-Back Testing Capabilities

TAXI−1 has no loop-back testing capabilities. As mentioned previously, the redundant inputs and outputs on the HOTLink devices allow in-system loop-back testing to be performed. An additional output from the transmitter can be connected to an unused input of the receiver. The transmitter/receiver pair of an individual port can be tested together by simply switching the receiver from the link input to the Loop-back input as shown in *Figure 8*.

Ability to Send Violations

The TAXI−1 Transmitter has no method of sending violations. The TAXI Receiver has no unambiguous violation indication. Many, but not all, errors will be indicated as C0 (SYNC) while others will be indicated as other commands. In many systems it is important to explicitly send violations. In normal system operation, a violation can be caused by either a received symbol having no corresponding decode

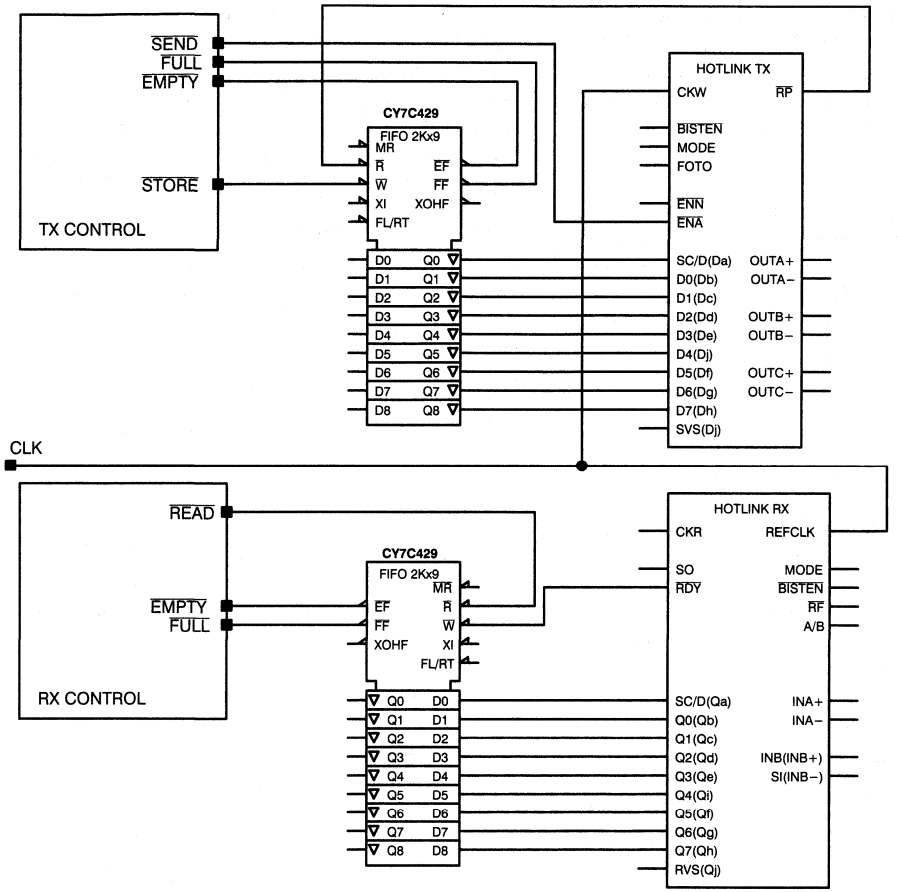


Figure 12. Asynchronous FIFO Interface

value in the receiver, or a valid code received with the wrong running disparity. Sending a violation code on purpose is useful for testing, signaling, and interrupting the receiving system.

The HOTLink Transmitter, on the other hand, provides two mechanisms to allow a system to send a pattern that will translate into a Code Rule Violation at the receiver. Various codes are included in the Special Character (SC) codes to send code rule and Running Disparity (RD) violations as part of the normal data stream. The SVS (Send Violation Symbol) pin allows an external supervisory system to force errors on an otherwise undisturbed data stream. Received errors are unambiguously indicated in the received data stream. All errors also generate an indication on the RVS (Received Violation Symbol) pin to be used by external supervisory logic.

Ability to Turn-Off Serial Output Stream

There is no method of turning off the serial output of the TAXI−1 devices. The FOTO (Fiberoptic Transmitter Off) is an input found on the HOTLink Transmitter that allows the OUTA and OUTB differential outputs to be logically turned off. Laser

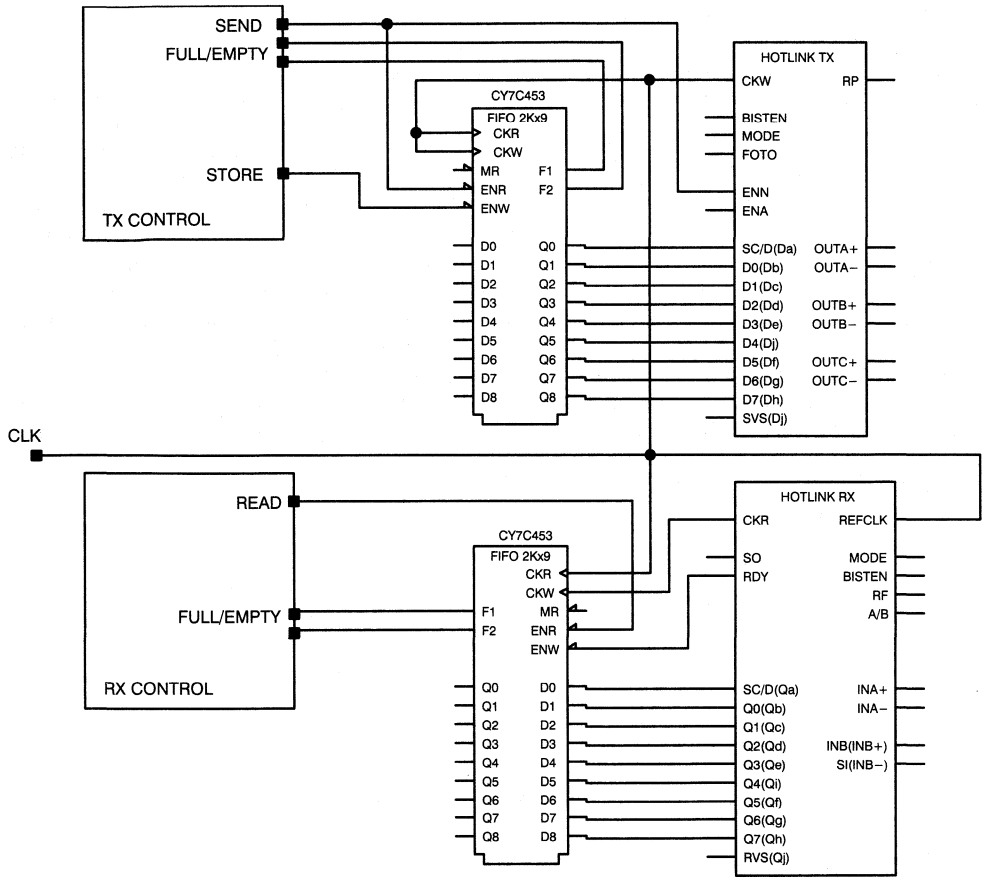


Figure 13. Clocked FIFO Interface

safety systems can use this input to shut off the lasers in the case of a fiber disconnect.

ECL to TTL Translator

The TAXI devices have no ECL to TTL translator. The HOTLink devices have a built-in ECL to TTL translator. The SI input takes the single-ended ECL 100K (+5V referenced) input and the translated TTL signal is presented at the SO output. The system can utilize this translator to convert a carrier-detect signal into its TTL equivalent for use by a controller.

Modifying the System for HOTLink

Listed below are some simple system modifications that can be performed in lieu of modifying the entire system architecture for designers who currently use the TAXI−1 devices and wish to easily upgrade to the HOTLink devices in order to take advantage of its performance and architectural improvements.

Multiplexed Command and Data

Most systems have, at some level, an integrated command and data path, much like that used in HOTLink. These systems explicitly demultiplex these paths to make themselves compatible with the TAXI architecture. These systems can easily take

advantage of the HOTLink architecture by removing the unnecessary multiplexing circuitry and allowing the demultiplexer control line and the data/command lines to drive HOTLink directly.

Some systems, however, send command codes out-of-band with respect to the data stream. These systems can be easily modified by adding a simple multiplexer external to HOTLink or external to the FIFO that drives HOTLink. The MUX select can be driven by the AND of the command lines.

Data Words Longer than 8 Bits

Most data words that need to be encoded are 8 bits in length. In a few cases, however, the data that needs to be encoded is 9, or even 10, bits in length. In these cases, an external multiplexer can be used external to a FIFO that would put each half of the 9- or 10-bit data word into the FIFO separately. At the receiving end, the same operation would be performed in reverse. This is possible due to the extended operating frequency of HOTLink.

Asynchronous Strobing

HOTLink provides a very user-friendly synchronous interface. For asynchronous operation, a FIFO can be used to interface the two asynchronous entities.

Mapping Command Codes

In 8-bit mode, TAXI-1 has 15 different command codes (see *Table 1*), while HOTLink can transmit and receive 12 specific codes (see *Table 2*). Several of these TAXI command codes have restrictions on their usage. HOTLink has no restrictions on the use of any codes. If the system must use more than 12 codes, an easy way to expand the command set is by utilizing a specific code that indicates that the next data word is also a command code. Using this method, or a simple extension of this method, allows nearly an infinite command code set to be transmitted and received.

Operating Frequency

The operating frequency of HOTLink is much faster than the TAXI devices. No design issues need to be considered in systems that wish to operate their parallel side at the same rate and take advantage of the increased system flexibility and functionality that HOTLink offers. When the system has no data to send over the transmission link, HOTLink simply sends strings of SYNC characters automatically. These SYNC characters are ignored on the receiving end. So, whenever the transmitting side of the link does not present data to the transmitter, a SYNC character will be sent. These characters, although used to keep the receiver in lock with the transmission stream, will not be presented as a character to the outputs.

Conclusion

The HOTLink Transmitter and Receiver have many advantages over the AMD Am7968 Transmitter and the Am7969 Receiver (TAXI−1). These advantages include those listed below.

- Multiplexed command and data
- More serial outputs
- More serial inputs
- Superior data encoding
- More robust reframing
- Higher operating frequency
- Built-in self-test
- Simplified synchronous interface
- Reduced power consumption
- Loop-back testing capabilities
- Ability to send violations
- Ability to turn off serial output stream
- ECL-to-TTL translator

Table 1. TAXIchip Command Symbols

Am7968 Transmitter Command Input				Am7969 Receiver Command Output	
HEX	**Binary**	**Encoded Symbol**	**Mnemonic**	**HEX**	**Binary**
8-Bit Mode					
0	0000	XXXXX XXXXX	Data	No Change	No Change
No STRB	No STRB	11000 10001	JK (8-bit Sync)	0	0000
1	0001	11111 11111	II	1	0001
2	0010	01101 01101	TT	2	0010
3	0011	01101 11001	TS	3	0011
4	0100	11111 00100	IH	4	0100
5	0101	01101 00111	TR	5	0101
6	0110	11001 00111	SR	6	0110
7	0111	11001 11001	SS	7	0111
8[1]	1000	00100 00100	HH	8	1000
9	1001	00100 11111	HI	9	1001
A[1]	1010	00100 00000	HQ	A	1010
B	1011	00111 00111	RR	B	1011
C	1100	00111 11001	RS	C	1100
D[1]	1101	00000 00100	QH	D	1101
E[1]	1110	00000 11111	QI	E	1110
F[1]	1111	00000 00000	QQ	F	1111

Note

1. While these Commands are legal data and will not disrupt normal operation if used occasionally, they may cause data errors if grouped into recurrent fields. Normal PLL operation cannot be guaranteed if one or more of these commands is continuously repeated.

Table 2. HOTLink Valid Special Character Codes and Sequences (SC/$\overline{D}$ = HIGH)

HOTLink Special Code Byte Name	Special Code Code Name		Bits HGF	Bits EDCBA	Current RD− abcdei	Current RD− fghj	Current RD+ abcdei	Current RD+ fghj	Receiver Output Code Name
K28.0	C0.0	(C00)	000	00000	001111	0100	110000	1011	C0.0
K28.1	C1.0	(C01)	000	00001	001111	1001	110000	0110	C1.0
K28.2	C2.0	(C02)	000	00010	001111	0101	110000	1010	C2.0
K28.3	C3.0	(C03)	000	00011	001111	0011	110000	1100	C3.0
K28.4	C4.0	(C04)	000	00100	001111	0010	110000	1101	C4.0
K28.5	C5.0	(C05)	000	00101	001111	1010	110000	0101	C5.0
K28.6	C6.0	(C06)	000	00110	001111	0110	110000	1001	C6.0
K28.7	C7.0	(C07)	000	00111	001111	1000	110000	0111	C7.0
K23.7	C8.0	(C08)	000	01000	111010	1000	000101	0111	C8.0
K27.7	C9.0	(C09)	000	01001	110110	1000	001001	0111	C9.0
K29.7	C10.0	(C0A)	000	01010	101110	1000	010001	0111	C10.0
K30.7	C11.0	(C0B)	000	01011	011110	1000	100001	0111	C11.0
Sequences									
Idle	C0.1	(C20)	001	00000	-K28.5+,D21.4,D21.5,D21.5, repeat				C5.0, D21.4, D21.5, D21.5
R_RDY	C1.1	(C21)	001	00001	-K28.5+,D21.4,D10.2,D10.2, repeat				C5.0, D21.4, D10.2, D10.2
EOFxx	C2.1	(C22)	001	00010	-K28.5,Dn.xxx0		+K28.5,Dn.xxx1		C5.0,Dn.xxx0 or C5.0,Dn.xxx1
Follows K28.1 for ESCON Connect-SOF (Rx indication only)									
C-SOF	C7.1	(C27)	001	00111	001111	1000	110000	0111	C7.1
Follows K28.5 for ESCON Passive-SOF (Rx indication only)									
P-SOF	C7.2	(C47)	010	00111	001111	1000	110000	0111	C7.2
Code Rule Violation and SVS Tx Pattern									
Exception	C0.7	(CE0)	111	00000	100111	1000	011000	0111	C0.7
-K28.5	C1.7	(CE1)	111	00001	001111	1010	001111	1010	C5.0 or C1.7
+K28.5	C2.7	(CE2)	111	00010	110000	0101	110000	0101	C5.0 or C2.7
Running Disparity Violation Pattern									
Exception	C4.7	(CE4)	111	00100	110111	0101	001000	1010	C4.7

References

1. Cypress Semiconductor, *CY7B923/CY7B933 HOTLink Transmitter/Receiver Preliminary Datasheet*, Cypress Semiconductor High Performance Data Book, August 1, 1993.
2. Cypress Semiconductor, *HOTLink Design Considerations Application Note*, October 1993.
3. Advanced Micro Devices, TAXIchip Integrated Circuits Transparent Asynchronous Transmitter/Receiver Interface Am7968/Am7969−125 Am7968/Am7969−175 Data Sheet and Technical Manual, 1992
4. Advanced Micro Devices, Am79168/Am79169−275 TAXI−275 Integrated Circuits Technical Manual, Rev. 1.0, 1993.
5. Advanced Micro Devices, Am79168/Am79169−275 TAXI−275 Transmitter/Receiver Transparent Asynchronous Transmitter/Receiver Interface Preliminary Data Sheet, March 1993 Rev B.

Document #: COM0008−0194

AMD, TAXI, and TAXIchip are trademarks of Advanced Micro Devices.
HOTLink is a trademark of Cypress Semiconductor.
ESCON is a trademark of International Business Machines.

Parallel Cyclic Redundancy Check (CRC) for HOTLink™

Introduction

This note discusses using CRC codes to insure data integrity over high-speed serial links, such as Fibre Channel, ESCON™ and other standards supported by Cypress's CY7B923 and CY7B933 HOTLink™ devices. It also shows why parity is not useful and then describes the most common CRC codes (CRC−16 and CRC−32) used in high-speed communications systems. Finally, an algorithm for calculating CRC−16 one byte at a time and for calculating CRC−32 two bytes at a time is presented. These algorithms are necessary because of the impracticality of running programmable devices at the 155 to 330 MHz required to calculate the CRCs in a bit-serial fashion.

Why Not Parity (or Why Some Parallel Interface Practices Don't Apply in the Serial World)?

Some systems go to great lengths to detect data corruption. Parity is sometimes used to provide a small measure of robustness by detecting certain bit errors. But, while parity can detect single-bit errors in data, it can detect only half of all multiple-bit errors in the same data. Other systems go further, employing CRC codes to not only detect, but in many instances correct, single- or even multi-bit errors. (Error Detection and Correction (EDAC) circuits use CRC codes.) Both of these approaches are applied to data in its parallel form.

It turns out that double-bit errors are the most prevalent error mode for single-event (burst) errors in high-speed serial channels, so parity is not very useful. Since an error of any kind is rare in typical systems (1 in 10^{12} bits), errors of greater than two bits are only a small portion of this number, which leaves us with predominantly double-bit errors with which to deal.

CRC−16 and CRC−32

In general CRC codes will detect:

- all single- and double-bit errors
- all odd numbers of errors
- all burst errors less than or equal to the degree of the polynomial used
- most burst errors greater than the degree of the polynomial used

So, CRC codes and their ability to detect multiple-bit errors are more suitable to systems employing high-speed serial links. Much work has been done with CRC codes and two suitable codes are CRC−16 and CRC−32. As the names imply, CRC−16 is 16 bits long and CRC−32 is 32 bits long. Additional information can be found in the references at the end of this note.

CRC−16 has a generator polynomial $G(x) = x^{16} + x^{15} + x^2 + 1$ and CRC−32 has a $G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$. The traditional way of calculating these CRCs is with Linear Feedback Shift Registers (LFSR) which are built from flip-flops and XOR gates, as shown in *Figure 1*. CRC−32 is similar. The superscripts over the flip-flops indicate the term in the CRC polynomial. For example, the 16 superscript indicates the flip-flop representing x^{16}. Be-

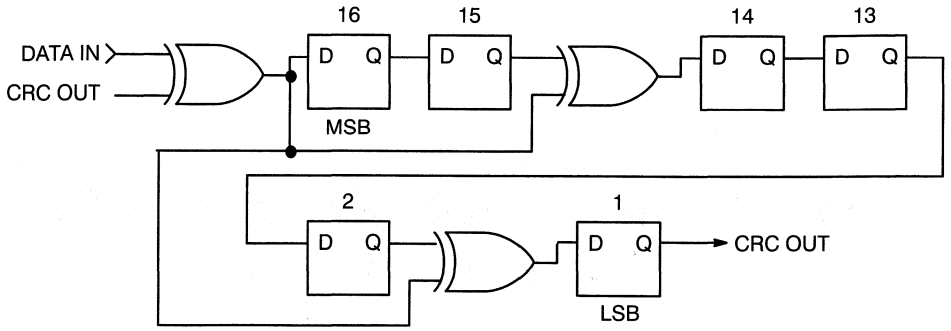


Figure 1. Linear Feedback Implementation of CRC−16

tween flip-flop 2 and 13 are nothing but flip-flops (no XOR gates).

Development of Parallel Algorithm

Following is a description of the algorithm and the results for calculating CRC−16 eight bits at a time (on a byte-by-byte basis). The procedure is similar for the CRC−32, except the results are calculated 16 bits at a time (on a word-by-word basis). The results for CRC−32 will be presented, but the actual calculations will not. This will be left as the proverbial "exercise for the reader". First, a few notes:

- R_i is the *i*th bit of the CRC register.
- C_i is the *i*th bit of the *initial* CRC register, before any shifts have taken place.
- Bit 1 is the least significant bit (LSB).
- The entries under each CRC register bit indicate the values to be XORed to generate the contents of that particular CRC register.
- A substitution will be made after eight shifts: $X_i = D_i$ XOR C_i

The results of the CRC are being calculated one bit at a time and the resulting equations for each bit are being examined. The CRC register after zero shifts is shown in *Table 1*. The CRC register after one shift is shown in *Table 2*. The CRC register after two shifts is shown in *Table 3*.

This process continues until eight shifts have occurred. *Table 4* lists the CRC register contents after eight shifts. X_i was substituted for the various D_i XOR C_i combinations. The following properties were used to simplify the equations:

Table 1. CRC−16 Register after Zero Shifts

R16	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
C16	C15	C14	C13	C12	C11	C10	C9	C8	C7	C6	C5	C4	C3	C2	C1

Table 2. CRC−16 Register after One Shift

R16	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
C1 D1	C16	C15 C1 D1	C14	C13	C12	C11	C10	C9	C8	C7	C6	C5	C4	C3	C2 C1 D1

Table 3. CRC−16 Register after Two Shifts

R16	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
C2 C1 D1 D2	C1 D1	C16 C2 C1 D1 D2	C15 C1 D1	C13	C12	C11	C10	C9	C8	C7	C6	C5	C4	C3	C3 C2 C1 D1 D2

Table 4. CRC−16 Register after Eight Shifts with X_i Substitution

R16	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
0 X8 X7 X6 X5 X4 X3 X2 X1	0 X7 X6 X5 X4 X3 X2 X1	0 X8 X7	0 X7 X6	0 X6 X5	0 X5 X4	0 X4 X3	0 X3 X2	C16 X2 X1	C15 X1	C14	C13	C12	C11	C10	C9 X8 X7 X6 X5 X4 X3 X2 X1

- Commutativity (A XOR B = B XOR A)
- Associativity (A XOR B XOR C = A XOR C XOR B)
- Involution (A XOR A = 0)
- Identity (A XOR 0 = A)

Careful study of *Table 4* reveals two interesting facts:

- the least-significant byte (bits 1..8) of the CRC register is dependent on the initial lower eight bits of the CRC register, the input data byte, and the initial contents of the high-order bits of the CRC register.
- the most-significant byte (bits 9..16) of the CRC register is only dependent on XOR combinations of the initial low-order byte of the CRC register and the input byte. Note that zeros have been placed into the CRC register making use of the identity property.

A conclusion can be made that it is possible to shift the CRC register high-order byte into the low-order byte, throw away the low-order byte and XOR the CRC register with some 16-bit word to generate the new word. For example, calculating a new value for R9 is accomplished by calculating X3 and X2 and exclusive-ORing them together.

Description of CRC−16 Parallel Algorithm

Finally, the parallel algorithm for CRC−16 can be stated as follows:

1. Calculate the X_is by XORing the input byte with the least-significant byte of the CRC register.
2. Shift the CRC register eight bits to the right.
3. Use *Table 4* to calculate the 16-bit word to be XORed with the CRC register.
4. XOR the CRC register with the 16-bit word just calculated.

Repeat these four steps for every data byte in the data stream.

Implementation Issues for CRC−16 Parallel Algorithm

The key implementation issue for this algorithm is the calculation of the 16-bit register value defined in *Table 4*. Since the word is only dependent on eight inputs, it can be calculated and stored in a 256 x 8

EPROM look-up table. A 33-MHz maximum byte rate would dictate a total cycle time of 30 ns or less, which would then dictate a look-up table speed of significantly less than 30 ns. Such EPROMs, like the Cypress CY7C291A with a 15-ns access time, do exist.

Another approach is to calculate the XOR functions directly in logic, as one would do with a Field Programmable Gate Array (FPGA). Some quick calculations are in order. After the X_i values are calculated, one can see from *Table 4* that the largest XOR to be calculated is that for R1, which contains nine terms. A quick calculation of logic levels required in an FPGA can be made if the widest single-level XOR is known. The Cypress pASIC380 FPGAs can calculate a three-input XOR in a single logic cell. It can be shown that two levels of logic requiring a total of four logic cells would be required to calculate a nine-input XOR. Because a register would be feeding this XOR tree, and a register would be at the output of this XOR tree (as defined by the algorithm), the timing would be determined by internal delays in the device. The CRC−16 parallel algorithm can be implemented using *Warp3*™ (Cypress's VHDL-based FPGA design tool) in a CY7C384A FPGA. The design will run at the 33-MHz maximum parallel data rate supported by HOTLink. Design files can be found on the Cypress BBS.

Description of CRC−32 Parallel Algorithm

The parallel algorithm for CRC−32 is derived in the same manner as CRC−16; the only difference is the polynomial being used and the fact that data is now handled 16 bits (1 word) at a time, instead of 8 bits at a time. The CRC−16 algorithm can be modified into the CRC−32 and can be stated as follows:

1. Calculate the X_is by XORing the input byte with the least-significant word of the CRC register.
2. Shift the CRC register 16 bits to the right.
3. Use *Tables 5* and *6* to calculate the 32-bit word to be XORed with the CRC register.
4. XOR the CRC register with the 32-bit word just calculated.

Repeat these four steps for every data word in the data stream.

Table 5 contains the XOR information for the least-significant word (LSW) of the CRC−32 register after 16 shifts, and *Table 6* contains the XOR information for the most-significant word (MSW) of the CRC−32 register after 16 shifts. Again, note that the MSW only depends on XOR combinations of the initial lower-order bits of the CRC−32 register and input word. The LSW depends on XOR combinations of the initial lower-order bits of the CRC−32 register, the input word, and the initial MSW of the CRC−32 register.

Table 5. CRC−32 Register (LSW) after 16 Shifts with X_i Substitution

R16	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
C32	C31	C30	C29	C28	C27	C26	C25	C24	C23	C22	C21	C20	C19	C18	C17
X1	X1	X2	X1	X1	X2	X1	X1	X2	X1	X1	X1	X4	X3	X2	X1
X2	X3	X3	X2	X3	X3	X2	X3	X3	X2	X2	X5	X10	X9	X8	X7
X4	X4	X5	X4	X4	X5	X4	X4	X4	X3	X6	X11	X11	X10	X9	X8
X5	X6	X6	X5	X6	X6	X5	X5	X8	X7	X12	X12	X14	X13	X12	X11
X7	X7	X8	X7	X7	X7	X6	X9	X14	X13	X13	X15				
X8	X9	X9	X8	X8	X11	X10	X15	X15	X14	X16					
X10	X10	X10	X9	X12		X16	X16								
X11	X11	X14	X13												
X12	X15														
X16															

Table 6. CRC−32 Register (MSW) after 16 Shifts with X_i Substitution

R32	R31	R30	R29	R28	R27	R26	R25	R24	R23	R22	R21	R20	R19	R18	R17
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
X6	X5	X4	X3	X2	X1	X1	X3	X2	X1	X6	X5	X4	X3	X2	X1
X7	X6	X5	X4	X3	X2	X4	X9	X8	X7	X7	X6	X5	X4	X3	X2
X10	X9	X8	X7	X6	X5	X10	X10	X9	X8	X8	X7	X6	X5	X4	X3
X16	X15	X14	X13	X12	X11	X11	X11	X10	X9	X10	X9	X8	X7	X6	X5
	X16	X15	X14	X13	X12	X12	X13	X12	X11	X11	X10	X9	X10	X7	X6
		X16	X15	X14	X13	X14	X14	X13	X12	X13	X12	X11	X11	X9	X8
				X16	X15	X15	X16	X15	X14	X14	X13	X12	X13	X10	X9
					X16			X16	X15	X16	X15	X14	X14	X12	X11
											X16	X15	X15	X13	X12
												X16		X14	X13

Implementation Issues for CRC−32 Parallel Algorithm

The issues confronting a designer wishing to implement this algorithm are the same as those for the CRC−16 algorithm, except that the magnitude is increased. Implementation of the X_i calculation in a look-up table now requires 16 inputs and, since we are calculating 16 bits at a time, 16 outputs. This implies a 64K x 16 EPROM. The difference is that HOTLink takes parallel data at a maximum of 33 Mbytes per second, and we are calculating the CRC on two bytes at a time. This gives us approximately 2 clock cycles (60 ns) to perform the calculation. Using an EPROM look-up table requires an extra component, compared with implementing the entire design in an FPGA. So, let's look at the critical path for a pASIC380-based design.

Looking at *Tables 5* and *6* shows that the largest XOR is an 11-term function feeding CRC register R16. Again, given single-logic cell capabilities of a three-input XOR the longest calculation now requires three levels of logic cells. This is one more than required for the CRC−16 implementation, but remember, we have twice as much time to calculate the CRC−32.

As data rates increase, so will timing constraints, but for the data rates supported by the HOTLink devices (160 to 330 MBits/sec), the pASIC380 devices can successfully implement parallel calculation of CRC−32. A conservative estimate of the critical path delay is determined in the discussion that follows. The actual design files can be found on the Cypress BBS.

Figure 2 is a graphical representation of the logic delays in the XOR-tree portion of the CRC−32 design.

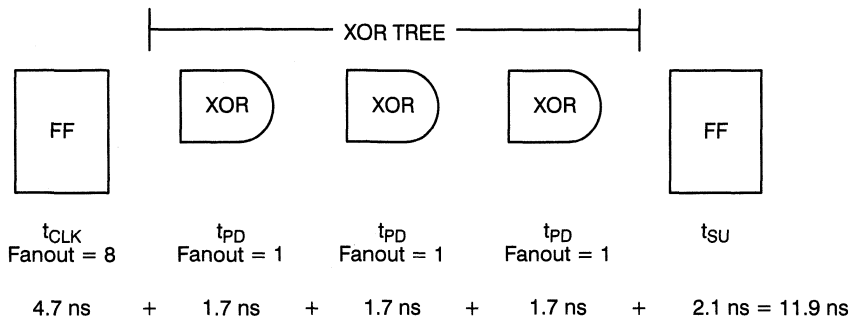


Figure 2. Critical Path Logic Delay Estimate for CRC−32

Note that routing delays are not included in this figure. Let's create a more realistic estimate:

1. round 11.9 ns to **12 ns**
2. add 50% of the logic delays for routing delays 12 ns + 0.5*12 ns = **18 ns**
3. find worst-case commercial timing by multiplying by the worst-case K factor for the −1 speed bin (18 ns x 1.33 = **24 ns**).

This is comfortably under the 60 ns figure mentioned earlier. All delay values, except for the 50% ratio mentioned above, are from the CY7C384A−1JC device datasheet.

Conclusion

This note has shown how to calculate a parallel implementation of any CRC polynomial and results for CRC−16 and CRC−32 were provided. Implementation issues were also discussed and performance data from implementing the byte-wise CRC−16 algorithm in a CY7C384A−1JC was presented. It easily ran at 33 MHz, the fastest parallel data rate supported by Cypress's HOTLink devices.

Estimated results for word-wise CRC−32 calculation in the CY7C384 were very close to the CRC−16 results reported. Since the speed requirements are halved, actual margins are greater than for the CRC−16.

Reference

1. A. Perez, "Byte-wise CRC Calculations," *IEEE MICRO*, June 1983, pp. 40−50.
2. A. K. Pandeya and T. J. Cassa, "Parallel CRC Lets Many Lines Use One Circuit," *Computer Design*, Sept. 1975, pp 87−91.
3. R. Swanson, "Understanding Cyclic Redundancy Codes," *Computer Design*, Nov. 1975, pp. 93−99.

Document #: COM0009−0394

HOTLink and *Warp3* are trademarks of Cypress Semiconductor.
ESCON is a trademark of IBM.

Fibre Channel Level 2 Design Considerations

Fibre Channel is emerging as an industry standard for high-speed point-to-point data communications. This Application Note explores the control, protocol, and hardware interface requirements between level 1 and level 2 of the Fibre Channel standard.

Fibre Channel Overview

Fibre Channel is a logical point-to-point serial data channel, intended to be used to freely intermix the transport of a wide variety of different protocols on a common medium. Explicitly supported protocols include SCSI, HIPPI, IPI3, and SBCCS. The flexibility of the structure allows ready adaptation of other protocols as well. Protocols are optimized for data block transfers between processors, storage systems, and output devices.

The logical point-to-point structure is supported by either a simple point-to-point topology or by a switching network, known as a Fabric. The Fabric, when used, functions as a logical crosspoint switch for dynamic routing of data channels.

The Fibre Channel standard specifies high bandwidth from 25 to 100 Mbytes per second on both fiber- and copper-based media.

Data Service Types

There are three classes of data services supported under the Fibre Channel standard, providing optimum efficiency for different network topologies.

Class 1 Data Service

Class 1 Data Service provides a dedicated connection between two network ports. This connection, once established, is maintained until removed. This data service class is intended for applications requiring relatively long, or even permanent connections between ports. This provides a facility for high-efficiency transfers, with minimal protocol overhead. There is an intermix option specified in the standard, which does allow other data service class connections to be made on an intermittent basis. This feature is intended to be used for utility and supervisory functions that normally would not significantly interfere with normal dedicated connection operations. Class 1 Data Service protocol operates with data acknowledgement, but does not require guaranteed data delivery.

Class 2 Data Service

Class 2 Data Service provides connectionless data transfer between two network ports with guaranteed data delivery. This data service, also known as multiplex class, is the most flexible in that multiple data transfers between multiple network ports is supported with out-of-order framing, if necessary, and guaranteed notification of non-delivery. In addition, provision is made to support multiple destinations for broadcast functions on the network. With the multiplex class, very dynamic network topologies can be implemented within the framework of a standard protocol.

Class 3 Data Service

Class 3 Data Service, usually called Datagram class, is also a connectionless class, but no provision is made for acknowledgement or notification of non-delivery. The class is intended to be used for applications where delivery of data is not time critical, or where minimizing loading on the network for the data transfer is required.

In summary, the combination of a point-to-point topology with flexible data service classes results in a network standard that is capable of supporting a wide variety of data communication applications.

Protocol Layers

There are five protocol layers specified in the Fibre Channel standard, and while this application note focuses on the interface requirements between layers 1 and 2, a brief summary of all the protocol layers will serve to place the discussion into proper context.

Layer 0

Layer 0 defines the physical link and media, with the transmitters and receivers with a wide variety of speeds. A link is defined as a pair of media, each operating with unidirectional data flow in opposite directions. The media pair then forms the basis for any communication model, ranging from simplex to full duplex operation. Both fiber and copper media, with appropriate connectors are specified in the standard.

Layer 1

Layer 1 defines the serial transmission protocol on the link. An adaptive 8B/10B run-length-limited code is used, which is optimized for DC balance on the media. Two types of characters are defined within the standard. An 8-bit data character is used for actual data transfer, and a set of 8-bit control characters, known as special characters, are used for control sequencing. There are a number of 4-character groups called ordered sets that contain combinations of special characters and data characters. The ordered sets are used within the protocol to identify frame boundaries and generate link control sequences. Transmitter and receiver state machines within Layer 1 are defined to establish and maintain synchronization and operational status of Layer 1 functions.

Layer 2

Layer 2 provides the link transport mechanism for data transfer. This includes the frame formats and link control. A port state machine is defined within Layer 2 and it is responsible for maintaining link functions. A set of primitive link protocols is defined to transparently establish and maintain logical link operations. In addition, Layer 2 is responsible for higher-level functions such as flow control, data routing, data sequence assembly and disassembly, fabric control and interface, and diagnostics.

Layer 3

Layer 3 is a higher-level protocol layer that defines the set of common logical services available to groups of ports within a network.

Layer 4

Layer 4 provides a standard definition of the channel mapping between the lower protocol layers and IPI and SCSI command sets, HIPPI data framing and other upper level protocols.

Protocol Layer Summary

The lower-level protocols through the lower levels of Layer 2 are usually implemented in hardware because of performance requirements. The more complex functions in Layer 2 will most likely require at least an embedded processor to handle the data management and upper-layer interfaces. Layers 3 and 4 are almost exclusively software based because of the protocol complexities, and will most likely be implemented on a host processor within the system. The focus of this application note will be confined to the hardware aspects of the Fibre Channel network, and will leave the software and processor aspects to a later discussion.

Ordered Sets

As previously stated, Fibre Channel defines special four-character groups, called ordered sets, which are used as the basic building block for frame structure and link signaling. All ordered sets begin with the special character K28.5, and are followed by a sequence of three standard data characters. There are three different types of ordered sets used:

- Frame Delimiters
- Primitive Signals

- Primitive Sequences

Frame Delimiters

Frame delimiters are used to denote the beginning and end of frames. There are a variety of start and end delimiters. Different start-of-frame delimiters are used to signal the data service class of the frame and identify the basic frame type. The frame type can be a Connect, Initiate, Normal, or Fabric. The different end-of-frame delimiters are used to further identify the frame type and advise status of the frame. The frame type and status for end-of-frame delimiters are Normal, Normal-Invalid, Terminate, Disconnect-Terminate, Disconnect-Terminate-Invalid, and Abort. The selection of the appropriate SOF and EOF is typically a higher-level Layer 2 function, determined by software.

Primitive Signals

Primitive signals are used for basic link control signaling. There are only two primitive signals. Receiver_Ready is used for basic flow control and when sent, signals that the sending port's receive buffer is now available for another frame. The other primitive signal is Idle, which is used for multiple purposes on the link. The Idle ordered set is used as a link filler when a port is operational, but has nothing else to send. The bit structure of Idle is optimized to enhance synchronization lock for clock recovery on the link, and thus, serves as an effective padding sequence between frames. There are requirements for multiple Idle signals between transmitted frames, and Receiver_Ready signals. In addition, when ports are establishing link-level communication with one another, the initial receipt of Idle signals is used to indicate that an operational link has been established. Generation and detection of primitive signals is a Fibre Channel Layer 1 function, and monitoring and reaction to the signals is a function of the port state machine in Layer 2.

Primitive Sequences

Primitive sequences are used by the Layer 2 link-control facility to initiate and maintain link communication with another port. The port state machines are responsible for implementing the actual primitive link protocol. The protocol is defined in the port state-machine description, later in this application note. A primitive sequence is intended to be sent repeatedly on the link until a state change in the port state machine occurs. Typically, the state transition occurs when a new primitive sequence is detected from the other port. Thus, the primitive sequences function as handshaking signals between the two ports. There are four primitive Sequences defined in Fibre Channel:

- Offline
- Not_Operational
- Link_Reset
- Link_Reset_Response

The character sets used for the ordered sets within Fibre Channel are defined in *Tables 1, 2,* and *3.*

Frame Structure

A frame is the basic data packet for data transfer within Fibre Channel, and the basic structure is illustrated in *Figure 1.* A frame is required to always be sent on a 4-byte boundary, and have a length that is an integer multiple of 4. The position of the 4-byte boundary is defined when initial link synchronization is established. In addition, a frame must always be preceded by at least two Idle Signals on the link. The Idle Signals are required to assure synchronized clock recover before the start of a new frame.

A frame consists of 5 elements, which are always present:

- Start Of Frame (SOF)
- Frame Header
- Data Field
- CRC
- End Of Frame (EOF)

Start Of Frame

The start of frame occurs first and is one of the start-of-frame ordered sets defined earlier. Fibre Channel requires that the running disparity always be negative for all start-of-frame ordered sets.

Table 1. Frame Delimiters

Delimiter	Ordered Set	
SOF Connect Class 1 (SOFc1)	K28.5 D21.5 D23.0 D23.0	
SOF Initiate Class 1 (SOFi1)	K28.5 D21.5 D23.2 D23.2	
SOF Normal Class 1 (SOFn1)	K28.5 D21.5 D23.1 D23.1	
SOF Initiate Class 2 (SOFi2)	K28.5 D21.5 D21.2 D21.2	
SOF Normal Class 2 (SOFn2)	K28.5 D21.5 D21.1 D21.1	
SOF Initiate Class 3 (SOFi3)	K28.5 D21.5 D22.2 D22.2	
SOF Normal Class 3 (SOFn3)	K28.5 D21.5 D22.1 D22.1	
SOF Fabric (SOFf)	K28.5 D21.5 D24.2 D24.2	
EOF Terminate (SOFt)	K28.5 D21.4 D21.3 D21.3	Neg RD
	K28.5 D21.5 D21.3 D21.3	Pos RD
EOF Disconnect-Terminate (SOFdt)	K28.5 D21.4 D21.4 D21.4	Neg RD
	K28.5 D21.5 D21.4 D21.4	Pos RD
EOF Abort (SOFa)	K28.5 D21.4 D21.7 D21.7	Neg RD
	K28.5 D21.5 D21.7 D21.7	Pos RD
EOF Normal (SOFn)	K28.5 D21.4 D21.6 D21.6	Neg RD
	K28.5 D21.5 D21.6 D21.6	Pos RD
EOF Disconnect−Terminate-Invalid (SOFdti)	K28.5 D10.4 D21.4 D21.4	Neg RD
	K28.5 D10.5 D21.4 D21.4	Pos RD
EOF Normal-Invalid (SOFni)	K28.5 D10.4 D21.6 D21.6	Neg RD
	K28.5 D10.5 D21.6 D21.6	Pos RD

Table 2. Primitive Signals

Primitive Signal	Ordered Set
Idle	K28.5 D21.4 D21.5 D21.5
Receiver_Ready (R_Rdy)	K28.5 D21.4 D10.2 D10.2

Table 3. Primitive Sequences

Primitive Sequence	Ordered Set
Off Line (OLS)	K28.5 D21.1 D10.4 D21.2
Not Operational (NOS)	K28.5 D21.2 D31.5 D5.2
Link Reset (LR)	K28.5 D9.2 D31.5 D9.2
Link Reset Response (LRR)	K28.5 D21.1 D31.5 D9.2

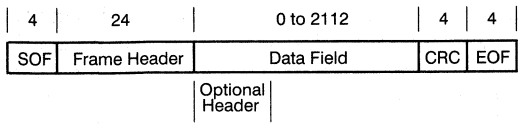


Figure 1. Data Frame Structure

Frame Header

The Frame Header consists of 24 bytes and is used to control link operations, provide destination and sequence identification of the frame, manage device protocol transfers, and support logical error detection in sequencing. The content of the frame header is determined by the higher-level elements in Layer 2, and thus, is typically built by a local software process, and is handled transparently by the Layer 1 and 2 hardware elements. The internal format of the frame header is provided for reference in *Figure 2*.

Data Field

The Data Field contains the actual payload for the frame and can range in size from 0 to 2112 bytes. Fibre Channel defines the use of optional headers within the data field, which can be used for auxiliary functions. The presence of optional headers is indicated within the frame header in the data field control byte. Because a Fibre Channel frame is required to have a length that is an integer multiple of four, data fields, which are not an integer multiple of four in length, are required to have fill bytes added. This can be one to three bytes of padding. The field control byte in the frame header contains indication of the presence of fill bytes in the data field. There is no provision in the frame format to explicitly indicate the size of the data field. The size must be computed from the distance between the start of the data field and the end-of-frame ordered set.

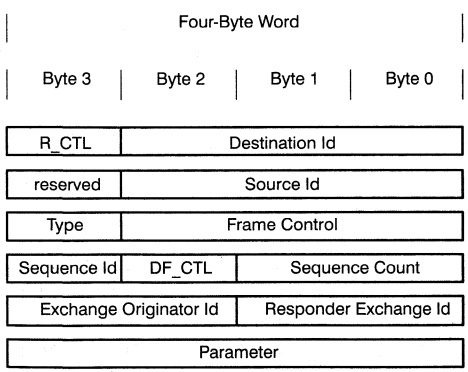


Figure 2. Frame Header Structure

CRC

The CRC is a 4-byte field that is used to verify the integrity of the frame. The CRC is computed across the entire frame, excluding the SOF and EOF ordered sets. The polynomial used is from the Fiber Distributed Data Interface (FDDI) standard. For reference, the polynomial is:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

End Of Frame

The End Of Frame field is an EOF ordered set defined previously. Because a Fibre-Channel frame is required to end with a negative running disparity, the EOF ordered set is required to have a modified running disparity in accordance with the accumulated running disparity of the entire frame, so that after the last byte of the EOF ordered set is sent, the running disparity will be negative. Provision must be made in Layer 1 hardware to accommodate this requirement.

Primitive Link Protocol

Primitive Link Protocol is part of the link control facility defined in Layer 2 of Fibre Channel and uses the primitive sequences described earlier. The protocol is intended to provide a mechanism where two Fibre Channel ports can reliably establish synchronization regardless of the initial state of the ports. At any point in time, a port can be in a variety of states that must be dealt with accordingly. The primitive link protocol defines nine possible states for a port:

Active	ACTIVE
Link Recovery Transmit	LRTX
Link Recovery Receive	LRRX
Link Reset Response Receive	LRRRX
Not Operational State Transmit	NOSTX
Not Operational State Receive	NOSRX

Off Line State Transmit	OLSTX
Off Line State Receive	OLSRX
Wait For OLS	WFOLS

The states are characterized by the primitive sequences to which the port has responded, or is transmitting. In general, a port will enter the state corresponding to the primitive sequence that is received, and a fixed protocol for further state transitions is defined to allow the port to migrate back to the Active state. In addition to responding to primitive sequences, the port is also required to monitor and react to Layer 1 status changes, such as loss of sync, or link failure, and to initiate primitive protocol sequences as needed by upper-level functions in Layer 2. These functions are typically requests for link reset, to enter Off Line or On Line state, or to initiate diagnostics. There is also a timeout function required for several of the states to allow detection of link failure, or loss of synchronization. A flowchart of the required state sequencing for the primitive link protocols is illustrated in *Figure 3*. The flowchart is relatively complex in detail, but simple in function. Interconnecting bubbles typically identify one of the nine possible states, and transitions from state to state occur on reception of an ordered set. The states and the transition sequences are dictated by the ANSI specification.

Port State Machine

This element implements the primitive link protocol (discussed previously) to establish and maintain link synchronization.

Layer 1 Receiver State Machine

The Receiver state machine in Layer 1 is responsible for establishing and maintaining bit and word synchronization of the incoming bit stream on the link. Bit synchronization establishes the byte boundary location in the incoming bit stream. This is accomplished through a reframing operation that synchronizes on the special character K28.5, which is defined as the first character in every ordered Set in the Fibre Channel standard. This facilitates bit-level synchronization on the link. Word synchronization is associated with establishing the location of the 4-byte boundary in the bit stream by means of synchronization with ordered sets detected in the bit stream. The word boundary, once established, is used as the reference point for all link protocol and data frame reception.

Fibre Channel defines a set of five states for the Layer 1 receiver state machine:

Sync Acquired State	SAS
Loss of Sync	LOS
Reset	RST
Loss of Sync Failure	LSYF
Loss of Signal Failure	LSGF

Sync Acquired State

This is the normal operational mode, where bit and word synchronization is established. To enter this state, the receiver is required to have established bit synchronization to some level of bit error rate, and detected three ordered sets with no errors. There are four sub-state transitions defined that must be traversed to enter the loss of sync state. The next sub-state is entered on detection of an invalid word condition so that four successive invalid words must be detected to enter the loss of sync state. The prior sub-state is reentered on detection of two valid words. The flowchart for this control sequence is provided in *Figure 4*. Note that the flowchart segment labeled "Test for Valid Word" is repeated three times in the Sync flowchart. An invalid word is detected under the following conditions:

- A code violation on a character basis
- A valid special character detected in the second, third, or fourth character position in a transmission word
- Detection of an ordered set with incorrect running disparity (not negative)

Loss of Sync State

This state uses a timeout function to transition to the Not Operational state, if synchronization is not regained. The standard three ordered set criteria is used to enter the Sync Acquired state. The flowchart in *Figure 5* defines the details of this receiver state.

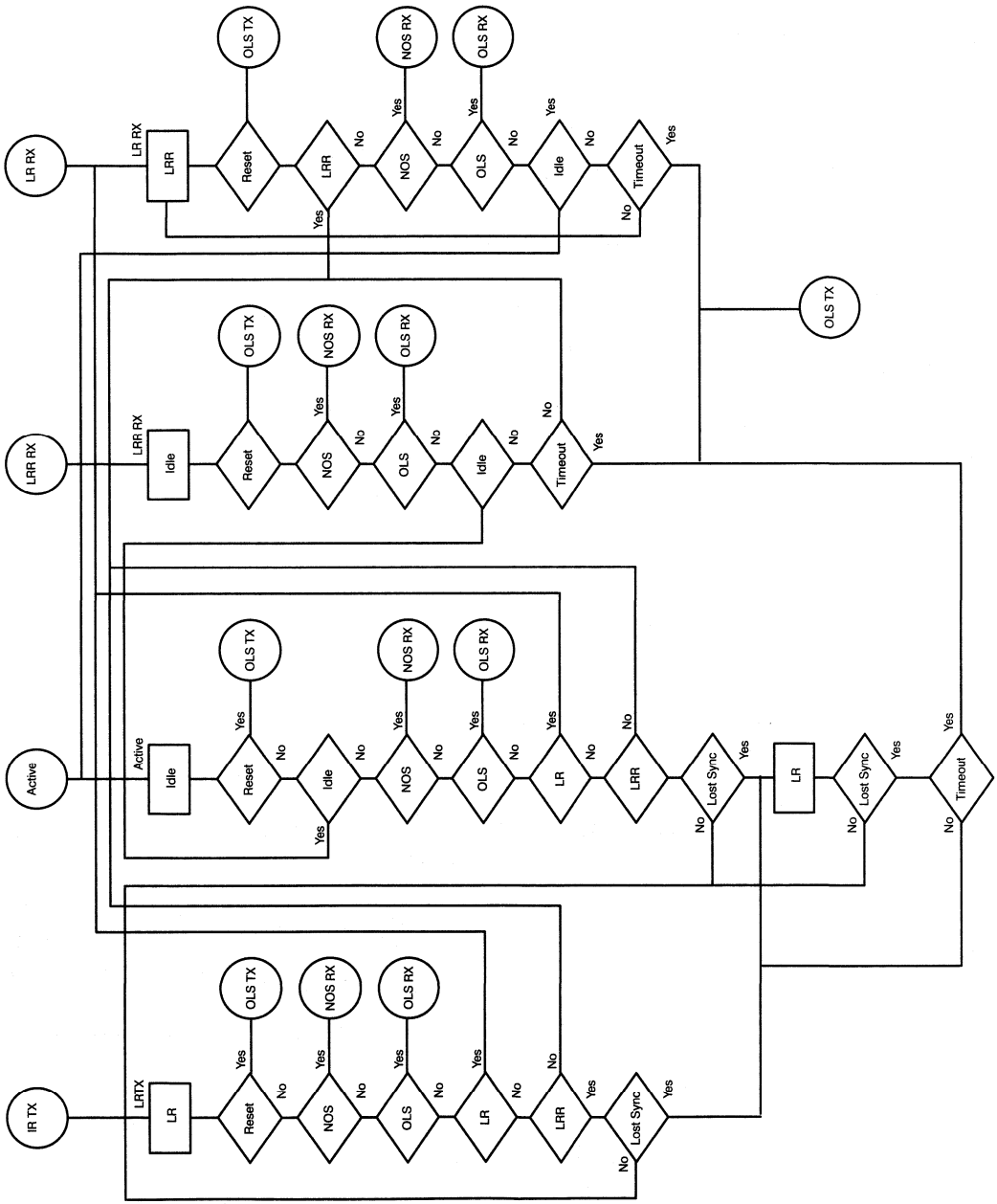


Figure 3a. Port State Machine

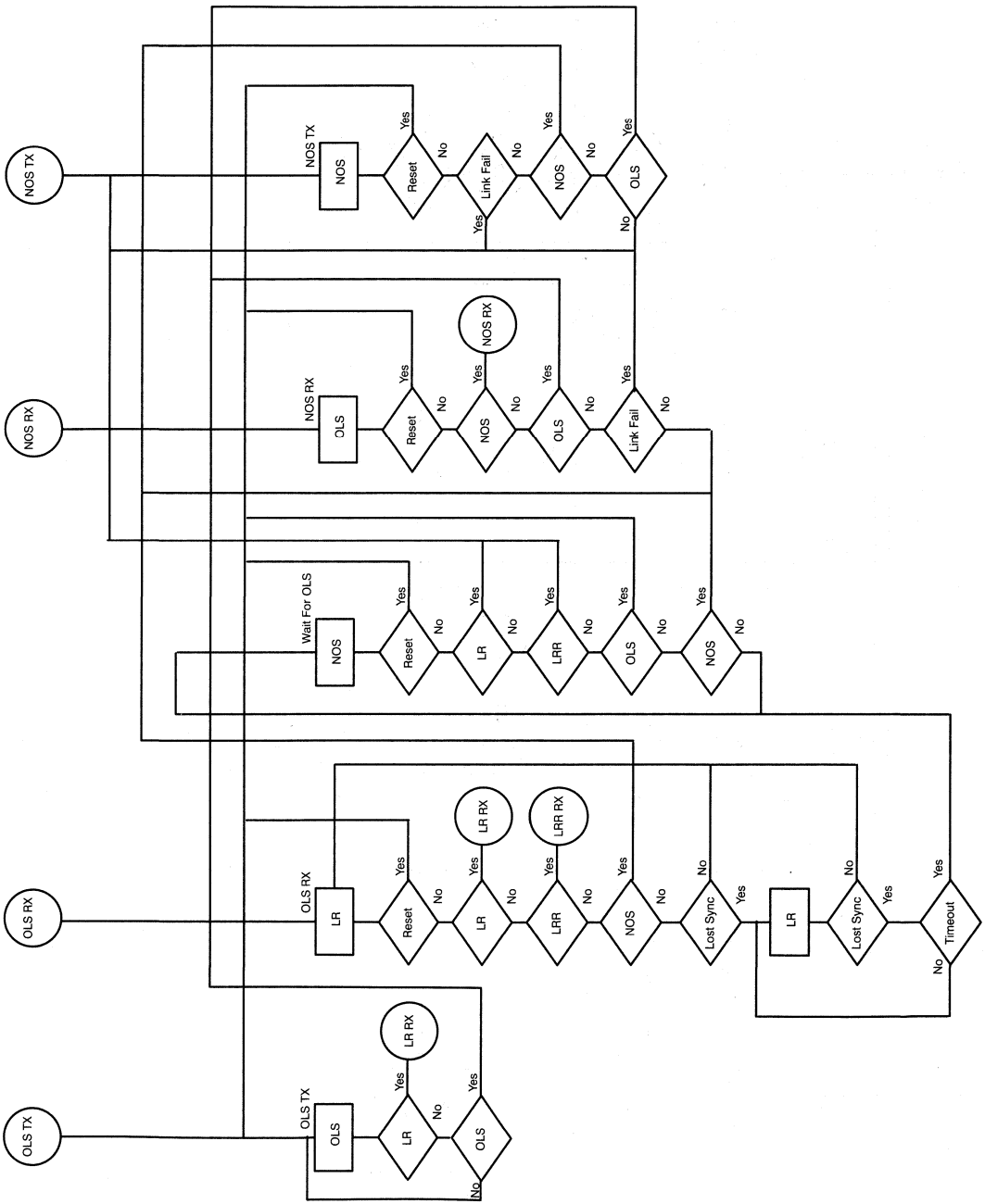


Figure 3b. Port State Machine

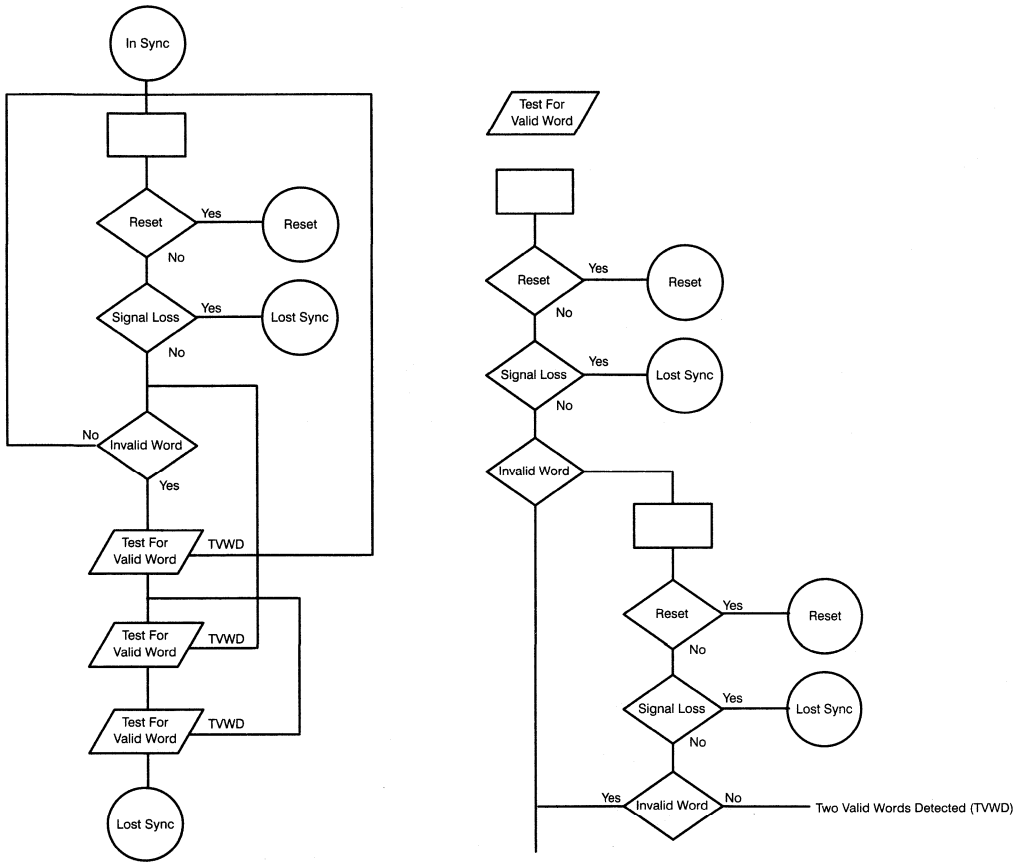


Figure 4. Receiver Flow Chart—Sync Acquired

Not Operational States

These states are illustrated in *Figure 6*, and reflect the control sequences for loss of sync with timeout, loss of signal with timeout, and the reset command.

On-Line, and Off-Line Condition

These conditions are controlled directly by Layer 2 processes, and only affect the reporting of loss of synchronization and receiver failure conditions to Layer 2. While in the Off-Line state, the receiver continues to send valid data to Layer 2, but does not report any error conditions. An Off-Line condition also survives a reset sequence.

Layer 1 Transmitter State Machine

The transmitter state machine in Layer 1 is responsible for transmitting a continuous bit stream onto the link. Transmission words are sent on a 4-byte boundary, which is established on entry to the working state.

Fibre Channel defines a set of 4 states for the Layer 1 transmitter state machine:

- Working
- Not Enabled
- Failure
- Open Fibre

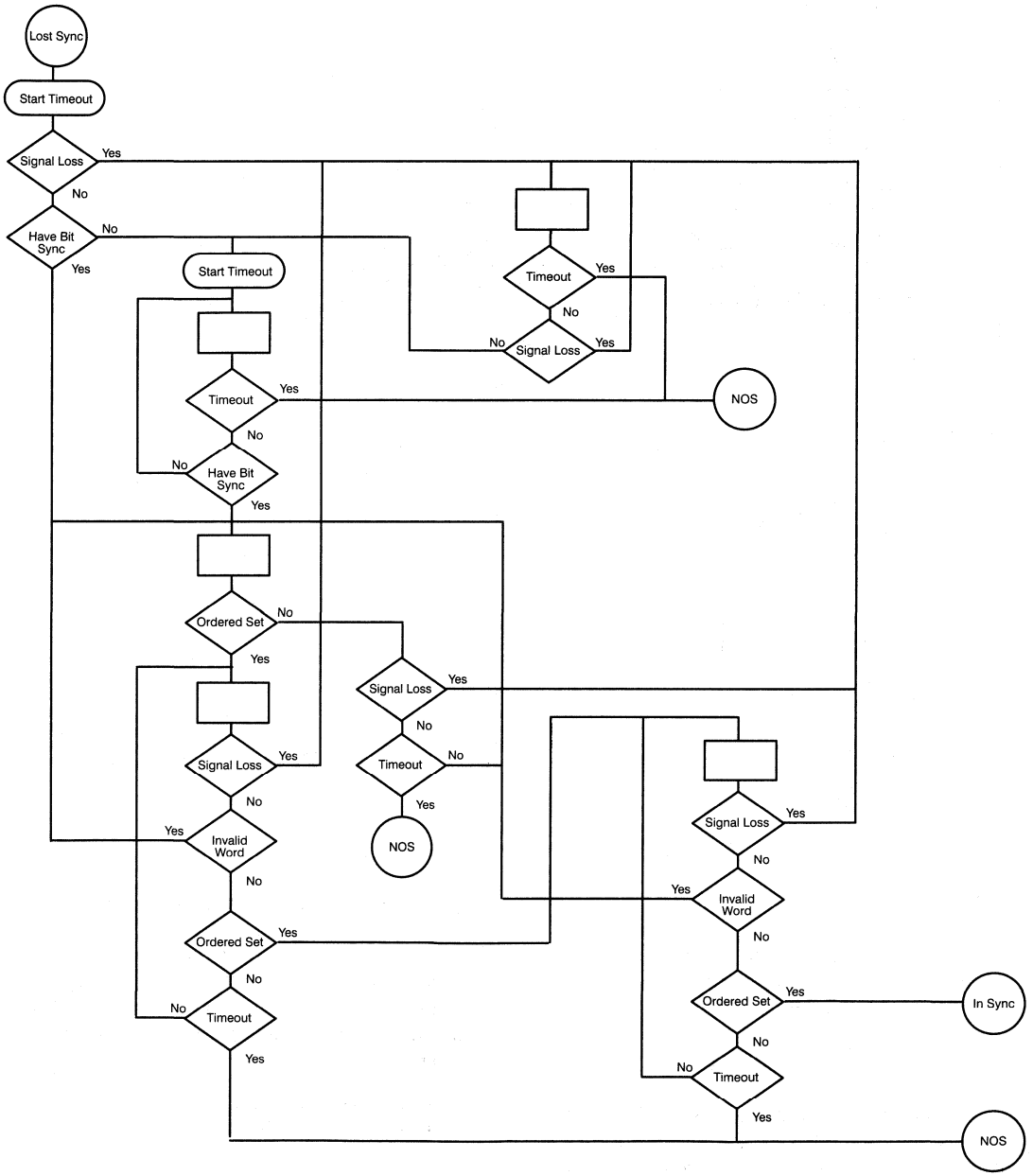


Figure 5. Receiver Flow Chart—Lost Sync

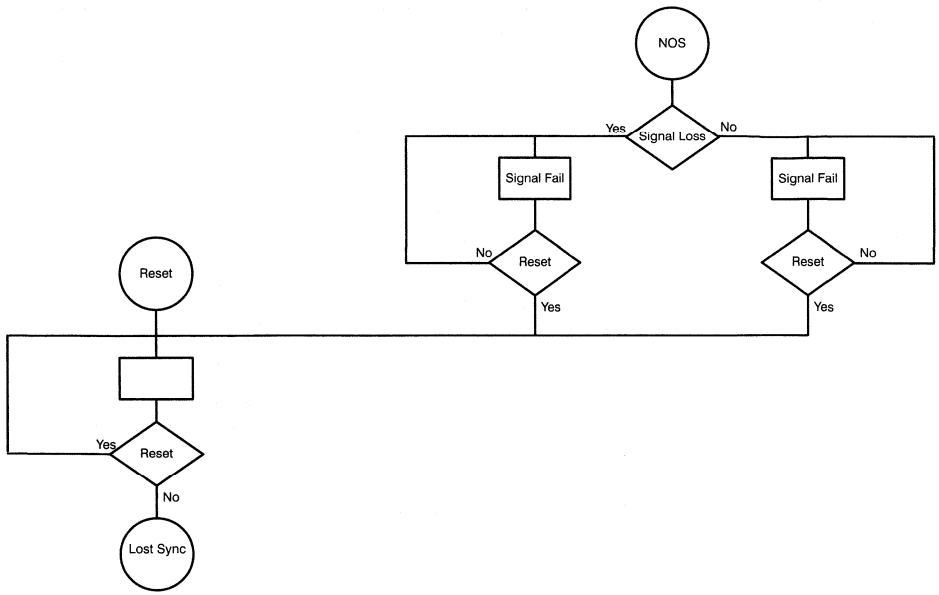


Figure 6. Receiver Flow Chart—Not Operational State (NOS)

Working State

This is the operational state, where a continuous bit stream is transmitted onto the link.

Not Enabled State

This state is entered by explicit request from Layer 2 and no data is transmitted.

Failure State

This state is entered when a request for transmission occurs, violating the following rules:

- Transmission-word boundary violation
- Lack of transmission word data

Open Fibre State

This is a safety state which is entered whenever it is determined that a laser safety condition exists. The detection of this condition is external to Layer 1 functions, and is usually a mechanical interlock that is asserted when no fiber is connected to port.

The operation of the Layer 1 transmitter state machine is defined in *Figure 7*.

Design Example

The functional elements which are typically implemented in hardware are shown in *Figures 8* and *9*, with the interface requirements as defined by Fibre Channel. This block diagram is used to derive a design example for the key hardware elements for Layer 1 and 2 of a Fibre Channel port.

Input Data Path Serial Decode

The serial decode function in Layer 2 of the Input Data Path can be implemented entirely by the CY7B933 HOTLink™ Receiver, which was designed for this specific application.

Input Data Path Receiver State Machine

This element shown in *Figure 10* implements the Layer 1 Receiver State Machine, discussed previously. The design partitions into two separate state machines. The bit sync state machine, described previously, requires 16 characters to be re-

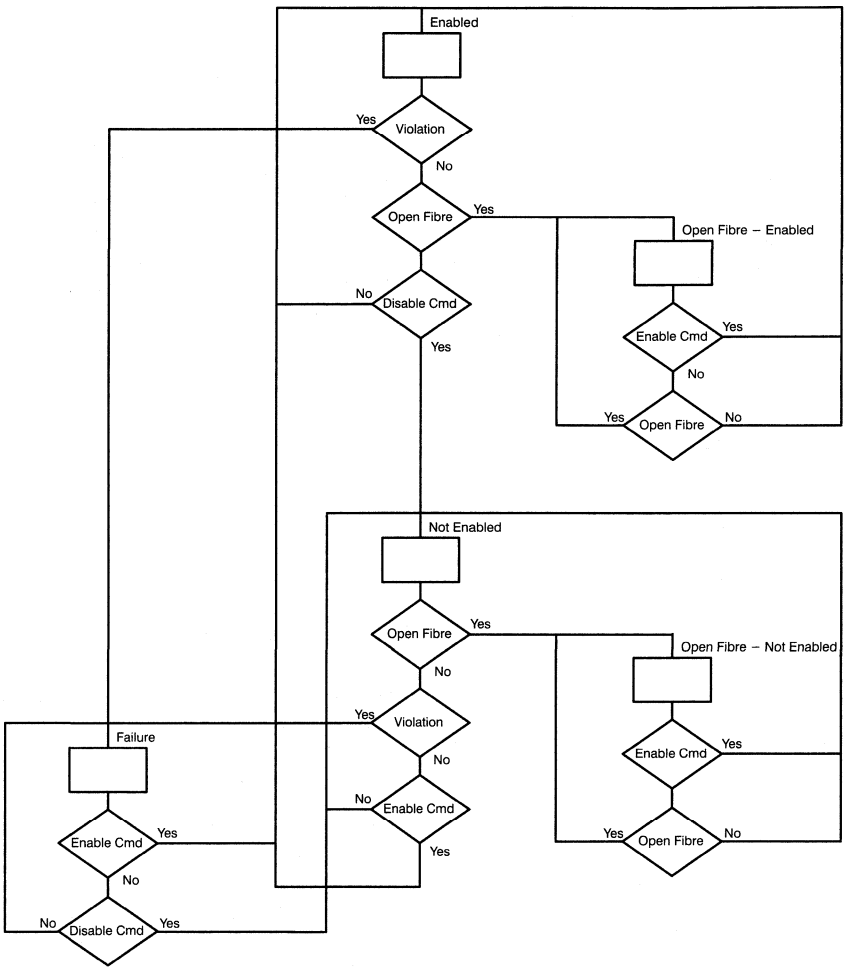


Figure 7. Transmitter Flow Chart

ceived without a code violation, as a criterion for bit-level synchronization. Once bit sync is acquired, the state machine generates a boundary strobe every fourth character until a bit Resync command is received. The second state machine implements the actual receiver state sequence defined in *Figures 4, 5,* and *6*. The timeout function used in the flowchart uses a 100-ms timebase as its reference. The entire function can be implemented in a single CY7C371 PLD.

Ordered Set Decoder

This functional block accumulates received characters and decodes the ordered sets defined by Fibre Channel. The result of the decode operation is a 4-bit code that is used by the port receiver state machine. The decoder is implemented using four state machines that individually decode a single character in a 4-byte ordered set. The current decode state from a state machine is passed to the next state ma-

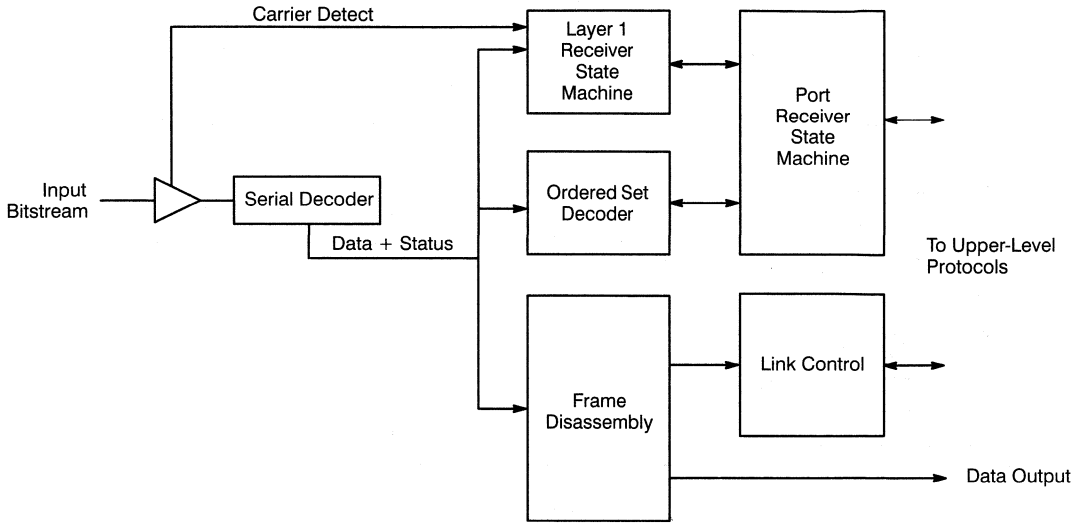


Figure 8. Receive Path Functional Elements

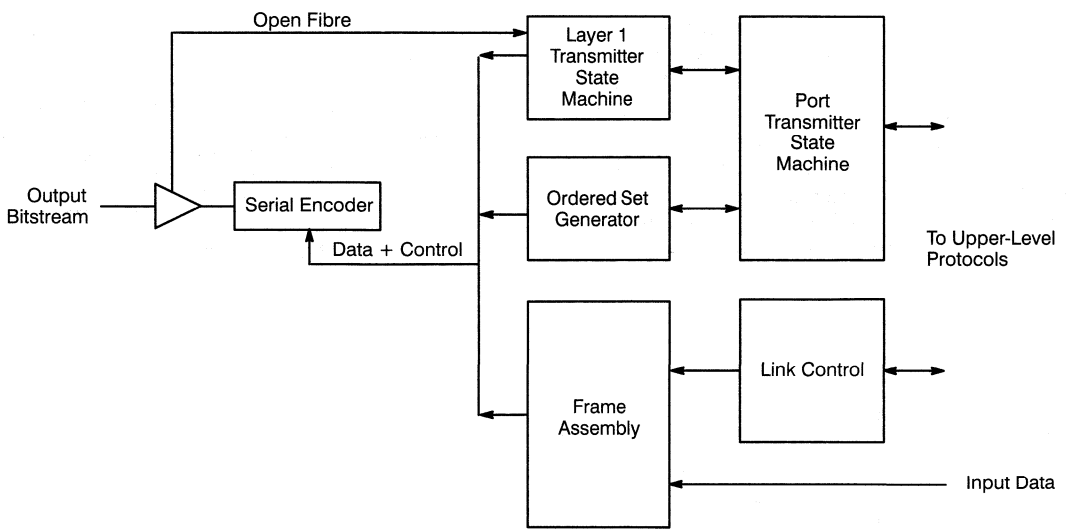


Figure 9. Transmit Path Functional Elements

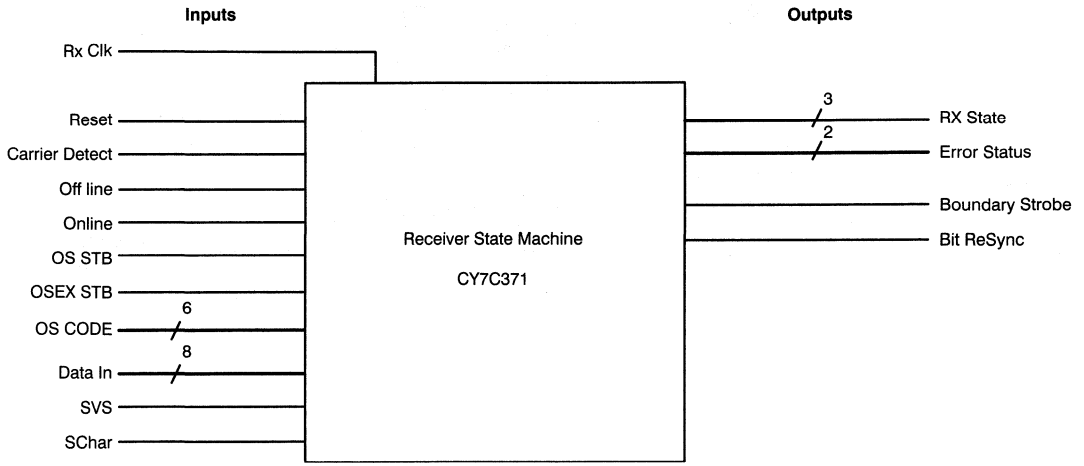


Figure 10. Layer 1 Receive State Machine

chine with the next character. The output of the last state machine is a 6-bit code that defines the decode result of the decoded transmission word. The bits are divided into 2-bit and 4-bit fields that define the decoded category and type of the ordered set. This was done to facilitate interpretation by the receiver state machine. The bit fields are defined in *Tables 4* and *5*. When a valid ordered set is detected, the resulting code is presented, and an ordered set strobe, OS STB, is generated. When an error is detected, such as an unrecognized set, an error code, defined in *Table 6*, is presented, and an ordered set exception strobe, OSEX STB, is generated.

Although four separate state machines are used for the decode function, each of the state machines are relatively simple, and the entire function can be implemented using a single CY7C371 with 21 macrocells required. A block diagram of the decoder is provided in *Figure 11*.

Table 4. Ordered Set Decode Category Bit Field

Category Bits		Ordered Set Category
0	0	Primitive Sequence
0	1	Primitive Signal
1	0	Start of Frame
1	1	End of Frame

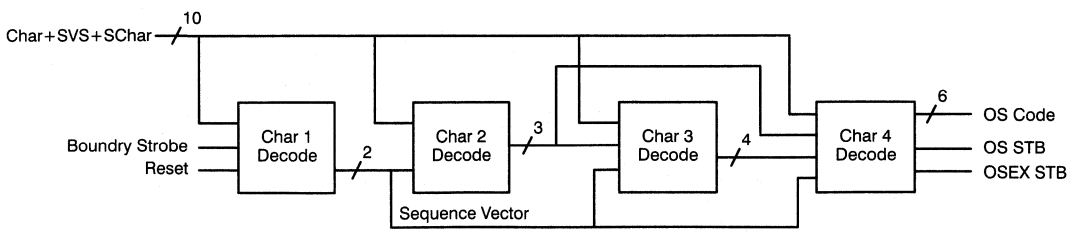


Figure 11. Ordered Set Decoder State Machine

Table 5. Ordered Set Decode Type Bit Field

Type Bits	SOF Set Type
X X 0 1	SOF Connect
X X 1 0	SOF Initiate
X X 0 0	SOF Normal
0 1 X X	SOF Class 1
1 0 X X	SOF Class 2
1 1 X X	SOF Class 3
0 0 X X	SOF Fabric
Type Bits*	**EOF Set Type**
I DA T RD	
X X 0 1	EOF Normal
X X 1 0	EOF Terminate
X X 0 0	EOF Disconnect-Terminate
0 1 X X	EOF Abort
Type Bits*	**EOF Set Type**
I DA T RD	
1 0 X X	EOF Disconnect-Terminate-Invalid
1 1 X X	EOF Normal-Invalid
Type Bits	**Primitive Signal Type**
0 0 0 1	Idle
0 0 1 0	R_RDY
Type Bits	**Primitive Sequence Type**
0 0 0 1	OLS
0 0 1 0	NOS
0 1 0 0	LR
1 0 0 0	LRR
* I = Invalid, DA = Disconnect/Abort, T = Terminate, RD = Running Disparity Sense	

Table 6. Ordered Set Error Codes

Bits	Error Code
0 0 0 1	Special Character in location 2, 3 4
0 0 1 0	Code Violation in set
0 0 1 1	Unknown set

Frame Disassembly

This portion of the input data path can be partitioned in a variety of ways, depending on the requirements of the design. The function of the block is to capture the different parts of a data frame. The three primary components of a data frame are the header, the data field, and the CRC. This block, because of the wide variability in design requirements, will not be expanded beyond a general discussion.

In general, it is likely that the header and data field will be logically separated in a frame. This is because the header is examined by software processes in Layer 2, and the data field is usually passed on to upper-level protocol layers. For this reason, distinct buffers for the header and data field would be used. In the simplest case, a header and a data FIFO could be used to buffer the two frame components separately. A dual-port RAM structure would allow multiple buffering with random access for Layer 2 decoding without requiring further buffering. The dual-port RAM approach, while offering greater performance, does have a higher level of complexity.

The CRC checker requires the generation of the FDDI polynomial, discussed previously. This is a relatively complex design block, which can accept data bytes at a rate from 25 to 100 Mbytes per second, and report at the end of a frame if the resulting polynomial contains all zeros. The high data rate, with the needed arithmetic operations, will probably require a highly pipelined design. The skew introduced by the pipelining will have to be accommodated in the architecture.

Finally, a state machine will be required for the frame disassembly block. The details of the state machine are determined by the design approach taken for the header and data buffering. In general, the state machine will have to monitor the output from the ordered set decoder, and when a SOF is encountered, the next 24 bytes must be loaded into the header buffer. Then successive bytes must be loaded into the data-field buffer until an EOF is detected. The CRC check must also be enabled during the header and data-field reception. The state machine must then report the frame reception and ending status to the appropriate Layer 2 process.

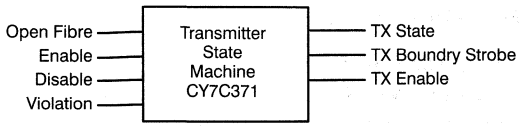


Figure 12. Layer 1 Transmitter State Machine

Output Data Path Serial Encode

The serial decode function in Layer 2 of the input data path can be implemented entirely by the CY7B923 HOTLink Transmitter, which was designed for this specific application.

Output Data Path Transmitter State Machine

This element implements the Layer 1 transmitter state machine, as discussed previously. The design is a simple 4-state machine. The block diagram is illustrated in *Figure 12*. The design is readily implemented in the same CY7C371 PLD used for the input data path state machine.

Ordered Set Encoder

This functional block generates, on command, the ordered sets defined by Fibre Channel. The function is simply a selectable 4-byte sequence generator. A 5-bit code is used to select the desired ordered set to be transmitted. This function is readily implemented in a single CY7C258, which is a PROM-based state machine.

Frame Assembly

In a manner similar to frame disassembly, this portion of the output data path can be partitioned in a variety of ways, depending on the requirements of the design. The function of the block is to generate the different parts of a data frame. The three primary components of a data frame are the header, the data field, and the CRC. This block, because of the wide variability in design requirements, will not be expanded beyond a general discussion.

In general, it is likely that the header and data field, in a frame, will be logically separated. This is because the header is generated by software processes in Layer 2 and the data field is usually obtained from upper-level protocol layers. For this reason, distinct buffers for the header and data field would be used. In the simplest case, a header and a data FIFO could be used to hold the two frame components separately for transmission. A dual-port RAM structure would allow multiple buffering with random access for Layer 2 generation without requiring further buffering. The dual-port RAM approach, while offering greater performance, does have a higher level of complexity.

The CRC generator requires the generation of the FDDI polynomial that was discussed previously. This is a relatively complex design block, which can accept data bytes at a rate from 25 to 100 Mbytes per second, and append the resulting polynomial at the end of a frame. The high data rate, with the arithmetic operations needed, will probably require a highly pipelined design. The skew introduced by the pipelining will have to be accommodated in the architecture.

Finally, a state machine will be required to control the frame assembly operation. As in the Frame Disassembly block, the details of the state machine are determined by the design approach taken for the header and data buffering. In general, the state machine will initiate the transmission of a frame, on command from a Layer 2 process. The state machine must request the proper SOH to be generated by the ordered set encoder, load the 24 bytes of the header buffer contents into the byte stream, pass the contents of the data buffer into the byte stream, steer the CRC bytes out to the byte stream, and finally, request the ordered set encoder to generate the appropriate EOF. The state machine must then report the frame completion to the appropriate Layer 2 process.

Port State Machine

The Port State Machine implements the primitive link protocol (discussed previously) and is a simple 10-state machine with a timeout counter. The block diagram for the element is illustrated in *Figure 13*, and is implemented in a CY7C371 PLD.

Design Example Summary

The key hardware components of Layers 1 and 2 of the Fibre Channel standard have been defined, and

it has been illustrated that these functional elements can be implemented in a small number of PLDs. VHDL source code for these components may be obtained by contacting your local Cypress sales office.

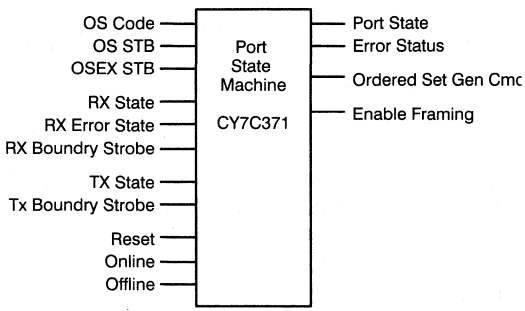


Figure 13. Layer 2 Port State Machine

HOTLink is a trademark of Cypress Semiconductor Corporation.

Summary

This application note has discussed in detail the requirements for hardware interface between Layer 1 and Layer 2 of the Fibre Channel standard. The key functional elements have been examined in detail, and a specific design examples for the Layer 1 transmitter and receiver state machines, ordered set encode and decode functions, and the Layer 2 port state machine have been provided.

References

1. Fibre Channel FC-PH Specification
2. Cypress Component Databook

Document #: COM0010–0294

Using High-Speed Serial Links to Supplement Parallel Data Buses

Today's designers face a multitude of problems when trying to move data within their systems. These problems range from overtaxed parallel-bus bandwidth to a lack of pins at the card edge connector. Even routing parallel buses around today's dense circuit boards is very difficult. This application note discusses using high-performance serial links as a solution to some of these bottlenecks. A serial approach provides three immediate benefits: first, bandwidth may be offloaded from the backplane bus; second, connector pins are saved; and, third, circuit board routing is made much easier since only two traces have to be routed for the data path (versus one for each data bus bit).

The ideal serial interface building block would be a chip set consisting of high-speed parallel-to-serial and serial-to-parallel converters (also referred to as transmitter and receiver). Additionally, this chip set would make the serial interface transparent to the user, i.e., parallel data would flow in one side and out the other. It would be able to use a variety of serial media directly such as coaxial cable, twisted-pair cable or even fiberoptic cable (when connected to the proper optical driver). It would also be easily adaptable to user-defined protocols for applications involving Direct Memory Access (DMA).

HOTLink™

Cypress' serial interface building blocks are the CY7B923 HOTLink Transmitter and CY7B933 HOTLink Receiver. These devices provide data rates of 160 to 330 Megabits/sec (16 to 33 Megabytes/sec) and conform to several communications standards.

This application note focuses on utilizing HOTLink to move data using a simple protocol. A block diagram of a typical HOTLink interface is shown in *Figure 1.*

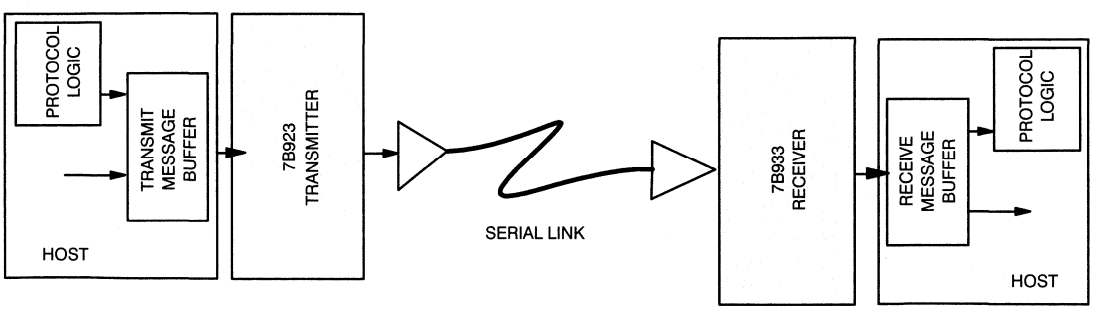


Figure 1. HOTLink System Connections

Preliminaries

For this application we will assume that the serial links in question will not exceed 3 or 4 feet. This length is adequate for most intra-board and board-to-board communications situations; and limiting ourselves to these distances removes several communications system issues from those that must be considered. Let's now discuss the general features of the HOTLinks.

In Encoded Mode, the HOTLink has an 8-bit parallel interface. Data bytes are encoded into 10-bit transmission words using 8B/10B encoding. In Bypass mode, HOTLink uses a 10-bit interface. 10-bit words bypass the encoder and go directly to the serializer. The 8B/10B code provides enough signal transitions on the serial interface to ensure proper PLL operation. It is also DC balanced, which prevents development of DC offset in the link over time. DC offset can result from more 1s being transmitted than 0s, so the encoder maps the 8-bit input word to multiple 10-bit output values to keep the number of 1's and 0's constant over time. Ideally, the time-averaged DC component should be zero, since DC offset over a long cable can cause increased noise susceptibility and power dissipation. In fiber systems excessive DC offset can burn out the LEDs used to drive the fiber.

The applications in this note use 8-bit Encoded Mode. In this mode HOTLink provides two control pins. A pin called SC/$\overline{D}$ indicates whether the byte on the parallel I/O pins is a special character or data. Another pin available in 8-bit mode, SVS (Send Violation Symbol), allows the data provider to force a violation symbol to be encoded and sent. The SC/$\overline{D}$ pins will be used to signify command words in the DMA protocol, which will be specified later. The SVS (RVS in the receiver) pin could be used for system testing and error checking, but will not be part of the design.

Parallel Interface

For details of the HOTLink parallel interface, please refer to the FIFO-HOTLink application notes located in this book.

Transmitter

The signals needed for the transmitter parallel interface are the 8-bit parallel inputs D(0..7), the SC/$\overline{D}$ bit, the $\overline{ENA}$ pin, the $\overline{RP}$ pin, and the CKW pin. Refer to the CY7B923/CY7B933 HOTLink datasheet for additional details.

When no data is enabled into the transmitter, it should be noted that the HOTLink Transmitter inserts a special character called SYNC. This SYNC character provides sufficient transitions to keep the PLLs locked to the bit stream.

Receiver

The signals needed for the receiver parallel interface are the 8 parallel data outputs D(0..7) and the SC/$\overline{D}$, $\overline{RDY}$, and CKR pins. Again, refer to the device datasheet for additional details.

When the transmitter is sending SYNC characters, the receiver detects these and does not output this character until the last SYNC character is received. Then the receiver outputs a single SYNC character.

Serial Interfaces

Figure 2 shows the multiple serial outputs of the transmitter, and the dual serial inputs of the receiver. OUTC is always on and in full duplex implementations, can be "looped back" to a receiver input for system diagnostics or used as another output. The other pair of outputs, OUTA and OUTB are enabled with the FOTO pin. This output pair makes it easy to transmit from one source to multiple destinations, making point-to-multi-point DMA architectures possible. The receiver, with its pair of inputs can use one input channel for data and the second to implement local loopback. The input selection is accomplished with the A/$\overline{B}$ pin. Note that the INB channel does not have to be used for diagnostics, but can be used as another data stream input. However, switching input channels requires the PLL to reacquire lock with the incoming data stream.

Implementing a Data Link

The discussion that follows deals with issues confronting a designer trying to move data from point

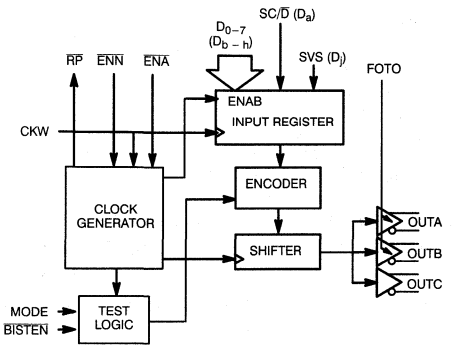


Figure 2a. CY7B923 Transmitter Logic Block Diagram

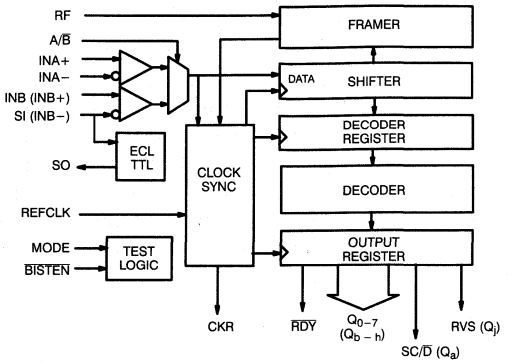


Figure 2b. CY7B933 Receiver Logic Block Diagram

A to point B using HOTLinks. *Table 1* shows the three implementations discussed.

I/O Space Model

The first example is simple. It assumes that the receive FIFO resides in the destination's I/O space. The receive controller (a microprocessor, perhaps) merely reads and interprets the data stream out of the Rx FIFO. Data does not get placed in local memory before being used. There are two issues to consider with this example: What if the receive logic cannot keep up with the received data? This is known as receiver overflow or transmitter overrun. A FIFO with a programmable almost full/empty flag can be used together with a PLD to generate a receive or transmit inhibit to the transmit controller. This is known as "flow control". *Figure 3* shows a receiver block diagram with a flow control signal labeled TXINH. If the FIFO becomes too full, this signal tells the transmitter to stop transmitting until the receiver catches up. Since we are limiting our links to 3 or 4 feet, this may be a viable approach. However, using this form of flow control wastes a lot of bandwidth. Correct sizing of the FIFO, after careful analysis of the communications requirements, can give a deterministic system that never overflows or underflows. Communications links of hundreds of feet, or even miles cannot afford this type of flow control, since the channel itself may be several hundreds or even thousands of bytes long and a large enough FIFO may not be available. The channel is like a pipeline, and once something enters the pipeline, it must come out the other end.

Table 1. Data Link Implementations

I/O Model	Transmitter	Receiver	Features
I/O Space	FIFO+HOTLink	HOTLink+FIFO+ Microprocessor	Rx FIFO in I/O Space Microprocessor Accesses Data
Direct Memory Access	FIFO+HOTLink	HOTLink+FIFO+ DMA Logic	Rx Controller Decodes DMA Info in Rx FIFO DMA Moves Data Microprocessor Free Local Bus Used
Shared Memory Space	FIFO+HOTLink	HOTLink+DUALPORT+ DMA Logic	Rx Controller Uses Semaphores to Move Data Directly to Shared Memory Microprocessor Free/Local Bus Free

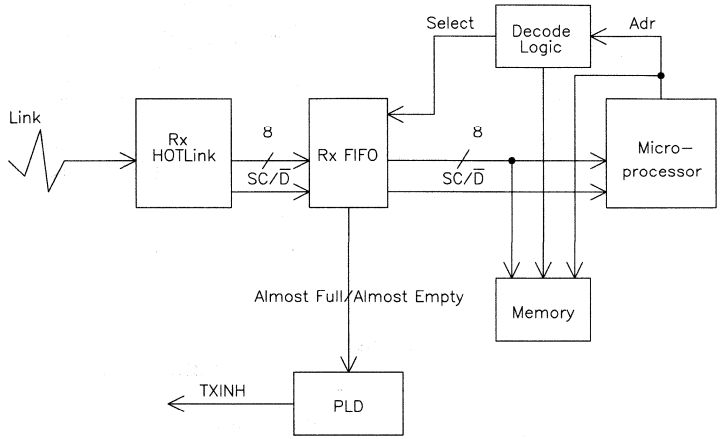


Figure 3. Receiver Flow Control

The second issue is that the microprocessor must be interrupted when data needs to be read from FIFO. This wastes microprocessor cycles and required lots of latency.

Direct Memory Access Model

So far we have discussed moving data from Point A to Point B using a microprocessor. Direct Memory Access (DMA) uses additional hardware, called a DMA controller or DMA Logic, to move the data from the FIFO to the memory. This frees the microprocessor of this task. Before proceeding, let's look at the bandwidth supported by HOTLink. This will determine the speed at which our DMA logic must operate. Refer to *Table 2*.

Table 2. Bandwidth Requirements

Part Number	Bandwidth (Mbyte/s)	Clock Period (ns)
CY7B923/933	16 – 33	63 – 30

Table 2 shows that DMA is probably a better solution than the I/O space model. The DMA Logic contains several basic functions. These are:

- control state machine
- address counter
- address (and sometimes data) latches and drivers

The control state machine detects when the receive FIFO contains data and issues a DMA request to the microprocessor. The DMA request asks the microprocessor to relinquish the memory address and data buses. The state machine also detects when the microprocessor has freed the buses. It then starts the actual transfer by loading the address counter with the initial address and placing the initial address and data word on the memory address and data buses. The control state machine then strobes the data into the memory, increments the address counter, reads the next data from the receive FIFO onto the memory data bus and strobes this data into the memory. The process then repeats until all the data has been placed in memory. When all the data has been placed in memory, the memory address and data buses are returned to the microprocessor's control. A block diagram is shown in *Figure 4*. Questions at this point might include, "Where did the starting address come from?", or "Where did the ending address come from?" To answer these questions, a protocol needs to be defined.

DMA Protocol Definition

Let's now discuss some concepts being introduced with our DMA protocol definition. First, we are now embedding command and control information in the data stream. Previously the information consisted of pure data (as far as our control logic was

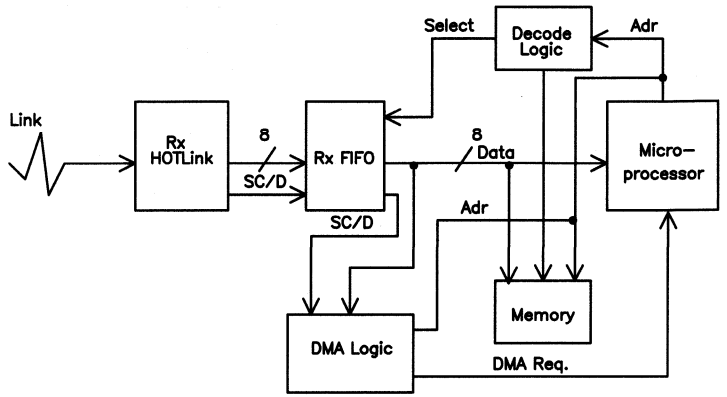


Figure 4. DMA Configuration

concerned). Second, we are now using dedicated hardware (a PLD) to move the data from the RX buffer to the location where it will be used, thus off–loading the main processor. Since the same protocol definition will be used for the shared memory implementation, let's define a protocol. First, the design will assume a fixed length DMA of 256 words. (The user is free to implement any length required, or to provide for variable length transfers.)

Table 3 defines a DMA Write message. The protocol will consist of a special character or message delimiter signifying a DMA write request, followed by characters defined as a broadcast address, and a starting address. A broadcast address can be thought of as a card or processor ID. The starting address indicates the first address to be written. This is followed by N–bytes of data, where N is equal to 256 in the example.

DMA reads will be identified by a unique message delimiter as shown in *Table 4*. Again, it is assumed that 256 bytes of data will be sent. The message defined in *Table 4* tells the recipient to send the 256 bytes of data beginning at the address indicated, and it also provides a destination address that can be used to create the DMA write message.

Finally, to assure proper initialization of the DMA hardware, *Table 5* defines a DMA Reset message.

The column labeled "Bits" in *Tables 3*, *4*, and *5* deserves further explanation. The labels HGF EDCBA are the 8B/10B designations for bits on the HOTLink parallel interface. Conventional notation for these bits is Q7..Q0 on the receiver outputs and D7..D0 on the transmitter inputs, with bit 7 being the most significant bit. In fact these signals correspond to the pins labeled identically on the HOTLink devices. The message delimiter characters are named according to Fibre Channel convention. Refer to the CY7B92X/CY7B93X datasheet for additional information.

The receiver DMA Logic in *Figure 4* needs to contain a state machine to detect the appropriate message delimiters and decode the broadcast address. If the broadcast address is for the module and the message is a DMA write, another state machine needs to issue a DMA request to the microprocessor and obtain the bus. After obtaining the bus, the starting address is read from the FIFO and loaded into an address counter. Since the address is defined as 32–bits, and the message length is defined as 256 bytes, the address must be loaded into 3 latches and an 8-bit counter. Then the state machine to reads the next 256 bytes out of the receive FIFO and writes it to memory. Then the bus is relinquished to the microprocessor and the counters and state machine are reset.

When a the message is a DMA read, the state machine is similar to that for a DMA write, but the ad-

dress counter is loaded with the address to read from, and the destination address is read out to be placed into a DMA write message. Creation of DMA write messages can be accomplished with additional resources in the DMA Logic. Suggested devices are the Cypress CY7C375 CPLD or the Cypress CY7C385 FPGA.

Table 3. DMA Write Message

SC/$\overline{D}$ Pin	Byte Name	Bits (HGF EDCBA)	Definition
1	K28.1	000 00001	Msg. Delimiter
0	8–bit address	–	Broadcast Address
0	Address byte 0	–	Most significant
0	Address byte 1	–	Next most signif.
0	Address byte 2	–	Next least signif.
0	Address byte 3	–	Least significant
0	Data byte 0	–	1st data byte
0	Data byte 1	–	2nd data byte
0	:	:	:
0	Data byte N	–	Last data byte
1	K28.1	000 00001	Msg. Delimiter

Table 4. DMA Read Message

SC/$\overline{D}$ Pin	Byte Name	Bits (HGF EDCBA)	Definition
1	K28.0	000 00000	Msg. Delimiter
0	8–bit address	–	Broadcast Address
0	Source Address byte 0	–	Most significant
0	Source Address byte 1	–	Next most signif.
0	Source Address byte 2	–	Next least signif.
0	Source Address byte 3	–	Least significant
0	Dest. Address byte 0	–	Most significant
0	Dest. Address byte 1	–	Next most signif.
0	Dest. Address byte 2	–	Next least significant
0	Dest. Address byte 3	–	Least significant
1	K28.0	000 00000	Msg. Delimiter

Table 5. DMA Reset Message

SC/$\overline{D}$ Pin	Code Name	Bits (HGF EDCBA)	Definition
1	K28.3	000 00011	Msg. Delimiter
0	8–bit address	–	Broadcast Address
1	K28.3	000 00011	Msg. Delimiter

The DMA Model offloads the data moving task from the microprocessor. However, DMA has one major disadvantage; it requires the bus, which means the microprocessor must be idle during the DMA. There is another approach and it is the next topic.

Shared Memory I/O Model

If a shared memory area (dual-ported) is implemented, then data can be made available to the local logic without grabbing the microprocessor bus to perform a DMA. Simultaneous accesses must be prevented, but dual-ported memory opens up several options. These options include dividing the dual-ported memory into segments and alternating the segments between DMA write and local side read. This is known as "ping-ponging" and prevents simultaneous access of a dual-port SRAM location. So, dual-ported memory is attractive for those cases where the local bus cannot be tied up with a true DMA. *Figure 5* shows a diagram of a HOTLink communications link implemented with dual-ported SRAM. The DMA Logic is virtually identical to that of the prior section.

Summary

This application note has presented the basic concepts for employing HOTLink high-speed serial communications devices to replace parallel data paths. It has also defined a simple protocol and described the logic necessary to implement the protocol. Finally, the advantages and disadvantages of three different approaches have been presented to allow the designer to choose the one that best fits their needs. The simplest is Memory Mapped I/O, the highest performing is the Shared Memory I/O Model employing dual-ported memory.

Document #: COM0007–0394

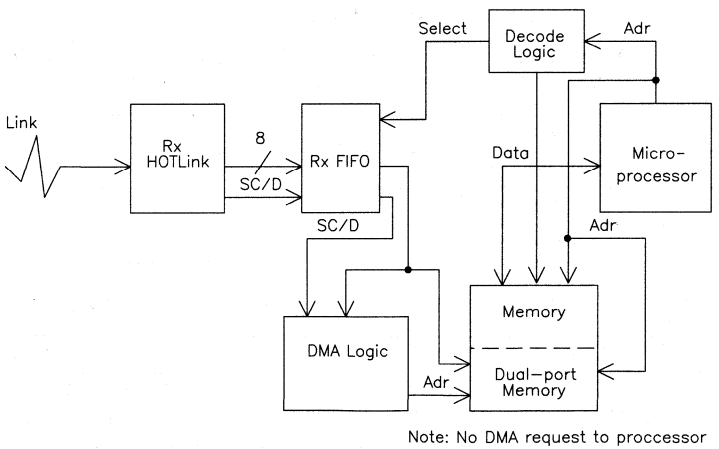


Figure 5. Dual-Ported Configuration

HOTLink is a trademark of Cypress Semiconductor Corporation.

Using the CY7B923 as an ECL Clock Source

Abstract

This application note details the use of an inexpensive data communications transmitter device as a high-precision, flexible, and programmable Emitter-Coupled-Logic (ECL) or Positive-Emitter-Coupled Logic (PECL) clock source. Issues concerning clock characteristics, stability, distribution and design techniques are discussed in detail. Information is provided to allow the user to configure the device for a variety of applications.

The Ideal Clock Circuit

The ideal clock source would provide the designer with several attributes that would benefit the eventual design. It would be flexible in that it would provide for a broad range of frequency coverage. Its frequency would be stable from one cycle to the next, its pulsewidth would be stable over time and both of these parameters would be consistent over temperature and voltage variations. The clock output transition time from one level to another (the rise and fall time) would be short in order to minimize the skew caused by sampling threshold effects at the receive end of the clock. It would be capable of sourcing significant amounts of current into multiple single-ended or differential PECL/ECL loads with a minimum amount of output skew. It would provide for relatively low power consumption when compared to PECL/ECL clock sources currently available. And lastly, it would be a low-cost device, available in a variety of packages for compatibility with commercial, industrial, military, and surface-mount applications. The device makes use of an inexpensive TTL clock oscillator instead of the expensive PECL/ECL devices typically used. The Cypress CY7B923 HOTLink™ Transmitter, although not specifically designed as an ECL clock source, provides the features to address these needs in a highly effective manner.

HOTLink Transmitter Features and Specifications

The HOTLink chip set is comprised of a pair of high-speed point-to-point communications building blocks that operate over high-speed serial data links (fiberoptic, coaxial cable, and twisted/parallel pair) at 160 to 330 Mbits/second. The HOTLink pair consists of the CY7B923 Transmitter and the CY7B933 Receiver. The Transmitter features a set of three positive 100K (referenced to +5) ECL differential output buffers, a data input register, an encoder to encode 8-bit data into a 10-bit word, a Built-In Self-Test pattern generator, a serializer to convert parallel data to serial data, and a clock generator to produce a bit-rate clock from the incoming word-rate clock input. These features of the CY7B923, with the exception of the encoder and Built-In Self-Test circuits, make it ideal for use as a clock generator device.

CY7B923 Block Diagram Description

The HOTLink Transmitter is designed to transform information from a word-rate or byte-rate parallel format into a high-speed serial format. A block diagram of the CY7B923 HOTLink Transmitter is shown in *Figure 1* and a description of each module follows.

Clock Generator

The clock generator contains a phase-locked-loop (PLL) that multiplies a word-rate reference clock (CKW) by a factor of ten (10) to produce the serial

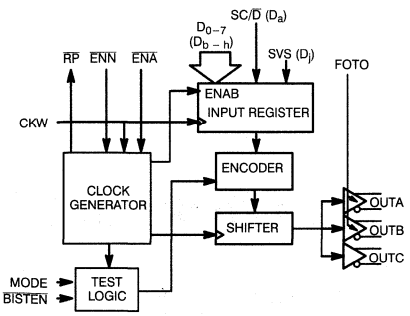


Figure 1. CY7B923 HOTLink Transmitter Logic Block Diagram

bit-rate. Data is clocked into the input register on the rising edge of CKW. The duty cycle of CKW does not affect the outgoing serial-bit stream since the PLL is capable of maintaining proper phase and duty cycle on its own. The clock generator also produces a signal called $\overline{RP}$ (Read Pulse) and is used to read new data from a FIFO in a data communications application. $\overline{RP}$ is not used when the HOTLink Transmitter is used as a clock source.

Input Register

The input register captures the data present at the Da through Dj inputs at the rising edge of the CKW clock. This parallel data is then loaded directly into the shifter for serialization if the encoder is disabled.

Encoder

The encoder is used to encoded the incoming data from the input register into an 8B/10B format for ANSI X3T9.3 (Fibre-Channel) or IBM-ESCON™ applications only. In unencoded mode, the data passes directly from the input register into the shifter. This application of the HOTLink Transmitter as an ECL clock source uses the CY7B923 in unencoded mode.

Shifter

The shifter accepts the the 10-bit word which was loaded into the input register. With the encoder disabled, the data is converted from parallel to serial with the data present on input pin Da (pin 19) shifted out first.

Output

The device supports three PECL (100K referenced to +5V) outputs. These outputs provide differential (true and compliment) capability, offer an enable pin (the FOTO pin for output pairs A & B) and the ability to drive 50Ω transmission lines directly.

Test Logic

The test logic is not used in the ECL clock source application. It contains the logic to generate the built-in self-test (BIST) pattern that is used to test the integrity of a data communications interface and link.

Fulfilling the Requirements

Frequency Range

Since the HOTLink Transmitter was designed to communicate or send data at a rate of 160 Mbps to 330 Mbps, it is ideally suited for the application of generating precise transitions or clock edges over a broad range of frequencies. As the transmitter operates, data in the form of 10-bit words are loaded into the serializer of the CY7B923 at the word-rate clock intervals. The on-board PLL takes the incoming word-rate clock and multiplies it by a factor of ten to generate the rate at which the individual bit transitions will be shifted out by the serializer. The encoder function is disabled when the transmitter is used as a clock source to provide maximum control of the data patterns being shifted out. The two primary factors that affect clock output frequency are word-rate clock frequency and the number of bit transitions within the 10-bit word. See *Equation 1* below:

$$\textit{clock out} = \textit{(word-rate clock)}\ \textit{(\# of transitions)} / 2 \quad \text{Eq. 1}$$

Where:

clock out = clock frequency present at the outputs of the transmitter

word-rate clock = rate at which the 10-bit words are loaded into the serializer

Table 1.

Data Pattern	Word Rate	Duty Cycle	Bit Transitions	Clock Frequency
0000011111	16 MHz	50%	2	16 MHz (Min. Rate)
0000001111	25 MHz	40%	2	25 MHz
0011100111	16 MHz	60%	4	32 MHz
0000011111	33 MHz	50%	2	33 MHz
0000100001	25 MHz	20%	4	50 MHz
0001100011	33 MHz	40%	4	66 MHz
0101010101	16 MHz	50%	10	80 MHz
0101010101	25 MHz	50%	10	125 MHz
0101010101	33 MHz	50%	10	165 MHz (Max. Rate)

NOTE: The minimum duty cycle is 10% and the maximum duty cycle is 90%. The minimum clock out is 16 MHz and the maximum clock out is 165 MHz.

# of transitions = the number of transitions between one logic level and another

Assume a 20-MHz word-rate clock and a data pattern of 0000011111:

Now, assume a data pattern of 0101010101:

clock out = (20 MHz) (10 transitions) / 2 => 100 MHz

The duty cycle, or relationship of clock HIGH time to clock LOW time, can also be affected by the data pattern loaded into the serializer. The duty cycle is controlled by the ratio of consecutive ones to consecutive zeros. If there are six consecutive ones and four consecutive zeros in the data pattern, the duty cycle would be 60%. If the pattern was 0001100011 the duty cycle would be three LOW and two HIGH or 40% but the number of bit transitions would double and so would the clock out frequency.

Table 1 shows examples of clock out frequencies and duty cycles that can be obtained using the data patterns and source frequencies given.

Test Circuit

A typical test circuit is shown in *Figure 2,* detailing the HOTLink Transmitter in a PECL clock generator application. The circuit uses the CY7B923 with a 10-position DIP switch to select the desired data pattern (e.g., *Table 1* patterns). The $\overline{\text{BISTEN}}$, and MODE pins are pulled to a logic HIGH while $\overline{\text{ENA}}$ or $\overline{\text{ENN}}$ are tied to a logic LOW. This configures the device to operate with Built-In Self-Test disabled, 8B/10B Encoding disabled, and data present at the Da to Dj to be loaded into the input register on each rising edge of the CKW input. The FOTO input is used as a clock output enable for output pairs OUTA and OUTB. When FOTO is LOW, transmit data will continuously be driven on output pairs A and B. When FOTO is HIGH, the output pairs A and B will remain at a logic zero state. Output pair OUTC is always enabled and will reflect the current state of the transmitter shifter output. The $\overline{\text{RP}}$ or Read Pulse output is typically used to indicate new data can be read from a FIFO or other storage device into the transmitter. It is not used in the clock generator application. In the test circuit shown, the word-rate clock could be any stable TTL clock source operating between 16 MHz and 33 MHz. As described above, the resultant output clock frequency is dependent on word clock frequency (CKW) and the number of 0-to-1 or 1-to-0 bit transitions present in the 10-bit word loaded into the input register of the transmitter.

Clock Issues

Since the CY7B923 was originally intended for very high-speed communications, the inherent stability of the communications device must be extremely

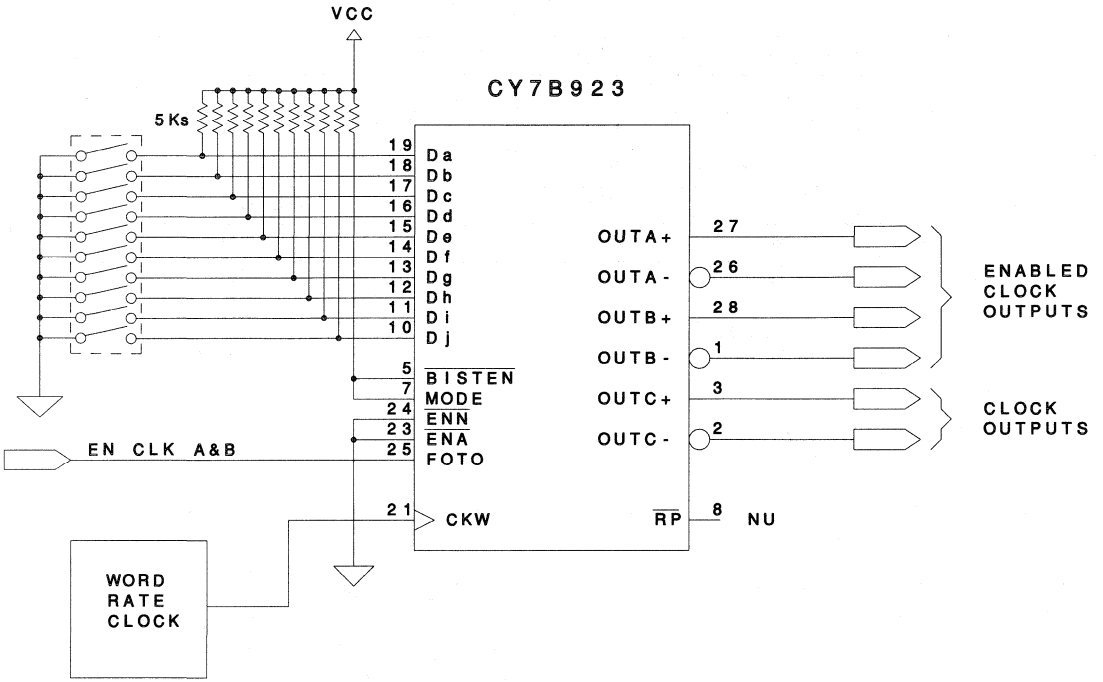


Figure 2. CY7B923 Clock Generator Test Circuit

good to prevent data communication errors and meet the rigid requirements of the standards imposed by industry. These same principles relating to clock stability also apply when the device is used as a PECL/ECL clock source. In general, the most critical factors relating to clock performance are jitter, duty cycle stability, rise and fall times, and output skew.

Jitter

Jitter is typically defined as the variation of one clock edge with respect to another. One source of jitter can be caused by noise-induced variations in the PLL, often known as random jitter. An additional form of jitter can result from the data patterns fed to the transmitter. This data dependent jitter is not relevant in the case of the clock generator because the pattern is constant and repeating. Jitter can also have an effect on the duty cycle of a clock waveform, generally referred to as duty cycle distortion. Refer to the CY7B923 datasheet for more specifications on jitter.

Duty Cycle Stability

For the clock generator application, the duty cycle, or relationship of a logic HIGH time period to a logic LOW time period, is dependent on three factors: random jitter, transmit data pattern, and rise and fall times. Random jitter has an affect on duty cycle based on the fact that it will vary the placement of one clock edge with respect to another. Another factor relating to duty cycle stems from the variation of the data pattern presented to the inputs of the transmitter. This is considered a very coarse adjustment as it can only be varied by a minimum of a single-serial bit-time. The last factor is rise and fall time, and is largely dependent on the circuit the outputs are driving.

Rise and Fall Time

Rise and fall times are defined as the period of time required for a signal to transition from a logic LOW to a logic HIGH or a logic HIGH to a logic LOW. The rise time of an ECL output is mainly determined by internal parameters such as the internal driver resistor and the parasitic capacitance of the output and is generally fixed. The fall time however, is generally based on the biasing of the output, the load capacitance, and the termination of the clock circuit. If each of the outputs are properly biased and treated as a transmission line, the driver is capable of matching rise and fall times. A proper biasing technique is to tie the PECL output to V_{CC} – 2.0V through a 50Ω resistor. Since ECL outputs switch at such high speeds, typically in the 1-ns range, most ECL circuit board traces greater than 1 inch in length should be treated as transmission lines and require termination (Reference 3). When a circuit board trace acts as a transmission line and is unterminated, it will exhibit a reflection of the energy pulse from the destination back to the source. If this reflection is significant, it can cause erroneous triggering of digital logic circuits. The CY7B923 datasheet indicates a maximum rise time and fall time of 1.2 ns measured at the 80% and 20% voltage points driving an ECL load of 5 pF and 50Ω terminated to V_{CC} – 2.0 Volts. This is specified as a guaranteed maximum. Typical rise and fall times are less than the maximum.

Termination

The two types of termination techniques generally used to control transmission line effects are series and parallel termination. A series termination is designed to match the source driver to the characteristic impedance of the line being driven. This termination approach is not recommended for the HOTLink Transmitter. A parallel termination, on the other hand, will match the characteristic impedance of the line being driven to the load. This termination procedure, also called a Thevenin Termination, consists of a pull-up resistor to the positive supply and a pull-down resistor to the negative supply. This termination can also double as a biasing network and serve both purposes: transmission line termination and ECL output biasing. An example of an improperly terminated ECL waveform is shown in *Figure 3*. Notice the excessive ringing on the logic LOW level. *Figure 4* shows what a properly terminated ECL signal should look like. Notice the symmetrical rise and fall times and the absence of any ringing on the waveform.

Refer to the Cypress Semiconductor *Applications Handbook* or the Cypress Semiconductor "HOTLink Design Considerations" Application Note for more information on transmission line termination techniques.

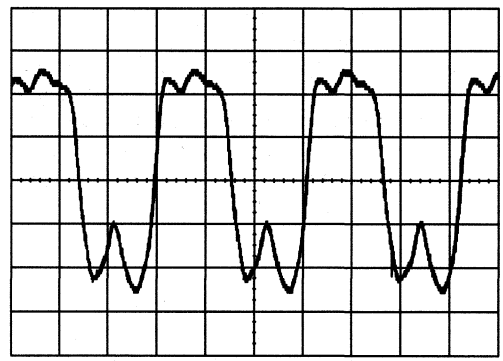


Figure 3. Improperly Terminated Waveform

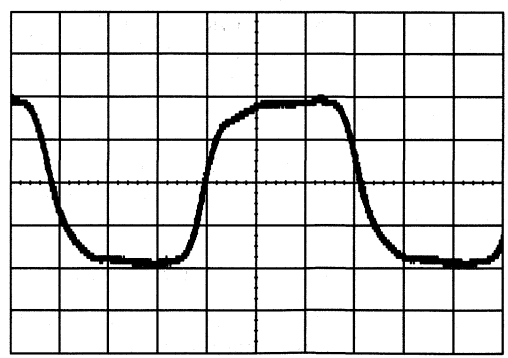


Figure 4. Properly Terminated ECL Waveform

Clock Skew

Clock skew is introduced into a digital system in two ways. The first is called output skew and is defined as the difference in time between clock edges being driven from the OUTA, OUTB, and OUTC transmitter output pairs. Output skew is caused internally by the clock driver circuit itself. It can result from the differences in output driver characteristics between output pairs or even in layout and placement differences of the physical driver structures on the die. The second source of clock skew is related to the printed circuit board layout and placement. Trace length, capacitive loading, termination components, printed circuit board characteristics, supply voltages, and many other factors affect these external delays. It is important that the designer understand the issues affecting clock skew because one must be able to accurately predict when clock edges will arrive at a load or destination for proper synchronization of a digital system.

Drive Capability

The HOTLink Transmitter features three sets of differential PECL/ECL outputs. Each of these outputs is capable of driving a 50Ω load with a maximum output current of 50 mA.

Power Supply Current

The HOTLink Transmitter has a maximum I_{CCT} specification of 85 mA for commercial and 95 mA for military temperature devices. Additionally, each enabled output pair contributes 35 mA to I_{CCT} when loaded to 50Ω. Unused outputs may be left open, or better yet, tied to V_{CC} to minimize the power dissipated by the output circuit and reduce a source of unwanted noise. A 5-mA power savings can be obtained by disabling the output current source in this manner.

HOTLink Transmitter Printed Circuit Layout

Care must be taken when laying out a printed circuit board for the HOTLink Transmitter and when designing any clock circuit in general. Proper power-supply filtering and bypass techniques must be employed to ensure reliable operation and the correct components must be selected. Everything from the oscillator used to feed the CKW input to the type and placement of the bypass capacitors used is critical. Refer to the Cypress Semiconductor "HOTLink Design Considerations" Application Note for specific details on circuit layout and bypassing.

Device Packaging

Like virtually all Cypress devices, the CY7B923 HOTLink Transmitter is available in commercial (0 to 70 degrees C), industrial (−40 to +85 degrees C) and military (−55 to +125 degrees C) temperature ranges at V_{CC} ± 10%. The device comes packaged in a 28-pin PLCC, 28-pin LCC, or a 28-pin 300-mil-wide SOIC to suit a broad range of packaging requirements. The device is not available in Dual-In-Line (DIP) or through-hole packages due to the excessive lead-frame inductance and its effect on device performance.

Conclusion

The HOTLink Transmitter offers the designers of pseudo ECL systems an alternative to the expensive, high-power clock sources currently available on the market. The combination of BiCMOS process technology and robust feature set makes CY7B923 suitable for many PECL logic circuit clock generation applications where cost, power, flexibility, and performance are of prime concern.

References

1. Blood Jr., William R., *MECL System Design Handbook*, Fourth Edition, 1988.
2. Cypress Semiconductor Corporation, *High Performance Databook*, 1993 Edition.
3. Cypress Semiconductor Corporation, HOTLink Design Considerations, Application Note.
4. Cypress Semiconductor Corporation, *Applications Handbook*, 1993 Edition.

Document #: COM0011−0394

HOTLink is a trademark of Cypress Semiconductor Corporation.

CY9266 HOTLink™ Demo Board User's Guide

Overview

This document describes the construction, interfaces, and operation of the CY9266-F (optical fiber), CY9266-T (shielded twisted pair/twinax), and CY9266-C (coaxial cable) HOTLink™ Demo Boards. These boards implement a complete bi-directional parallel-to-serial and serial-to-parallel communications link, capable of operation at serial rates of 160- to 330-Mbits/second (16- to 33-Mbytes/second). The supported rate of communication may be limited by the specific type and speed-grade of optical module or copper cable type used.

The CY9266 demo board is electrically and mechanically compatible with the ANSI X3T11 Fibre Channel Interface, as documented in the ANSI standard ANS X3.230–199X. It provides three different methods of access for the TTL parallel interface and supervisor functions, for testing or exercising the serial data link.

Block Diagram

The block diagram in *Figure 1* illustrates the major functional blocks contained in the CY9266. These include

- 10-bit TTL parallel transmit data input
- 10-bit TTL parallel receive data output
- Selectable Encoded or Bypass operation modes
- On-board oscillator
- Selectable internal/external clocking
- Selectable carrier-detect polarity
- Selectable local loopback
- Power supply voltage monitor
- Built-In Self-Test (BIST) pattern generation and checking hardware with error/status display

Board Connectors

This board offers three primary methods of TTL-level access

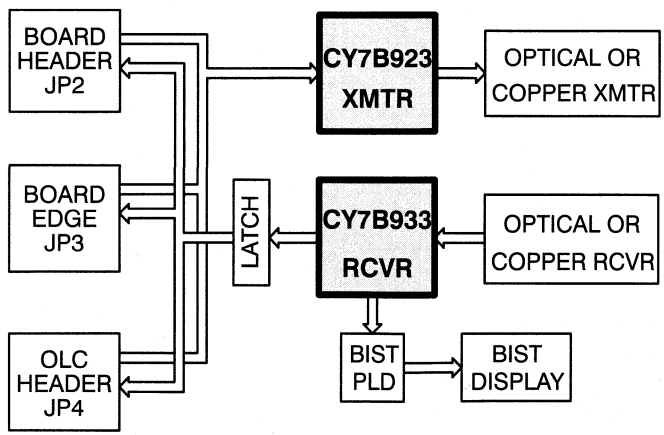


Figure 1. HOTLink Optical Demo Board Block Diagram

- JP2—A 58-position (2 x 29) set of holes, capable of accepting a 0.025″ sq. pin-header on the top or bottom of the board
- JP3—A 60-position (2 x 30) 0.1″ spaced board-edge finger stock
- JP4—A 48-position (4 x 12 matrix) 0.025″ sq. pin-header mounted on the bottom of the board

Connectors JP2 and JP3 provided access to all data input and output buses as well as all BIST, control, and clocking signals for the HOTLink Transmitter and Receiver. These connectors may be used individually or together since all signals present on JP2 are also present on JP3. Power for the board is also brought in through these same connectors.

Connector JP4 is positioned and pinned to match up with the connector and signals present on other industry standard Fibre Channel modules. Unlike these other modules (which may contain two full-duplex channels), this demo board only provides a single full-duplex channel. While sufficient room exists to build a board with two channels, other functionality was added (on-board oscillator, BIST PLD and display, etc.) in this space to allow better testing and demonstration of the enhanced capabilities present in the Cypress HOTLink parts.

An additional jumper block (JP1) is used to configure three of the operating characteristics of the board: clock sourcing, serial output enable (FOTO), and local loopback control.

Optical Modules

The CY9266-F demo board is designed to operate with industry-standard footprint optical modules. The demo board contains low-profile socket pins so the user may select and test optical modules from different vendors. This board accepts both the four-row DIP and the single-row endfire types of modules.

These modules are available from multiple vendors with either ST- or SC-type optical fiber connectors. Because these modules are all LED-based, they are not required to meet many of the safety standards (ANSI Z136.1 and Z136.2, F.D.A. regulation 21 CFR subchapter J, and IEC 825) necessary for laser-based modules. These modules should be used with 62.5/125-μm multimode fiber.

Coaxial Cables

The CY9266-C demo board is configured to support 75Ω coaxial cables that attach through BNC/TNC connectors. Other cable impedances may be used with the board by changing the value of the termination resistors on the board.

Shielded-Pair Cables

The CY9266-T demo board is configured to support 150Ω shielded twisted pair or twinaxial cable that attaches through a 9-pin D-sub connector. Other cable impedances may be used with the board by changing the value of the termination resistors on the board.

BIST Support

The CY9266 contains an on-board control PLD and a two-digit error-count display that are used in conjunction with the BIST (Built-In Self-Test) capability of the Cypress Semiconductor HOTLink Transmitter and Receiver. This capability allows the parts, and any serial link, to be exercised and monitored at their full data rate without the use of expensive external test equipment.

The BIST PLD (CY7C344) contains a simple state machine that monitors the HOTLink Receiver BIST state, and an error-counter that drives an external display. The complete contents of this PLD are documented in Appendix C.

This BIST PLD also drives the four decimal point LEDs on the displays. These indicators are used to present additional status information about the state of the board, the BIST state machine, and the serial link.

Design Criteria

The CY9266 demo board was designed as a low-cost demonstration vehicle for the Cypress Semiconductor HOTLink family of data communications parts. The goals of this board are to

- Present a Fibre Channel interface board that is fully compliant with the mechanical, electrical,

optical, coding, and protocol specifications in levels 0 and 1 of the ANSI Fibre Channel standard

- Allow full data rate testing of the serial link without expensive test equipment
- Allow the user to exercise all modes of operation of the receiver and transmitter
- Offer various parallel attachment methods for simplified system interfacing
- Offer various media types for evaluation
- Allow simple interfacing to existing OLC-compatible test platforms

Because of the flexibility inherent in the HOTLink parts, these goals were easily achieved.

Three electrical connection methods are provided: a 60-pin board-edge connector, a 58-pin (2 x 29) 0.025″ square pin-header, and a 48-pin (4 x 12) 0.025″ square pin-header. These forms of attachment allow the user to select the connector form that best suits their desired mode of attachment.

The HOTLink Transmitter and Receiver contain a Built-In Self-Test (BIST) capability. This capability was designed into the HOTLink parts to allow high-speed serial testing without expensive test equipment. All hardware necessary to exercise and monitor the BIST function is present on the CY9266 board. This hardware allows a Bit-Error-Rate (BER) test to be performed without additional equipment.

The BIST capability of the HOTLink Transmitter and Receiver allows offline testing of the transmitter, receiver, and serial link, by performing a byte-by-byte comparison of the data while a 511-byte pseudo-random byte stream is repeatedly sent, received, and checked.

Through use of either JP2 or JP3, users may exercise all modes of operation of the parts. JP4 is configured as a functional system interface, and thus does not include all the mode, clock, and special control signals present on JP2 and JP3, all of which may be selected or controlled in JP1 or S1.

Connector Pin Numbering

JP2 – 58-Position Pin-Header

The 58-position pin-header (JP2) holes are located next to the board-edge connector. Pin 1 of this connector area is identified on the board by a square solder pad. The remaining pin locations use a round solder pad.

The connector hole pattern is made to accept 58 0.025″ square pins soldered into the board. The numbering for this connector is shown in *Figure 2*.

Note: The numbering of this connector is specified to match up with standard 0.050″ centerline flat cable connectors. Because of the location of pin 1 of this hole pattern, the mating pins for this connector should normally be on the bottom of the board. If a connector is instead attached to the top side of the board, the even- and odd-numbered pins of the connector are effectively swapped. This means that conductor 1 of a cable attached to the top side of the

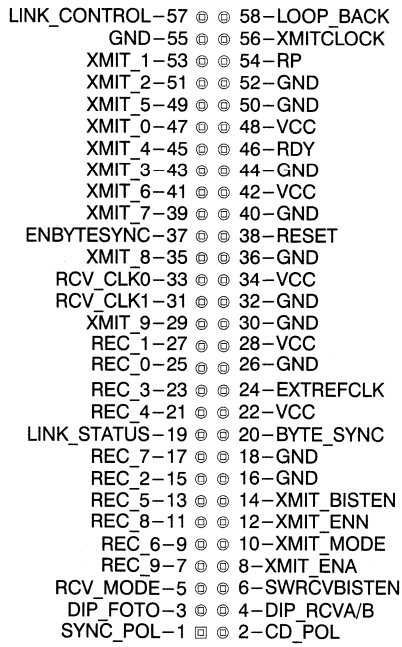


Figure 2. JP2 Pin Numbering, Top Side of Board View

board is in reality connected to the signal listed for pin 2 in *Table 1*.

JP3 – 60-Position Board-Edge

The 60-position board-edge connector (JP3) is a section of gold plated 0.062″ board finger-stock that connects to the same signals as JP2. Contact centerline for this connector is 0.1″, with even- and odd-numbered signals on opposing sides of the board.

To prevent the demo board from being plugged into a mating connector backwards (and possibly damaging it), a 0.040″ x 0.450″ keying slot is present between contacts 3/4 and 5/6. The pin numbering for this connector is shown in *Figure 3*.

Note: The numbering of this connector is specified to match up with standard 0.050″ centerline flat-cable connectors. Because of the location of pin 1 of this board-edge connector, the mating connector would normally be a mass-terminate board-edge to flat-cable type connector. If a standard board-edge connector is used instead, the even and odd numbered pins of the connector are effectively swapped. This means that pin 1 of a standard board-edge connector is in reality connected to the signal listed for pin 2 in *Table 1*.

JP4 – OLC-Compatibility Connector

The JP4 (OLC-compatibility) connector is located on the bottom (passive-component) side of the board. Pin 1 of this connector is identified on the board by a square solder pad. The remaining pins use a round solder pad.

For the CY9266 demo board, pins of sufficient length are present so that analysis equipment may be attached to these signal pins on the top (active-component) side of the board while it is plugged into a mating connector. The numbering sequence for the JP4 connector pins is shown in *Figure 4*.

The connector is made from 48 0.025″ square pins soldered into the board. To allow full mating with an

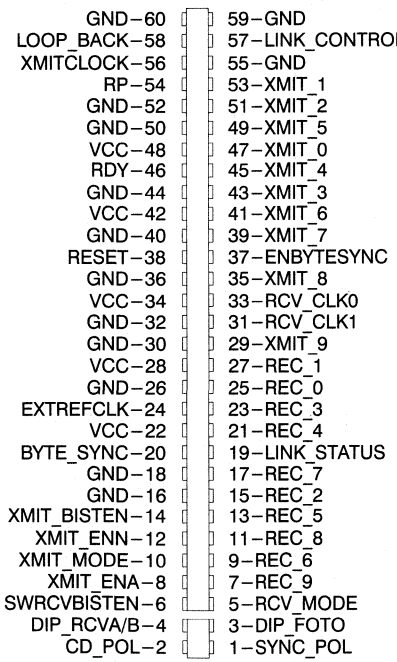


Figure 3. JP3 Pin Numbering, Edge of Board

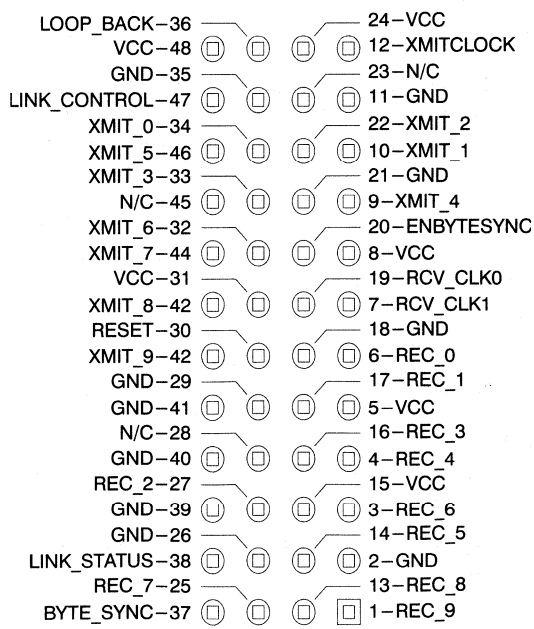


Figure 4. JP4 Pin Numbering, Top Side of Board View (Pins Are On The Bottom)

OLC-compatible connector, these pins must extend at least 0.250″ beyond the bottom surface of the board.

Connector Pinouts

The CY9266 provides three interface connectors to the user: JP2, JP3, and JP4. *Table 1* shows which signal is present on each connector pin.

Table 1. I/O Connector Pinouts

Pin No.	JP3	JP2	JP4	Pin No.	JP3	JP2	JP4
1	SYNC_POL	SYNC_POL	REC_9	31	RCV_CLK1	RCV_CLK1	VCC
2	CD_POL	CD_POL	GND	32	GND	GND	XMIT_6
3	DIP_FOTO	DIP_FOTO	REC_6	33	RCV_CLK0	RCV_CLK0	XMIT_3
4	DIP_RCVA/B	DIP_RCVA/B	REC_4	34	VCC	VCC	XMIT_0
5	RCV_MODE	RCV_MODE	VCC	35	XMIT_8	XMIT_8	GND
6	SWRCVBIS-TEN	SWRCVBIS-TEN	REC_0	36	GND	GND	LOOP_BACK
7	REC_9	REC_9	RCV_CLK1	37	ENBYTE-SYNC	ENBYTE-SYNC	BYTE_SYNC
8	XMIT_ENA	XMIT_ENA	VCC	38	RESET	RESET	LINK_STA-TUS
9	REC_6	REC_6	XMIT_4	39	XMIT_7	XMIT_7	GND
10	XMIT_MODE	XMIT_MODE	XMIT_1	40	GND	GND	GND
11	REC_8	REC_8	GND	41	XMIT_6	XMIT_6	GND
12	XMIT_ENN	XMIT_ENN	XMITCLOCK	42	VCC	VCC	XMIT_9
13	REC_5	REC_5	REC_8	43	XMIT_3	XMIT_3	XMIT_8
14	XMIT_BIS-TEN	XMIT_BIS-TEN	REC_5	44	GND	GND	XMIT_7
15	REC_2	REC_2	VCC	45	XMIT_4	XMIT_4	N/C
16	GND	GND	REC_3	46	RDY	RDY	XMIT_5
17	REC_7	REC_7	REC_1	47	XMIT_0	XMIT_0	LINK_CON-TROL
18	GND	GND	GND	48	VCC	VCC	VCC
19	LINK_STA-TUS	LINK_STA-TUS	RCV_CLK0	49	XMIT_5	XMIT_5	
20	BYTE_SYNC	BYTE_SYNC	ENBYTE-SYNC	50	GND	GND	
21	REC_4	REC_4	GND	51	XMIT_2	XMIT_2	
22	VCC	VCC	XMIT_2	52	GND	GND	
23	REC_3	REC_3	N/C	53	XMIT_1	XMIT_1	
24	EXTREFCLK	EXTREFCLK	VCC	54	RP	RP	
25	REC_0	REC_0	REC_7	55	GND	GND	
26	GND	GND	GND	56	XMITCLOCK	XMITCLOCK	
27	REC_1	REC_1	REC_2	57	LINK_CON-TROL	LINK_CON-TROL	
28	VCC	VCC	N/C	58	LOOP_BACK	LOOP_BACK	
29	XMIT_9	XMIT_9	GND	59	GND		
30	GND	GND	RESET	60	GND		

Table 2. Transmit Bus Signal Name Map

Transmit Bus Input Pin Name	HOTLink Transmitter Pin Name	
	Encoded Mode	Bypass Mode
XMIT_0	SC/$\overline{D}$	Da
XMIT_1	D0	Db
XMIT_2	D1	Dc
XMIT_3	D2	Dd
XMIT_4	D3	De
XMIT_5	D4	Di
XMIT_6	D5	Df
XMIT_7	D6	Dg
XMIT_8	D7	Dh
XMIT_9	SVS	Dj

Signal Naming Conventions

There are three types of signal names used throughout this document: I/O connector pin names, on-board signal names, and HOTLink Transmitter and Receiver pin names. Except for the transmit and receive data buses, these names are unique.

The names used for the transmit and receive data bus pins on connectors JP2, JP3, and JP4 are different from the signal names present on the HOTLink Transmitter and Receiver. The functional names for these signals also change depending on the current operating mode of the HOTLink Transmitter or Receiver. *Table 2* lists the transmit data bus signals and the names mapped to them in each transmitter mode.

The output data bus from the HOTLink Receiver is pipelined with a single register stage between the receiver outputs and the board output pins. *Table 3* lists the receive data bus signals and the names mapped to them in each receiver mode.

Table 3. Receive Bus Signal Name Map

Receive Bus Output Pin Name	HOTLink Receiver Pin Name	
	Decode Mode	Bypass Mode
REC_0	SC/$\overline{D}$	Qa
REC_1	Q0	Qb
REC_2	Q1	Qc
REC_3	Q2	Qd
REC_4	Q3	Qe
REC_5	Q4	Qi
REC_6	Q5	Qf
REC_7	Q6	Qg
REC_8	Q7	Qh
REC_9	RVS	Qj

Signal Descriptions

The I/O signals listed in *Table 1* fall into six groups: power, switched control, control, status, clock, and data. These signals are described in *Table 4*.

Table 4. I/O Signal Descriptions

Signal Name	Group	Description
V_{CC}	Power	+5 VDC @ 1.0A typical
GND	Power	Ground
XMIT_BISTEN	Input, Switched Control	**Transmitter BIST Enable (S1-1).** When this signal is LOW, the HOTLink Transmitter is placed into its BIST mode. Exact operation of the transmitter is also determined by the settings of the $\overline{ENA}$ (S1-4) and $\overline{ENN}$ (S1-3) signals. With both $\overline{ENA}$ and $\overline{ENN}$ HIGH, the transmitter outputs an alternating 0−1 pattern (D10.2 or D21.5). If either $\overline{ENA}$ or $\overline{ENN}$ is LOW, the transmitter sends a repeating 511-character test sequence. The receiver contains a matching mode that allows this transmitter BIST mode to be used to test the entire serial link without external hardware. The transmitter BIST enable is kept separate from the receiver BIST enable on this board to allow each component to be tested with external patterns that are not part of the built-in BIST sequence.

Table 4. I/O Signal Descriptions (continued)

Signal Name	Group	Description
XMIT_MODE	Input, Switched Control	**Encoder Mode Select (S1-2).** This signal is used to select whether pre-encoded (10-bit) or non-encoded (8-bit) data is clocked into the HOTLink Transmitter. When LOW (Encoded mode), this input enables the internal 8B/10B encoder and accepts 8-bit parallel data from the transmitter data bus (D0−D7 as listed in *Table 2*). When HIGH (Bypass mode), the encoder is bypassed and a 10-bit pattern is accepted (Da−Dj as listed in *Table 2).*
XMIT_ENN	Input, Switched Control	**Enable Next Parallel Transmitter Data (S1-3).** This signal is used to control when data is loaded into the HOTLink Transmitter. When this signal is LOW at the rising edge of CKW, the data present on the transmitter inputs at the next rising edge of CKW is loaded, processed, and sent. When this signal is HIGH, the transmitter ignores the data present on its inputs at the next rising edge of CKW and instead inserts a SYNC character (K28.5) to fill in the data stream. When $\overline{\text{ENA}}$ is used for data control, the $\overline{\text{ENN}}$ signal should be tied HIGH, but may be used to enable BIST mode.
XMIT_ENA	Input, Switched Control	**Enable Parallel Transmitter Data (S1-4).** This signal is used to control when data is loaded into the HOTLink Transmitter. When LOW at the rising edge of CKW, the data present on the transmitter inputs is loaded, processed, and sent. When this signal is HIGH, the transmitter ignores the data present on its inputs and instead inserts a SYNC character (K28.5) to fill in the data stream. When $\overline{\text{ENN}}$ is used for data control, the $\overline{\text{ENA}}$ signal should be tied HIGH, but may be used to enable BIST mode.
SWRCVBIS-TEN	Input, Switched Control	**Receiver BIST Enable (S1-5).** When this signal is LOW, the HOTLink Receiver monitors the data stream for the BIST loop initialization character (D0.0). This signal also enables the BIST PLD (CY7C344-U8), which is used to monitor the progress and status of the BIST loop through the receiver $\overline{\text{RDY}}$ and RVS outputs. When the receiver detects the initialization character, it begins comparing received data with a built-in data sequence that can be used to verify the proper functionality of the transmitter, receiver, and the serial link connecting them. The receiver BIST enable is kept separate from the transmitter BIST enable on this board to allow each component to be tested with external patterns that are not part of the built-in BIST sequence.
RCV_MODE	Input, Switched Control	**Receiver Mode Select (S1-6).** This signal is used to select whether encoded (10-bit) or non-encoded (8-bit) data is output from the receiver. When LOW (Decode mode), this input enables the internal 10B/8B decoder and outputs 8-bit parallel data (Q0−Q7 as listed in *Table 3*). When HIGH (Bypass mode), the decoder is bypassed and a 10-bit pattern is output (Qa−Qj as listed in *Table 3*).

Table 4. I/O Signal Descriptions (continued)

Signal Name	Group	Description
DIP_RCVA/B	Input, Switched Control	**DIP-Switch Controlled Receiver A/$\overline{B}$ Port Select (S1-7).** This signal is used to determine which port (INA± or INB±) the receiver uses for the input serial data stream. When LOW, this signal selects the receiver B port that is directly connected to the C port on the transmitter. When HIGH, this signal selects the receiver A port that is connected to the optical receiver output. This signal is also routed through jumper block JP1. In order for this signal to control the port selection of the receiver, it is necessary to have a shorting jumper across the X and Y pins of JP1-C. To allow the LOOP_BACK signal on the I/O connectors (JP2, JP3, and JP4) to control the A/$\overline{B}$ port selection, this jumper should be moved to JP1-B.
DIP_FOTO	Input, Switched Control	**DIP-Switch Controlled FOTO (S1-8).** This signal is used to enable the A and B differential output drivers of the HOTLink Transmitter. When this signal is LOW, the differential outputs are allowed to follow the pattern of the data serialized by the transmitter. When this signal is HIGH, the A and B differential outputs of the transmitter are driven to a logic zero state (+ output is logic HIGH, − output is logic LOW). This places an attached optical transmitter in a state where no light is output. This signal is also routed through jumper block JP1. In order for this signal to control the FOTO enable on the transmitter, it is necessary to have a shorting jumper across the X and Y pins of JP1-E. To allow the LINK_CONTROL signal on the I/O connectors (JP2, JP3, and JP4) to control the FOTO enable, this jumper should be moved to JP1-F.
CD_POL	Input, Switched Control	**Carrier-Detect Polarity Select (S1-9).** This input selects the output polarity of the LINK_STATUS signal. When LOW, the LINK_STATUS signal is HIGH when a valid carrier is present. When HIGH, the LINK_STATUS signal is LOW when a valid carrier is present.
SYNC_POL	Input, Switched Control	**Byte Sync Polarity Select (S1-10).** This input, in conjunction with the HOTLink Receiver MODE input, selects the active level of the BYTE_SYNC signal. When LOW with the receiver in Bypass mode, the BYTE_SYNC signal is LOW when a K28.5 SYNC character is present on the receive data bus. When HIGH with the receiver in Bypass mode, the BYTE_SYNC signal is HIGH when a K28.5 SYNC character is present on the receive data bus. When LOW with the receiver in Decode mode, the BYTE_SYNC output remains HIGH for strings of K28.5 SYNC characters, or while awaiting the first K28.5 SYNC character after being placed into Reframe mode (RF is set HIGH). When HIGH with the receiver in Decode mode, the BYTE_SYNC output remains LOW for strings of K28.5 SYNC characters, or while awaiting the first K28.5 SYNC character after being placed into Reframe mode (RF is set HIGH).

Table 4. I/O Signal Descriptions (continued)

Signal Name	Group	Description
LOOP_BACK	Input, Control	**Loopback Control.** This signal is used to determine which port (A or B) the HOTLink Receiver uses for the input serial data stream. When LOW, this signal selects the receiver B port that is connected directly to the transmitter C port. When HIGH, this signal selects the receiver A port that is connected to the optical receiver output. This signal is also routed through jumper block JP1. In order for this signal to control the port selection of the receiver, it is necessary to have a shorting jumper across the X and Y pins of JP1-B. To allow the DIP_RCVA/B signal (S1-7, also present on JP2 and JP3) to control the A/$\overline{B}$ port selection, this jumper should be moved to JP1-C.
ENBYTESYNC	Input, Control	**Enable Byte Sync Detect.** This signal controls when the HOTLink receiver is allowed to reframe to the incoming serial data (e.g., acquire byte sync). When this signal is HIGH, each K28.5 SYNC character received in the shifter will frame the data that follows. When this signal is LOW, the framing logic in the receiver is disabled. Because the CKR output of the receiver must line up with the reframed data, it is possible to generate significant phase jumps in the CKR clock. To prevent the generation of very short high or low pulses on the CKR output (which could cause timing violations in downstream logic) the Cypress HOTLink Receiver uses look-ahead hardware to prevent these short pulses. Instead, a portion of the clock period for the character preceding the reframed data is lengthened.
LINK_CONTROL	Input, Control	**Link Control.** This signal is used to enable the A and B differential output drivers of the HOTLink Transmitter. When this signal is LOW, the differential outputs are allowed to follow the pattern of the data serialized by the transmitter. When this signal is HIGH, the A and B differential outputs of the transmitter are driven to a logic zero state (+ output is logic HIGH, − output is logic LOW). This places an attached optical transmitter in a state where no light is output. This signal is also routed through jumper block JP1. In order for this signal to control the FOTO enable on the transmitter, it is necessary to have a shorting jumper across the X and Y pins of JP1-F. To allow the DIP_FOTO signal on the I/O connectors (JP2 and JP3) to control the FOTO enable, this jumper should be moved to JP1-E.
RESET	Output, Status	**Reset/Power OK.** This output is used to emulate the voltage monitor function present on the OLC card. It remains active (LOW) until the V_{CC} input to the board is above 4.65 VDC. This output also becomes active when the BIST RESET switch (S2) is pressed.
LINK_STATUS	Output, Status	**Link Status.** This signal operates as a carrier-detect status for the serial interface. The polarity of this signal is determined by the CD_POL input (S1-9). When CD_POL is LOW, LINK_STATUS drives HIGH when a carrier is present. When CD_POL is HIGH, LINK_STATUS drives LOW when a carrier is present.

Table 4. I/O Signal Descriptions (continued)

Signal Name	Group	Description
RP	Output, Clock	**Read Pulse.** This is a 60% LOW duty-cycle pulse train suitable for clocking data out of Cypress's CY7C42X family of asynchronous FIFOs. This pulse is generated by the HOTLink Transmitter in response to the XMIT_ENA input being active at the rising edge of CKW. For repeated pulses the $\overline{RP}$ period is the same as CKW, yet is totally independent of the duty cycle of CKW. When the transmitter is in BIST mode, the $\overline{RP}$ signal remains HIGH for all but the last byte of the BIST loop, where it pulses LOW.
XMITCLOCK	Input, Output, Clock	**Transmitter External Clock.** This is the external byte-rate clock input. This clock is used to drive the transmitter CKW input. To allow for operation using the on-board oscillator, the XMITCLOCK signal is run through jumper block JP1. To operate using an external HOTLink Transmitter clock source, a shorting jumper should be placed across pins X and Y of JP1-G. To use the on-board oscillator instead, this shorting jumper should be moved to connect pin JP1-GY to JP1-HY. When operated from XMITCLOCK, the receiver REFCLK may also be set to use this same clock. This is done by placing a shorting jumper across pins JP1-GX and JP1-HX. To allow the receiver REFCLK to operate from the on-board oscillator, this jumper should be moved to connect the X and Y pins of JP1-I. The on-board oscillator may also be driven out on the XMITCLOCK line by placing a shorting jumper across pins X and Y of JP1-H.
EXTREFCLK	Input, Output, Clock	**External Reference Clock.** This byte-rate clock is used to drive the HOTLink Receiver REFCLK from an external source other than XMITCLOCK. This input may be used to test the tracking and capture range of the receiver PLL. It may also be used to operate the receiver at a different data rate from the transmitter. To allow the receiver PLL to properly lock to the received serial stream, this clock must be within 0.1% of the clock used to generate the received serial data. To drive the receiver REFCLK from this clock source, a shorting jumper should be placed across pins JP1-IX and JP1-JX. The on-board oscillator may also be selected to drive the EXTREFCLK line by placing a shorting jumper across pins X and Y of JP1-J. With this jumper in place it is still possible to drive the receiver REFCLK input from the on-board oscillator by placing a shorting jumper across the X and Y pins of JP1-I.
RCV_CLK0	Output, Clock	**Receive Clock 0.** This is the byte-rate recovered clock used for received data. The period of this clock is determined by the serial data rate entering the HOTLink Receiver. The duty-cycle of this signal is determined by the receiver and is fixed at 50%. This clock may experience a large phase jump when reframing to a serial data stream. The phasing on this clock is such that the rising edge of the clock occurs coincident with the start of each interval where a character is present on the output received data bus. This signal is a buffered form of the HOTLink Receiver CKR clock.

Table 4. I/O Signal Descriptions (continued)

Signal Name	Group	Description
RCV_CLK1	Output, Clock	**Receive Clock 1.** This is the byte-rate recovered clock used for received data. The period of this clock is determined by the serial data rate entering the HOTLink Receiver. The duty-cycle of this signal is determined by the receiver and is fixed at 50%. This clock may experience a large phase jump when reframing to a serial data stream. The phasing on this clock is such that the rising edge of the clock occurs near the center of each interval where a character is present on the output received data bus. This signal is a buffered and inverted form of the HOTLink Receiver CKR clock.
RDY	Output, Clock	**$\overline{\text{RDY}}$ (Ready).** This signal is used both as a HOTLink Receiver data output clock and a status indicator for the receiver when in BIST mode. This is an unbuffered output from the receiver. It is normally used to clock valid data from the receiver data bus into asynchronous FIFOs. Because of the additional pipeline register in the data bus (added for OLC compatibility) this signal will operate one byte *prior* to the data being available at the I/O connectors.
BYTE_SYNC	Output, Data	**Byte Sync Detected.** This signal is a pipelined form of the receiver $\overline{\text{RDY}}$ output. This additional pipeline stage for the $\overline{\text{RDY}}$ signal (and the rest of the receiver data bus) was added to match the specific timing of the OLC Byte Sync signal. The active level of this output is determined both by the operating mode of the HOTLink Receiver and by the state of the SYNC_POL input. With the HOTLink Receiver in Bypass mode, the BYTE_SYNC signal is used as a K28.5 SYNC character indicator. With SYNC_POL LOW, BYTE_SYNC is LOW when a K28.5 SYNC character is present on the receive data bus. With SYNC_POL HIGH, BYTE_SYNC is HIGH when a K28.5 SYNC character is present on the receive data bus. With the receiver in Decode mode, the BYTE_SYNC signal is used as a valid data indicator. With SYNC_POL LOW, BYTE_SYNC is LOW whenever a usable data byte is present on the receive data bus. With SYNC_POL HIGH, BYTE_SYNC is HIGH whenever a usable data byte is present on the receive data bus.
REC_9	Output, Data	**RVS(Qj).** This signal is a series-terminated, pipelined form of the HOTLink Receiver RVS(Qj) signal. This termination and additional pipeline stage for the RVS(Qj) signal (and the rest of the receive data bus) was added to match the specific timing and signal characteristics of the OLC card.
REC_8	Output, Data	**Q7(Qh).** This signal is a series-terminated, pipelined form of the HOTLink Receiver Q7(Qh) signal.
REC_7	Output, Data	**Q6(Qg).** This signal is a series-terminated, pipelined form of the HOTLink Receiver Q6(Qg) signal.
REC_6	Output, Data	**Q5(Qf).** This signal is a series-terminated, pipelined form of the HOTLink Receiver Q5(Qf) signal.

Table 4. I/O Signal Descriptions (continued)

Signal Name	Group	Description
REC_5	Output, Data	**Q4(Qi).** This signal is a series-terminated, pipelined form of the HOTLink Receiver Q4(Qi) signal.
REC_4	Output, Data	**Q3(Qe).** This signal is a series-terminated, pipelined form of the HOTLink Receiver Q3(Qe) signal.
REC_3	Output, Data	**Q2(Qd).** This signal is a series-terminated, pipelined form of the HOTLink Receiver Q2(Qd) signal.
REC_2	Output, Data	**Q1(Qc).** This signal is a series-terminated, pipelined form of the HOTLink Receiver Q1(Qc) signal.
REC_1	Output, Data	**Q0(Qb).** This signal is a series-terminated, pipelined form of the HOTLink Receiver Q0(Qb) signal.
REC_0	Output, Data	**SC/$\overline{D}$(Qa).** This signal is a series-terminated, pipelined form of the HOTLink Receiver SC/$\overline{D}$(Qa) signal.
XMIT_9	Input, Data	**SVS(Dj).** This signal is the SVS(Dj) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_8	Input, Data	**D7(Dh).** This signal is the D7(Dh) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_7	Input, Data	**D6(Dg).** This signal is the D6(Dg) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_6	Input, Data	**D5(Df).** This signal is the D5(Df) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_5	Input, Data	**D4(Di).** This signal is the D4(Di) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_4	Input, Data	**D3(De).** This signal is the D3(De) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_3	Input, Data	**D2(Dd).** This signal is the D2(Dd) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_2	Input, Data	**D1(Dc).** This signal is the D1(Dc) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_1	Input, Data	**D0(Db).** This signal is the D0(Db) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.
XMIT_0	Input, Data	**SC/$\overline{D}$(Da).** This signal is the SC/$\overline{D}$(Da) input to the HOTLink Transmitter. It is latched into the transmitter in the rising edge of CKW, when enabled by $\overline{ENA}$ or $\overline{ENN}$.

Power Signals

The CY9266 demo board is designed to operate from a single +5V ±10% DC supply capable of delivering 1.0A (typical). All V_{CC} and GND pins on JP2, JP3, and JP4 are (respectively) common to each other. There are no distinctions made for separate supplies pins for the different logic sections.

Switched Control Signals

The CY9266 demo board contains a 10-position DIP switch (S1). This switch is connected in parallel with a number of control signals on JP2 and JP3. Each of these control signals is pulled-up by a 5-kΩ resistor through R-pack R20. None of these Switched Control signals are available at the JP4 connector.

The signals present in this group are

- XMIT_BISTEN (S1-1)
- XMIT_MODE (S1-2)
- XMIT_ENN (S1-3)
- XMIT_ENA (S1-4)
- SWRCVBISTEN (S1-5)
- RCV_MODE (S1-6)
- DIP_RCVA/B (S1-7)
- DIP_FOTO (S1-8)
- CD_POL (S1-9)
- SYNC_POL (S1-10)

To allow these signals to be controlled through the external connectors (JP2 and JP3), the corresponding S1 switch must be in the off (open) position. Care should be taken when driving these signals, as any switch inadvertently left in the closed position will present a direct short to ground for an attached driver.

Control Signals

In addition to the Switched Control signals that are only present on JP2 and JP3, three additional control inputs are present that connect to JP2, JP3, and JP4.

These control signals are

- LOOP_BACK
- ENBYTSYNC
- LINK_CONTROL

These control inputs are connected directly to the HOTLink Transmitter or Receiver. Because the HOTLink parts contain internal pull-up resistors on their TTL compatible inputs, these signals may be driven with either open-collector buffers, CMOS, or TTL drive levels.

Status Signals

Two status output signals (RESET and LINK_STATUS) are provided at all three I/O connectors. The RESET signal is a slow-speed signal and does not require the series termination used with LINK_STATUS.

Clock Signals

Six signals are available at the I/O connectors that are used as clocks in some form. Two of these (XMITCLOCK and EXTREFCLK) are input/output clocks that are routed through the JP1 jumper block, and three are output clocks.

These clock signals are

- XMITCLOCK
- EXTREFCLK
- $\overline{RP}$
- $\overline{RDY}$
- RCV_CLK0
- RCV_CLK1

Of the output clocks, the $\overline{RP}$ and $\overline{RDY}$ signals are only available at JP2 and JP3. The $\overline{RP}$ signal is generated in the HOTLink Transmitter and is used for reading data from asynchronous FIFOs, while the $\overline{RDY}$ signal is generated in the HOTLink Receiver and is used for writing data into asynchronous FIFOs. When interfacing to clocked FIFOs (CY7C44X, CY7C45X), the $\overline{RP}$ signal is not normally used. Because these signals are not present in JP4, they are not series terminated.

The other two output clocks (RCV_CLK0 and RCV_CLK1) are a buffered form of the recovered CKR clock from the receiver. The RCV_CLK1 signal is an inverted form of RCV_CLK0.

Data Signals

The CY9266 demo board has two data buses: one input (to the HOTLink Transmitter) and one output (from the HOTLink Receiver).

The input data bus consists of ten parallel transmit data signals that are sampled at the rising edge of the HOTLink Transmitter CKW clock. In addition to these ten signals, $\overline{ENN}$ and $\overline{ENA}$ (while part of the Switched Control signals) may also be considered part of the data bus as they are also sampled at this same time. While the XMIT_BISTEN input is also sampled at this same time, it is not normally used to transfer data and is therefore not considered part of the input data bus.

The output data bus is comprised of ten parallel received data signals that are synchronous to the HOTLink Receiver CKR clock. To meet specific timing requirements for OLC compatibility, there is also an external pipeline register between the HOTLink Receiver data bus output, and the received data bus connected to JP2, JP3, and JP4.

One other signal, BYTE_SYNC, is also clocked through this pipeline register and is thus considered part of the data bus.

All signals on this output bus are series-terminated with a 22Ω inline resistor to minimize transmission line ringing.

Configuration Settings

The CY9266 board may be user-configured to allow many modes of operation. This configuration is performed through the jumper block JP1 and the option select switch S1.

JP1 Jumper Block

The JP1 jumper block is used for configuring those options of the CY9266 that are (primarily) either to protect the board from signal contention, or for those signals having multiple sources and destinations. These functions are

- Receiver Mode Select
- Receiver Loopback Source Select
- Transmitter Mode Select
- Transmitter FOTO Source Select
- Transmitter Clock (CKW) Source Select
- Receiver Reference Clock (REFCLK) Source Select

JP1 exists as a 2 x 10 matrix of 0.025″ square pins on the top of the board. The rows in this matrix are identified on the top silk screen as A through J. The columns are identified as X and Y. A drawing of the JP1 jumper block is shown in *Figure 5*.

Receiver Mode Select

This jumper ties pins X and Y of JP1-A together. It is used to connect the receiver's MODE select pin to the option select switch (S1-6), and to allow the HOTLink Receiver mode to be set to the clock Test mode (see *Figure 13*). The three modes of receiver operation are

- Decode Mode — S1-6 ON (closed)
- Bypass Mode — S1-6 OFF (open)
- Test Mode — JP1-A, X and Y open

Because this clock Test mode is not normally used for communications testing, the jumper (JP1-A) is permanently wired in place with a foil trace on the bottom of the board. For those users who wish to actually place the receiver in Test mode, it may be necessary to cut this foil on the back of the board.

Once this foil has been cut, it will be necessary to use a shorting jumper across pins X and Y of JP1-A to

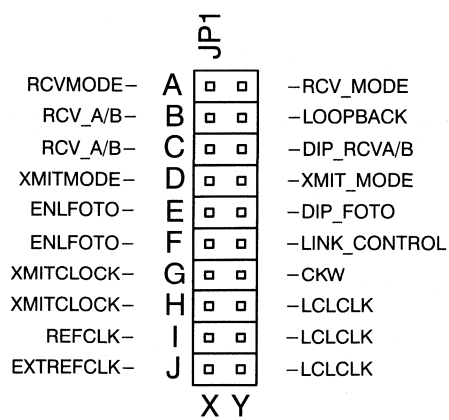


Figure 5. JP1, Top Side View

allow the two data modes of the receiver to be set by the option select switch (S1-6) and the RCV_MODE signal on JP2 and JP3.

Receiver Source Loopback Select

This function uses two positions (JP1-B and JP1-C) of the jumper block to select the source of the HOTLink Receiver loopback signal. Because this jumper is used to select between one of two sources, only one of these two positions (JP1-B or JP1-C) may contain a shorting jumper at any one time (see *Figures 10* and *11*).

By placing a shorting jumper across pins X and Y of JP1-B, the receiver loopback ($A/\overline{B}$) input is then controlled by the LOOP_BACK signal on JP2, JP3, and JP4. If this shorting jumper is moved to JP1-C, then the receiver loopback input is controlled by the option select switch (S1-7) and the RCV_MODE signal on JP2 and JP3. If a jumper is not present in either position, the INA± path is selected (external serial data).

Transmitter Mode Select

This jumper ties pins X and Y of JP1-D together. It is used to connect the transmitter MODE select pin to the option select switch, and to allow the HOTLink transmitter mode to be set to the clock Test mode (see *Figure 7*). The three modes of transmitter operation are

- Encode Mode — S1-2 ON (closed)
- Bypass Mode — S1-2 OFF (open)
- Test Mode — JP1-D, X and Y open

Because this clock Test mode is not expected to be used for normal data communications testing, the jumper (JP1-D) is permanently wired in place with a foil trace on the bottom of the board. For those users who wish to actually place the transmitter in Test mode, it may be necessary to cut this foil on the back of the board.

Once this foil has been cut, it will be necessary to use a jumper across JP1-D to allow the two data modes of the transmitter to be set by the option select switch (S1-2) and the XMIT_MODE signal on JP2 and JP3.

Transmitter FOTO Source Select

This function uses two positions (JP1-E and JP1-F) of the jumper block to select the source of the HOTLink Transmitter FOTO signal. Because this jumper is used to select from one of two sources, only one of these two positions (E or F) may contain a jumper at any one time (see *Figures 8* and *9*).

By placing a shorting jumper across pins X and Y of JP1-F, the HOTLink Transmitter FOTO signal is then controlled by the LINK_CONTROL signal on JP2, JP3, and JP4. If this shorting jumper is moved to JP1-E, then the transmitter FOTO signal is controlled by the option select switch (S1-8) and the DIP_FOTO signal on JP2 and JP3. If a jumper is not present in either position, the transmitter OUTA± and OUTB± differential drivers are disabled.

Transmitter Clock Source Select

The HOTLink Transmitter CKW clock can be sourced from two different signals: LCLCLK from the on-board oscillator and XMITCLOCK from JP2, JP3, and JP4 (see *Figure 7*).

To select the on-board oscillator, a shorting jumper should be placed across pins JP1-GY and JP1-HY. To select the XMITCLOCK signal, this shorting jumper should be moved to connect pins X and Y of JP1-G. To allow the transmitter to operate, it is necessary for a jumper to be in one (and only one) of these two positions.

Receiver Reference Clock Source Select

The HOTLink Receiver REFCLK signal can be sourced from three different signals: LCLCLK from the on-board oscillator, XMITCLOCK (from JP2, JP3, and JP4), and EXTREFCLK (from JP2 and JP3) (see *Figure 13*).

To select the on-board oscillator, a shorting jumper should be placed across the X and Y pins of JP1-I. To select the XMITCLOCK signal, this shorting jumper should be moved to connect pin X of JP1-I to pin X of JP1-H. To select the EXTREFCLK signal (used for PLL range testing), the shorting jumper should be placed across pin X of JP1-I and pin X of JP1-J. To allow the receiver to operate it is necessary for a jumper to be in one (and only one) of these three positions.

S1 Option Select Switch

The S1 Option Select Switch is used for configuring those options of the CY9266 that may be changed on a regular basis or are used to operate the board in a standalone mode. These functions are

- Transmitter BIST Enable
- Encoder Mode Select
- Enable Next Parallel Transmitter Data
- Enable Parallel Transmitter Data
- Receiver BIST Enable
- Receiver Mode Select
- Receiver A/B Port Select
- Transmitter FOTO Enable
- Carrier-Detect Polarity
- Byte Sync Polarity

S1 exists as a 10-position DIP switch. The switch positions (numbered 1 through 10) are identified on the top of the switch. When a switch is on (closed), the signal connected to that switch is tied directly to ground. When a switch is off (open), the signal on that switch is pulled up through a 5-kΩ resistor in R-pack R20.

These signals are also connected to pins on JP2 and JP3 to allow external logic to control these functions. A drawing of the S1 option select switch is shown in *Figure 6*.

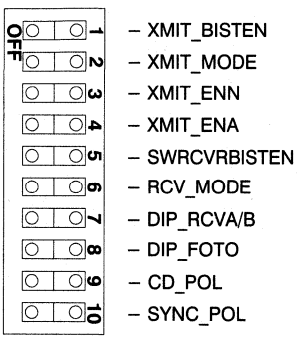


Figure 6. S1 Option Select Switch

Transmitter BIST Enable

Switch S1-1 (XMIT_BISTEN) is used to enable the HOTLink Transmitter BIST function. When this switch is on (closed), the $\overline{\text{BISTEN}}$ input to the transmitter is pulled LOW, placing the transmitter into its BIST loop. The exact patterns transmitted are determined by the levels on the XMIT_ENN and XMIT_ENA signals, located on S1-3 and S1-4 respectively (see *Figure 7*).

Encoder Mode Select

Switch S1-2 (XMIT_MODE) is used to select the data encoding mode of the HOTLink Transmitter. When this switch is on (closed), the internal 8B/10B encoder is enabled and the 8-bit data characters are encoded into 10-bit transmission characters. When this switch is off (open), the encoder is bypassed and the transmitter accepts 10-bit patterns for direct serialization (see *Figure 7*).

Enable Next Parallel Transmitter Data

Switch S1-3 (XMIT_ENN) is used, along with S1-1 (transmitter BIST enable) and S1-4 (XMIT_ENA), to select which data patterns are sent during HOTLink Transmitter BIST operations (see *Figure 7*).

If BIST is enabled (S1-1 on and S1-4 off), setting this switch off (open) causes the transmitter to send an alternating 1−0 pattern (D10.2 or D21.5). When turned on (closed), it enables an internal pattern generator in the transmitter that generates a repeating sequence of 511 10-bit patterns.

For normal data transfer operations this switch should remain off, with the XMIT_ENN signal controlled externally through JP2 and JP3.

Enable Parallel Transmitter Data

Switch S1-4 (XMIT_ENA) is used, along with S1-1 (transmitter BIST enable) and S1-3 (XMIT_ENN), to select which data patterns are sent by the HOTLink Transmitter during BIST operations (see *Figure 7*).

If BIST is enabled (S1-1 on and S1-3 off), setting S1-4 off (open) causes the transmitter to send an alternating 1−0 pattern (D10.2 or D21.5). When turned on (closed), it enables an internal pattern generator in the transmitter that produces a repeating sequence of 511 10-bit patterns.

For normal data transfer operations this switch should remain off, with the XMIT_ENA signal controlled externally through JP2 and JP3.

When operated from the JP4 system connector, this switch should be turned on (closed), because the system hardware is required to provide a valid 10-bit transmission character or data byte for each CKW clock.

Receiver BIST Enable

Switch S1-5 (SWRCVBISTEN) is used to enable the HOTLink Receiver BIST function (see *Figure 13*). When this switch is on (closed), the receiver awaits a D0.0 transmission character (sent once per BIST loop). When this character is detected the BIST state machine in the receiver begins matching the following received transmission characters with its internal pattern generator. This pattern generator follows the same sequence of patterns as those sent by the HOTLink Transmitter when sending its BIST sequence.

When this switch is off (open), the HOTLink Receiver operates in one of its two data modes (Decode or Bypass).

Receiver Mode Select

Switch S1-6 (RCV_MODE) is used to select the data decoding mode of the HOTLink Receiver (see *Figure 13*). When this switch is on (closed), the internal 10B/8B decoder is enabled and the received 10-bit transmission characters are decoded into 8-bit data characters. When this switch is off (open), the decoder is bypassed and the receiver outputs 10-bit transmission characters directly to the output data and status pins.

Receiver A/$\overline{B}$ Port Select

Switch S1-7 (DIP_RCVA/B) is used to select which input port (A or $\overline{B}$) the HOTLink Receiver should use for receiving serial data (see *Figures 10* and *11*). While the A/$\overline{B}$ input of the receiver is a 100K ECL compatible input, it is connected here to allow control from a switch or TTL driver. This requires use of an external resistor network, connected between that input and the select switch, to allow full rail-to-rail swings to be used.

When this switch is on (closed), the INB+ input to the HOTLink Receiver is selected. This input is directly connected to the OUTC+ output from the HOTLink Transmitter. This is the local loopback mode for the CY9266 demo board that allows the transmitter and receiver to be tested without an external serial data cable or optical module.

When this switch is off (open), the INA± differential input of the receiver is enabled to accept data from the optical module (U4) or copper cable.

Transmitter FOTO Enable

Switch S1-8 (DIP_FOTO) is used to enable the OUTA± and OUTB± differential output drivers of the HOTLink Transmitter. When this switch is on (closed), the differential outputs are allowed to follow the pattern of the data serialized by the transmitter (see *Figures 8* and *9*). When this switch is off (open), the OUTA± and OUTB± differential outputs of the transmitter are driven to a logic zero state (+ output is logic HIGH, − output is logic LOW). This places an attached optical transmitter in a state where no light is output, or presents no transitions on a copper cable.

Carrier-Detect Polarity

Switch S1-9 is used to control the active level of the carrier-detect output signal, LINK_STATUS. When this switch is on (closed) LINK_STATUS is driven HIGH when a carrier is present and LOW when one is not. When this switch is off (open) these levels are reversed (see *Figure 13*).

The carrier-detect status is also displayed on one of the decimal point indicators of the two-digit BIST display. When the indicator is on, a carrier is present. The state of S1-9 has no affect on the operation of this indicator.

Byte Sync Polarity

Switch S1-10 is used to control the active level of the BYTE_SYNC output signal. This level is also affected by the operating mode of the HOTLink Receiver (S1-6) (see *Figure 13*).

With the HOTLink Receiver in Bypass mode, the BYTE_SYNC signal is used as a K28.5 SYNC character indicator. With SYNC_POL LOW,

BYTE_SYNC is LOW when a K28.5 SYNC character is present on the receive data bus. With SYNC_POL HIGH, BYTE_SYNC is HIGH when a K28.5 SYNC character is present on the receive data bus.

With the receiver in Decode mode, the BYTE_SYNC signal is used as a valid data indicator. With SYNC_POL LOW, BYTE_SYNC is LOW whenever a usable data byte is present on the receive data bus. With SYNC_POL HIGH, BYTE_SYNC is HIGH whenever a usable data byte is present on the receive data bus.

CY9266 Schematic

The complete schematic for the CY9266-F demo board is shown in Appendix A, and the schematic for the CY9266-C and CY9266-T demo boards is shown in Appendix B.

Sheet 1 of the top-level schematic contains four functional blocks, which are detailed on the remaining pages of the schematic.

Sheet 2 contains the power-supply filtering and bypass capacitors. It also contains a sacrificial Zener diode that is used to protect the components on the board in case of over voltage or incorrect connection of the power supply.

Sheet 3 contains the BIST PLD and the error/status displays.

Sheet 4 of Appendix A contains the HOTLink Transmitter and Receiver, as well as the LED optical interface module. It also contains the on-board oscillator and option-select DIP switch.

Sheet 4 of Appendix B contains the HOTLink Transmitter and Receiver, as well as the copper interface and carrier-detect circuit. It also contains the on-board oscillator and option-select DIP switch.

Sheet 5 contains the parallel interface connectors, the voltage monitor/reset generator, and the OLC-compatibility registers.

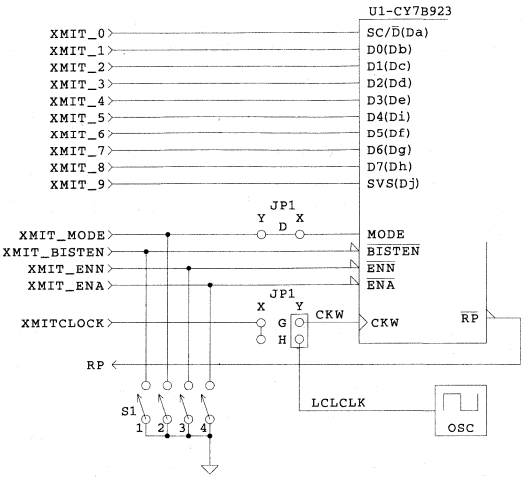


Figure 7. Transmitter Parallel Interface

Theory of Operation

The CY9266 demo board operation is broken down into five functional sections:

- Transmitter Parallel Interface
- Transmitter to Optical Module or Copper Serial Interface
- Optical Module or Copper to Receiver Serial Interface
- Receiver Parallel Interface
- BIST and Support Hardware

Transmitter Parallel Interface

The purpose of the transmitter parallel interface is to load parallel data from an external source and move that data to the shifter inside the transmitter. This portion of the design consists of three parts: the transmit data bus, transmitter control signals, and transmitter clocks. A simplified schematic of this interface is shown in *Figure 7.*

Transmit Data Bus

The transmit data bus is composed of the ten signals named XMIT_0 through XMIT_9. This bus may be driven from any of three possible sources: JP2, JP3,

or JP4. The data present on this bus is sampled by the HOTLink Transmitter (U1-CY7B923) at the rising edge of CKW.

The information present on the transmit data bus is interpreted by the HOTLink Transmitter in one of two ways, based on the setting of the MODE input to the transmitter. When MODE is HIGH (Bypass mode), all ten signals are accepted as the actual data to be transmitted and are fed directly to the shifter. The letter form (Da–Dj, as illustrated in *Figure 7*) of the bit identifiers is followed for this setting. These designators specify which encoded data bit is connected to a specific XMIT_0 to XMIT_9 signal. In this mode the user must encode the data into the 10-bit patterns used to send data across the serial interface. While it is not necessary to use the 8B/10B code described in the HOTLink datasheet, it is advised that this code be used for simplicity. If another code is used, its is the user's responsibility to insure that sufficient transitions are present in the serial data stream to allow the receiver to properly phase-lock to the serial data stream. For the HOTLink Receiver to provide byte framing and synchronization, the K28.5 pattern must be used for framing initialization.

When the MODE input is LOW (Encode mode), the internal 8B/10B encoder is enabled. In this mode the ten input bits are partitioned into eight data bits (D0–D7) and two data-modifier bits (SC/$\overline{D}$ and SVS). For transmitting normal data patterns, both the SVS and SC/$\overline{D}$ pins must be LOW. In this setting the 8-bit data character present on D0–D7 is latched at the rising edge of CKW and presented to the encoder. The encoder then converts the data character into the appropriate 10-bit transmission character. Following conversion, the transmission character is loaded into the shifter.

The two data-modifier bits, SC/$\overline{D}$ (Special Character/Data Select) and SVS (Send Violation Symbol), are used to send transmission characters other than those used to represent data. When the SC/$\overline{D}$ input is HIGH, the normal 8B/10B encoding of the data characters present on D0–D7 is changed. Now special control codes are generated (see listing in the CY7B923/CY7B933 datasheet). These control codes are used to send framing, control, status, and other supervisory functions across the interface.

The SVS pin is used for diagnostic purposes. When this input is HIGH, the HOTLink Transmitter shifter is loaded with a 10-bit pattern that is *not* a valid 8B/10B transmission character. When the HOTLink Receiver detects this encoding violation it responds with its RVS (Received Violation Symbol) output.

Note: The SVS input is intended for diagnostic purposes *only*. If used within normal message traffic, it may cause unexpected receive errors.

Transmitter Control Signals

In addition to the transmit data bus, four other signals are used to control the serial data stream generated by the HOTLink Transmitter. Two of these signals ($\overline{BISTEN}$ and MODE) control operating modes of the transmitter. The other two signals ($\overline{ENN}$ and $\overline{ENA}$) are used to specify when valid data is present on the transmit data bus.

Unlike the transmit data bus, these control signals are not connected to JP4, but are instead connected to JP2, JP3, and separate switches of S1. These switches allow the control inputs to be set LOW or HIGH when an external controller is not present. These switches are used both to control BIST mode for standalone applications and to set the proper operating characteristics for systems which only connect to JP4.

The $\overline{BISTEN}$ and MODE inputs are used to control which transmission characters are generated by the HOTLink Transmitter. Setting $\overline{BISTEN}$ LOW places the HOTLink Transmitter into one of two auto pattern-generation modes.

When $\overline{BISTEN}$ is LOW and both $\overline{ENN}$ and $\overline{ENA}$ are HIGH, the HOTLink Transmitter sends an alternating 1–0 pattern (D10.2 or D21.5). This pattern provides the highest baseband output frequency that the transmitter can generate, and is equal to 5x the frequency of CKW. This pattern may be useful to test or characterize various serial link components (i.e., fiberoptic modules, jitter tests, etc.).

When $\overline{BISTEN}$ is LOW and either $\overline{ENN}$ or $\overline{ENA}$ is also LOW, the HOTLink Transmitter begins a re-

peating test sequence that allows the transmitter and receiver to work together to test the functionality of the entire serial link. The repeating sequence is 511 characters in length and includes all standard codes as well as patterns that are normally considered code violations. This sequence may also be useful for performing serial link margin tests.

The MODE input pin is used to select both how the data on the transmit data bus is interpreted (encoded or non-encoded) and to place the HOTLink Transmitter into a clock test mode. This input is capable of selecting one of these three possible modes from a single pin by use of an internal three-level comparator. These modes are

- Encode Mode — S1-2 ON (closed)
- Bypass Mode — S1-2 OFF (open)
- Test Mode — JP1-D, X and Y open

When the MODE input is LOW (Encode mode), the internal 8B/10B encoder is enabled. This allows the transmit data bus to be interpreted as an 8-bit data bus (D0−D7) with two control bits (SC/$\overline{D}$ and SVS). When the MODE input is HIGH (Bypass mode), the internal encoder is bypassed. This allows the data bus to be interpreted as a 10-bit bus (Da−Dj). Either of these modes may be set from JP2, JP3, or S1-2.

The clock Test mode is accessed by allowing the MODE input pin to float. Through use of an internal bias network in the transmitter, the MODE input pin is placed at $V_{CC}/2$. This clock Test mode can be accessed two ways on the board. The easiest is to cut the foil on the bottom of the board that shorts the X and Y pins of JP1-D together. Once cut it will be necessary to place a shorting jumper across these pins to allow JP2, JP3, or S1 to place the transmitter into one of its normal data modes.

The other method of accessing this mode is to actively bias the XMIT_MODE pin on JP2 or JP3 to $V_{CC}/2$. When doing so, keep in mind that this input also has a 5-kΩ pull-up resistor attached to this signal.

The $\overline{ENN}$ (ENable Next parallel data) and $\overline{ENA}$ (ENAble parallel data) inputs are normally used to specify when valid data is present on the transmit data bus. Both of these inputs are sampled on the rising edge of CKW at the same time as the 10-bit transmit data bus.

If $\overline{ENA}$ is LOW and $\overline{ENN}$ is HIGH at the rising edge of CKW, the data present on the transmit data bus is loaded, processed, and sent to the shifter. If both $\overline{ENA}$ and $\overline{ENN}$ are HIGH at the rising edge of CKW, the latched data is ignored and a K28.5 SYNC code is sent in its place.

If $\overline{ENN}$ is LOW and $\overline{ENA}$ is HIGH at the rising edge of CKW, the data present on the transmit data bus at the **next** rising edge of CKW is loaded, processed, and sent to the shifter. If both $\overline{ENN}$ and $\overline{ENA}$ are HIGH at the rising edge of CKW, the data latched on the next rising edge of CKW is ignored and a K28.5 SYNC code is sent in its place.

These two enable control signals are used to allow different hardware interfaces to be implemented with the least amount (usually none) of additional data pipelining hardware. When one of these enable inputs is used for enable control, the other is usually tied HIGH, but may be used in conjunction with BISTEN for link testing without affecting the data path controller.

Transmitter Clocks

The transmitter interface operates with both an input clock (CKW) and an output clock ($\overline{RP}$). The input clock is used to generate both the internal shifter clock and the output clock.

The CKW input clock can be sourced from either the on-board oscillator or from the XMITCLOCK signal. This selection is made through jumper block JP1.

All internal operations of the HOTLink Transmitter are based on the rising edge of the CKW clock. The CKW clock must be generated from a crystal-based source. While the duty cycle of the CKW clock source is relatively unimportant, it must still meet certain minimum pulse-width times as listed in the CY7B923/CY7B933 datasheet.

The $\overline{RP}$ output clock pulse is a modified duty cycle pulse whose HIGH and LOW components are set for operation with asynchronous FIFOs (CY7C42X family). The phase relationship of this clock pulse

to CKW, and its duty cycle (both set by the internal PLL), are positioned to have valid data on the transmit data bus at the rising edge of CKW.

This $\overline{RP}$ clock pulse may be directly connected to the read control pin ($\overline{R}$) of an attached FIFO. Because the presence of this pulse signifies a FIFO read operation, it is only generated in response to the $\overline{ENA}$ input being pulled LOW.

Transmitter to Optical Module Serial Interface

The transmitter has three differential output pairs that each output the same serial data stream from the shifter. Because of the switching speeds used for these serial outputs (and for compatibility with optical interface modules) they are all implemented using Positive referenced 100K ECL (Emitter Coupled Logic) compatible drivers. A simplified schematic of the interface present on the CY9266-F is shown in *Figure 8*.

The normal mode of ECL operation is for all signaling to be done at voltages below ground. Because the ground point for ECL is only a reference, the same signaling can also be implemented above ground. When this is done the reference point changes from ground to V_{CC}. When operated in this mode ECL is often referred to as PECL (Psuedo-ECL or Positive-ECL). This is the mode of operation for the serial outputs on the transmitter.

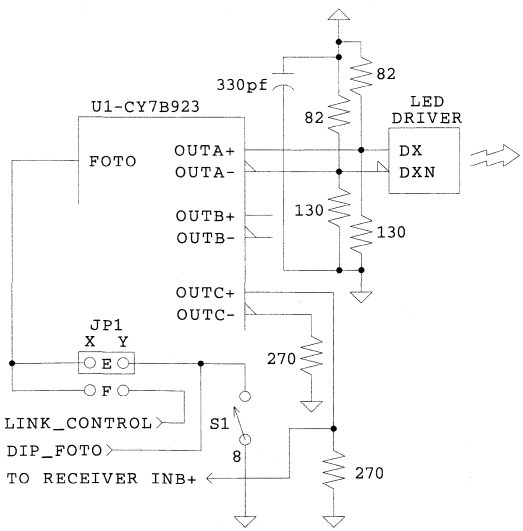


Figure 8. HOTLink Transmitter-to-Optical Serial Interface

Two of the differential outputs (OUTA± and OUTB±) are also controlled by a TTL-level enable pin called FOTO (FiberOptic Transmitter-Off). This control input is used to disable all light output from the optical module. While not specifically necessary for LED-based optical modules, the ability to disable all light output is a safety requirement for all laser-based links (ANSI Z136.1 and Z136.2, F.D.A. regulation 21 CFR subchapter J, and IEC 825). When FOTO is HIGH, the OUTA± and OUTB± differential pairs are forced to a logic 0 state (OUT+ is LOW and OUT− is HIGH). When FOTO is LOW, the OUTA± and OUTB± differential outputs are allowed to follow the serial data pattern from the shifter.

The FOTO pin on the HOTLink Transmitter may be configured to be controlled from either the JP2, JP3, or JP4 connectors (LINK_CONTROL) or from S1-8 (DIP_FOTO). To avoid possible signal contention from these sources, this signal is first run through jumper block JP1.

Placing a shorting jumper across the X and Y pins of JP1-F allows the receiver port selection to be controlled from the LINK_CONTROL signal. Moving this jumper to JP1-E allows this selection to be made through S1-8 or through the DIP_FOTO signal on JP2 and JP3. If the jumper is omitted from the board, the OUTA± and OUTB± outputs are placed in the disabled state.

The OUTC± differential output is not controlled by FOTO. This output continues to follow the serial shifter data at all times. Because it is never disabled, this signal is used for the local loopback. While this signal is available differentially, it is connected to the receiver single-ended. This allows the INB− input on the receiver to be used as an ECL-to-TTL translator for the receive optical module's carrier-detect signal.

Because ECL signals are only *active* in one direction, it is necessary to provide a bias/load network of some type for the signals to properly switch. The

typically specified load for ECL signals is 50Ω connected to V_{CC} – 2V (i.e., +3V for PECL).

This type of load can be created in many ways. For large ECL systems a separate power supply is usually present to generate this termination voltage. This provides the lowest power dissipation. For small systems (like this one), a simpler method is to use two resistors to create a network whose Thevenin equivalent is this same 50Ω connected to V_{CC} – 2V. This is used for the A± differential pair. The capacitor present across the Thevenin pair is necessary to produce an AC short between the power and ground planes.

The OUTB± output pair is not used on this demo board. While normal ECL drivers left in this mode would still dissipate a significant amount of power, the HOTLink ECL outputs contain additional internal structures to sense if an output is used or left open, and disables the internal current sources of unused output drivers. This results in a current savings of approximately 5 mA (25 mW) for each unused output pair.

The OUTC± output pair is biased to V_{CC} – 5V (ground) through 270Ω resistors. This bias arrangement is used here to reduce the overall component count. This type of load may be used for short connections because it provides a similar current load to a Thevenin termination but, due to asymmetric rise and fall times, it induces more jitter into the data. This type of biasing should not be considered as a type of line termination. If the switching speeds and length of circuit traces dictate that the line should be terminated, a Thevenin bias network should be used to match the line impedance.

Even in those cases where the connection to the optical modules is short and a 270Ω resistor to V_{CC} – 5V may seem to be usable, it should not be used. While this type of connection may work for very short optical cable lengths, the jitter introduced by the bias network reduces the overall system jitter margin.

Transmitter to Copper Cable Serial Interface

On the CY9266-C and CY9266-T boards, the transmitter output is configured to drive either a coaxial or shielded-pair cable. A simplified schematic of this interface is shown in *Figure 9*.

The copper-based CY9266-C and CY9266-T boards use a transformer-coupled interface. Transformer coupling is called out in the ANSI Fibre Channel standard for copper-based interfaces. Its primary advantages are excellent common mode rejection, balanced-to-unbalanced conversion (for coaxial cables), and DC isolation (2 kV hi-pot tested).

The CY9266-C and CY9266-T boards are designed to allow other modes of line biasing and coupling to be used for presenting a signal into the cable. Pads are present on the board to allow a Thevenin bias to be used on OUTA±. These resistors are identified as R72 and R73 on Sheet 4 of the CY9266-C/T schematic (see Appendix B).

The CY9266-C and CY9266-T are designed to operate with cable systems providing a reflection coefficient of zero. This means that the receiving end of the cable should be terminated in the characteristic impedance of the cable.

Pads are also present to allow both source termination and AC coupling to the transformer. These components are identified as R54, R55, C25, and C26 on Sheet 4 of the CY9266-C/T schematic (see Appendix B). To use parts in these locations it is

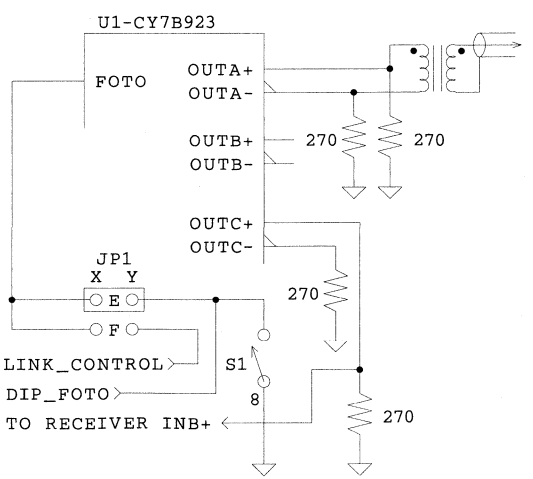


Figure 9. HOTLink Transmitter to Copper Serial Interface

necessary to remove the foil shorts across these component pads on the circuit board.

The control signal inputs for copper-based interfaces operate identically to those of the optical interface. The difference in operation is that when the OUTA± outputs are disabled through the use of the FOTO signal, instead of disabling all light, all output transitions are disabled.

Optical Module to Receiver Serial Interface

The HOTLink Receiver has two differential input pairs (INA± and INB±) that can both be used to receive the high-speed serial data streams generated directly by the transmitter or as output from an optical receiver. These serial inputs are also PECL and are compatible with the HOTLink Transmitter. ECL was chosen for these signals for the same reasons (speed, low noise, compatibility with optical modules) it was used for the transmitter.

A separate PECL input signal (A/$\overline{B}$) is used to select which input pair (INA± or INB±) is actually fed to the receiver shifter and PLL. A simplified schematic of the optical module-to-receiver serial interface on the CY9266-F is shown in *Figure 10*.

Optical Module Signals

The optical receiver generates two signals, a 100K PECL differential received data signal, and a single-ended carrier-detect signal. While the DIP package form of the optical module does provide both + and − forms of the carrier-detect signal, only the + form is available on the endfire package. To allow the same circuitry to be used with either module type, only the + carrier-detect signal is used.

Receiver Data Inputs

The HOTLink Receiver differential A and B inputs are similar, but not identical. While the INA± inputs must always operate as a differential pair, the INB± signals do not. This allows the INB± inputs to be split into two separate ECL inputs: INB+, which feeds the shifter and PLL, and INB−, which feeds an ECL-to-TTL translator.

The configuration of the INB± inputs is controlled by the SO output of the translator. While technically an output, the SO pin on the HOTLink Receiver also contains sense circuits that monitor the voltage level on the pin during power-up. If the SO output is connected to V_{CC}, the INB− input becomes part of the INB± differential serial input. If the SO output is normally loaded (no resistive pull-up to V_{CC}), the INB+ input becomes a single-ended serial data receiver and the INB− input becomes part of a PECL-to-TTL translator.

This split mode is used on the CY9266 demo board. It allows the INB− input to be used to convert the PECL carrier-detect output of the optical module (SIGO) to the TTL-level signal needed on the receiver parallel interface.

Receiver Port Select

The HOTLink Receiver uses a single-ended PECL input (A/$\overline{B}$) to control which serial input is fed to the shifter and PLL. When the A/$\overline{B}$ input is HIGH, the differential INA± pair is connected to the shifter and PLL. When the A/$\overline{B}$ input is LOW, the INB+ input is fed to the shifter and PLL. Because the INB+ input is directly connected to the OUTC+

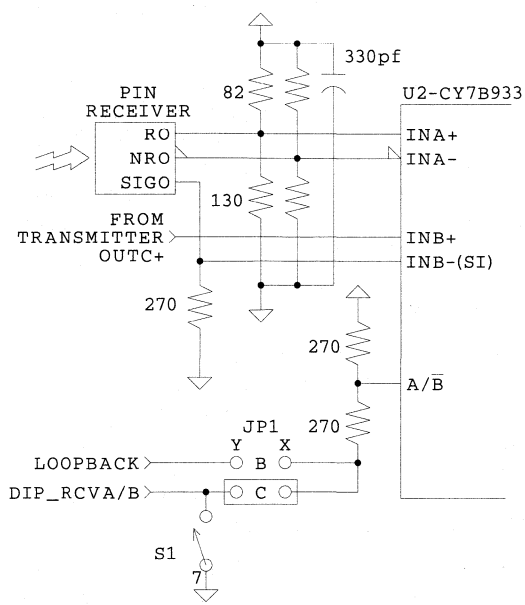


Figure 10. Optical-to-HOTLink Receiver Serial Interface

output from the HOTLink Transmitter, this LOW setting is used for a local loopback and allows the transmitter and receiver to communicate without using an optical module.

The A/$\overline{B}$ input is a PECL input and normal TTL or CMOS logic swings will not work to control it. This input uses PECL (or larger) signal swings. These can still be achieved in a TTL environment through use of a resistive divider network.

Using this network, a TTL LOW level on the input to the divider creates a PECL LOW at the A/$\overline{B}$ input to the receiver. With a TTL (or CMOS) HIGH into the divider, the A/$\overline{B}$ input is placed at (or above) a PECL HIGH. While standard 100K ECL inputs should never be taken above V_{CC} – 700 mV, the ECL inputs on the HOTLink Receiver may be connected directly to V_{CC} without degradation or damage.

The divider network on this demo board may be configured to be controlled from either the JP4 connector (LOOP_BACK) or from S1-7 (DIP_RCVA/B). To avoid possible signal contention from these sources the signal is first run through jumper block JP1.

Placing a shorting jumper across the X and Y pins of JP1-B allows the receiver port selection to be controlled from the LOOP_BACK signal. Moving this jumper to JP1-C allows this selection to be made through S1-7 or through the DIP_RCVA/B signal on JP2 and JP3. If the jumper is left off the board, the A± pair is selected.

Copper to Receiver Serial Interface

The CY9266-C and CY9266-T Demo Boards replace the optical module with a transformer coupled electrical interface. The transformer used here provides the same functionality as the one used at the transmit end of the cable. A simplified schematic of the copper cable-to-receiver serial interface on the CY9266-C/T is shown in *Figure 11*.

The output side of the transformer connects to two resistors. These resistors provide the line termination for the transmission line connected to the transformer. Two resistors are used to allow a reference

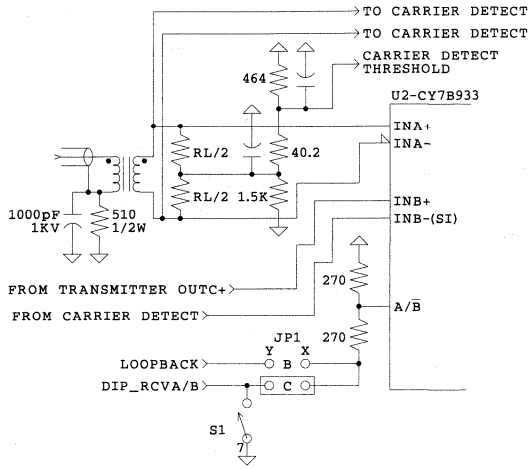


Figure 11. Copper-to-HOTLink Receiver Serial Interface

voltage to be set for the center of the received signal. This reference point is set by an external 3-resistor divider, and is set in this circuit to V_{CC} – 1.3V. This is near the center of the common mode range of the MC10H116 ECL receiver that is used to build a carrier detection circuit. If this carrier-detect circuit is not used, it would be better to bias this point at V_{CC} – 1.5V, the center of the HOTLink Receiver's common mode range.

Both of these reference points must be bypassed to allow them to remain stable under dynamic signal conditions.

Unlike the optical receiver, which outputs a hard zero in the absence of light (INA+ = 0, INA– = 1), the AC-coupled interface used for copper connections does not. When the signal is removed, the INA+ and INA– inputs to the HOTLink Receiver are set to the same voltage. Because of the high gain present in the HOTLink Receiver, to allow use with long cables (low amplitude received data), the HOTLink Receiver will probably oscillate. This oscillation under a no-signal condition can be corrected by forcing an offset between the INA+ and INA– inputs, but this offset will induce more jitter into the data stream and limit the usable length of a copper-based serial link. Rather than compromise operational length, a carrier detection circuit

can be added to validate the received data (in addition to the validation mechanisms present in the data itself).

The CY9266-C and CY9266-T boards also contain the pads and routing necessary for implementing an equalizer to allow longer cables to be used. The function of an equalizer is to present a frequency selective attenuation to the received signal that brings the amplitude and phase of the frequency components in that signal into the same amplitude and phase. Because signals transmitted over copper cables are effectively run through a high-frequency attenuator, the equalizer used for copper cables is a form of low-frequency attenuator (high-pass filter).

The equalizer used is implemented in a bridged-H configuration that is designed for balanced line operation. It is shown on Sheet 4 of the CY9266-C schematic in Appendix B and is constructed using R64, R65, R66, R67, R68, R69, R70, R71, C29, C30, and L1. To implement this equalizer it is necessary to remove the foil shorts across R64 and R71.

Copper Carrier-Detect

The carrier-detect circuit used on the CY9266-C and CY9266-T boards is shown in *Figure 12*. This circuit uses two ECL differential receivers as level comparators to detect the presence of 1- and 0-level pulses on the incoming signal. The gate connected to the top side of the transformer (shown in *Figure 11*) detects the presence of received 1 pulses while the gate connected to the bottom of this transformer detects the presence of received 0 pulses. The input capacitance of these two comparators is isolated from the actual received signal through two 100Ω resistors to prevent this additional load from distorting the received signal.

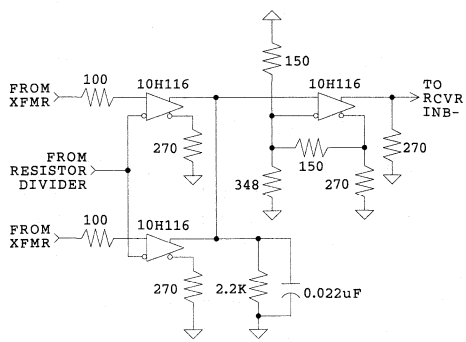


Figure 12. Copper Interface Carrier-Detect

The input signal amplitude necessary to detect either a 1 or a 0 is set by the resistor divider shown in *Figure 11*. To prevent the 10H116 gate from oscillating it is recommended that this threshold be set to a minimum of 50 mV above the termination reference voltage.

The outputs of these two gates are then wire-ORed together to charge a capacitor. Because of the low on resistance of the emitter follower output transistors of the 10H116 gates, the capacitor can be charged quite quickly. In the absence of 1 or 0 transitions above the set threshold level, this capacitor is discharged both by a bleeder resistor to V_{EE}, and through the input of the third gate.

The third gate is configured as a comparator with feedback to form a Schmitt trigger. This feedback is necessary because of the slow transition rate of the input signal to this gate. If feedback was not used, this gate would oscillate as the input signal slowly passes through the the threshold region of the gate. The output of this Schmitt trigger is then connected to the HOTLink Receiver INB− input, which is configured as a PECL-to-TTL translator.

Receiver Parallel Interface

The receiver parallel interface is used to move the character framed in the HOTLink Receiver to the external world where it can be used. This portion of the design consists of five sections: receiver parallel data output, OLC-compatibility registers, receiver clocks, receiver control inputs, and receiver status outputs. A simplified schematic of this interface is shown in *Figure 13*.

Receiver Parallel Data Output

The receiver data bus is composed of ten signals named REC_0 through REC_9. This bus drives all three I/O connectors (JP2, JP3, and JP4). Due to the external register in the data path, these outputs change coincidental with the rising edge of RCV_CLK0 (CKR).

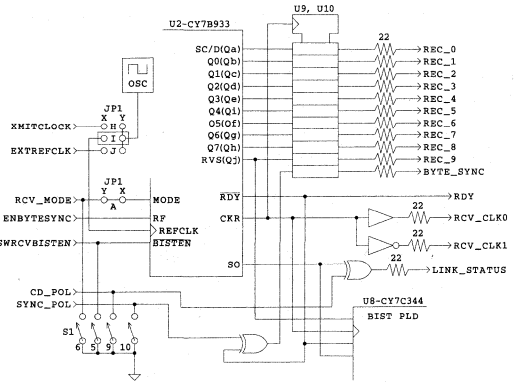


Figure 13. Receiver Parallel Interface

The information placed on the receiver data bus is determined by the HOTLink Receiver MODE select pin. When MODE is HIGH (Bypass mode), all ten outputs are the ten bits that were received and framed. The letter form (Qa–Qj, as illustrated in *Figure 13*) of the bit identifiers is followed for this setting. These designators specify which encoded data bit is connected to a specific REC_0 to REC_9 signal. In this mode the user must decode the data from the 10-bit patterns used to send the data across the serial interface.

While it is not necessary to use the 8B/10B code described in the HOTLink datasheet, it is advised that this code be used for simplicity. If another code is used, it is the user's responsibility to insure that sufficient transitions are present in the data stream to allow the HOTLink Receiver to properly phase-lock to the serial data stream.

For the HOTLink Receiver to maintain byte framing and synchronization, the K28.5 pattern must also be used for framing initialization. For those systems that perform their own framing (SONET, etc.), the HOTLink Receiver will phase-lock to a serial data stream without a K28.5 code present and clock out a character every 10 bit-clocks. These systems must operate in Bypass mode as the HOTLink Receiver decoder requires operation with the 8B/10B code and must acquire byte sync to recover valid data. These systems must provide external byte framing.

When the HOTLink Receiver MODE input is LOW, the internal 10B/8B decoder is enabled. In this mode, the ten output bits from the shifter are sent to the decode register once every ten bit-clocks, as determined by the framer. The 8-bit output from this decoder is then placed on the receiver output data bus bits Q0–Q7, along with the two data status bits SC/$\overline{D}$ and RVS.

When receiving normal data patterns both the RVS and SC/$\overline{D}$ pins are LOW. In this setting, the 8-bit data character present on Q0–Q7 is latched at the rising edge of CKR into the external register and presented to the output of the board.

The two status bits, SC/$\overline{D}$ (Special Character/Data Select) and RVS (Received Violation Symbol), are used to indicate reception of characters other than those used to represent data. When the SC/$\overline{D}$ output is HIGH, special control codes (see listing in the CY7B923/CY7B933 datasheet) have been decoded. These control codes are used to indicate framing, control, status, and other supervisory functions across the interface.

The RVS pin is used for diagnostic purposes. When this output is HIGH, the HOTLink Receiver decoder has detected a 10-bit pattern that is **not** a valid 8B/10B transmission character or sequence. When the receiver detects this encoding violation, it asserts RVS and places information on the Q0–Q7 outputs to represent the type of error detected. Because all of these errors are represented with Special Codes (C0.7, C1.7, C2.7, and C4.7) the SC/$\overline{D}$ output is always HIGH whenever RVS is HIGH. These possible error-type codes are listed in the HOTLink datasheet.

OLC-Compatibility Registers

In order for this demo board to operate in an OLC-266 compatible system, the timing of the $\overline{RDY}$ signal had to be modified. This signal from the receiver is used for four functions: to indicate when a K28.5 SYNC character has been received, to indicate that valid data has been received, to clock valid data into an external asynchronous FIFO, and to indicate the end of a BIST loop.

To support these different functions from a single pin requires the addition of a single register to convert the waveform generated by the $\overline{RDY}$ signal into the BYTE_SYNC status signal the OLC card generates. Additional registers were then added to the data bits to keep them in the same byte-phase relationship as the BYTE_SYNC signal (which is now delayed one clock).

The 22Ω series termination present on these signals should not be necessary for most systems, but are added here to allow a flat-cable-type attachment to this card.

Figure 14 shows the relative timing relationships between the receiver data, the $\overline{RDY}$ signal, the BYTE_SYNC signal, and the output clocks. For $\overline{RDY}$ to operate in this fashion, the RF (Reframe enable) control input must be HIGH and the receiver must be in Bypass mode (receiver MODE is HIGH).

When RF is LOW, the $\overline{RDY}$ and BYTE_SYNC outputs operate the same as that shown in *Figure 14*. The difference is that the clocks are not allowed to change phase or width upon detection of a K28.5 SYNC character.

The functionality of the $\overline{RDY}$ (and thus BYTE_SYNC) signal changes when the receiver is in Decode mode (receiver MODE is LOW). Here the the $\overline{RDY}$ signal pulses LOW for every character received *including* the K28.5 SYNC character. When multiple consecutive SYNC characters are received, $\overline{RDY}$ is inhibited except for the last K28.5 character received. This is done to prevent overfilling a receiver FIFO with non-data information. *Figure 15* shows the relative timing relationships for this type of operation.

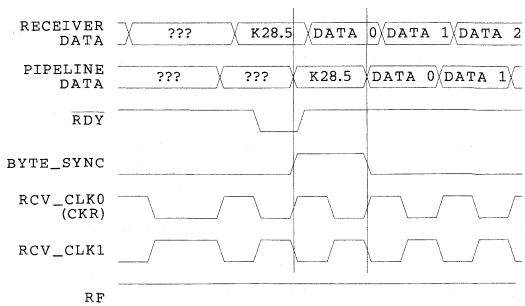


Figure 14. Receiver Data Timing, Bypass Mode, RF HIGH

Because RF is LOW in *Figure 15*, the CKR clock (and thus RCV_CLK0 and RCV_CLK1) is not allowed to reframe on new K28.5 SYNC characters detected. When RF is HIGH in Decode mode, the HOTLink Receiver $\overline{RDY}$ output ceases pulsing until the first K28.5 SYNC code is detected, after which the behavior illustrated in *Figure 15* is resumed.

Receiver Clocks

The HOTLink Receiver parallel interface (see *Figure 13*) operates with a single input clock (REFCLK) and two output clocks (CKR and $\overline{RDY}$).

The REFCLK input clock does not directly clock anything in the receiver, but is used as a reference for the receiver PLL. This clock is required to be both stable and reasonably accurate. It must match the byte-rate frequency of the received data within ±0.1%. Unlike an OLC card, which requires a special sequencing of the LOCK_TO_REF signal to allow the receiver to track to a reference clock, the HOTLink Receiver PLL continuously operates in a mode that compares its frequency to that of the reference clock, even when valid data is being received.

If the frequency of the received data varies outside of specific fixed limits, the HOTLink Receiver stops locking to the serial data and reverts to the REFCLK. Once the received serial data stream returns to an acceptable frequency, the PLL again locks to the received data. Since it is likely that byte sync has been lost, a reframe cycle should be done

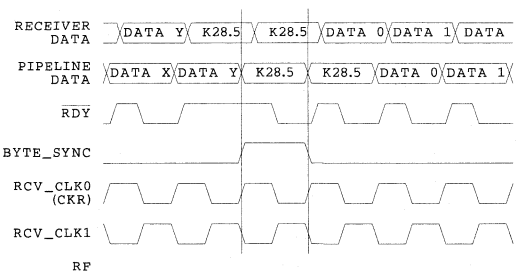


Figure 15. Receiver Data Timing, Decode Mode, RF LOW

to allow the framer to lock up again. Detection of this and the recovery process is normally handled automatically by higher-level functions in the communications system.

The REFCLK input to the receiver can be sourced from three different signals on the demo board: the on-board oscillator, the XMITCLOCK input, or the EXTREFCLK input. Selection of the clock source can only be done through jumper block JP1.

The on-board oscillator is used primarily for standalone operation and testing using the BIST capabilities of the HOTLink parts. This clock is selected by placing a shorting jumper across pins X and Y of JP1-I.

The XMITCLOCK input is used for normal data transmit/receive functions and for OLC-compatibility mode. This clock is selected by placing a shorting jumper across pins JP1-HX and JP1-IX.

The EXTREFCLK input is used for those instances when the transmitter and receiver are to be clocked with different frequency clocks. This is expected to be used only to test for PLL capture/lock range testing of the receiver, or when the HOTLink Receiver is connected to a transmitter operating at a different frequency from the local HOTLink Transmitter. This clock is selected by placing a shorting jumper across pins JP1-JX and JP1-IX.

The CKR output clock is generated in the HOTLink Receiver and is based directly on the internal PLL frequency. This output is synchronous with the receiver output data bus and may be used to clock the data into an associated register (as is done on this board) or into synchronous FIFOs.

The period and duty cycle of the CKR output clock are fixed by the logic in the receiver. To achieve compatibility with OLC-type systems, the CKR signal is used to generate two new clock signals (RCV_CLK0 and RCV_CLK1) that are true and complement copies of the CKR clock. To keep matched delays and to minimize the number of additional logic packages on the board, these two clocks are generated using XOR gates.

When framing occurs, the CKR clock can experience large phase changes. These changes are exhibited by a lengthening of either the HIGH or LOW portion of the CKR waveform. This can be seen in the waveforms shown in *Figure 14*. While this functionality is not required by the ANSI Fibre Channel Standard, it is included in the HOTLink Receiver to protect downstream clocked logic from the narrow pulses or glitches that can occur otherwise.

The $\overline{\text{RDY}}$ output signal is used both as a status output and as a clock. Its use as a clock is primarily for clocking data present on the receiver data bus outputs into asynchronous FIFOs. The duty cycle of the $\overline{\text{RDY}}$ pulse and its position relative to the output data is such that it may be directly connected to the $\overline{\text{W}}$ (write) input on CY7C42X FIFOs.

Receiver Control Inputs

The receiver parallel interface is controlled by three input signals: RF (Reframe), MODE (Receiver Mode select), and $\overline{\text{BISTEN}}$ (BIST Enable).

The RF input is used to select when the HOTLink Receiver is allowed to reframe (acquire byte-sync) to the incoming serial data stream. This input is present to prevent the receiver from mis-framing on aliased K28.5 SYNC codes, which would cause long running decode errors.

When RF is LOW the framer is disabled; it does not change the starting bit location of each received character. Any received K28.5 SYNC code is treated as normal data and is clocked out with the CKR and $\overline{\text{RDY}}$ clocks. If this SYNC code is received across two character boundaries, the framer does not reframe. If the HOTLink Receiver is operating in Decode mode, the existence of such a pattern may generate one or more characters in error.

When RF rises, the $\overline{\text{RDY}}$ output is inhibited. With RF held HIGH, the framer continuously monitors the serial data stream for either disparity form of the K28.5 SYNC character. When this character is detected, the bit counter used to count off serial data bits and specify received character boundaries is asynchronously reset to properly frame the following received bits on character boundaries.

If the receiver is set to Decode mode, the $\overline{\text{RDY}}$ output assumes its normal function of pulsing LOW for each byte after the first K28.5 SYNC code is detected. If the receiver is instead set to Bypass mode,

the $\overline{RDY}$ signal pulses LOW only for the SYNC (K28.5) characters while RF is HIGH or LOW.

Because of characteristics of the 8B/10B code, it is possible to transmit legal character sequences that can cause incorrect framing (this requires sending control codes other than K28.5). These codes should be avoided while RF is HIGH. Once the framer is disabled (RF LOW) these sequences may be used to pass control information across the interface without causing the receiver to incorrectly frame the data that follows.

The MODE input pin on the HOTLink Receiver is used to select both how the received serial data is to be presented on the data bus (encoded 10-bit character or decoded 8-bit character), and to place the receiver into a clock Test mode. This input is capable of selecting one of these three possible modes from a single pin by use of an internal three-level comparator.

When the MODE input is LOW, the internal 10B/8B decoder is enabled (Decode mode). This allows the receiver output data bus to be interpreted as an 8-bit data bus (Q0−Q7) with two status bits (SC/$\overline{D}$ and RVS). When the MODE input is HIGH, the internal decoder is bypassed (Bypass mode). This allows the data bus to be interpreted as a 10-bit bus (Qa−Qj). Either of these modes may be set from JP2, JP3, or S1-6.

The clock Test mode is accessed by allowing the MODE input pin to float. Through use of an internal bias network in the receiver, the MODE input pin is placed at $V_{CC}/2$. This clock Test mode can be accessed two ways on the board. The easiest is to cut the foil on the bottom of the board that shorts the X and Y pins of JP1-A together. Following this, it will be necessary to place a shorting jumper across these pins to allow JP2, JP3, or S1-6 to place the receiver into one of its normal data modes.

The other method of accessing this mode is to actively bias the RCV_MODE pin on JP2 or JP3 to $V_{CC}/2$. When doing so, keep in mind that this input also has a 5-kΩ pull-up resistor attached to the signal.

The $\overline{BISTEN}$ input pin is used to place the HOTLink Receiver in a special pattern verification mode. This mode is designed to work in conjunction with a matching pattern generation mode in the transmitter. While not shown on the schematic in *Figure 13*, the $\overline{BISTEN}$ input is actually run through the BIST PLD (U8-CY7C344). This is not necessary but is done here to allow other conditioning of the $\overline{BISTEN}$ signal if desired.

When the HOTLink Receiver $\overline{BISTEN}$ input is set LOW, the receiver's BIST state machine is enabled and enters its self-test mode. At this point it sets $\overline{RDY}$ HIGH and begins looking for the BIST start-of-loop character (D0.0) in the serial data stream. Once this character is detected, the $\overline{RDY}$ output is driven LOW, where it remains until the end of the 511-character BIST loop. At this point $\overline{RDY}$ pulses HIGH for one character and starts the next 511-byte loop.

While the BIST test mode is enabled, the RVS output is used to indicate that a pattern mismatch has occurred. This means that the 10-bit pattern received did not *exactly* match the 10-bit pattern that was expected (expected code violations are not errors).

Receiver Status Outputs

The HOTLink Receiver parallel interface generates two status output signals: $\overline{RDY}$ and SO.

The $\overline{RDY}$ output is used both for status information and as a clock. As a status output, its information is valid at the rising edge of CKR. This means that the $\overline{RDY}$ signal must be registered to present its status information. For normal data transfer modes, the registered form of $\overline{RDY}$ is used to identify the presence of multiple K28.5 SYNC characters (HIGH at rising edge of CKR) and of data or control characters (LOW at the rising edge of CKR). This registered form of $\overline{RDY}$ generates the BYTE_SYNC signal.

The $\overline{RDY}$ signal is also used to identify what phase the HOTLink Receiver BIST mode is in. When HIGH for two or more CKR clocks, the receiver is looking for the start character of the BIST loop. When LOW, the receiver is in the BIST loop. When HIGH for a single clock, the receiver has completed another BIST loop.

The SO output is used as part of an ECL-to-TTL translator to specify the current state of the carrier on the serial interface, and is used to drive the LINK_STATUS signal. When a valid carrier is present and S1-9 (CD_POL) is off (open), LINK_STATUS is LOW. This polarity is reversed by turning S1-9 on (closed) or pulling SYNC_POL LOW.

BIST and Support Hardware

The CY9266 Demo Board contains not only those components necessary to form a serial link, but also a few support components to enhance OLC compatibility and to support the BIST capability in the HOTLink Transmitter and Receiver. A simplified schematic of these additional components is shown in *Figure 16*.

The MAX707 is used to monitor the power-supply voltage and remove the RESET signal when V_{CC} is above 4.65V. This is a close approximation to the 4.75V RESET threshold specified for the OLC card. This part also supports an external mechanical switch input that also controls the RESET output. This input is controlled by the BIST reset push-button switch (S2). When this switch is depressed, the RESET output is driven LOW until 200 ms after the switch is released. This RESET signal is used to clear the BIST error-counter located in the BIST PLD (U8). The PWR ON indicator is extinguished as long as RESET is active.

The BIST PLD is a Cypress CY7C344 MAX EPLD programmed with the counters and state machines necessary to monitor the status of the receiver outputs and count when receive errors are detected. This PLD also drives the decimal points on the attached displays to indicate four status signals. These status signals are

- PWR ON – Lit when power is present and above the 4.65V sense threshold
- CAR DET – Lit when a valid carrier is present
- BIST WAIT – Lit when BIST is enabled but the receiver has not detected the start of the BIST loop
- BIST OVFL – Lit when the BIST error count exceeds 99

BIST State Machine

The BIST state machine has six states that control when a counter is enabled to count pattern-match errors. A bubble diagram of this state machine is shown in *Figure 17* while the MAX+PLUS source file for this state machine is listed in Appendix C.

This state machine controls when the error counter is enabled to count. It operates off of two input signals: BISTEN and $\overline{RDY}$. Whenever BISTEN is not present, the machine is returned to the WAIT0 state (while all state transition arrows are shown for these transitions, not all of them are labeled).

Once BISTEN becomes active, the machine goes through two secondary wait states (WAIT1 and WAIT2) before starting to look for $\overline{RDY}$ being ac-

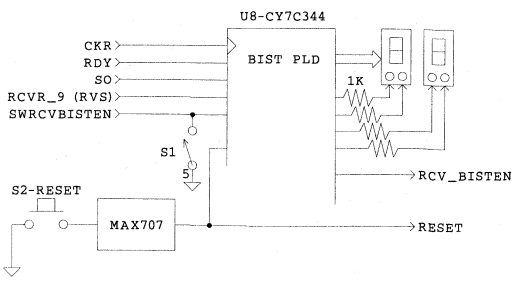


Figure 16. BIST Support Hardware

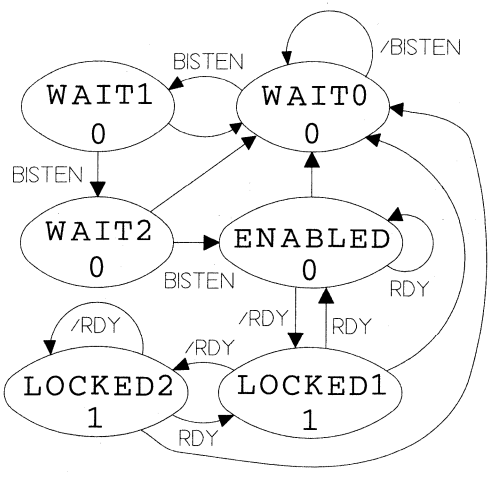


Figure 17. BIST State Machine Bubble Diagram

tive. These wait states are necessary to allow the receiver time to recognize the BISTEN signal and bring $\overline{RDY}$ high.

When the ENABLED state is reached, the machine remains in this state until $\overline{RDY}$ goes LOW, causing the machine to move to the first of the two LOCKED states. This signifies that the receiver has received the start-of-loop character (D0.0) and is now performing matching of the received data bits to its internal pattern generator.

In the LOCKED states, the external counter is enabled to count errors. The reason two LOCKED states are present is to allow for the single pulse on $\overline{RDY}$ that indicates the end of a BIST loop. If $\overline{RDY}$ is ever HIGH for more than one clock, the HOTLink Receiver has determined that it is no longer in sync with the transmitter and it starts looking for the start-of-loop character again.

Other BIST PLD Functions

The complete schematic for the BIST PLD is shown in Appendix C. Other than the BIST state machine, the other main logic functions present in the part are for driving the four status indicators and the actual error counter.

Error Display

The error display is made from two hexadecimal LED displays (TIL311). These displays are each capable of showing the entire hexadecimal character set (0–9, A–F) as well as having two independent decimal points. These decimal points are used as individual status indicators for the board.

External Serial Interface Connections

The primary difference between the CY9266 card types is in the external high-speed serial interface. Each of the card types operates with not only a different media type (optical, coaxial, shielded twisted pair), but also different connectors and cable types.

CY9266-F Serial Interface Connections

The CY9266-F HOTLink Demonstration Board implements a fiberoptic-based serial interface. This interface uses industry-standard LED-based fiberoptic modules that accept SC-type fiberoptic connectors.

Optical Modules

The CY9266-F HOTLink Demonstration Board is designed to operate using a *de facto* standard-footprint optical module. Any optical module meeting the pinout and dimensions of this *de facto* standard (established originally for FDDI) should operate with the CY9266-F.

Note: These standard-footprint optical modules are available in a wide range of operating data rates. Because the operating data rate for some of these modules may be outside the 160- to 330-Mbit/second operating range of the HOTLink Transmitter and Receiver, care should be exercised when selecting an optical module.

This footprint supports two types of optical modules: those with four rows of vertical pins, and those with a single row of pins along the bottom edge. In vendor literature these are referred to as DIP- and endfire-type packages.

While specified originally for FDDI, modules meeting this footprint are also available for Fibre Channel and ATM data rates, and meet all optical and mechanical specifications of the Fibre Channel Standard. *Figure 18* shows the mechanical footprint dimensions of this *de facto* standard package.

Both package types operate from a +5V supply and interface directly with 100K ECL/PECL. The biggest mechanical difference between them is that the endfire type packages have two oversized pins (1 and 32) that are used only to hold the package in place. The main electrical difference between the packages types is that the DIP package drives the Signal Detect output differentially while the endfire package only provides the active HIGH output. *Table 5* lists the pinouts for this standard-footprint optical module.

The active signals listed in *Table 5* are

- SD — Signal Detect
- TD — Transmit Data
- RD — Receive Data
- Case — Outer Case of Module

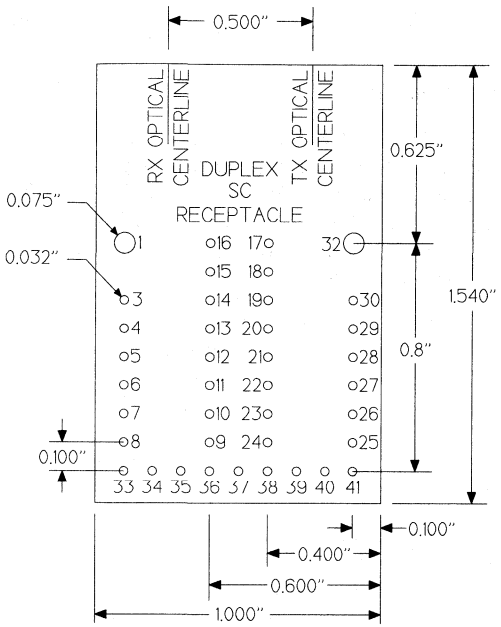


Figure 18. Optical Module, Top View Dimensions

- V_{CC} — Positive Supply Voltage
- V_{EE} — Negative Supply Voltage

Pins marked "Case" are not necessarily isolated pins. Because the optical module is used in the CY9266-F in a PECL mode, these Case pins are connected to the V_{EE} (ground) supply. When selecting an optical module, care should be taken to insure that the pins marked "Case" are either floating or are attached to the appropriate power supply rail.

To allow evaluation of different types of optical modules, the CY9266-F demo board is built using low-profile socket pins for the optical module. This allows the modules to be easily replaced. In addition, two slotted holes are provided for a cable-tie to hold the module in place.

Table 5. Optical Module Pinout

DIP Pin Assignments			
Pin	**Signal**	**Pin**	**Signal**
1	Case	2	No Pin
3	Case	4	V_{EE}
5	V_{EE}	6	+SD
7	−SD	8	Case
9	Case	10	−RD
11	+RD	12	V_{CC}
13	V_{CC}	14	V_{CC}
15	Case	16	Case
17	Case	18	Case
19	V_{CC}	20	V_{CC}
21	Case	22	+TD
23	−TD	24	Case
25	Case	26	V_{BB}
27	Case	28	Case
29	V_{EE}	30	V_{EE}
31	No Pin	32	Case
Endfire Pin Assignments			
Pin	**Signal**	**Pin**	**Signal**
33	V_{EE}	34	+RD
35	−RD	36	+SD
37	V_{CC}	38	V_{CC}
39	−TD	40	+TD
41	V_{EE}		

Fiberoptic Connector

The optical modules specified for use on the CY9266-F HOTLink demonstration card (listed in Appendix A, item U4) are designed to accept SC-type fiberoptic connectors. These connectors are available in both simplex (single-fiber) and duplex (dual-fiber) versions. *Figure 19* shows a simplex SC fiberoptic connector. A duplex connector is formed either by joining two simplex connectors together with a clip (sometimes referred to as a "Z" clip) or by using a connector that supports two fibers in the same form factor. The standard optical fiber type

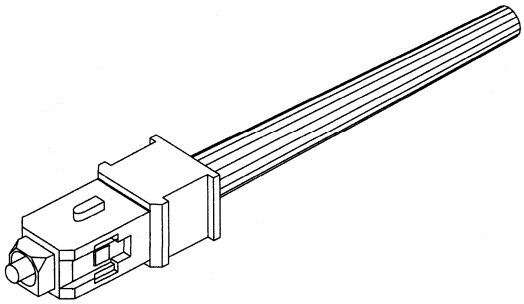


Figure 19. SC Simplex Fiberoptic Connector

used with these connectors and LED-based optical modules is 62.5/125-µm multimode-fiber.

When using duplex connector cables, the cable construction controls which fiber is connected to the transmit LED and which is connected to the receive photodetector. When using simplex cables, this polarization control is left to the user. The transmit and receive connectors on the fiberoptic module are shown in *Figure 20*.

CY9266-C Serial Interface Connections

The CY9266-C HOTLink Demonstration Board implements a copper-based serial interface. This interface uses 75Ω coaxial cables having BNC- and TNC-type connectors.

Coaxial Board Connectors

The CY9266-C HOTLink Demonstration Board has two right-angle female coaxial cable connectors: a BNC (Bayonet Neil-Councilman) for the J1 transmit

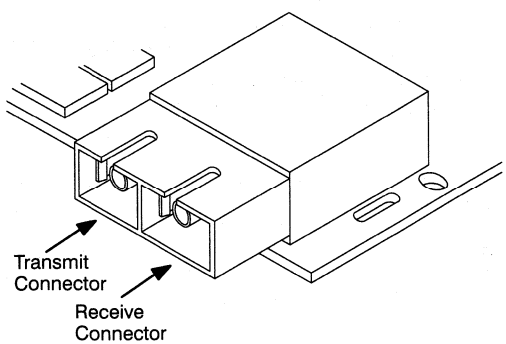


Figure 20. U4 Fiberoptic Module Connectors

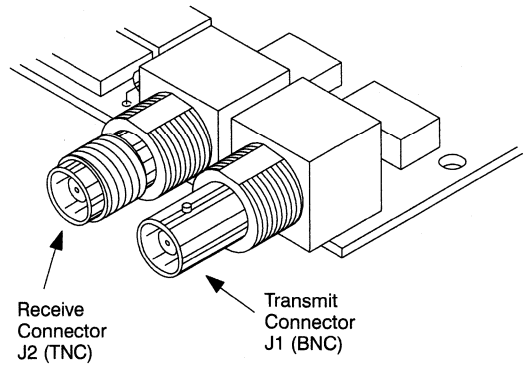


Figure 21. J1 and J2 Coaxial Board Connectors

connector, and a TNC (Threaded Neil-Councilman) as the J2 receive connector. These connectors and their location on the board are shown in *Figure 21*.

Coaxial Cable Connectors

Many different coaxial cables may be used with the CY9266-C HOTLink Demonstration Board. The only requirements for the cable are 75Ω characteristic impedance and BNC/TNC connectors at each end to attach to the board. Other cable impedances may also be used, however, the termination resistors on the board (R40 and R41) must then be changed for correct operation.

Coaxial cables for the CY9266-C should have a BNC connector on one end and a TNC connector on the other. This dual connector mechanism is specified by ANSI to prevent the inadvertent cabling of a transmitter to another transmitter, or a receiver to another receiver. When connecting cables to a CY9266-C board, the cable BNC connector always attaches to a transmit port (J1) and the cable TNC connector always attaches to a receiver port (J2). TNC/BNC dual-female barrel connectors (e.g., Amphenol #76400) are available to allow splicing of cables to evaluate multiple lengths of cable. *Figure 22* illustrates typical TNC and BNC connectors.

CY9266-T Serial Interface Connections

The CY9266-T HOTLink Demonstration Board implements a copper-based serial interface. This interface uses 150Ω shielded twisted pair (STP) cables with 9-pin male D-subminiature-type connectors.

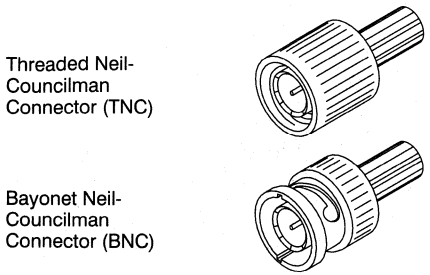


Figure 22. TNC/BNC Cable Connectors

STP Board Connectors

The CY9266-T HOTLink Demonstration Board has a right-angle female 9-pin D-subminiature cable connector. Unlike the coaxial cable version of the CY9266, which uses separate connectors for transmit and receive, the CY9266-T uses only a single connector (P1) for both. This connector and its location on the board is shown in *Figure 23*.

STP Cable Connector

There are presently two STP cable types identified by ANSI for use with Fibre Channel; both specify 150Ω differential characteristic impedance. These cable types are known as either EIA/TIA568 Type-1 and Type-2, or more generically as IBM® Type-1 or Type-2. Both of these cable types contain two individually shielded pairs of solid conductors. The Type-2 cable also contains four non-shielded conductors that are often used for either low-speed signaling or voice-grade communications.

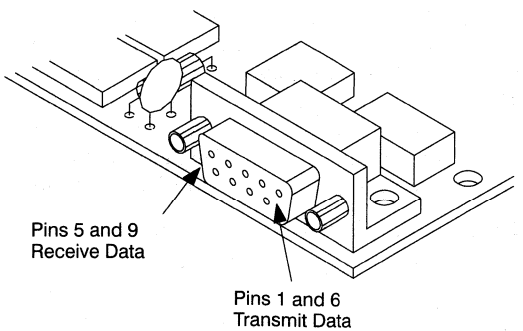


Figure 23. STP P1 Board Connector

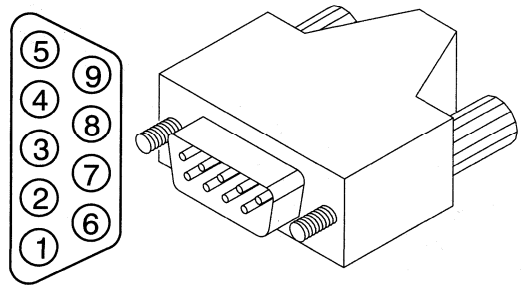


Figure 24. STP Cable Connector and Connector Pinout

For installations where the cables may see more flexing, a stranded conductor cable is available that meets the 150Ω impedance. This cable type is commonly known as IBM Type-6. Other cable types may also be used with the CY9266-T HOTLink Demonstration Board. The only requirements for the cable are 150Ω differential characteristic impedance and a properly wired (see *Figure 25*) 9-pin male D-subminiature connector at each end of the cable. Other cable impedances may also be used, however, the termination resistors on the board (R40 and R41) must then be changed for correct operation.

Figure 24 shows an example of a compatible STP cable connector and how the pins in the connector are numbered. This is a 9-pin male D-subminiature connector. While connectors of this type are available with a plastic housing, proper operation with STP cables requires using connectors having a metal or conductive shell. When properly connected, as shown in *Figure 25*, the shield of each pair in the cable is attached to the conductive front shell of the connector. To maintain shielding effectiveness it is recommended that the connector backshell/strain relief also be metallic or conductive.

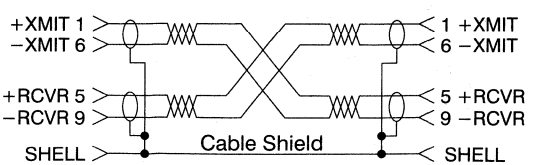


Figure 25. STP Cable Connections

The STP cable is wired in a crossover fashion where the transmit connections at one end of the cable are connected to the receive connections at the other end of the cable, as illustrated here in *Figure 25*. The cable shields for both pairs are tied together and connected to the D-sub shell at each end.

OLC Mode Configuration

The CY9266 demo board may be configured to operate in an OLC-266 compatible system. This emulation is strictly at the TTL parallel interface level; the optical and electrical serial interfaces are not compatible. In addition, the CY9266 is only a single-channel board while the OLC-266 is available in either single- or dual-channel versions.

The TTL parallel interface attachment is provided through the JP4 connector. This connector is pinned and positioned to mate with host systems designed for the OLC-266 board.

The following configuration sets the CY9266 for 10-bit data and Bypass mode on both the transmitter and receiver. The transmitter and receiver are both clocked by the XMITCLOCK signal on JP4, and the receiver A/$\overline{B}$ selection is controlled by the LOOP_BACK signal on JP4.

JP1 Settings

The CY9266 jumper block JP1 controls many of the options on the board. For the CY9266 to operate in an OLC socket, jumper block JP1 must be configured with shorting jumpers as shown in *Figure 26*.

The shorting jumper across pins X and Y of JP1-B allows the LOOP_BACK signal in the JP4 connector to control the A/$\overline{B}$ input selection on the HOTLink Receiver. The jumper across pins X and Y of JP1-F allows the LINK_CONTROL signal to control the FOTO enable of the HOTLink Transmitter. The jumper connecting pins X and Y of JP1-G connects the XMITCLOCK input to the HOTLink Transmitter CKW clock. The jumper connecting pins JP1-HX to JP1-IX connects the XMITCLOCK input to the HOTLink Receiver REFCLK input.

Note: The active signal level of the LOOPBACK signal, as implemented on the CY9266, is opposite that

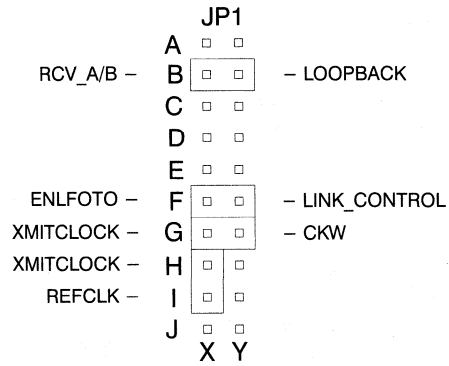


Figure 26. JP1 OLC-Compatibility Settings

of an actual OLC-266 card. If this signal is under software control, it should be programmed to allow signal loopback when the signal is active LOW. For hardware controlled systems an external signal inversion is necessary, or the signal may be jumpered at JP1 for operation from the S1-7 DIP switch.

S1 Settings

The S1 DIP switch is also used to configure many of the HOTLink Transmitter and Receiver options. The settings for these switches are listed in *Table 6*.

Table 6. S1 OLC-Compatibility Settings

DIP Switch Settings		
Sw #	**State**	**Controlled Signal**
1	Off	Transmitter BIST Enable
2	Off	Transmitter Mode Select
3	Off	Enable Next Parallel Xmit Data
4	On	Enable Parallel Xmit Data
5	Off	Receiver BIST Enable
6	Off	Receiver Mode Select
7	N/A	Switch Controlled Loopback
8	N/A	Switch Controlled FOTO
9	Off	Carrier-Detect Polarity Select
10	Off	BYTE_SYNC Polarity Select

The setting of switches S1-7 and S1-8 are not applicable when jumpers JP1-B and JP1-F are in place.

Assembly and Options

The design of the CY9266-F and CY9266-C/T Demo Boards offer many different assembly options for those users interested in making modifications for their own evaluation.

Optical Module

Optical module U4 on the CY9266-F is socketed for user evaluation of different optical modules. The hole pattern on the board supports direct soldering of the optical module to the board. This should not be attempted on a board that is already equipped with a socket for the module because removal of the socket pins may damage the board.

Transmitter

The HOTLink Transmitter B± differential output signals on the board are left open to conserve power. Pads are present on the bottom of the board (labeled R1, R2, R3, and R4) for bias/termination resistors for these outputs. While these resistors are present on the board schematic, they are not part of the delivered assembly. If the B± outputs are used for probing or test purposes, resistors must be added in these locations to enable the output drivers.

Oscillator

The on-board oscillator (U5) is used primarily for exercising the BIST capability of the board in a standalone mode. If the board is only to be used with an external clock, the oscillator does not need to be present. This part is socketed to allow the user to select the operating frequency.

When selecting an oscillator, care must be taken to insure the frequency stability and jitter characteristics of the oscillator are within the specifications of the HOTLink Receiver and Transmitter and the intended system application.

The hole pattern on the board supports direct soldering of the oscillator to the board. This should not be attempted on a board that is already equipped with a socket for the oscillator, as removal of the socket pins may damage the board.

BIST Support Hardware

The BIST support hardware does not interact with the functionality of the HOTLink Transmitter or Receiver and is not part of the communications link. If there is no requirement for BIST test and display hardware, the following components may be removed from the board:

- U6 and U7 – TIL311 Hex Displays
- U8 – CY7C344 EPLD
- S2 – Reset Switch
- R21, R22, R23, and R24 – 1 kΩ
- C13 – 0.022 µF
- C18 – 100 pF

Voltage Monitor

The voltage monitor (U11) is also used as part of the BIST function but does drive the RESET signal on JP2, JP3, and JP4. If monitoring of the specific voltage is not necessary (and BIST capability is not used) this part may be removed.

If U11 is removed, it may be necessary to bias the RESET line to allow an external system controller to properly sense a high on the RESET output. This may be done by soldering a jumper wire from pin 7 of U11 to pin 2 of R20.

JP2

The area of the board labeled as JP2 provides a hole pattern designed to accept multiple types of headers and connectors. These connectors allow access to all the same signals present on JP4 and JP3.

The current pin 1 designation for JP2 assumes a pin-header connector designed for flat cable is attached to bottom of the board. If this type of connector is instead attached to the top of the board, the even and odd pins are effectively swapped in the connector and cable, from those listed in *Table 1*.

OLC-Compatibility Registers

The 74F174 hex D-registers (U9 and U10) are used to provide compatibility with OLC-266 sockets. For those users not requiring this capability, or for those who wish to use the receiver $\overline{\text{RDY}}$ signal to clock re-

ceived data into asynchronous FIFOs, these registers can be removed.

Once U9 and U10 are removed, it is necessary to short eleven adjacent pad pairs on U9 and U10 to allow the receiver data bus to connect to the output connectors. The pairs that must be shorted are listed in *Table 7.*

Table 7. OLC-Compatibility Register Bypass Connections

Register Pin Connections		
Part	**Pins**	**Signal Name**
U10	14, 15	RCVR_0
U10	12, 13	RCVR_1
U10	10, 11	RCVR_2
U10	6, 7	RCVR_3
U10	4, 5	RCVR_4
U10	2, 3	RCVR_5
U9	10, 11	RCVR_6
U9	12, 13	RCVR_7
U9	14, 15	RCVR_8
U9	6, 7	RCVR_9
U9	4, 5	RDY

Copper Cable Connectors

The CY9266-C and CY9266-T are assembled on the same substrate and may be configured for use with either coaxial or shielded-pair cables. Changing from coax to shielded-pair requires the removal of the J1 BNC and J2 TNC connectors and replacing them with a female 9-pin D-sub connector at location P1 (see Appendix B for manufacturer part numbers). Also, the foil traces that connect pins 6 and 9 of P1 to the shield of J1 and J2 (located on the bottom of the board) must be cut. Because the cable impedance used for shielded-pair cable is different from that of coax cable, the line termination resistors R40 and R41 must be replaced with 75Ω resistors, and coupling transformer T1 must also change to the higher inductance type.

Changing from shielded-pair to coax requires removal of the P1 D-sub connector and the addition of connectors J1 and J2. It is necessary to connect pin 6 of the P1 pad set to the shield pin of J1, and pin 9 of P1 to the shield pin of J2. Because the cable impedance used for coax cable is different from that of shielded-pair cable, the line termination resistors R40 and R41 must be replaced with 37.4Ω resistors, and coupling transformer T1 must also change to the lower inductance type.

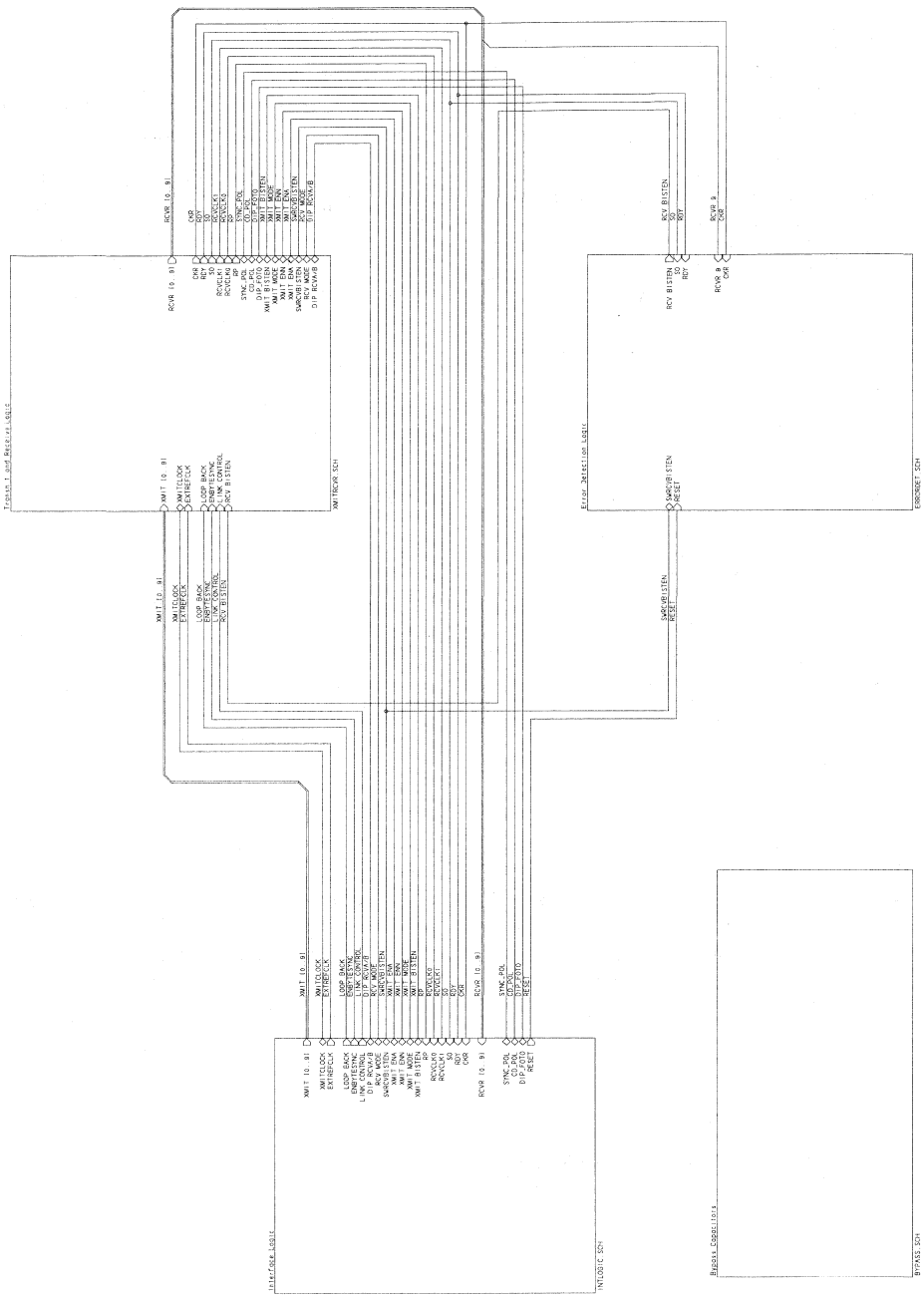
Redesign Capability

The CY9266-F, CY9266-C, and CY9266-T boards were designed strictly as a demonstration vehicle for the Cypress Semiconductor HOTLink family of communications parts. The designs shown here may not be optimal for most applications, as these are expected to be more specialized and may not require all the configuration and BIST demonstration hardware contained on these boards.

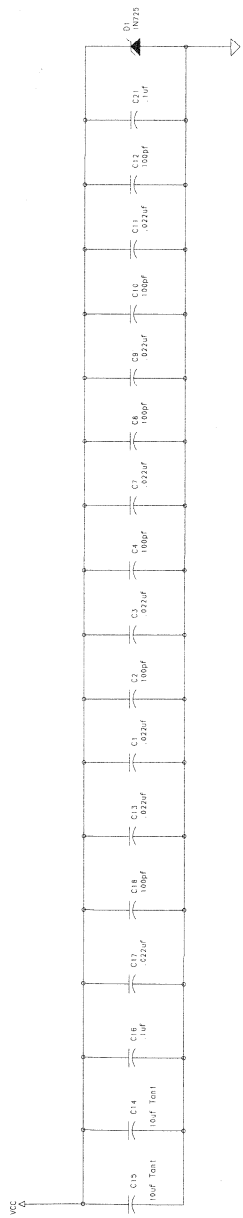
Examination of the demo boards will show that the components necessary for creating a serial link are all on one half of the board, while the components used for configuration and BIST support are located on the other half of the board. This placement of parts was intentional, and shows that two complete channels may be placed on a board of the same size as the CY9266 *without* placing active components on both sides of the board.

HOTLink is a trademark of Cypress Semiconductor Corporation.
IBM is a registered trademark of International Business Machines.

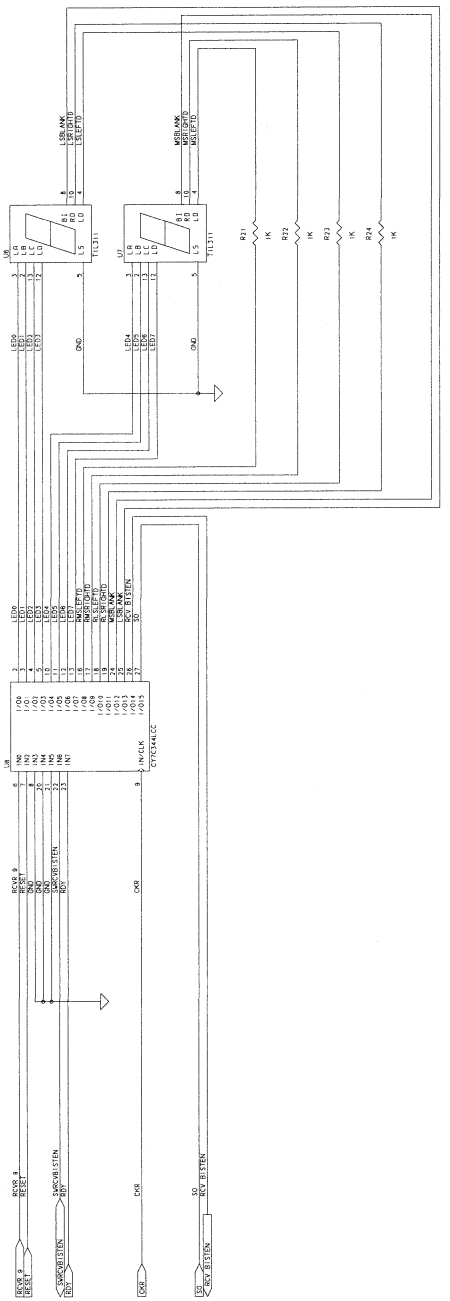
Appendix A. CY9266-F Schematic (Sheet 1 of 5)



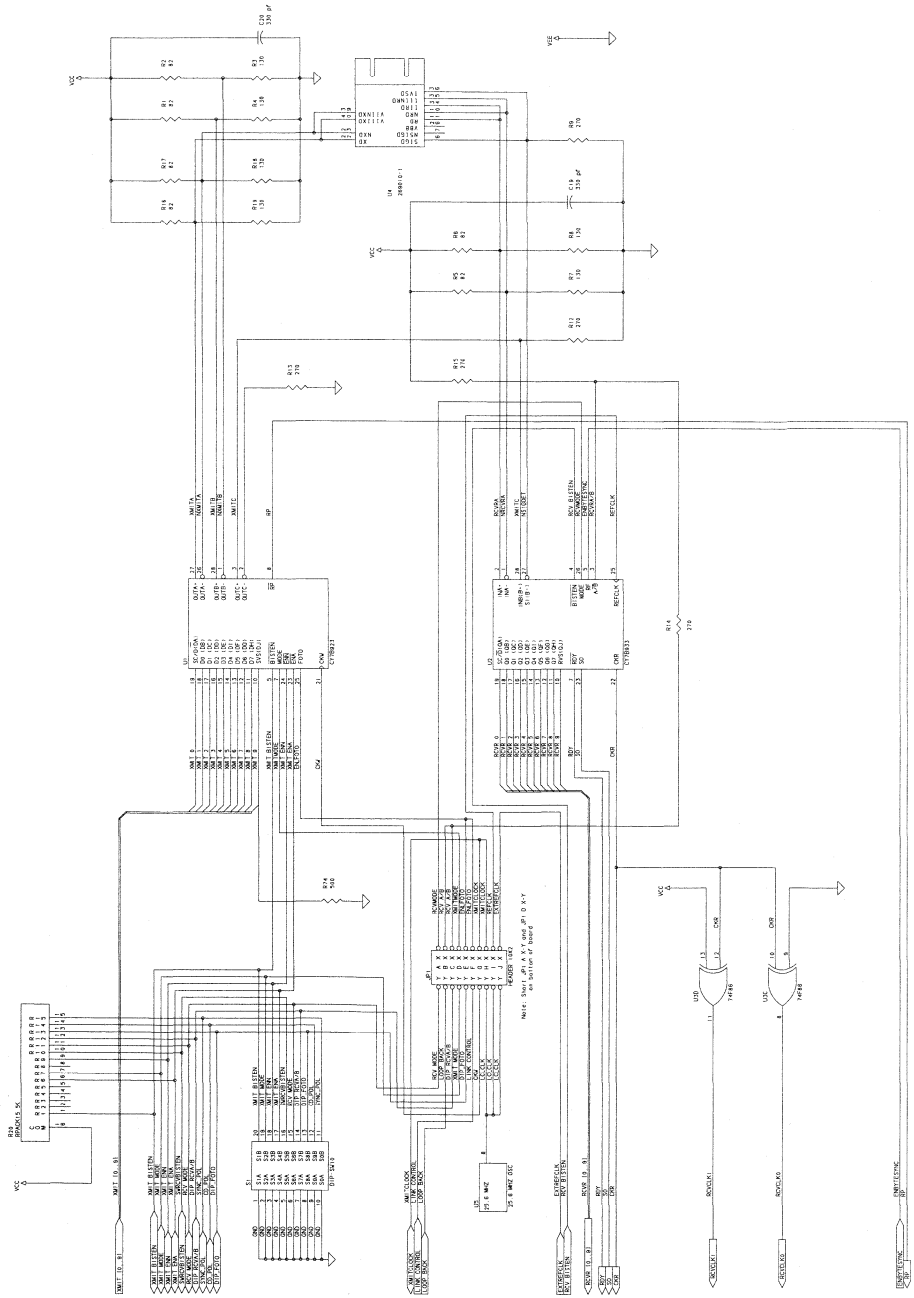
Appendix A. CY9266-F Schematic (Sheet 2 of 5)



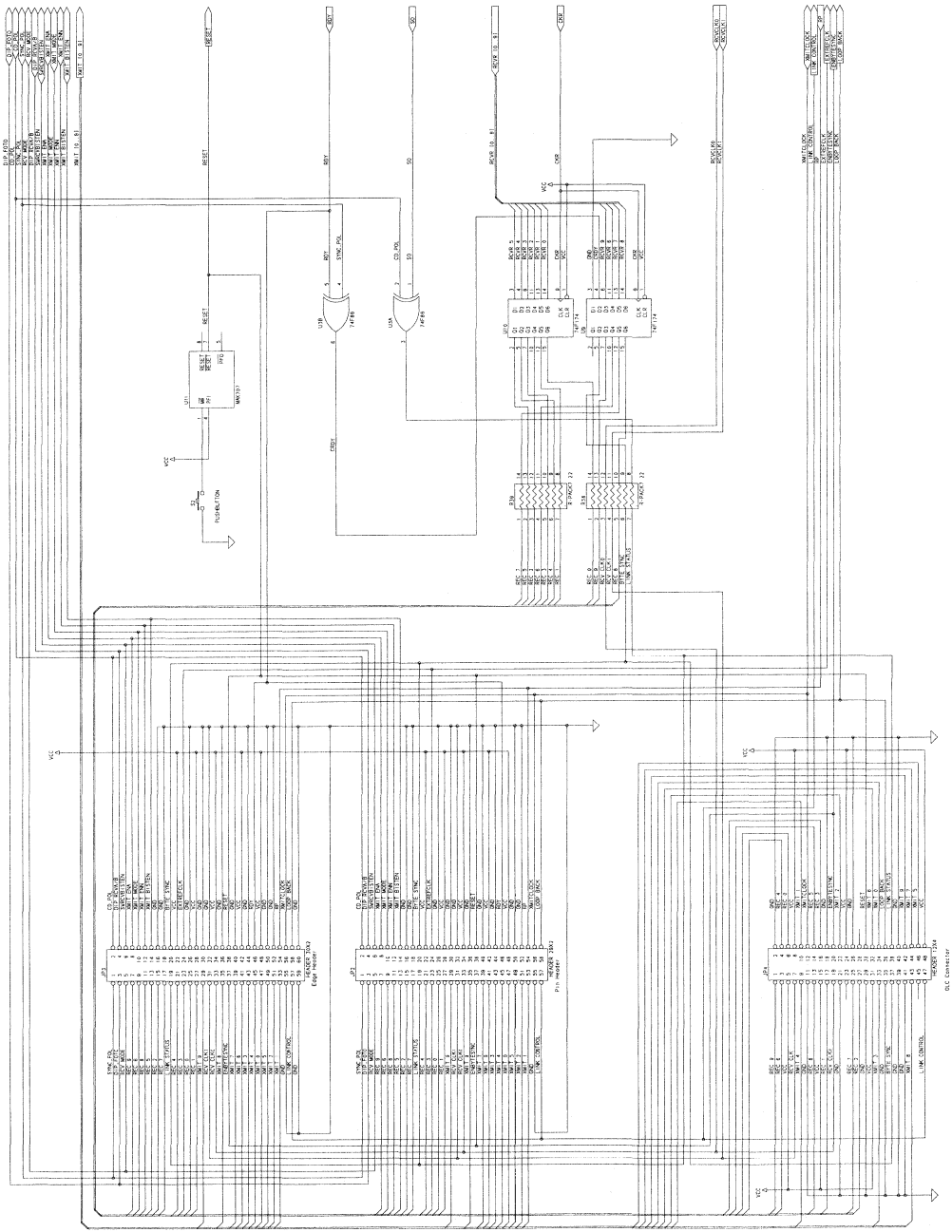
Appendix A. CY9266-F Schematic (Sheet 3 of 5)



Appendix A. CY9266-F Schematic (Sheet 4 of 5)



Appendix A. CY9266-F Schematic (Sheet 5 of 5)

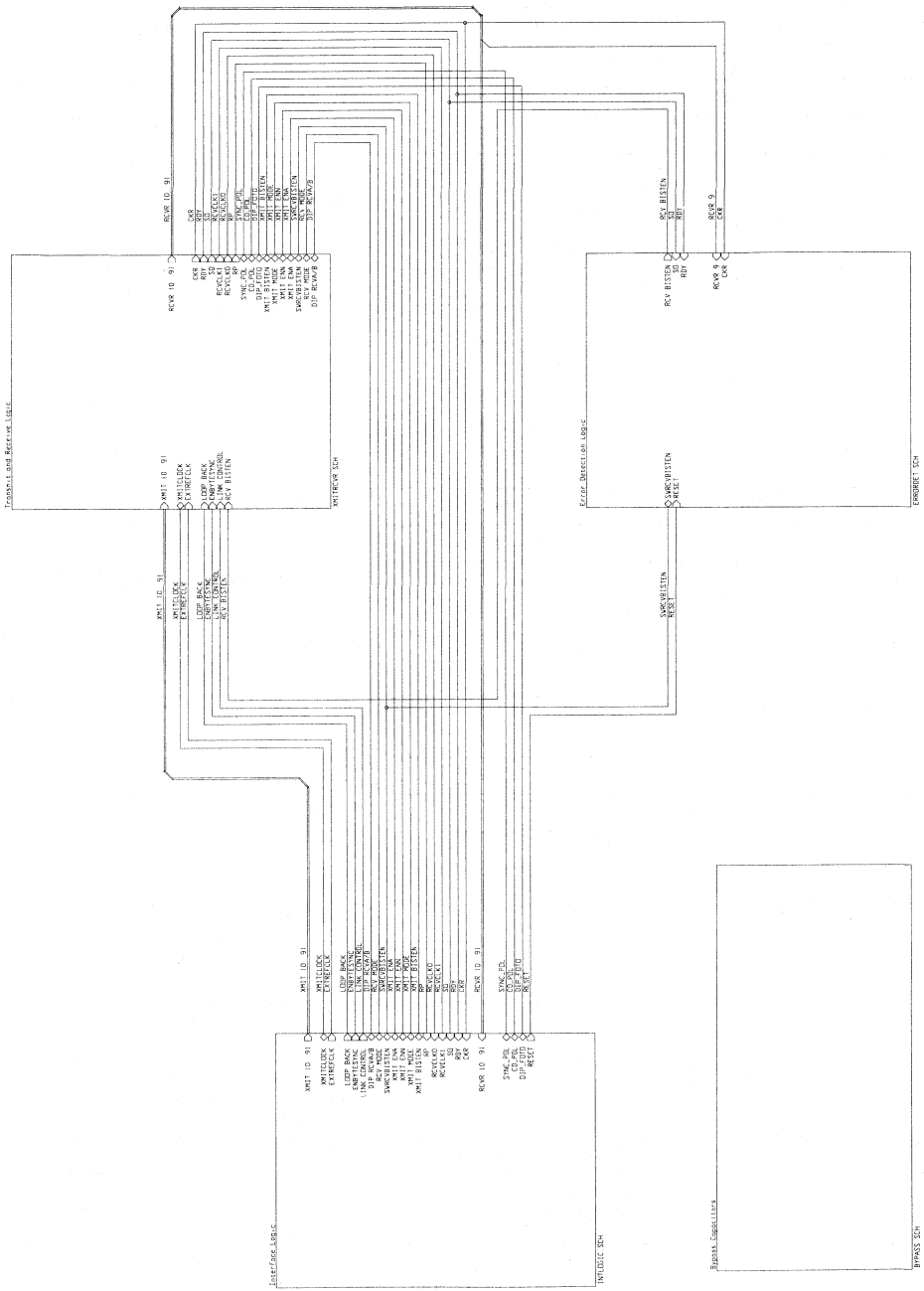


Appendix A. CY9266-F Parts List

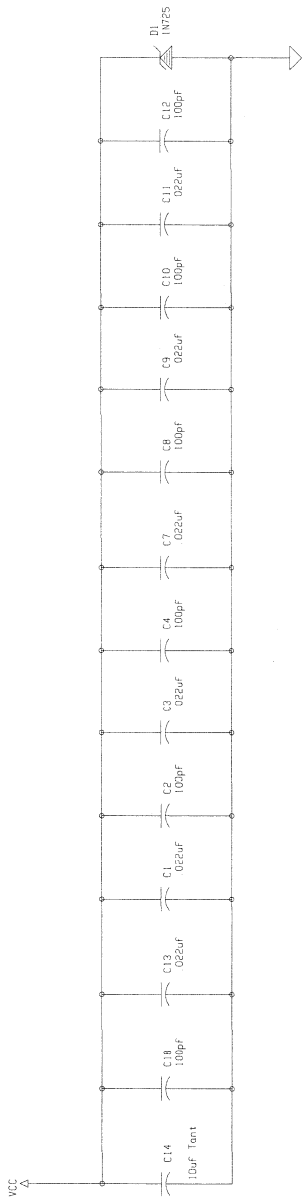
Instance	Part Number	Description
U1	Cypress CY7B923-JC	HOTLink Transmitter
U2	Cypress CY7B933-JC	HOTLink Receiver
U3	74F86	Quad XOR Gate SOIC Package
U4	AMP/Lytel 269010-1, Hewlett Packard HFBR-5302, Siemens TC-266C2EP, AT&T 1408N, or Equivalent	266-MB/s 1300-nm LED Transceiver Module
U5*	CTS CTX126 or Equivalent	25-MHz TTL Clock Oscillator
U6*,U7*	TI TIL311	Hex Display With Logic
U8*	Cypress CY7C344-15HC	32-Macrocell MAX EPLD
U9,U10	74F174	Hex D-Register, SOIC Package
U11*	Maxim MAX707CSA or Equivalent	Voltage Monitor
D1*	1N4735A	1W, 6.2V Zener Diode
S1*	AMP 3-435668-0 or Equivalent	10-position DIP Switch
S2*	ECG 520-01-3 or Equivalent	Momentary Pushbutton Switch
JP1*	Sullins PZC10DAAN or Equivalent	2 x 10 Position 0.25″ Sq. Pin-Header
JP4	Sullins PZC12DFBN or Equivalent	2 – 2 x 12 Position 0.25″ Sq. Pin-Header
C1, C3, C7, C9, C11, C13, C17	0.022 µF MLC X7R	0805 Chip Cap
C2, C4, C8, C10, C12, C18	100 pF MLC NPO	0805 Chip Cap
C14, C15	10 µF 16V Tantalum	Electrolytic Cap
C16, C21	0.1 µF MLC X7R	1206 Chip Cap
C19, C20	330 pF MLC NPO	0805 Chip Cap
R5, R6, R16, R17	82Ω 1/8W	1206 Chip Resistor
R7, R8, R18, R19	130Ω 1/8W	1206 Chip Resistor
R9, R12, R13, R14, R15	270Ω 1/8W	1206 Chip Resistor
R21*, R22*, R23*, R24*	1-kΩ 1/8W, 5%	1206 Chip Resistor
R74	510Ω 1/8W, 5%	1206 Chip Resistor
R20	CTS 766-161-R512 or Equivalent	5.1-kΩ R-Pack-15 SO16
R38, R39	CTS 766-143-R220 or Equivalent	22Ω R-Pack-7 SO14
	AMP 645955-2 or Equivalent	41 – Low Profile Socket-Pin
	3M 929955-06 or Equivalent	4 – 0.1″ Centerline Shorting Jumper

* — Used only for supervisory functions. Not needed for communications.

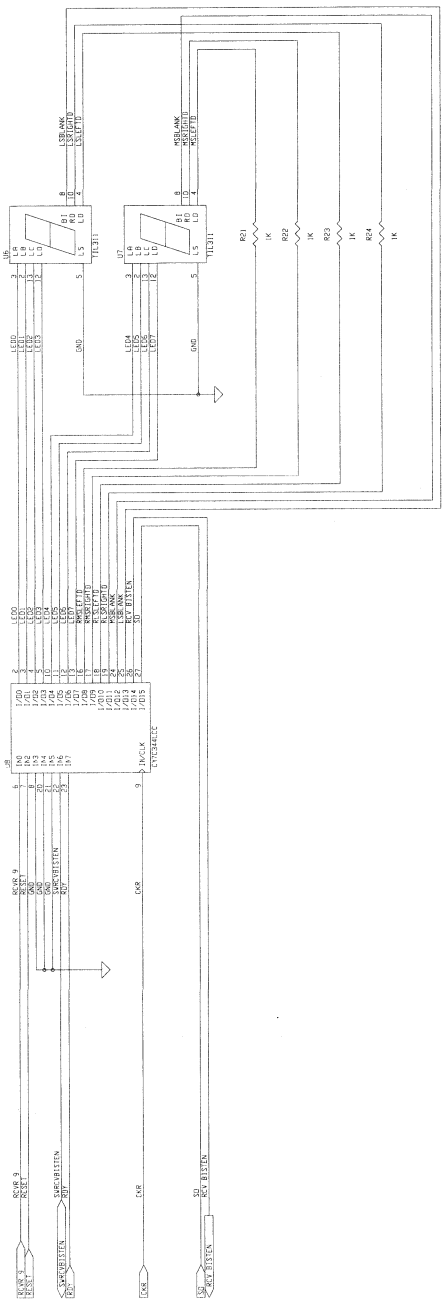
Appendix B. CY9266-C/T Schematic (Sheet 1 of 5)



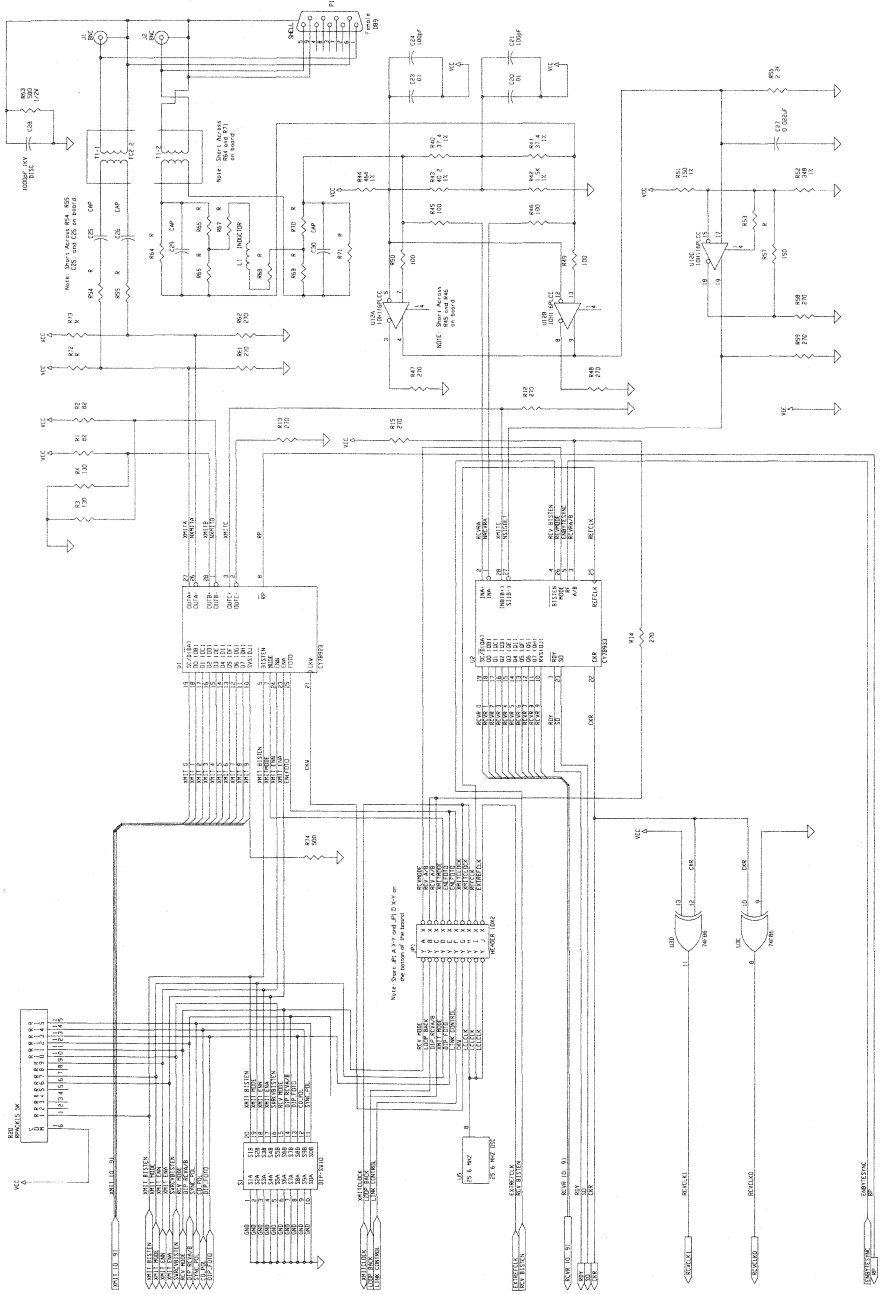
Appendix B. CY9266-C/T Schematic (Sheet 2 of 5)



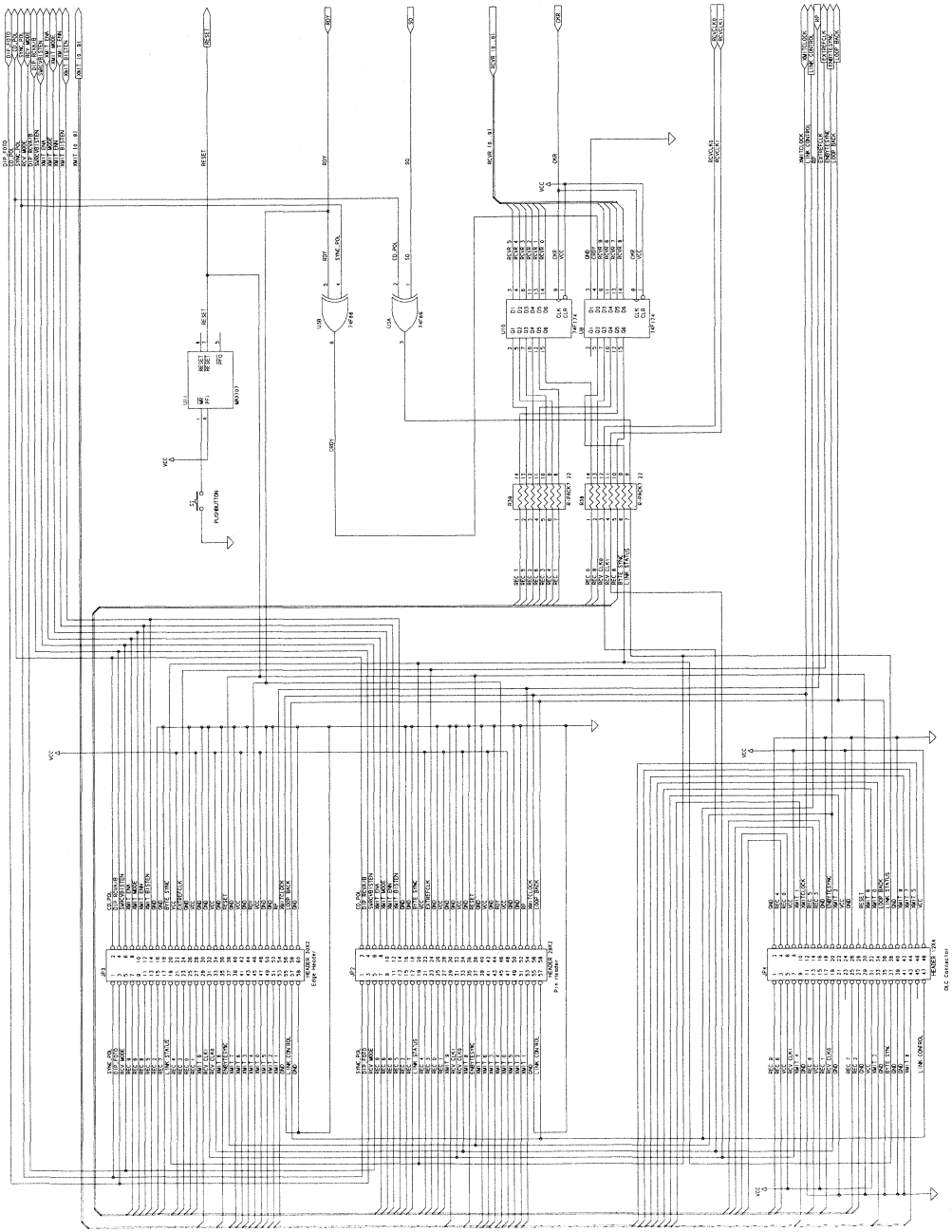
Appendix B. CY9266-C/T Schematic (Sheet 3 of 5)



Appendix B. CY9266-C/T Schematic (Sheet 4 of 5)



Appendix B. CY9266-C/T Schematic (Sheet 5 of 5)



Appendix B. CY9266-C/T Parts List

Instance	Part Number	Description
U1	Cypress CY7B923-JC	HOTLink Transmitter
U2	Cypress CY7B933-JC	HOTLink Receiver
U3	74F86	Quad XOR Gate SOIC Package
U5*	CTS CTX126 or Equivalent	25-MHz TTL Clock Oscillator
U6*,U7*	TI TIL311	Hex Display With Logic
U8*	Cypress CY7C344-15HC	32-Macrocell MAX EPLD
U9,U10	74F174	Hex D-register, SOIC Package
U11*	Maxim MAX707CSA or Equivalent	Voltage Monitor
U12*	Motorola MC10H116FN	ECL Tripple Line Receiver
D1*	1N4735A	1W, 6.2V Zener Diode
S1*	AMP 3-435668-0 or Equivalent	10-position DIP Switch
S2*	ECG 520-01-3 or Equivalent	Momentary Pushbutton Switch
J1	227161-3 or Equivalent	RA Female BNC Connector
J2	227818-1 or Equivalent	RA Female TNC Connector
JP1*	Sullins PZC10DAAN or Equivalent	2 x 10 Position 0.25″ Sq. Pin-Header
JP4	Sullins PZC12DFBN or Equivalent	2 – 2 x 12 Position 0.25″ Sq. Pin-Header
P1	747844-6 or Equivalent	RA Female 9-Pin D-Sub Connector
C14	10 µF 16V Tantalum	Electrolytic Cap
C1, C3, C7, C9, C11, C13, C27*	0.022 µF MLC X7R	0805 Chip Cap
C2, C4, C8, C10, C12, C18, C21, C24*	100 pF MLC NPO	0805 Chip Cap
C20, C23*	0.01 µF MLC X7R	0805 Chip Cap
C28	1000 pF 1 kV, Y5P	Disc Cap
T1	Pulse Engineering PE-65507 for STP Pulse Engineering PE-65508 for coax	Dual-Wideband Pulse Transformer
R12, R13, R14, R15	270Ω 1/8W, 5%	1206 Chip Resistor
R74	510Ω 1/8W, 5%	1206 Chip Resistor
R21*, R22*, R23*, R24*	1-kΩ 1/8W, 5%	1206 Chip Resistor
R40, R41	37.4Ω 1/10W, 1% for Coax 75.0Ω 1/10W, 1% for STP	0805 Chip Resistor
R43	40.2Ω 1/10W, 1%	0805 Chip Resistor
R49*, R50*	100Ω 1/10W, 5%	0805 Chip Resistor

Appendix B. CY9266-C/T Parts List (continued)

Instance	Part Number	Description
R51*, R57*	150Ω 1/10W, 1%	0805 Chip Resistor
R47*, R48*, R58*, R59*, R61, R62	270Ω 1/10W, 5%	0805 Chip Resistor
R52*	348Ω 1/10W, 1%	0805 Chip Resistor
R44	464Ω 1/10W, 1%	0805 Chip Resistor
R42	1.5-kΩ 1/10W, 1%	0805 Chip Resistor
R56*	2.2-kΩ 1/10W, 5%	0805 Chip Resistor
R63	510Ω 1/2W	Axial Lead Resistor
R20	CTS 766-161-R512 or Equivalent	5.1-kΩ R-Pack-15 SO16
R38, R39	CTS 766-143-R220 or Equivalent	22Ω R-Pack-7 SO14
	AMP 645955-2 or Equivalent	4 – Low Profile Socket-Pin
	3M 929955-06 or Equivalent	4 – 0.1″ Centerline Shorting Jumper

* — Used only for supervisory functions. Not needed for communications.

Appendix C. BIST PLD State Machine Source Code

```
SUBDESIGN bist_sm  (ready, bisten, clock : INPUT;
                    enable               : OUTPUT)

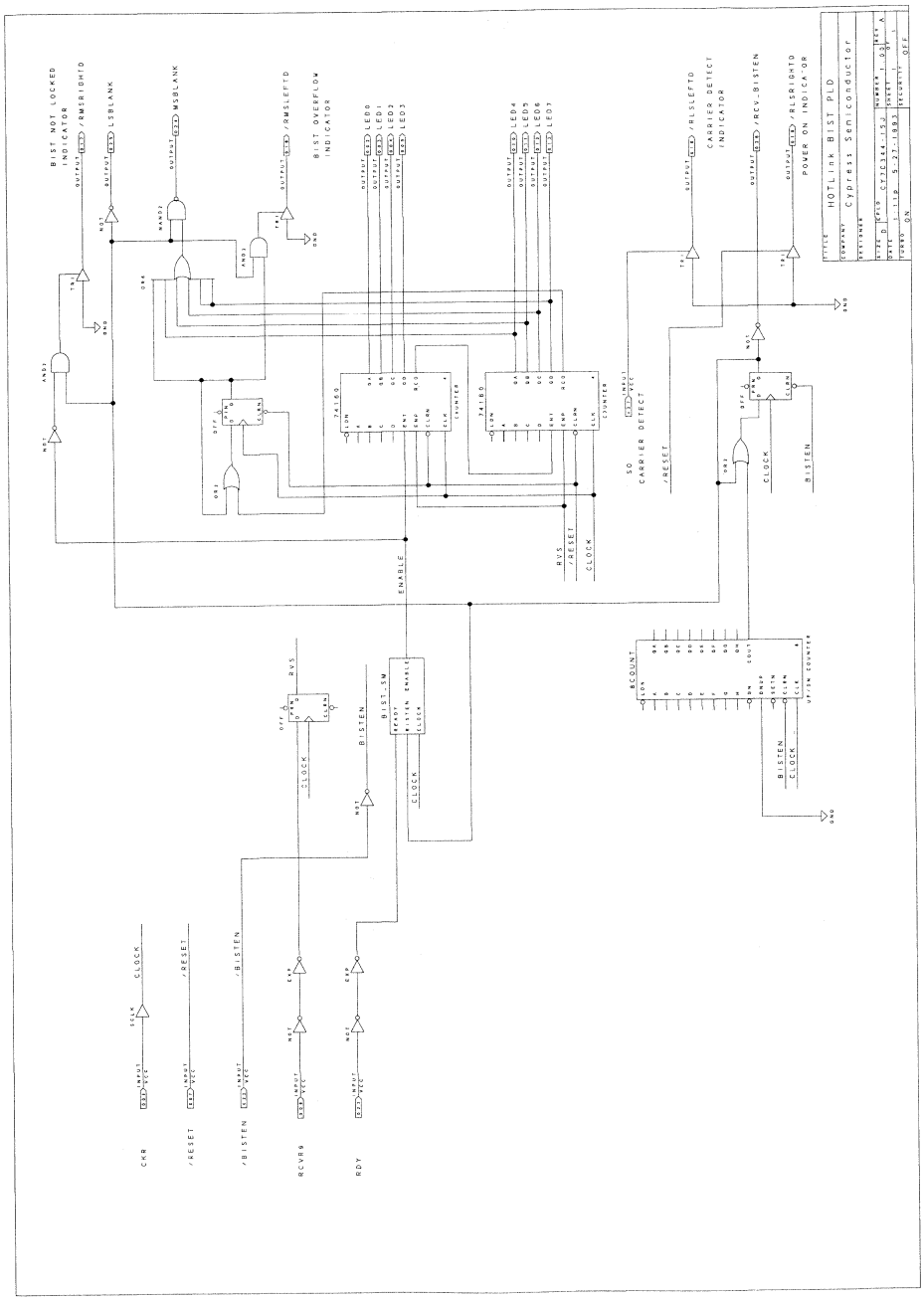

VARIABLE
       ss : MACHINE OF BITS  (enable_q)
                     %state    output%
            WITH STATES (wait0    =   0,
                         wait1    =   0,
                         wait2    =   0,
                         enabled  =   0,
                         locked1  =   1,
                         locked2  =   1);

BEGIN
       ss.clk = clock;              %assign machine clock%
       enable = enable_q;           %assign output of machine%

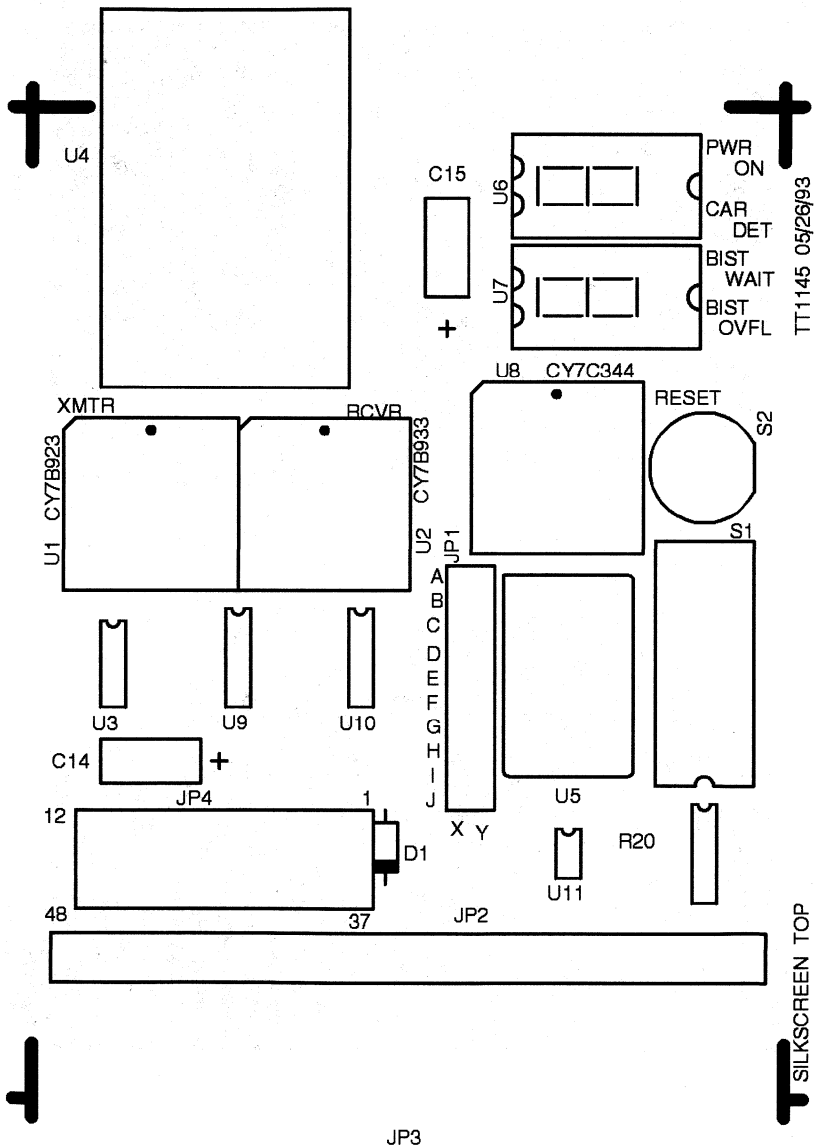
       TABLE
              %present    present            next %
              % state     inputs             state%
                 ss,   bisten, ready =>      ss;

       % define reset vectors %
                wait0,      0,   x    =>     wait0;
                wait1,      0,   x    =>     wait0;
                wait2,      0,   x    =>     wait0;
                enabled,    0,   x    =>     wait0;
                locked1,    0,   x    =>     wait0;
                locked2,    0,   x    =>     wait0;
       % define operational vectors %
                wait0,      1,   x    =>     wait1;
                wait1,      1,   x    =>     wait2;
                wait2,      1,   x    =>     enabled;
                enabled,    1,   1    =>     enabled;
                enabled,    1,   0    =>     locked1;
                locked1,    1,   1    =>     enabled;
                locked1,    1,   0    =>     locked2;
                locked2,    1,   1    =>     locked1;
                locked2,    1,   0    =>     locked2;
       END TABLE;
END;
```

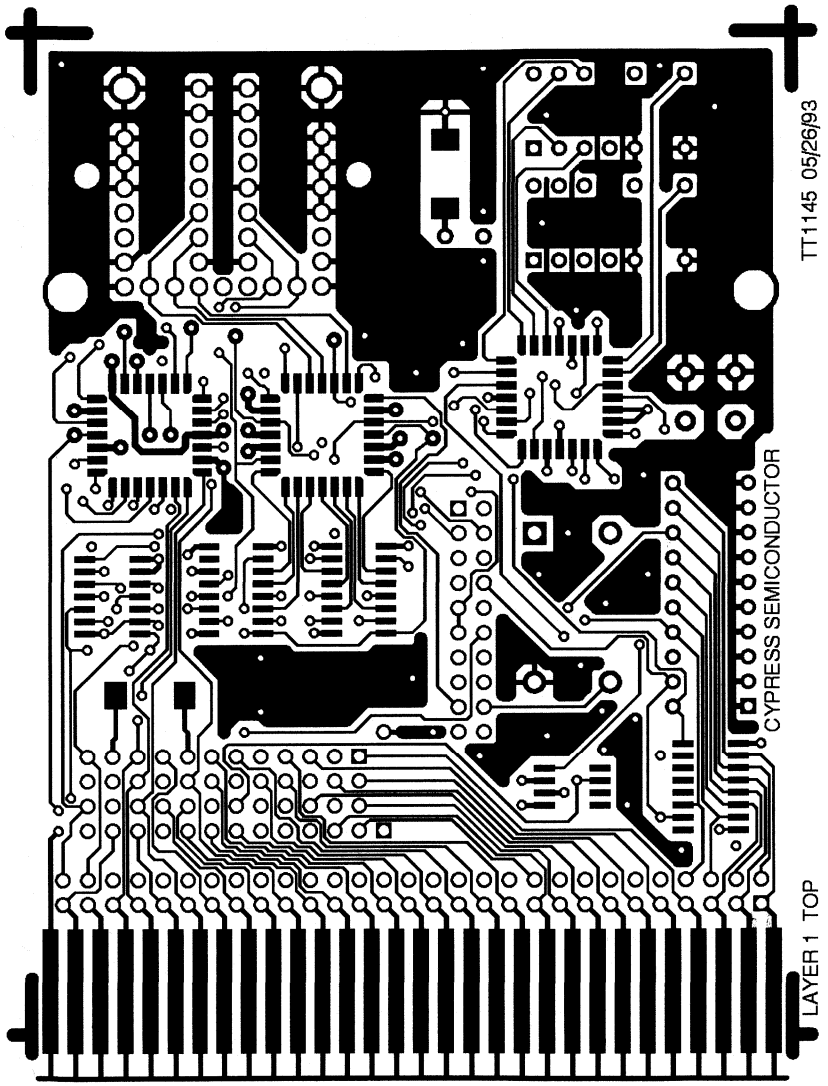
Appendix C. BIST PLD Logic Schematic



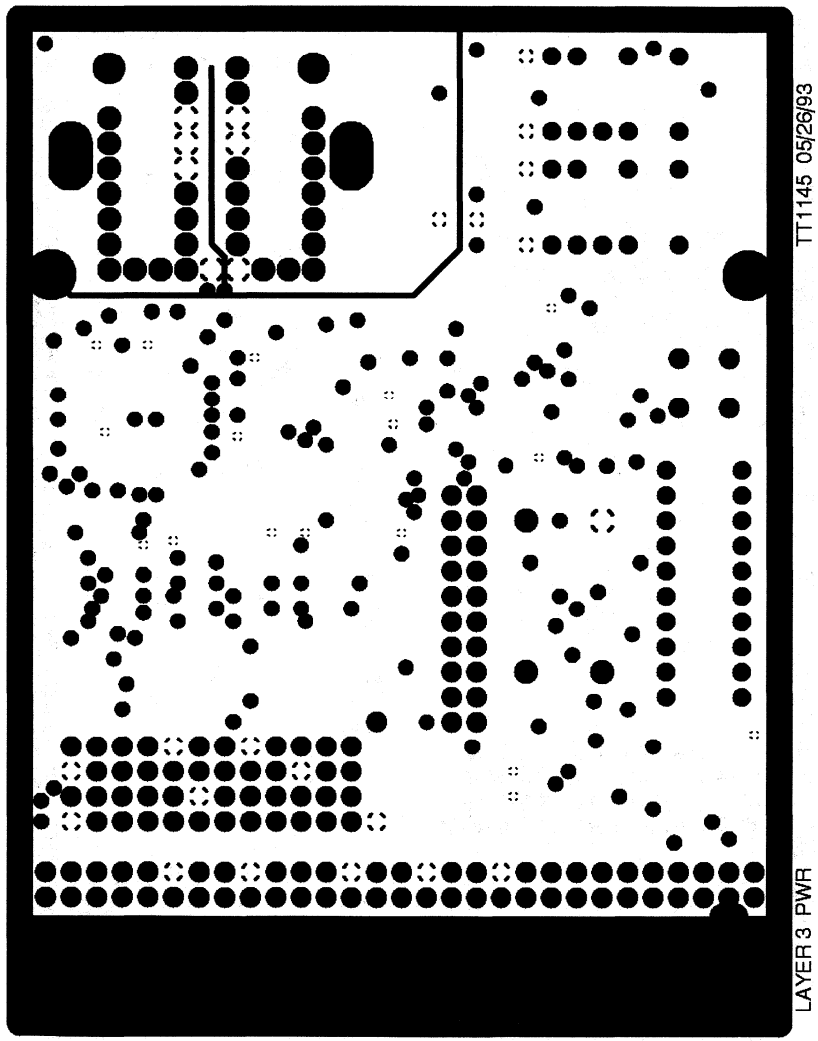
Appendix D. CY9266-F Artwork — Top Silkscreen



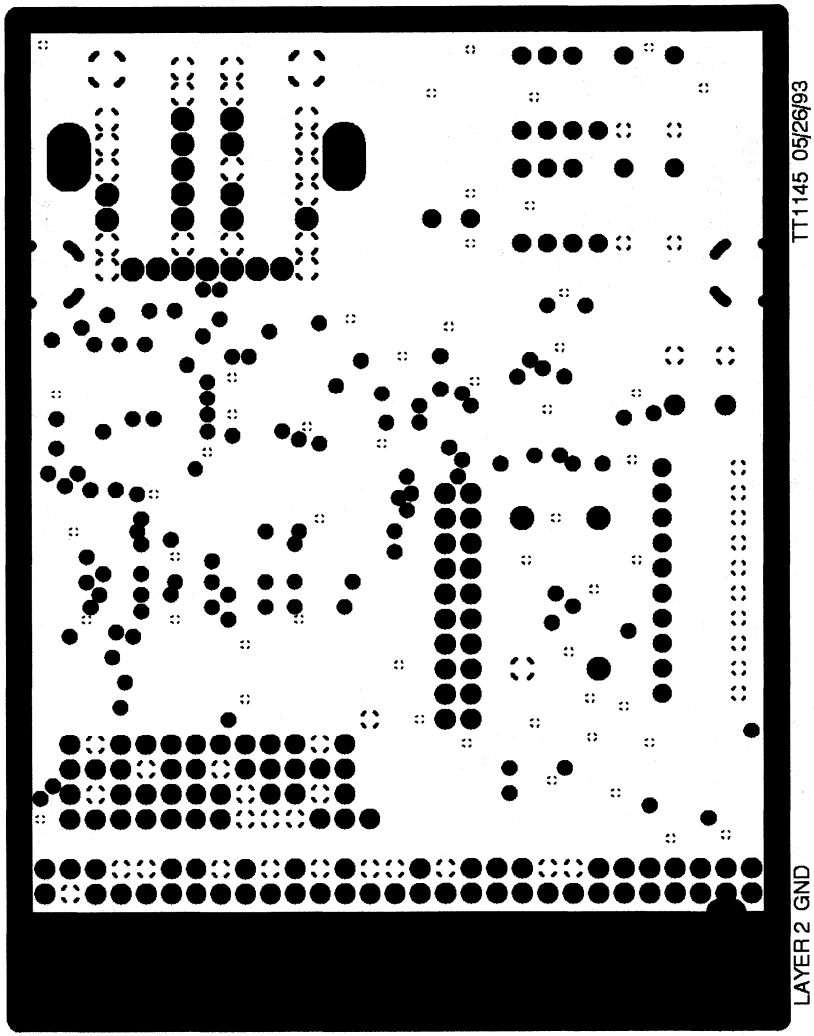
Appendix D. CY9266-F Artwork — Top Layer Copper



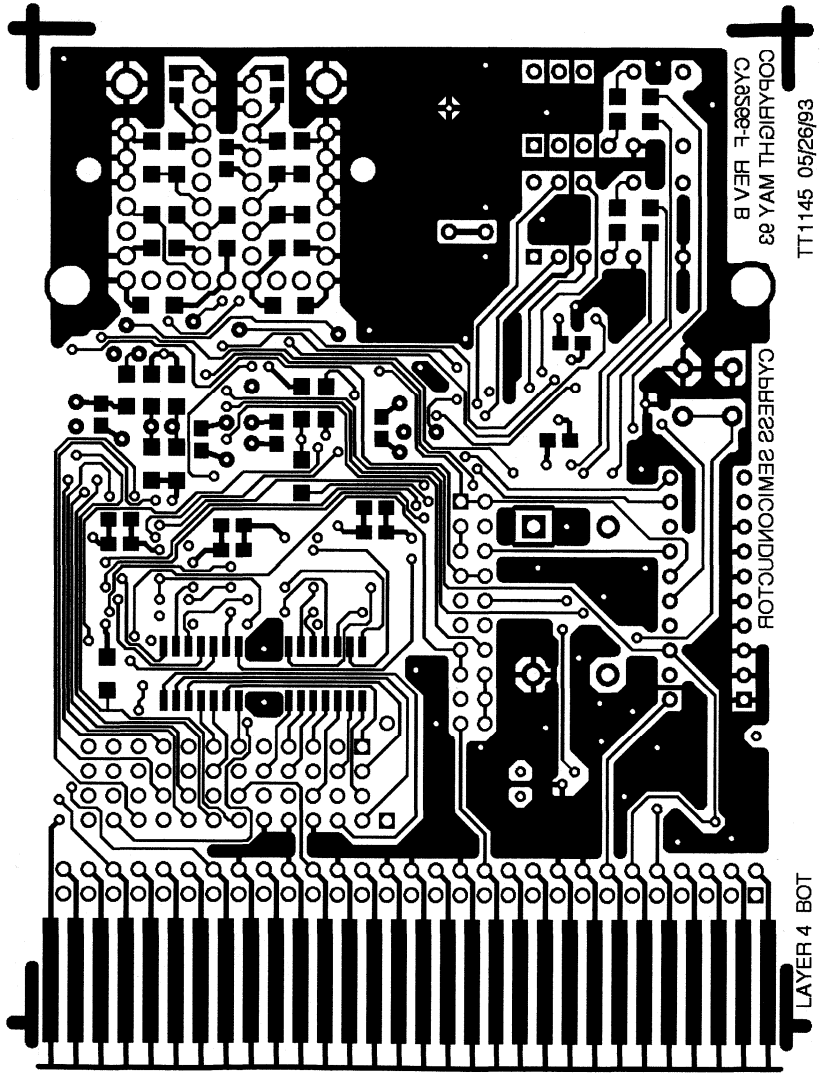
Appendix D. CY9266-F Artwork — Power layer



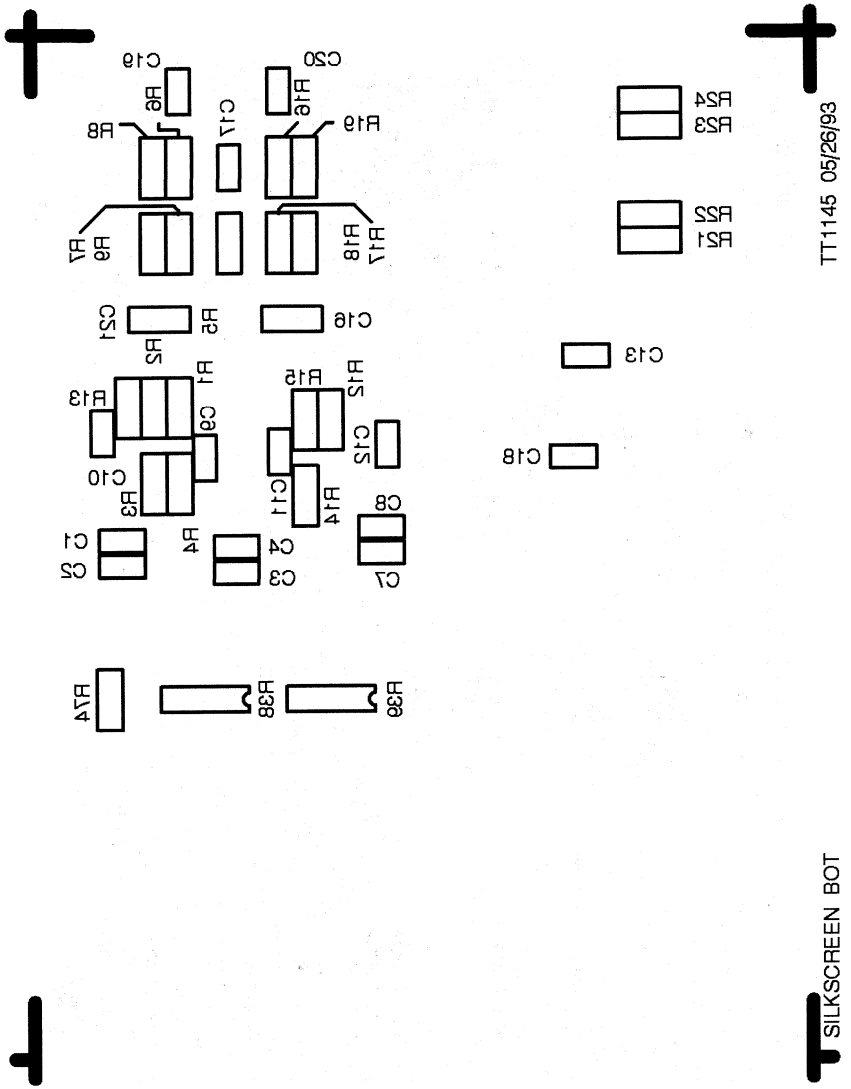
Appendix D. CY9266-F Artwork — Ground layer



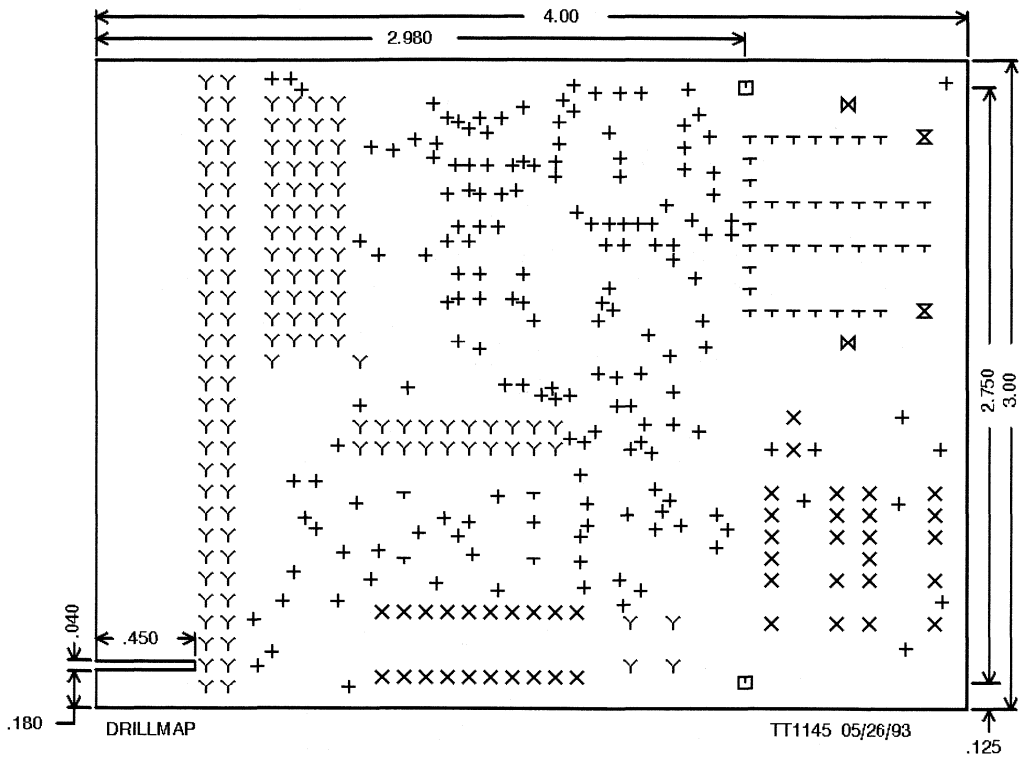
Appendix D. CY9266-F Artwork — Bottom Layer Copper



Appendix D. CY9266-F Artwork — Bottom Silkscreen



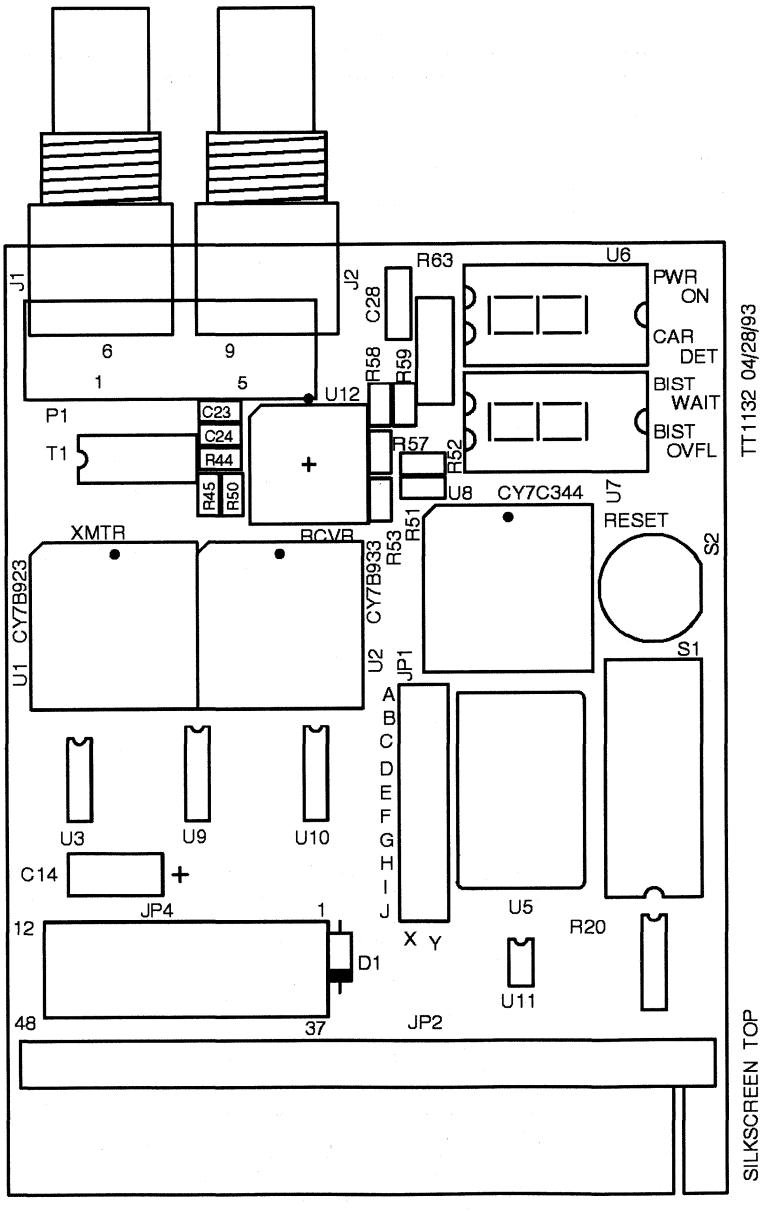
Appendix D. CY9266-F Artwork — Drill Chart



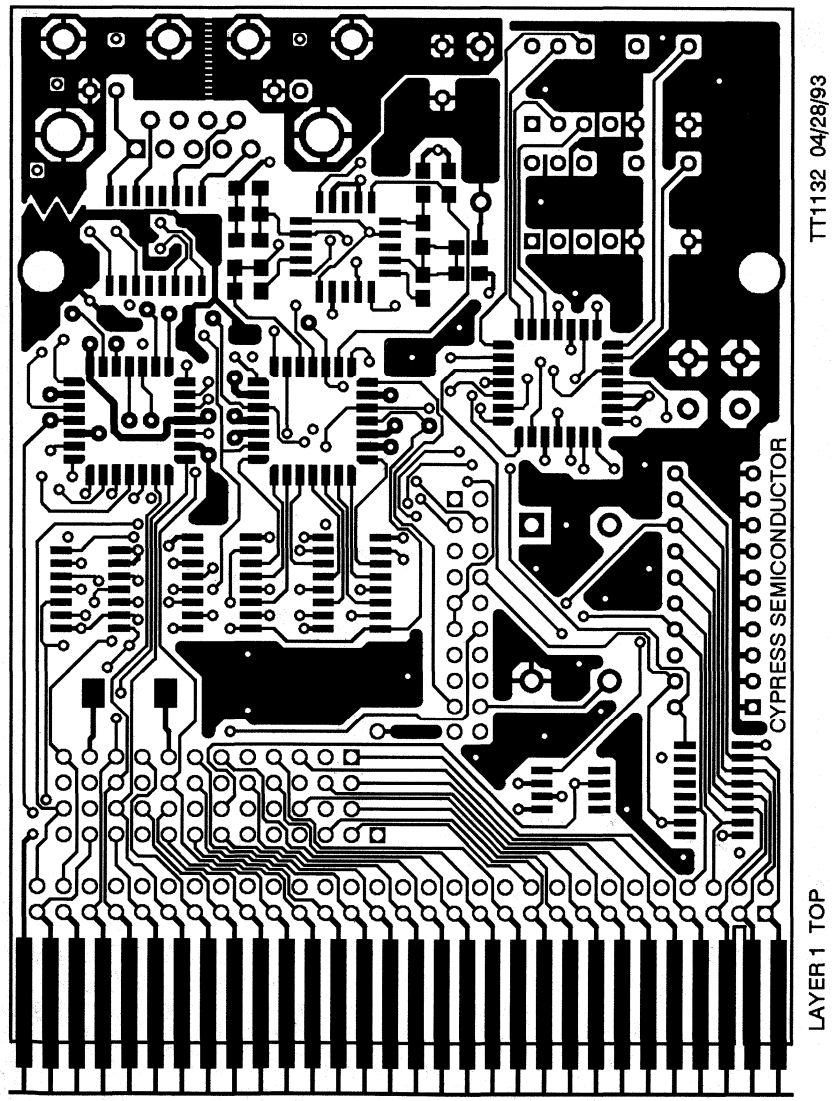
TOOL	SIZE	SYM	QTY	PLT
T01	.020	+	162	Y
T02	.032	X	44	Y
T03	.040	Y	132	Y
T04	.052	⊤	41	Y
T05	.085	⧖	4	Y
T06	.156	⊡	2	Y
T06	↓	⋈	2	Y

.200 X .100 OVAL HOLE

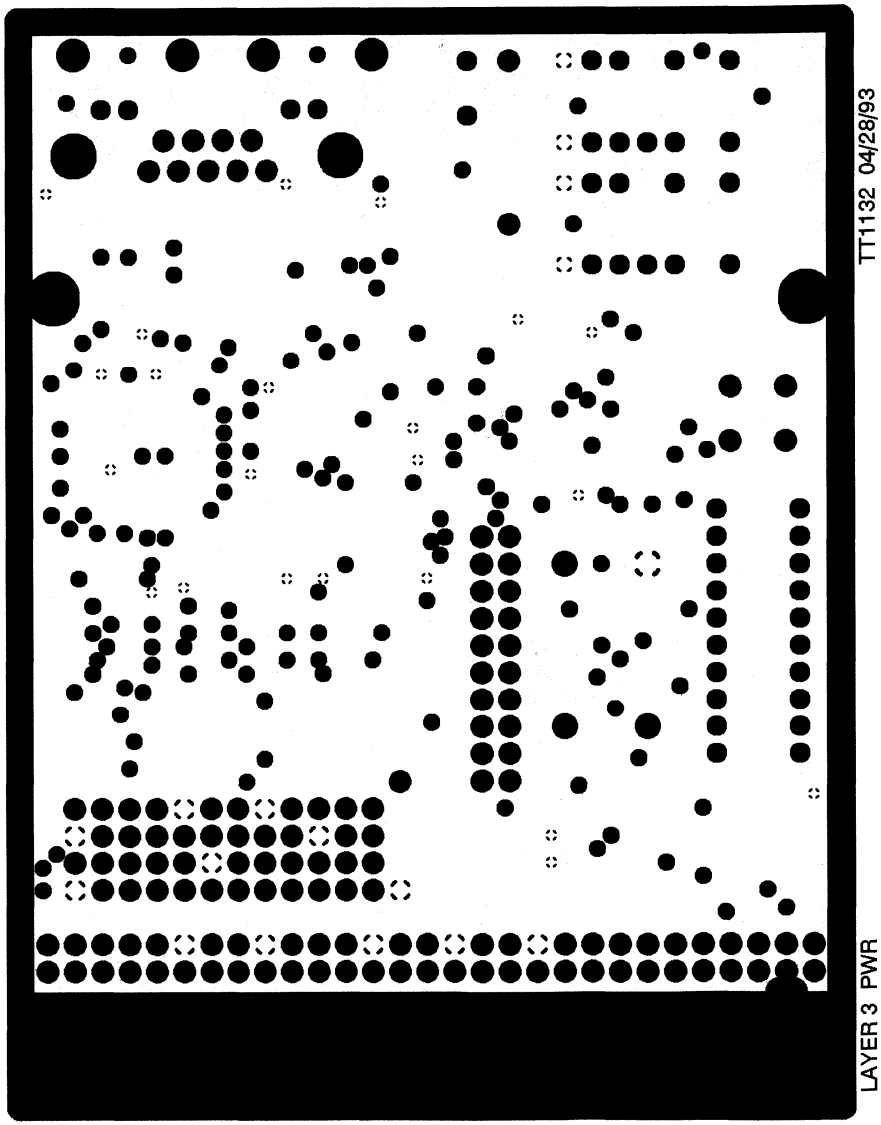
Appendix E. CY9266-C/T Artwork — Top Silkscreen



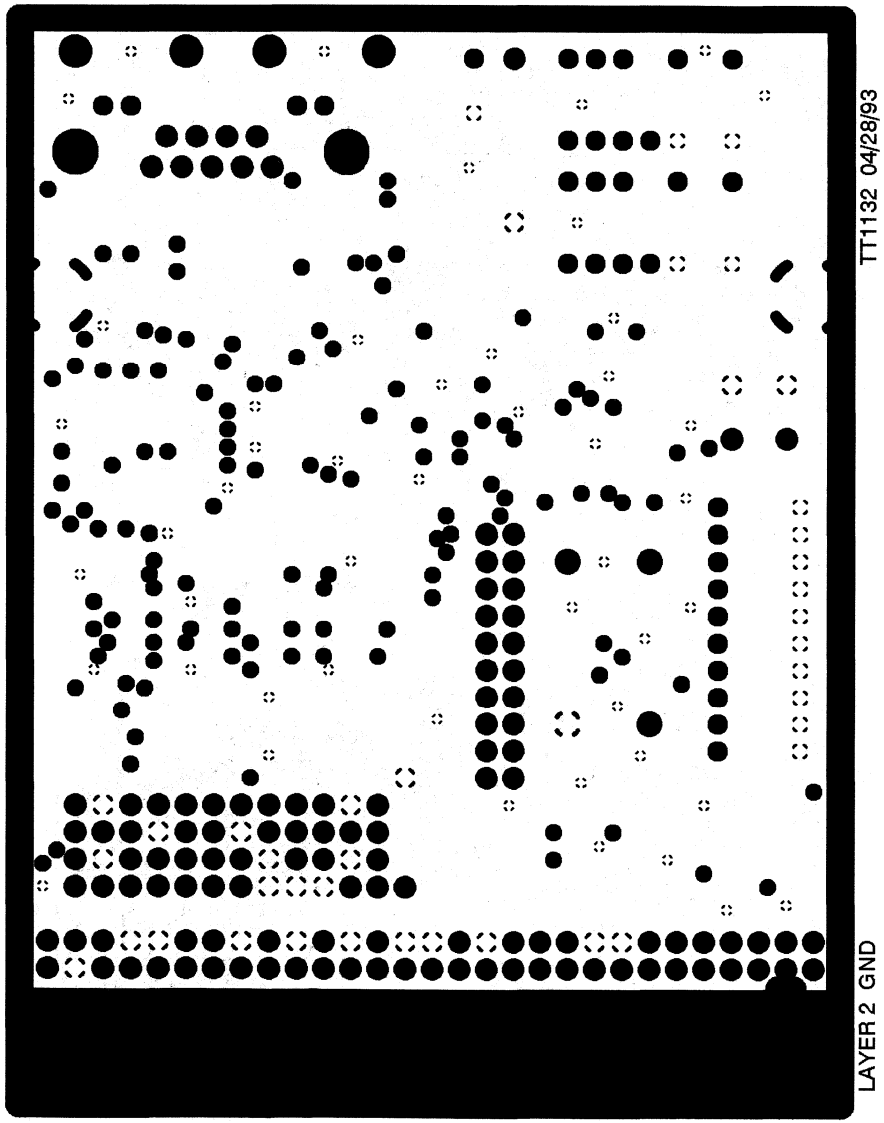
Appendix E. CY9266-C/T Artwork — Top Layer Copper



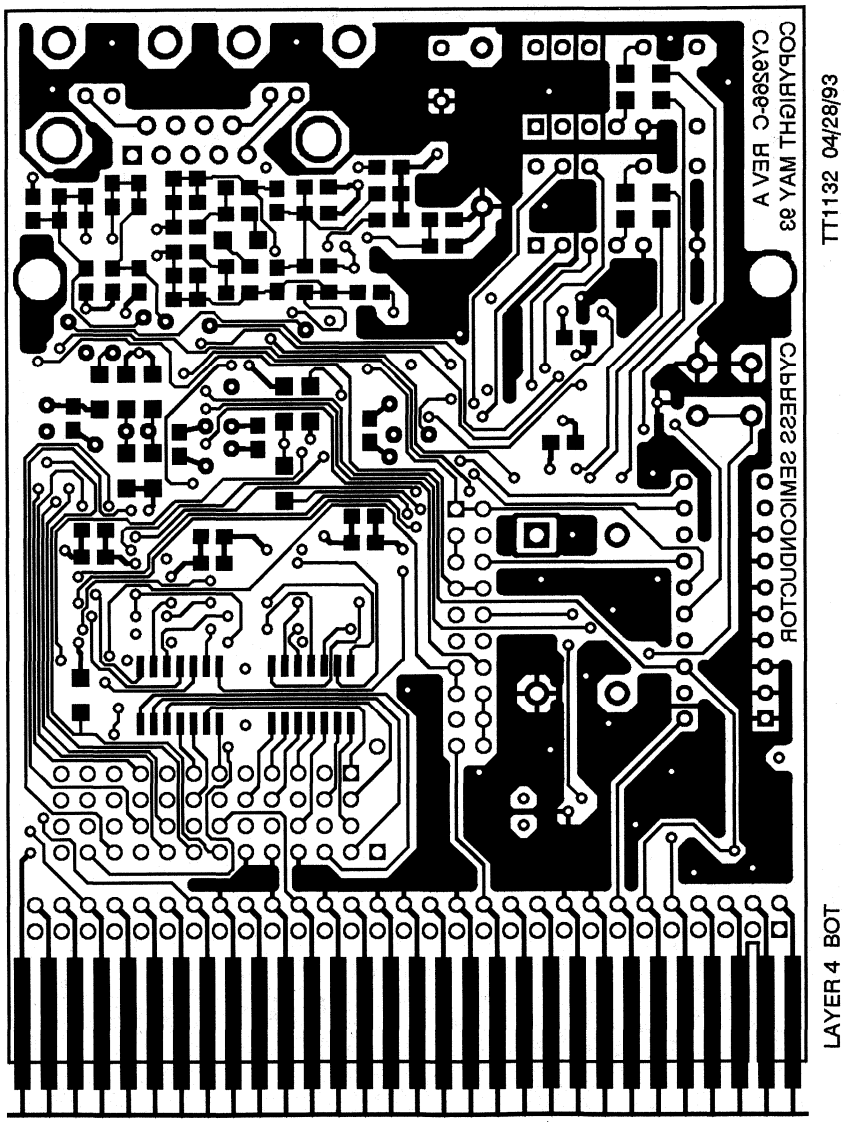
Appendix E. CY9266-C/T Artwork — Power Layer



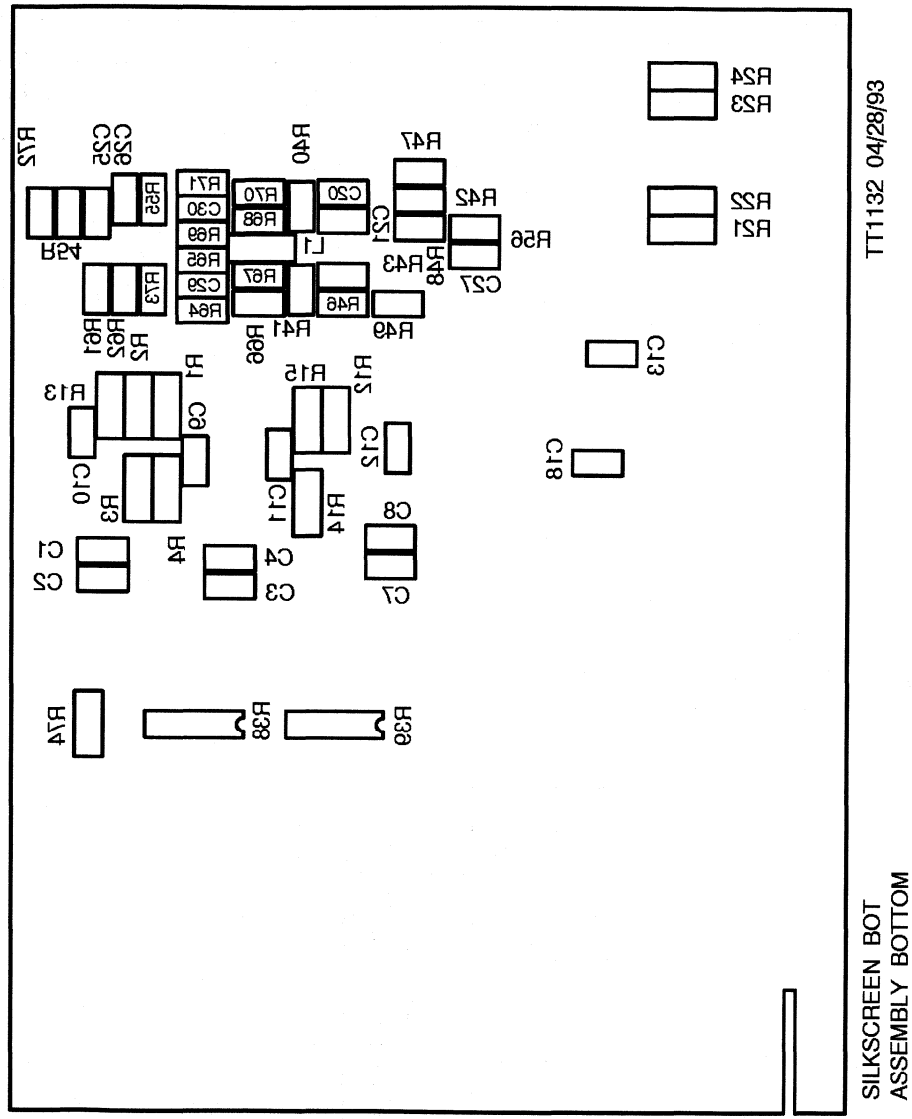
Appendix E. CY9266-C/T Artwork — Ground Layer



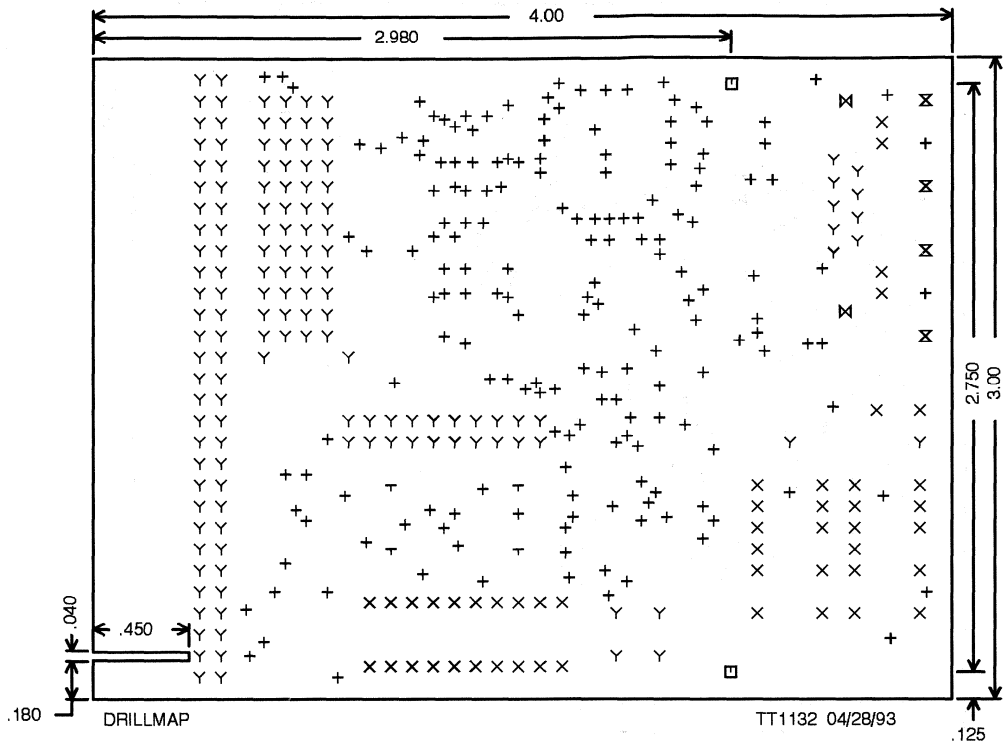
Appendix E. CY9266-C/T Artwork — Bottom Layer Copper



Appendix E. CY9266-C/T Artwork — Bottom Silkscreen



Appendix E. CY9266-C/T Artwork — Drill Chart



TOOL	SIZE	SYM	QTY	PLT
T01	.020	+	179	Y
T02	.032	×	48	Y
T03	.040	Y	143	Y
T04	.052	⊤	4	Y
T05	.080	⧖	4	Y
T06	.125	⋈	2	Y
T06	.156	⊡	2	Y

Appendix F. CY9266 Configuration Guide

Function	JP1 Jumper Settings	Switch SW1 Settings (0=on, 1 = off)									
		1	2	3	4	5	6	7	8	9	10
Xmtr BIST Enable *		0									
Xmtr BIST External *		1									
Xmtr Encode Mode *			0								
Xmtr Bypass Mode *			1								
Xmtr ENA Active *				0							
Xmtr ENA External *				1							
Xmtr ENN Active *					0						
Xmtr ENN External *					1						
Rcvr BIST Enable *						0					
Rcvr BIST External *						1					
Rcvr Decode Mode *							0				
Rcvr Bypass Mode *							1				
Rcvr Port A Selected *	CX–CY							1			
Rcvr Port B Selected *								0			
Xmtr Enabled (FOTO Off) *	EX–EY								0		
Xmtr External *									1		
Active High Carrier Detect *										0	
Active Low Carrier Detect *										1	
Active High Byte Sync *							1				1
							0				0
Active Low Byte Sync *							1				0
							0				1
Rcvr Port Select DIP	CX–CY										
Rcvr Port Select External	BX–BY										
FOTO Select DIP	EX–EY										
FOTO Select External	FX–FY										
Xmtr Clock Local Oscillator	GY–HY										
Xmtr Clock XMITCLOCK	GX–GY										
Rcvr Clock Local Oscillator	IX–IY										
Rcvr Clock XMITCLOCK	HX–IX										
Rcvr Clock EXTREFCLK	IX–JX										
OLC–266 Mode	BX-BY, FX-FY, GX-GY, HX-IX	1	1	1	0	1	1			1	1
BIST Mode w/Cable (Standalone)	CX-CY, EX-EY, GY-HY, IX-IY	0		0	1	0		1	0		
BIST Mode wo/Cable (Standalone)	CX-CY, EX-EY, GY-HY, IX-IY	0		0	1	0		0	0		

* – These SW1 controlled signals have a 5.1-kΩ pull-up resistor on the CY9266 card, and may be controlled externally when the SW1 switch is in the off position. With no attached external driver these signals go to a logic-1 state when the SW1 switch position is off.

Document #: COM0004–1193

HOTLink™ CY7B933 $\overline{\text{RDY}}$ Pin Description

This application note describes the behavior of the $\overline{\text{RDY}}$ (Ready) pin in several modes of operation: Encoded, Bypass, and BIST (Built-In Self-Test). The $\overline{\text{RDY}}$ pin indicates the status of the HOTLink™ Receiver control logic and output pins. Its function and timing are dependent on the state of the MODE, $\overline{\text{BISTEN}}$ (Built-In Self-Test Enable), and RF (Reframe) pins. The following sections describe $\overline{\text{RDY}}$ behavior in detail.

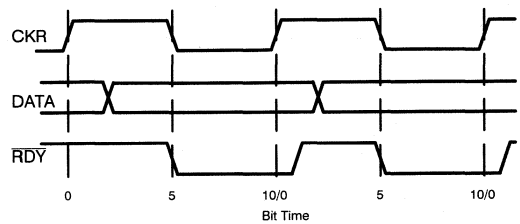


Figure 1. Normal $\overline{\text{RDY}}$ Timing

Normal $\overline{\text{RDY}}$ Timing

The HOTLink CY7B923/933 datasheet specifies signal transitions in bit times relative to the rising edge of CKR. A bit time refers to the period of the internal receiver bit rate clock. The period of the recovered byte rate clock, CKR, is ten times the bit period (bit period $t_B = t_{CKR} \div 10$). In the following discussions on timing, the rising edge of CKR is referenced as bit time zero. The next rising edge of CKR occurs ten bit times later (unless CKR stretches due to reframing). Transitions on other signal pins are defined in bit times relative to bit time zero. These timing conventions are adhered to throughout this application note.

The normal timing of the $\overline{\text{RDY}}$ pin refers to its behavior in Encoded or Bypass mode with $\overline{\text{BISTEN}}$ HIGH (Built-In Self-Test disabled). In either of these modes, $\overline{\text{RDY}}$ rests HIGH in its inactive state. During its active state, $\overline{\text{RDY}}$ transitions LOW on bit time five and then transitions HIGH on bit time one of the next clock cycle. *Figure 1* illustrates $\overline{\text{RDY}}$ timing in relation to CKR and DATA. For the exact timing margins[1] of these signals, refer to the HOTLink datasheet. In BIST mode, $\overline{\text{RDY}}$ assumes much different behavior and timing. These differences are explained later in the sections on BIST.

$\overline{\text{RDY}}$ in Encoded Mode

This section describes the operation of $\overline{\text{RDY}}$ in Encoded mode (MODE = LOW). In Encoded mode, the raw ten bit serial data is decoded in the 8B/10B decoder and then presented at the parallel output pins.

Normal Operation

The normal operation of the $\overline{\text{RDY}}$ pin in Encoded mode (MODE = LOW, RF = LOW, $\overline{\text{BISTEN}}$ = HIGH) is to signal when data is available at the parallel output pins (Q_{0-7}, SC/$\overline{\text{D}}$, RVS). $\overline{\text{RDY}}$ pulses LOW with a 60/40 (LOW/HIGH) duty cycle only when valid data is present at the output. The timing of $\overline{\text{RDY}}$ is optimized for a seamless interface to industry standard FIFOs (First-In First-Out memories). $\overline{\text{RDY}}$ does not pulse LOW in a field of SYNC (K28.5) characters; however, $\overline{\text{RDY}}$ does pulse LOW for the last K28.5 in the field or for any single K28.5. This behavior helps prevent a FIFO from filling with meaningless strings of Sync characters. *Figure 2* illustrates normal $\overline{\text{RDY}}$ behavior in Encoded mode.

Entering Framing

When the RF pin is asserted HIGH, the receiver byte framer is enabled and the $\overline{\text{RDY}}$ pin leaves nor-

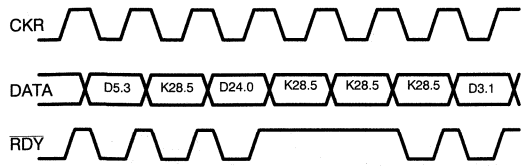


Figure 2. Normal $\overline{\text{RDY}}$ Operation in Encoded Mode

mal Encoded mode operation. The receiver latches the RF signal on the falling edge of CKR. When RF is latched HIGH, $\overline{\text{RDY}}$ is forced HIGH one bit time after the next rising edge of CKR (approximately $6t_B$ later). The exception to this is when there is a K28.5 in the framer when RF is asserted HIGH. In this case, an additional $\overline{\text{RDY}}$ pulse will occur after RF is latched HIGH. $\overline{\text{RDY}}$ will then pulse LOW when the data byte boundary is framed to an incoming SYNC character (K28.5). The latency of the receiver data pipeline and control logic insure that $\overline{\text{RDY}}$ will not pulse LOW any earlier than the fourth clock cycle after RF is latched HIGH. External framing logic should be designed to examine the $\overline{\text{RDY}}$ pin only after the 4 clock cycle delay.

After the data has been framed, $\overline{\text{RDY}}$ will assume its normal Encoded mode behavior (pulsing LOW for every character except strings of K28.5s). If RF remains HIGH, the framer still continues to frame the data to any K28.5 pattern found in the data stream. If RF is asserted HIGH for more than 2048 clock cycles, the framer converts to a double-byte framer requiring two K28.5s within four bytes for framing. The function and timing of $\overline{\text{RDY}}$, however, remain unchanged. The timing of $\overline{\text{RDY}}$ while entering framing is outlined in *Figure 3*.

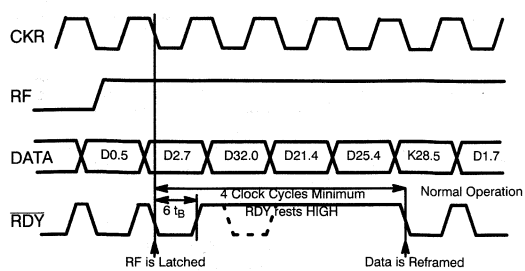


Figure 3. $\overline{\text{RDY}}$ During Framing in Encoded Mode

Leaving Framing

When RF is deasserted, the framer is disabled and the $\overline{\text{RDY}}$ pin assumes its normal Encoded mode operation. If the data was framed during the assertion of RF, $\overline{\text{RDY}}$ will have already assumed its normal operation. If the framer is disabled without having framed the data, one clock cycle will pass before $\overline{\text{RDY}}$ assumes normal operation. *Figure 4* shows the framer being disabled before the data is framed. $\overline{\text{RDY}}$ resumes normal operation one cycle after RF is latched LOW.

$\overline{\text{RDY}}$ in Bypass Mode

This sections describes the operation of $\overline{\text{RDY}}$ in Bypass mode (MODE = HIGH). In Bypass mode, the raw ten bit serial data bypasses the 8B/10B decoder and is presented at the parallel output pins.

Normal Operation

The normal operation of the $\overline{\text{RDY}}$ pin in Bypass mode (MODE=HIGH, RF=LOW, $\overline{\text{BISTEN}}$=HIGH) is to signal when a data pattern matching K28.5 character is present on the receiver's parallel output pins (Q_{a-j}). $\overline{\text{RDY}}$ will remain HIGH during all other data patterns. *Figure 5* shows an example of $\overline{\text{RDY}}$ in Bypass mode.

Entering Framing

The behavior of $\overline{\text{RDY}}$ while entering framing from Bypass mode is very similar to entering from Encoded mode. When RF is latched HIGH, $\overline{\text{RDY}}$ leaves normal Bypass mode operation and is forced HIGH one bit time after the next rising edge of CKR. When the framer is enabled, a LOW pulse on

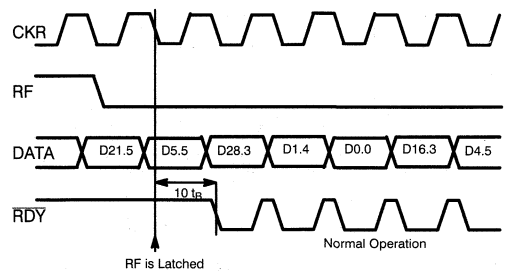


Figure 4. $\overline{\text{RDY}}$ While Leaving Framing

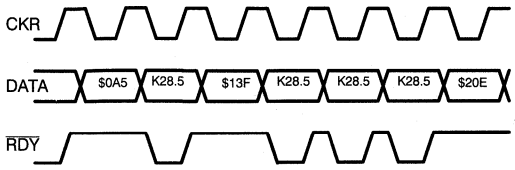


Figure 5. Normal $\overline{RDY}$ Operation in Bypass Mode

$\overline{RDY}$ indicates that the serial data has been framed to an incoming SYNC character (K28.5). The latency of the data pipeline and control logic insure that $\overline{RDY}$ does not pulse LOW any earlier than the fourth clock cycle after RF is latched HIGH. External framing logic should be designed to examine the $\overline{RDY}$ pin only after the 4 clock cycle delay. After the data has been framed, $\overline{RDY}$ assumes its normal Bypass mode behavior (pulsing LOW only on K28.5 characters). While RF is HIGH, the framer continues to frame the data to any K28.5 pattern in the data stream. The timing of $\overline{RDY}$ while entering framing from Bypass Mode is outlined in *Figure 6*.

Leaving Framing

When RF is deasserted, the framer is disabled and the $\overline{RDY}$ pin assumes normal Bypass mode behavior. If the data was framed during the assertion of RF, $\overline{RDY}$ will have already assumed its normal operation. If Reframe is exited without having framed the data, one clock cycle passes before $\overline{RDY}$ assumes normal operation. *Figure 7* shows RF deasserted before the serial data has been framed.

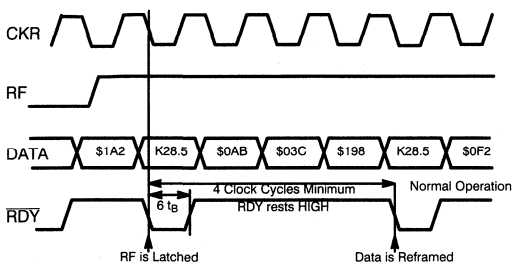


Figure 6. $\overline{RDY}$ During Framing in Bypass Mode

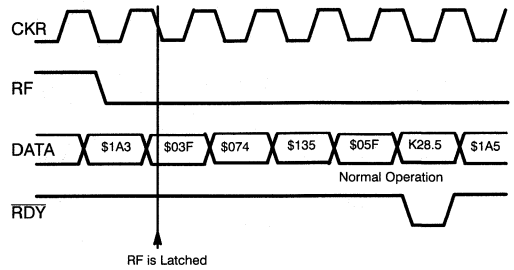


Figure 7. $\overline{RDY}$ While Leaving Framing

$\overline{RDY}$ and CKR Stretching

During framing (RF = HIGH), $\overline{RDY}$ and CKR may stretch as the byte boundary is synchronized to an incoming K28.5 character. If a K28.5 pattern is found in the serial data stream that is not aligned with the current byte boundary, the framer will realign the phase of CKR so that the receiver shift register properly deserializes the K28.5 character (and the following data). The HIGH or LOW phase of CKR and $\overline{RDY}$ will be stretched so that these signals maintain proper byte synchronization with the data. *Figure 8* shows $\overline{RDY}$ and CKR being stretched during framing due to a K28.5 character in the data stream. In this example, RF is held HIGH so that the framer remains enabled after has $\overline{RDY}$ assumed its normal operation according to the MODE pin (Encoded mode). The period of $\overline{RDY}$ and CKR may stretch up to a length of 19 bit times depending on the position of the K28.5 character relative to the old byte boundary. Note that the K28.5 character

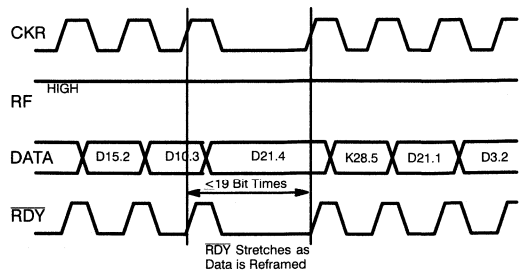


Figure 8. $\overline{RDY}$ and CKR Stretching (Encoded Mode)

comes out of the receiver one cycle after the CKR and $\overline{\text{RDY}}$ stretch due to the receiver pipeline.

$\overline{\text{RDY}}$ in BIST Mode

The Built-In Self-Test (BIST) feature provides a simple but exhaustive method for testing the integrity of the physical layer. BIST Mode is entered by asserting the $\overline{\text{BISTEN}}$ pin LOW in either Encoded or Bypass mode. $\overline{\text{RDY}}$ has two normal modes of operation while in BIST. $\overline{\text{RDY}}$ initially rests HIGH when BIST is entered, signaling that the BIST logic has not started checking the received data. When a valid start of BIST sequence is received, the $\overline{\text{RDY}}$ pin will rest LOW, indicating that BIST checking is in progress. The timing of these transitions is discussed below. For more information on BIST, consult the "HOTLink Built-In Self-Test" application note.

Entering BIST Mode

BIST mode is entered by asserting $\overline{\text{BISTEN}}$ LOW. $\overline{\text{BISTEN}}$ is latched into the receiver on the falling edge of CKR. When $\overline{\text{BISTEN}}$ is latched LOW, $\overline{\text{RDY}}$ leaves its current mode of operation (Encoded or Bypass) and is asserted LOW one CKR cycle. On bit time one of the next clock cycle, $\overline{\text{RDY}}$ is forced HIGH. The BIST logic will check the incoming data stream for the start of BIST sequence (D1.0 followed by D0.0). $\overline{\text{RDY}}$ rests HIGH while the BIST logic waits for this sequence. *Figure 9* shows the behavior of $\overline{\text{RDY}}$ when $\overline{\text{BISTEN}}$ is asserted LOW.

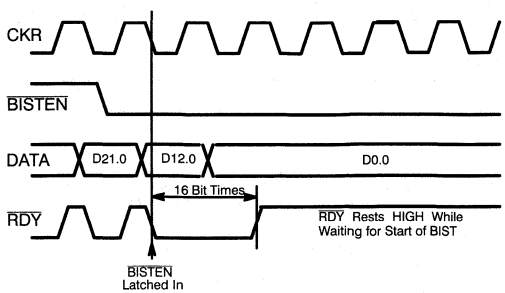


Figure 9. $\overline{\text{RDY}}$ while Entering BIST

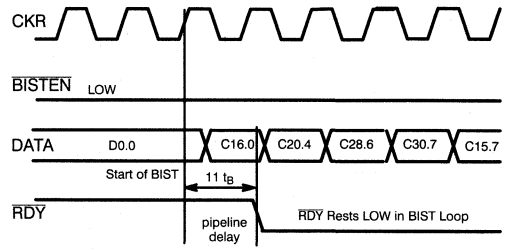


Figure 10. $\overline{\text{RDY}}$ at Start of BIST

Start of BIST

When the start of BIST pattern is found, $\overline{\text{RDY}}$ will transition LOW one bit time after CKR rises. Due to the pipeline nature of the receiver, there is a one cycle delay from when start of BIST is detected and when $\overline{\text{RDY}}$ is asserted LOW. $\overline{\text{RDY}}$ will remain LOW for the duration of BIST except to pulse HIGH for one clock cycle each time a BIST Loop starts (once every 511 bytes). *Figure 10* shows the $\overline{\text{RDY}}$ pin during the start of BIST sequence.

BIST Loop

Figure 11 shows $\overline{\text{RDY}}$ behavior once BIST checking has begun. $\overline{\text{RDY}}$ rests LOW and pulses HIGH at the start of each new BIST loop. During this pulse, $\overline{\text{RDY}}$ rises on bit time one and then falls one cycle later on bit time one. This pulse is useful for counting the number of BIST loops completed.

Leaving BIST

BIST is disabled by asserting $\overline{\text{BISTEN}}$ HIGH. $\overline{\text{RDY}}$ will assume the behavior dictated by the MODE pin (Encoded or Bypass) one clock cycle after $\overline{\text{BISTEN}}$

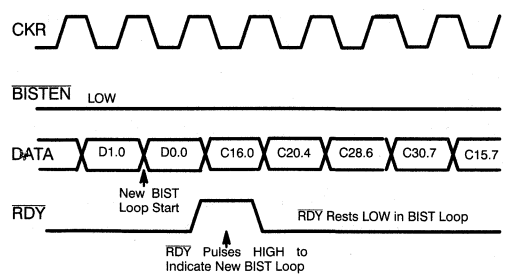


Figure 11. $\overline{\text{RDY}}$ in BIST Loop

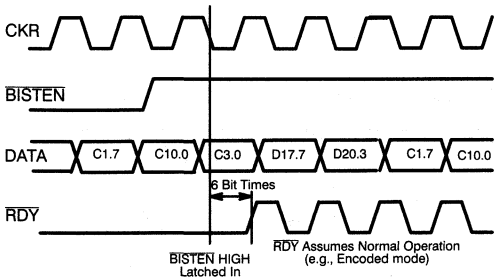


Figure 12. $\overline{RDY}$ While Leaving BIST

is latched HIGH. *Figure 12* shows the $\overline{RDY}$ pin while leaving BIST Mode.

Framing While in BIST

Framing may be performed while in BIST Mode. The BIST pattern includes one alias K28.5 and several instances of byte aligned characters. If the framer is enabled (RF = HIGH), the data byte boundaries are aligned to any incoming K28.5 characters found in the serial data. $\overline{RDY}$ ceases its normal BIST behavior and rests HIGH while the framer waits for a K28.5 character. The timing for the $\overline{RDY}$ pin to be forced HIGH is the same as the timing discussed in the preceding sections on entering framing (i.e., $6t_B$ after RF is latched HIGH). When a K28.5 character is found, $\overline{RDY}$ will pulse LOW for one clock cycle. During this cycle, $\overline{RDY}$ falls on bit time five and then rises on bit time one of the next clock cycle. $\overline{RDY}$ then resumes its normal BIST behavior after one more clock cycle (see *Figure 13* and *Figure 14*).

Figure 13 shows RF asserted HIGH (framer enabled) while BIST is in the middle of checking the data. $\overline{RDY}$ initially rests LOW and then transitions HIGH when the framer is enabled. When a K28.5 character is found, $\overline{RDY}$ pulses LOW and then rests HIGH again. One cycle later, $\overline{RDY}$ transitions LOW as it resumes its normal BIST behavior (resting LOW during BIST).

Figure 14 shows $\overline{RDY}$ behavior while BIST is waiting for the start of BIST sequence. Initially, $\overline{RDY}$ rests HIGH while waiting for the start of BIST sequence. When RF is asserted HIGH, the framer checks the serial data for a K28.5 character. $\overline{RDY}$ pulses LOW when a K28.5 is encountered and then returns HIGH. $\overline{RDY}$ then returns to its normal mode of operation (resting HIGH until start of BIST is received).

If RF is deasserted before a K28.5 is found by the framer, $\overline{RDY}$ will resume its normal BIST behavior on the next clock cycle.

Enabling the framer while in BIST Mode may cause the BIST data to become temporarily misaligned. If the enabled framer encounters the alias K28.5 character in the BIST data stream, the BIST data will be aligned to the incorrect byte boundary. This will result in a large number of errors reported on the RVS (Receive Violation Symbol) pin until the data is framed again to one of the properly aligned K28.5's. If RF is asserted HIGH for less than 2048 clock cycles, the BIST data will be misaligned each time the alias K28.5 is found (once per BIST loop). If RF is asserted for more than 2048 clock cycles (>4

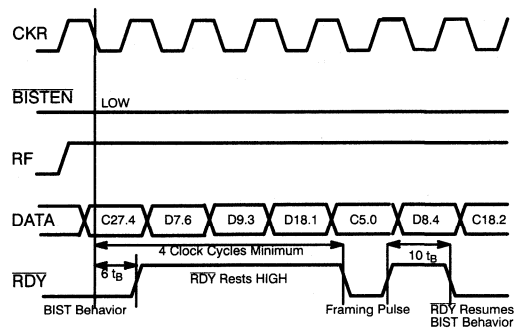


Figure 13. $\overline{RDY}$ While Framing in BIST

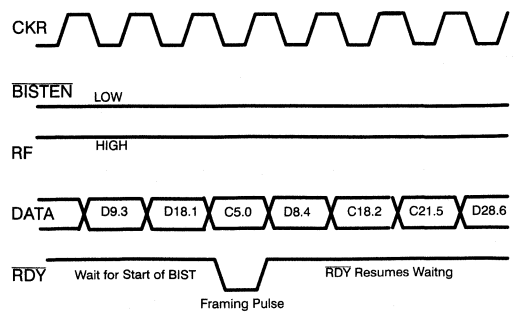


Figure 14. $\overline{RDY}$ While Framing in BIST

BIST Loops), the double-byte framer will be enabled, and the framer will no longer frame the data to the alias K28.5 character.

Conclusion

The Receiver $\overline{\text{RDY}}$ pin indicates the status of the control logic and data pins in various modes of operation. The behavior and timing of the $\overline{\text{RDY}}$ pin have been optimized for easy integration with interface control logic and FIFO memories. The detailed information contained in this application note should serve as an aid when integrating the $\overline{\text{RDY}}$ pin into the interface logic.

Notes

1. Datasheet timing parameters that are defined in terms of bit times (t_B) include additional timing margin to account for internal buffer and routing delays and output load (e.g., $t_A = 2t_B$ +4/−2 ns).

Document #: COM0005−1293

HOTLink is a trademark of Cypress Semiconductor Corporation.

HOTLink™ CY7B9331 OLC Receiver

Introduction

This application note describes the special characteristics of the HOTLink™ CY7B9331 OLC Receiver. The characteristics described in this document are intended to make the HOTLink OLC Receiver exactly compatible with the functionality and timing of the IBM Optical Link Card (OLC−266). *Figure 1* illustrates how the CY7B9331 creates a seamless HOTLink−OLC interface.

The CY7B9331 OLC Receiver is a derivative of the CY7B933 Receiver. The names and behavior of several OLC Receiver pins have been modified for the CY7B9331 to make them compatible with the OLC system interface. The behavior of these OLC specific pins is described in detail in the following sections.

LOOP Pin

The polarity and function of the LOOP pin on the CY7B9331 is equivalent to the OLC Loopback Enable signal. When LOOP is HIGH, the INB serial port is selected as the local loopback data source. When LOOP is LOW, the INA serial port is selected as the external data source. Internally, the LOOP signal is connected to a multiplexer that selects between the two data sources. *Figure 1* shows how the local loopback connection would be implemented in a typical OLC compatible system.

STATUS Pin

The STATUS pin matches the polarity of the OLC Link Status signal. The STATUS pin translates the ECL logic level of the SI (Status In) pin to its in-

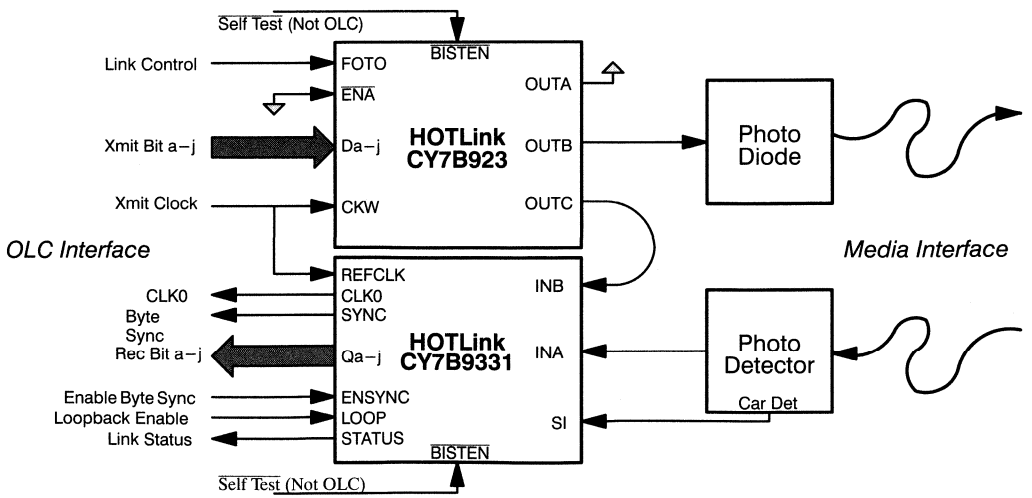


Figure 1. HOTLink−OLC System Interface

verted TTL equivalent. The SI pin is normally connected to the carrier detect signal of an optical receiver module. End-fire style optical modules only provide a single-ended active HIGH ECL carrier detect (asserted HIGH when a carrier is present); however, the OLC *Link Status* signal should be asserted HIGH (TTL) in the absence of a carrier. The required inversion and translation is provided by the SI/STATUS pins.

CLK0 Pin

The CLK0 pin generates the OLC CLK0 signal. CLK0 is the byte rate clock recovered from the the serial data stream. The frequency of CLK0 is one tenth the serial bit rate of the incoming data.

ENSYNC Pin

The ENSYNC pin interfaces to the OLC Enable Byte Sync signal. When asserted HIGH, the ENSYNC pin enables the receiver byte framer. The byte framer, when enabled, aligns the receiver parallel data path with the boundary of K28.5 characters detected in the serial data stream. The phase of CLK0 is adjusted to maintain proper synchronization with the parallel byte boundary.

SYNC Pin

The behavior and timing of the SYNC pin are tailored to match the OLC electrical interface requirements of the Byte Sync signal. No external glue logic is required. The details of the behavior and timing of the SYNC pin are discussed in the following sections.

Normal SYNC Pin Timing

The HOTLink CY7B923/933 data sheet specifies signal transitions in bit-times relative to the rising edge of CLK0. A bit-time refers to the period of the recovered internal bit rate clock. The period of the recovered byte rate clock, CLK0, is ten times the bit period (bit period $t_B = t_{CKR} \div 10$). In the following discussions on timing, the rising edge of CLK0 is referenced as bit-time zero. The next rising edge of CKR occurs ten bit-times later (unless CLK0 stretches due to reframing). Transitions on other signal pins are defined in bit-times relative to bit-time zero. These timing conventions are used throughout this application note.

The SYNC pin makes transitions on bit-time two simultaneous with the Qa–j (DATA) transitions. SYNC transitions are specified by the CY7B9331 Receiver data sheet data access timing parameter, t_A ($t_A = 2t_B + 4$ ns max. and $2t_B - 2$ ns min.)[1]. Access time is measured from the rising edge of CLK0 (bit-time zero). The SYNC and DATA timing meet the OLC timing specifications. The following sections explain under what conditions SYNC will make transitions. *Figure 2* illustrates SYNC timing in relation to CLK0 and DATA.

SYNC in Bypass Mode

This section describes the behavior of SYNC in Bypass Mode (MODE = HIGH). This is the mode of operation for an OLC compatible system interface. In Bypass Mode, the raw ten bit serial data bypasses the 8B/10B decoder and is presented at the parallel data output pins. In this mode of operation, the SYNC pin behavior matches the OLC Byte Sync signal.

Normal Operation

The normal operation of SYNC in Bypass Mode (MODE = HIGH, ENSYNC = LOW, $\overline{\text{BISTEN}}$ = HIGH) is to remain LOW at all times unless the byte framer is enabled. *Figure 3* shows an example of SYNC in Bypass Mode.

Entering Framing

When ENSYNC is asserted HIGH, the byte framer is enabled and the behavior of the SYNC pin

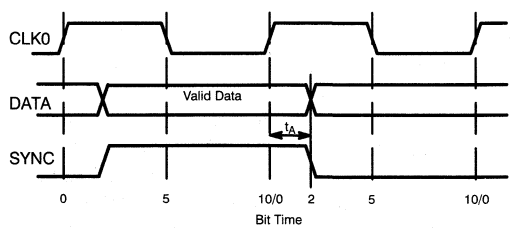


Figure 2. Normal SYNC Timing

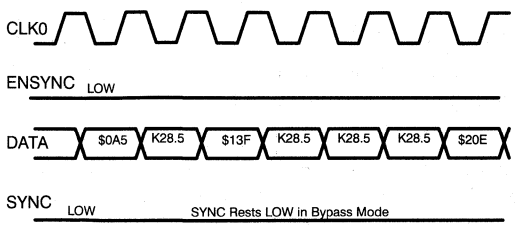


Figure 3. Normal SYNC Operation in Bypass Mode

changes. The ENSYNC signal is latched on the falling edge of CLK0. When ENSYNC is latched HIGH, SYNC continues to rest LOW while the framer searches for a K28.5 character. When the framer detects a K28.5 character, the SYNC pin is asserted HIGH for one cycle when the K28.5 appears at the output pins. This pulse on SYNC indicates that the serial data has been framed to the incoming synchronization character (K28.5). The latency of the data pipeline and control logic insure that SYNC framing signal does not pulse HIGH any earlier than the third clock cycle after ENSYNC is latched HIGH. The HIGH and LOW transitions of SYNC during framing are bit synchronous with the Qa–j transitions (bit-time two after CLK0 rises).

An exception to this behavior occurs if the framer is enabled when a K28.5 is present in the shift register. SYNC will pulse HIGH for one CLK0 cycle during the cycle after RF is asserted HIGH. This pulse should not be interpreted as a signal that the data has been framed. External controllers should be designed to ignore this pulse and should wait for three clock cycles after RF is latched HIGH before examining the SYNC pin. The timing and function of SYNC while entering framing are outlined in *Figure 4*.

After the data has been framed, SYNC will assume its normal Bypass Mode operation (resting LOW) while the framer continues to frame the data to any K28.5 pattern in the data stream. When another K28.5 character is found, SYNC will be asserted HIGH for the clock cycle when the framed K28.5 is present at the parallel output. If ENSYNC is asserted HIGH for more than 2048 bytes, the framer converts to a double-byte framer requiring two K28.5s within four bytes for framing, but the function of SYNC remains unchanged.

Leaving Framing

When ENSYNC is deasserted, the framer is disabled and the SYNC pin assumes normal operation (resting LOW). The SYNC pin assumes normal operation on the clock cycle after ENSYNC is latched LOW. If the data was reframed to a K28.5 during the assertion of ENSYNC HIGH, SYNC will have already assumed its normal operation. *Figure 5* illustrates SYNC when ENSYNC is deasserted just after the data has been reframed.

SYNC Pin in Encoded Mode

This section describes the operation of the SYNC pin in Encoded Mode (MODE = LOW). Unlike the unencoded OLC interface hardware, the HOTLink Transmitter/Receiver has a built in 8B/10B Encoder/Decoder which can be enabled in place of any external coding logic. In Encoded Mode, the Receiver decoder converts the encoded ten bit serial data into its eight bit decoded equivalent. The re-

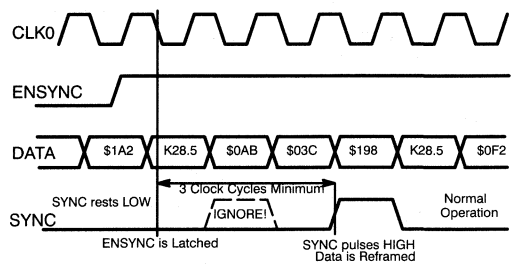


Figure 4. SYNC During Framing in Bypass Mode

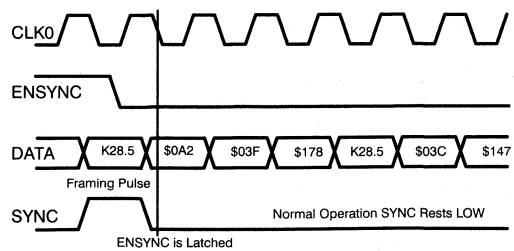


Figure 5. SYNC While Leaving Framing

sulting byte is then presented at the parallel output pins.

Normal Operation

The normal operation of the SYNC pin in Encoded Mode (MODE = LOW, ENSYNC = LOW, $\overline{\text{BIS-TEN}}$ = HIGH) is to signal when data is available at the parallel output pins (Q0–7, SC/$\overline{\text{D}}$, RVS). SYNC is asserted HIGH when user data is present at the parallel output pins. In a field of idle characters (K28.5s), SYNC will rest LOW; however, SYNC will be asserted HIGH for the last K28.5 in the field or for any single K28.5. The SYNC pin makes transitions simultaneous with the Qa–j outputs. *Figure 6* illustrates normal SYNC behavior in Encoded Mode.

Entering Framing

The receiver byte framer is enabled by asserting the ENSYNC pin HIGH. When ENSYNC is latched HIGH, SYNC leaves normal Encoded Mode operation and rests LOW. SYNC will then be asserted HIGH for one clock cycle when the data is framed to an incoming synchronization character (K28.5). The receiver latches the ENSYNC signal on the falling edge of CLK0. After ENSYNC is latched HIGH, SYNC is forced LOW two bit-times after the next rising edge of CLK0 (approximately $7t_B$ later). The exception to this rule occurs when there is a K28.5 in the shifter when RF is latched HIGH. In this case, SYNC will be asserted LOW after an additional clock cycle of delay.

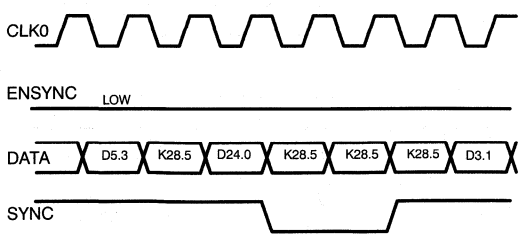


Figure 6. Normal SYNC Operation in Encoded Mode

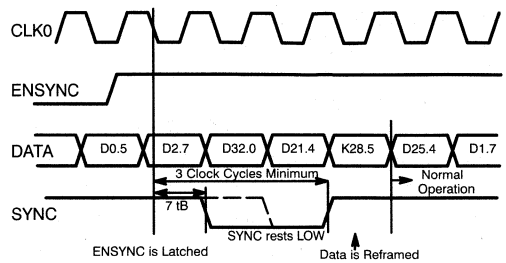


Figure 7. SYNC During Framing in Encoded Mode

The latency of the receiver data pipeline and control logic insure that the SYNC framing signal will not pulse HIGH any earlier than the third CLK0 clock cycle after ENSYNC is latched HIGH. External framing controllers should be designed to examine the SYNC pin after the third CLK0 cycle. After the data has been framed, SYNC will assume its normal operation (HIGH for any character except strings of K28.5s). If SYNC remains HIGH, the framer will continue to frame the data to any K28.5 pattern found in the data stream. The timing of these events is outlined in *Figure 7*.

Leaving Framing

When ENSYNC is deasserted, the framer is disabled and the SYNC pin assumes normal operation. If the data was framed during the assertion of ENSYNC, SYNC will have already assumed its normal operation. If the framer is disabled without having framed the data, one clock cycle will pass before SYNC assumes normal operation. *Figure 8* shows the framer being disabled before the data is framed. SYNC resumes normal operation one cycle after ENSYNC is latched LOW.

SYNC and CLK0 Stretching

During framing (ENSYNC = HIGH), CLK0 and SYNC may stretch as the byte boundary is synchronized to an incoming K28.5 character. If a K28.5 pattern is found in the serial data stream that is not aligned with the current byte boundary, the framer will realign the phase of CLK0 so that the Receiver shift register properly deserializes the K28.5 charac-

ter (and the following data). The HIGH or LOW phase of CLK0 will be stretched so that the clock maintains proper byte synchronization with the data. SYNC will maintain its synchronization with the stretched clock. *Figure 9* shows CLK0 being stretched during framing due to a K28.5 character in the data stream. In this example, ENSYNC is held HIGH so that the framer remains enabled while SYNC has already assumed normal operation (i.e., a K28.5 had been found previously in the data). Note that the K28.5 appears at the DATA outputs one cycle after CLK0 stretches due to the pipelined data path. The period CLK0 may stretch up to a length of 19 bit-times depending on the position of the K28.5 character relative to the old byte boundary.

SYNC in BIST Mode

The Built-In Self-Test (BIST) feature provides a simple but exhaustive method for testing the integrity of the physical layer. A built-in diagnostic test that checks the functionality of the physical layer is not included in the OLC specification; however, the HOTLink BIST feature can be added to an OLC-compatible system with little cost and no impact on OLC functionality or timing. For more information on BIST consult the HOTLink Built-In Self-Test application note.

Receiver BIST Mode is entered by asserting the $\overline{\text{BISTEN}}$ pin LOW in either Encoded or Bypass Mode. SYNC has two normal modes of operation while in BIST. SYNC initially rests LOW when BIST is entered, signaling that the BIST logic has not started checking the received data. When a valid start of BIST sequence is received, the SYNC pin rests HIGH indicating that BIST checking is in progress. The timing of these transitions is discussed in the following sections. For more information on BIST see the "HOTLink Built-In Self-Test" application note.

Entering BIST Mode

BIST Mode is entered by asserting $\overline{\text{BISTEN}}$ LOW. $\overline{\text{BISTEN}}$ is latched into the Receiver on the falling edge of CLK0. When $\overline{\text{BISTEN}}$ LOW is latched, SYNC leaves its current mode of operation (Encoded or Bypass) and pulses HIGH for one CLK0 cycle. On bit-time two of the next clock cycle, SYNC will be forced LOW. The BIST logic then checks the incoming data stream for the start of the BIST sequence (D1.0 followed by D0.0). SYNC rests LOW while the BIST logic waits for this sequence. *Figure 10* shows the behavior of SYNC when $\overline{\text{BISTEN}}$ is asserted LOW from Bypass Mode. Note that the data at the parallel outputs reflects the encoded BIST data being received.

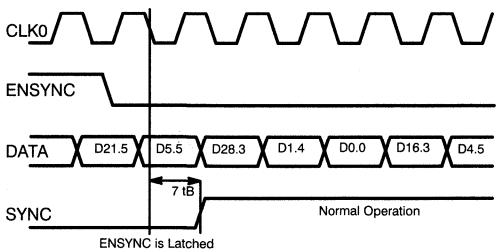


Figure 8. SYNC While Leaving Framing

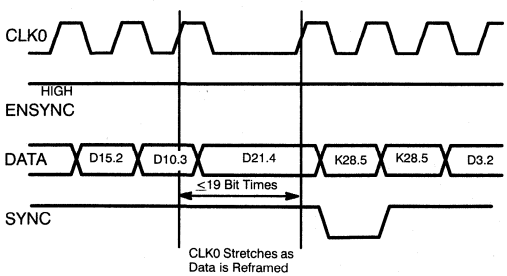


Figure 9. SYNC and CLK0 Stretching

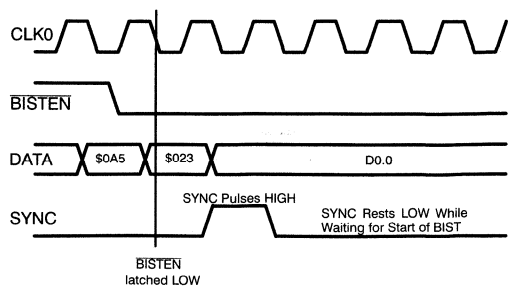


Figure 10. RDY while Entering BIST

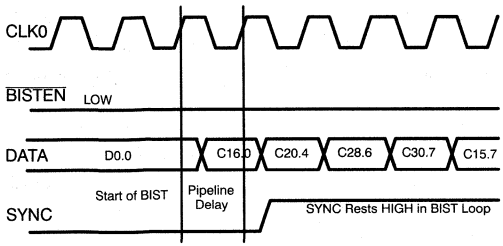


Figure 11. SYNC at Start of BIST

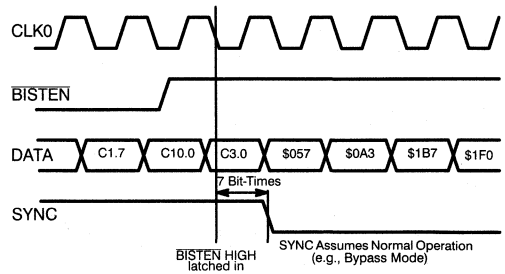


Figure 13. SYNC While Leaving BIST

Start of BIST

When the start of a BIST pattern is found, a one-cycle pipeline delay will occur and then SYNC will transition HIGH on bit-time two of the next CLK0 cycle. SYNC will remain HIGH for the duration of BIST except to pulse LOW for one clock cycle each time a BIST Loop starts (once every 511 bytes). *Figure 11* shows the SYNC pin during the start of the BIST sequence.

BIST Loop

Figure 12 shows the behavior SYNC once BIST checking has begun. SYNC rests HIGH while BIST checking is in progress and pulses LOW at the start of each new BIST loop. During this pulse, SYNC falls on bit-time two and then rises one CLK0 cycle later on bit-time two.

Leaving BIST

BIST is disabled by deasserting $\overline{\text{BISTEN}}$ HIGH. SYNC then assumes the behavior dictated by the MODE pin (Encoded or Bypass) on the clock cycle after $\overline{\text{BISTEN}}$ HIGH is latched in. *Figure 13* shows the SYNC pin timing while leaving BIST Mode and entering its normal Bypass Mode behavior.

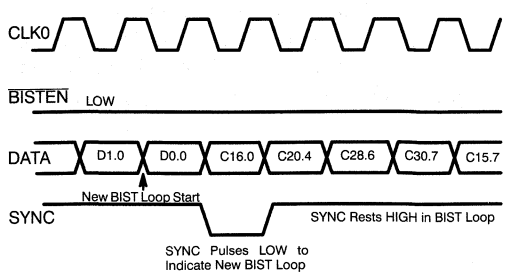


Figure 12. SYNC in BIST Loop

Reframing While in BIST

Framing may be performed while in BIST Mode. The BIST pattern includes one alias K28.5 character and multiple instances of byte aligned characters. If the framer is enabled (ENSYNC = HIGH), the data byte boundaries are aligned to any incoming K28.5 characters found in the serial data. SYNC ceases its normal BIST behavior and rests LOW while the framer waits for a K28.5 character. The timing for the SYNC pin to transition LOW is the same as the timing discussed in the preceding sections on entering framing (i.e., $7t_B$ after ENSYNC is latched HIGH, unless a K28.5 is in the shifter). When a K28.5 character is found, SYNC will pulse HIGH for one clock cycle. During this cycle, SYNC rises on bit-time two and then falls on bit-time two of the next CLK0 clock cycle. SYNC then resumes its normal BIST behavior after one more clock cycle, but the framer will continue to frame to K28.5s until it is disabled (see *Figure 14* and *Figure 15*).

Figure 14 shows ENSYNC asserting HIGH (enable framer) while BIST is in the middle of checking the data. SYNC initially rests HIGH and then transitions LOW when the framer is enabled. When a K28.5 character is found, SYNC pulses HIGH. During the next CLK0 cycle, SYNC resumes its normal BIST behavior (resting HIGH during BIST).

Figure 15 shows SYNC behavior while BIST is waiting for the start of the BIST sequence. Initially, SYNC rests LOW while waiting for the start of BIST sequence. When ENSYNC is asserted HIGH, the framer checks the serial data for a K28.5 character.

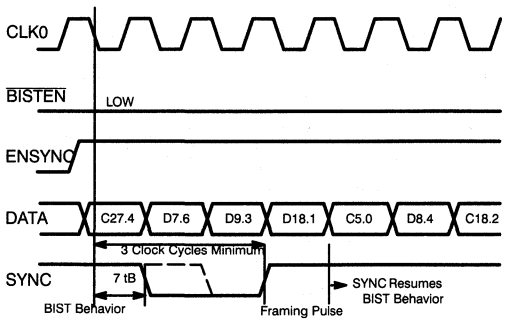


Figure 14. SYNC While Framing in BIST

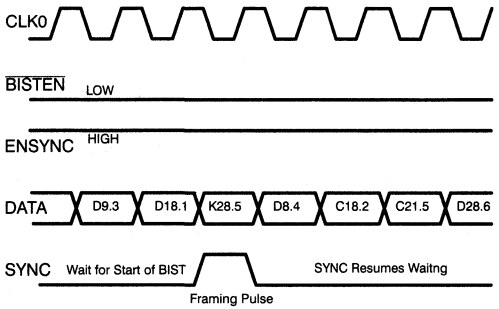


Figure 15. SYNC While Framing in BIST

SYNC pulses HIGH when a K28.5 is encountered and then returns LOW. SYNC then returns to its normal mode of operation (resting LOW until start of BIST is received).

If ENSYNC is deasserted before a K28.5 is found by the framer, SYNC will resume its normal BIST behavior on the next clock cycle.

Conclusion

The CY7B9331 OLC HOTLink Receiver is a derivative of the CY7B933 Receiver. Unlike the CY7B933, the CY7B9331 system interface matches the OLC electrical interface requirements. The names and behavior of CY7B9331 OLC pins is summarized below:

- The timing and behavior of the SYNC pin in Bypass Mode matches the OLC Byte Sync signal. The SYNC pin makes transitions simultaneous with the parallel data.
- The LOOP pin matches the OLC Loopback Enable signal. When asserted HIGH, LOOP selects the INB (local loopback source) as the data source.
- The STATUS pin inverts and translates the ECL signal on the SI pin to TTL levels. STATUS is used to generate the OLC Link Status signal from a carrier detect signal.
- The ENSYNC pin interfaces to the OLC Enable Byte Sync signal. When asserted HIGH, the ENSYNC pin enables the Receiver byte framer.
- The CLK0 pin generates the OLC CLK0 signal. CLK0 is the recovered byte rate clock.

In addition to the points mentioned above, this application note covers detailed information on how the SYNC pin behaves during normal, framing and BIST modes of operation. This information is presented as an aid to those integrating these features into a system.

Notes

1. Datasheet timing parameters that are defined in terms of bit times (t_B) include additional timing margin to account for internal buffer and routing delays and output load (e.g., $t_A = 2t_B$ +4/−2 ns).

Document #: COM0006−0194

HOTLink is a trademark of Cypress Semiconductor Corporation.

Interfacing with the SST

This application note describes how to interface the CY7B951 SONET/SDH Serial Transceiver (SST) with other physical-layer devices. The SST performs clock and data recovery from a SONET (SDH) 51.84 or 155.52 Mb/sec interfaces and can be used in a variety of SONET and ATM applications. The application note will begin with a brief introduction to the SST. Next, interface examples will be given that illustrate how to connect the SST to two different ATM controller devices; the first from PMC–Sierra called the PM5345 SUNI and the second called the WAC–013 from Integrated Telecom Technologies (IgT).

Introduction

The CY7B951 Serial SONET/SDH Transceiver (SST) is used in SONET/SDH and ATM applications to recover clock and data information from a 155.52-MHz or 51.84-MHz NRZ (Non Return to Zero) or NRZI (Non Return to Zero Invert on ones) serial data stream. This device also provides a bit-rate Transmit clock, from a byte-rate source through the use of a frequency multiplier PLL, and differential data buffering for the Transmit side of the system (see *Figure 1*). The pinout is shown in *Figure 2.*

Operating Frequency

The SST operates at either of two frequency ranges. The MODE input selects which of the two frequency ranges the Transmit frequency multiplier PLL and the Receive clock and data recovery PLL will operate. When MODE is connected to V_{CC}, the highest operating range of the device is selected. A 19.44-MHz ±1% source must drive the REFCLK input and the two PLLs will multiply this rate by 8 to provide output clocks that operate at 155.52 MHz ±1%. When the MODE input is connected to ground (GND), the lowest operating range of the

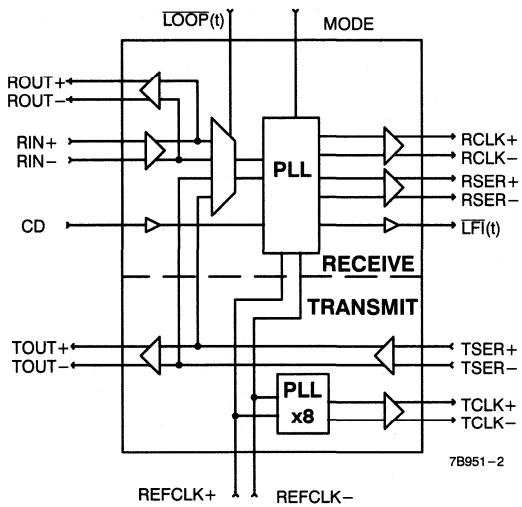


Figure 1. SST Block Diagram

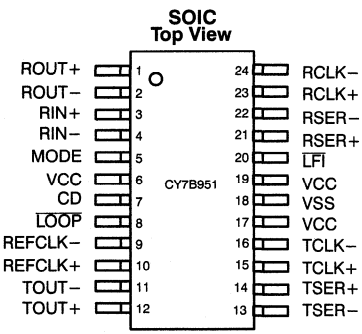


Figure 2. SST Pinout

device is selected. A 6.48-MHz ±1% source must drive the REFCLK inputs and the two PLLs will multiply this rate by 8 to provide output clocks that operate at 51.84 MHz ±1%. In addition, when the MODE input is left unconnected or forced to approximately $V_{CC}/2$, the device enters Test Mode.

Transmit Functions

The Transmit section of the SST contains a PLL that takes a REFCLK input and multiplies it by 8 (REFCLK*8) to produce a PECL (Pseudo ECL) differential output clock (TCLK±). The Transmitter has two operating ranges that are selectable with the three-level MODE pin as explained above. The SST Transmit frequency multiplier PLL allows low-cost byte-rate clock sources to be used to time the upstream serial data transmitter.

The REFCLK± input can be configured in three different ways. When both REFCLK+ and REFCLK− are connected to a differential 100K compatible PECL source, the REFCLK input will behave as a differential PECL input. When either the REFCLK− or the REFCLK+ input is at a TTL LOW, the other REFCLK input becomes a TTL-level input allowing it to be connected to a low cost TTL crystal oscillator. The REFCLK input structure, therefore, can be used as a differential PECL input, a single TTL input, or as a dual TTL clock multiplexing input.

The Transmit PECL differential input pair (TSER±) is buffered by the SST yielding the differential data outputs (TOUT±). These outputs can be used to directly drive transmission media such as Printed Circuit Board (PCB) traces, optical drivers, twisted pair, or coaxial cable.

Receive Functions

The primary function of the Receiver is to recover clock (RCLK±) and data (RSER±) from the incoming differential PECL data stream (RIN±) without the need for external buffering. These built−in line receiver inputs, as well as the TSER± inputs mentioned above, have a wide common-mode range (2-5V) and the ability to receive signals with as little as 50 mV differential voltage. They are compatible with all PECL signals and any copper media.

The clock recovery function is performed using an embedded PLL. The recovered clock is not only passed to the RCLK± outputs, but also used internally to sample the input serial stream in order to recover the data pattern. The Receive PLL uses the REFCLK input as a byte-rate reference. This input is multiplied by 8 (REFCLK*8) and is used to improve PLL lock time and to provide a center frequency for operation in the absence of input data stream transitions. The Receiver can recover clock and data in two different frequency ranges depending on the state of the three-level MODE pin as explained earlier. To insure accurate data and clock recovery, REFCLK*8 must be within 1000 ppm of the transmit bit rate. The standards, however, specify that the REFCLK*8 frequency accuracy be within 20−100 ppm.

The differential input serial data (RIN±) is not only used by the PLL to recover the clock and data, but it is also buffered and presented as the PECL differential output pair ROUT±. This output pair can be used as part of the transmission line interface circuit for base-line wander compensation, improving system performance by providing reduced input jitter and increased data eye opening.

Carrier Detect (CD) and Link Fault Indicator ($\overline{\text{LFI}}$) Functions

The Link Fault Indicator ($\overline{\text{LFI}}$) output is a TTL-level output that indicates the status of the Receiver. This output can be used by an external controller for Loss of Signal (LOS), Loss of Frame (LOF), or Out of Frame (OOF) indications. $\overline{\text{LFI}}$ is controlled by the Carrier Detect input, the internal Transitions Detector, and the PLL Out of Lock (OOL) circuitry.

The CD input may be driven by external circuitry that is monitoring the incoming data stream. Optical modules have CD outputs that indicate the presence of light on the optical fiber and some copper-based systems use external threshold detection circuitry to monitor the incoming data stream. The CD input is a 100K PECL-compatible signal that should be held HIGH when the incoming data stream is valid. When CD is pulled to a PECL LOW

(≤2.5V Max), the $\overline{LFI}$ output will transition LOW, the Receiver PLL will align itself with the REFCLK*8 frequency, and the recovered data outputs (RSER) will remain LOW regardless of the signal level on the Receive data stream inputs (RIN).

In addition, the SST has a built-in transitions detector that also checks the quality of the incoming data stream. The absence of data transition can be caused by a broken transmission media, a broken transmitter, or a problem with the transmit or receive media coupling. The SST will detect a quiet link by counting the number of bit times that have passed without a data transition. A bit time is defined as the period of RCLK±. When 256 bit times have passed without a data transition on RIN±, $\overline{LFI}$ will transition LOW. The Receiver will assume that the serial data stream is invalid and, instead of allowing the RCLK± frequency to wander in the absence of data, the PLL will lock to the REFCLK*8 frequency. This will insure that RCLK± is as close to the correct link operating frequency as the REFCLK accuracy. $\overline{LFI}$ will be driven HIGH again and the Receiver will recover clock and data from the incoming data stream when the transition detection circuitry determines that at least 64 transitions have been detected within 256 bit times.

The Transition Detector can be turned off by pulling the CD input to a TTL LOW (≤0.8V). When CD is pulled to a TTL LOW the $\overline{LFI}$ will only be driven LOW if the incoming data stream frequency is not within 1000 ppm of the REFCLK*8 frequency. $\overline{LFI}$ LOW in this case will only indicate that the Receiver PLL is Out of Lock (OOL). When this pin is left unconnected, an internal pull-down resistor will pull this input to Ground.

Loop Back Testing

The TTL level $\overline{LOOP}$ pin is used to perform loopback testing. When $\overline{LOOP}$ is asserted (held LOW) the Transmitter serial input (TSER±) is used by the Receiver PLL for clock and data recovery. This allows in-system testing to be performed on the entire device except for the differential Transmit drivers (TOUT±) and the differential Receiver inputs (RIN±). For example, an ATM controller can present ATM cells to the input of the ATM cell processor and check to see that these same cells are received. When the $\overline{LOOP}$ input is deasserted (held HIGH) the Receive PLL is once again connected to the Receiver serial inputs (RIN±).

The $\overline{LOOP}$ feature can also be used in applications where clock and data recovery are to be performed from either of two data streams. In these systems the $\overline{LOOP}$ pin is used to select weather the TSER± or the RIN± inputs are used by the Receive PLL for clock and data recovery.

Power-Down Modes

There are several power-down features on the SST. Any of the differential output drivers can be powered down by either tying both outputs to V_{CC} or by simply leaving them unconnected internal pull-up resistors will force these outputs to V_{CC}. This will save approximately 4 mA per output pair in addition to the associated output current. If the TOUT± or ROUT± outputs are tied to V_{CC} or left unconnected, the Transmit buffer or Receive buffer path respectively will be turned off. If the TCLK± outputs are tied to V_{CC} or left unconnected the entire Transmit PLL will be powered down.

By leaving both the RCLK± and RSER± outputs unconnected or tied to V_{CC} the entire Receive PLL is turned off. Even though the Receive PLL may be turned off, the Link Fault Indicator ($\overline{LFI}$) will still reflect the state of the Carrier Detect (CD) input. This feature can be used for aggressive power management.

Interfacing with the PM5345

The PM5345 is used in ATM application for SONET frame processing, ATM cell processing, and error monitoring. The PMC−Sierra SUNI device requires Receive serial data aligned with a bit-rate clock. These signals need to be supplied through the RXD± and RXC± inputs respectively. A 155.52 MHz PECL Transmit clock (TXC±) is required to provide PM5345 transmit side clocking. For copper-based systems, the TXD± outputs must be buffered in order to drive transmission lines with low impedances. Lastly, a Loss of Signal detection is required from the clock and data recovery engine to aid in the determination of the LOS, Loss of Frame,

Out of Frame error conditions reported by the SUNI device. This signal is brought in through the SUN GPIN (General purpose input). Before the introduction of the SST, clock and data recovery devices were interfaced to the PMC−SUNI as shown in *Figure 3*

Figure 4 shows the SST signal connections with the PMC−Sierra PM5345 SUNI. The SST together with the PM5345 provide a complete Physical interface. The Receive section of the SST provides serial SONET/SDH data at 155.52 Mb/sec to the receive section of the PM5345 (RXC± and RXD±). The Transmit section of the SST provides the transmit side 155.52-MHz clock that is used by the PM5345 TXCI± input by multiplying a 19.44-MHz oscillator by eight. This function eliminates the need for an expensive 155.52-MHz oscillator to be used in the system. The SST buffers the TXD± output signals from the SUNI device for driving copper-based systems or for improved operation in fiber-based systems.

The $\overline{\text{LFI}}$ output is used to drive the GPIN input. This output will transition LOW when any of the following occur: the CD (Carrier Detect) input transitions LOW, the frequency of the incoming data is outside of the lock range of the Receive PLL, or

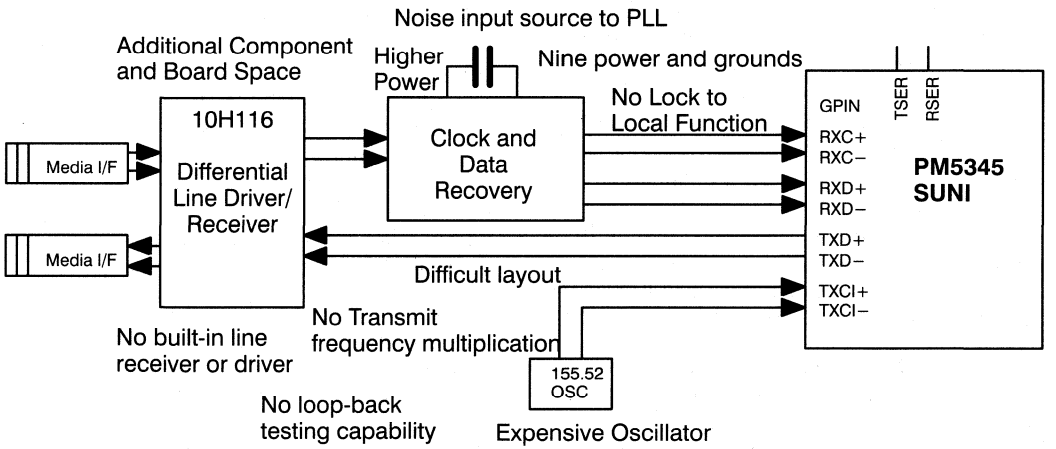


Figure 3. Typical SUNI interface without the Use of the SST

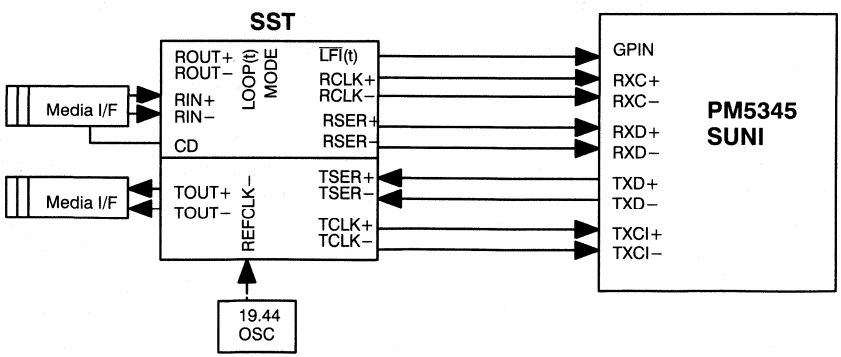


Figure 4. SST to PMC−Sierra PM5345 SUNI Connection Diagram

there have been no transitions in the incoming data stream for the last 512 bit times. Additionally, when the CD input is forced LOW by an output from a source such as the signal detect of an optical module or an external transition detection circuitry for copper-based systems, the SST will force the RSER± outputs LOW. This will aid the SUNI device in the determination of the LOS state and minimize the length of time needed to determine an error condition.

Figures 5 and *6* show two electrical interfaces of the SST to the PMC−SUNI device. The first figure shows a low-power interface. Each SST PECL output is AC coupled into the PMC inputs with a 100-pF capacitor. The outputs are not directly connected to the PMC device because these inputs are single ended with an internal reference voltage. The SST outputs have a load resistor of 270Ω placed at the output pin and a terminating resistor of 100Ω between the output pair at the load end of the transmission line. The TXD+ and TXD− outputs are terminated with 100Ω pull-up resistors to TAVD. This interface example assumes that all SST PECL outputs see a 50Ω transmission line and that the TXD outputs see a 100Ω transmission line.

The TXD+ and TXD− outputs of the PMC−SUNI device can drive impedances as low as 100Ω. If higher impedance transmission lines are driven with these outputs the terminating resistor should be changed to equal the transmission line impedance. The VT2 reference resistor (RVT2) should be changed to 6.98 x Z0 and RVT1 should be changed to 10.98 x Z0.

The SST can drive impedances as low as 50Ω. For other transmission line impedances the load resistors and terminating resistors should be scaled ac-

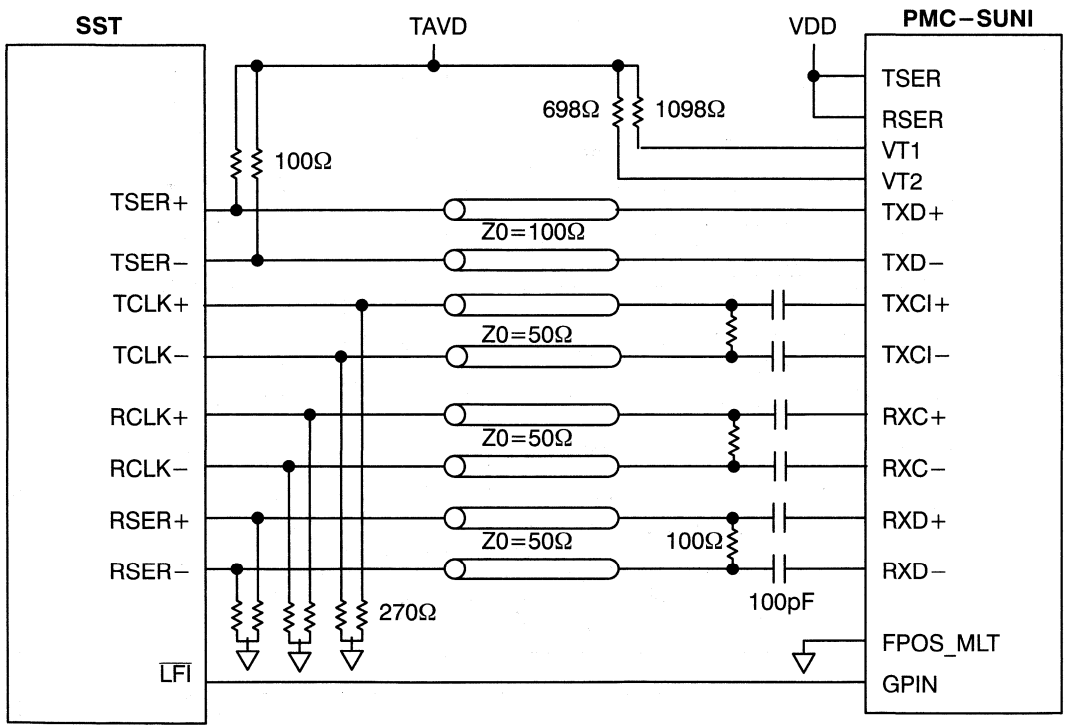


Figure 5. Low Power SST to PMC−SUNI Interface

cordingly. For example if the impedance (Z0) of the PCB trace was 100Ω instead of 50Ω then the load resistor would be changed to 540Ω and the terminating resistor would be changed to 200Ω

Figure 6 shows a high-performance SST to PMC–SUNI interface. This interface will provide more symmetric output waveforms at the expense of higher power consumption. The SST outputs are loaded and terminated with 80Ω pull-up resistors and 130Ω pull-down resistors at the PMC inputs. This provides a 50Ω termination to V_{CC} – 2V. These resistors are also bypassed with a .01-μF capacitor to provide the high-speed switching current requirements. For PCB trace impedances higher than 50Ω the terminating resistors should be scaled accordingly. For example, a 100Ω transmission line would require a pull-up resistor of 160Ω and a 260Ω pull-down resistor.

Interfacing with the IgT WAC–013

The Integrated Telecom Technology (IgT) WAC–013 provides SONET frame processing, ATM cell processing, and error monitoring. The IgT device requires differential PECL Receive data (RS_SER_DATA) aligned with a differential PECL bit rate clock (RS_SER_CLK). These signals represent the recovered clock and data from a SONET/SDH STS–3/STM–1 data stream of 155.52 Mb/sec or a SONET STS–1 data stream of 51.84 Mb/sec. The WAC–013 also requires a bit-rate transmit-clock (TS_SER_CLK) for Transmit Side clocking. The transmit data (TS_SER_DATA) should also be

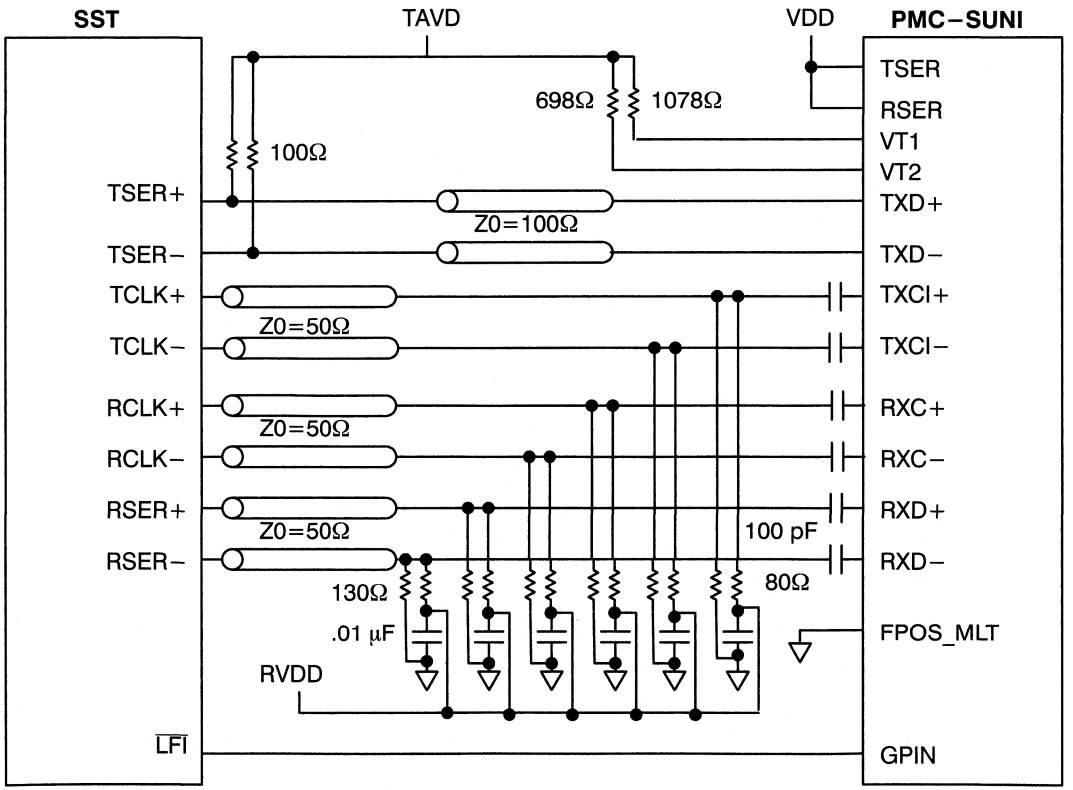


Figure 6. High Performance SST to PMC SUNI Interface

buffered for driving low impedance transmission lines or copper transmission media. Prior to the introduction of the SST, clock and data recovery devices were connected to the WAC−013 as shown in *Figure 7*.

Figure 8 shows the SST signal connections with the IgT WAC−013. The SST together with the WAC−013 provides a complete physical-layer interface. The Receive section of the SST provides serial SONET/SDH data at 155.52 Mb/sec or 51.84 Mb/sec (depending on the state of the SST MODE pin) to the Receive section of the IgT RS_SER_DATA and RS_SER_CLK inputs. The Transmit section of the SST provides the bit-rate clock (TS_SER_CLK) and Transmit buffering of the TS_SER_DATA outputs. The SST multiples a 19.44 MHz (6.48 for STS−1 applications) by eight to produce the 155.52-MHz (51.84 MHz) transmit clock. This frequency multiplication function eliminates the need for an expensive 155.52-MHz crystal oscillator.

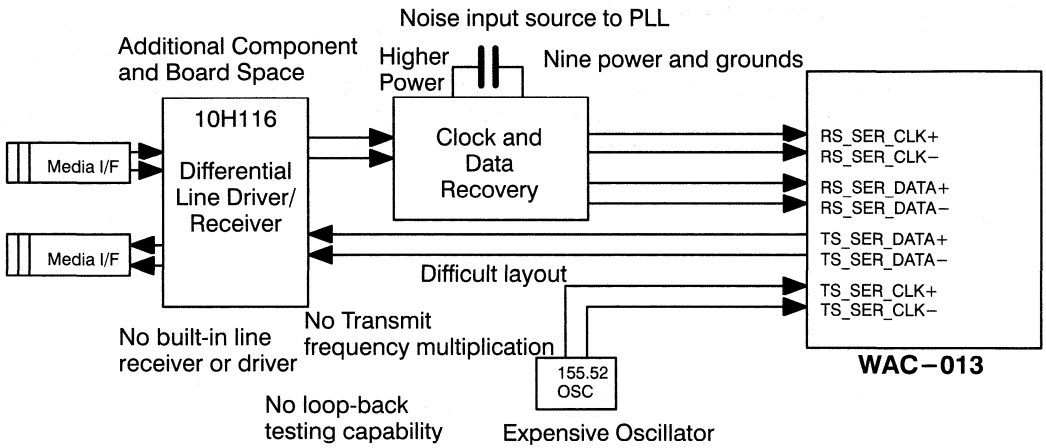


Figure 7. Typical WAC−013 interface without the Use of the SST

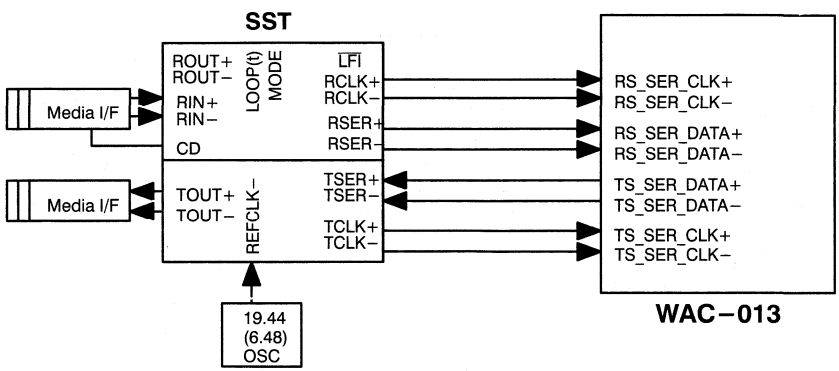


Figure 8. SST to IgT WAC−013 Connection Diagram

Figures 9 and *10* show two electrical interfaces of the SST to the WAC−013. The first figure shows a low-power interface. Each output has a load resistor of 270Ω to ground placed at the output and a terminating resistor of 100Ω placed between the output pair at the load.

This interface example assumes that all outputs see a 50Ω transmission line. For other transmission line impedances, the load resistors and terminating resistors should be scaled accordingly. For example, if the impedance (Z0) of the PCB trace was 100Ω instead of 50Ω then the load resistor would change to 540Ω and the terminating resistor would change to 200Ω

Figure 10 shows a high-performance SST to WAC−013 interface. This interface will provide more symmetric output waveforms at the expense of higher power consumption. The outputs are loaded and terminated with 80Ω pull-up resistors and 130Ω pull-down resistors at the load. This provides a 50Ω termination to V_{CC}−2V. These resistors are also bypassed with a .01-μF capacitor to provide the high-speed switching current requirements. For PCB trace impedances higher than 50Ω the terminating resistors should be scaled accordingly. For example, a 100Ω transmission line would require a pull-up resistor of 160Ω and a pull-down resistor of 260Ω

Conclusion

The previous interface examples showed how to connect the SST to the PMC−Sierra PM5345 SUNI and the IgT WAC−013. Together these devices provide a complete physical-layer solution for ATM applications over SONET/SDH at 155.52 Mb/sec and

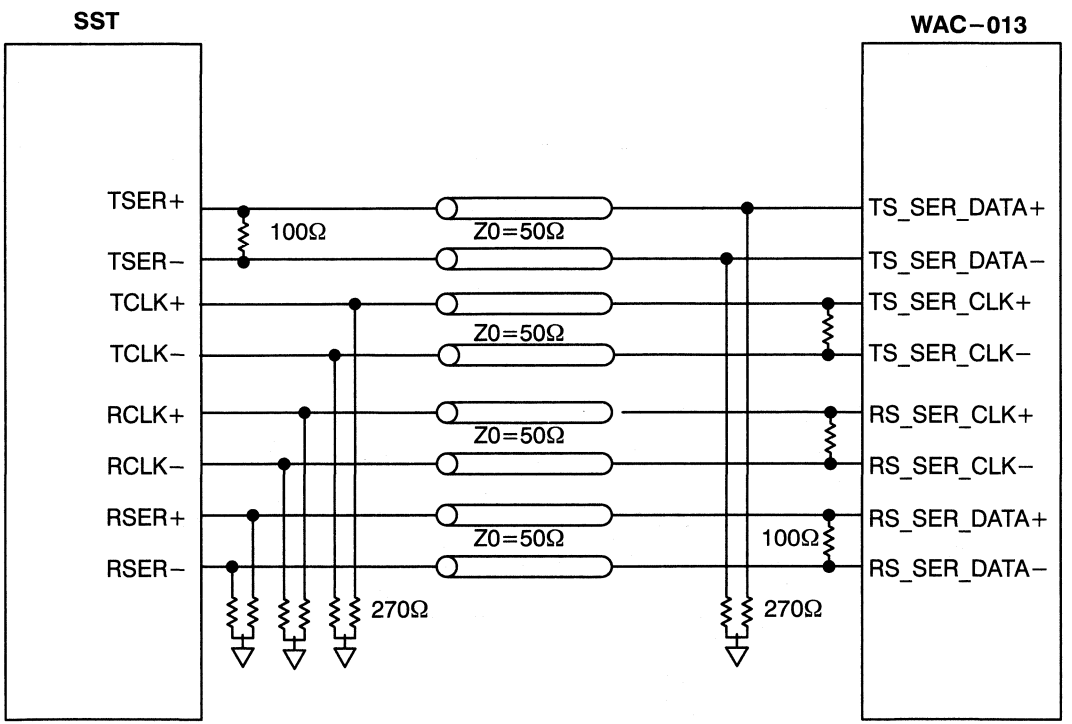


Figure 9. Low Power SST to WAC−013 Interface

51.84 Mb/sec. The SST greatly simplifies the physical-layer implementation with its ability to generate a Loss of Signal indication, its capability to lock to the local reference clock during error conditions, and its capacity to buffer the transmit data stream for driving low-impedance transmission lines. The SST also reduces the cost of physical-layer implementations by eliminating the need for a 155.52-MHz crystal oscillator with its ability to multiply a byte-rate clock to provide the bit-rate transmit source. Cypress' expertise in PLL-based clock and data recovery as well as the added features of the SST provide designers with the capacity to create simple, low cost, and robust ATM physical-layer designs.

Document #: COM0012−0494

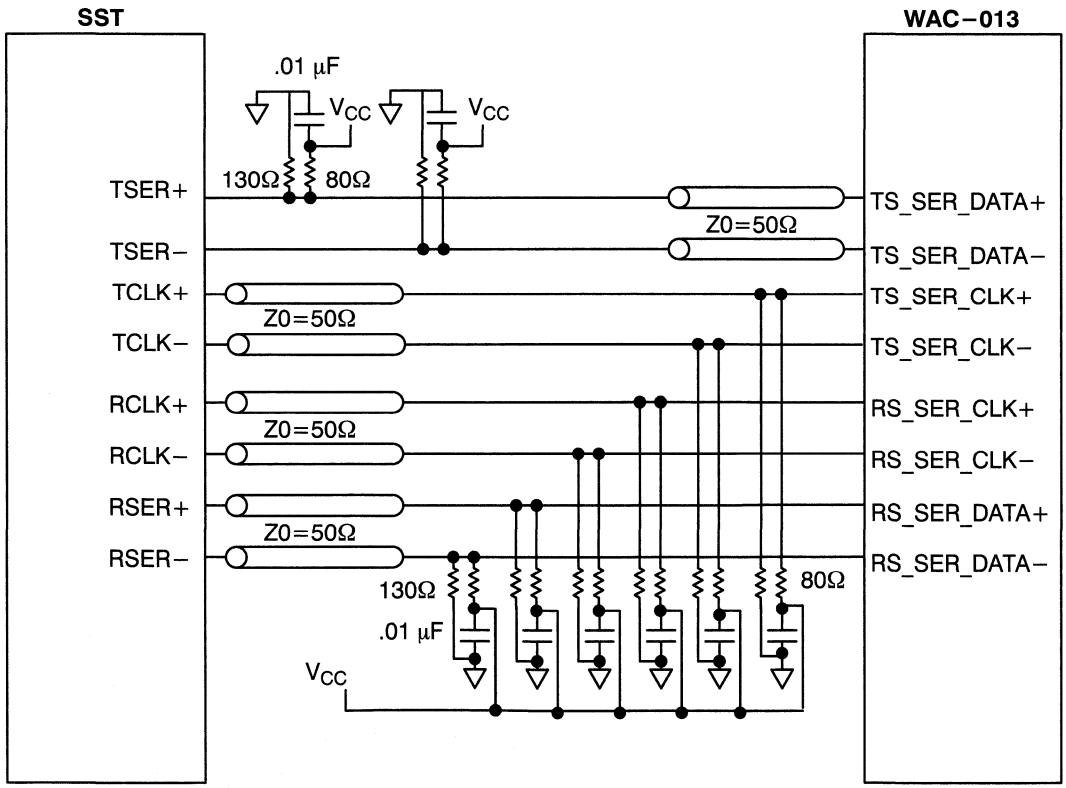


Figure 10. High Performance SST to WAC−013 Interface

Timing Products – 6



Timing Products Section Contents and Abstracts

Everything You Need to Know About CY7B991/CY7B992 (RoboClock) But Were Afraid to Ask 6–1

The following application note provides a detailed description of the CY7B991 and CY7B992 Programmable Skew Clock Buffers (PSCB). The application note begins with a brief description of clock distribution definitions and solutions. The note follows with RoboClock system design considerations including board decoupling and PCB transmission line analysis, effects, and terminations including actual waveforms and V-I characteristics. A detailed description comes next that explains the device architecture, device configuration, device operation, functional implementations, and a detailed analysis of the AC specifications. The application note ends with a brief AC characterization of the output rise time, fall time, and duty cycle variation.

Innovative RoboClock Application 6–41

This application note presents a unique application of RoboClock, whose complex and precise waveform generation capability is utilized to implement PWM to enhance color images and increase the resolution of laser printers. The first section of this application note provides a brief description of Roboclock and presents three methods that users could employ to configure it. Then, a brief background on image and resolution enhancement is presented. Finally, the required waveform to implement the image enhancement, and the configuration of Roboclock is presented.

Using the CY7B991 with the 50-MHz 486 Cache Module and the 40-MHz R3000 6–53

This application note begins with a brief introduction to clock distribution. A functional description of RoboClock follows that explains the various datasheet parameters and configuration options including programmable phase adjustment, frequency multiplication and division, and phase inversion. Lastly, two design examples are given, including a detailed configuration and timing analysis, that show how to design both a 486 Cache Module and a R3000 system using RoboClock.

CY7B991 and CY7B992 (RoboClock) Test Mode 6–77

This application note discusses the Test mode capabilities of the CY7B991 and CY7B992 (RoboClock) devices. It begins with an introduction to these devices and then discusses how to use the Test mode features. These features stop the PLL of the device to allow operation in single-step mode while maintaining selected clock output configuration.

Crystal Oscillator Topics 6–81

This application note discusses the important aspects of using quartz crystals with IC Designs products. The note discusses how to correctly specify crystals to be used with ICD products. The impact of frequency accuracy is discussed in relation to time-critical applications. In addition, several crystal vendors are referenced.

ECL Outputs 6–83

This application note covers the principal advantages of using ECL outputs and makes recommendations concerning layout and wiring methods for parts such as the ICD2062.

Everything You Need to Know About CY7B991/CY7B992 (RoboClock) But Were Afraid to Ask

Introduction

The following application note provides a detailed description of the CY7B991 and CY7B992 Programmable Skew Clock Buffers (PSCB). It also provides an overview of clock distribution and transmission line analysis. This application note is divided into the following sections:

- General Description
- Clock Distribution
- System Design Considerations
- Detailed Device Description
- AC Specifications
- AC Characterization
- DC Specifications

General Description

Figure 1 is a general block diagram of Cypress's Programmable Skew Clock Buffer (internally called RoboClock). RoboClock employs a phase-locked-loop architecture to provide output clocks that are aligned both in phase and in frequency with a reference input clock. Each of the four output pairs is controlled by two dedicated three-level function select inputs that allow the outputs to be phase adjusted by as much as ±18 ns, divided, multiplied, or inverted. In all, over 26,000 different output combinations are possible.

A three-level frequency select (FS) input selects one of three PLL operating ranges that allow the outputs of the PSCB to operate from 3.75 to 80 MHz. All of these device configurations are possible while still maintaining an output-to-output skew and a propagation delay no greater than 500 ps.

The following section discusses the effects of skew on system performance, showing why RoboClock is ideally suited for solving clock distribution problems.

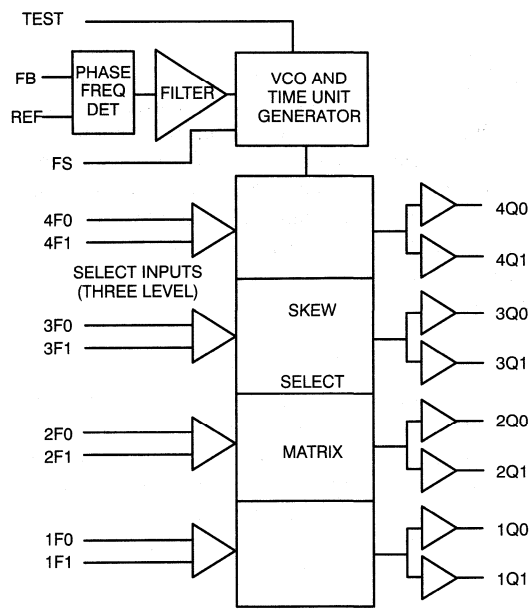


Figure 1. Logic Block Diagram

Clock Distribution

Skew is the variation in arrival time of two signals specified to occur at the same time. Skew is composed of the output skew of the driving device and variation in the board delays caused by the layout variation of the output traces.

Skew affects synchronous systems primarily in the form of clock skew. Since the clock signal drives many components of the system, and since all of these components should receive their clock signal at precisely the same time in order to be synchronized, any variation in the arrival of the clock signal at its destination will directly impact system performance. Skew directly affects system margins by eroding the predictability of the arrival of a clock edge. Because elements in a synchronized system require clock signals arrive at the same time, clock skew reduces the cycle time within which information can be passed from one device to the next.

As system speeds increase, clock skew becomes an increasingly large portion of the total cycle time. When cycle times were 50 ns, clock skew was rarely a design priority. It could be as much as 20% of the cycle time. As cycle times approach 15 ns and less, however, clock skew requires an ever-increasing amount of design resource. Typically, these high-speed systems can have only 10% of their timing budget dedicated to clock skew, so obviously, it must be reduced. The following sections will look at the two types of clock skew and how each affects system performance. The first type of skew is caused by the clock driver itself. It is referred to as intrinsic skew. The second type of clock skew is caused by the PCB layout and design and is referred to as extrinsic skew.

Clock Driver Skew (Intrinsic Skew)

Intrinisic clock skew is the amount of skew caused by the clock driver or buffer by itself. Intrinsic skew is not caused by board layout or any other design issues except for the specification stated on the clock driver data sheet. There are two main types of clock driver architectures: a buffer-type device and a feedback-type device.

Buffer Devices

In a buffer-style clock driver the input waveform propagates through the device and is "redriven" by the output buffers. This output signal directly follows the input signal. The output skew of these devices is caused by the differences in propagation delay between the input signal through the device and the precision of the matching and tuning of the internal circuit elements.

A member of this type of clock driver is the 74F244, which is available from several manufacturers. This device contains eight inputs that individually drive their respective outputs, and it is made into a clock distribution device by tying its inputs together to minimize the contribution of input skew to the device skew. The output skew of this device, if it is not listed on the datasheet, can be calculated by subtracting the minimum propagation delay from the maximum propagation delay. This calculated skew can be up to 3 ns.

This 3 ns clock driver skew does not even take into account the affects the board layout and design. In a 20- or 25-MHz system this is an acceptable amount of skew, but for systems running at 33 MHz and beyond, another method is needed.

To eliminate the device propagation delay variation that contributes to skew, manufacturers have designed devices that are specifically tuned to drive clock traces with low skew outputs. These manufacturers have specified the maximum variation in propagation delay through the device. In these devices, three types of skew parameters are usually listed. Output skew specifies the maximum amount of propagation delay variance between output pins. Duty cycle variation indicates a measure of the propagation delay difference between a LOW-to-HIGH output edge transition and a HIGH-to-LOW output edge transition. And part-to-part skew indicates the difference in output skew from device to device. Output skew of these devices has, in many cases, been reduced from the 3 ns mentioned above to 500 ps.

These devices still face the problems of device propagation delay. The propagation delay through these devices is about 5 ns. This delay will cause skew in systems where both the reference clock to the buffer and the outputs of the buffer need to be aligned.

These devices also have the drawback that the output waveform is directly based on the input waveform. If the input waveform is a non-50% duty-cycle clock, the output waveform will also have a less-than-ideal duty cycle. Expensive crystal oscillators are needed when using this type of buffer in systems requiring near 50/50 outputs.

These devices also lack the ability to phase or frequency adjust their outputs. Phase adjustment allows the clock driver to compensate for trace propagation delay mismatches and set-up and hold time differences, and frequency adjustment allows the distribution of high- and low-frequency clocks from the same common reference. Expensive additional components and time-consuming board routing techniques must be used to compensate for the functional shortcomings of these buffer-style clock driver devices.

PLL Clock Driver Devices

The second type of clock distribution device uses a feedback input that is a function of one of the outputs. This type of device is usually based upon one or more phase-locked loops (PLL) that are used to align the phase and frequency of the feedback input and the reference input. In this way the propagation delay through the device can be virtually eliminated. Cypress's Programmable Skew Clock Buffer family is based on this architecture, and will be explained in greater detail in the following sections.

In addition to very low device propagation delay, this type of architecture enables output signals to be phase shifted to compensate for board-level trace-length mismatches, and outputs can be selectively divided, multiplied, or inverted while still maintaining very low output skew.

Board Design Skew (Extrinsic Skew)

Just as the clock driver had to be evaluated to contribute minimal clock skew, the board layout and design must also be evaluated. Issues that affect board-level clock skew include trace length, capacitive loading, transmission line termination, and threshold voltages at the loads.

The time that it takes for a signal to propagate down the trace is dependent on factors such as the material that the PCB is constructed from, the length of the signal trace, the width of the trace, and capacitive loading. Variations in these factors from trace to trace will cause signals to arrive at their destination at different times.

In addition to this, threshold voltage variation on the receiving devices can play a significant role in the time of the recieved clock signals at various loads. If one load device has a threshold of 1.2V and another load device has a threshold of 1.7V and the rising edge rate is 1V/ns, there will be 500 ps of skew caused by the point at which the load device switches based on the input signal.

The most obvious way to reduce board design clock skew is to make the physical length of all clock traces the same. The propagation delay of an electrical signal down a trace is about 2 ns per foot. If one clock trace is just 3″ longer than the next, this will cause 500 ps of clock skew; which is as much skew as the clock driver itself contributes. But this is not enough. Impedance variation causes signal velocity variation, so physically matched lines may not be electrically identical.

Capacitive loading also contributes to clock skew. The differences in capacitive loading will cause differences in clock edge rate at the load. The variation between the edge rate of a lightly loaded trace and a heavily loaded trace will directly affect the time at which the clock edge crosses the input threshold and therefore the affect clock skew of these two devices.

Transmission line termination also plays a significant role in board-induced clock skew. Remember that a transmission line is any trace that has a propagation delay longer than one-half the driving device edge rate. With the extremely fast edge rates of today's clock drivers approaching 500 ps, traces with lengths of only 2″ must be considered transmission lines. Without proper termination, the clock signals

present on these traces will exhibit transmission-line effects such as voltage reflections that will, in the best case, cause a variability in clock-edge position and, at worst, might cause multiple clocking of the load.

Many of the issues mentioned in this section will be discussed in more detail in the System Design Considerations section.

System Design Considerations

Board Decoupling

Figure 2 shows the pinout for the PSCB family of devices. These parts are offered in both 32-pin PLCC and LCC packages. Each device contains 6 power pins (Vcc) and 5 ground pins. Each output pair (e.g., 1Q0 and 1Q1) have a dedicated power and ground pin immediately adjacent to them. For example, the 1Q0 and 1Q1 output pair have pins 25 and 22 as their dedicated power and ground supplies, respectively. These dedicated power pins are only used for the output driver pair. This provides the RoboClock outputs with very high drive while maintaining very high crosstalk immunity from adjacent outputs.

The other two pairs of power and ground pins are used to supply the internal PLL and associated circuitry. These power pins are completely separated from the power pins supplying the output buffers. This minimizes the output switching noise effects on the PLL and, therefore, minimizing output jitter.

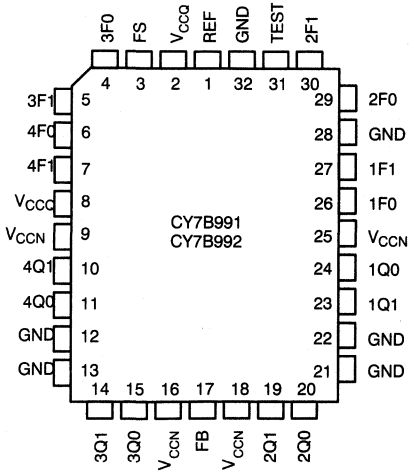


Figure 2. RoboClock PLCC/LCC Pinout

The CY7B99x family requires common high-speed power-supply decoupling and bypassing. Power-supply bypassing requires adding capacitance between the power and ground supply of a device in order to supply instantaneous current for its rapidly changing signals. This capacitor prevents the device from becoming current starved by providing the "instantaneous" transient current and thus preventing the local power-supply-voltage dip during current demands.

If all capacitors were ideal, this would be a trivial task. Capacitors, however, are not ideal. They are made up, in a first-order approximation, of an effective series inductor (ESL), effective series resistor (ESR), and a capacitor, as shown in *Figure 3.* Their response, therefore, is not constant over the frequency spectrum. At some frequencies this circuit (the bypass capacitor) will look most like a capacitor, at some frequencies an inductor, and at a particular frequency it will behave as nothing more than a resistor. This point is called series resonance and occurs when

$$\omega = \frac{1}{\sqrt{LC}} \qquad \text{Eq. 1}$$

In general, for a given capacitor construction, the series resonant frequency increases as the capacitance decreases. It would seem simple to choose a capacitor that has a series resonance at a higher frequency than the frequency of the current it needed to supply: To select a capacitor with a resonant point beyond 80 MHz, for example, if that were the operating frequency of the clock outputs. The selection process is not that simple.

Figure 4 shows the spectrum analysis of an 80 MHz clock signal with 1 ns edge rates. Notice that the fre-

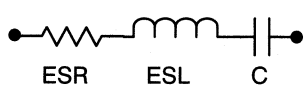


Figure 3. Equivalent Capacitor Diagram

quency components necessary to build this near ideal square wave have significant energy all the way out to 2 GHz. This means that in order to supply the current demands of these frequency components, a capacitor would have to have a series resonant frequency beyond 2 GHz. Generally available capacitors, however, have series resonant frequencies of about 400 MHz or less. These devices will be sufficient for supplying the majority of the instantaneous current.

At least two or three capacitors representing different capacitance ranges should be used for circuit bypassing. The first type of capacitor should be a 100- to 500-pF capacitor made of an NPO dielectric. This capacitor should be rated for operation at frequencies equal to or greater than 350 MHz. The second type of capacitor should be a 0.1-μF capacitor made from an X7R or a similar dielectric. This capacitor will supply the majority of the low frequency current requirements. If space permits, a third capacitor can be chosen that has a capacitance between that of the .1-μF and the 100- to 500-pF capacitors. This will provide a broad range of noise filtering and current supply.

The series resonant frequency can be decreased in two ways; lowering the capacitance or lowering the inductance. The major contributor to the inductance of a capacitor is the leads themselves. Surface-mount capacitors have a much smaller inductance than leaded capacitors and therefore have a higher resonant point. The benefits of a surface-mount capacitor can be completely nullified, however, if it is not properly integrated into the PCB environment.

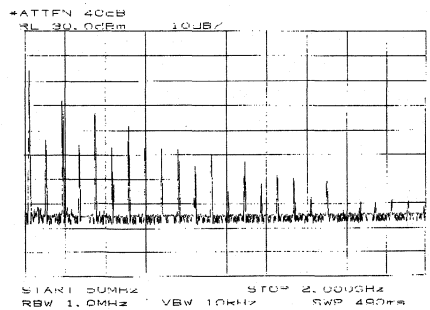


Figure 4. Frequency Components of an 80-MHz Clock Signal

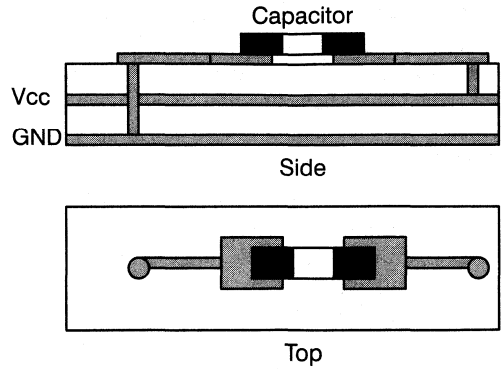


Figure 5. Typical Capacitor Layout

Figure 5 shows a typical method of integrating a surface-mount capacitor into a PCB environment. A surface-mount capacitor is used because of its low lead inductance, but no attention to reducing power connection trace inductance is made. In this case a leaded capacitor would provide less total inductance. The surface-mount capacitor, in this case, may not provide any high-frequency bypassing.

A better method of laying out surface-mount capacitors is shown in *Figure 6.* Here the multiple short leads are made to the power planes. The trace

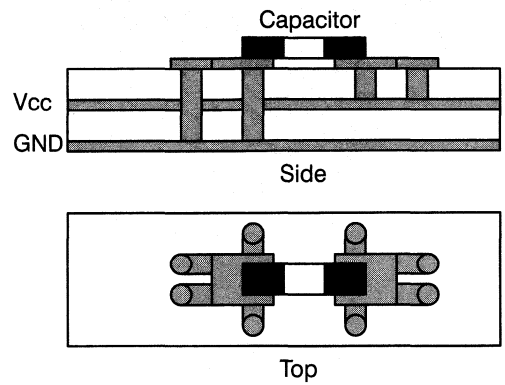


Figure 6. Better Capacitor Layout

widths and via hold sizes are also increased. These methods reduce the inductance of the power connections of the capacitor. Only when proper attention is paid to the selection and layout of the capacitor will the true benefits of circuit bypassing be realized.

Figure 7 shows a sample layout of RoboClock on a multilayer printed circuit board. This figure assumes that an internal V_{CC} and GND plane exist. The internal board V_{CC} and GND planes are connected to the device V_{CC} and GND planes through multiple via holes shown as black dots in the figure. Multiple via holes and connection of the chip power pins to local power planes reduces the amount of inductance that these pins have to their respective power connections.

Two sets of capacitors are used. They are placed on the same side of the board as the device. Each set consists of a 0.1-µF and a 100-pF high-frequency capacitors. Mutliple via holes are also included for the bypass capacitor connections to reduce the inductance that the V_{CC} pins see in series with their capacitor. The FB pin is connected to the 2Q1 pin in this diagram as an example of how easy the FB pin can be connected to either the 3Q0 and 2Q1 pins in order to reduce trace length and minimize potential problems associated with voltage reflections on transmission lines.

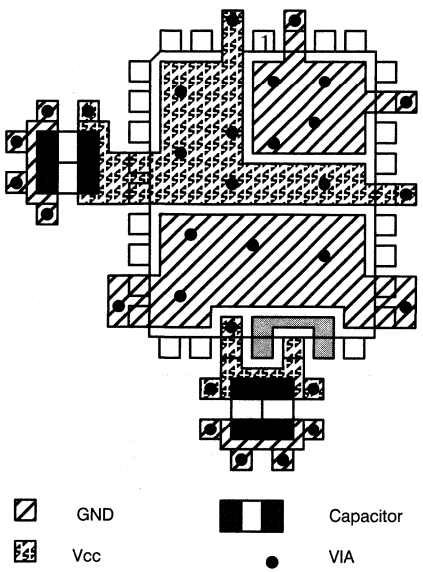


Figure 7. Sample RoboClock Layout

This layout is not the only way that these devices can be laid out, but this figure shows examples of good high-performance layout techniques. It is assumed that the board in which RoboClock will be placed will contain at least one dedicated V_{CC} plane and one dedicated ground plane and that the device will be surface mounted directly to the PCB without the use of a socket. The reason for the last constraint is that the additional lead inductance of the socket directly impacts the output skew of the device.

A more detailed discussion of series resonant frequency and other capacitor characteristics can be found in the materials supplied by capacitor manufacturers such as American Technical Ceramics [(516) 547-5700] and AVX [(803) 448-9411].

Transmission Lines

Transmission line theory states that a signal sent down a transmission line that has a constant characteristic impedance will propagate undistorted along the line. At the end, a voltage reflection will occur if the load impedance is not equal to the characteristic impedance of the transmission line. These voltage reflections are always present in electrical interconnections between devices and have traditionally been ignored. With the lengthening of Printed Circuit Board (PCB) traces and the decreasing of the edge rates of the driving element of these electrical signals, however, these effects become more pronounced. Transmission line effects cause many undesirable results in high-speed systems, such as delays and ringing. These effects will be discussed in greater detail.

In general, the effects of voltage reflections should be considered when laying out clock lines and any other PCB trace, if the propagation delay of the trace is greater than twice the faster of the rise time or fall time of the driving signal (*Equation 2*).

$$MIN[t_r, t_f] < 2\ t_{pd} \qquad \text{Eq. 2}$$

In other words, if the rise time (or fall time) of the source is less than the two-way propagation delay, then the rising signal will not hide the effects of the signal propagating down a transmission line. In this case, the switching wave will have enough time to propagate down the transmission line, reflect off of the load, and be seen at the source. These voltage reflections can cause decreased signal integrity, that will manifest itself, in the case of clock traces, as increased rise and fall times, non-ideal duty cycle performance, and possibly even unwanted clock pulses due to voltage reflections that cross the threshold of the load device.

The first step, then, in determining if a trace should be considered a transmission line, is to evaluate the propagation delay and characteristic impedance. The propagation delay is needed in order to determine when a trace must be considered a transmission line and the characteristic impedance is needed in order to determine how to reduce voltage reflections on these traces as will be shown in the section entitled Transmission Line Termination. The following discussion will focus on calculating the propagation delay and characteristic impedance on various PCB traces. This analysis, however, can be easily extended to include other types of transmission media such as coax, twisted pair, and wire-wrapped environments.

The analysis of transmission line effects on PCB traces begins with a simplified circuit analysis of the trace itself (*Figure 8*). This figure models the trace as a distributed intrinsic resistance (R_O), inductance (L_O), and capacitance (C_O). For the purposes of this discussion, a lossless transmission line will be assumed, that implies that the intrinsic series resistance will be equal to 0. The effect of this resistance on the characteristic impedance is extremely small, and only on very long traces will the effects of this component result in a noticeable drop in the voltage realized at the load.

The characteristic impedance, therefore, can be expressed as:

$$Z_0 = \sqrt{\frac{L_0}{C_0}}\ \Omega \qquad \text{Eq. 3}$$

And the propagation delay can be expressed as:

$$t_{pd} = \sqrt{(L_0 \times C_0)} = Z_0 C_0 \quad ns/length \qquad \text{Eq. 4}$$

In many cases it may be hard to measure the intrinsic inductance and capacitance of the trace in order to determine the magnitude or even the existence of transmission line effects. In this case, equations are needed in order to determine these values.

The following two sections will give equations for the characteristic impedance and propagation delay of two typical types of PCB trace construction; microstrip and strip line. Many sources exist that give an analysis of the equations listed below. Some sources give an even a more detailed analysis, but the minor differences are overshadowed by errors caused by factors not related to the analysis such as

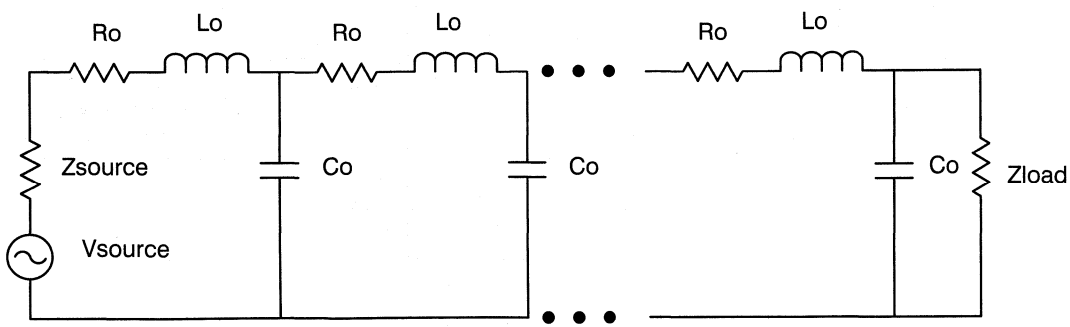


Figure 8. Simplified PCB Trace Model

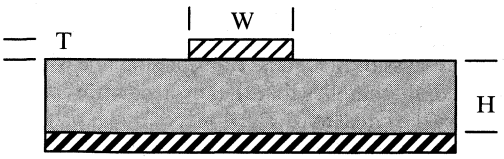


Figure 9. Microstrip

component variation and manufacturing uncertainties.

Microstrip

A microstrip trace is a signal separated from the ground plane by a dielectric, as shown in *Figure 9*. This type of trace is most commonly found as the top or bottom traces on a multilayer printed circuit board. The formula for calculating the characteristic impedance is given as:

$$Z_0 = \frac{87}{\sqrt{\varepsilon_r + 1.41}} \ln\left(\frac{5.98H}{0.8W + T}\right) \quad \text{Eq. 5}$$

and the propagation delay can be expressed as

$$t_{pd} = 1.017\sqrt{0.475\varepsilon_r + 0.67} \quad \text{Eq. 6}$$

where

ε_r is the dielectric constant of the material used for the PCB construction
H is the distance the trace lies away from the ground plane (board thickness)
W is the trace width (wire width)
T is the trace thickness (wire thickness)

This formula will not yield the exact impedance of the trace, but is meant as a guideline for estimating the trace impedance. Differences between the calculated and real trace impedance will be caused by slight errors in the equation itself and in process and layout variability in parameters such as dielectric constant, ground plane continuity, capacitive loading, trace width and thickness, and board thickness.

All of these variables, except for possibly the dielectric constant and trace thickness, are under the control of the designer during board layout. Dielectric constants of material used in the construction of fiberglass PCBs have a value between 4.0 and 5.5. Board manufacturers should be able to provide this parameter upon request. *Figure 10* is a graph showing how trace impedance varies with trace width and dielectric thickness. The graph assumes a dielectric constant of 4.5 and a trace thickness of 1.4 mils.

For example, if the designer wishes to create a microstrip trace with a 50Ω impedance, the designer would first have to know the thickness of the dielectric. If the board is a four-layer board (two signal layers and two routing layers) and if the trace will be on the component side with the power plane directly beneath it, then the dielectric thickness is roughly

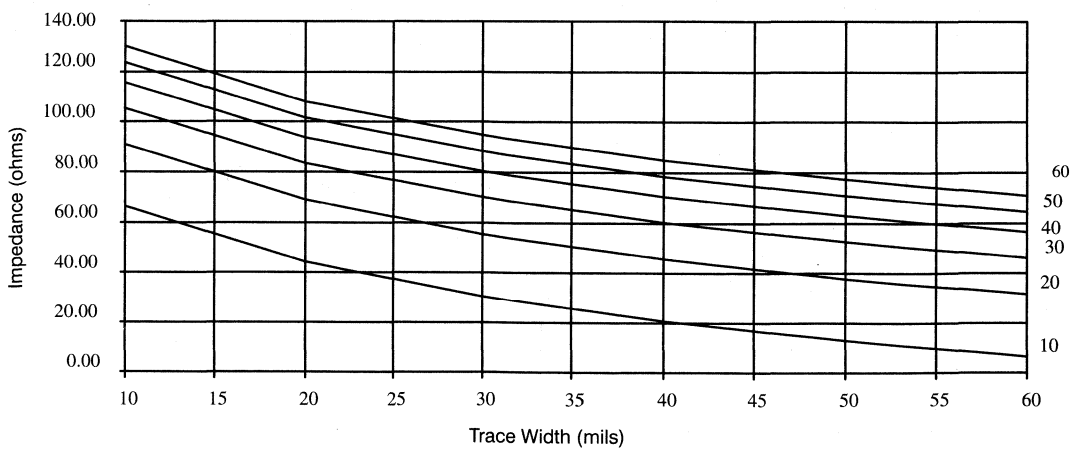


Figure 10. Impedance vs. Trace Width over Dielectric Thickness (Microstrip)

the board thickness divided by three. For a 62.5 mil board this would translate to a dielectric thickness of about 20 mils.

The designer would also have to know the thickness of the trace (approximately 1.4mils for standard 1 oz copper traces), and the dielectric constant (assume 4.4). All that is left now is the thickness of the trace, which can be directly controlled as part of the layout process. In this example, if the trace width is 36 mils, the characteristic impedance of the trace would be

$$Z_0 = \frac{87}{\sqrt{4.4 + .41}} \ln\left(\frac{5.98x20}{0.8 * 36 + 1.4}\right) = 49.68\Omega \quad \text{Eq. 7}$$

The propagation of a signal along this trace is independent of everything but the dielectric constant and is calculated in this case as:

$$t_{pd} = 1.017\sqrt{0.475 * 4.4 + 0.67} = 1.69\ ns/foot \quad \text{Eq. 8}$$

Both the equation for characteristic impedance and propagation delay will be useful for estimating the magnitude of voltage reflections and for determining the correct method of eliminating these reflections, which will be discussed in the Transmission Line Termination section.

Strip Line

Strip Line is analogous to a buried trace in a multilayered PCB between two power planes as shown in *Figure 11* .

The characteristic impedance for this type of trace can be expressed as

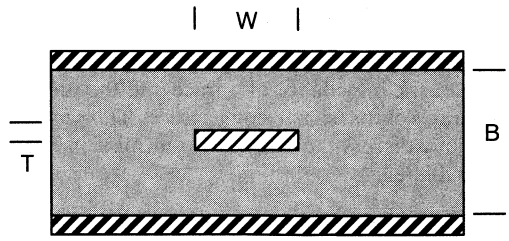


Figure 11. Strip Line

$$Z_0 = \frac{60}{\sqrt{\varepsilon_r}} \ln\left[\frac{4B}{0.67\pi W(0.8 + \frac{T}{W})}\right] \quad \text{Eq. 9}$$

For cases when

$$\frac{W}{B - T} < 0.35 \quad \text{and} \quad \frac{T}{B} < 0.25$$

With a propagation delay of

$$t_{pd} = 1.017\sqrt{\varepsilon_r} \quad \text{Eq. 10}$$

Figure 12 shows how the impedance of a strip line trace varies with the trace width and dielectric thickness. The graph shows that to create a strip line trace with a 50Ω impedance in a multilayer board with 10 mils of epoxy between layers requires approximately a 7-mil trace. This assumes a trace thickness of 1.4 mils and a dielectric constant of 4.5.

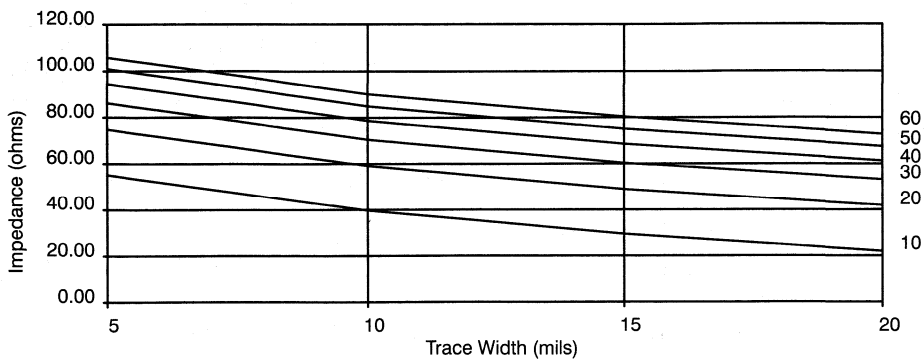


Figure 12. Impedance vs. Trace Width over Dielectric Thickness (Strip Line)

Transmission Line Effects

The previous section discussed how to calculate the characteristic impedance of a trace and the propagation rates of that signal along a trace. This section will briefly discuss the effects of transmission lines on signal integrity.

The cause of transmission line effects is impedance mismatches. These mismatches occur because of the impedance differences between the transmission line and the source and load. They are also caused by impedance discontinuities and unequal loading of the transmission line along its length. Transmission line stubs and vias are examples of impedance discontinuities otherwise known as impedance bumps.

Any time the impedance along a transmission line changes, a voltage reflection will occur. *Figure 13* shows a simplified diagram of the transmission line environment that will be used to illustrate the concept of voltage reflections.

At time 0, the voltage source provides a current source to the load equal to

$$I_S = \frac{V_S}{Z_S + Z_0'} \qquad \text{Eq. 11}$$

Since no voltage is dropped across the transmission line (assume a lossless transmission line as before), the total supplied current must be utilized by the load. If the load resistance is not equal to the characteristic impedance of the trace, a voltage reflection will occur. The reflection coefficient at the load is expressed as

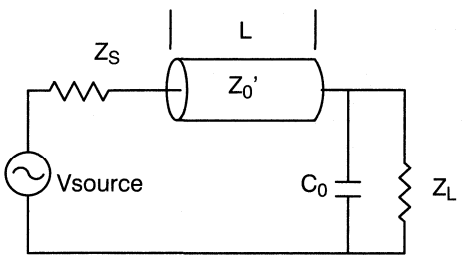


Figure 13. Simplified Transmission Line

$$\varrho_L = \frac{reflected\ voltage}{incident\ voltage} = \frac{V_{S1}}{V_S} = \frac{Z_L - Z_0'}{Z_L + Z_{0'}} \qquad \text{Eq. 12}$$

and the reflection coefficient at the source is specified as

$$\varrho_S = \frac{reflected\ voltage}{incident\ voltage} = \frac{V_{S2}}{V_{S1}} = \frac{Z_S - Z_0'}{Z_S + Z_{0'}} \qquad \text{Eq. 13}$$

If the load has a lower impedance than the characteristic impedance of the transmission line, then a negative voltage reflection will be sent back to the load, indicating that the load over used the available current. If the load has a higher impedance than the characteristic impedance of the transmission line, then a positive voltage will be sent back to the source, indicating that the load under used the amount of available current.

The following example shows the magnitude of these voltage reflections in a typical unterminated transmission line being driven by a CY7B991. For this example, assume the parameters have values listed in in *Table 1.*

Table 1. Sample Parameters

Param	Description	Value
ε_r	Dielectric Constant	4.4
H	Distance from ground	20 mils
T	Trace Thickness	1.4 mils
W	Trace Width	12 mils
Z_{PUS}	Source Pull-Up	25Ω
Z_{PDS}	Source Pull-Down	10Ω
L	Trace Length	6”
R_L	Load Impedance	12 KΩ
t_R	Output Rise Time	1.5 ns
t_F	Output Fall Time	1.5 ns

Solving the appropriate equations for Z_0, t_{PD}, and Z_0 yields

$$Z_0 = \frac{87}{\sqrt{4.4 + 1.41}} \ln\left(\frac{5.98 * 20}{0.8 * 12 + 1.4}\right) = 86.26\Omega \qquad \text{Eq. 14}$$

$$t_{pd} = 1.017\sqrt{0.475 * 4.4 + 0.67} = 1.66\ ns/foot \qquad \text{Eq. 15}$$

From this, the load and source reflection coefficients can be calculated:

$$\varrho_{SLH} = \frac{25 - 86.26}{25 + 86.26} = -0.55 \quad \text{Eq. 16}$$

$$\varrho_{SHL} = \frac{10 - 86.26}{10 + 86.26} = -0.79 \quad \text{Eq. 17}$$

$$\varrho_{L} = \frac{120,000 - 86.26}{120,000 + 86.26} = 1.00 \quad \text{Eq. 18}$$

Two source reflection coefficients need to be calculated for the LOW-to-HIGH output transition and the HIGH-to-LOW output transition. The reason for this is that TTL-style output drivers have different resistances depending if they are driving HIGH or LOW. In the calculation of the source coefficients, the actual output impedances for the output drivers and the input impedance of the REF input of the PSCB were used. They are listed in *Table 1.*

This analysis assumes a transmission line exists. To confirm this, the propagation delay is used in *Equation 2:*

$$MIN[t_R, t_F] < 2 * t_{pd}'$$
$$1.5 < 2 * 1.69 \quad \text{Eq. 19}$$

A complete DC analysis of voltage reflections on this type of transmission line is not conducted here. Refer to Reference 4. at the end of this note for a sample analysis. It is important to note from this that the voltage reflection at the load is positive and the reflection at the source is negative. The opposite signs of these two reflection-coefficients and the magnitude of these two constants indicate that ringing along the transmission line could potentially cause unwanted clocks pulses to be seen at the load.

Figure 14 shows the effects of an unterminated transmission line. This plot is taken from a Tektronix DSA 602A Digitizing Signal Analyzer. This plot indicates that the the voltage scale is 1V/div and the time scale is 5 ns/div. The ground signal is indicated by the ground symbol found on the left-hand side of the figure, three voltage divisions from the bottom. The scope trigger, indicated by an arrow, is the 1.5V level of the source waveform.

This analyzer is capable of taking measurements on the waveforms. The bottom quarter of the plot displays the various measurements taken on the source waveform. The rise and fall measurements are taken between the 0.8V and 2.0V level, which is consistent with the measurement points of the CY7B991. The pulse width measurement is taken at the 1.5V level and is used to indicate the duration of time that the waveform spends above 1.5V. This is used to calculate the output duty cycle variation. The last measurement shown on this plot is the frequency. In this case, a 25-MHz source waveform is used as the trigger.

The source waveform shown in *Figure 14* is the 4Q0 output of a typical CY7B991-5. The load waveform is the end of 18″ of coaxial cable having a characteristic impedance of 50Ω. In this particular example the load is completely unloaded except for the scope probe and associated connector (approximately 5 pF). As can be seen, both the source and the load are riddled with transmission line effects. This is caused by the positive voltage reflection from the load due to the infinite load impedance and the negative voltage reflection from the source due to the source impedance being less than the characteristic impedance of the coaxial cable. Notice that an additional clock pulse occurs (below the word "load" in the figure) due to voltage reflections.

A complete analysis of the voltage reflections causing this waveform would be extremely boring and

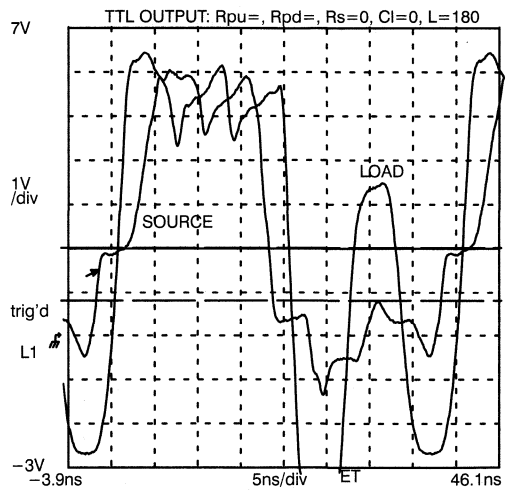


Figure 14. Unterminated Transmission Line

yield little additional information aside from the fact that this is an unacceptable clock waveform. The next section will give some methods for eliminating voltage reflections by terminating the transmission line.

Transmission Line Termination

The goal of transmission line termination is to make the source and/or load impedance match the characteristic impedance of the transmission line, insuring an optimal delivery of signal to the load. Two types of termination will be discussed: parallel termination and series termination.

Parallel Termination

Parallel termination, also known as Thevenin termination, attempts to match the load impedance with that of the transmission line. It is accomplished by placing a pull-up and pull-down resistor pair at the end of the transmission line nearest the destination as shown in *Figure 15*. The Thevenin equivalent resistance must be equal to Z_0 and the Thevenin voltage must be somewhere near the middle of the normal TTL voltage swing. Since typical CMOS inputs have steady-state impedances in the 100kΩ to 1MΩ the Thevenin resistance can be simplified to R_{PU} and R_{PD} in parallel. *Table 2* gives the recommended parallel termination resistor values for both the CY7B991 (TTL) and the CY7B992 (CMOS) devices for given transmission line impedances (Z_0).

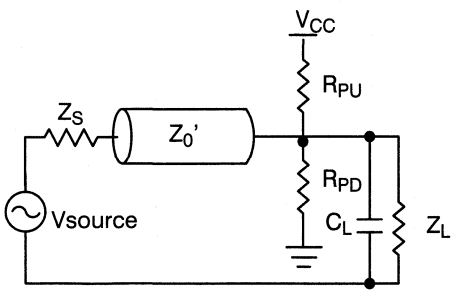


Figure 15. Parallel Termination

Table 2. PSCB Parallel Termination Recommendations

Z_0	TTL (R_{PU}/R_{PD})	CMOS (R_{PU}/R_{PD})
50Ω	130/91	100/100
65Ω	158/111	130/130
75Ω	182/128	150/150
100Ω	243/170	200/200

Figure 16 shows a 100Ω parallel termination of a transmission line with a 50Ω impedance. This figure is a plot taken from the same analyzer mentioned above. As before, the source waveform is taken at the RoboClock output pin and the load waveform is taken at the end of 18″ of coaxial cable with a 50Ω impedance. This figure shows the positive voltage reflection from the load because the load impedance is greater than the trace impedance. This reflection manifests itself as the source waveform stepping up to the 3.7V level as shown in *Figure 16*. The magnitude of the intitial source waveform in transmission line environment can be calculated by determining the the current in the transmission line before switching. The reflected voltage can be calculated by multiplying the incident voltage by the calculated reflection coefficient.

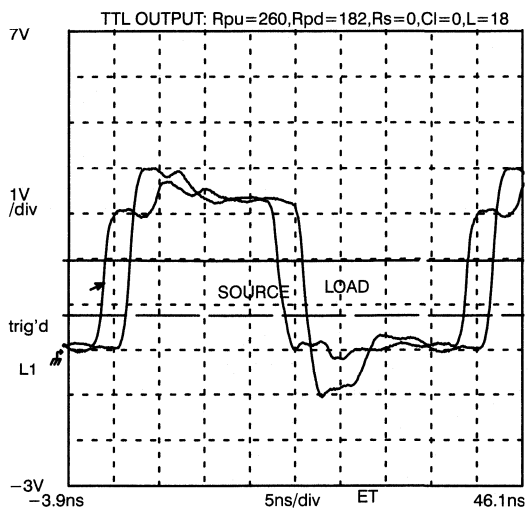


Figure 16. Unloaded 100Ω Parallel Termination

This waveform looks much better than in the previous unterminated case, but there is still a 1V undershoot at the falling edge of the waveform that may cause problems in some devices due to substrate current generation.

Figure 17 shows a 15Ω parallel termination. This transmission line is under terminated due to the fact that the load impedance is much less than the characteristic impedance of the trace. This manifests itself as a negative voltage reflection from the load. While the rising edge of this trace looks fairly good, the falling edge will cause problems. Notice that the negative voltage reflection ringing during the falling edge causes a positive-going transition in the threshold region of the load. This could cause spurious clocking.

Figure 18 shows a unloaded terminated transmission line. This transmission line, unlike in the other three cases, is terminated in its characteristic impedance. Both the source and the load waveforms are unaffected by voltage reflections. The only evidence of transmission line effects is the ringing during the HIGH and LOW time of the waveform, caused by slight impedance mismatches between the source, load, and transmission line, and the impedance "bumps" caused by the impedance differences at the various probe and coax connectors along this transmission line.

This ringing is also caused by the imperfect nature of terminating a transmission line with a complex load and source impedance. Each of the capacitances hanging on this transmission line have different impedances based on the input frequency. And, since the input waveform is a wideband signal containing many frequency components, a simple termination as given here will not eliminate all voltage reflections. The important thing to note is that this termination gives good results on the given transmission line.

Figure 19 shows a 50Ω parallel termination with a 22-pf load. Notice the voltage spikes on the source waveform. This is caused by the impedance of the load capacitor at a particular frequency being incorrectly terminated and causing a voltage reflection.

Figure 20 shows a 50Ω parallel termination with a 50-pF load. Again, the source waveform exhibits voltage glitches caused by brief negative voltage reflections, but in this case the voltage reflections come dangerously close to the input threshold region of a device that was placed on the transmission line somewhere between the source and the load shown in the figure. For this reason, it is recommended that all transmission lines drive either single loads or loads that are lumped at the end of the transmission line. If not, voltage spikes, even in

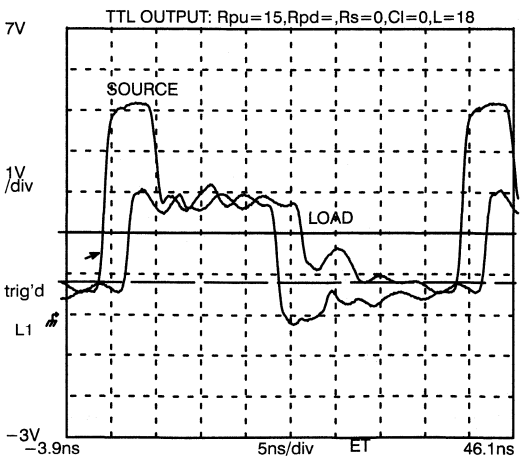


Figure 17. Unloaded 15Ω Parallel Termination

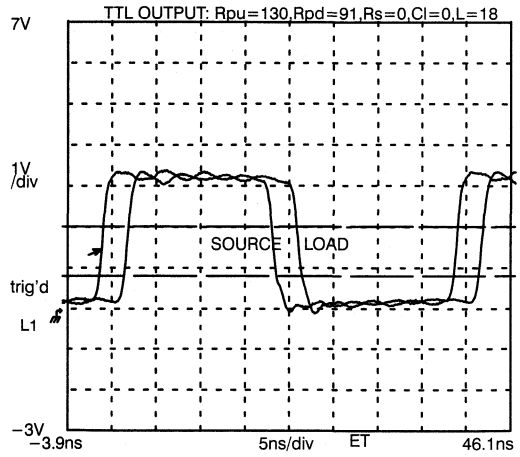


Figure 18. 50Ω Parallel Termination, Unloaded

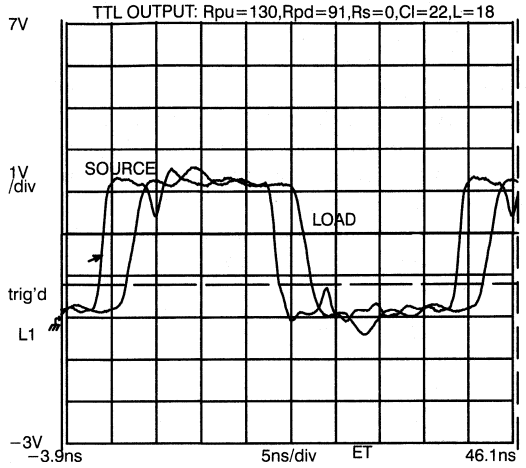


Figure 19. 50Ω Parallel Termination, 22-pF Load

reasonably well terminated transmission lines, can cause unwanted clocking.

Series Termination

The purpose of series termination is to match the source impedance with the transmission line impedance as shown in *Figure 21.* This will prevent voltage reflections occurring from the load will not reflect back from the source. The value of R_S is chosen such that the series combination of the output impedance

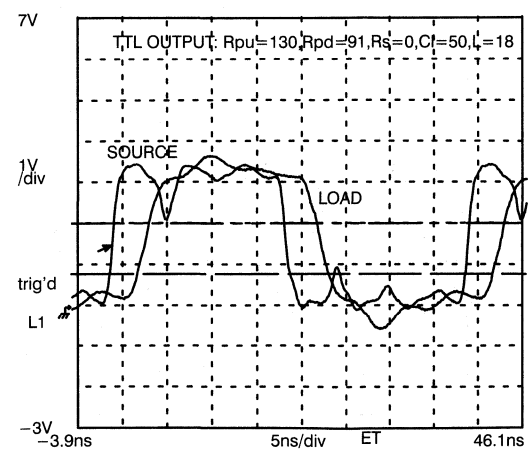


Figure 20. 50Ω Parallel Termination, 50-pF Load

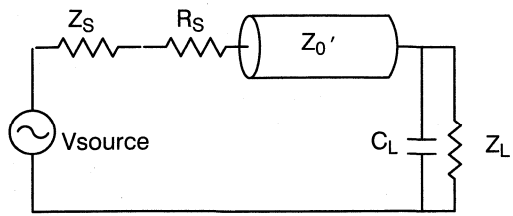


Figure 21. Series Termination

of the source devices and R_S is equal to Z_0. This series termination will absorb any voltage reflections from the load. This, however, is not a simple task.

TTL outputs have different LOW-to-HIGH and HIGH-to-LOW output impedances. The output drive has an asymmetrical output impedance. *Figure 22* shows the HIGH-to-LOW linearize Voltage vs. Current (V-I) curve for a typical CY7B991 output. The output impedance that should be used is the resistance when the output is LOW (less than 0.45V). *Figure 23* shows the LOW-to-HIGH linearized curve. The output impedance that should be used in this case is the resistance when the output is HIGH (greater than 2.4V). For these curves the output high resistance (27Ω) and the output LOW resistance (7Ω) can be determined.

Figure 24 shows an unloaded, 100Ω series terminated transmission line. This figure shows the classic "stair stepping" that occurs when a transmission line is terminated with series resistance that is too large. Several back-and-forth voltage reflections

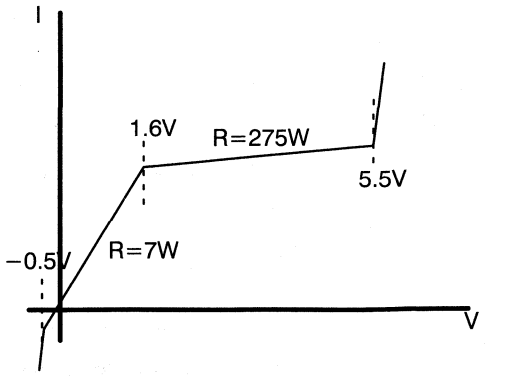


Figure 22. Output LOW Linearized V-I Curve

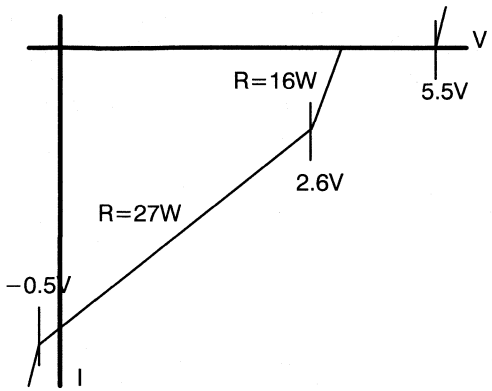


Figure 23. Output HIGH Linearized V-I Curve

are required before the load rests at its final voltage. This improper termination can cause duty-cycle distortion, increased rise and fall times, and unexpected clocking.

Figure 25 and *Figure 26* show unloaded and loaded 50Ω series terminated transmission lines, respectively. The resultant waveforms look much better in this case because the terminating resistor more closely matches the characteristic impedance.

Figure 27 shows a 27Ω terminated transmission line with a 22 pF load. The rising edge of this waveform looks much better than either of the previous two

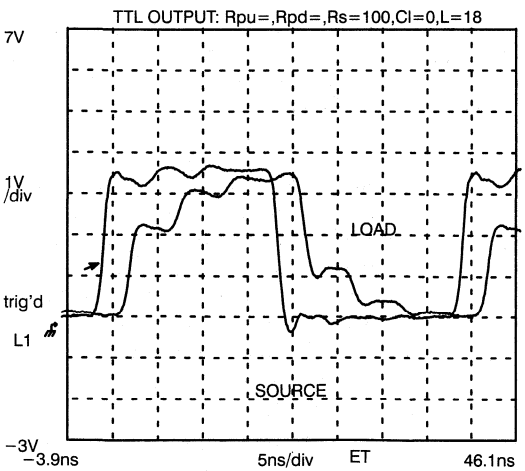


Figure 24. 100Ω Series Terminated, Unloaded

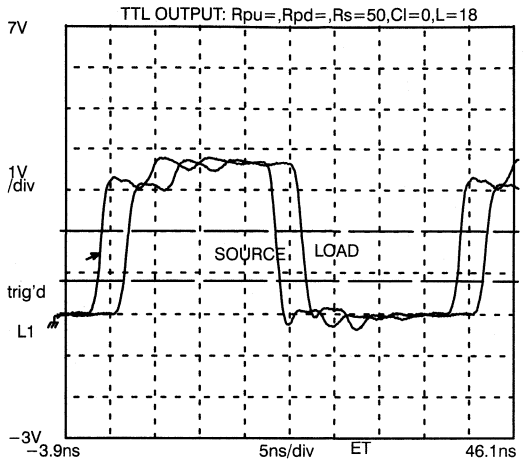


Figure 25. 50Ω Series Terminated, Unloaded

cases, but the falling edge has more undershoot than in the 50Ω termination example. The reason for this is that on the LOW-to-HIGH output transition the 27Ω terminating resistor plus the 27Ω output impedance closely match Z_0, but on the HIGH-to-LOW output transition the 27Ω resistor plus the 7Ω output impedance do not exactly match Z_0. With series termination a trade off has to be made between the LOW-to-HIGH transition and the HIGH-to-LOW transition.

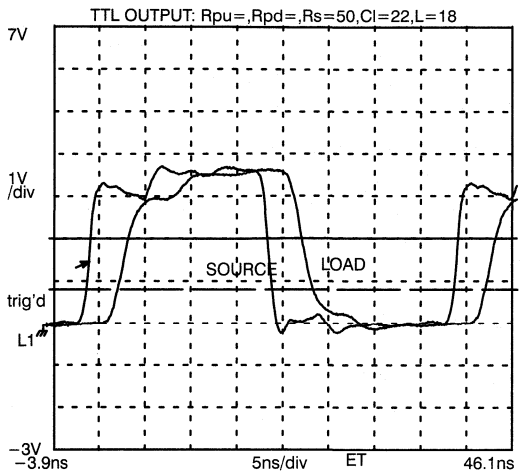


Figure 26. 50Ω Series Terminated, 22-pF Load

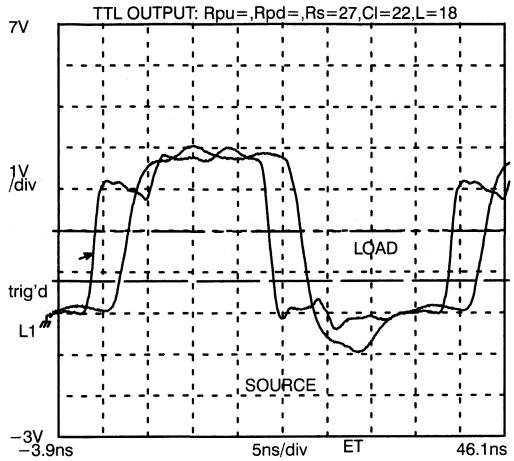


Figure 27. Series Terminated, 22-pF Load

When using series termination, no loads may exist along the transmission line. All loads have to be located at the end of the trace. The reason for this is that the series-terminating resistor acts like a voltage divider. Any loads not located at the end of the trace will see a voltage at some indeterminate level until the voltage reflection from the load builds the voltage level to its final resting value.

An advantage of series termination over parallel termination is that there is no DC power consumption. In a parallel termination, whether the output device is driving HIGH or LOW, there will always be current flow that does not drive the load but simply establishes the terminations.

Another note on transmission lines is that the rise time of the output waveform does not depend on any transmission-line loading considerations. By looking at the rise times of the source waveform in the previous example, it is clear that the method of transmission line termination and the output loading play a negligible role in the output rise time. In a properly terminated transmission line, the output rise and fall time will be a function of the characteristic impedance of the trace and the capacitive loading of the load.

The recommendation for terminating transmission lines is either a parallel termination with an R_{TH} = Z_0 and a V_{TH} = 2.06V or a series termination where $R_S = Z_0 - 20\Omega$. If more than one load has to be driven by a single device output, make sure that all loads are located very near the end of the line, or create a "star" layout by having each load have its own trace starting at the RoboClock output. Terminate each trace as if it were the only load being driven. In the case of multiple loads being driven by a single output, the series resistor should be calculated with

$$R_S = Z_0 - \frac{20}{\#\ of\ Traces} \qquad \text{Eq. 20}$$

All lines must be matched or the loads will "talk" to each other through the voltage reflections.

Trace impedances below 50Ω can be driven by tying more than one output together. For example, a 25Ω trace can be driven by tying two outputs of the same output pair together.

Never "daisy-chain" loads together. This will not only immediately add load-to-load skew, but it will also cause unpredictable transmission line effects.

Detailed Description

RoboClock is an eight-output clock driver device. It differs from traditional clock drivers and buffers in that its outputs, while having very low output skew, can also be phase adjusted, inverted, divided, and multiplied. These capabilities would be impossible to implement in a device such as a simple redrive buffer like a 74F244. A 74F244, while potentially providing very low output skew, does not have the capability to dynamically phase adjust its outputs.

Phase adjustment allows outputs to shift in time relative to a reference point. The input clock to the device is usually taken as this reference point. Phase adjustment is useful for compensating for differences in trace delay from one load to the next, and also for equalizing differences in set-up and hold time between load devices. A 74F244 would have great difficulties shifting the arrival time of its outputs both in the positive sense (output edge arrives later that the reference edge) and in the negative sense (output edge arrives before the reference edge). In order for a 74F244 to accomplish this task, it would have to predict the time at which the next

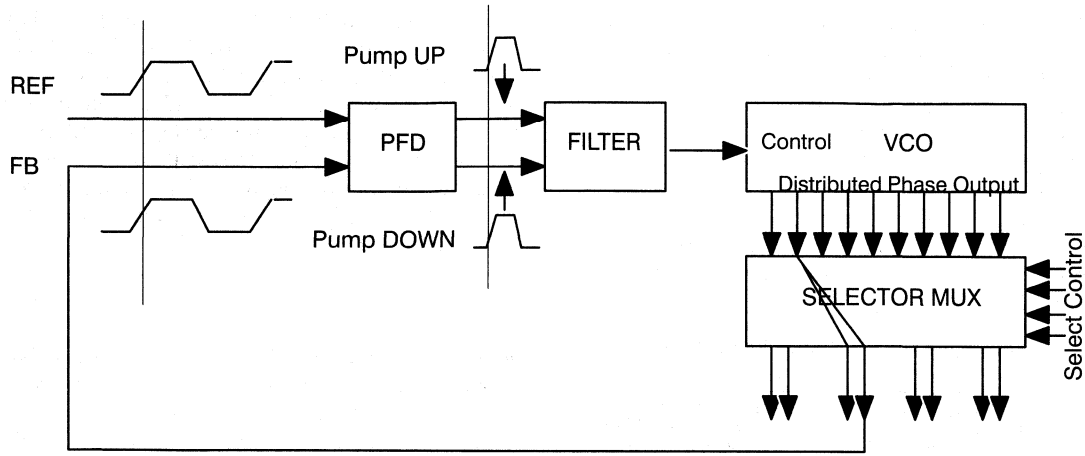


Figure 28. Simplified PLL Architecture

input edge would occur. Obviously, a new approach is needed.

PLL Operation

RoboClock includes a phase-locked loop (PLL) to achieve zero propagation delay. A completely integrated PLL allows you to align both the phase and the frequency of the reference (REF) inputs with an output. With this approach, the next occurrence of an input edge can be predicted with great accuracy while maintaining very low propagation delay through the device.

The PLL has three distinct parts; the phase/frequency detector, the filter, and the distributed-phase clock oscillator (more simply known as a voltage-controlled oscillator). In order for the PLL to align the REF input with any output, an output must be selected to be fed back to the input of the PLL. This input (FB) is then used as the alignment on which all other outputs are based. (See *Figure 1*).

Phase Detector and Filter

Figure 28 shows a simplified view of the RoboClock PLL architecture. The Phase Frequency Detector (labeled PFD) evaluates the rising edge of the REF input with respect to the FB input. If the REF input occurs before the FB input, indicating that the Voltage Controlled Oscillator (VCO) is running too slow, the PFD produces a Pump Up signal that lasts until the rising edge of the FB input. If the FB input occurs before the REF input, on the other hand, the PFD produces a Pump Down signal that is triggered on the rising edge of the FB input and lasts until the rising edge of REF. This Pump Down pulse forces the VCO to run slower. In this way, the PFD forces the VCO to run faster or slower based on the relationship of the REF and FB inputs. In the absence of a REF input, the device will function at approximately its slowest operating speed.

The Filter converts these Pump Up and Pump Down signals into a single control voltage. The magnitude of this voltage is dependent on the number of previous Pump Up and Pump Down pulses that have occurred. The range of the voltage produced by the filter is guaranteed to be able to force the VCO into any frequency within the selected frequency range.

Distributed-Phase Clock Oscillator

Figure 29 shows the Distributed-Phase Clock Oscillator ring and the Output Adjust Matrix. The RoboClock Distributed Phase Clock Oscillator (also known as a ring oscillator) has three frequency ranges of operation. These frequency ranges are se-

lected with the FS pin with range values shown in *Table 3.* At first glance, it may seem odd that a single pin (FS) has three possible selections. These three-state inputs are another feature of RoboClock. All function select inputs (FS, TEST, and xFn) have the ability to be connected to one of three states; HIGH, LOW, and MID. HIGH indicates a connection to VCC, LOW indicates a connection to Ground, and MID indicates an open connection. When a three-level input is left unconnected, internal resistors pull this input to approximately $V_{CC}/2$.

Table 3. Frequency Range Select and t_U Calculation

FS[2]	f_{NOM} (MHz) Min.	f_{NOM} (MHz) Max.	$t_U = \frac{1}{f_{NOM} \times N}$ where N =	Approximate Frequency (MHz) At Which t_U = 1.0 ns
LOW	15	30	44	22.7
MID	25	50	26	38.5
HIGH	40	80	16	62.5

The three different frequency ranges correspond to the number of stages in the oscillator. When FS is connected to ground, the oscillator contains its maximum number of stages: 22. When FS is left unconnected, the oscillator contains 13 stages. And when FS is connected to ground, the oscillator contains its minimum number of stages: 8. The operating frequency of the oscillator can be calculated with the following formula:

$$f = \frac{1}{N * t_U} \qquad \text{Eq. 21}$$

where N is the number of stages and t_U is the delay through each stage. The reason that N at the bottom of *Equation 21* is twice the number of stages in the oscillator is because, in order for the ring to oscillate, first the true and then the inverted signal must pass through each stage of the oscillator. This is accomplished through an inversion from the last stage to the first stage.

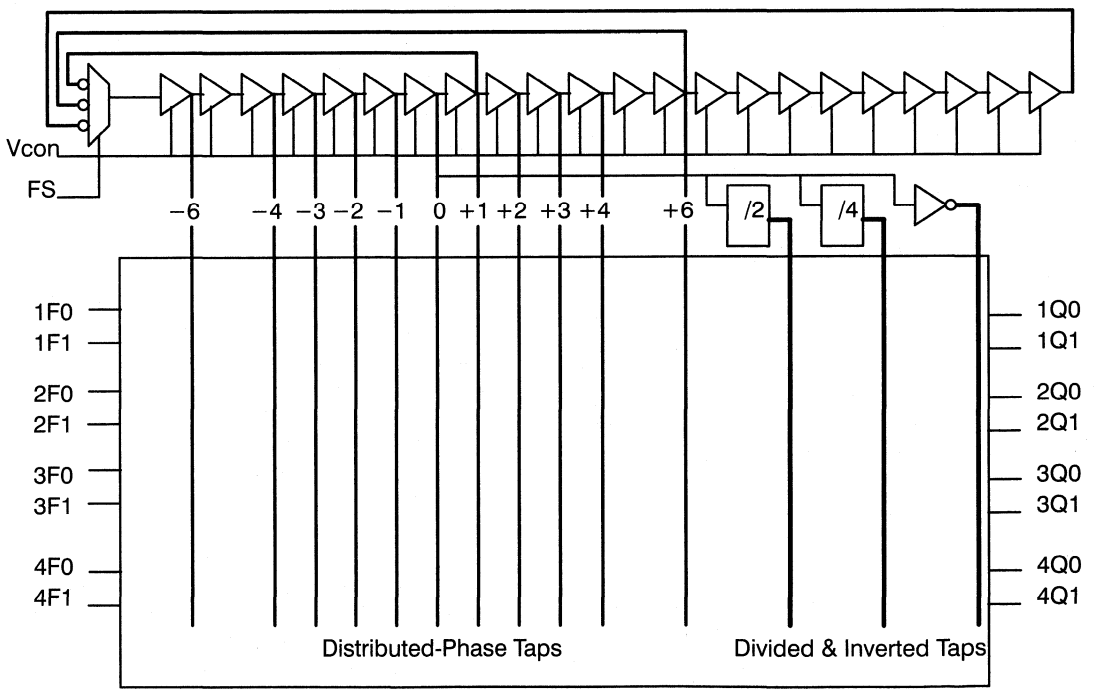


Figure 29. Distributed-Phase Clock Oscillator and Output Adjust Matrix

For example, if the delay through each stage is exactly 1 ns and the FS pin was tied to ground, then the operating frequency would be

$$f = \frac{1}{2 * 22 * 1ns} = 22.7\ MHz \qquad \text{Eq. 22}$$

The delay through each stage is controlled by the voltage on the V_{CON} line, which is simply the voltage generated by the PLL filter. From *Table 3* it is obvious that some overlap exists between the various frequency ranges. This allows a choice of stage delays within some frequency ranges, which in turn allows system designers a choice of two different increments of phase adjustment.

Within the first thirteen stages of the oscillator, 11 taps are sent to the Output Adjust Matrix. These taps represent various phase relationships to the center, or 0 time unit (t_U) position. The taps range from $-6\ t_U$ to $+6\ t_U$, as shown in *Figure 29.* This allows the outputs to shifted, either early or late, with respect to the FB input to adjust for various system requirements.

The value of t_U shown in *Figure 30* is determined by the operating frequency and the number of stages in the distributed phase oscillator. The formula for calculating t_U is shown in *Table 3* and given here:

$$t_U = \frac{1}{f_{nom} \times N} \qquad \text{Eq. 23}$$

For example, *Equation 24* calculates the stage delay (t_U) when the ring oscillator is running at 25 MHz and the FS pin is tied to ground

$$t_U = \frac{1}{25MHz \times 44} = 0.91\ ns \qquad \text{Eq. 24}$$

The value of t_U, on the other hand, if the FS pin were left unconnected, is

$$t_U = \frac{1}{25MHz \times 26} = 1.54\ ns \qquad \text{Eq. 25}$$

This shows that at the same operating frequency (f_{NOM}), two different stage delays are possible, depending on the connection of the FS pin.

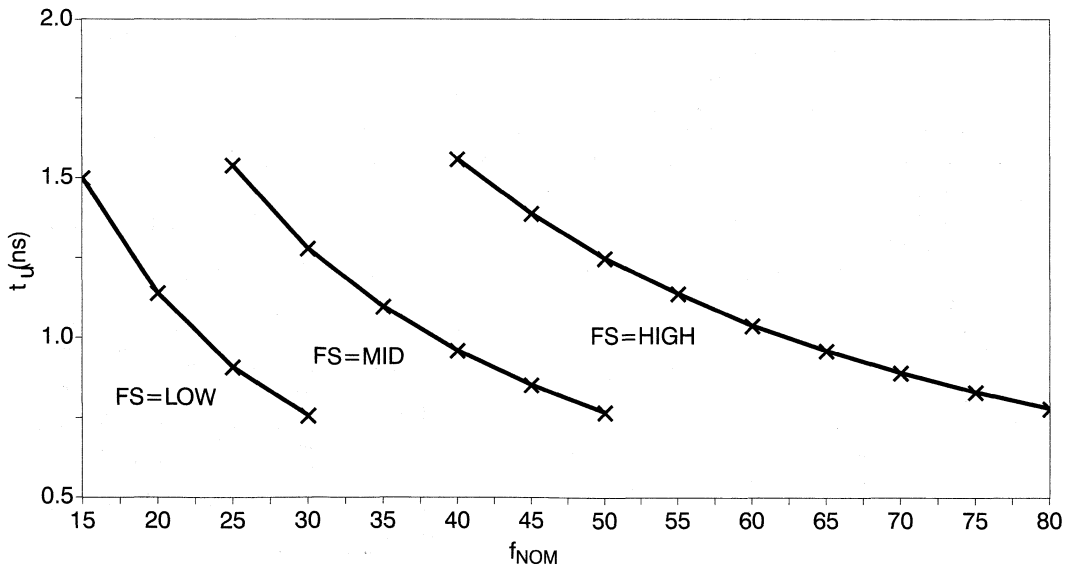


Figure 30. Time Unit (t_U) vs. Frequency

Output Adjust Matrix

The output adjust matrix allows the outputs to be configured in up to 26,000 ways (more on this later). The output options are generated by the Distributed Phase Clock Oscillator, and selected by the output function select (xFn) inputs.

In addition to the 11 taps from the distributed phase oscillator, the Output Adjust Matrix contains a divide-by-two option, a divide-by-four option, and an invert option.

The eight RoboClock outputs are configured as four output pairs. Each member of a pair of outputs operates identically to the other. The output adjustment for each output pair is controlled by its associated pair of function select inputs. For example, the 1Qn outputs are controlled by the 1Fn inputs.

The function select inputs are three-state inputs that operate in the same manner as the FS input. These inputs can be tied HIGH, tied LOW, or left unconnected (MID). The three-level input capabilities of the function select inputs allow each output to have nine different output selections with the use of only two pins.

Each pair of outputs has nine different possible output timing positions based on the appropriate connection of the function select input. The possible output combinations are shown in *Table 4.* These output adjustment configurations assume that an output with a 0 t_U configuration is used as the FB input. Output adjustment configurations with a non-0 t_U tap output selected as the FB input will be discussed in the section titled Change in Operation with FB selection.

The following example refers to *Figure 31. A*ssume that 2Q0 is used as the FB input and that 2F1 and 2F0 are both left unconnected. This will select both of the 2Qx outputs to have a 0-phase-adjusted output (0 t_U), and by connecting 2Q0 to the FB input it will also force these outputs to be phase and frequency aligned with the REF input.

Table 4. Output Adjustment Configurations

Function Selects		Output Functions		
1F1, 2F1, 3F1, 4F1	**1F0, 2F0, 3F0, 4F0**	**1Q0, 1Q1, 2Q0, 2Q1**	**3Q0, 3Q1**	**4Q0, 4Q1**
LOW	LOW	$-4t_U$	Divide by 2	Divide by 2
LOW	MID	$-3t_U$	$-6t_U$	$-6t_U$
LOW	HIGH	$-2t_U$	$-4t_U$	$-4t_U$
MID	LOW	$-1t_U$	$-2t_U$	$-2t_U$
MID	MID	$0t_U$	$0t_U$	$0t_U$
MID	HIGH	$+1t_U$	$+2t_U$	$+2t_U$
HIGH	LOW	$+2t_U$	$+4t_U$	$+4t_U$
HIGH	MID	$+3t_U$	$+6t_U$	$+6t_U$
HIGH	HIGH	$+4t_U$	Divide by 4	Inverted

If, in this scenario, 1F1 were tied to ground and 1F0 were left unconnected, then the 1Q0 and 1Q1 output edges would precede the output used as the FB input (2Q0 in this case) by three time units. Alternately, if 1F1 were tied HIGH and 1F0 were again left unconnected, then the 1Q0 and 1Q1 output edge would follow the 2Q0 output by three time units.

If 3F1 and 3F0 were both tied HIGH, then the 3Qn outputs would both operate at one-quarter the frequency of the 2Qn outputs. And if the 4Fn function select inputs were both tied LOW, the 4Qn outputs would both operate at one-half the frequency of the 2Qn outputs.

An important point to note is the frequency and phase relationship between the 1/2 and 1/4 outputs

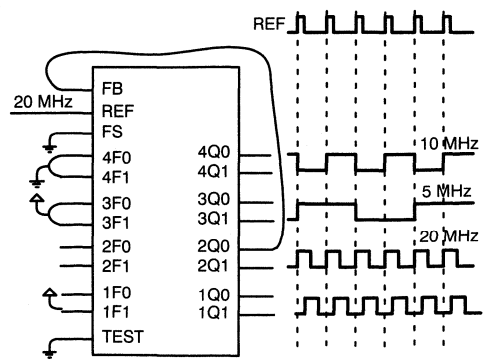


Figure 31. Frequency Divider Connections

(3Qn and 4Qn). The divide-by-two and divide-by-four outputs fall at the same time, but never rise at the same time. This feature of RoboClock makes it possible to use the rising edges of the 1/2 frequency and 1/4 frequency outputs without concern for skew mismatch. It also provides the ability to clock different parts of the system on different phases of the master clock.

The previous example showed the phase shifting and frequency division capabilities of RoboClock. Another output feature available on the 4Qx outputs is phase inversion. This output adjustment is configured by tying both 4F1 and 4F0 inputs HIGH. In this mode the 4Qx outputs will have and inverted sense with respect to the FB input.

Change in Operation with FB Selection

The previous discussion assumed that an output with a 0 t_U phase adjustment was used as the FB input. With this assumption, RoboClock has nearly 3000 different configurations. This is calculated by taking the number of possible configurations of each output pair (9) to the number of outputs pairs not being used as the FB input (3) times the number of choices of output pairs to be used as the FB input.

$$Combos = 9^3 * 4 \qquad \text{Eq. 26}$$

If the output used as the FB input is also phase or frequency adjusted, then RoboClock offers over 26,000 different configurations.

Phase Adjusted Output Used as FB Input

By feeding back an output that was selected for 0 t_U, all of the other outputs were referenced from the 0 tap position shown in *Figure 29*. This is due to the fact that the PLL aligns the REF input and the FB input in both phase and frequency.

It is not necessary to use an output with a 0 t_U configuration as the FB input. An output with any configuration can be used as the FB input. For example, if an output with a −3 t_U tap was used as the FB input, the PLL would align this output with the REF input. It would no longer exhibit a shift of −3 time units when compared with REF. The output used as the FB input is always aligned with REF.

By using an output with this configuration as the FB input, all other outputs are now referenced to this tap position within the ring oscillator. The possible tap selections for the other outputs are still the same as in the 0 tap used as FB case, but now they have a −3 tap time reference. The +6 tap can still be selected as an output configuration, but it will occur 9 time units after the FB output. The −6 tap can also be selected as an output configuration, but instead of occurring 6 time units before the corresponding edge of FB, it will occur 3 time units before the output used as the FB input.

Tables 5 through *7* illustrate the various possible output configurations with different FB selections. *Table 5* gives the 2Qn, 3Qn, and 4Qn output configurations when a 1Qn output is used as the FB input. It also gives the 1Qn, 3Qn, and 4Qn output configurations when a 2Qn output is used as the FB input. The reason for this is that the 1Qn and 2Qn outputs have the same possible configurations. If either is used as the FB input, the other outputs will have the same output configuration options.

Table 5 is broken into three parts corresponding to the configurations for each of the three pairs of outputs not used as the FB input. The leftmost two columns of each table indicate the various configurations of the FB input, and the right portion of the table gives the output possibilities for the given output.

For example, the first part of *Table 5* gives the possible output configurations for the 2Qn outputs assuming that a 1Qn output is used as the FB input. Alternately, this table gives the output configurations for the 1Qn outputs assuming a 2Qn output is used as the FB input. This is true because the 1Qn and 2Qn outputs have the same possible output configurations as shown in *Table 4*. For the remainder of this example, a 1Qn output is assumed to be the FB input.

The left side of the table gives the function select input settings for the FB output. L represents a connection to ground, M represents an input left open, and H represents an input connected to V_{CC}. Once a selection is made for the function select inputs of

the FB output, all of the other outputs will be referenced to that tap.

For example, if the 1F1 input is tied to ground and the 1F0 input is left unconnected, then all of the other outputs will be referenced to the -3 t_U tap used as the FB input. The available configurations on the remaining outputs will remain the same as given in *Table 4*, but the values in this table will all be shifted by +3 because this is the number of stage delays between the new reference point and 0 t_U, the reference point of *Table 4*.

All of the possible output configurations can be found in the same row in the following tables as the selection made for the FB output. If 1Fn = LM (1F1 tied to ground, and 1F0 left unconnected), then the possible selections for the 2Qn output are from -1 t_U to $+7$ t_U. This is shown in the first part of *Table 5* as a shaded row. By connecting 2Fn = HM, the 2Qn outputs will have outputs that lag the FB outputs by 6 time units or by connectng 2Fn = LL the 2Qn outputs will precede the reference by 1 time unit (-1t). Once the output configurations for the FB input are made, all other outputs will be referenced to this new reference point.

The second part of *Table 5* gives the possible output configurations for the 3Qn outputs. Assuming a 1Qn (2Qn) output is used for the FB input, the 3Qn outputs can be phase shifted with a granularity of 2 time units from -3 to +9 with respect to the FB input. Additionally, the 3Qn outputs can be divided by two and divided by four, but since the reference point for the dividing circuit for these configurations is the 0 tap (as shown in *Figure 29*), these outputs are shifted by three time units with respect to the FB input.

The third part of *Table 5* gives the possible output configurations for the 4Qn outputs, again assuming that a 1Qn output is used as the FB input. This table looks much the same as the second part with the only exception being the last column. The only difference between the 3Qn and 4Qn outputs is the ability to divide by four or invert respectively. The last column shows how RoboClock can be configured to phase shift and invert an input signal.

Table 5. 1Qx or 2Qx Output Connected to FB Input (Part 1)

1Qn(2Qn)➧FB		2Qn(1Qn) Outputs with respect to FB										
		Input Select	2F1 (1F1)	L	L	L	M	M	M	H	H	H
1F1 (2F1)	1F0 (2F0)		2F0 (1F0)	L	M	H	L	M	H	L	M	H
L	L	Output Configuration		0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t	+8t
L	M			−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t
L	H			−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t
M	L			−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t
M	M			−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t
M	H			−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t
H	L			−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t
H	M			−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t
H	H			−8t	−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t

Table 5. 1Qx or 2Qx Output Connected to FB Input (Part 2)

1Qx(2Qx)◗FB		3Qn Outputs with respect to FB									
		Input Select	3F1	L	L	L	M	M	M	H	H	H
1F1 (2F1)	1F0 (2F0)		3F0	L	M	H	L	M	H	L	M	H
L	L	Output Configuration		+4t f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t f/4
L	M			+3t f/2	−3t	−1t	+1t	+3t	+5t	+7t	+9t	+3t f/4
L	H			+2t f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t f/4
M	L			+1t f/2	−5t	−3t	−1t	+1t	+3t	+5t	+7t	+1t f/4
M	M			0t f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t f/4
M	H			−1t f/2	−7t	−5t	−3t	−1t	+1t	+3t	+5t	−1t f/4
H	L			−2t f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t f/4
H	M			−3t f/2	−9t	−7t	−5t	−3t	−1t	+1t	+3t	−3t f/4
H	H			−4t f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t f/4

Table 5. 1Qx or 2Qx Output Connected to FB Input (Part 3)

1Qn(2Qn)➧FB		4Qn Output with respect to FB										
		Input Select	4F1	L	L	L	M	M	M	H	H	H
1F1 (2F1)	1F0 (2F0)		4F0	L	M	H	L	M	H	L	M	H
L	L	Output Configuration		+4t f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t INV
L	M			+3t f/2	−3t	−1t	+1t	+3t	+5t	+7t	+9t	+3t INV
L	H			+2t f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t INV
M	L			+1t f/2	−5t	−3t	−1t	+1t	+3t	+5t	+7t	+1t INV
M	M			0t f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t INV
M	H			−1t f/2	−7t	−5t	−3t	−1t	+1t	+3t	+5t	−1t INV
H	L			−2t f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t INV
H	M			−3t f/2	−9t	−7t	−5t	−3t	−1t	+1t	+3t	−3t INV
H	H			−4t f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t INV

Tables 6 and *7* have slightly different output configurations. These tables represent the possible output configurations when a 3Qn or 4Qn output is used as the FB input.

The first part of *Table 6* gives the possible output configurations for the 1Qn (2Qn) outputs when a 3Qn output is used as the FB input. When the 3Qn outputs are configured from $-6\ t_U$ (3Fn = LM) to $+6\ t_U$ (3Fn = HM) the 1Qn outputs have a range of $+10\ t_U$ (1Fn = HH) to $-10\ t_U$ (1Fn = LL). This, again, is because if the 3Qn outputs have a −6 tap reference and the +4 tap is selected for the 1Qn outputs, then the total time delay between the FB input and the 1Qn output is +10 time units. This feature gives RoboClock a tremendous phase adjustment range.

What if a divided output is used as the FB input? The last row of *Table 6* (Part 1) shows that if 3Fn = HH (3Qx in divide-by-four mode), then all of the outputs are multiplied by four. RoboClock has become a frequency multiplier. To understand why this happens, remember that the PLL aligns the FB with the REF input in both phase and frequency. Even though the 3Qn outputs were selected to divide by four, the PLL forces them to run at the same rate as the REF input. This means that, in order for these outputs to operate at this speed, the VCO itself must operate at four times the REF frequency.

The ability to multiply an input frequency is useful in board-level designs where the distribution of a low-frequency signal is needed to reduce EMI emissions or where a faster clock is needed to increase the operating frequency of state machine logic. *Figure 32* shows how RoboClock can be configured as a frequency multiplier and a phase adjuster. By selecting 3Fn = HH, RoboClock will multiply the REF frequency by four, by forcing its PLL to operate at four times the REF clock rate. 4Fn = LL se-

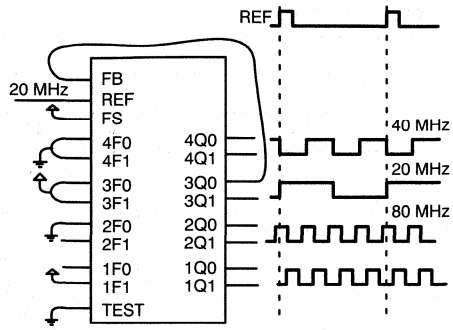


Figure 32. Frequency Multiplier with Phase Adjustment

lects the divide-by-two option at the 4Qn outputs. Since the PLL is operating at 80 MHz, the 4Qn outputs will operate at 40 MHz. Selecting 2Fn = ML configures the 2Qn outputs to precede the rising edge of the FB input by 1 time unit. And selecting 1Fn = HM makes the 1Qn outputs arrive 3 time units later than the FB input. Both the 1Qn and 2Qn outputs will run at 80 MHz. The xFn configurations for this example can be found in the shaded area of *Table 6*.

Table 7 appears very similar to *Table 6*. The first part gives the 1Qn and 2Qn output configurations when a 4Qn output is used as the FB input. The second part of the table gives the 3Qn output configurations when a 4Qn output is used as the FB input. The major variation is that if a 4Qn output is used as the FB input with 4Fn = HH, then all outputs will be inverted. Since the PLL phase aligns the REF and FB input, the 4Qn outputs will operate identically with the REF input. The other outputs will have a 180°phase shift from the REF input. This is useful for applications requiring more inverted clock signals than non-inverted clock signals.

Table 6. 3Qx Output Connected to FB Input (Part 1)

3Qn ➧ FB		1Qn (2Qn) Output with respect to FB										
		Input Select	1F1, (2F1)	L	L	L	M	M	M	H	H	H
3F1	3F0		1F0, (2F0)	L	M	H	L	M	H	L	M	H
L	L	Output Configuration		−4t f*2	−3t f*2	−2t f*2	−1t f*2	0t f*2	+1t f*2	+2t f*2	+3t f*2	+4t f*2
L	M			+2t	+3t	+4t	+5t	+6t	+7t	+8t	+9t	+10t
L	H			0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t	+8t
M	L			−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t
M	M			−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t
M	H			−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t
H	L			−8t	−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t
H	M			−10t	−9t	−8t	−7t	−6t	−5t	−4t	−3t	−2t
H	H			−4t f*4	−3t f*4	−2t f*4	−1t f*4	0t f*4	+1t f*4	+2t f*4	+3t f*4	+4t f*4

Table 6. 3Qx Output Connected to FB Input (Part 2)

3Qn ➧ FB		4Qn Output with respect to FB										
		Input Select	4F1	L	L	L	M	M	M	H	H	H
3F1	3F0		4F0	L	M	H	L	M	H	L	M	H
L	L	Output Configuration		0t	−6t f*2	−4t f*2	−2t f*2	0t f*2	+2t f*2	+4t f*2	+6t f*2	INV f*2
L	M			+6t f/2	0t	+2t	+4t	+6t	+8t	+10t	+12t	+6t INV
L	H			+4t f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t INV
M	L			+2t f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t INV
M	M			0t f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t INV
M	H			−2t f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t INV
H	L			−4t f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t INV
H	M			−6t f/2	−12t	−10t	−8t	−6t	−4t	−2t	0t	−6t INV
H	H			0t f*2	−6t f*4	−4t f*4	−2t f*4	0t f*4	+2t f*4	+4t f*4	+6t f*4	INV f*4

Table 7. 4Qx Output Connected to FB Input (Part 1)

4Qn ➧ FB		1Qn (2Qn) Output with respect to FB										
		Input Select	1F1, (2F1)	L	L	L	M	M	M	H	H	H
4F1	4F0		1F0, (2F0)	L	M	H	L	M	H	L	M	H
L	L	Output Configuration		−4t f*2	−3t f*2	−2t f*2	−1t f*2	0t f*2	+1t f*2	+2t f*2	+3t f*2	+4t f*2
L	M			+2t	+3t	+4t	+5t	+6t	+7t	+8t	+9t	+10t
L	H			0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t	+8t
M	L			−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t
M	M			−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t
M	H			−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t
H	L			−8t	−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t
H	M			−10t	−9t	−8t	−7t	−6t	−5t	−4t	−3t	−2t
H	H			−4t INV	−3t INV	−2t INV	−1t INV	0t INV	+1t INV	+2t INV	+3t INV	+4t INV

Table 7. 4Qx Output Connected to FB Input (Part 2)

4Qn ⧫ FB		3Qn Output with respect to FB										
		Input Select	3F1	L	L	L	M	M	M	H	H	H
4F1	4F0		3F0	L	M	H	L	M	H	L	M	H
L	L	Output Configuration		0t	−6t f*2	−4t f*2	−2t f*2	0t f*2	+2t f*2	+4t f*2	+6t f*2	0t f/2
L	M			+6t f/2	0t	+2t	+4t	+6t	+8t	+10t	+12t	+6t f/4
L	H			+4t f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t f/4
M	L			+2t f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t f/4
M	M			0t f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t f/4
M	H			−2t f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t f/4
H	L			−4t f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t f/4
H	M			−6t f/2	−12t	−10t	−8t	−6t	−4t	−2t	0t	−6t f/4
H	H			INV f/2	−6t INV	−4t INV	−2t INV	0t INV	+2t INV	+4t INV	+6t INV	INV f/4

Functional Implementations

Obviously, RoboClock has abilities to solve even the most complex problems. The challenge is to determine how to configure RoboClock to solve these problems. The following examples will give a brief overview of how to configure RoboClock to accomplish various tasks.

Low-Skew Clock Buffer

The easiest way to configure RoboClock is as a low-skew clock buffer, as shown in *Figure 33.* In this type of configuration, all xFn inputs are left open (unconnected) and the FB input can be taken from any output. The REF input waveform is shown with a very unequal duty cycle to illustrate the point that because RoboClock is a PLL-based clock buffer, the duty cycle of the outputs is 50/50 regardless of the duty cycle of the REF input. This feat would be impossible in a non-PLL based device.

If this implementation is used, either the 2Q1 or 3Q0 output is the most convenient choice to be used as the FB input because of their proximity. The shortest wire possible should be used to connect the xQn output to the FB input to avoid stub reflection effects on the actual clock line connected to the

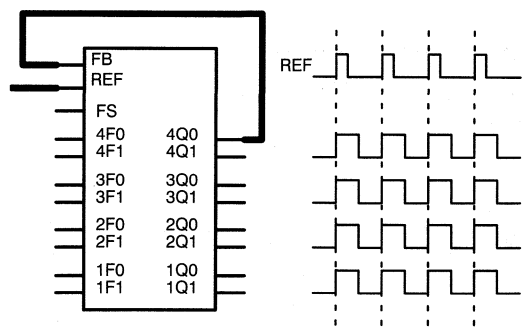


Figure 33. Low-Skew Clock Buffer

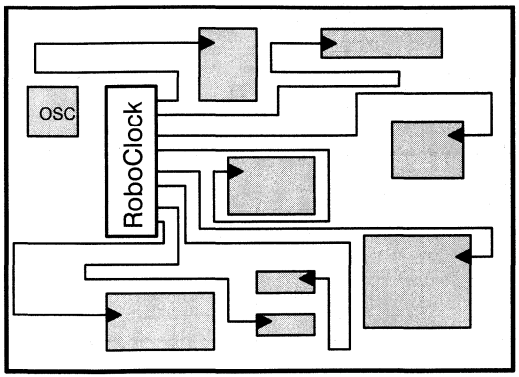


Figure 34. RoboClock in Low Skew Buffer Application

same xQn output *(Figure 2)*. This short route will reduce noise and transmission line effects from affecting the FB input.

This configuration assumes that the clock routes to the various loads they are driving are all the same length, so that each of the clocks arrives at its load at virtually the same time. *Figure 34* shows how RoboClock might be used in an application requiring only a device with low output skew.

Programmable Phase Adjustment

This type of application requires the use of RoboClock's additional features. RoboClock provides phase shifting in ranges from −12 time units to +12 time units. *Figure 35* shows RoboClock configured to phase shift its 2Qn outputs so that they lag the FB input, and to phase shift its 1Qn outputs so that they lead the FB input. Any output can be used as the FB input to implement phase shifting, but the greatest unidirectional shift (12 time units) will be achieved by selecting either the 3Qn or 4Qn outputs to be used as the FB input.

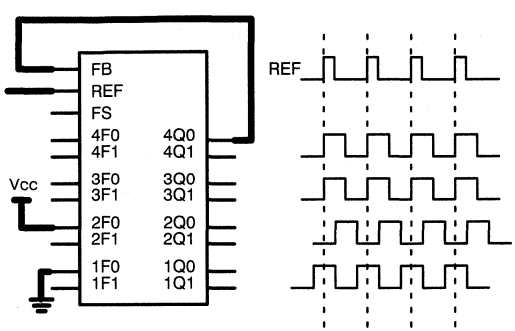


Figure 35. Programmable Phase Adjustment

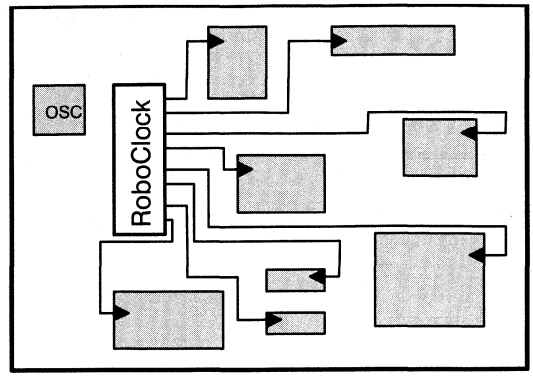


Figure 36. RoboClock in Programmable Phase Adjustment Application

Figure 36 shows RoboClock in a programmable phase adjustment application. This application differs from *Figure 34* in that the sophisticated phase adjustment abilities of RoboClock are used to compensate for trace delays and set-up and hold time mismatches.

Inverted Output Clock Driver

Figure 37 shows RoboClock configured as a inverted output clock driver. 4Fn = HH configures the 4Qn

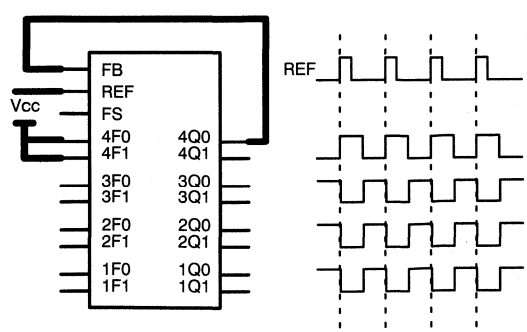


Figure 37. Inverted Output Clock Driver

outputs to operate in inverted mode. If one of these outputs is used as the FB input, then the PLL will align the rising edge of 4Q0 with REF and all of the other outputs operate 180° phase shifted from the FB input. This type of configuration is useful for system designs that require a greater number of clocks with 180° phase shift (inverted). These inverted clocks are useful for clocking logic at twice the frequency without distributing a higher frequency clock. In this configuration, RoboClock offers 6 inverted outputs and 2 non-inverted outputs. This configuration also has the advantage that all of the inverted outputs can be phase shifted and the 3Qn outputs can even be configured to divide the REF frequency by two or four while still maintaining phase inversion.

If only two inverted clocks are needed, then any output except the 4Qn outputs can be connected to the FB input. This will allow the 4Qn outputs to be selected for output inversion without affecting the other outputs. If the REF clock is not being used in other parts of the system, both of these two configuration options yield the same net effect.

Frequency Divider

RoboClock provides frequency division while still maintaining very low skew between the various output edges (more on this in the AC Specifications section). *Figure 38* shows RoboClock configured as a frequency divider. Since the PLL is operating at 20 MHz, the FS pin in this configuration is tied to ground indicating the selection of the 15– to 30-MHz operating range. By selecting 4Fn = LL, the 4Qn outputs will divide the FB frequency by two, and by selecting 3Fn = HH, the 3Qn outputs will divide the FB frequency by four.

Any of the 1Qn or 2Qn outputs could have been selected as the FB input with equivalent results. Note again that the divide-by-two and divide-by-four outputs have no coincidental rising edges. This features allows the system designer to use both outputs for multiphase clocking without concern for skew requirements between the rising edges of these two outputs.

Frequency Multiplier

RoboClock provides frequency multiplication by selecting an output configured to divide by two or divide by four as the FB input. *Figure 39* shows RoboClock multiplying up the REF input frequency. The 3Fn function select inputs are both tied to V_{CC}, configuring the 3Qn outputs to divide by four. But because 3Q0 is used as the FB input, the 3Qn outputs operate at the same frequency as the REF input. The RoboClock PLL, therefore, now operates at four times the REF frequency. This is the reason that the FS pin is tied to V_{CC} indicating the selection of the fastest operating frequency range. The selection of FS is based not on the operating frequency of the REF input, but rather on the operating frequency of the VCO. The operating frequency of the VCO is the same frequency as the 1Qn and 2Qn outputs at all times and also the 3Qn and 4Qn outputs when frequency division is not selected.

The 4Fn function select inputs are both tied to ground, which configures the 4Qn output for divide by two mode. Note that this figure looks much the same as *Figure 38* except for the selection of the FS

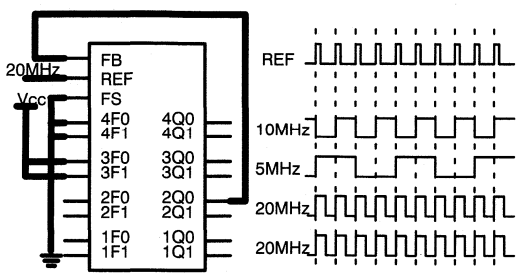


Figure 38. Frequency Divider

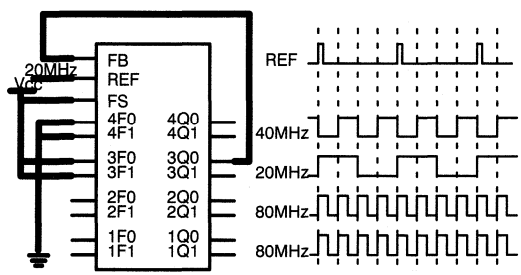


Figure 39. Frequency Multiplier

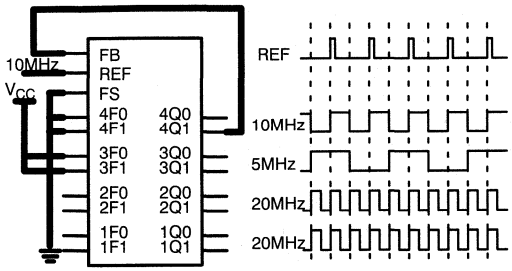


Figure 40. Frequency Divider and Multiplier

input and the selection of the output used as the FB input.

Frequency Divider and Multiplier

Figure 40 illustrates how RoboClock can be used to both multiply and divide the REF frequency. Here the VCO is running at 20MHz. The 4Qn output is used to divide the PLL frequency by two and the 3Qn outputs are used to divide the PLL frequency by four. 4Q1 is used as the FB input doubling the PLL rate.

Multi-Function Clock Driver

RoboClock is truly a multi-function clock driver. It has the ability to multiply up the REF frequency by two or four, divide down the REF frequency by two or four, perform phase inversion, create phase adjustments up to ±12 time units, while always providing very low output skew.

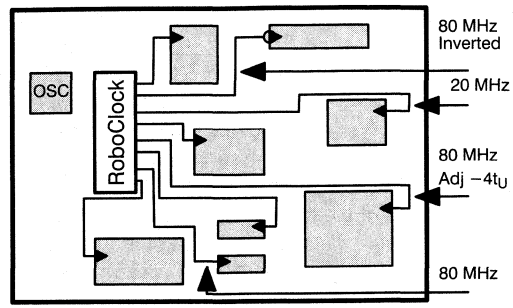


Figure 42. Multi-Function Clock Buffer Application

Figure 41 shows RoboClock configured as a multi-function clock driver. This figure shows how Roboclock can simultaneously multiply the REF frequency by four to a speed of 80 MHz and allow these high-speed outputs to be phase shifted and inverted. *Figure 42* shows how this configuration can be integrated to perform various system clocking functions.

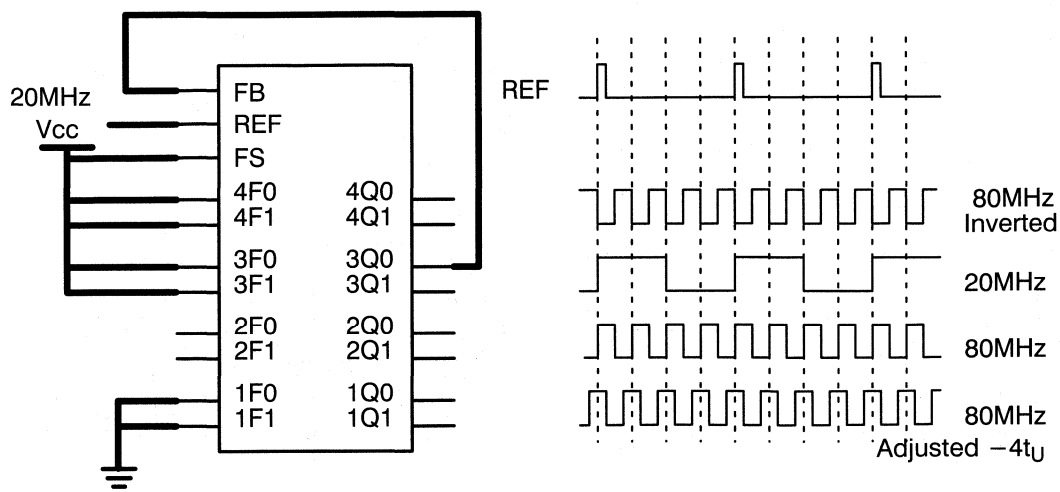


Figure 41. Multi-Function Clock Buffer

Many other examples exist of the tremendous capabilities of RoboClock. External circuitry, for example, can be placed between the RoboClock output and the FB input. This may be the case when RoboClock is used to drive external buffers, frequency dividers, or multipliers. Virtually any delay element can be placed between a RoboClock output and the FB input. A few things, however, must be remembered when doing this:

- The FB input must always be function of one of the RoboClock outputs.
- The other outputs will be referenced to the input of the external device and the delay may not be well defined due to the timing characteristics of the external devices.
- The output edge placement will be dependent on the function and skew characteristics of the external device.
- An external divider of no greater than 16 may be used in the FB path (jitter specifications may be compromised).

AC Specifications

Many AC Specifications exist for RoboClock. The following discussion will explain what these specifications mean to the designer. There are four different parts in the RoboClock family. The CY7B991 is a TTL-output (0 to 3V swing) device and the CY7B992 is a CMOS output (0 to V_{CC} swing) device. The CY7B99x–5 is the lowest skew device that Cypress offers. Its output skew has a typical value of just 250 ps with a maximum skew of 500 ps while the CY7B99x–7 has a maximum output skew of only 750 ps. The following sections will begin with the relevant datasheet specification. Only the CY7B99x–5 specifications will be explained, but all specifications apply directly to the CY7B99x–7 devices except for parameter value. Also, only the CY7B991–5 device will be explained unless the CY7B992–5 device has a different specification.

f_{NOM}: Operating Clock Frequency in MHz

Parameter	Description	CY7B99x			
		Min.	Typ.	Max.	Unit
f_{NOM}	FS = LOW	15		30	ns
	FS = MID	25		50	ns
	FS = HIGH	40		80*	ns

* The maximum operating frequency of the 7B992–7 devices is 50 MHz.

This parameter indicates the frequency range relative to given selections of the FS pin. The three-level FS input can either be tied to ground (LOW), left unconnected (MID), or tied to V_{CC} (HIGH).

The operating frequency these ranges refer to is based on the operating frequency of the VCO. If a divide-by-two output is used as the FB input, then the operating frequency of the VCO is twice that of the REF input. If a divide-by-four output is used as the FB input, then the operating frequency of the VCO is four times that of the REF input. If a non-divided output is used as the FB input then the selection of the FS pin can be made based upon the operating frequency of the REF input.

For example if a divide-by-four output is used as the FB input then the total possible REF input frequency range is 3.75 to 20 MHz (¼*15 to ¼*80 MHz). And the frequency range of the REF input when the FS pin is tied to ground, in particular, is 3.75 to 7.50 MHz.

t_{RPWH}, t_{RPWL} REF Pulse Width High and Low

Parameter	Min.	Typ.	Max.	Unit
t_{RPWH}	5.0			ns
t_{RPWL}	5.0			ns

The frequency and phase detector uses only the rising edge of the REF and FB inputs for alignment purposes. RoboClock, therefore, does not require a 50/50 duty cycle clock. The t_{RPWx} parameter is measured at the 1.5V level.

t_{PD} : Propagation Delay, REF Rise to FB Rise

Parameter	CY7B99x–5 Min.	Typ.	Max.	Unit
t_{PD}	– 0.5	0.0	+0.5	ns
Parameter	**CY7B99x–7 Min.**	**Typ.**	**Max.**	**Unit**
t_{PD}	– 0.7	0.0	+0.7	ns

Because of the PLL architecture of RoboClock there is no true propagation delay through the device as there is in a device such as a 74F244 buffer. The outputs of RoboClock are independent of the wave shape or duty cycle of the REF input. The PLL uses the REF and FB pins to generate the outputs by aligning these two inputs in both phase and frequency.

This static misalignment (t_{PD} in the datasheet) may be either positive or negative and is a function of the REF frequency. The ±500 ps specification is to accommodate the normal variation in process, voltage, temperature, and frequency. Two parts operating at the same frequency and similar voltage and temperature (approximately ±100 mV and ±10°C) will never have a t_{PD} variation more than 200 ps while the total magnitude of either might be 500 ps). This part-to-part variation appears as t_{SKEW5} in the datasheet and is discussed below. While the PLL aligns the REF and FB pins, some time difference may exist between the REF and FB pin (See *Figure 43*).

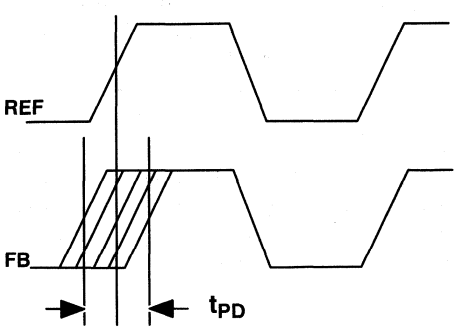


Figure 43. Propagation Delay

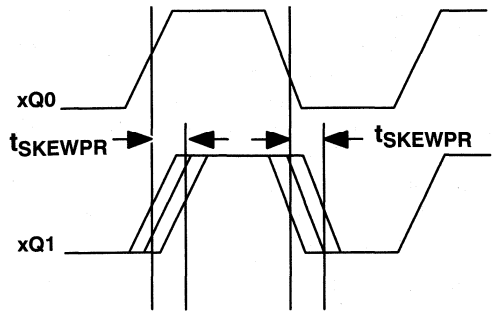


Figure 44. Zero-Output Matched Pair Skew

t_{SKEW} : Output Skew

There are six different parameters specifying output skew. Each of these parameters relates to different output configurations and to different output edges. Each of the following sections will describe the particular skew parameter with a diagram and include an example showing when this parameter would be used to calculate output skew.

t_{SKEWPR} : Zero Output Matched-Pair Skew

Parameter	Min.	Typ.	Max.	Unit
t_{SKEWPR}		0.1	0.25	ns

This parameter specifies the maximum amount of skew between two outputs of the same pair (e.g., 1Q1 and 1Q0) when all output are configured for 0 t_U. This specification has a maximum value of 250 ps. Bench characterization, however, indicates that t_{SKEWPR} is rarely greater than 100 ps. The additional margin is included for tester guard band. The reason that both outputs of the same pair can be so tightly coupled is that each pair has a dedicated power and ground pin, that they lie adjacent to each other allowing them to reinforce each other with crosstalk effects, and that they are separated from adjacent outputs by at least two non-switching pins.

Figure 44 shows that skew is measured from the first output to the last output and that t_{SKEWPR} as well as all other skew parameters pertains to both edges of the output waveform. *Figure 45* shows the 4Qn output pair loaded with the datasheet load. Although it may look like one output waveform, both outputs of the 4Qn pair are displayed. *Figure 46* shows the measurement of t_{SKEWPR} with an expanded voltage and time scale as being only 27 ps.

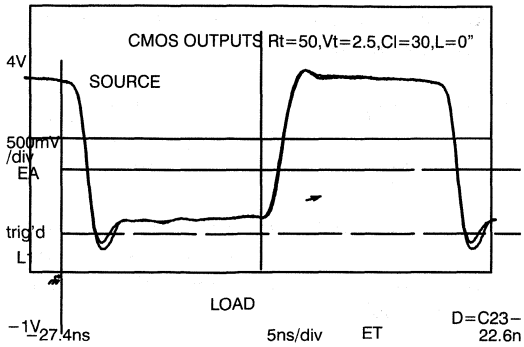


Figure 45. Lumped Load with Datasheet Load

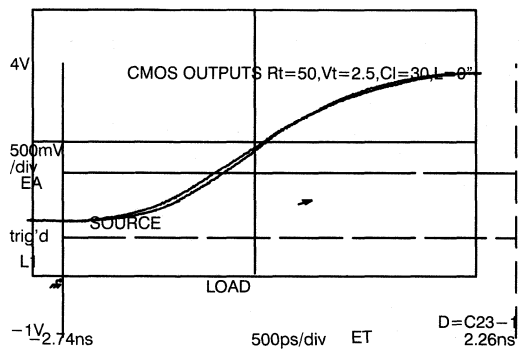


Figure 46. t_{SKEWPR} Measurement

t_{SKEW0} : Zero Output Skew

Parameter	CY7B99x−5 Min.	Typ.	Max.	Unit
t_{SKEW0}		0.25	0.5	ns
Parameter	**CY7B99x−7 Min.**	**Typ.**	**Max.**	**Unit**
t_{SKEW0}		0.3	0.7	ns

t_{SKEW0} is the maximum skew between the first output edge and the last output edge of all outputs when all outputs are configured for 0 t_U. This specification also applies to outputs that are not adjusted when another output is divided-by-two or divided-by-four. Bench data indicates that these skew values are usually no greater than 350 ps.

For example the skew between any of the outputs in *Figure 33* is no greater than 500 ps. The maximum skew between the 1Qn and 2Qn outputs in *Figure 38* is also 500 ps, even though both the 3Qn and 4Qn outputs are divided. This assumes that a CY7B99x−5 is used to generate these signals.

t_{SKEW1} : Output Skew (Rise-Rise, Fall-Fall, Same Class Outputs)

Parameter	CY7B99x−5 Min.	Typ.	Max.	Unit
t_{SKEW1}		0.25	0.5	ns
Parameter	**CY7B99x−7 Min.**	**Typ.**	**Max.**	**Unit**
t_{SKEW1}		0.3	0.7	ns

t_{SKEW1} is specified as the maximum amount of skew between outputs of the same output class selected for the same output adjustment without restrictions on the placement or function of other outputs. For the purposes of skew specification, there are three types of output classes:

- *Nominal*: Outputs that are selected for phase adjustment, but not inversion, division, or multiplication are of the Nominal output class. This also includes outputs that are not phase adjusted when other outputs have a non 0t_U configuration.
- *Divided*: This class of outputs includes the 3Qn and 4Qn outputs that are configured for divide-by-two or -four mode (3Fn=HH, or LL, and 4Fn = LL). Even when these outputs are configured this way and selected as the FB input, the are still considered to be part of the Divided output class.
- *Inverted*: This class of outputs includes the 4Qn outputs configured in the inverted mode of operation (4Fn = HH). The Inverted output class also applies to a 4Qn output configured for phase inversion and used as the FB input.

This parameter, as in the case of $t_{SKEWPR, 0, and 3}$ applies not only to rising edge to rising edge output skew, but also to falling edge to falling edge skew.

Figure 47 illustrates a an example of when to use t_{SKEW1} to calculate output skew. The maximum skew between the 1Qn and 2Qn outputs, when they are selected in this case for −4 time unit adjustment, is no greater than 700 ps. The maximum skew be-

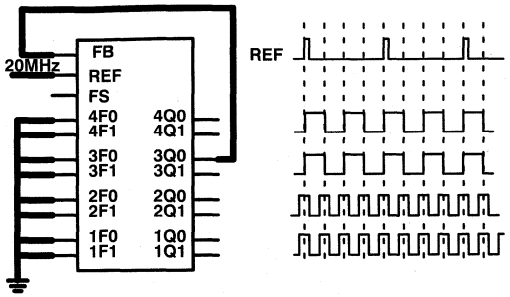


Figure 47. Multi-Function Clock Buffer

tween the 4Qn and 3Qn when both are selected for divide−by−two is also 700 ps.

Figure 48 shows a diagram of an output (Adjusted Q) that has been programmed to occur N time units (t_U) later than another output (Q). The maximum amount of time between these two outputs will be

$$t_{DIFF} = N \times t_U + t_{SKEW1} \quad \text{Eq. 27}$$

and the minimum amount of difference between these two outputs is

$$t_{DIFF} = N \times t_U - t_{SKEW1} \quad \text{Eq. 28}$$

Where N is difference in the calculated tap delays between these two outputs. There is no need to add twice the t_{SKEW1} time in order to calculate the minimum and maximum time difference.

For example, the minimum and maximum time between the 1Qn and 2Qn outputs in a system configured as in *Figure 35* using a CY7B991−5 operating at 50 MHz would be

$$t_{DIFF(min)} = 2 * \left(\frac{1}{50 * 10^6 \times 26}\right) - .5 = 1.04\ ns \quad \text{Eq. 29}$$

$$t_{DIFF(max)} = 2 * \left(\frac{1}{50 * 10^6 \times 26}\right) + .5 = 2.04\ ns \quad \text{Eq. 30}$$

The time difference between the 3Qn and 4Qn outputs (which were not phase adjusted) and the 1Qn and 2Qn (which were phase adjusted in opposite directions) would be

$$t_{DIFF(min)} = \left(\frac{1}{50 * 10^6 \times 26}\right) - .5 = .269\ ns \quad \text{Eq. 31}$$

$$t_{DIFF(max)} = \left(\frac{1}{50 * 10^6 \times 26}\right) + .5 = 1.269\ ns \quad \text{Eq. 32}$$

These equations give the worst-case time value between outputs skewed by 1 time unit when operating at 50 MHz. No additional output skew parameters need to be added. The ordering, including skew, between two outputs that are phase adjusted, can always be determined from the functional input selections. This is shown in the above example where the minimum time between two output adjusted by one time unit was determined to be 269 ps.

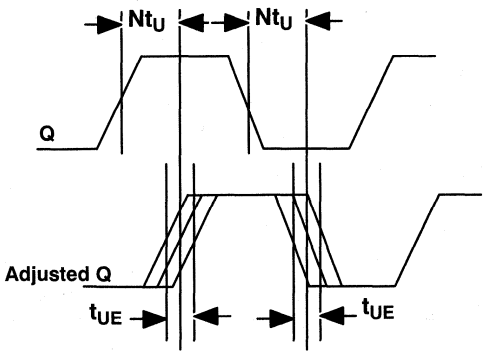


Figure 48. Programmable Adjustment Error

t_{SKEW2}: Output Skew (Rise-Fall, Nominal-Inverted, Divided-Divided)

Parameter	CY7B99x−5 Min.	Typ.	Max.	Unit
t_{SKEW2}		0.6	1.2	ns

Parameter	CY7B99x−7 Min.	Typ.	Max.	Unit
t_{SKEW2}		1.0	1.5	ns

This skew parameter specifies the amount of output skew between the rising or falling edge of a Nominal output and the opposite edge of an Inverted output as generalized in *Figure 49*. This parameter also applies to opposite edge transitions between Divided outputs.

For example, in *Figure 39,* the output skew between the opposite edge transitions of the the 3Qn and 4Qn outputs selected for divided mode is no greater than 1.2 ns. The magnitude of this number compensates for the different in the rising and falling edge rates.

t_{SKEW3} : Output Skew (Rise-Rise, Fall-Fall, Different Class Outputs)

Parameter	CY7B99x-5 Min.	Typ.	Max.	Unit
t_{SKEW3}		0.6	1.0	ns
Parameter	**CY7B99x−7 Min.**	**Typ.**	**Max.**	**Unit**
t_{SKEW3}		0.7	1.2	ns

This output skew parameter specifies the maximum same edge transition difference between different class outputs. In *Figure 39*, the difference between the rising edge of the 4Qn or 3Qn outputs that are configured for divided mode and the rising edge of the 1Qn and 2Qn outputs will be no greater than 1 ns.

t_{SKEW4} : Output Skew (Rise-Fall, Nominal-Divided, Divided-Inverted)

Parameter	CY7B99x−5 Min.	Typ.	Max.	Unit
t_{SKEW4}		0.6	1.3	ns
Parameter	**CY7B99x−7 Min.**	**Typ.**	**Max.**	**Unit**
t_{SKEW4}		1.2	1.7	ns

This output skew parameter specifies the maximum opposite edge transition difference between different class outputs as generalized in *Figure 49*. In *Figure 39,* the maximum difference between the opposite edge transition of the 4Qn or 3Qn outputs and the 1Qn and 2Qn outputs would be no greater than 1.3 ns

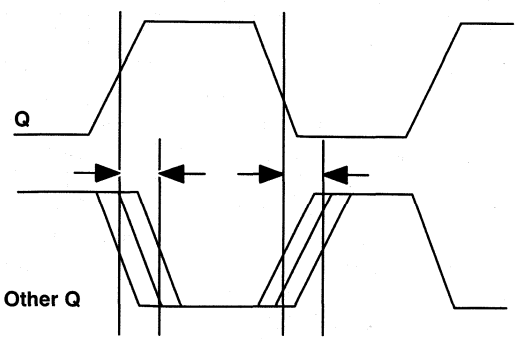


Figure 49. t_{SKEW2} and t_{SKEW4} Measurement

t_{SKEW5} : Device-to-Device Output Skew

Parameter	CY7B99x Min.	Typ.	Max.	Unit
t_{SKEW5}			0.2	ns

Taken by itself, this parameter has very little meaning in a system design. It must be used in conjunction with the other output skew parameters discussed above. This parameter states that the maximum variation in t_{PD} between two devices operating at the same frequency, temperature, and voltage will be no greater than 200 ps. This means that the device to device skew between two outputs used as the FB input will be no greater than 200 ps under these circumstances.

Figure 50 (an adapted version of *Figure 39*) shows two RoboClock devices configured as in *Figure 39* connected in parallel. To calculate the device to device skew for outputs configured for 0 t_U use the t_{SKEW0} parameter of each device plus t_{SKEW5} as in the following equation:

$$t_{SKEW} = t_{SKEW0(1)} + t_{SKEW0(2)} + t_{SKEW5}$$
$$t_{SKEW} = 0.5 + 0.5 + 0.2 = 1.2\ ns \qquad \text{Eq. 33}$$

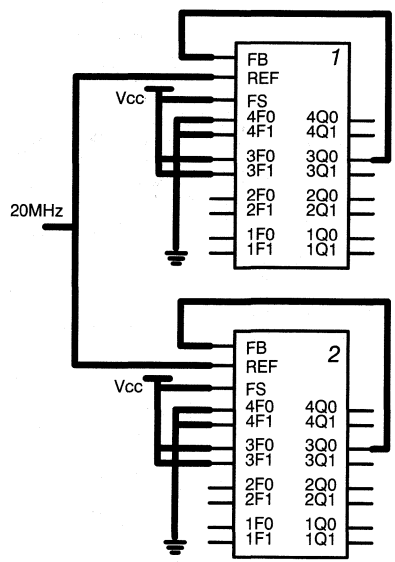


Figure 50. Part-to-Part Skew Example

This will give the output skew between the 2Qn and 1Qn output skew between devices 1 and 2. To calculate the output skew between the rising edges of the divided outputs and the 0-phase-adjusted outputs, use the following equation:

$$t_{SKEW} = t_{SKEW3(1)} + t_{SKEW3(2)} + t_{SKEW5}$$
$$t_{SKEW} = 1.0 + 1.0 + 0.2 = 2.2ns \quad \text{Eq. 34}$$

In general, to calculate the skews between two devices, add the relevant skew component from each device and add 200 ps.

Figure 51 gives a slightly more complicated example to illustrate how to calculate device-to-device skew. Device 3 has been added to the previous example. The outputs of this device are driving the REF inputs of devices 1 and 2. The only additional component to add to the output to output skew for devices 1 and 2 is the output-to-output skew of device 3. The part-to-part skew between devices 1 and 2 for outputs that are left unconfigured (1Qn and 2Qn) is now

$$t_{SKEW} = t_{SKEW0(3)} + t_{SKEW1(1)} + t_{SKEW1(2)} + t_{SKEW5}$$
$$t_{SKEW} = 0.5 + 0.5 + 0.5 + 0.2 = 1.7ns \quad \text{Eq. 35}$$

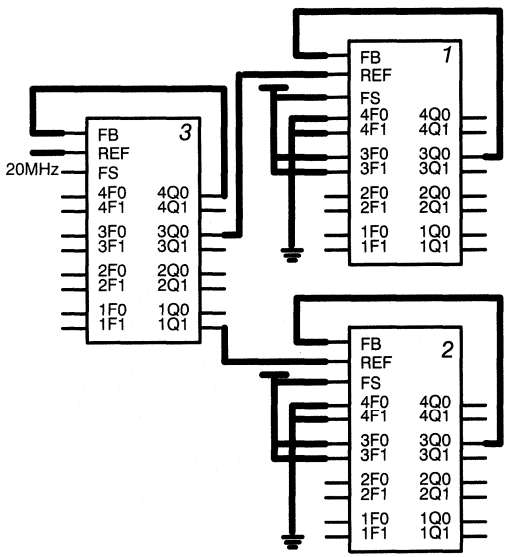


Figure 51. Devices in Parallel and Serial

t_{ODCV} : Output Duty Cycle Variation

Parameter	CY7B99x–5 Min.	Typ.	Max.	Unit
t_{ODCV}	– 1.0	0.0	+1.0	ns

Parameter	CY7B99x–5 Min.	Typ.	Max.	Unit
t_{ODCV}	– 1.2	0.0	+1.2	ns

This parameter specifies the difference in the output duty cycle from 50%. It is measured at 1.5V. This parameter indicates, for example, that the outputs of the CY7B99x–5 have a worst-case duty cycle of 42/58, and at 15 MHz the duty cycle is 48.5/51.5 worst case. The AC Characterization section contains data on how the value of this parameter varies with loading, voltage, and temperature.

t_{ORISE}, t_{OFALL} : Output Rise and Fall Time

Parameter	CY7B991–5 Min.	Typ.	Max.	Unit
t_{ORISE}	0.15	1.0	1.5	ns
t_{OFALL}	0.15	1.0	1.5	ns

Parameter	CY7B992–5 Min.	Typ.	Max.	Unit
t_{ORISE}	0.5	2.0	2.5	ns
t_{OFALL}	0.5	2.0	2.5	ns

Parameter	CY7B991–7 Min.	Typ.	Max.	Unit
t_{ORISE}	0.15	1.5	2.5	ns
t_{OFALL}	0.15	1.5	2.5	ns

Parameter	CY7B992–7 Min.	Typ.	Max.	Unit
t_{ORISE}	0.5	3.0	5.0	ns
t_{OFALL}	0.5	3.0	5.0	ns

The output rise and fall time are measured with different loads for each of the four devices. The CY7B991 devices (TTL) are tested with the load shown in *Figure 52*. The 130Ω over 91Ω load is the recommended parallel termination for 50Ω transmission lines. For the –7 devices a 50-pF load is used to test all AC parameters for the –5 devices a 30-pF load is used. *Figure 53* shows the CMOS device test load (CY7B992).

The TTL and CMOS devices are also measured between different voltage levels. *Figure 54* shows that

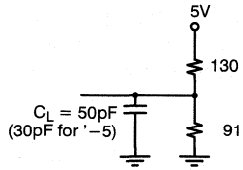


Figure 52. TTL AC Test Load

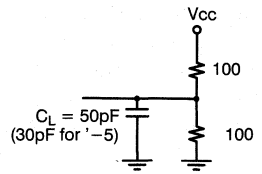


Figure 53. CMOS AC Test Load

the TTL rise and fall time parameters are measured between 0.8 and 2.0V, and *Figure 55* shows that the CMOS rise and fall times are measured between 20% and 80% of V_{CC}.

The outputs edge rates of these devices are controlled to about 1 V/ns to minimize the generation of system noise and transmission line effects. The AC Characterization section will give examples of how this parameter varies with loading, voltage, and temperature.

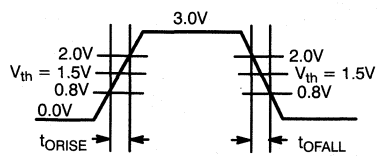


Figure 54. TTL Output Voltage Levels

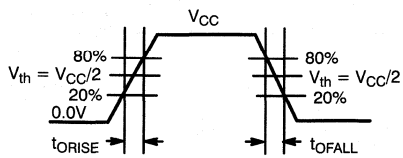


Figure 55. CMOS Output Voltage Levels

t_{PWH}, t_{PWL} : Output High and Low Time Deviation from 50%

Parameter	CY7B991–5 Min.	Typ.	Max.	Unit
t_{PWH}			2.5	ns
t_{PWL}			3	ns
Parameter	**CY7B992–5 Min.**	**Typ.**	**Max.**	**Unit**
t_{PWH}			3.5	ns
t_{PWL}			3.5	ns
Parameter	**CY7B991–7 Min.**	**Typ.**	**Max.**	**Unit**
t_{PWH}			3	ns
t_{PWL}			3.5	ns
Parameter	**CY7B992–7 Min.**	**Typ.**	**Max.**	**Unit**
t_{PWH}			5.5	ns
t_{PWL}			5.5	ns

The output pulse width high and low times are specified as deviations from an ideal 50/50 duty cycle. t_{PWH} is measured above the 2.0V (80% V_{CC}) for the TTL (CMOS) devices and t_{PWL} is measured below 0.8V (20% V_{CC}) for the TTL (CMOS) devices. The value of these parameters can be calculated from the combination of the t_{ODCV} parameter and the t_{ORISE} and t_{OFALL} parameters. The specifications can be calculated as follows

$$t_{PWH} = 2 * t_{OR/OF} * \left(\frac{V_{MAX} - V_{TH}}{V_{MAX} - V_{MIN}}\right) + t_{ODCV} \quad \text{Eq. 36}$$

$$t_{PWL} = 2 * t_{OR/OF} * \left(\frac{V_{TH} - V_{MIN}}{V_{MAX} - V_{MIN}}\right) + t_{ODCV} \quad \text{Eq. 37}$$

Where t_{OROF} represents either the rise time or fall time of the output since they are equal, V_{TH} represents the measurement threshold (TTL = 1.5V and CMOS = $V_{CC}/2$), V_{MAX} represents the maximum voltage point of rise and fall time measurements and V_{MIN} represents the minimum voltage point for rise and fall time measurements.

AC Characterization

Included with this application note are output rise time, output fall time, and output duty cycle varia-

tion versus temperature, voltage, capacitive loading, and termination voltage.

Output Rise Time

As explained previously, output rise time (t_{ORISE}) is the maximum amount of time it takes the output to rise from the 0.8V to the 2.0V level.

Figure 56 shows the variation in output rise time based on capacitive loading. This graph can be used to calculate the skew caused by the unequal loading of outputs.

Figure 57 shows the output rise time versus voltage over temperature. Within the normal operating limits of RoboClock (4.5 to 5.5V), the output rise time varies very little with respect to temperature. This means that, within a normal board environment, output rise time will not significantly vary due to minor variations in temperature or voltage.

Figure 58 shows the output rise time vs. termination voltage. In the section entitled Transmission Line Termination the parallel termination Thevenin voltage was specified as 2.06V for the CY7B991–x devices. This figure shows that because output rise time does not vary with termination voltage, convenient resistor value can be chosen that maintain $R_{TH} = Z_0$ within the limits of $1.5V < V_{TH} < 2.5V$.

Output Fall Time

Output fall time is the amount of time it takes for the output to swing from 2.0V to 0.8V.

Figure 59 shows the variation in output fall time with load capacitance. *Figure 60* shows the output fall time vs. chip voltage at various temperatures. Notice that there is almost no variation in output fall time due to changes in device temperature. *Figure 61* shows the output fall time vs. termination voltage.

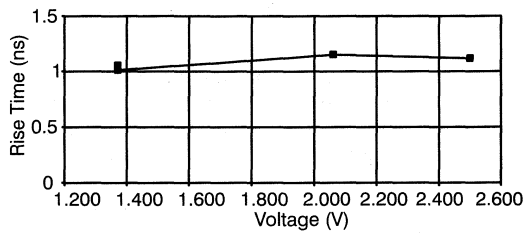


Figure 58. Rise Time vs. Termination Voltage

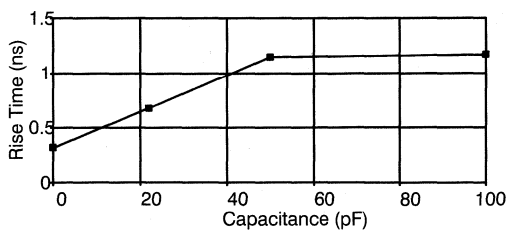


Figure 56. Rise Time vs. Capacitance

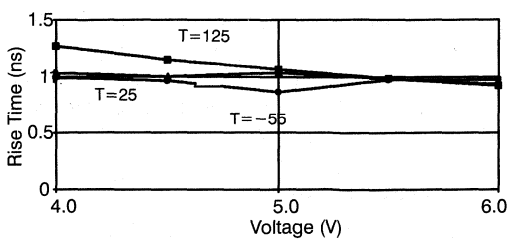


Figure 57. Rise Time vs. Voltage over Temperature

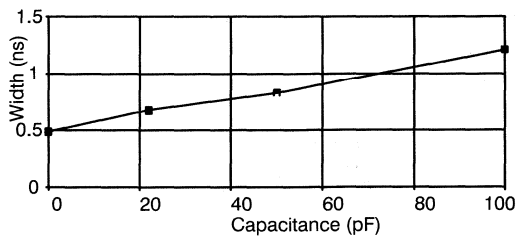


Figure 59. Fall Time vs. Capacitance

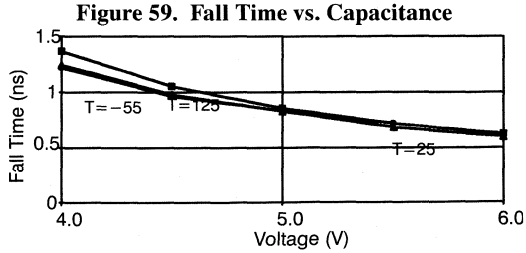


Figure 60. Fall Time vs. Voltage over Temperature

Output Duty Cycle Variation

Output duty cycle variation is the difference in the output pulse width from the ideal 50%. This parameter is measured at the 1.5V level. Characterization of this parameter was performed by measuring the output pulse width high. Measurements were taken at two different REF input cycle times (Tref): 50 ns and 12 ns.

Figure 62 and *Figure 63* show the variation in output duty cycle due to variations in output capacitance. These graphs indicate that if the outputs are loaded according to the datasheet specification, the output duty cycle will be very near 50%.

Figure 64 and *Figure 65* show the output pulse width high vs. device voltage over temperature. These graphs indicate that the propagation delay difference between the rising and falling edge of the outputs varies according to frequency. For operation within the ±10% V_{CC} and the commercial temperature range, the output duty cycle specification is ±500 ps.

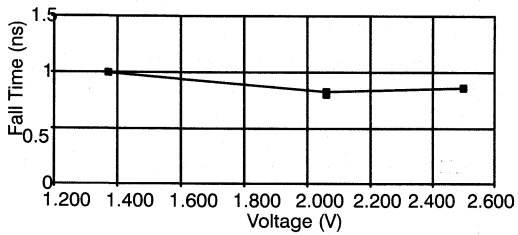


Figure 61. Fall Time vs. Termination Voltage

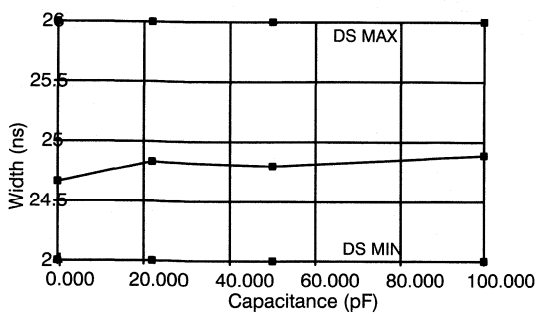


Figure 62. Pulse Width High vs. Capacitance t_{REF} = 50ns

Figure 66 and *Figure 67* show the output pulse width high vs. termination voltage. Both of these graphs indicate that with normal variance in termination

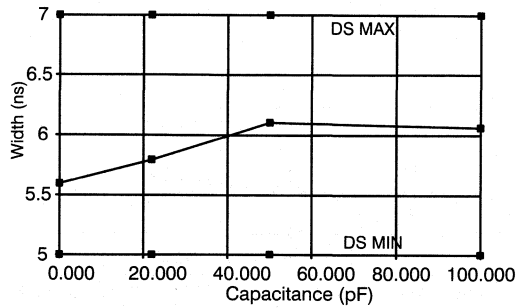


Figure 63. Pulse Width High vs. Capacitance t_{REF} = 12ns

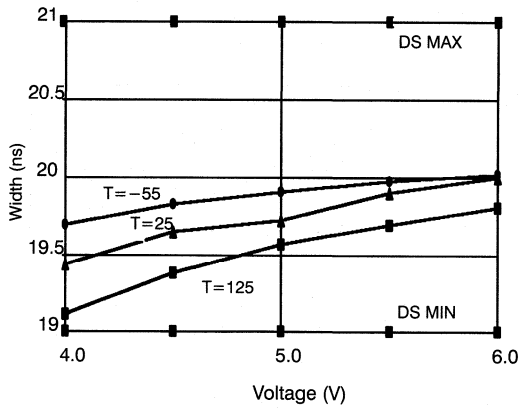


Figure 64. Pulse Width High vs. Voltage over Temperature

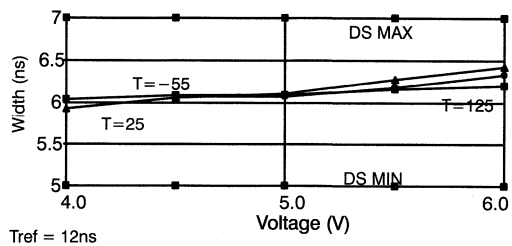


Figure 65. Pulse Width High vs. Voltage over Temperature

voltage the output duty cycle specification remains within the ±500 ps range.

Conclusion

RoboClock provides system designers with a multifunctional resource that solves most clock distribution problems. The third-generation PLL architecture based on a distributed phase clock oscillator allows phase shifting, division, multiplication and inversion of outputs with over 26,000 possible output configurations. These features combine to offer compensation for trace-length differences, elimination of set-up and hold time mismatches, multiplication of lower-frequency system clocks, division of system clocks for lower-performance system components, and phase inversion for multiphase clocking. All of these benefits are combined with extremely low skew outputs, low device propagation delay, and high-frequency operation to provide the most full-featured device available.

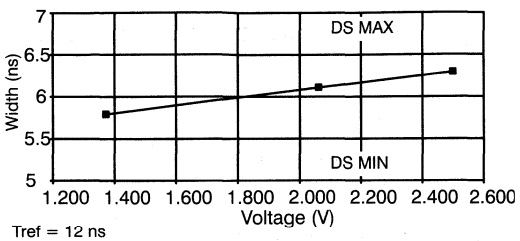


Figure 66. Pulse Width High vs. Termination Voltage

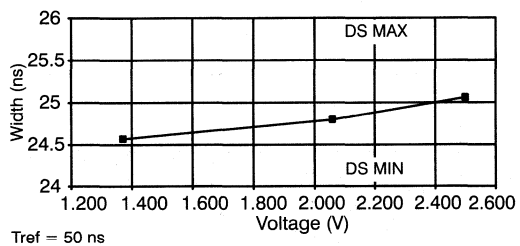


Figure 67. Pulse Width High vs. Termination Voltage

References

1. American Technical Ceramics, *The RF Capacitor Handbook,* Revision B, December 1991.
2. Blood Jr., William R., *MECL System Design Handbook*, Fourth Edition, 1988.
3. BTL Working Group, "A Guide to Backplane Electrical Performance Measurements," P1194.0/D2, July 1989.
4. Motorola, Inc., *Application Note AN1051* "Transmission Line Effects in PCB Applications," 1990.
5. Wescon Technical Conference, "Solving Clock-Distribution Problems in High-Speed Systems," Session 5, Nov. 1991.

Document #: LOG0008–0293

Innovative RoboClock Application

Introduction

This application note presents a unique application of RoboClock, using its complex and precise waveform generation capability to implement PWM (Pulse Width Modulation) to enhance color images and increase the resolution of laser printers. The first section of this application note provides a brief description of RoboClock and presents three methods that the user can employ to configure it. Second, a brief background on image and resolution enhancement is presented. Finally, the required waveform to implement the image enhancement and the configuration of RoboClock is presented.

Overview of RoboClock

The CY7B991 and CY7B992, commonly known as RoboClock, are programmable skew clock buffers capable of generating thousands of various clocking combinations. As shown in *Figure 1*, the eight high drive outputs are arranged in four pairs, which can be configured by three-level inputs (HIGH, LOW, and MID logic level). The internal PLL is fully self-contained and does not require any external components to operate. The PLL buffer stages are differential, greatly enhancing the robustness of the PLL operation in terms of jitter over voltage, and temperature variations.

Basically, the PLL aligns the output clock in both phase and frequency with the reference clock. The simplest mode of operation is the low-skew output mode. In this mode the outputs are virtually skewless. The maximum skew is only a few hundred picoseconds. Please refer to the CY7B991/992 datasheet for various skew specifications. The second mode is the programmable skew mode. The outputs of RoboClock can be skewed (advanced and delayed) by increments of one time unit (t_U), which is 0.7 to 1.5 ns, determined by the operating frequency and range of the PLL.

$$t_U = 1/(F_{nom} * N) \qquad \text{Eq. 1}$$

As shown in *Table 1*, the frequency range of the PLL is determined by the three-level FS input. For each frequency range, there is a corresponding integer "N" that can be used in *Equation 1* to calculate t_U. Up to $\pm 12 t_U$ skew can be achieved between the outputs of RoboClock (positive t_U represents delaying

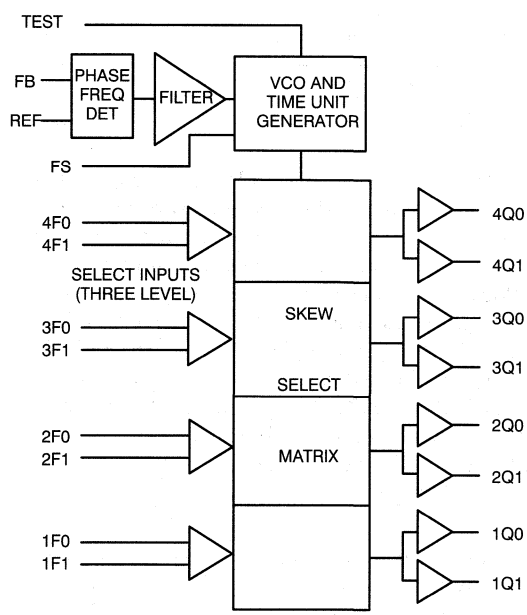


Figure 1. Logic Block Diagram

the output with respect to REF and negative t_U represents advancing the output with respect to REF).

The third mode of operation is the Multi-function mode. In this mode the outputs may be multiplied by 2 or 4, divided by 2 or 4, or inverted. Most importantly, the skew features can be combined with multiply, divide, and invert functions. This results in, literally, over 26,000 timing configurations. For more detailed information on the operation of the RoboClock, please refer to the following application notes "Using the CY7B991 with the 50-MHz 486 Cache Module and the 40-MHz R3000" and "Everything You Need to Know About CY7B991/CY7B992 (RoboClock) But Were Afraid to Ask." This application note is meant to complement the topics discussed in above mentioned application notes.

Table 1. Frequency Range Select and t_U Calculation

FS	f_{NOM} (MHz) Min.	f_{NOM} (MHz) Max.	$t_U = \frac{1}{f_{NOM} \times N}$ where N =	Approximate Frequency (MHz) At Which t_U = 1.0 ns
LOW	15	30	44	22.7
MID	25	50	26	38.5
HIGH	40	80	16	62.5

Usually, one of the outputs of RoboClock is used as the Feedback input. If the desired waveform is not directly generated by RoboClock, an imaginative user may run an output of RoboClock through a logic block, then send it back to the FB input of the RoboClock. Through this scheme, unlimited additional functions may be implemented by RoboClock. Note that in this case, all the other outputs of the RoboClock will be shifted by a period equal to the delay through the external logic block, because the PLL will align the FB input with the REF input, both in phase and frequency.

Cascading two RoboClocks in series will also dramatically increase the output possibilities. In this case, one of the outputs of the first stage will serve as the REF input for the second stage. Multiple feedback configurations are possible, which can result in an innovative set of outputs.

RoboClock Configuration Methodologies

Using One Small Table

The entire set of programmable skew configurations is summarized in a single small table shown in *Table 2*. Every possible combination can be driven from this small table. For example, if $+2t_U$ is required from 3Qx (3Q0 or 3Q1) outputs, based on *Table 2*, the corresponding 3Fx inputs should be set as 3F1= MID and 3F0=HIGH. Any one of 1Qx, 2Qx, or 4Qx outputs may be used as FB input, by leaving its corresponding 1Fx, 2Fx, or 4Fx inputs floating (i.e., 1F1= MID , 1F0= MID). Note that *Table 2* represents only the cases where the feedback is an output with no skew, divide, or invert function. Basically, a $0t_U$ output is used for FB input.

Table 2. Output Adjustment Configurations

Function Selects		Output Functions		
1F1, 2F1, 3F1, 4F1	1F0, 2F0, 3F0, 4F0	1Q0, 1Q1, 2Q0, 2Q1	3Q0, 3Q1	4Q0, 4Q1
LOW	LOW	$-4t_U$	Divide by 2	Divide by 2
LOW	MID	$-3t_U$	$-6t_U$	$-6t_U$
LOW	HIGH	$-2t_U$	$-4t_U$	$-4t_U$
MID	LOW	$-1t_U$	$-2t_U$	$-2t_U$
MID	MID	$0t_U$	$0t_U$	$0t_U$
MID	HIGH	$+1t_U$	$+2t_U$	$+2t_U$
HIGH	LOW	$+2t_U$	$+4t_U$	$+4t_U$
HIGH	MID	$+3t_U$	$+6t_U$	$+6t_U$
HIGH	HIGH	$+4t_U$	Divide by 4	Inverted

Now, to generate additional output functions, if the feedback output is programmed to skew, divide or invert, then output functions of other outputs may not be directly read from *Table 2*. In this case, to figure out the final output function observed on the output, simply subtract whatever the feedback term is programmed to, from the output function programmed on the corresponding output. Therefore, by using only *Table 2* and the following simple algorithm, every single combination of RoboClock can be figured out.

Final Output Function = Output Function − FB Function

If there is any ambiguity, the following example should clarify the use of this method. Let's say $+7t_U$

of delay is required. Obviously, $+7t_U$ is not a choice, available in *Table 2*. However, any two functions from two different outputs of *Table 2* may be combined to achieve a desired function. For this example, there are several solutions, and only one of them will be presented. One way to achieve $+7t_U$ is to subtract $-3t_U$ from $+4t_U$.

$$+7t_U = +4t_U - (-3t_U)$$

Therefore, if 1Qx output is programmed to have $-3t_U$ of skew (1F1=LOW, 1F0=MID), and used as the FB input, and if the 3Qx is programmed to have $+4t_U$ of skew (3F1=HIGH, 3F0=LOW), the final output function observed on 3Qx will be $+7t_U$.

One exception to this simple rule is that if a divided output is used as the FB input, then the other outputs will be multiplied by the same factor (2 or 4). The reason for this is that the PLL will force the FB to align with the REF both in phase and frequency. Therefore, if the FB term is programmed to divide by 2, the PLL will speed up twice to force the FB term to align with the REF frequency. As an example, if advance by $6t_U$ and multiply by 4 function is required ($-6t_U$ and f*4), then

$$\textit{Final Function} = -6t_U - \textit{(divide by 4)} => (-6t_U \textit{ and } f*4)$$

The solution for this example is to program 3Qx to divide by 4 (3F1=HIGH, 3F0=HIGH) and use it as FB, and program 4Qx to have $-6t_U$ of skew (4F1=LOW, 4F0=MID). The final function observed on 4Qx will be REF frequency multiplied by 4 and advanced by $6t_U$ ($-6t_U$ and f*4).

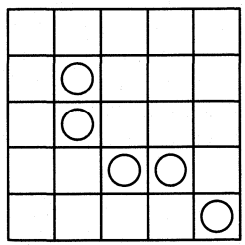
By this method, one can easily determine if a desired function can be implemented by RoboClock or not. RoboClock can generate a waveform composed of any two functions from two different outputs of *Table 2*.

Using Three Tables for Multiple Outputs

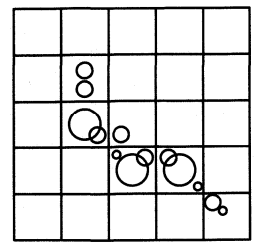
If multiple outputs with various functions are required, using the previous method could be a little cumbersome. All the possible combinations of RoboClock outputs are in three tables, illustrated in *Tables 3* through *5*. Each table represents all the possible output combinations with a given output connected to FB input. For example, once 3Qx output is used as FB input, then all the possible output combinations could be found in *Table 4*. These three tables are extremely valuable tools in determining what FB term to use and how to configure the RoboClock, when multiple outputs with various functions are required.

Once the required multiple functions are determined in terms of t_U, an effort should be made to locate one row in one of the three tables that contains the required functions. For example, if one of the desired functions is divide by 2 and delay $4t_U$ ($+4t_U$ and f/2), then by observation, that choice can be located in row 1 of *Table 3*, row 3 of *Table 4*, and row 3 of *Table 5*. Now, the one to be selected as a solution would depend on what the other required functions are, because once an output, which is programmed to perform a certain function, is selected as FB input, all the outputs of a RoboClock are limited to a single row found in *Tables 3* through *5*. If in the previous example, the second required function happens to be invert and skew by $4t_U$ ($+4t_U$ and INV), then the only solution is row 1 of *Table 3*. In this case 1Qx could be used as the FB input with its inputs hardwired to GND (1F1=LOW, 1F0=LOW), and 3F1=HIGH, 3F0=HIGH to generate ($+4t_U$ and f/2) function on 3Qx outputs, and set 4F1=HIGH, 4F0=HIGH to generate (+4tu and INV) function on 4Qx outputs. In this configuration, the 2Qx outputs could be programmed to have any one of $0t_U$ through $+8t_U$ skew. For example, if $+7t_U$ is another required output, then 2F1=High, 2F0=Mid will generate $+7t_U$ skew on 2Qx. Note that even though the 1Qx outputs are programmed to have $-4t_U$ skew, they are forced by PLL to align with the REF frequency, therefore 1Qx output could be used as a $0t_U$ output.

The Table method is recommended for multiple outputs with various function requirements. If the exact required outputs cannot all be found in one row, then the designer can use the three tables to understand the design choices that are available within the three tables. Based on the design requirements, the user can make a judgement on what outputs must exactly meet the required specs, and what outputs may be slightly compromised. If the required outputs are not found in one row of the three tables,



a. Standard unmodulated



b. Modulated laser beam with shorter "on" periods and variable-size dots.

Figure 2. Laser Images

and no compromise can be made on the requirements, then two or more RoboClocks may be used to meet the specific required outputs.

Using RoboClock in Resolution Enhancement of a Laser Printer

Background

Laser printers are no different from any other electrical systems, in that the higher the resolution or the accuracy of the system, the higher the complexity and cost of the system. It has been and will always be the goal of system and design engineers to achieve the highest performance and resolution at the lowest cost.

In the case of a laser printer, to achieve higher resolution than the nominal low-end 300 DPI (dot per inch), the throughput of the processor, size of the memory, and the glue logic should increase accordingly. In many cases, the additional hardware cost does not justify the enhancement in the resolution. A few years ago, a new technique called Resolution Enhancement Technology (RET) was developed by Hewlett Packard. The main advantage of this technique versus conventional resolution enhancement techniques is that the resolution enhancement is gained with hardly any increase in the throughput of the processor or memory size. Therefore, this approach is a very economical way of gaining enhanced resolution.

For obvious reasons, the entire laser printer industry is using some form of this technique. Various flavors of the same technique are being applied to different image-enhancement machines. The halftones or gray scales are common in most laser printers. The underlying technique is fairly simple. This is accomplished by modulating the laser beam, as opposed to the conventional "on" or "off" state of the laser beam, for the entire cycle time. The laser beam could be turned "on" or "off" 25%, 50%, 75% or 100% of the period. By this, large and small dots can be produced on a given image, therefore gaining much higher "perceived" resolution compared with images constructed by only one size dot. The varying size dots produce much smoother text files and generate much sharper images through shades of gray. Refer to *Figure 2a* and *2b*. The same idea is used in color image enhancement where much smoother and more pleasant images may be produced in a given color image by dilating black dots and shrinking the red. This filtering or color enhancement feature can be used to produce various special effects, or simply be employed to create more appealing color images. Please refer to *Figure 3a* and *3b*. To further clarify this technique, the true resolution, technically, still remains the same, but the images are "perceived" to be higher resolution. It does not matter how the "better image" was



a. Black dot dilated **b. Red dot shrunk**

Figure 3. Color Enhancements

created, as long as it looks good and the cost of the hardware is affordable.

Design Implementation

RoboClock is used to generate precise complex waveforms needed for laser beam modulation. The particular laser system discussed in this application note requires eight levels of modulation, which consists of 100% on, 75% on, 50% on, 25% on, 100% off, 75% off, 50% off, and 25% off. The eight waveforms are shown in *Figure 4*.

Note that all waveforms are synchronized to the rising edge of the system clock.

Analyzing the entire circuitry of the laser printer is beyond the scope of this application note, and only the waveform modulation section is discussed. The system clock runs at 66.67 MHz, which translates into 15 ns cycle time. The simplified diagram of the modulation section is shown in *Figure 5*. The modulation section consists of a RoboClock, a 256K*4 SRAM that contains the pixel information, four NOR gates with complemented outputs, and an 8:1 MUX. In this application, since the laser head interface uses ECL levels, 500 ps ECL NOR gate with complemented outputs (MC10E101) and ECL 8:1 MUX (MC10E163) is used. Unused inputs of the quad four input NOR gates are tied LOW. The TTL outputs of CY7B991 are translated into ECL levels by Cypress Semiconductor high-speed, low-skew TTL to ECL translator (CY10E384L). To keep the modulation logic diagram simple, the translate block is not shown in *Figure 5*. Note that, generally, 74AS logic parts are used to implement external logic functions, which requires no translate logic to interface with RoboClock.

The 66.67-MHz system clock is fed to the REF input of the RoboClock. RoboClock generates very precise waveforms and, with one level of gating, all the six modulated waveforms are produced and fed to the 8:1 MUX. For this design, RoboClock generates precise 90-degree phase-shifted, true and complemented versions of the 66.67-MHz REF input frequency. Note that only six waveforms are generated. The "100% on" and "100% off" modes are hardwired HIGH and LOW to the 8:1 MUX. Three bits of the SRAM are used to select one out of eight possible modulated signals. The output of very fast MUX is directly sent to the laser head. Therefore, all eight levels of modulated waveforms are present at the input of the MUX at all times. Only one is routed to the laser head, depending on the required modulation level stored in the SRAM.

Generally, one should be very cautious about using the output of a MUX, since during the period when MUX select bits are changing, the MUX output will usually be glitching, until the MUX select bits are stabilized. This behavior is due to the fact that all the select bits do not arrive at exactly the same time. Even if they did arrive at the same time, delay path variations and logic switching internal to MUX may create a glitch on the output. As a word of caution, the above mentioned scheme should not be used for a clocking scheme. When a new MUX input is se-

lected, there will probably be a glitch on the output. If the first cycle glitch can be tolerated or masked, then this scheme can be used for clock distribution. A delayed clocked version of the MUX output could be safely used for clock distribution. Obviously, the delay should be larger than the maximum propagation delay of MUX. For this particular application, the glitch is not as important, because the total duration of ON and OFF times of the laser beam is the concern, not the rising or falling edges of the waveform. Also, the laser head is turned off during the MUX address selection, totally masking any possible glitches.

Configuring RoboClock and Design Analysis

A close observation of waveforms shown in *Figure 5* reveals the fundamental idea behind generating all six modulated waveforms. Simply by gating the 90-degree phase-shifted REF with the true and complemented version of the REF clock, all six waveforms are generated. For simplicity's sake, lets call the 90-degree phase-shifted waveform $+4t_U$, 66.67-MHz clock F, and the inverted clock $\overline{F}$. Let's look at how each one is generated.

75% HIGH: $(+4t_U)$ OR (F)

50% HIGH: F

25% HIGH: $(+4t_U)$ NOR $(\overline{F})$

75% LOW: $(+4t_U)$ NOR (F)

50% LOW: $\overline{F}$

25% LOW: $(+4t_U)$ OR $(\overline{F})$

Note that very fast NOR gates with true and complemented outputs were selected to achieve uniform delay for all outputs. Also, the 50% HIGH and 50% LOW signals are routed OR gates configured as buffers to ensure matched delay signals. Please note that all three RoboClock outputs, $+4t_U$, F, and $\overline{F}$, have the same number of loads. It is very impor-

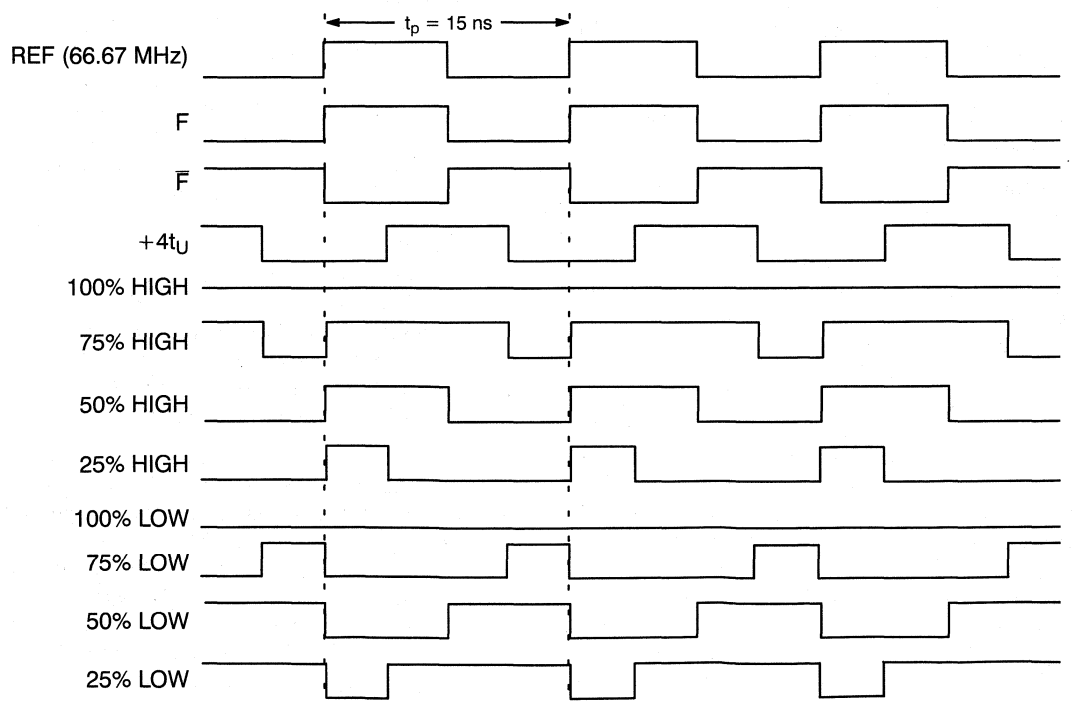


Figure 4. Generated Waveforms

tant during layout to match all the trace lengths from RoboClock to NOR gates and from the NOR gates to the MUX to prevent undesirable skew, which will translate into phase shift and pulsewidth variation on the laser beam.

Over voltage and temperature variation, all the outputs of the RoboClock are very stable. The PLL inside the RoboClock is constructed from differential stages, which makes it self-compensating against voltage and temperature variations. Consequently, RoboClock generates robust output waveforms in terms of phase and frequency. The external OR gates may distort the waveform due to the effects of voltage and temperature variation.

Earlier, the 90-degree phase-shifted waveform was equated with $+4t_U$. Let's see how that was derived. Based on *Table 1*, each time unit is calculated by the following equation:

$$t_U = 1/(F_{nom} * N) \quad \text{Eq. 1}$$

Where, as indicated in the *Table 1*, N can be any one of 44, 26, or 16 integer numbers, depending on the maximum output frequency of the RoboClock. Since the output frequency is 66.67 MHz, then FS is selected to be HIGH (frequency range of 40 to 80 MHz), N = 16, and F_{nom} = 66.67 MHz. By simply plugging the numbers in the *Equation 1*, the time unit or the t_U can be calculated as:

$t_U = 1/(66.67\ MHZ * 16)$

$t_U = 0.9375\ ns$

In terms of phase shift, if 66.67 MHz or 15-ns cycle time is 360 degrees, then the 90-degree phase-shift is essentially 15 ns divided by 4.

90 Degree Phase Shift = 15 ns / 4 = 3.75 ns

Therefore, the number of time units to shift to obtain 90-degree phase-shift, is simply derived by dividing 3.75 ns by t_U.

Number of Time Units = 3.75 ns / 0.9375 = 4

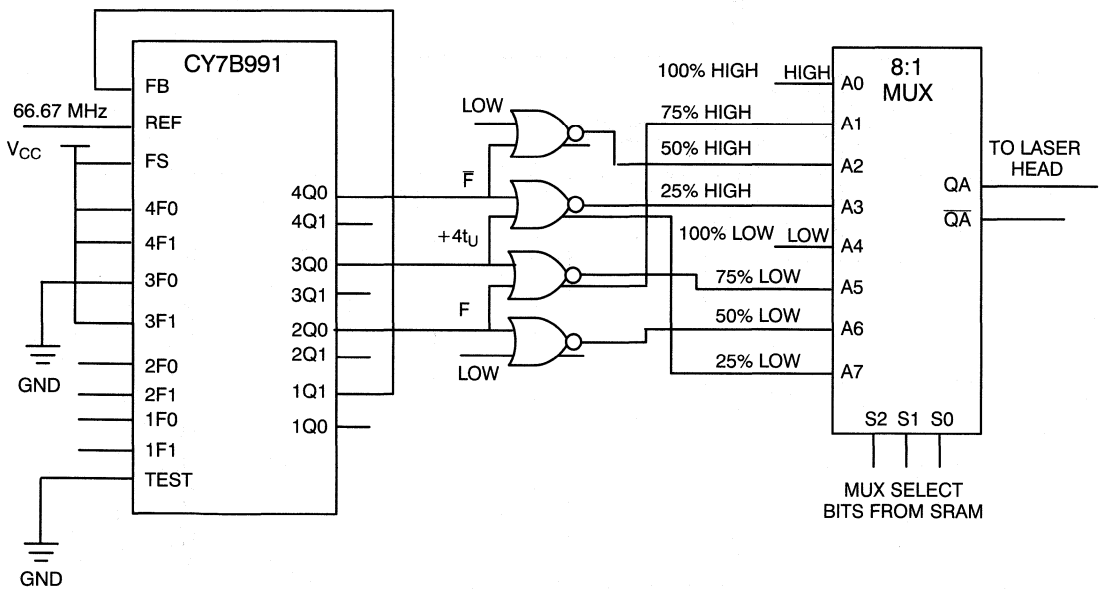


Figure 5. Simplified Laser Modulation Diagram

Therefore, in FS = HIGH mode, 4 t_U translates into 90-degree phase-shift. An observant reader might have already noticed the fact that in FS = HIGH mode, N is equal to 16, based on *Table 1*. This means that, in FS = HIGH mode, an entire cycle or 360 degrees, is equivalent to the delay through 16 stages of ring oscillator, and each stage represents one t_U (In FS = LOW the number of delay stages or N is 44 and in FS = MID it is 26. As shown in *Figure 6*, note that the actual number of ring oscillator buffer stages is half the N, because each cycle contains a LOW and HIGH period, which means to complete a full cycle the signal propagates through the ring oscillator twice.) In order to derive a 90-degree phase-shift, all one needs to do is to multiply N by 1/4 (where 90/360 = 1/4 cycle). Therefore, 16/4 = 4 time units, in FS = HIGH mode, represents a 90 degree phase shift.

The same simple methodology can be used to figure out the number of time units of delay or advance to implement a n arbitrary degree of phase shift. The number of units of skew (T_U needed for an arbitrary phase shift is calculated as follows:

$$\#\ T_U = \frac{N - phaseshift}{360} \qquad \text{Eq. 2}$$

Rounding this number to the nearest integer will introduce a small phase error from the desired phase shift. For example, if 60 degree phase shift is required when FS = LOW, then:

Required Phase Shift = 60/360 = 1/6 cycle

*Number of Time Units = N * 1/6 = 44/6 = 7.33 t_U*

Since the number of PLL stages for each FS mode is an integer number, then the nearest time unit shift, in this case, will be seven. Obviously, this will create a phase error of 0.33 t_U.

Let's go back and discuss how the +4t_U, F and $\overline{F}$ waveforms are generated by RoboClock. Since multiple outputs from a single RoboClock is expected, as an exercise, let's use the three-table method. There are several solutions for the current requirement; only one of the simplest is presented. By observing *Table 3*, titled as "1Qx/2Qx Output Connected to FB Input," one may select the 1Q0 to be used as FB, and leave the corresponding inputs floating (1F0=1F1=MID). Thus, essentially, we have ac-

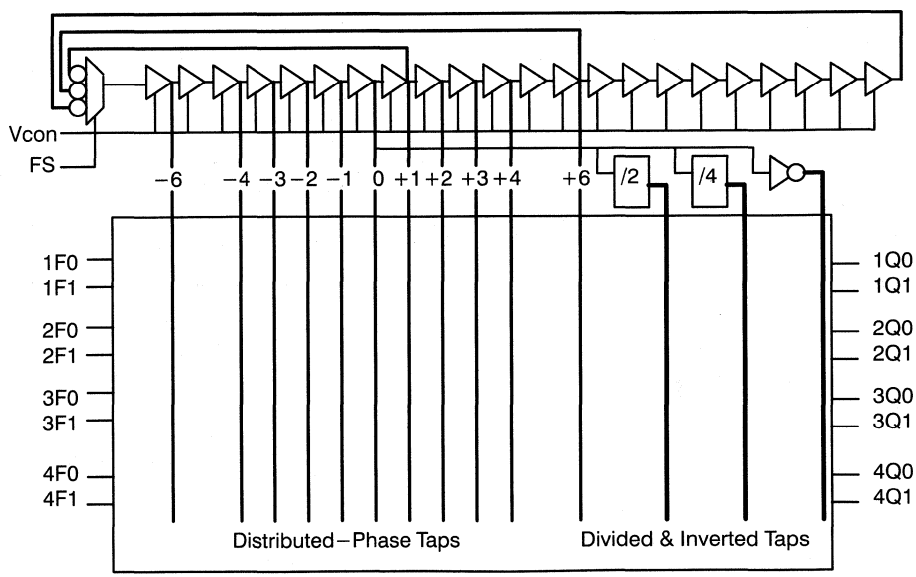


Figure 6. Distributed-Phase Clock Oscillator and Output Adjust Matrix

cess to all the terms available in row two of the given table. Now, by selecting the inputs, the RoboClock may be configured to generate various waveforms. By setting 2F0=2F1=MID or floating the 2Fx inputs, the 2Q0 and 2Q1 will generate the required F signal (also, 1Qx could have been used for F signal). Setting 3F0=LOW and 3F1=HIGH will generate $+4t_U$ signal at 3Q0 and 3Q1. Finally, setting 4F0=4F1=HIGH will generate the $\overline{F}$ signal. As mentioned earlier, by fixing the feedback term, in this case, all the elements of the 2nd row of *Table 3* are available for the user. RoboClock is a flexible clock distribution buffer that may be reconfigured easily during the prototyping phase of a design. For example, if instead of generating a 0 T_U output on 2Qx, it is required to have the 2Qx signals advanced by $2t_U$, then this can be accomplished simply by setting 2F0=HIGH and 2F1=LOW. This is one of the commonly used features of RoboClock that offers thousands of variations for prototyping purposes. Often, during prototyping phase, some modification in the clock or the waveform is required.

RoboClock, with its thousands of configurations, resolves some of the unexpected timing problems. In fact, during prototyping, if multiple timing variations are expected, it is advised to use a three-state register to drive the RoboClock inputs. Then, each output of the register must have a 10K pull-up and 10K pull-down resistor, to ensure that MID level is held at half the supply voltage when the register is three-stated. In this case, the user may write a word in the input register, and by doing so, reconfigure the entire operation of the RoboClock, without using any jumpers. Note that not all the inputs need to be reconfigurable for a given design. Often, a couple of reconfigurable signals are all that is needed. In that case, most inputs may be hardwired and the inputs needed to reconfigure various outputs may be registered with the 10K pull-up and pull-down resistors.

Summary

RoboClock was used to generate very precise complex waveforms to enhance color images and increase the resolution of laser printers. Even though RoboClock is widely used for clock distribution, this application note presented an alternative use of RoboClock for complex precise waveform generation.

Table 3. 1Qx or 2Qx Output Connected to FB Input (Part 1)

Feedback Section		2Qx Output Section									
1Qx(2Qx)◗FB		Configuration Block	2Qx(1Qx) Outputs with respect to REF								
			2F1 (1F1)	L	L	L	M	M	M	H	H	H
1F1 (2F1)	1F0 (2F0)		2F0 (1F0)	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t	+8t
L	M			−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t
L	H			−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t
M	L			−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t
M	M			−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t
M	H			−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t
H	L			−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t
H	M			−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t
H	H			−8t	−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t

Table 3. 1Qx or 2Qx Output Connected to FB Input (Part 2)

Feedback Section		3Qx Output Section									
1Qx(2Qx)→FB		Configuration Block	3Qx Output with respect to REF								
			3F1	L	L	L	M	M	M	H	H	H
1F1 (2F1)	1F0 (2F0)		3F0	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		+4t, f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t, f/4
L	M			+3t, f/2	−3t	−1t	+1t	+3t	+5t	+7t	+9t	+3t, f/4
L	H			+2t, f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t, f/4
M	L			+1t, f/2	−5t	−3t	−1t	+1t	+3t	+5t	+7t	+1t, f/4
M	M			0t, f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t, f/4
M	H			−1t, f/2	−7t	−5t	−3t	−1t	+1t	+3t	+5t	−1t, f/4
H	L			−2t, f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t, f/4
H	M			−3t, f/2	−9t	−7t	−5t	−3t	−1t	+1t	+3t	−3t, f/4
H	H			−4t, f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t, f/4

Table 3. 1Qx or 2Qx Output Connected to FB Input (Part 3)

Feedback Section		4Qx Output Section									
1Qx(2Qx)→FB		Configuration Block	4Qx Output with respect to REF								
			4F1	L	L	L	M	M	M	H	H	H
1F1 (2F1)	1F0 (2F0)		4F0	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		+4t, f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t, INV
L	M			+3t, f/2	−3t	−1t	+1t	+3t	+5t	+7t	+9t	+3t, INV
L	H			+2t, f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t, INV
M	L			+1t, f/2	−5t	−3t	−1t	+1t	+3t	+5t	+7t	+1t, INV
M	M			0t, f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t, INV
M	H			−1t, f/2	−7t	−5t	−3t	−1t	+1t	+3t	+5t	−1t, INV
H	L			−2t, f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t, INV
H	M			−3t, f/2	−9t	−7t	−5t	−3t	−1t	+1t	+3t	−3t, INV
H	H			−4t, f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t, INV

Table 4. 3Qx Output Connected to FB Input (Part 1)

Feedback Section		1Qx, 2Qx Output Section									
3Qx ➧ FB		Configuration Block	1Qx (2Qx) Output Delay to REF								
			1F1, (2F1)	L	L	L	M	M	M	H	H	H
3F1	3F0		1F0, (2F0)	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		−4t, f*2	−3t, f*2	−2t, f*2	−1t, f*2	0t, f*2	+1t, f*2	+2t, f*2	+3t, f*2	+4t, f*2
L	M			+2t	+3t	+4t	+5t	+6t	+7t	+8t	+9t	+10t
L	H			0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t	+8t
M	L			−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t
M	M			−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t
M	H			−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t
H	L			−8t	−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t
H	M			−10t	−9t	−8t	−7t	−6t	−5t	−4t	−3t	−2t
H	H			−4t, f*4	−3t, f*4	−2t, f*4	−1t, f*4	0t, f*4	+1t, f*4	+2t, f*4	+3t, f*4	+4t, f*4

Table 4. 3Qx Output Connected to FB Input (Part 2)

Feedback Section		3Qx Output Section									
3Qx ➧ FB		Configuration Block	4Qx Output with Delay to REF								
			4F1	L	L	L	M	M	M	H	H	H
3F1	3F0		4F0	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		0t	−6t, f*2	−4t, f*2	−2t, f*2	0t, f*2	+2t, f*2	+4t, f*2	+6t, f*2	INV, f*2
L	M			+6t, f/2	0t	+2t	+4t	+6t	+8t	+10t	+12t	+6t, INV
L	H			+4t, f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t, INV
M	L			+2t, f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t, INV
M	M			0t, f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t, INV
M	H			−2t, f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t, INV
H	L			−4t, f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t, INV
H	M			−6t, f/2	−12t	−10t	−8t	−6t	−4t	−2t	0t	−6t, INV
H	H			0t, f*2	−6t, f*4	−4t, f*4	−2t, f*4	0t, f*4	+2t, f*4	+4t, f*4	+6t, f*4	INV, f*4

Table 5. 4Qx Output Connected to FB Input (Part 1)

Feedback Section		1Qx, 2Qx Output Section									
4Qx ➧ FB		Configuration Block	1Qx, 2Qx Output with respect to REF								
			1F1, 2F1	L	L	L	M	M	M	H	H	H
4F1	4F0		1F0, 2F0	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		−4t, f*2	−3t, f*2	−2t, f*2	−1t, f*2	0t, f*2	+1t, f*2	+2t, f*2	+3t, f*2	+4t, f*2
L	M			+2t	+3t	+4t	+5t	+6t	+7t	+8t	+9t	+10t
L	H			0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t	+8t
M	L			−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t
M	M			−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t
M	H			−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t
H	L			−8t	−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t
H	M			−10t	−9t	−8t	−7t	−6t	−5t	−4t	−3t	−2t
H	H			−4t, INV	−3t, INV	−2t, INV	−1t, INV	0t, INV	+1t, INV	+2t, INV	+3t, INV	+4t, INV

Table 5. 4Qx Output Connected to FB Input (Part 2)

Feedback Section		1Qx, 2Qx Output Section									
4Qx ➧ FB		Configuration Block	4Qx Output with respect to REF								
			3F1	L	L	L	M	M	M	H	H	H
4F1	4F0		3F0	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		0t	−6t, f*2	−4t, f*2	−2t, f*2	0t, f*2	+2t, f*2	+4t, f*2	+6t, f*2	0t, f/2
L	M			+6t, f/2	0t	+2t	+4t	+6t	+8t	+10t	+12t	+6t, f/4
L	H			+4t, f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t, f/4
M	L			+2t, f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t, f/4
M	M			0t, f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t, f/4
M	H			−2t, f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t, f/4
H	L			−4t, f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t, f/4
H	M			−6t, f/2	−12t	−10t	−8t	−6t	−4t	−2t	0t	−6t, f/4
H	H			INV, f/2	−6t, INV	−4t, INV	−2t, INV	0t, INV	+2t, INV	+4t, INV	+6t, INV	INV, f/4

Using the CY7B991 with the 50-MHz 486 Cache Module and the 40-MHz R3000

As clock speeds increase, clock distribution plays an increasingly significant role in system design. Not only do high-speed systems have less time to waste on clock skew problems, but as systems become more and more synchronous, clock signals must be distributed to more and more circuits on the board over a greater distance. Therefore the challenge for you, the designer, is to get all clock signals to their destination at very nearly the same time without sacrificing signal quality. You must minimize clock-signal skew while maximizing clock-signal integrity.

This application note includes

- A brief introduction to clock distribution
- A CY7B991/CY7B992 functional description
- A 486DX design example
- A MIPS R3000 design example

The Challenges of Clock Distribution

Many factors prevent the minimization of clock skew while maximizing clock-signal integrity. These include trace length, trace routing, capacitive loading of the clock line, clock-line termination, trace impedance, clock-driver strength, clock-edge transition time, and clock-driver output skew.

You have control over many of these factors. By making the trace length of clock signals approximately the same length, for example, the clock signal will reach its destination without the clock skew caused by the variance in propagation time of signal down traces of different lengths.

Traces, however, cannot always be made the same length due to board-space considerations or the fact that the clock driver output may be driving multiple loads, all of which lie on various points along the clock trace. Also, when the clock trace exceeds a particular length, the signal has to be considered a transmission line. Transmission lines must be properly terminated according to the characteristic impedance of the trace to prevent voltage reflections from causing clock-signal degradation.

Signal degradation can also be prevented by reducing the capacitive loading on clock lines or by eliminating crosstalk caused by routing high-frequency signals too close together.

You can avoid clock distribution problems created by weak clock drivers, slow clock-edge transition times, and poor clock-driver output skew with the proper selection of a clock driver device. Many devices that have been used in the past for clock distribution, such as flip-flops, buffers, and line drivers, do not explicitly express output skew. The difference in output delay times must be inferred from their datasheets.

The 74FCT244, for example, is a popular octal buffer that is used to drive multiple clock lines by tying all of the buffer inputs to a common clock source. You must calculate the output skew from this type of clock driver, since it is not stated in the datasheet, as the difference between the maximum and the minimum propagation delay through the device. This device can have a skew of 3 ns or more between outputs on the same package. In a 40-MHz system this accounts for more than 10% of the cycle time budget. Obviously this is an unacceptable solution.

Fortunately, the CY7B991 clock drivers from Cypress Semiconductor are designed specifically to distribute clock signals in high-speed environments.

The CY7B991 Functional Description

The following part description focuses on those aspects of the CY7B991 that pertain to the two design examples in this application note. We will focus on the low output-skew parameters of these devices along with their programmable features including output phase shifting, output frequency division/multiplication, and output inversion.

General Information

Internally code named "RoboClock," this family consists of two parts: the CY7B991 and the CY7B992. The CY7B991 has TTL (0 to 3V) outputs and will source 16 mA and sink 24 mA while the CY7B992 has CMOS (0 to V_{CC}) outputs and will source 40 mA and sink 46 mA.

The CY7B991 (*Figure 1*) contains eight outputs grouped in four sets of two. Two select lines control the functionality of each pair of outputs. Both outputs of an output pair function identically based on the selection of the function select inputs.

The input-to-output delay of the device is kept to under ±700 ps by employing a phase-locked-loop (PLL) architecture. The magnitude of this parameter is primarily a function of frequency, and typically will be very close to 0 ps.

The propagation delay parameter, t_{PD}, is shown in *Figure 2*. Connecting an output to the FB (feedback) input of the device causes the PLL to align the output both in phase and in frequency with the REF (reference) input. This allows very low input-to-output delay as well as allowing a system to use more than one of these devices for clock distribution and still maintain very low device-to-device skew between various clocks.

Figure 2 also shows other datasheet parameters that are related to the REF and FB inputs. t_{REF} specifies the maximum operating frequency (minimum clock period). The CY7B991 has an output maximum operating frequency of 80 MHz. The minimum high and low input pulse width (t_{RPWH} and t_{RPWL}) is specified as 5.0 ns, so you do not have to supply the CY7B991 with a 50% duty cycle REF input in order for the outputs to have a 50% duty cycle. In fact, the high and low time of the output clock output will differ by less than ±1.4 ns regardless of the input duty cycle.

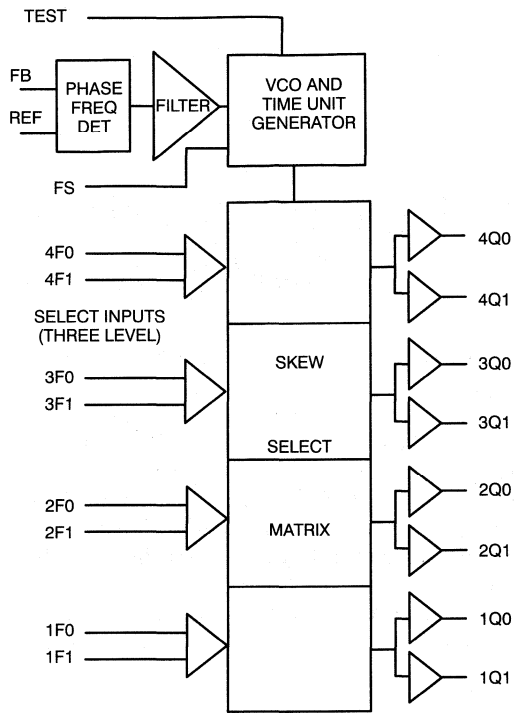


Figure 1. CY7B991 Block Diagram

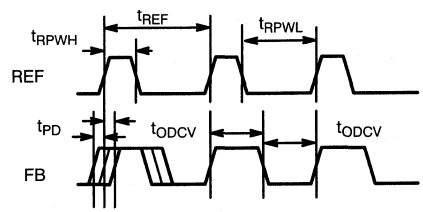


Figure 2. Datasheet Parameters

Programmable Delay Adjustment

In addition to low skew outputs, two select lines (xF1, xF0) control each set of two outputs (xQ1, xQ0) providing output programmability. These three-state function select inputs allow you to hard-wire each output pair to one of nine delay or function configurations. Each function select input can be connected to V_{CC} (HIGH), left unconnected (MID), or connected to ground (LOW). *Table 1* shows the programmable skew configurations assuming that an output with both function select inputs left unconnected (xFx=mid) is used as the FB input.

Table 1. Programmable Skew Configurations

Function Selects		Output Functions		
1F1, 2F1, 3F1, 4F1	**1F0, 2F0, 3F0, 4F0**	**1Q0, 1Q1, 2Q0, 2Q1**	**3Q0, 3Q1**	**4Q0, 4Q1**
LOW	LOW	$-4t_U$	Divide by 2	Divide by 2
LOW	MID	$-3t_U$	$-6t_U$	$-6t_U$
LOW	HIGH	$-2t_U$	$-4t_U$	$-4t_U$
MID	LOW	$-1t_U$	$-2t_U$	$-2t_U$
MID	MID	$0t_U$	$0t_U$	$0t_U$
MID	HIGH	$+1t_U$	$+2t_U$	$+2t_U$
HIGH	LOW	$+2t_U$	$+4t_U$	$+4t_U$
HIGH	MID	$+3t_U$	$+6t_U$	$+6t_U$
HIGH	HIGH	$+4t_U$	Divide by 4	Inverted

t_U in *Table 1* refers to the programmable time unit by which an output may be skewed with respect to the REF input. t_U is a function of the frequency at which the 1Q0 output is operating. The CY7B991 offers frequency agility from 15 MHz to 80 MHz with the use of the three-state frequency select input, FS. *Table 2* shows the operating frequency range for various connections of FS. The appropriate FS selection should be made based on the anticipated operating frequency of the 1Q0 output. The FS pin may have two possible connections when operating at certain frequencies. You should select the appropriate connection of the FS pin, in this case, based on the value of the time unit, t_U, required for your application. *Table 2* also shows the programmable time unit (t_U) calculation.

For example, if you were designing a system with a clock speed of 33 MHz, you would, according to *Table 2*, leave the FS pin unconnected. The programmable skew unit, based on this assumption, would be

$$t_U = \frac{1}{f_{1Q0} \times N} = \frac{1}{33\ MHz \times 26} = 1.17\ ns \qquad \text{Eq. 1}$$

In other words, you can adjust the position with which the rising and falling edges of the outputs move with respect to the corresponding REF input edge with a resolution of 1.17 ns when operating the device at 33 MHz.

Table 2. Frequency Range Select and t_U Calculation

FS	f_{1Q0} (MHz)		$t_U = \frac{1}{f_{1Q0} \times N}$ where N =	Approximate Frequency At Which t_U = 1.0 ns (MHz)
	Min.	Max.		
LOW	15	30	44	22.7
MID	25	50	26	37.5
HIGH	40	80	16	62.5

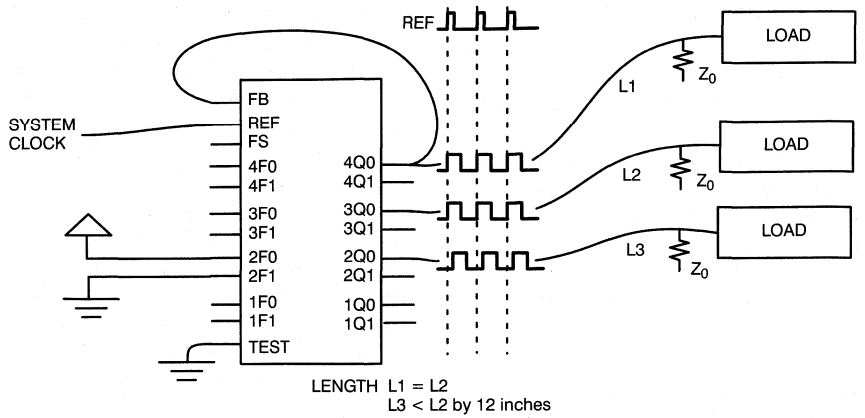


Figure 3. Programmable-Skew Clock Driver

This adjustability allows for the compensation of traces of varying lengths as shown in *Figure 3*. For example, the difference of 12 inches in trace length between two clock inputs driven by two clock outputs would account for 2 ns in clock skew due to the difference in propagating delay along these signal lines. You can compensate for these delays with the CY7B991 by either advancing the clock on the longer trace or retarding the clock on the slower trace by two time units. If the 2Q0 output drives the shortest trace in a 33-MHz system, then the 2F1 input could be tied LOW and the 2F0 input could be tied HIGH to retard the clock signal on the 2Q0 and 2Q1 output by 2.34 ns, allowing it to arrive at approximately the same time as the signal on the longer trace.

The programmability can also be used to adjust the clocks to synchronous RAMs, ASICs, and programmable logic to compensate for set-up and hold time mismatches. Let us use a PAL22V10 to illustrate this point. The 22V10 has a 0-ns hold time after the clock is HIGH. If the outputs from this PAL feed another device that requires a 5-ns hold time after clock HIGH, it will probably not work. To solve this problem, you can use the CY7B991 to generate the clocks for both devices. Delaying the time at which the 22V10 receives its clock by 6 ns guarantees that the other device will see an equivalent hold time of at least 5 ns when operating at 33 MHz. You can accomplish this by driving the 22V10 clock input with the 3Q0 output and connecting the 3F1 input to ground and leaving the 3F0 input unconnected.

Programmable Function Configuration

In addition to programmable delay adjustment, the CY7B991 also offers programmable function abilities. With proper selection of the function select inputs, the outputs can divide the clock by two, divide it by four, and invert the clock when compared to the REF input.

The 3Qx outputs have two frequency division output selections. If 3F1 and 3F0 are both tied to ground, then the frequency of the 3Qx outputs will be half that of the REF frequency. If 3F1 and 3F0 are both tied to V_{CC}, then the 3Qx outputs will be one quarter that of the REF frequency.

The 4Qx outputs have almost identical output selections as the 3Qx outputs with the exception of the selection made when both 4F1 and 4F0 are tied HIGH. In this case, the output phase is inverted with respect to the REF input.

Output Skew Parameters

Several parameters determine the accuracy and skew of the outputs. For completeness, we define skew as the time between the earliest and the latest output transition among all outputs with the same t_U delay. t_{SKEWPR} (*Figure 4*) is the skew between

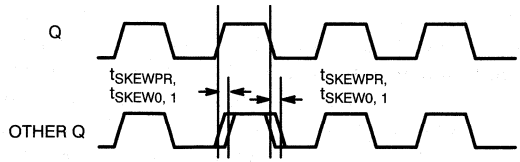


Figure 4. t_{SKEWPR}, t_{SKEW0}, t_{SKEW1}

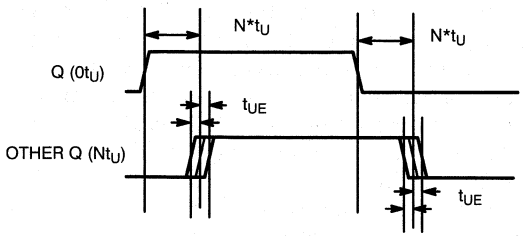


Figure 5. Programmable Skew Unit Error

outputs of the same output pair when all outputs are selected for $0t_U$ skew adjustment (xFx = MID). The maximum value of t_{SKEWPR} is less than 250 ps. The typical magnitude of this skew parameter is 100 ps or less. The major contributor to this skew parameter is the variance in output loading.

t_{SKEW0} is the amount of skew between all outputs of the device when all outputs are selected for $0t_U$ skew adjustment. The datasheet states that the value of t_{SKEW0} is equal to or less than 700 ps. The typical value of this parameter is less than 300 ps caused primarily by a small difference in the internal routing of the device and output loading mismatches.

We have already discussed the method by which we can calculate t_U. t_{UE} is the amount of error associated with the placement of t_U. This parameter has a maximum value of ±700 ps around the position indicated by the value of t_U (*Figure 5*). Typically this parameter will be 0 ps, indicating that the position of t_U will be the same as that calculated using the formula. This means that if, as in the previous example, we calculate t_U = 1.17 ns, this edge will appear at the outputs between 417 ps and 1.87 ns after the corresponding edge of the REF input. The primary contributors to this error term are the slight inaccuracy in the formula used for calculation, the slight internal component mismatches, and the slight variation in output edge placement caused by the switching of adjacent outputs.

The last parameter that is used to calculate the skew of outputs that have a programmable delay is t_{SKEW1}, which is also shown in *Figure 4*. It is the skew between outputs that have the same t_U selection. For example, if two outputs have the same output skew selection of $-4t_U$ and they are operating at 33 MHz, then the edges of these outputs would appear between 3.98 ns and 5.38 ns before the corresponding edge on the REF input with a maximum skew between any of these outputs of less than 1 ns. Again, the typical value of this parameter is less than 600 ps with the majority of the skew due to the switching of adjacent outputs.

Several parameters determine the skew between the programmable function outputs. The CY7B991 has three classes of outputs: Nominal (a multiple of t_U delay), Inverted, and Divided. t_{SKEW2} governs the position between the rising edge and falling edge of Nominal-to-Inverted outputs, and Divided-to-Divided outputs. *Figure 6* shows an example of t_{SKEW2}. The rising edge of the output used as the REF feedback and the falling edge of an Inverted output will have a skew of no more than that specified by t_{SKEW2}. The datasheet maximum value of this specification is 1.5 ns with a typical value of 1.0 ns.

t_{SKEW3} defines the skew between the rising and falling edges of different classes of outputs. For example, the falling edge of an Inverted output and the falling edge of a Divided output will vary by no more than 1.2 ns. Finally, t_{SKEW4} indicates that the rising-to-falling edge skew between Nominal-to-Divided, and Divided-to-Inverted outputs will vary by no more than 1.7 ns. *Figure 7* shows an example of both t_{SKEW3} and t_{SKEW4}.

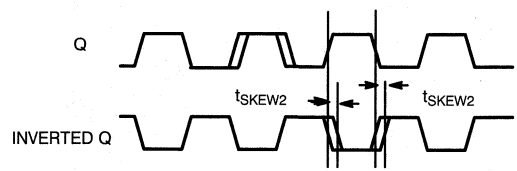


Figure 6. Example of t_{SKEW2}

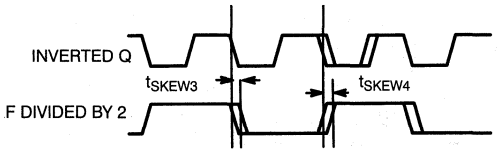


Figure 7. Examples of t_{SKEW3} and t_{SKEW4}

Using An Output without a $0t_U$ Selection as the FB Input

The previous description of the operation of the part assured that an output with a $0t_U$ (xF1, xF0 = mid) selection was used as the FB input. By stating this we insure that the other outputs are aligned strictly as a function of the REF input and their respective function select inputs. By using an output with its function select inputs other than xF1=xF0 = mid as the FB input, the additional outputs will have an alignment dependent not only on their function select inputs and the REF input frequency, but also on the configuration of the function select inputs of the output used as FB.

Connecting an output with a programmable function selected to the FB input enables the CY7B991 to multiply the incoming REF frequency and it also allows skew adjustment from −12 to + 12 time units. In all, more than 26,000 output combinations are possible with proper selection of the FB input and proper connection of the function select inputs. *Tables 3* through *5* show the various output possibilities available.

As shown in *Table 1*, the 1Qx and 2Qx outputs have the same possible selections with the same configuration of the function select inputs (1Fx, 2Fx). If a non-0-ns tap is used as the FB input, the output selections change from those given in *Table 1*. This is because the internal PLL aligns the FB input with the REF input, in both phase and frequency. Because the FB input is no longer being taken from a 0-ns tap output, the delay of all other outputs, not including the set of outputs from which the FB input came, will have delays dependent not only on the select pin (xFx) connections, but also on the select pin connections of the output used as the FB input. Regardless of which select pin connections are being used for the outputs used as the FB input, these outputs will have a 0-ns output delay with respect to the REF input.

Table 3. 1Qx or 2Qx Output Connected to FB Input

FeedBack Section			2Qx(1Qx) Output Section								
1Qx(2Qx) ⬮ FB		Configuration Block	2Qx(1Qx) Outputs with respect to REF								
			2F1 (1F1)	L	L	L	M	M	M	H	H	H
1F1 (2F1)	1F0 (2F0)		2F0 (1F0)	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t	+8t
L	M			−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t
L	H			−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t
M	L			−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t
M	M			−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t
M	H			−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t
H	L			−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t
H	M			−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t
H	H			−8t	−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t

Table 3. 1Qx or 2Qx Output Connected to FB Input (continued)

FeedBack Section			3Qx Output Section								
1Qx(2Qx) ➧ FB		Configuration Block	3Qx Output with respect to REF								
			3F1	L	L	L	M	M	M	H	H	H
1F1 (2F1)	1F0 (2F0)		3F0	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		+4t f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t f/4
L	M			+3t f/2	−3t	−1t	+1t	+3t	+5t	+7t	+9t	+3t f/4
L	H			+2t f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t f/4
M	L			+1t f/2	−5t	−3t	−1t	+1t	+3t	+5t	+7t	+1t f/4
M	M			0t f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t f/4
M	H			−1t f/2	−7t	−5t	−3t	−1t	+1t	+3t	+5t	−1t f/4
H	L			−2t f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t f/4
H	M			−3t f/2	−9t	−7t	−5t	−3t	−1t	+1t	+3t	−3t f/4
H	H			−4t f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t f/4

Table 3. 1Qx or 2Qx Output Connected to FB Input (continued)

FeedBack Section			4Qx Output Section								
1Qx(2Qx) ⧫ FB		Configuration Block	4Qx Output with respect to REF								
			4F1	L	L	L	M	M	M	H	H	H
1F1 (2F1)	1F0 (2F0)		4F0	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		+4t f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t INV
L	M			+3t f/2	−3t	−1t	+1t	+3t	+5t	+7t	+9t	+3t INV
L	H			+2t f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t INV
M	L			+1t f/2	−5t	−3t	−1t	+1t	+3t	+5t	+7t	+1t INV
M	M			0t f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t INV
M	H			−1t f/2	−7t	−5t	−3t	−1t	+1t	+3t	+5t	−1t INV
H	L			−2t f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t INV
H	M			−3t f/2	−9t	−7t	−5t	−3t	−1t	+1t	+3t	−3t INV
H	H			−4t f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t INV

Table 3 shows output configurations when either 1Qx or 2Qx outputs are used as the FB input. This table applies to either set of outputs since they have the same possible output selections. This table is broken up into four sections: FeedBack, 2Qx Output, 3Qx Output, and 4Qx Output. The first section indicates the status of the function select pins of the 1Qx (2Qx) outputs used as the FB input. The second section indicates the 2Qx (1Qx) output combinations available for a given selection of the 1Fx and 2Fx select lines. The third and fourth sections give the output possibilities for the 3Qx and 4Qx outputs, respectively.

In the first section, the first two columns indicate the 1Fx select line connections. As stated previously, this table also can be used for the 2Qx used as the FB input. For the remainder of the discussion on *Table 3* we will assume that a 1Qx output is used as the FB input and leave the case where a 2Qx output is used as the FB input as an obvious extension to this illustration.

The remaining three sections list the output configuration and selections for given connections of 1F1 and 1F0. The table, therefore, gives 2Qx, 3Qx, and 4Qx outputs for given connections of 1F1 and 1F0. These sections are further broken down into two blocks, Configuration and Output Selection. The first block gives the various connections of the select lines for the three output sets. These are indicated by xF1 and xF0 at the beginning of each section. Both the select line values of the outputs used as the FB input as well as the select line values of the other

outputs are shown in order to specify the output selection. All values in the table are given with respect to the REF input.

The Output Selection Block of the last three sections indicates the various output selections on the 2Qx outputs; for example, possible with various input connections of the 1Fx select lines. For example, the first row of the Output Selection Block in the 2Qx Output Section indicates the nine possible values that the 2Qx outputs may have when the 1Fx inputs both are tied to ground. The 2Qx outputs, in this case, may take on any value between 0t and 8t in $1t_U$ increments.

To understand why these values are different than those values of 2Qx found in *Table 1,* we must understand how the device operates. First of all, it is important to remember that the CY7B991 internal PLL aligns the REF input with the FB input. When the 1F1 and 1F0 function select inputs are both connected to ground, this selects, according to *Table 1*, the $-4t_U$ function on the 1Qx outputs. If one of these outputs is used as the FB input, the PLL will align this output with its REF input: it will no longer have any delay with respect to the REF input even though its function select inputs are not left unconnected. Since an output that had a $-4t_U$ now is aligned with REF, the internal time reference of the device has shifted by the same amount ($4t_U$). For this reason the outputs 2Qx, 3Qx, and 4Qx have an output magnitude that is shifted $4t_U$ from that which they would if an output with a $0t_U$ selection was used as the FB input. This is why the 2Qx outputs, when both 1F0 and 1F1 are tied to ground, have an output range of $0t_U$ to $8t_U$ instead of $-4t_U$ to $+4t_U$. The range length is still $8t_U$.

For example, if 1F1 is unconnected (M), 1F0 is tied HIGH, 2F1 is tied LOW , and 2F0 is tied HIGH, the output selections on the 2Qx outputs (according to the sixth row and the third column of the Output Selection Block in *Table 3*) will rise 3 time units before the rising edge of the REF input. Similarly, if a 2Qx output is used as FB and the 2Fx select lines are connected in the same manner (M, H) as the 1Fx lines were, and if the 1Fx select lines are connected in the same manner as the 2Fx lines were (L, H), the 1Qx outputs will rise 3 time units before the rising edge of the REF input. This result is found in the same location in *Table 3* as the previous result.

If the 1F1 select pin is tied HIGH, the 1F0 select pin is tied HIGH, 3F1 is tied LOW, and 3F0 is tied LOW, the first column, ninth row in the 3Qx Output Section in *Table 3* states that the frequency of the 3Qx outputs will be half of the REF input, and that both the rising edge and the falling edge of the outputs will precede the REF input rising edge by four time units.

If the 2Qx output is used as the FB input, the 2F1 pin is tied LOW, the 2F0 pin is tied HIGH, the 4F1 pin is tied HIGH, and the 4F0 pin is tied HIGH, the entry in the 4Qx Output Section in *Table 3* indicates that the falling edge of the 4Qx outputs will happen 2 time units after the rising edge of the REF input. In other words, this output is delayed and inverted.

Tables 4 and *5* look slightly different from *Table 3*. These two tables contain only three sections as opposed to four sections because the second section (1Qx, 2Qx Output Section) applies to both the 1Qx and 2Qx outputs. These outputs have the same output possibilities for a given selection of the 3Fx function select inputs (*Table 4*) or 4Fx function select inputs (*Table 5*).

Notice that in both the first and last row in the Output Selection Block, nearly every column contains a f*2, a f*4 as seen in *Table 4*, or an INV as seen in *Table 5*. The f*2 and f*4 symbols represent the REF frequency multiplied by two and four respectively, and the INV symbol represents the inversion of the REF frequency.

When the 3F1 and 3F0 are tied to ground, as shown in *Table 4*, the outputs are selected to divide the REF frequency by 2. If one of these outputs is used as the FB input, however, the PLL once again aligns this output with the REF input. Since this output now runs at the same rate as the REF input (instead of REF/2) every other output will run at twice its normal rate.

This same situation happens when 3F1 and 3F0 are tied HIGH. In this case, the 3Qx outputs, when used as the FB input, will run at the same rate as REF instead of REF/4 in *Table 4*. This will leave the 3Qx outputs operating at the REF frequency and every

other output running at four times the rate at which it is selected.

If a 3Qx output is used as the FB input, the 3F1 pin is tied LOW, the 3F0 pin is tied LOW, the 4F1 pin is tied HIGH, and the 4F0 pin is tied HIGH, the entry in the 4Qx Output Section in *Table 4* indicates that the frequency of the 4Qx outputs will be twice that of the REF frequency and that the falling and rising edge of the 4Qx outputs will be aligned with the rising edge of the REF input. In other words the 4Qx outputs will have twice the frequency of the REF input and be 180° out of phase with the REF input.

When both 4F1 and 4F0 are connected to V_{CC}, as shown in *Table 5*, the outputs are selected to be 180° out of phase with respect to the REF frequency. But because one of these outputs is connected to FB, the internal PLL aligns its rising edge with the REF frequency, causing all of the other outputs to be inverted with respect to the REF input.

Table 4. 3Qx Output Connected to FB Input

FeedBack Section			1Qx, 2Qx Output Section									
3Qx ➧ FB		Configuration Block	1Qx, 2Qx Output Delay to REF									
			1F1, 2F1	L	L	L	M	M	M	H	H	H
3F1	3F0		1F0, 2F0	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		−4t f*2	−3t f*2	−2t f*2	−1t f*2	0t f*2	+1t f*2	+2t f*2	+3t f*2	+4t f*2
L	M			+2t	+3t	+4t	+5t	+6t	+7t	+8t	+9t	+10t
L	H			0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t	+8t
M	L			−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t
M	M			−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t
M	H			−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t
H	L			−8t	−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t
H	M			−10t	−9t	−8t	−7t	−6t	−5t	−4t	−3t	−2t
H	H			−4t f*4	−3t f*4	−2t f*4	−1t f*4	0t f*4	+1t f*4	+2t f*4	+3t f*4	+4t f*4

Table 4. 3Qx Output Connected to FB Input (continued)

FeedBack Section			4Qx Output Section									
3Qx ⧫ FB		Configuration Block	4Qx Output Delay to REF									
			4F1	L	L	L	M	M	M	H	H	H
3F1	3F0		4F0	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		0t	−6t f*2	−4t f*2	−2t f*2	0t f*2	+2t f*2	+4t f*2	+6t f*2	INV f*2
L	M			+6t f/2	0t	+2t	+4t	+6t	+8t	+10t	+12t	+6t INV
L	H			+4t f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t INV
M	L			+2t f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t INV
M	M			0t f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t INV
M	H			−2t f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t INV
H	L			−4t f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t INV
H	M			−6t f/2	−12t	−10t	−8t	−6t	−4t	−2t	0t	−6t INV
H	H			0t f*2	−6t f*4	−4t f*4	−2t f*4	0t f*4	+2t f*4	+4t f*4	+6t f*4	INV f*4

Table 5. 4Qx Output Connected to FB Input

FeedBack Section			1Qx, 2Qx Output Section								
4Qx ⧫ FB		Configuration Block	1Qx, 2Qx Output Delay to REF								
			1F1, 2F1	L	L	L	M	M	M	H	H	H
4F1	4F0		1F0, 2F0	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		−4t f*2	−3t f*2	−2t f*2	−1t f*2	0t f*2	+1t f*2	+2t f*2	+3t f*2	+4t f*2
L	M			+2t	+3t	+4t	+5t	+6t	+7t	+8t	+9t	+10t
L	H			0t	+1t	+2t	+3t	+4t	+5t	+6t	+7t	+8t
M	L			−2t	−1t	0t	+1t	+2t	+3t	+4t	+5t	+6t
M	M			−4t	−3t	−2t	−1t	0t	+1t	+2t	+3t	+4t
M	H			−6t	−5t	−4t	−3t	−2t	−1t	0t	+1t	+2t
H	L			−8t	−7t	−6t	−5t	−4t	−3t	−2t	−1t	0t
H	M			−10t	−9t	−8t	−7t	−6t	−5t	−4t	−3t	−2t
H	H			−4t INV	−3t INV	−2t INV	−1t INV	0t INV	+1t INV	+2t INV	+3t INV	+4t INV

Table 5. 4Qx Output Connected to FB Input (continued)

FeedBack Section			4Qx Output Section									
4Qx ◗ FB		Configuration Block	3Qx Output Delay to REF									
			3F1	L	L	L	M	M	M	H	H	H
4F1	4F0		3F0	L	M	H	L	M	H	L	M	H
L	L	Output Selection Block		0t	−6t f*2	−4t f*2	−2t f*2	0t f*2	+2t f*2	+4t f*2	+6t f*2	0t f/2
L	M			+6t f/2	0t	+2t	+4t	+6t	+8t	+10t	+12t	+6t f/4
L	H			+4t f/2	−2t	0t	+2t	+4t	+6t	+8t	+10t	+4t f/4
M	L			+2t f/2	−4t	−2t	0t	+2t	+4t	+6t	+8t	+2t f/4
M	M			0t f/2	−6t	−4t	−2t	0t	+2t	+4t	+6t	0t f/4
M	H			−2t f/2	−8t	−6t	−4t	−2t	0t	+2t	+4t	−2t f/4
H	L			−4t f/2	−10t	−8t	−6t	−4t	−2t	0t	+2t	−4t f/4
H	M			−6t f/2	−12t	−10t	−8t	−6t	−4t	−2t	0t	−6t f/4
H	H			INV f/2	−6t INV	−4t INV	−2t INV	0t INV	+2t INV	+4t INV	+6t INV	INV f/4

If a 4Qx output is used as the FB input, 4F1 and 4F0 are tied HIGH, 2F1 is tied HIGH, and 2F0 is tied LOW, the 2Qx outputs will have the inverted sense of the REF input and follow the REF input by 2 time units as shown in *Table 5*.

The CY7B991's ability to multiply the REF frequency by two or four allows the distribution of low-frequency signals across backplanes to reduce noise, for example, and then increase their frequencies locally. It also allows you to use one clock driver device in applications where the CPU may operate at different speeds than the system bus. The multiplied and divided outputs, as well as the inverted outputs, have the same time delay flexibility as the Nominal outputs due to the feedback options of the CY7B991 driver.

Using the CY7B991 with the 486DX CPU−Cache Module

Overview

The following is an example of using the CY7B991 with the popular 486DX CPU. Specifically, this design shows the interface of the Intel CPU−Cache Module running at 50 MHz to a Memory Bus Controller (MBC) running at 25 MHz. The MBC will be designed to interface directly with an existing 486DX socket on a motherboard so that an existing system's clock rate can be increased to 50 MHz with 256Kb of zero-wait-state secondary cache.

Due to its high speed, the CPU–Cache Module has tight clock skew restrictions that make the use of a clock-driver chip imperative. The CY7B991 is a perfect match for such an application.

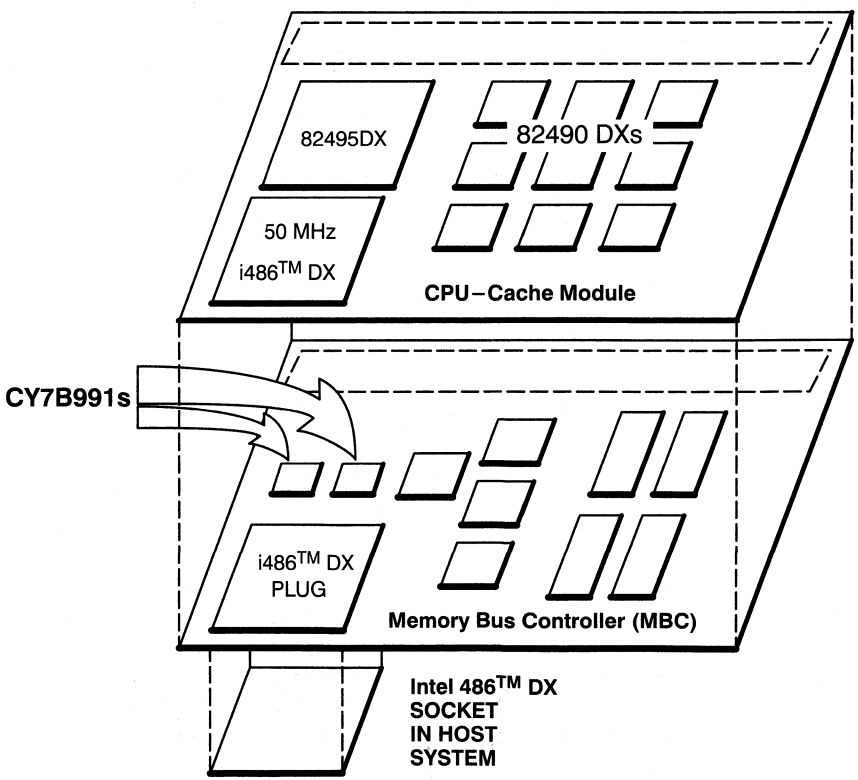
A board-level system overview is shown in *Figure 8*. This gives a concept of how the systems integrate to form the final desired result. For more details regarding the 486DX CPU–Cache Module and its specifications, see the Intel datasheet.

Memory Bus Controller

The memory bus controller lies between the CPU–Cache Module and the memory bus. The memory bus could be EISA, ISA, a proprietary memory bus implementation, or a 486DX CPU bus. In this example, the idea is to configure the MBC, its interface logic, and the CPU–Cache Module so that it can fit into an existing 486DX socket on an existing motherboard.

This example will focus on the design issues surrounding clock generation and the effects of skew. It will not cover details regarding PALs, cache protocols, and initialization of systems.

Since the MBC will be driven from an existing 486DX socket, a 25-MHz clock is already present. All clock signals for the CPU–Cache Module and MBC will be derived from this existing 25-MHz clock (SYSCLK). As seen in *Figure 9*, we will derive



Intel 486 is a trademark of Intel Corporation.

Figure 8. Board-Level System Overview

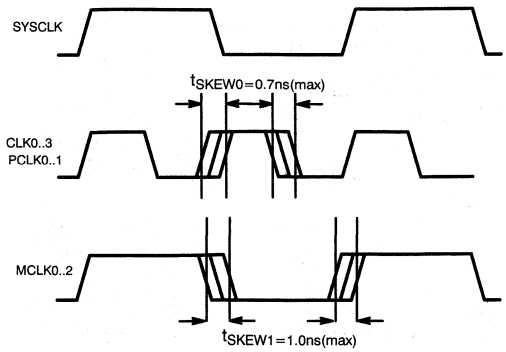


Figure 9. MBC and Module Clock Generation

MCLK0..2 at 25 MHz and CLK0..3 and PCLK0..1 at 50 MHz. Two CY7B991 devices are used.

486DX CPU–Cache Module

The Intel CPU–Cache Module is a very carefully designed multichip module with attention given to clock signal rise and fall times, equalizing trace length on the module, and equalizing trace impedance at the connector. One of the tightest AC parameter specifications given by the CPU–Cache Module datasheet concerns the maximum skew between CLK0, CLK1, CLK2, and CLK3. This parameter, specified as t_{6a}, is 0.7 ns. The CY7B991 specifies its maximum output-to-output skew to be 0.7 ns in zero-skew mode, which makes it a perfect match with the CPU–Cache Module. The other tight specification in the Intel datasheet is the maximum skew between MCLK0, MCLK1, and MCLK2. This parameter, t_{6b}, is 1.0 ns.

CLK0..CLK3 provide the fundamental timing for the module and all of its components. There are four separate clock inputs to reduce loading, equalize rise and fall times, and equalize the loading on the traces within the module. PCLK0..1 are state machine clocks necessary for MBC implementation. They are not described in any detail here and are only shown for completeness. Due to the CY7B991's paired outputs design, it will be able to drive all six CLK signals with only three paired outputs. This leaves two other outputs free to be used as 25-MHz, 12.5-MHz, or 50-MHz outputs with or without skew.

Intel takes care to match the impedance at the connector of the module for all clock lines to be the specified 55Ω ± 10%. This means the clock driver chip has to match 55Ω at the connector and have a skew of less than 700 ps. The CY7B991 is suited for both cases. Its outputs are able to drive 50Ω terminated transmission lines and its maximum skew between zero-skew outputs is 700 ps.

MCLK0..2 are the memory bus clocks. MCLK is an input to the module. All memory bus signals and data are sampled on the rising edge of MCLK. In this example, since the MBC is interfacing to a 25-MHz memory bus, MCLK is 25 MHz and is synchronized with the 50-MHz CLK0..3 signals. These signals must also maintain low skew. The maximum skew allowed between MCLK0..2 is 1.0 ns. The CY7B991 meets this specification at 1.0 ns as t_{SKEW1}. Since the CY7B991 #1 in *Figure 10* is being used as a frequency doubler, the multiple output phases and associated crosstalk increase output-to-output skew from 0.7 ns (t_{SKEW0}) to 1.0 ns (t_{SKEW1}).

CY7B991 Configuration

In this example, one CY7B991 is using the frequency doubling function while the other CY7B991 is using the zero-skew output function. To make a double frequency output on CY7B991 #1 to be fed to CY7B991 #2, the output pin that will be used as feedback must be a ½-frequency output, done by tying its input select lines LOW (See *Figure 10*). These ½-frequency outputs will actually turn out to be 1x outputs (25 MHz) due to the internal PLL. This effectively makes all 1x selected outputs operate as double frequency outputs (50 MHz).

MCLK0..2 are the outputs 4Q1, 3Q1, and 3Q0 respectively. One pair of outputs is selected as double frequency by leaving its input select lines unconnected (M). This output (2Q0) may now feed a 50-MHz clock signal to the second CY7B991.

The second CY7B991 is using the zero-skew function. By leaving all 4Fx, 3Fx, and 2Fx input select lines unconnected (M), we select those outputs to be zero-skew outputs. Their skew will be less than 0.7 ns. These outputs will be CLK0..3 and PCLK0..1.

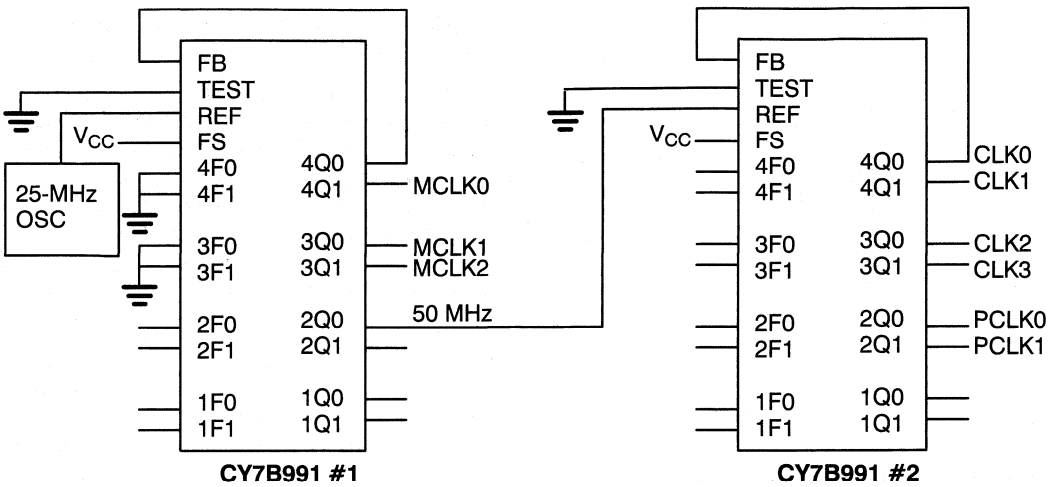


Figure 10. 50-MHz 486DX Cache Module Clock Generation

MIPS R3000 Design Example

The MIPS R3000 is used as the core CPU engine in many of today's high-end systems. This high-performance RISC machine operates at speeds from 16 to 40 MHz. With this high performance, however, comes the challenges of designing a system that utilizes many high-performance techniques usually reserved for high-end mainframes.

The R3000 employs a five-stage pipeline supported by a memory bus with a bandwidth of 320 MB/s. In order to supply the CPU with instructions and data without the latency associated with main memory, a direct-mapped cache is used with R3000 systems.

Figure 11 shows the cache subsystem. This system implementation uses two off-chip direct-mapped caches. These two caches, the instruction cache (ICache) and the data cache (DCache), are accessed alternately during each CPU cycle. During the first phase of the CPU clock, the address is provided to the ICache while the data is being read from the previously accessed DCache. During the second phase, while the ICache is providing an instruction to the CPU, the address is provided to the DCache. External transparent latches are used to capture the alternating addresses to each cache.

These caches can range in size from 4 to 256 KB. Data stored in the cache is comprised of 60 bits: 32 bits of data, 4 bits of data parity, 20 bits of tag, and a valid bit. Shown here are two 64-KB caches made up of two banks of 15 Cypress CY7B166 16K x 4 cache RAMS. In order to insure data consistency, all data that is written into the cache is also written into main memory through a write buffer. The R3000 system typically consists of a read/write buffer for main-memory accesses.

Four two-phase clocks control the timing of the buses and cache control signals. These clocks operate at twice the rate of the SysOut clock that is output by the processor. These clocks, shown in *Figure 12*, have delays with respect with one another, which maximize performance for a given set of static RAM parameters.

Clk2xSys determines the position of SysOut. The correct positioning of Clk2xSys will prevent ICache-to-DCache contention on back-to-back reads and system-bus-to-cache-bus contention at the end of read stalls.

Clk2xSmp determines the point at which the processor samples data, terminates cache write strobes, and terminates the address latch clocks (IClk and DClk).

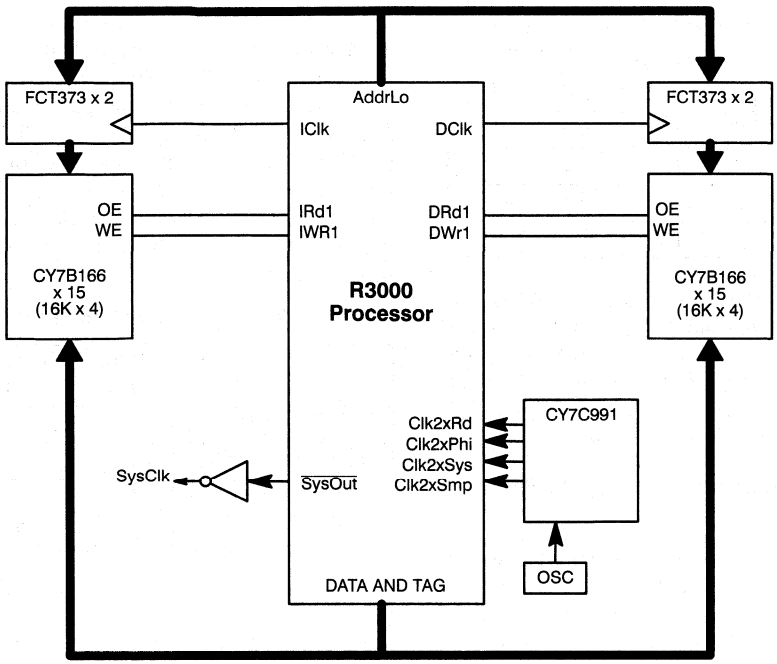


Figure 11. Block Diagram of Cache Subsystem with CY7B991, Cypress CY7B166, and the R3000

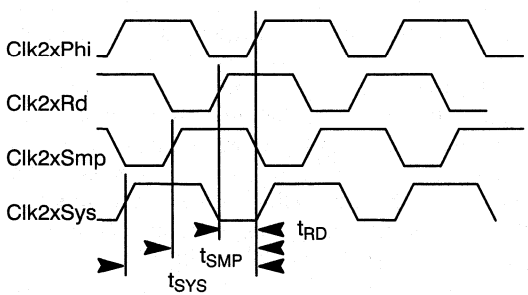


Figure 12. 2x Input Clocks

Clk2xRd begins the cache and read buffer read strobes, terminates the drive of the data and tag buses, and initiates the address latch clocks. The proper positioning of Clk2xRd will guarantee adequate data set-up to sample.

Clk2xPhi initiates the drive of the major processor outputs including address, data, and tag. All of the other 2x clocks are positioned relative to Clk2xPhi.

Many equations based on parameters such as the cache RAM hold time, '373 latch propagation delay time, and the processor data valid time determine the relative positioning between these 2x clocks. *Table 6* gives the generally accepted approximate values for the positioning of these clocks. For more information on how these values were obtained, see the references listed at the end of this application note.

Table 6. Relative Positioning of 2x Clocks

Parameter	16 MHz	20 MHz	25 MHz	33 MHz	40 MHz
t_{PHI}	0	0	0	0	0
t_{RD}	10	8	6	4.5	4
t_{SMP}	10	8	6	4.5	4
t_{SYS}	16	14	12	9	8

These timing differences between the various 2x clocks have to be maintained very accurately in order to insure correct functionality of the memory accesses. In the following discussion we will look at how to configure the CY7B991 to accurately control the 2x clocks in a 40-MHz R3000 system.

CY7B991 Configuration for a 40-MHz R3000 System

Figure 13 shows how RoboClock can be connected to achieve the phase delays listed in the last column of *Table 6*. This configuration uses an 80-MHz oscillator to produce the 2x clock phase adjustments..

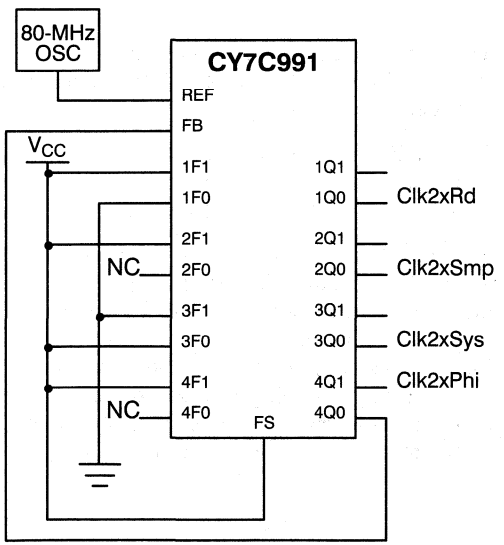


Figure 13. CY7B991 Configuration for a 80-MHz R300 System

If the CY7B991 runs at 80 MHz, according to *Table 2*, the programmable skew time unit (t_U) will be .781 ns. We will adjust Clk2xSmp to precede Clk2xPhi by $5t_U$ (3.91 ns), Clk2xRd to precede Clk2xPhi by $4t_U$ (3.12 ns), and Clk2xSys to precede Clk2xPhi by $10t_U$ (7.81 ns). *Table 7* lists the settings of these clocks with respect to Clk2xPhi. It also lists the minimum and maximum delay of these clocks based on the t_{UE} parameter explained above and given as 700 ps in the datasheet.

Table 7. Delay Setting for 40-MHz R3000

	# of Taps	Min. (ns)	Typical (ns)	Max. (ns)
Clk2xPhi	0		0	
Clk2xRd	$-4t_U$	−2.42	−3.12	−3.82
Clk2xSmp	$-5t_U$	−3.21	−3.91	−4.61
Clk2xSys	$-10t_U$	−7.11	−7.81	−8.51

Notice that these values are slightly different than those in *Table 6*. This is because these delay settings were optimized for the system assumptions mentioned above and analyzed with the equations given in the next section.

We can configure the CY7B991 in several ways to achieve this functionality. *Figure 13* shows one configuration. By connecting a 4Qx output to the FB input, we can achieve a greater range of programmable delays. This is evident from the entries in *Table 5*. If we wish to adjust an output so that it precedes the REF input by 10 time units, we can connect 4F1 to V_{CC} and leave 4F0 unconnected. The available output possibilities using a 4Qx output as the FB input and connecting its function select inputs are shown in the eighth row in *Table 5*.

Since we need Clk2xSys to precede Clk2xPhi by 10 time units, we can use a 3Qx output for this clock and connect 3F1 to ground and connect 3F0 to V_{CC}. In order to have Clk2xSmp precede Clk2xPhi by 5 time units, we can use either the 1Qx outputs or the 2Qx outputs for these signals and connect their function select inputs such that xF1 is left unconnected and xF0 is tied to V_{CC}. To achieve an output that precedes the REF input by 4 time units we could, again, use either the 1Qx or 2Qx outputs and connect xF1 to V_{CC} and xF0 to ground. 1Q0 is used to provide

the R3000 with Clk2xRd and 2Q0 is used to provide the R3000 with Clk2xSmp in this example. Finally, Clk2xPhi is connected to the 4Q1 output. As stated previously, the output that is used as the feedback to the internal PLL of the CY7B991 will be precisely aligned with the REF input and, therefore, it will be a logical choice for the Clk2xPhi output.

You have many choices when using the CY7B991 in this example. Either the 1Qx or 2Qx outputs could have been used for the Clk2xSmp and Clk2xRd. Also, any output could have been used for the FB input and accomplish the same results. These choices could be based on routing convenience or additional required functionality.

Derating Calculations

To prove that these delay selections will allow this memory subsystem to operate correctly we must insure that various parameters of the cache memories, latches, and other system components are met. These timing equations take into account the effect of capacitive loading on the various signal lines. To begin with, we must first calculate the additional time that a signal takes to become valid at its destination. In other words, we must calculate the output deratings to insure that the memory subsystem operates correctly under all conditions.

To illustrate why we need to be concerned about the additional time that it takes for an output to become valid, we will calculate the derating of the address bus signal lines. *Figure 14* shows a surface mount layout of a R3000 dual processor board. We will use this figure along with its associated distance measurements to calculate the address valid time derating.

We will make the following assumptions:

- We are using surface-mount packages.
- The trace has a capacitance of 2 pF/inch.
- The signal propagation delay is 2 ns/foot in copper.
- The outputs of the R3000 are specified with a loading of 25 pF. For every 25 pF of loading beyond this, a 1-ns delay is incurred.
- The distance between the R3000 and the latches is approximately 1 inch.
- The distance between the R3000 and the SRAMs is approximately 4 inches.
- Each address line is connected to 5 latches:
  - — 1 to the main memory interface buffer
  - — 2 to the instruction cache and tag
  - — 2 to the data cache and tag.

Deratings are caused by two factors. The first is derating due to capacitive loading on the signal lines beyond the tested capacitance of the output driver, and the second is the derating due to the time that

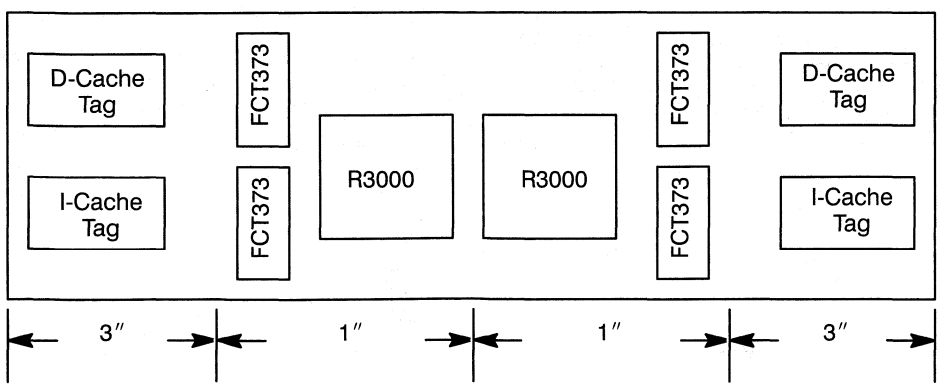


Figure 14. Surface Mount Layout of R3000 Board

it takes for the signal to travel down the trace that connects its output to its associated input. The following steps estimate the output deratings of the address lines due to each of these two factors:

- Total CPU to latch trace length = 4 inches
- Capacitance of trace = 8 pF
- Input capacitance of latch = 10 pF
- Total latch capacitance = 50 pF
- Total capacitive load = 58 pF
- Delay due to capacitance = (58−25)/25 = 1.32 ns
- Delay due to signal propagation delay = 2 ns/foot*4 inches = 0.67 ns
- Total signal delay ≈ 2 ns

In other words we must add 2 ns to the time that it takes for these signals to become valid. If we were to go through the calculations we would see that all of the control, data, and address lines should be derated by approximately 2 ns to insure reliable board operation. We can further assume that this derating is split almost equally between capacitance and trace delay. See Reference 1 for more information on these derating calculations.

Critical SRAM Timing Calculations

This section illustrates how the above phase delay selections will insure the proper operation of the R3000 memory subsystem. Several equations are presented to show the critical timing parameters of the SRAMs used as both the ICache and DCache.

These parameters are shown in the timing diagram (*Figure 15*) of the R3000 performing a store followed by a load. This is the worst-case example for SRAM timing parameters. This diagram shows selected signals from this subsystem. The SysClk* signal is an output of the R3000 based on Clk2xSys; the SysClk signal is the buffered version of this signal, which is used by various other components in the R3000 system, including the write/read buffer to the main memory.

SysPhi is a fictitious output that is shown to give a timing reference to various signals. The IRd and DWr signals are the instruction cache read signal and data cache write signals, respectively. The vertical lines shown in this diagram indicate the location of the internal 2x clocks that are generated by the CY7B991. Since $t_{smp} \approx t_{rd}$, a single vertical line represents these two clock positions. The numbers in the diagram represent the equation numbers that will be explained next.

The following equations assume that we are using the R3000A operating at 40 MHz, Cypress CY7B166 RAMS with timing characteristics listed in *Table 8,* and 74FCT373 latches with characteristics shown in *Table 9*. A superscript "D" indicates that this parameter references the derating on this signal line. The maximum derating is equal to 2 ns. The minimum derating is assumed to be half of this amount because the derating due to capacitive loading can be eliminated if the output driver is strong enough to drive a capacitive load beyond that for which it was tested or if the stated capacitive load of the latches, for example, is not as high as that specified. The derating due to trace propagation delay cannot be avoided.

In the equations that follow, the parameters and values in bold indicate the CY7B991 parameters that are responsible for insuring proper memory system operation.

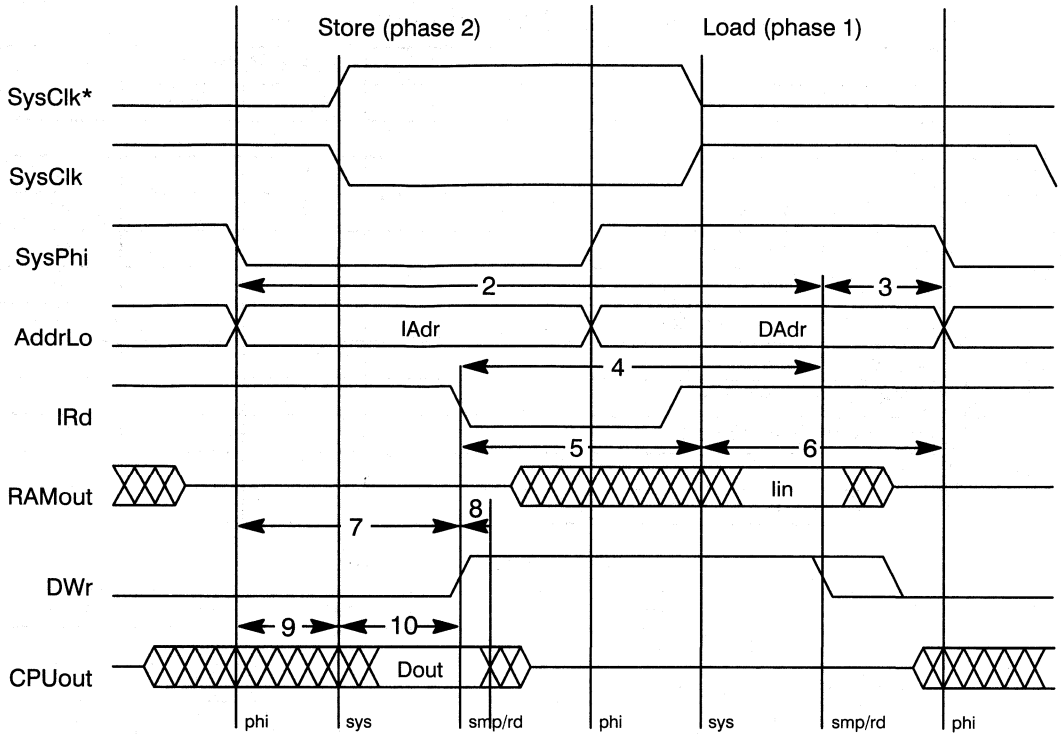


Figure 15. Cache Timing for 40-MHz R3000 Store Followed by Load

Table 8. CY7B166−8 Switching Characteristics

		CY7B166−8		
Symbol	**Description**	**Min.**	**Max.**	**Unit**
READ CYCLE				
t_{RC}	Read Cycle Time	8		ns
t_{AA}	Address to Data Valid		8	ns
t_{OHA}	Output Hold from Addr Change	2.5		ns
t_{ACE}	$\overline{CE}$ LOW to Data Valid		8	ns
t_{DOE}	$\overline{OE}$ LOW to Data Valid		4.2	ns
t_{LZOE}	$\overline{OE}$ LOW to Low Z	1.5		ns
t_{HZOE}	$\overline{OE}$ HIGH to High Z		4	ns
t_{LZCE}	$\overline{CE}$ LOW to Low Z	2		ns
t_{HZCE}	$\overline{CE}$ HIGH to High Z		4	ns

Table 8. CY7B166–8 Switching Characteristics (continued)

		CY7B166–8		
Symbol	**Description**	**Min.**	**Max.**	**Unit**
WRITE CYCLE				
t_{WC}	Write Cycle Time	8		ns
t_{SCE}	$\overline{CE}$ LOW to Write End	7		ns
t_{AW}	Address Set-Up to Write End	7		ns
t_{HA}	Address Hold from Write End	0		ns
t_{SA}	Address Set-Up to Write Start	0		ns
t_{PWE}	$\overline{WE}$ Pulse Width	6.5		ns
t_{SD}	Data Set-Up to Write End	4		ns
t_{HD}	Data Hold from Write End	0		ns
t_{LZWE}	$\overline{WE}$ HIGH to Low Z	2		ns
t_{HZWE}	$\overline{WE}$ LOW to High Z		4	ns

Table 9. FCT373 Switching Characteristics

Symbol	Description	Min.	Max.	Unit
t_{PLH}, t_{PHL}	Propagation Delay Dn to On	1.5	4.2	ns
t_{PLH}, t_{PHL}	Propagation Delay LE to On	2.0	5.5	ns
t_{PZH}, t_{PZL}	Output Enable Time	1.5	5.5	ns
t_{PHZ}, t_{PLZ}	Output Disable Time	1.5	5.0	ns
t_{SU}	Set-up Time HIGH or LOW Dn to LE	2.0		ns
t_H	Hold Time HIGH or LOW Dn to LE	1.5		ns
t_W	LE Pulse Width HIGH	5.0		ns

Internal Sample to Phase Delay

This parameter is used to insure that the internal sample to phase delay is met. This is given as a data-sheet parameter.

$$t_{SMP} \geq t_{SMP(min.)}$$
$$t_{SMP} \geq \mathbf{3.2} \qquad \text{Eq. 2}$$

When we set t_{SMP} with the CY7B991, we selected a $5t_U$ tap for a 3.91-ns variance between Clk2xSmp and Clk2xPhi. From the CY7B991 datasheet, the window within which this output edge will appear is 700 ps around the edge selected with the function select inputs. This gives a positioning of approximately 3.21 to 4.61 ns. Using the minimum delay of 3.21 ns obviously meets the requirement of Equation 2.

Address Access to Sample

This equations guarantees that the SRAMs have sufficient access time before the processor needs the data.

$$t_{RAMAA} \leq t_{CYC} - t_{SMP} - t_{DS} - t_{373PD} - t_{ADRIO}{}^{D} - t_{RAMAA}{}^{D}$$
$$t_{RAMAA} \leq 25 - \mathbf{4.6} - 4 - 4.2 - 2 - 2$$
$$= 8.2 \geq 8.0 = t_{AA} \qquad \text{Eq. 3}$$

In this equation we used the maximum t_{SMP} positions (4.6 ns) to produce the worst-case address access time. The t_{AA} of the CY7B166 is less than the 8.2 ns allowed with this choice of t_{SMP}.

Cache Enable to Sample

This parameter insures that the cache is enabled soon enough to meet the processor's input set-up time.

$$t_{RAMDOE} \le t_{CYC/2} - t_{RD}{}^{D} - t_{DS} - \mathbf{t_{RD-SMP}} - t_{RAMDOE}{}^{D}$$
$$t_{RAMDOE} \le 12.5 - 2 - 4 - (-.1) - 2$$
$$= 4.6 \ge 4.2 = t_{DOE} \qquad \text{Eq. 4}$$

The typical values of t_{RD} and t_{SMP} are given as 3.1 and 3.9 respectively. Subtracting these values gives us −800 ps. We then add an additional 700 ps (t_{UE}) as the worst-case possibility. In other words, the worst case for these two signals is when they occur as close together as possible. Only one t_{UE} was added to this number instead of two because two outputs of the same class of outputs that are skewed will be affected in the same way by factors influencing programmable edge placement. In this equation the difference between t_{SMP} and t_{RD} is given as 100 ps.

Minimum Read Pulse Width

This constraint guarantees that cache output enable time will be less than or equal to the read pulse generated by the processor.

$$t_{RAMDOE} \le t_{CYC/2} - \mathbf{t_{SYS-RD}} - t_{RAMDOE}{}^{D}$$
$$t_{RAMDOE} \le 12.5 - 5.4 - 2$$
$$= 5.1 \ge 4.2 = t_{DOE} \qquad \text{Eq. 5}$$

Here we used the maximum difference between t_{SYS} and t_{RD}, which is 5.4 ns.

Read-Write ICache Data Bus Contention

This parameter insures that the cache output disable time to high-impedance state (t_{HZOE}) is less than the data enable time of the processor during a store.

$$t_{RAMHZOE} \le t_{SYS} - t_{RD}{}^{D} + t_{DEN}$$
$$t_{RAMHZOE} \le 7.1 - 2 + (-1)$$
$$= 4.1 \ge 4 = t_{HZOE} \qquad \text{Eq. 6}$$

Here we used 7.1 (10*.781 − .700) as the minimum placement of t_{SYS} as the worst-case timing parameter. t_{DEN} is the data enable time of the CPU data on the data bus. It is specified as −1.0 ns.

Processor Data Set-up to Write End

This constraint insures that the SRAMs have sufficient data set-up time during processor stores.

$$t_{RAMDS} \le t_{CYC/2} - t_{SYS-SMP} - t_{DVAL} - t_{DVAL}{}^{D} - t_{WR}{}^{D}$$
$$t_{RAMDS} \le 12.5 - 4.6 - 1.5 - 2 + 1$$
$$= 5.4 \ge t_{DS} = 4.0 \qquad \text{Eq. 7}$$

Again the maximum Clk2xSys to Clk2xSmp was used as the worst case in this equation. t_{DVAL} is the amount of time that it takes for data to become valid on the data bus from the R3000.

Data Hold from Write End

This constraint guarantees that the cache has sufficient data hold time when either the processor or data buffer is writing to them.

$$t_{RAMHD} \le t_{SMP-RD} + t_{RAMLZ}$$
$$t_{RAMHD} \le .1 + 1.5 = 1.6 \ge t_{HD} = 0 \qquad \text{Eq. 8}$$

Here we used the minimum t_{SMP} to t_{RD} skew as the worst-case calculation.

Data Set-up to SysClk

This constraint guarantees that the set-up time into an external register is met. The external register is clocked with SysClk, which is simply the $\overline{\text{SysOut}}$ signal buffered by a 74FCT240. The minimum propagation delay for a FCT240 in the following equations is 1.5 ns.

$$t_{SETUPSYS} \le t_{CYC/2} - \mathbf{t_{SYS}} - t_{DVAL} - t_{DVAL}{}^{D} - \mathbf{t_{SYS}}^{D} + t_{240PD(min.)}$$
$$t_{SETUPSYS} \le 12.5 - 8.5 - 1.5 - 2 + 1 + 1.5$$
$$= 3 \qquad \text{Eq. 9}$$

Data Hold from SysClk

This constraint guarantees that the data is held at an external register on a processor store.

$$t_{HOLDSYS} \le t_{SYS-RD} - \mathbf{t_{SYS}}^{D} - t_{240PD(max.)} + t_{RAMLZ} + t_{RD}{}^{D}$$
$$t_{HOLDSYS} \le 4.0 - 2 - 4.3 + 1.5 + 1 = .2 \qquad \text{Eq. 10}$$

Address Set-up to End of Write

(*Equations 11* through *13* are not shown in *Figure 15.* These equations, however, are important to insure that the entire subsystem operates properly.)

The t_{RAMAW} parameter insures that the cache RAMs have sufficient data set-up before write end on a store operation.

$$t_{RAMAW} \le t_{CYC} - t_{SMP-SYS} - t_{ADRIO}{}^{D} - t_{373PD} - t_{WR}{}^{D}$$
$$t_{RAMAW} \le 25 - 4.6 - 2 - 5.5 + 1$$
$$= 13.9 \ge t_{AW} = 7 \qquad \text{Eq. 11}$$

Minimum Write Pulse Width

The constraint insures that the minimum write pulse width is not violated.

$$t_{RAMPWE} \le t_{CYC/2} - t_{WRDLY}$$
$$t_{RAMPWE} \le 12.5 - 1.5 = 11 \ge t_{PWE} \qquad \text{Eq. 12}$$

Read-Read, ICache DCache Contention

This parameter insures that there is no contention between successive reads between the instruction cache and the data cache. $t_{RAMEDMIS}$ stands for the difference between the RAMs enable time and its disable time.

$$t_{RAMEDMIS} \le t_{SYS-RD} + t_{IRD}{}^{D} - t_{DRD}{}^{D}$$
$$t_{RAMEDMIS} \le 4 + 1 - 2$$
$$= 3 \ge t_{HZOE} - t_{LZOE} = 2.5 \qquad \text{Eq. 13}$$

Here we assume that the RAMs are controlled by the OE pin to give this worst-case number.

As you can see from the previous analysis, the CY7B991 makes an ideal clock driver for this 40-MHz R3000 design.

Conclusion

This application note provided information on how to design a 50-MHz 486DX Cache–Module and a 40-MHz R3000 system using the CY7B991 clock driver from Cypress Semiconductor. The extremely low-skew abilities of its outputs provide precise clock distribution. The programmable delay functionality provides compensation for set-up and hold time differences as well as needed sampling delays, as shown in the R3000 application. In addition, the programmable function selects combined with the phase-locked-loop architecture provide a clock doubling ability for increased performance, as shown in the 486 application.

The integration of these features on a single device as well as the CY7B991's high output drive capability, low part-to-part skew, wide operating frequency range, and low input-to-output propagation delay gives an overall performance and flexibility that is not available with any other clock distribution IC.

References

1. Simha, Satyanarayana, (IDT Inc.), "Application Note AN–85: SRAM Timing Parameters for 40-MHz R3000A Cache Design," IDT Databook Update 1, 1991, pp Update 1 C: 66 – 70.
2. Integrated Device Technologies, RISC Databook, 1991.
3. LSI Logic, LSI Logic LR3000 Processor Interface Manual, October 27, 1988.
4. Ma, Taufik, Silas, Isic, "Application Note AP–458: Designing a Memory Bus Controller for a 50-MHz Intel486DX CPU–Cache System," Microprocessors Volume 1, November 1991, pp 2–709 to 2–761.
5. Jolly, Rich, "Application Note AP–453: Clock Design in 50-MHz Intel486 Systems", Microprocessors Volume 1, October 1991, pp 2–672 to 2–689.
6. Intel, "Intel486 DX CPU–Cache Module," Microprocessors Volume 1, November 1991, pp 2–438 to 2–474.

Document #: LOG0009–0592

CY7B991 and CY7B992 (RoboClock) Test Mode

This application note discusses the Test mode capabilities of the CY7B991 and CY7B992 (RoboClock) devices. It begins with an introduction to these devices and then discusses how to use the Test mode features.

Introduction

The RoboClock family consists of two parts: the CY7B991 and CY7B992. The CY7B991 has TTL (0 to 3V) outputs and the CY7B992 has CMOS (0 to Vcc) outputs. Each device will drive 50Ω terminated transmission lines. *Figure 1* shows the PLCC and LCC pin configurations for these devices.

RoboClock (*Figure 2*) employs a phase-locked-loop architecture. Connecting an output to the FB (feedback) input of the device causes the PLL to synchronize and align this output both in phase and in frequency with the REF (reference) input. This results in very low input to output delay and allows a system to connect RoboClocks in parallel for clock distribution while maintaining very low skew between various clocks from different devices.

RoboClock contains eight outputs grouped in four sets of two. Two function select lines (xF0, xF1) control the functionality of each pair of outputs (xQ0, xQ1). The outputs of an output pair operate identically.

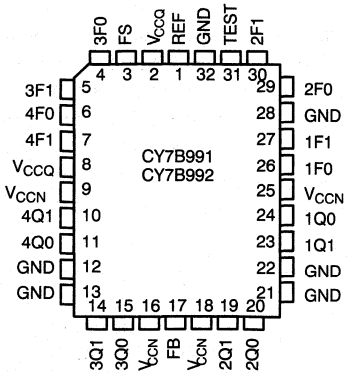


Figure 1. PLCC and LCC Pin Configuration

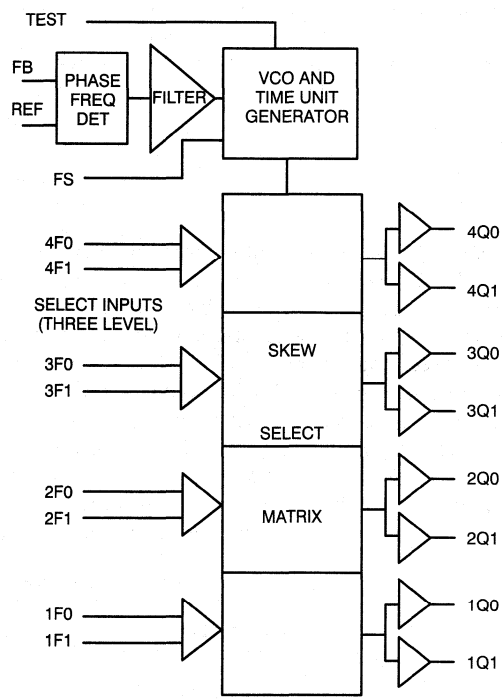


Figure 2. RoboClock Block Diagram

Table 1. Programmable Skew Configurations

Function Selects		Output Functions		
1F1, 2F1, 3F1, 4F1	**1F0, 2F0, 3F0, 4F0**	**1Q0, 1Q1, 2Q0, 2Q1**	**3Q0, 3Q1**	**4Q0, 4Q1**
LOW	LOW	$-4t_U$	Divide by 2	Divide by 2
LOW	MID	$-3t_U$	$-6t_U$	$-6t_U$
LOW	HIGH	$-2t_U$	$-4t_U$	$-4t_U$
MID	LOW	$-1t_U$	$-2t_U$	$-2t_U$
MID	MID	$0t_U$	$0t_U$	$0t_U$
MID	HIGH	$+1t_U$	$+2t_U$	$+2t_U$
HIGH	LOW	$+2t_U$	$+4t_U$	$+4t_U$
HIGH	MID	$+3t_U$	$+6t_U$	$+6t_U$
HIGH	HIGH	$+4t_U$	Divide by 4	Inverted

Each pair of three-level function select inputs allows you to hardwire the operation of each output pair to one of nine delay or functional configurations. Each function select input pin can be connected to V_{CC} (HIGH), left unconnected (MID), or connected to ground (LOW). *Table 1* shows the programmable skew configurations available on each output pair. The function select configurations in *Table 1* assume that the output connected to FB is set for "zero" skew.

Table 1 shows the range of t_U over which an output may be skewed with respect to the REF input. t_U is a function of the frequency at which the 1Q0 output is operating. RoboClock offers frequency coverage with three ranges from 15 MHz to 80 MHz with the use of the three-level FS (frequency select) input. *Table 2* shows the operating frequency range for each of the three levels of FS. The appropriate FS level selection must be made such that the anticipated operating frequency of the 1Q0 output is within the specified limits. There may be two acceptable levels for the FS pin when operating at certain frequencies. The appropriate connection of the FS pin, in this case, would be based on the value of the time unit, t_U, required for the application. *2* also shows an equation that can be used to calculate t_U as well as the approximate operating frequency where t_U is equal to 1 ns.

For example, according to *2*, a system using RoboClock with a clock speed of 33 MHz would leave the FS pin unconnected. The programmable time unit, t_U, based on this operating frequency, would be

$$t_U = \frac{1}{f_{1Q0} \times N} = \frac{1}{33\ \text{MHz} \times 26} = 1.17\ \text{ns} \qquad \text{Eq 1}$$

In other words, you can adjust the position with which the rising and falling edges of the outputs move with respect to the corresponding REF input edge with a resolution of 1.17 ns when operating the device at 33 MHz. At 25 MHz the t_U could be either .91 ns or 1.54 ns depending on whether the FS pin is tied LOW or left unconnected, respectively.

Table 2. Frequency Range Select and t_U Calculation

	f_{1Q0} (MHz)		$t_U = \frac{1}{f_{1Q0} \times N}$	
FS	**Min.**	**Max.**	**where N =**	**Approximate Frequency At Which t_U = 1.0 ns**
LOW	15	30	44	22.7 MHz
MID	25	50	26	38.5 MHz
HIGH	40	80	16	62.5 MHz

Test Mode Features

In some situations you may need to stop the PLL of the device. For instance, in many board-level testing applications you may need to supply a clock input to the system that may not meet the REF input requirements of RoboClock. This scenario can occur in bed-of-nails testing or single-step microprocessor execution. Use of the TEST input of RoboClock will allow operation in single-step mode.

The TEST input is a three-level input. In normal system operation, this pin is connected to ground, allowing RoboClock to operate as previously explained. (For testing purposes, any of the three-level inputs can have a removable jumper to ground, or be tied LOW through a 100Ω resistor. This will allow an external tester to change the state of these pins.)

If the TEST input is forced to its mid or HIGH state, the device will operate with its internal phase-locked-loop disconnected. The TEST input must be forced to less than 1V to insure its LOW level, to $V_{CC}/\pm 500$ mV to insure its MID level, and to $V_{CC} - 1V$ to insure its HIGH level.

When RoboClock is put in Test mode, after a few REF cycles, input levels supplied to REF will appear at all outputs after a 15- to 80-ns delay. The circuit effectively becomes a long chain of delay elements. The level on the TEST input affects the length of time it takes for the REF signal to propagate through each delay element. When the TEST input is forced HIGH, each delay element will be selected to have its shortest delay (< 700 ps). This is known as "contracted" mode. When the TEST input is forced to its mid state, the delay through each element will be as long as possible (> 1.5 ns). This is referred to as "extended" mode.

The level placed on the FS pin also determines the operation of RoboClock when it is in Test mode. The FS input is used to control the number of delay elements that the REF input will propagate through, as shown in *Figure 3*. When FS is held HIGH, REF will pass through only the last 13 delay stages. When FS is placed in the MID or LOW position, REF will propagate through all 22 delay elements.

In contrast with normal operation (TEST tied LOW), FB will not have any affect on the operation of the outputs. All outputs will function based only on the connection of their own function select inputs (xF0 and xF1) and the waveform characteristics of the REF input.

Outputs that have the divide-by-two output configuration selected will change state at every second REF input, and outputs that have the divide-by-four

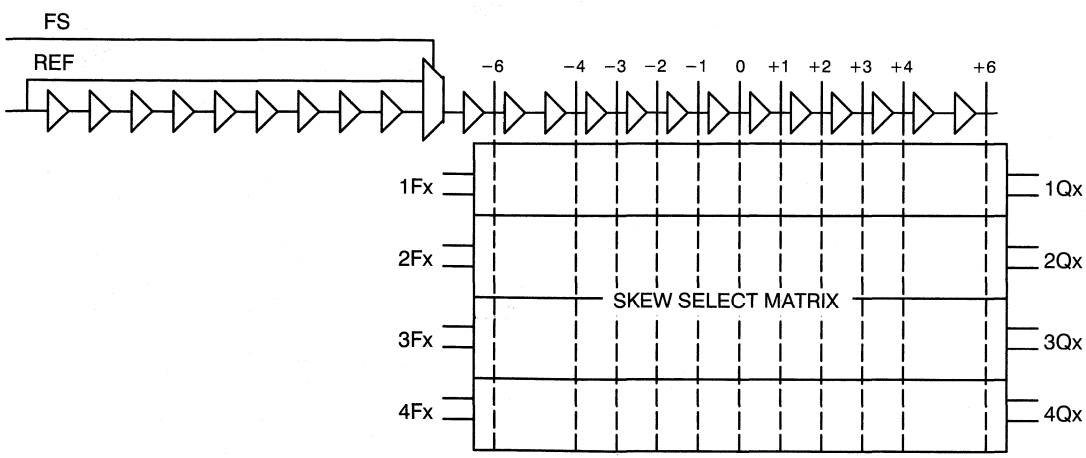


Figure 3. RoboClock Test Mode

option selected will change state at every fourth REF input. An output selected for inverted operation will drive the opposite sense of the REF input.

A counter reset is available for the divided outputs. To reset the counters, the 3F0 and 4F0 function selects must be placed in their MID position and a clock applied to the REF input. If the 3Qx or 4Qx outputs are then selected for a divided function (3Fx = LOW, LOW, HIGH, HIGH or 4Fx = LOW, LOW) then the 4Qx or 3Qx outputs will be in their HIGH state. The first REF clock will cause these outputs selected for divided operation to transition LOW and, subsequent REF clocks will cause these outputs to continue their normal output divided output pattern.

Conclusion

RoboClock's Test mode feature stops the phase-locked-loop allowing board-level testing and evaluation. This mode allows operation at frequencies below the minimum operating frequency. It also provides the ability to apply input pulses with varying width and period to the device without requiring the cycle-to-cycle frequency accuracy necessary to keep the feedback loop in lock.

Document #: LOG0010–1192

Crystal Oscillator Topics

Parallel vs. Serial Resonant Oscillator Circuits

All frequency synthesis components from IC DESIGNS require a reference frequency, usually provided by a quartz crystal. The on-board crystal oscillators are parallel-resonant designs. Although most oscillator circuits are designed to use parallel-resonant crystals, most inexpensive off-the-shelf crystals are series-resonant devices. Accordingly, many oscillator applications (such as CPU clock/timer designs) often run about 0.05% too fast.

When a crystal is manufactured, the manufacturing specifications must specify the type of oscillator circuit in which it is intended to be used. The two types will result in a difference in the output frequency of approximately 500 ppm (0.05%). To guarantee accurate output, the specification must therefore take this difference into account when the crystal is manufactured.

For maximum accuracy, a parallel-resonant crystal of the proper specification will be required. Whether or not the extra cost is justified depends on the application. In many cases, an error of 0.05% is acceptable; in applications such as time-keeping, however, it is often not.

Specifying a Custom Parallel-Resonant Crystal

To order a specially made crystal, the correct specification to give the vendor is parallel-resonant, plus a value for the load capacitance presented by the oscillator. As an example, the correct specification crystal oscillator would read as follows:

Type = Parallel-resonant
Freq = 14.31818 MHz
C_{load} = 17 pF

As an alternative to a custom crystal, parallel-resonant crystals with a specified C_{load} = 20 pF may be available from dealer stock. Such a crystal may be a viable alternative to a custom crystal, offering only slightly less accuracy at lower unit cost.

Accuracy Considerations for Time-Keeping in PC Design

Time-keeping is a stringent application, where a 50 ppm error in reference frequency produces a real-time clocking error of 2 minutes per month. In many personal computer applications, both 32.768 KHz and 14.31818 MHz are used for time-keeping. Highly accurate values of both these frequencies are therefore required.

Most IC DESIGNS frequency synthesis components produce a 14.31818-MHz signal on one or more output pins. However, as previously pointed out, the accuracy of these outputs will depend upon proper specifications of the 14.31818-MHz reference crystal by the designer. Correctly specifying the reference crystal is therefore important in time-keeping applications.

Some IC Designs components also offer an additional output from an on-board 32.768-KHz oscillator. This oscillator is designed to work with an industry-standard 32.768-KHz crystal specified as parallel-resonant, C_{load} = 12 pF and is readily available from dealer stock, so the resulting output should always be accurate enough for time-keeping applications.

Selecting a High-Accuracy Reference Crystal

Background

Our monolithic oscillator products deliver a wide range of output frequencies, replacing many individual quartz oscillators. And yet, one quartz crystal is needed to set the reference frequency, usually 14.31818 MHz. In this section, the emphasis is on maximizing the accuracy of the reference. Specific part numbers will be named, and a worst-case analysis performed.

Changing Requirements

IC DESIGNS' earliest design-ins did not specifically recommend a parallel-resonant crystal instead of the more common (but less suitable) series-resonant type. Such designs tended to have error rates of 500–600 ppm.

This was acceptable at the time because VGA monitors can tolerate a pixel frequency deviation of 1% from nominal (10,000 ppm), so reference accuracy was not a major factor.

Today, however, IC DESIGNS' expanding product line targets many time-critical applications where the reference frequency must be held within ±100 ppm over the operating temperature range.

Crystal Specifics

Earlier in this note, we recommended specifying a parallel-resonant crystal, with load capacitance (C_{load}) of 17–20 pF. Here are some specific part numbers for surface-mount devices:

- **Fox Crystal #FPX143–20** – With C_{load} = 20 pF, it is close enough to our 17 pF specification to be only 35 ppm high. If a tighter initial accuracy is desired, one can add two external load capacitors to ground, or specify a different load capacitance. Telephone: (813) 693–0099.
- **Epson America #MA-506 14.313M-B** – This sells off-the-shelf with C_{load} = 16 pF. In our lab, we find them to be nominally on target; they are the best off-the-shelf match available. Telephone: (310) 782–0770.

Drive Level Precautions

Any quartz crystal is subject to mechanical failure if oscillated at an excessive voltage level. For a CMOS amplifier, the drive level is around 700 µW. However, the parts are tested at 100 µW; fortunately, a maximum rating exists of 1000–2000 µW.

Do not use any crystal that does not have a 1000 µW rating. It may drift excessively, or even become a delayed failure mechanism.

Error Budget Analysis

We can guarantee ±100 ppm worst case with the Epson #MA-506 14.318M-B. This is important for products like the ICD2028. To verify this, one simply adds up the relevant sources of error in an error budget (this is true for any crystal).

Source of Error	Amount
Crystal frequency tolerance	±50 ppm
Crystal temperature tolerance	±30 ppm
ICD2028 frequency tolerance	±05 ppm
ICD2028 temperature tolerance	±06 ppm
ICD2028 supply tolerance	±03 ppm
Total	±94 ppm

One can also look at the well respected "Monte Carlo Analysis," which says that, if a number of uncorrelated variables are changing randomly, it is not reasonable to add up the individual worst-case figures to come up with an aggregate worst-case value.

Instead, if one has n variables, X1, X2,...Xn, all varying randomly and independently, then the overall variation is:

$$X_{total} = \sqrt{X_2^1 + X_2^2 + \ldots X_n^2}$$

According to this calculation, the design will remain within 59 ppm, which is much lower than the simple addition of the uncorrelated variables.

Document #: ICD0002–0394

ECL Outputs

Introduction

The IC DESIGNS oscillator family features ECL-compatible outputs in products such as the ICD2062. These outputs allow clocking at frequencies above 160 MHz, with all the inherent advantages of differential ECL signal transmission.

This application note covers the principal advantages of using ELC outputs and makes recommendations concerning layout and wiring methods for parts such as the ICD2062.

Power Supplies (PECL vs. ECL)

The ECL V_{DD} and V_{EE} pins have traditionally been powered from a −5.2V supply, V_{DD} being grounded and V_{EE} set at −5.2V—the intent is to achieve the lowest V_{DD} noise by grounding the V_{DD} pins. In more recent designs, however, ECL is often used with +5.0V instead of −5.2V. (V_{DD} set to + 5.0V and V_{EE} tied to ground.) Since V_{DD} noise is not a major concern, this permits the use of a standard logic supply. This application note will focus on +5.0V ECL designs (sometimes called PECL).

ECL Advantages

As clock speeds rise beyond 100 MHz, the advantages of using ECL become more obvious. Most of these advantages involve the use of differential signal transmission.

Differential signals are less susceptible to ground noise problems, as all noise becomes common-mode. Single-ended CMOS is much more susceptible, since ground bounce and other noise affect logic thresholds, degrading noise immunity. ECL signals remain unaffected, since noise rides on both signals as an average level. Logic levels are also less critical, since the threshold is the differential cross point, which can tolerate significant signal attenuation. Differential circuits also tend to generate less noise in the power supply.

ECL is designed with termination resistors that allow high-frequency signals to propagate with minimal overshoot and reflection.

These advantages are most pronounced in a bipolar implementation, but many of the same benefits can be realized in CMOS designs.

Pad Structure

Referring to *Figure 1*, transistors Q1 and Q2 form differential ECL output drivers. Unlike single-ended outputs (see *Figure 2*), N-type transistors are not required, since termination resistors are always present and serve as pull-downs.

The ECL output drive logic guarantees that when Q1 switches ON, Q2 switches OFF (and vice versa). A complementary logic state is always maintained, assuring a constant current supply draw in either output state.

Logic Levels

The V_{OH}/V_{OL} logic levels are approximately 4.1V to 3.2V. This gives a differential signal of 0.9 V. This is more than adequate, since bipolar ECL V_{OH}/V_{OL} are typically 4.1V to 3.3V for a differential swing of 0.8V. If the termination resistors are reduced from the 220Ω/330Ω suggested value, the V_{OH} level can be reduced somewhat.

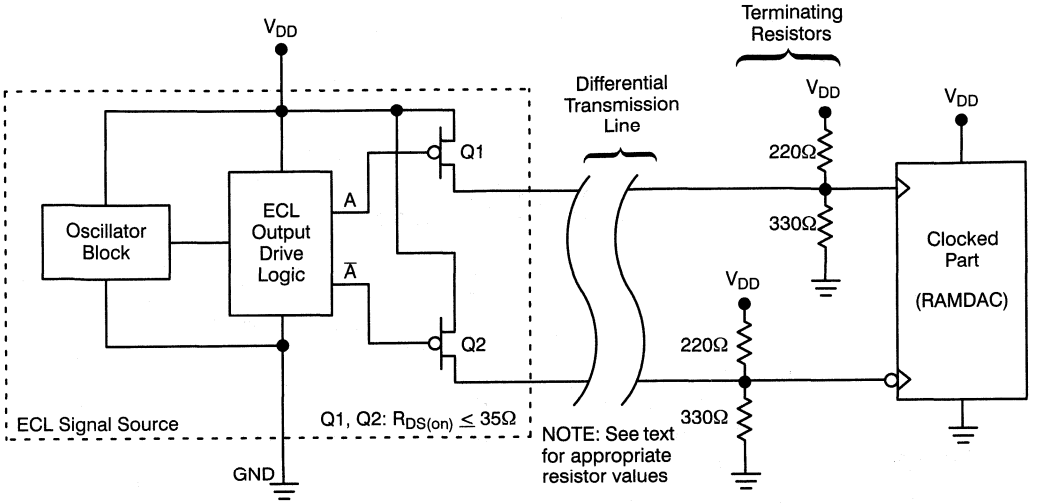


Figure 1. Differential ECL Output Driver

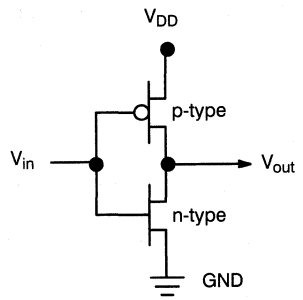


Figure 2. Single-Ended Output Driver

Output Routing and Board Layout Issues

ECL signals maintain their integrity over long routing distances. A 100-MHz single-ended trace should be limited to a few inches, but ECL can travel several feet at that frequency. This is due to ECL's termination resistors at the receiving devices.

For good signal integrity, ECL traces are laid out in pairs. By controlling the board layout and specifying appropriate electrical properties, a constant characteristic impedance of 100 – 150 Ω can be achieved. By employing transmission live techniques high quality signals are ensured.

To reduce V_{DD} noise the clock generator must be properly bypassed at the supply pins, as should the parts to be clocked. Terminating resistors should also be bypassed separately if they are not located near a bypass capacitor.

Terminating Resistor Values

The 220Ω/330Ω values represent a trade-off between low-power draw and best high-frequency performance. For frequencies above 80 MHz, they may pull down too weakly, resulting in inadequate signal swing. These values can go as low as 68Ω/100Ω at 160 MHz (always maintaining an approximate 2:3 ratio). Low values work satisfactorily at both low and high frequencies, the only drawback being a higher current drain.

Summary

A general overview of ECL logic has been presented, with emphasis on the interface of ECL logic to +5V powered CMOS clock generators. Should additional support be required, contact IC DESIGNS for assistance.

Document #: ICD0003–0394

Bus Products – 7



Bus Products Section Contents and Abstracts

Features of the VIC068A VMEbus Interface Controller 7–1

This application note gives a broad overview of the VIC068A. It outlines some of the major features of the device including: master write posting, slave write posting, read-modify-write cycles, block transfer cycles, interprocessor communication facilities, and interrupt handling.

Interfacing the VIC068A to the MC68020 7–6

This application note explains some of the implementation details of interfacing the VIC068A to a Motorola MC68020 microprocessor. Emphasis is given for A24/D16 type designs. Resetting the VIC068A is given much attention in this application note. A ROM remapping circuit is described showing how the MC68020 obtains its stack pointer and program counter at reset. A sample schematic shows how to interface the VIC068A to the MC68020. PLD equations are given which provide the address decoding. Finally, master and slave cycles are described showing how all these pieces are used together to provide a full function interface.

Using VIC068A on a Board Without a Microprocessor 7–13

This application note describes one technique on how to use the VIC068A on a board without a microprocessor. In a typical VIC068A design, a microprocessor is available to program the VIC068A's internal registers. In many slave-only applications, however, no microprocessor is available. This application note describes how to enable the VIC068A registers to be programmed from another VMEbus master as opposed to having local state machines perform this function.

Connecting the VIC068/VAC068 to the TI TMS320C40: A Prototype Design 7–31

This application note provides high-level as well as low-level details of interfacing VIC/VAC to TMS320C40. This allows for techniques to be implemented to minimize design time for subsequent efforts since this design has not been optimized for either size or speed. The Design Requisites section provides the design goals established prior to design as well as relevant background regarding devices involved. Hardware details, including schematics and programmable logic source code, represent the central focus of the paper. In addition, software initialization of the chip, set by the TMS320C40, is covered. Throughout this note, it is assumed that the reader is familiar with the TMS320C40 architecture, the basics of the VIC068A/VAC068A, and the VMEbus and its protocol(s).

Software Considerations for the VIC64 7–69

This application note provides a VIC64 (or VIC068A) designer with proven tips and examples for both configuring and operating the VIC64 or VIC068A. The software described was based on an actual VIC64 design. This application note also describes configuring the CY7C964 address comparator functions. Sample C source files are also described (the actual source files are available on the Cypress BBS) showing a Block Transfer utility.

VIC64 to Motorola 68040 Interface 7–84

This application note shows how the VIC64 can be interfaced to a Motorola 68040 microprocessor operating at 40 MHz. The issues and assumptions that go into designing such an interface are considerable and complex; thus, this application note will not attempt to design a complete VME board that can do everything. It will

cover some of the issues that are pertinent when designing a 68040-based VMEbus board and will focus on the circuitry required for VIC64 to 68040 interfacing.

Interfacing the CY7C611A with the VIC64 ..... **7–125**

This application note describes an interface between the CY7C611A SPARC microprocessor and the VIC64. The interface described within this application note couples the synchronous bus of the CY7C611A to the asynchronous bus of the VIC64. The interface is high performance and preserves many of the features necessary for VMEbus applications, such as the memory exception facility.

The application note discusses the high and low level implementation of a the interface. A CY7C361 and 22V10 PLDs implement the design. State diagrams and timing waveforms are included. The PLD source files for the design are available on the Cypress Semiconductor BBS.

Using the CY7C964 with VIC **7–138**

This application note introduces the CY7C964. CY7C964 operating modes and features are described. Also discussed is the ease of use with either the VIC64 or VIC068A. A sample circuit schematic is included showing the CY7C964 to VIC interface.

Information is provided on the different signals present on the CY7C964 and the potential problems that could be encountered when using the device. This application note compliments the information provided within the VIC64/CY7C964 Design Notes.

Features of the VIC068A VMEbus Interface Controller

This application note describes some features of the Cypress VIC068A and provides information on how to use the device.

The VIC068A was designed by a consortium of VMEbus manufacturers in partnership with Cypress. The major goals of this consortium were to achieve a standardized, reasonably priced VMEbus interface that was not dominated by any board manufacturer. Manufacturing this specialized chip requires a high-speed process (125 MHz) and high-power I/O pins (64 and 48 mA).

The VIC068A adheres to the ANSI/IEEE Standard 1014, which minimizes the problems of interfacing among the VMEbus boards of various manufacturers. A block diagram detailing the device's functional blocks appears in *Figure 1*.

VIC068A Highlights

With very precise timing, based on a 64-MHz clock that is used internally to make decisions on 8-ns intervals, you can reach the theoretical limits of the VMEbus transfer rates—a block transfer rate of 40 Mbytes/s.

Because all logic resides in a single chip, the VIC068A greatly reduces the board space necessary to interface to the VMEbus. Even a highly sophisticated interface with an A32/D32 system controller and block transfer support requires no more than 60 square cm or 20 percent of a double eurocard (6U card).

Special care has been taken to speed up the VIC068A's VMEbus access. Although many of today's CPU boards use megabytes of high-speed local RAM to limit the number of VMEbus accesses, the accesses that do occur for I/O or data reads and writes must be done efficiently to avoid slowing the rest of the system.

For both types of data transfers, the VIC068A offers special support. For single-write cycles, you can program the VIC068A to operate in the so-called master or slave write-posting mode (*Figure 2*), the local VMEbus write cycle is terminated locally as soon as data is latched in the VMEbus latches. This allows the local CPU to continue with instruction fetches or other operations while the VIC068A transfers data over the VMEbus.

In slave write-posting mode (*Figure 3*), the same function happens with write cycles form the VMEbus to the local bus. As soon as the data is latched, the VMEbus cycle is terminated and the local cycle can finish independently of further VMEbus traffic. Both modes reduce CPU overhead and VMEbus utilization, providing higher bandwidth in single-cycle writes.

The VMEbus prohibits a similar function in single-cycle reads because every read cycle on the VMEbus could turn out to be a read-modify-write (RMW) cycle. This cannot be foreseen because the only difference is that address strobe is held low between the two cycles. Therefore, if the VMEbus address strobe were released during the two portions of the same RMW cycle, another VMEbus master could break into that cycle and modify the same data.

To move blocks of data over the VMEbus, the VIC068A uses the block transfer mode. In its standard form, this mode allows a processor to transfer up to 256 bytes with just one starting address sup-

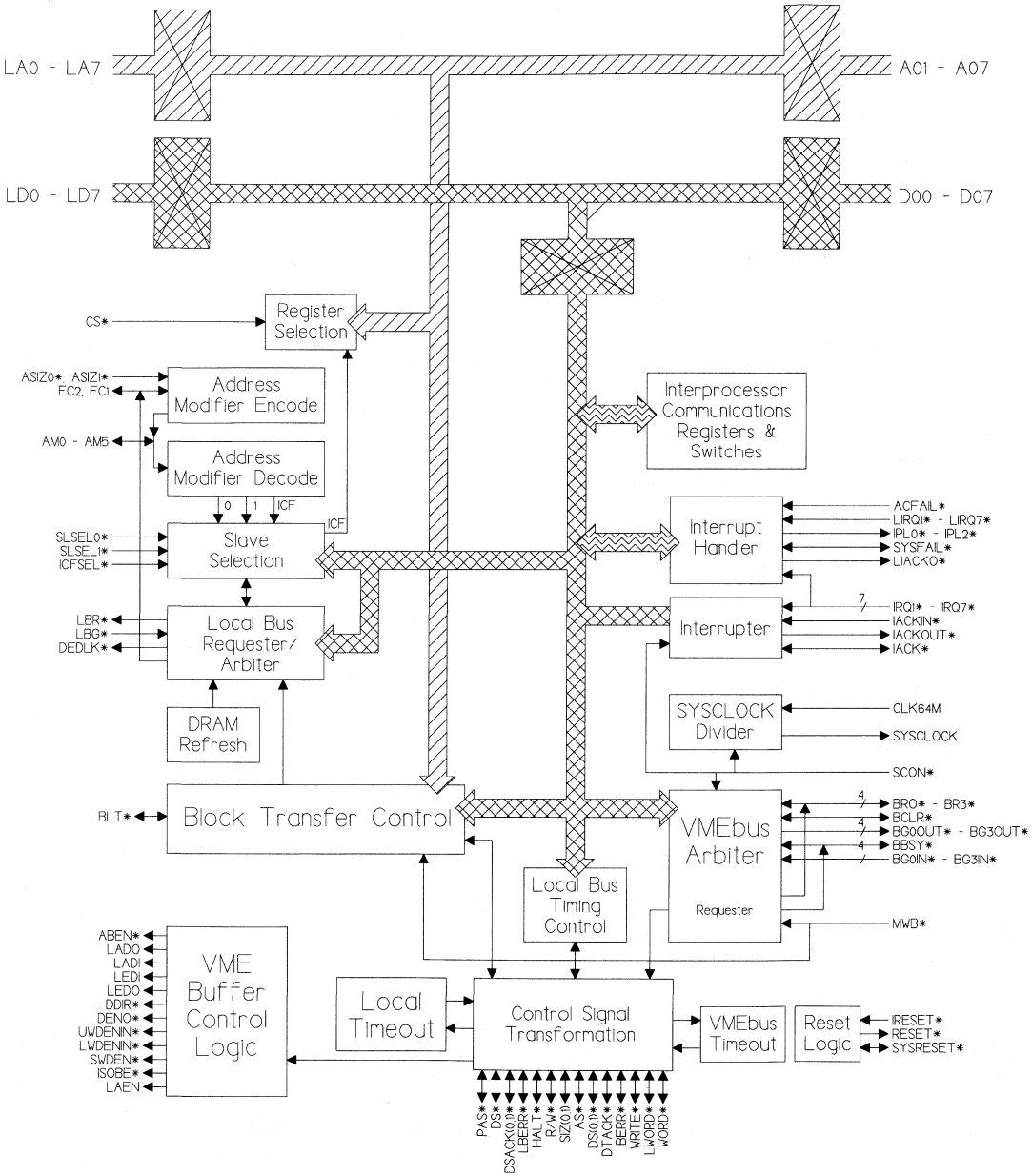


Figure 1. VIC068A Functional Block Diagram

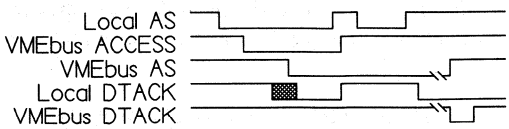


Figure 2. Master Write Posting

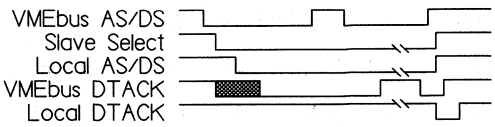


Figure 3. Slave Write Posting

plied to the VMEbus. Additionally, the VIC068A uses a type of pipelining to accelerate VMEbus throughput. On a block transfer read cycle, the slave VIC068A automatically prefetches the n + 1 data byte during the same read. The nth data byte is transferred across the VMEBus, and the n − 1 byte is latched in local RAM. As shown in *Figure 4*, this operation uses all three buses in overlapped and parallel operation to speed up the transfer. Write transfers use the same mechanism.

The limiting factor on the VMEbus transfer rate is either the VMEbus's many timing restrictions or the source or destination memories. If the memory consists of dynamic RAM, the restriction is probably the cycle time of the chips used, often as slow as 200 ns. To overcome this limitation, the VIC068A offers a programmable access mode so that attached DRAM can be used in page mode.

After a starting row address cycle (RAS), all subsequent cycles need only a column address (CAS) to reduce the access time, often by as much as half. For a slave interface, the VIC068A contains all the necessary counters and timing elements for local AS, DS, and address generation.

A master block transfer needs two or three additional latches for the higher address lines during the local DMA part of the block transfer. Thus, even with low-cost DRAMs, the VIC068's block transfer rate can reach 40 Mbytes/s, limited only by the VMEbus specification and the physical characteristics of the VMEbus.

This transfer rate decreases the time needed to load programs or move data to graphics boards, as well as increasing the VMEbus's bandwidth, thereby allowing more CPUs to work together in a multiprocessor system.

Mailbox Signaling

To add greater capability to multiprocessor systems, the VIC068A has four interprocessor communication global switches (ICGS) and four interprocessor communication module switches (ICMS). These are all byte-wide mailbox registers that generate a

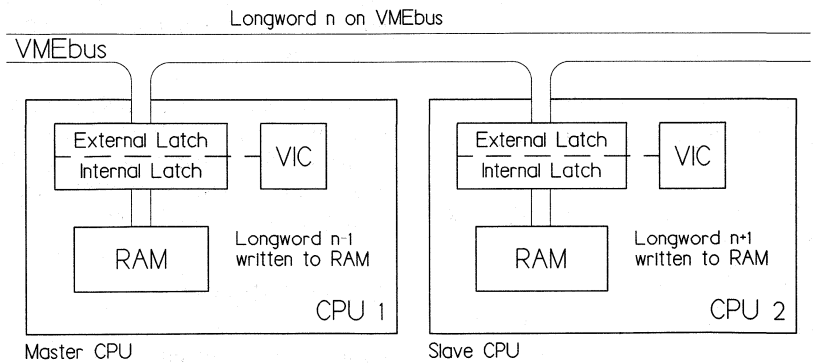


Figure 4. Block Transfer Read Cycle

local interrupt when accessed from the VMEbus. The ICGSs of one group reside at the same address and are accessed with a write cycle, which behaves as a broadcast to all members of the group. Because the ICMSs are at different addresses, one dedicated processor can be activated with a local interrupt request (LIRQ).

A processor can inform a logical group of processors about a new task via a broadcast using the ICGSs and can then communicate with single processors about the task using the ICMSs.

Eight-byte-wide interprocessor communication registers (ICR) are also available. Five of these registers serve as general-purpose read/write registers, and three are dedicated to control local activities (Halt, Reset, Mask ID, etc.). The ICRs can be read and written from the local side or the VMEbus without interfering with each other.

Interrupt Generator

The VIC068A handles up to seven simultaneous pending IRQs with separate vectors. The VIC068A also provides independent local IRQ vectors, if external IRQs are served.

Miscellaneous Features

The VIC068A furnishes several features for VMEbus support:

- SYSFAIL generation
- Software reset
- ACFAIL
- BERR register for detailed information

For local support, the VIC068A provides these features:

- Seven local IRQ sources, all level, polarity, edge and vector programmable
- Local bus timeout (2–512 ms)
- With /without VMEbus request time included
- 21 different local IRQ vectors
- VIC ID register

In addition to the VIC068, the following parts or equivalents are required for a minimum hardware interface:

- Three address latches and drivers (74xx543)
- Three data latches and drivers (74xx543)
- Four isolation buffers (74xx245)

You might also need the following:

- One to two PLDs for slave address decoding
- Two to three latches for master block transfer
- ½ PLD for block transfer glue logic

Interfacing

To connect a processor other than the 680x0 to the VIC068A, it is often easiest to map the processor control signals into the control signals available on a 680x0-type of processor. This type of transition interface offers the advantage of compatibility with a large family of 680x0-compatible peripheral parts, which you can then use elsewhere in the design.

Figure 5 shows a sample interface, whose four address latches store the multiplexed Mbus of the MC88000 processor. Four data latches store the data bytes after the acknowledge of the 680x0 bus and then start calculating parity the processor's MBus. The reason for this approach lies in some older peripheral I/O chips, which change their data lines when they should remain stable (i.e., transmit data buffer empty, etc.).

Two other data latches emulate the MC68020's dynamic bus sizing. The last buffer, between D0 – D7 of the 680x0 bus and AD16 – AD23 of the MBus, emulates the 680x0 bus's IRQ cycles with normal read cycles of the MC88000.

Acknowledgment

Cypress Semiconductor wishes to thank JÜrgen Bullacher of Eltec GMbH and Eltec International S.A.R.L. for submitting this article.

Document #: BUS0003−1092

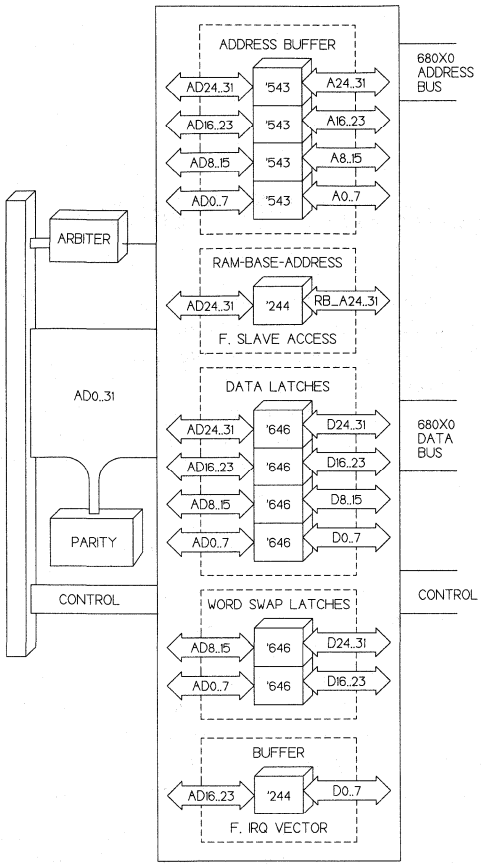


Figure 5. Sample Interface

Interfacing the VIC068A to the MC68020

This application note explains some of the features of the Cypress VIC068A and provides the first-time VIC068A user with simple implementations of these features. The VIC068A offers the most highly integrated VMEbus interface available today. It reduces the number of parts needed and saves board space. The emphasis in this application note is on interfacing the VIC068A as VMEbus A24/A16 D16/D08(E0) master/slave to the Motorola 68020.

Reset Operation

The VIC068A performs three distinct reset operations:

- Internal reset, activated by the IRESET pin, which initializes most of the internal registers
- System reset, essentially the same as IRESET, but is activated by writing ($F0) to the system reset register, or by asserting IRESET when the VIC068A is the VMEbus controller (SCON pin asserted)
- Global reset initializes all the VIC068A registers

After a reset, the 680X0 processor reads its initial stack pointer (SSP) and program counter (PC) from addresses $0 through $7. One way to handle this is to remap the boot-up ROMs to the low addresses for the first few cycles of the processor.

Figure 1 shows a circuit you can use to do this. The circuit uses a serial-in/parallel-out shift register (the 74HC164) to generate the MAP signal. This active-Low signal can be used with address-decode logic to force boot ROM access to the lower addresses during initial power up. Asserting the 74HC164 CLEAR pin drives all the parallel outputs Low, which asserts the selected MAP signal. With the two serial inputs tied High, each Low-to-High transition of the 68020 AS clocks the High through the shift register and out each of the parallel outputs. By picking the proper output for the MAP signal, you can decode from 1 to 8 of the initial processor cycles. You can use the MAP signals on memory configurations that are 8, 16, or 32 bits wide by using the QH, QD, or QB outputs, respectively.

Using the Processor RESET Instruction

The OR gate in *Figure 1* ensures that the 74HC164 is cleared only when HALT and RESET are both asserted. This allows the use of the 68020 RESET instruction without inadvertently reasserting MAP. An alternative to this approach is to use two small-signal diodes (1N4148) and a pull-down resistor in place of the OR gate. This change reduces the design's parts count by eliminating the 74HC32.

A ROM remapping circuit must be used whether the RESET instruction is issued or not because of the way the VIC068A arbitrates local bus contention between the 68020 and the VMEbus. Contention occurs when both master and slave operations are requested concurrently (MWB asserted and SLSEL0, SLSEL1, or IFCSEL asserted). The VIC068A indicates this contention by asserting DEDLK. You can deal with the condition by setting bit 4 of the VIC068's interface configuration register ($AF) to assert HALT along with LBERR when DEDLK occurs (68020 bus retry sequence). The VIC06 then waits for the 68020 to deassert the MWB input. Once this happens, the VIC068A releases LBERR but continues to assert HALT to keep the 68020 off the local bus. The VIC068A then allows the slave operation to complete and deasserts HALT. The 68020 can now retry the contested bus cycle.

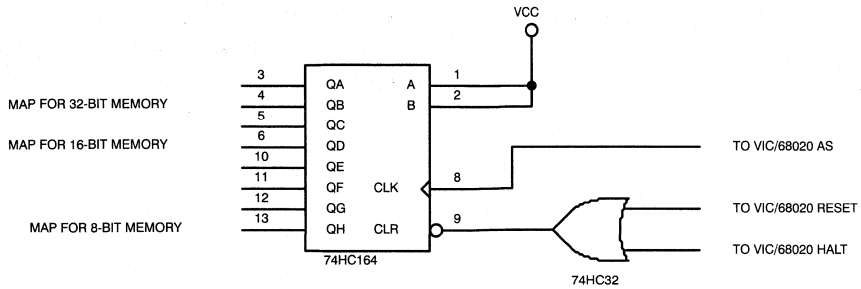


Figure 1. ROM Remapping Circuit

Internal Reset

At first glance, the IRESET might seem the logical choice for implementing the power-on reset. Because the IRESET input has some built-in hysteresis, a simple RC circuit would be appropriate for applying the power-on signal.

IRESET does not initialize the local bus timing register nor any of the slave select registers, however. Additionally, the VIC068A powers-up with the DRAM refresh option enabled (bit 4 of the arbiter/requester configuration register $B3 High). This condition is acceptable if you are using DRAM but adversely affects the external reset circuit in *Figure 1*. Specifically, for the first DRAM refresh cycle, the VIC068A deasserts RESET but maintains HALT in the active (Low) state and toggles AS. This action causes shift operations in the 74HC164. You can activate DRAM refresh after reset by writing a 1 to bit 4 of the arbiter/requester configuration register ($B3).

System Reset

The assertion of SYSRESET on the VMEbus typically activates system reset, but only when a global reset is not taking place. When the VIC068A is configured as the system controller (SCON pin asserted), it drives the SYSRESET pin for the required 200 ms during an internal or global reset.

Global Reset

The global reset is the most useful for power-up purposes because it places all the VIC068A registers in a known state. You initiate a global reset by asserting IPL(0) concurrent with or just after asserting IRESET. These reset signals should not be asserted until the V_{CC} power source has stabilized at 5 volts. Because IPL(0) is also one of the encoded interrupt lines for the 68020, you must assert this signal with an open-collector or three-state device.

In using global reset, bear in mind that when the VIC068A powers-up it ignores the VMEbus SYSRESET. The VIC068A releases HALT and RESET after the 200-ms time out even if the current VMEbus master asserts SYSRESET past this required minimum time. This automatic release is a useful feature because it eliminates reliance on the system controller to release SYSRESET to start the power-up sequence. Refer to the *VIC068A/VAC068A User's Guide* for more information on global reset.

The VIC068A generates a LBERR if you try to access the VMEbus or any of the VIC068A registers before SYSRESET is deasserted. One solution to this problem is to structure the software so that the VIC068A registers are set up as late as possible in the power-up sequence. You can also temporarily point the 68020 BERR exception vector to an address containing an RTE instruction and let the 68020 cycle in a BERR/RTE loop until SYSRESET is deasserted. The latter approach provides an op-

portunity to be the first board in a system to request VMEbus mastership.

Connecting the Bus Lines

Figure 2 shows the standard buffer configuration for an A24/D16 VMEbus connection. This design also supports A16 and D08(E0) operation.

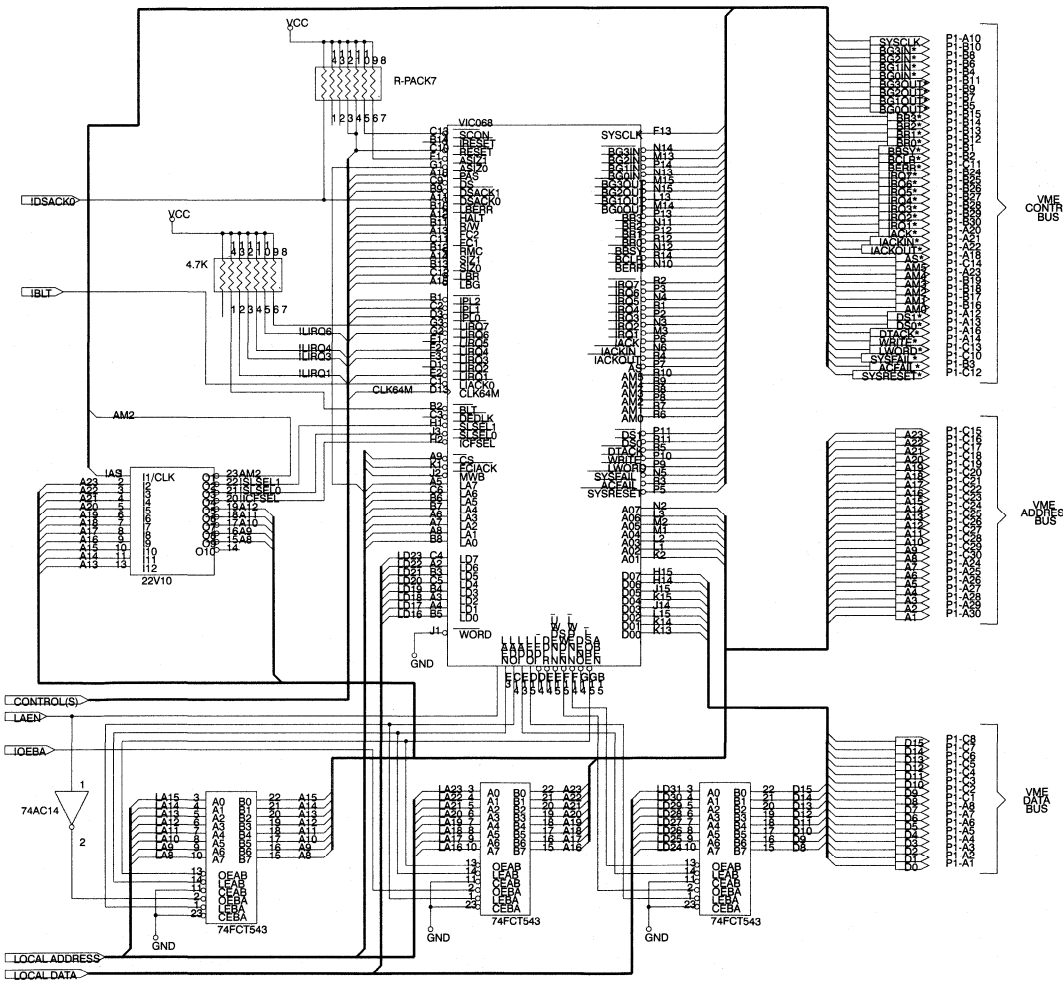


Figure 2. Address and Data Bus Connections

The D16/D08(E0) Data Bus

Connect the VIC068A to the 68020 as you would any 16-bit peripheral device. The 74FCT543 data buffer connects between the 68020 data bus's upper byte (D31 – 24) and the VMEbus D15 – 8 data lines. The lower byte (LD7 – LD0) is buffered through the VIC068A to the VMEbus low byte (D7 – D0). Several control signals connect directly from the VIC068A to the 74FCT543: DENO (data enable out) to OEAB (Output enable A-to-B), LWDENIN (lower word data enable) to OEBA (Output enable B-to-A), LEDO (latch enable data out) to LEAB (Latch enable A-to-B), and LEDI (latch enable data in) to LEBA (latch enable B-to-A).

The Address Bus

The A24/A16 configuration requires the use of two more 74FCT543 devices to buffer and control the VMEbus A23 through A8 signals. The 74FCT543 LEAB, LEBA, and OEBA inputs connect directly to the VIC068A LADO (latch address out control), LADI (latch address in control), and ABEN (enable address out control) outputs, respectively. The output of the VIC068A LAEN (local-address enable control) must be connected to the 74FCT543 OEBA input through an inverter because LAEN is an active-High output and OEBA is an active-Low input.

Connecting the DSACK Lines

During the normal local bus operation, the 68020's slave devices (i.e., memory, UART, parallel port) must tell the processor the size of their data bus. This is done by asserting the DSACK1 inputs, which tells the 68020 that the port is a 16-bit device. Asserting DSACK0 instead indicates that the port is an 8-bit device. Asserting both DSACK1 and DSACK0 indicates that the port is 32 bits wide. To configure the VIC068A as a 16-bit port, simply connect the 68020 DSACK1 to the VIC068A DSACK1.

So long as there you have no requirement for VMEbus access to 8-bit devices on the local bus, you do not need to do anything with the VIC068A DSACK0 pin except terminate it (pull it High).

When you do need to access 8-bit devices, a small problem arises with the way the VIC068A acknowledges register accesses and interrupt-acknowledge cycles. During these cycles, the VIC068A always asserts both DSACK1 and DSACK0, whether the WORD input is asserted or not. And in VMEbus master cycles, when talking to an 8-bit device on the VMEbus, the VIC068A responds with DASCK0 to acknowledge the 8-bit transfer completion.

The solution to the DSACK0 problem is simple but can be complicated to implement: You must break the DASCK0 connection between the VIC068A and the 68020 during interrupt acknowledge or VIC068A register access (CS) cycles. The circuit needed to do this is a bidirectional, open-collector buffer between the VIC068A and 68020. The buffer should be inactive in both directions only when the VIC068A FCIACK or CS inputs are asserted. In *Figure 3*'s PAL equations, the DSACK0_020 and VIC068A DSACK0 equation illustrates how to handle the DSACK0 connection.

Master Operation

VMEbus master operation with the VIC068A is easily accomplished with the use of the MWB (module-wants-bus) input. The VMEbus can be requested at any level (0 – 3). The VMEbus can also be dynamically changed via the arbiter/requester configuration register ($B3), which eliminates the need for hardware jumpers. All VMEbus release modes are supported through the release control register ($D3). Support for write posting means that the local processor can write to the VMEbus without having to wait for the current bus master to release the bus or for the arbitration logic to assert the correct BGIN 9 (bus grant in) line. The VIC068A takes cares of this overhead for the local processor, improving system throughput.

To request VMEbus mastership, the 68020 asserts the MWB input. You can think of MWB as a VMEbus chip select. When interfacing to the VMEbus as an A24 or A16 device, you can have access to the whole VMEbus address space by decoding a 32-Mbyte area of the 68020 address space for VMEbus operations. The ASIZ1-0 pins tell the VIC068A whether the current cycles represent an A32, A24,

or A16 operation. You can use the upper 16-Mbyte address space (A24 High) for VMEbus A23 operation and the lower half (A24 Low) for VMEbus A16 operation by following three steps: decode A31 through A25 to generate MWB, tie the ASIZ1 input High, and connect the 68020 A24 address line to the VIC068's ASIZ0 input. *Figure 3* demonstrates this way of decoding MWB.

When the VIC068A recognizes a valid slave access, the device asserts LBR (68020 BR input) and waits for LBG assertion (68020 BG output). Once the VIC068A receives LBG, the device becomes the local bus master at the conclusion of the current cycle and completes the requested VMEbus slave operation. If the VIC068A is the only DMA device on the local bus, there is no need to generate BGACK (bus grant acknowledge) for the 68020. But if any other devices are capable of local bus mastership, you have to provide the arbitration logic and the BGACK signal for the 68020. Keep in mind, too, that other DMA devices must be able to recognize and deal appropriately with the 68020 bus-cycle entry operation (BERR and HALT asserted).

Slave Operation

The VIC068A can provide full VMEbus slave operation by dual-porting local memory with little or no 68020 overhead. The normal slave access operation starts by providing SLSEL0 or SLSEL1 through VMEbus address decoding. The circuits in *Figures 2* and *3* use a 22V10 PAL for this purpose. Always qualify VMEbus address decoding with the AS and/or DS1-0.

Decoding SLSEL0, SLSEL1, and IFCSEL

Figure 3 illustrates a typical PAL specification that you can use to provide address decoding for SLSEL0, SLSEL1, and IFCSEL. The VIC068A uses all the address modifier lines (AM5 – 0) to qualify the access mode. Address decoding can ignore these inputs. The VIC068A then decides if the access mode is legal and completes the cycle or generates the VMEbus BERR signal, depending on the value programmed in the slave select registers. You can also qualify the select outputs with the address modifiers and let the initiating device time-out if the access is not legal.

The IFCSEL input gives the VMEbus access to some of the VIC068A control registers and the interprocessor communication registers. These registers are available only through an A16 privileged-mode access.

The PAL specification in *Figure 3* configures SLSEL0 to dual-port a 4-Kbyte (minus 256 bytes) space of local RAM as an A16 non-privileged access input and decodes IFCSEL in the SLSEL0 area's upper 256 bytes. You can use this 256-byte space for mailbox communication between boards in a multimaster system.

SLSEL1 is decoded as an A24 supervisory-only access and provides full dual-porting of the 68020 board's E2PROM program memory. This allows the VMEbus system controller to put the system in a reset and hold state by asserting bit 6 of the VIC068's interprocessor communications register 7. The VMEbus master can then reprogram the entire program memory space. Once that operation is complete, the controller can use the interprocessor communications register 7 to release the reset and hold state. The board comes up running the newly installed program.

Take care when decoding SLSEL0, SLSEL1, and IFCSEL. The VIC068's operation is undefined when more that one of these inputs is active simultaneously.

Decoding for Supervisor/User Mode

You can use the VMEbus AM2 signal to select user (AM2 Low) or supervisor (AM2 High) modes. The AM2 input is used as part of the slave-select decoding shown in *Figure 3*.

Dealing with A24 and A16 Slave Accesses

Regardless of the access address size, the 74FCT543 address buffer outputs are enabled. Typically, the backplane pulls unused VMEbus address lines High passively, but most masters drive these lines regardless of the bus-cycle-address size. If this is not desir-

```
module_CYCLE_DECODE;
   Cycle_decode device 'PV22V10';
   VCC,GND                              pin 24,12;

"inputs (15)
   A31,A30,A29,A28,A27,A26,A25,A19      pin 1,2,3,4,5,6,7,8;
   SLSEL1, SLSEL0                       pin 9,10;
   FC2,FC1,FC0,AS,LBG                   pin 13,14,15,16,17  "for FCIACK and VIC_Cycle output

"outputs (6)
   VIC_DSACK0,DSACK0_020    pin 18,19;    "To VIC DSACK0 and local system DASCK0
   VIC_CYCLE                pin 20;         "current bus cycle is VMEbus
   FCIACK                   pin 21;         "Interrupt Acknowledge Cycle
   PRE_MWB,MWB              pin 22,23;      "VIC module-wants-bus (with and without AS)

"output type declarations
   VIC_CYCLE,PRE_MWB,MWB          istype 'com';
   FCIACK,VIC_DSACK0,DSACK0_020   istype 'com';
   VIC_CYCLE.OE,FCIACK.OE         istype 'com';
   PRE_MWB.OE,MWB.OE              istype 'com';
   VIC_DSACK0.OE,DSACK0_020.OE    istype 'com';

equations in CYCLE_DECODE
"Enable ALL outputs except DSACK's
   VIC_CYCLE.OE     =1;
   PRE_MWB.OE       =1;
   MWB.OE           =1;
   FCIACK.OE        =1;

"This signal tells everybody that the VIC068A is controlling the current bus cycle
   !VIC_CYCLE<T>=!LBG & AS<T><T><T><T>"signal is asserted while AS is still high
           #!VIC_CYCLE & !LBG &!AS "maintain signal through entire cycle

"Interrupt acknowledge cycle (68020 to VIC).  Use VIC_CYCLE to insure this is not a VMEbus
master cycle
   !FCIACK = A31 & A30 & A29 & A28 & A27 & A26 & A25 & A19 & FC2 & FC1 & FC0 & !AS &
VIC_CYCLE;

"VME A24 access is at addresses $04000000 - $04FFFFFF.  A16 access is at addresses $0500000 -
$05FFFFFF (ASIZ0 is tied to LA24)
   !MWB = !A31 & !A30 & !A29 & !A28 & !A27 & A26 & !A25 & VIC_CYCLE &!(FC2 & FC1 & FC0);

"This is the same signal as MWB but the AS input is removed to provide an early VMEbus master
cycle indication input to other PLDS
   !PRE_MWB = !A31 & !A30 & !A29 & !A28 & !A27 & A26 & !A25 & VIC_CYCLE &!(FC2 & FC1 & FC0);

"This signal is connected directly to the VIC DSACK0.  It generates the VIC DSACK0 for VMEbus
slave accesses to 8 bit device
   !VIC_DSACK0 = !VIC_CYCLE & !DSACK0_020;

"This enables VIC_DSACK0 only when VIC is the local bus master (slave accesses)
   VIC_DSACK0.OE = !VIC_CYCLE & (!SLSEL0 # !SLSEL1);

"This signal is connected to the 68020 DSACK).  It generates the 68020 DSACK0 for VMEbus mas-
ter accesses to 8 bit devices
   !DSACK0_020 = !MWB & VIC_CYCLE & !VIC_DSACK0;

"This enables the 68020 DSACK0 only when the VIC is the VMEbus master
   DSACK0_020.020 = !MWB & VIC_CYCLE;
   end_CYCLE_DECODE
```

Figure 3. ABEL Equations for PALC22V10 Cycle Decoding

able, control the output-enable signals with the upper address line buffers using the VMEbus address modifiers. *Table 1* illustrates how to use AM5 and AM4 to determine the bus-cycle-address size.

You can derive individual enables for each of the VMEbus address latches by ANDing one or both of these address modifiers with the VIC068A LAEN (local-address enable) signal; modify both if operating in an A32 system.

Remember to provide a stable level for the local-address lines because nothing drives them during VMEbus accesses. You can ensure a stable level using 4.7Ω pull-up or pull-down resistors on the local-bus A31–A16 lines. The local-bus address buffers can be set to the desired address state and enabled with the same latch-enable signals.

Dual-Porting Local Memory

The PAL specification in *Figure 3* generates a signal called VIC_CYCLE than can serve as part of the local-address decoding to re-map local memory for dual-porting on the VMEbus. This approach allows memory placement at a VMEbus address independent of the local address.

Interrupts

The VIC068A interrupt structure is very versatile. One of the most useful features is the ability to redefine interrupt levels, and thus priorities, under normal program control. The VIC068A supports all seven levels of VMEbus interrupt as well as the seven local-interrupt levels. Interrupts are also available to notify the 68020 of VMEbus status and error conditions.

Figure 3 shows how to decode the 68020 interrupt acknowledge bus cycle to generate the VIC068A FCIACK input. You can omit A19–A16 from the equation if you do not use breakpoints, a memory management unit (MMU), or a coprocessor (68881/68882).

Using LIACK0

The LIACK0 output is typically connected to the 68020 AVEC input to initiate autovectoring of interrupts to which the VIC068A has not been programmed to respond. You can also use LIACK0 with the IPL(2–0) outputs to generate interrupt-acknowledge signals to other 680x0-compatible interrupting devices.

LIRQ7–1 Inputs

The LIRQ7–1 inputs are the interrupt request inputs to the VIC068. The control register for each input allows you to determine the input's polarity (high/low) and sensitivity (level or edge). The control register also allows you to define whether the VIC068A supplies the vector during interrupt acknowledge cycles or asserts LIACKO (local-interrupt acknowledge out), sets the level of interrupt the 68020 sees on IPL2–0, and enables or disables the interrupt. You do not need to terminate these inputs if you leave them unconnected, but you must pull them up externally if they are used.

The local interrupts (IPL2–0) are grouped and have a common vector base register ($57). This vector base is added to the encoded interrupt level programmed in each of the interrupt control registers to supply a unique vector to the 68020 for each interrupt input.

LIRQ2 is a special case because it can be used as an interrupt clock tick timer. You enable the timer through bits 2 and 3 of slave-select control register 0($C3). When enabled, LIRQ2 becomes the timer output, and the local-interrupt control register 2 ($2B) becomes the timer's interrupt-control register. The timer's periodic interrupt can be set to 50, 100, or 1000 Hz. If you plan on using the tick timer, do not connect the external interrupts to LIRQ2 because this pin becomes an output.

Table 1. Determining Bus-Cycle-Address Size

AM5	AM4	Cycle
H	H	A24 Access
H	L	A16 Access
L	L	A32 Access

Document #: BUS0004–1092

Using VIC068A on a Board Without a Microprocessor

This application note shows how the VIC068A, normally a peripheral device to a processor on a VMEbus board, can be used without a local processor. Essentially, the VIC068A will be used to build a high-performance VMEbus slave board. However, we will not go into the specifics of using the VIC068A's slave operations. This data is well covered in the VIC068A User's Guide. Instead, we will go into the details of the circuitry required to use the VIC068A without a local processor.

General Concept

The VIC068A normally resides on a bus along with a local processor such as a Motorola 680x0 family part when it is used on a VMEbus board. Both the VIC068A and the local processor can be masters of the local bus; however, the VIC068A requires a number of internal registers to be set up prior to operation. Once configured, the VIC068A can be a master or a slave to the VMEbus, and a master or a slave on the local bus, depending on the type of transfer that the VIC068A is called upon to perform.

Without the presence of a processor on the local bus, the VIC068A cannot be set up in the normal fashion. Obviously, there must be some method to communicate with the VIC068A's internal registers. Unfortunately, these registers that must be set up are not accessible from the VMEbus side of the VIC068A. Thus, another VMEbus master cannot access the VIC068A to set it up, because it cannot access the local bus.

So to use the VIC068A without a processor, we must find a way to access the local side of the VIC068A from the VMEbus, or to have the VIC068A "boot" itself via some type of state machine. The former technique is used in this application note for two reasons.

First, using a state machine on the local side is akin to having a processor there! A simple state machine with significant versatility could be built cheaply from a microcontroller. But our goal in this application note is to avoid using a processor.

Second, using a processor requires a software development effort, which is one of the things we wish to avoid. A VMEbus slave board should not be a configuration management nightmare. To avoid this, the design will only require software in the form of PLD code.

To access the VIC068A from the VMEbus, a slave module distinct from the VIC068A's slave capabilities is utilized. Thus, the board will in effect have two slave interfaces—one whose sole purpose is to access the local side of the VIC068A (this will be considered the "configuration slave" in future discussion), and the other being the VIC068A itself. The VIC068A will perform the "classical" slave interface, accessing on-board resources requested of a VMEbus master, such as memory, UARTs, registers, buffers, etc., under normal operation.

Capabilities of the VIC068A

As mentioned above, the VIC068A can be used as slave or a master on both the local bus and the VMEbus. After a minimal initialization from the VMEbus (which is the only time that the VIC068A

is a slave on the local bus), the VIC068A can be used in the two operations discussed in the following sections.

Slave Interface with Slave Write Posting

This is the obvious operation for the VIC068A to perform. A VMEbus master requests access to on-board peripheral devices and the VIC068A, as slave, performs the required sequencing to transfer data. The VIC068A acts as master on the local bus in this mode. To improve performance, write posting is enabled. Write posting allows a VMEbus master accessing a VIC068A slave to deposit (post) its data on the slave board and immediately get acknowledged by the VIC068A. The VIC068A then completes the transfer of the posted data while the master is freed up to do other tasks. The master is not burdened with the overhead of a slave device with a long time period between data strobes and DTACK*.

Slave Interface with Block Transfer

Even more performance can be obtained by accessing the slave board in block transfer mode. The VIC068A handles all the address sequencing tasks when used in this type of transfer, while also acting as master of the local bus. Depending on the speed of the on-board peripherals, the VIC068A can be set up to run in Turbo mode, thus allowing the fastest mode of data transfer across the VMEbus.

Why use the VIC068A?

An obvious question one might ask is, "Isn't the VIC068A wasted on this type of application? It can do so much, but so little is utilized. Isn't it easier to just build up the interface from discrete components?" The VIC068A has four advantages when used in this application: simplicity, sanity, space savings, and speed. It is simple because the VIC068A, along with a few additional buffers and PLDs, form an effective interface to the VMEbus. It will keep you sane because you will not have to read and try to understand the VMEbus specification (IEEE-1014). The specification is riddled with nuances that require careful thought in implementation. The VIC068A takes care of these for you. It also saves space. For the most part, developing a slave interface for the VMEbus is a relatively simple task (assuming you've read the specification five or six times!). But as soon as you start adding the features that the VIC068A handles, such as write posting and block transfers, the board area required to do the interface grows tremendously. Finally, it improves speed. The VIC068A handles all operations from one integrated circuit running at 64 MHz, which is designed for performance. All the timing required by the VMEbus has been accounted for and optimized to the greatest extent possible.

System Components

The VIC068A

The VIC068A is used in a typical fashion as shown in *Figure 1*. Yet, there are a few connections that differ from those shown in the VIC068A User's Guide. First and foremost, the design uses a 62.5-MHz oscillator rather than one running at 64 MHz so that a 125-MHz CY7C361 can be used in the design. The CY7C361 is covered in a subsequent section. Since the VIC068A in this application is not used as a master of the VMEbus, the BBUSY*, BCLR*, and BRx* signals are pulled High (inactive). It is assumed that the board being designed is not a system controller, so SCON* is pulled High (inactive). The board could easily perform system controller operations by connecting these lines and pulling SCON* Low. Since no processor resides on the local bus, both RMC* and MWB* can be pulled High (inactive). In addition, BLT*, HALT*, FCIACK*, IFSEL*, ASIZ0*, and ASIZ1* are pulled High (inactive) due to the absence of a local processor. One last item of note is that the WORD* signal is pulled High, which indicates that 32-bit transfers are allowed.

Standard Bus Interface Logic and Control PLD

The standard bus interface logic is part of *Figure 2*. 74FCT543s and 74FCT245s are utilized as shown in the *VIC068A/VAC068A User's Guide*, but with a slight modification. Because two slave interfaces exist in the design, the control signals LWDENIN*, LAEN, DENO*, and ISOBE* must be modified to allow data and address information to pass onto the

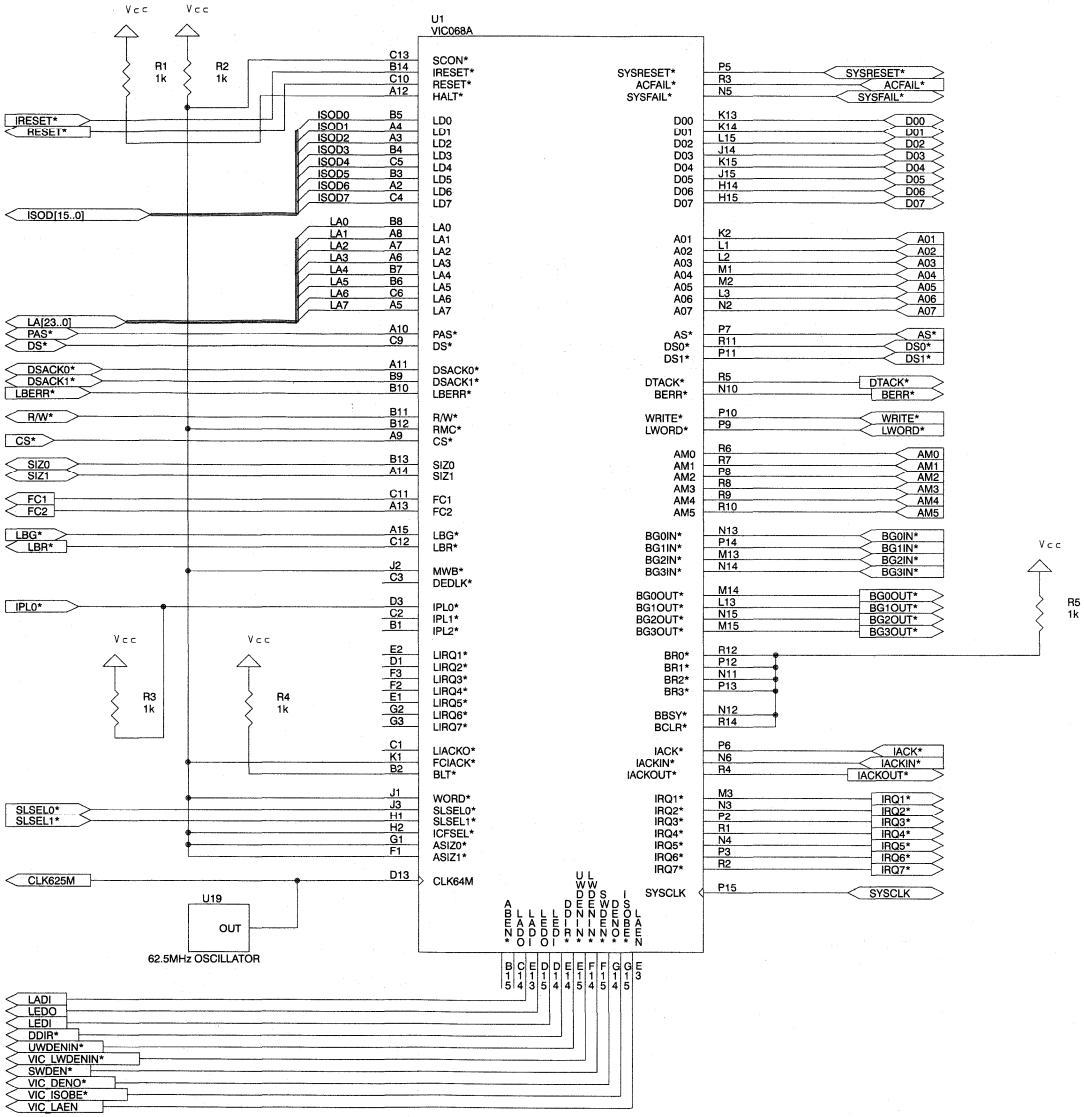


Figure 1. VIC068A Connections

local bus when the configuration slave is accessed. These signals are modified in U14 (a Cypress 5-ns PAL16L8) if the configuration slave is being used instead of the VIC068A's slave capabilities. Other than this, the circuitry implements an A24/D32 interface scheme as defined by the VMEbus specification.

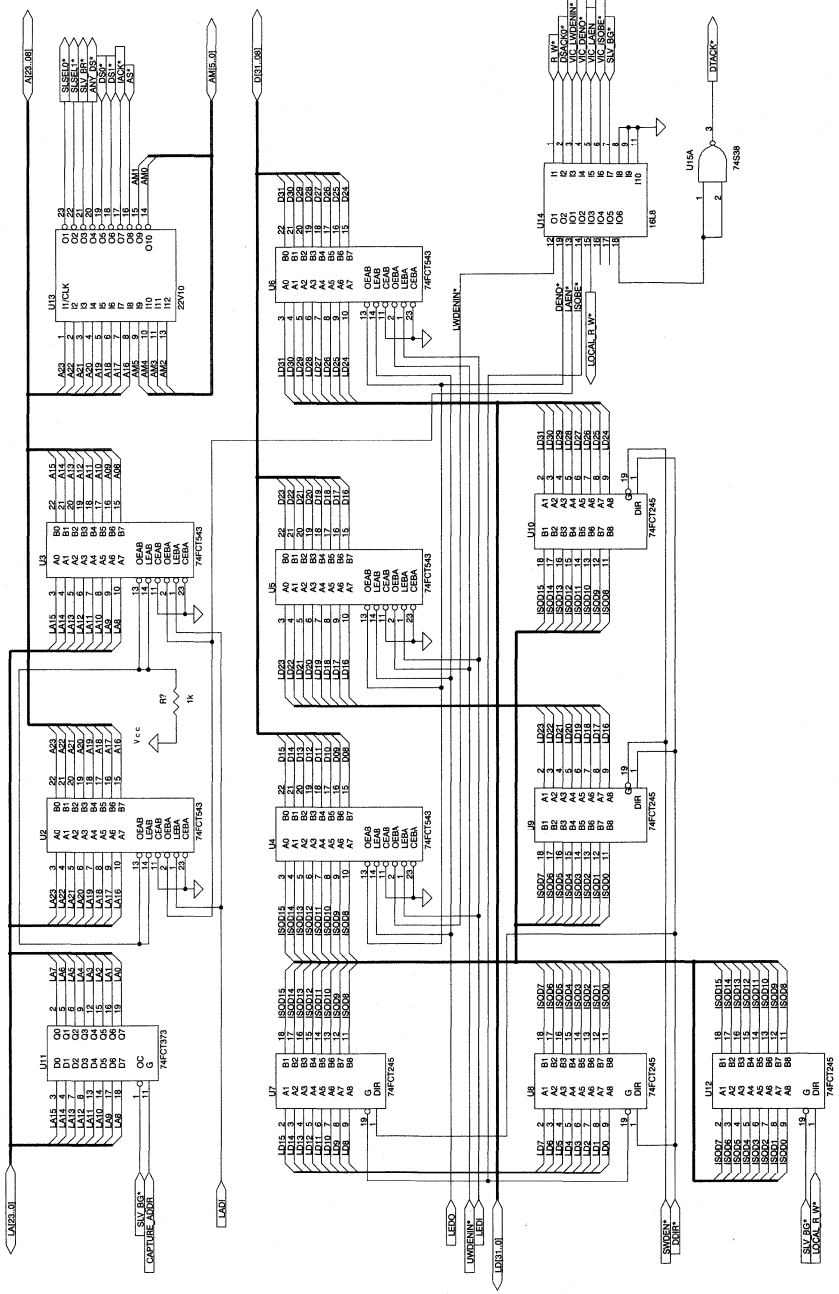


Figure 2. Standard Bus Interface Logic

Specialized Bus Interface Logic

In addition to the standard bus interface logic, a 74FCT245 and a 74FCT373 are needed for the configuration slave interface (*Figure 2*). To program the VIC068A's internal configuration registers, address and data must be present on the VIC068A's LA[7..0] and LD[7..0] buses respectively. No direct path exists between the VMEbus and these buses on the VIC068A. Thus, the 74FCT373 is used to capture and swap addresses from the LA[15..8] bus to the LA[7..0] bus, and the 74FCT245 is used to swap data from the ISOD[15..8] bus to the ISOD[7..0] bus. This swapping only occurs when the configuration slave is granted the local bus by the controller/arbiter. This will be shown in a subsequent section.

The swapping architecture for the configuration slave dictates that a master requiring access to the VIC068A's internal registers must perform an A16/D16 or an A16/D32 data transfer. The lower byte of the address (A[07..01]) is inconsequential, but the upper byte (A[15..08]) must be the address of the register that the master wants to access. Data to a register on a write must be passed on D[15..08], and on a read, register data is valid on D[15..08].

The Selection PLD

U13, the Selection PLD (*Figure 2*), is a Cypress PAL22V10C-10 that decodes VMEbus addresses and control signals to determine what type of access is being made to the board. SLSEL0* and SLSEL1* decode two distinct user defined memory maps in the A24 address space and access the VIC068A as a slave. SLV/BR* decodes *any* access to an A16 address space as an access to the configuration slave. Thus, this board can be the only A16 device in the VMEbus system. (An alternative is to use SLV_BR* to decode another A24 address to allow for multiple boards in the system.) The use of the VMEbus control signals DS0*, DS1*, IACK*, AS*, and AMx in this PLD is only for the configuration slave interface. These signals are used by the VIC068A along with SLSELx* for qualification of a slave access requested of the VIC068A, and thus do not need to be used as qualifiers for SLSELx*.

CY7C361 Controller/Arbiter, 74FCT244 Buffer

The CY7C361 and 74FCT244 are shown in *Figure 3*. The CY7C361 performs the control and arbitration functions necessary to allow two slaves to coexist on one board. It also controls the resetting of the VIC068A during power-up by executing a global reset.

The 74FCT244 buffers the control signals for the configuration slave only when the configuration slave is granted control over the local bus. When the PAS*, DS*, and CS* signals are all active, an access to the VIC068A's registers is performed. They only need to be driven by one control signal from the CY7C361 because there is no required timing relationship between them. The R_W* to the VIC068A is buffered from the LOCAL_R_W*, which is driven from the Control PLD. The SIZ1 and SIZ0 signals are driven to High and Low levels, respectively. This requires that the register address driven on A[15..08] has both A08 and A09 at logic 1 levels. This is consistent with the register addresses given in the VIC068A User's Guide.

Programmable Logic Circuit Description

The Selection PLD

The PLD ToolKit source code for the Selection PLD is listed in Appendix A. The Selection PLD is a Cypress PAL22V10C-10 and has a reference of U13. The SLVBR* signal indicates that an access is being made to the board through the configuration slave. Only the AMx signals are used to determine if the slave interface is being accessed, no address lines are used. Thus any A16 access (AMx = 2Dh) will request access to the local bus. SLVBR* is qualified with AS* and DSx* to guarantee that valid address and data information has propagated through the buffers and latches before local bus access is requested. IACK* is used to guarantee that the cycle is not an interrupt acknowledge cycle. Once SLVBR* is asserted, only the release of the DSx* signals will release the SLVBR* signal. AS* is not involved at this point. AS* could go inactive and then active again during a cycle if address pipelining is supported by the master driving the bus.

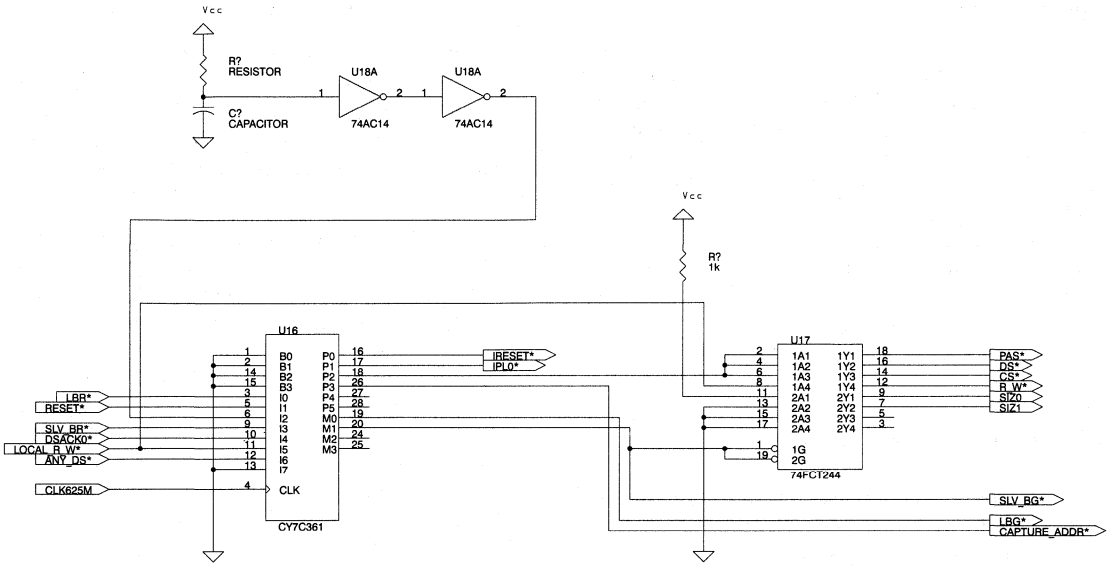


Figure 3. CY7C361 PLD and 74FCT244 Buffer

The SLSELx* signals simply decode the upper bits of the A24 address into two distinct 1-Meg address areas. AS* is used to qualify the SLSELx* signals due to an outstanding design consideration at the time of printing of this application note.

One last miscellaneous signal generated by this PLD is the ANY_DS signal. It's use is also due to an outstanding design consideration at the time of printing of this application note. It merely indicates that one or both of the DSx* signals from the VMEbus are active.

The Control PLD

The PLD ToolKit source code for the Control PLD is listed in Appendix B. The Control PLD is a Cypress PAL16L8-5 and has a reference of U14. The Control PLD has one primary function, and that is to modify the LWDENIN*, DENO*, LAEN, and ISOBE* signals when the configuration slave has access to the local bus. The SLVBG* signal indicates when this access is granted. Additionally, the Control PLD buffers the VMEbus R_W* signal so that it can be distributed across the design without violating any VMEbus specifications for loading or trace length. Finally, the DSACK0* signal is qualified with the SLVBG* signal to generate a DTACK* for the configuration slave.

The Controller/Arbiter PLD

The PLD ToolKit source code for the CY7C361 PLD is listed in Appendix C. The Controller/Arbiter PLD is a Cypress CY7C361-125 and has a reference of U16. There are three functions performed by the Controller/Arbiter PLD. First, there is the state machine that executes a global reset after a power up reset has been completed. This is to insure that all signals and power are stable before the VIC068A is globally reset.

Second, there is the local bus arbiter. The arbiter takes bus requests from the VIC068A and the configuration slave and grants access to the local bus. Obviously, only one access can be granted to the local bus during a given period of time. The priority in the arbiter is given to the configuration slave; but

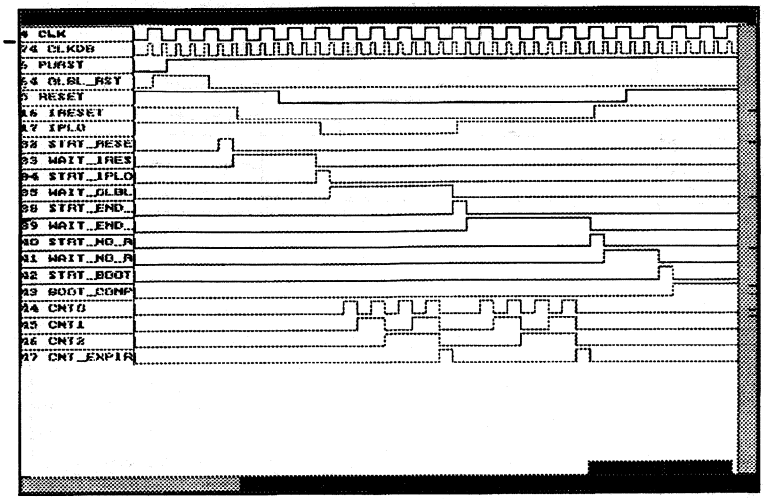


Figure 4. Global Reset State Machine Timing

there is no reason why the VIC068A couldn't have been given the priority instead.

Third, there is the local bus control state machine. This state machine is active when the configuration slave has been granted access to the local bus. Its purpose is to drive the control signals PAS*, DS*, and CS* to the VIC068A, and release the signals when the VIC068A indicates cycle completion with the DSACK0* signal. It also captures the register address in the 74FCT373 and holds it throughout the bus cycle. This insures that if a master supports address pipelining and changes its address during a cycle, the register address is unchanged.

System Timing and Functional Operation

Global Reset

The first function performed when the circuity is powered up is the power-up reset to the Controller/Arbiter, which is done using the RC network and the Schmitt Trigger inverters (74AC14, U18), shown in *Figure 3*. The selection of R and C values depends on the settling time of the power supply and the frequency of the oscillator (U19) used in the design. The power-up reset performs a global reset within the CY7C361 and prepares it to execute its first function, a global reset to the VIC068A. The VIC068A requires that power and signals be stable to the part before it receives a global reset. To insure this, the Controller/Arbiter waits for the completion of the power-up and then begins the timing chain that globally resets the VIC068A. This timing for the CY7C361 is shown in *Figure 4*.

Once the PURST signal goes to a logic-High level, indicating that the power-up reset is complete, The IRESET signal is driven Low. The state machine then waits for the VIC068A to recognize the IRESET*, which the VIC068A indicated by driving its RESET* output Low. The state machine responds by driving IPL0* Low to indicate a global reset is to take place. IPL0* is held Low for five cycles of the 62.5-MHz clock, which equals 80 ns. The VIC068A only requires 50 ns of IPL0* Low time for a proper global reset. After the IPL0* is returned High, IRESET* is held Low for additional clock cycles. This insures that if the VIC068A were to be configured as a VMEbus system controller (it is not in this application note), a proper SYSRESET* (200 ms

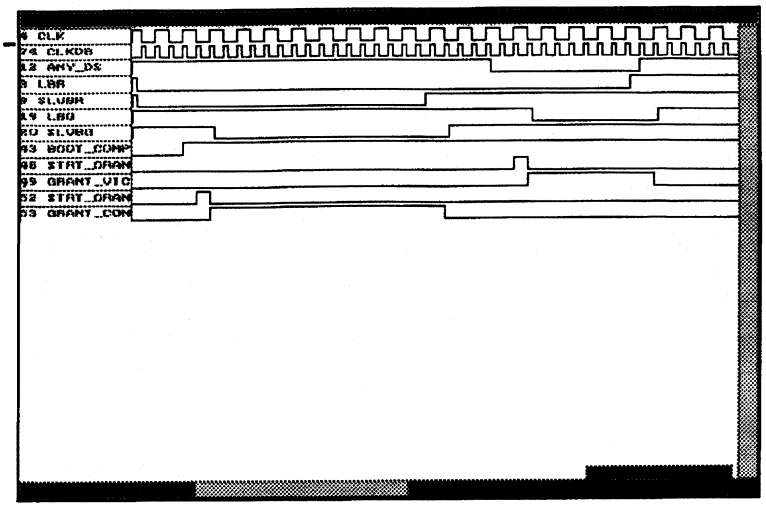


Figure 5. Arbitration Timing

minimum) would be performed on the VMEbus. Finally, after IRESET* is returned High, the state machine waits for RESET* from the VIC068A to go inactive before entering the BOOT_COMPLETE state. Once in the BOOT_COMPLETE state, the arbitration state machine is enabled.

Slave Access to R/W the VIC068A

A VMEbus read cycle begins when a master presents address, address modifier, and control information on the VMEbus and pulls AS* Low. The cycle then continues with DSx* pulled Low. When the master reaches this point in the cycle, it is waiting for the slave to retrieve data and acknowledge that the data is available with the DTACK* signal. The cycle completes when the master returns the AS* and DSx*s High, at which time the slave responds by bringing DTACK* High.

For the master to access the VIC068A registers through the configuration slave, it must present an A16 address to the Selection PLD and then pull AS* and both DS0* and DS1* Low. This indicates that either a 16-bit or 32-bit data transfer is requested. Both allow the proper transfer of data via the configuration slave. Once the A16 address has been qualified, the Selection PLD asserts the SLVBR* signal to indicate that the local bus is requested by the master to read the VIC068A registers.

Figure 5 shows a typical arbitration sequence for the local bus. Here both the LBR* from the VIC068A and the SLVBR* from the Selection PLD go active at the same time. Note that the configuration slave is granted access via the SLVBG* signal, but not before the BOOT_COMPLETE state has been entered. Once the SLVBR* is released, the LBG* can be asserted. To insure that the LBG* doesn't go active before either of the DSx* signals go active (from the design considerations for the VIC068A), the STRT_GRANT_VIC068A state is qualified with ANY_DS.

Figure 6 shows the operation of the control state machine for a read from the VIC068A registers. Once SLVBG* has been asserted, U11 is enabled onto the VIC068A's address bus, and U12 is enabled to drive the VIC068A's data bus onto the VMEbus via U4. In addition, U17 is enabled to drive control signals to the VIC068A. The counter comprised of PCNT0, PCNT1, and PCNT2 implements a variable delay from the falling edge of SLVBG* to the falling edge of both CTRL_PULSE* and CAPT_ADDR*. This delay allows the address information to propagate

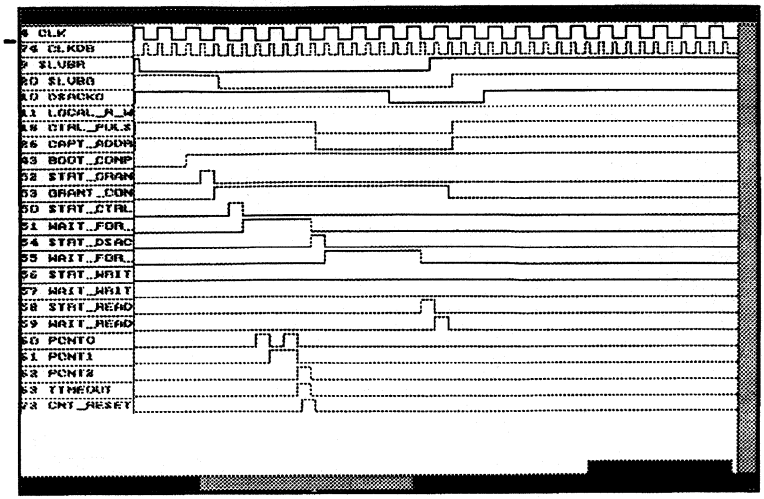


Figure 6. Controller State Machine Read Timing

through the bus interface logic to the VIC068A before the VIC068A is strobed. This delay can be optimized to the speed of the bus interface logic used. An arbitrary delay was chosen for this example, and no optimization was attempted.

When the delay times out, the CAPT_ADDR* and CTRL_PULSE* signals are driven Low. CAPT_ADDR* latches the address into U11 so that if the master supports address pipelining, the address to the VIC068A is unchanged during the current cycle. CTRL_PULSE* is buffered by U17 to drive the PAS*, DS*, and CS* signals to the VIC068A. This initiates the read cycle from the VIC068A.

When the VIC068A has the data available on its bus, it brings the DSACK0* signal Low. This is buffered by the Control PLD and then driven onto the VMEbus via U15. The master, upon receiving DTACK Low, ends its cycle by removing AS* and DSx*. This will cause the Selection PLD to release SLVBR* and then the Controller/Arbiter to release SLVBG*, CAPT_ADDR*, and CTRL_PULSE*. When the CTRL_PULSE* is seen High by the VIC068A, it releases DSACK0* and waits for another access.

The VIC068A register write timing is shown in *Figure 7.* The timing is identical to the read timing except for the CTRL_PULSE signal. The CTRL_PULSE* is terminated as soon as the DSACK0* from the VIC068A is sensed Low. This will strobe data into the VIC068A well before the end of the cycle. This is just a safety measure so that addresses and data are guaranteed stable when the cycle is terminated.

Extensions

Different Data Bus Size

In the interest of saving space on the VMEbus board, the designer may wish to go from a 32-bit data path down to a 16-bit path. This would immediately eliminate the need for U5, U6, U9, and U10, but still allow proper operation of the configuration slave, which only needs D[15..00] for transferring data. Also, WORD* to the VIC068A should be pulled Low instead of High to indicate only a D16 data path exists.

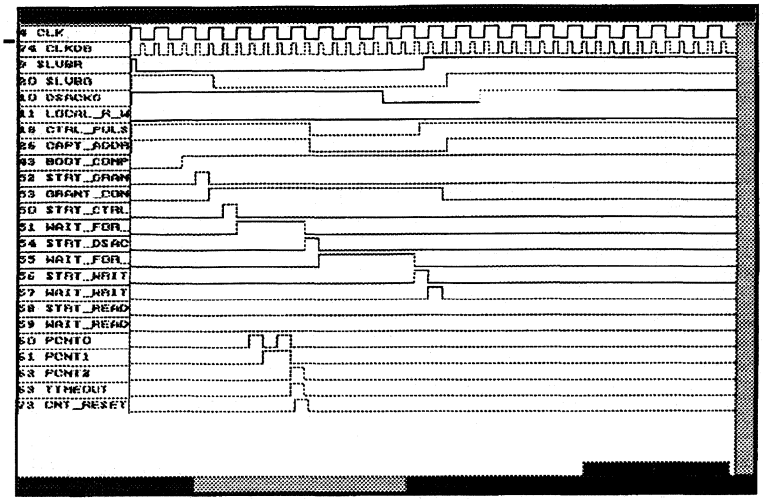


Figure 7. Controller State Machine Write Timing

Unattended DMA Transfers

There are advantages to using the VIC068A for master block transfer when designing a slave board without a processor. The VIC068A would be set up to attain mastership of both the VMEbus and the local bus to transfer data between its board resources and another VMEbus board in the system. An example of this would be executing a DMA transfer of communication packets between boards without intervention of a processor board. Thus, when a packet is received, a processor board can initiate a DMA transfer and then go on to other tasks while the transfer completes.

At first this may sound like a trivial task to initiate. However, it is complicated by the need for a BLT initiation cycle. This cycle is a pseudo write to a VMEbus destination address with the data of the transfer being the local bus starting address. This would require bus interface logic and control circuitry that could take the data coming in from the master via the configuration slave and latch it for master transition on subsequent cycles. In addition, the MWB* to the VIC068A must be controlled, whereas it is unnecessary for the slave operations described in this application note. The implementation of a block transfer circuit is left as an exercise for the reader.

Appendix A. Selection PLD

```
C22V10;

{Selection PLD for "Using VIC068A Without a Processor" Appnote}
{Designed by Cypress Semiconductor}
{Reference Designator U13}

CONFIGURE;

{Address lines from the VMEbus.  Only A[23..16] are decoded for access
within the A23 address space}
A23, A22, A21, A20, A19, A18, A17, A16,

{Address modifiers from the VMEbus.  These are used to determine when an
A16 access is requested for the configuration slave}
AM5, AM4, AM3, AM2(node=13), AM0(noreg,ninv), AM1(noreg,ninv),

{Control Signals from the VMEbus.  These are used to qualify the A16 access
for the configuration slave.}
AS(noreg,ninv), IACK(noreg,ninv), DS1(noreg,ninv), DS0(noreg,ninv),

{Signal that indicates that either of the VMEbus data strobes is active.}
ANY_DS(noreg),

{Decode outputs}
SLVBR(node=21,noreg), SLSEL1(noreg), SLSEL0(noreg),

EQUATIONS;

{A slave bus request is asserted when an A16 supervisor address is present
on the VMEbus and both AS* and both DS1* and DS0* is active.  IACK* is also
included to assure that we are not in an interrupt acknowledge cycle.  To
prevent complications with address pipelining on the VMEbus, the SLVBR
signal,once asserted, will only be released when the data strobes go
inactive}
!SLVBR = <oe>
         <sum> AM5 & !AM4 & AM3 & AM2 & !AM1 & AM0 & IACK & !AS &
            !DS0 & !DS1
         <sum> !SLVBR & !DS0
         <sum> !SLVBR & !DS1;

{SLSEL0 selects a 1M chunk of the A24 address space.  It is selected when
the address = [0000xxxxxxxxxxxxxxxxxxxx].  AS is used to qualify the
address due to the fact that the VIC068A will not release the local bus if
a master continues to drive valid addresses and address modifier codes
during idle periods of the bus.  (Design Consideration #124)}
!SLSEL0  = <oe>
           <sum> !A23 & !A22 & !A21 & !A20 & !AS;
```

Appendix A. Selection PLD (continued)

```
{SLSEL1 selects a 128k chunk of the A24 address space.  It is selected when
the address = [0001000xxxxxxxxxxxxxxxxx].  AS is used to qualify the
address due to the fact that the VIC068A will not release the local bus if
a master continues to drive valid addresses and address modifier codes
during idle periods of the bus.  (Design Consideration #124)}
!SLSEL1  = <oe>
           <sum> !A23 & !A22 & !A21 & A20 & !A19 & !A18 & !A17 & !AS;

{ANY_DS is used by the arbitration circuitry to assure that the LBG* to the
VIC068A is not asserted until a data strobe is asserted on the VMEbus.
This is required for proper slave write posting operation (Design
Consideration #126).}
!ANY_DS  = <oe>
           <sum> !DS0
           <sum> !DS1;
```

Appendix B. Control PLD

```
C16L8;
{Control PLD for "Using VIC068A Without a Processor" appnote}
{Designed by Cypress Semiconductor}
{Reference Designator U14}

CONFIGURE;

{R_W is from the VMEbus, DSACK0 is from the VIC068A}
R_W, DSACK0,

{These four signals normally go directly to the standard bus interface
circuitry.  They must be modified to allow the configuration slave to
operate properly.}
VIC_LWDENIN, VIC_DENO, VIC_LAEN, VIC_ISOBE,

{SLVBG indicates that the configuration slave has been granted access to
the local bus}
SLVBG,

{These are the modified outputs to the standard bus interface circuitry}
LWDENIN(node=12), DENO(node=19), LAEN(node=13), ISOBE,

{This is just a buffered version of the VMEbus R_W signal}
LOCAL_R_W,

{DTACK is driven to the VMEbus through a 74S38}
DTACK(node=18),

EQUATIONS;

!LOCAL_R_W = <oe>
             <sum> !R_W;

{DTACK gets driven when the configuration slave has the local bus and the
VIC068A is acknowledging the local cycle with DSACK0*}
!DTACK   = <oe>
           <sum> !DSACK0 & !SLVBG;

{LWDENIN normally remains inactive High when the VIC068A registers are
accessed.  For the configuration slave to work properly, LWDENIN must be
brought active Low when the bus is granted to the configuration slave
during a write cycle.}
!LWDENIN = <oe>
           <sum> !VIC_LWDENIN
           <sum> !SLVBG & !R_W;
```

Appendix B. Control PLD (continued)

```
{DENO normally remains inactive High when the VIC068A registers are
accessed.  For the configuration slave to work properly, DENO must be
brought active
Low when the bus is granted to the configuration slave during a read
cycle.}
!DENO    = <oe>
           <sum> !VIC_DENO
           <sum> !SLVBG & R_W;

{LAEN normally requires an inversion to work properly.  It is normally low
which when inverted is inactive High.  During configuration slave accesses,
it must be brought low to allow the VMEbus address onto the local bus.}
!LAEN    = <oe>
           <sum> VIC_LAEN
           <sum> !SLVBG;

{ISOBE normally goes active low during accesses to the VIC068A registers.
This
would cause contention on the local bus connected to the VIC068A.  Thus,
ISOBE is kept High when the configuration slave is active.}
!ISOBE   = <oe>
           <sum> !VIC_ISOBE & SLVBG;
```

Appendix C. Controller/Arbiter PLD

```
CY7C361;
{Controller/Arbiter PLD for "Using VIC068A Without a Processor" appnote}
{Designed by Cypress Semiconductor}
{Reference Designator U16}

CONFIGURE;
{-- Misc Internals ------------------------------------------------------}
CLKDB(node=74, dbl_clk),   {Run with clock doubled internally}
IEN1(node=29),             {Input Enable 1}
IEN2(node=30),             {Input Enable 2}
IEN3(node=31),             {Input Enable 3}
GND(node=73),              {Used to permanently assert signals}
GLBL_RST(node=64),         {Global reset to all macrocells}

{-- Inputs --------------------------------------------------------------}
CLK(node=4),               {62.5MHz input clock}
LBR(node=3,ireg),          {Local bus request from VIC068A}
RESET(node=5,iireg),       {Reset from VIC068A}
PURST(node=6,iireg),       {Power up reset}
SLVBR(node=9,ireg),        {Bus request from configuration slave}
DSACK0(node=10,iireg),     {DSACK0 from VIC068A}
LOCAL_R_W(node=11,ireg),   {Local Read/Write}
ANY_DS(node=12,ireg),      {Either DS0* or DS1* is active}

{-- Outputs -------------------------------------------------------------}
IRESET(node=16),           {IRESET signal to VIC068A}
IPL0(node=17),             {IPL0 signal to VIC068A}
CTRL_PULSE(node=18),       {Signal to generate PAS*, DS*, CS* to VIC068A}
CAPT_ADDR(node=26),        {Signal to capture address in 74FCT373}
LBG(node=19),              {Grant of bus to VIC068A}
SLVBG(node=20),            {Grant of bus to configuration slave}

{-- State Macrocells for the reset function -----------------------------}
STRT_RESET(node=32,start),
WAIT_IRESET(cin,term),     {Wait for VIC068A to acknowledge start of reset}
STRT_IPL0(node=34,start),
WAIT_GLBL_RST(cin,term),   {Do global reset}
STRT_END_GR(node=38,start),
WAIT_END_GR(cin,term),     {Finish reset cycle}
STRT_NO_RST(node=40,start),
WAIT_NO_RST(cin,term),     {Wait for VIC068A to come out of reset}
STRT_BOOT_COMPLETE(node=42,start),
BOOT_COMPLETE(cin,term),   {Signal arbiter that reset is finished}
CNT0(node=44,tog),
CNT1(node=45,tog),
CNT2(node=46,tog),         {Counter to produce proper signal lengths}
CNT_EXPIRED(node=47,start),
```

Appendix C. Controller/Arbiter PLD (continued)

```
{-- State Macrocells for the arbitration function ------------------------}
STRT_GRANT_VIC(node=48,start),
GRANT_VIC(cin,term),       {Bus granted to VIC}
STRT_GRANT_CONFIG(node=52,start),
GRANT_CONFIG(cin,term),    {Bus granted to configuration slave}

{-- State Macrocells for the local bus sequencing function
-------------------}
STRT_CTRL(node=50,start),
WAIT_FOR_PROP(cin,term),   {Variable delay between grant of bus and start
                           of local bus cycle}
STRT_DSACK(node=54,start),
WAIT_FOR_DSACK(cin,term), {Wait for acknowledgement from VIC068A}
STRT_WRITE_END(start),
WAIT_WRITE_END(cin,term), {End a write cycle}
STRT_READ_END(start),
WAIT_READ_END(cin,term),   {End a read cycle}

PCNT0(node=60,tog),
PCNT1(node=61,tog),
PCNT2(node=62,tog),
TIMEOUT(node=63,start),    {Variable length counter between grant of bus
                           and start of local bus cycle}

CNT_RESET(node=72),        {Reset for counter when it reaches its
                           programmed maximum}

EQUATIONS;

{-- Misc Housekeeping ----------------------------------------------------}
IEN1 = <inv_sum> /GND;     {Enable inputs}
IEN2 = <inv_sum> /GND;     {Enable inputs}
IEN3 = <inv_sum> /GND;     {Enable inputs}
GLBL_RST = <prod> /PURST; {Reset the 361 on power up}

{-- Reset State machine --------------------------------------------------}
STRT_RESET = <prod> PURST;          {Start global reset right after power up
                                    reset finishes.  This makes sure that
                                    everything is stable before the VIC068A
                                    is reset}
WAIT_IRESET = <prod> /RESET;        {Wait for the VIC068A to acknowledge the
                                    reset}
STRT_IPL0 = <prod> WAIT_IRESET & /RESET;
WAIT_GLBL_RST = <prod> CNT_EXPIRED;    {produce a global reset}
STRT_END_GR = <prod> WAIT_GLBL_RST & CNT_EXPIRED;
WAIT_END_GR = <prod> CNT_EXPIRED;  {hold IRESET after IPL0 removed}
STRT_NO_RST = <prod> WAIT_END_GR & CNT_EXPIRED;
WAIT_NO_RST = <prod> RESET;        {Wait for VIC068A to come out of reset}
```

Appendix C. Controller/Arbiter PLD (continued)

```
STRT_BOOT_COMPLETE = <prod> WAIT_NO_RST & RESET;
BOOT_COMPLETE = <inv_prod>;          {Indicate to arbiter that the local bus
                                     can be granted}
CNT0 = <inv_prod> /WAIT_GLBL_RST & /WAIT_END_GR
     <prod> /CNT_EXPIRED;
CNT1 = <inv_prod> /WAIT_GLBL_RST & /WAIT_END_GR
     <prod> /CNT_EXPIRED & CNT0;
CNT2 = <inv_prod> /WAIT_GLBL_RST & /WAIT_END_GR
    <prod> CNT0 & CNT1 & /CNT_EXPIRED;
                              {Delay counter to produce pulses of valid length}
CNT_EXPIRED = <prod> CNT0 & CNT1 & CNT2;

{-- Reset State machine outputs ------------------------------------------}
/IRESET = <inv_sum> /WAIT_IRESET & /STRT_IPL0 & /WAIT_GLBL_RST &
/STRT_END_GR
         & /WAIT_END_GR;
/IPL0 = <inv_sum> /STRT_IPL0 & /WAIT_GLBL_RST;

{-- Arbiter state machine ------------------------------------------------}
{Grant to the VIC068A only if the configuration slave doesn't have the bus
and is not requesting it.}
STRT_GRANT_VIC068A = <prod> /LBR & SLVBR & /STRT_GRANT_CONFIG &
/GRANT_CONFIG &
            BOOT_COMPLETE & /ANY_DS;
GRANT_VIC068A = <prod> LBR;
{Grant to the configuration slave only if the VIC068A doesn't have the bus.
The configuration slave has priority over the VIC068A if both request the
bus at the same time.}
STRT_GRANT_CONFIG = <prod> /SLVBR & /STRT_GRANT_VIC068A & /GRANT_VIC068A &
              BOOT_COMPLETE;
GRANT_CONFIG = <prod> SLVBR;

{-- Arbiter state machine outputs ----------------------------------------}
/LBG = <inv_sum> /GRANT_VIC;
/SLVBG = <inv_sum> /GRANT_CONFIG;

{-- Control state machine ------------------------------------------------}
STRT_CTRL = <prod> GRANT_CONFIG;     {Begin local bus cycle when
                                     configuration slave gets the bus}
WAIT_FOR_PROP = <prod> TIMEOUT;      {Wait a programmed amount of time to
                                     allow the address to propagate to the
                                     VIC068A and then start the cycle}
STRT_DSACK = <prod> TIMEOUT;
WAIT_FOR_DSACK = <prod> /DSACK0;     {Start the cycle to the VIC068A and wait
                                     for it to acknowledge with DSACK0}
STRT_WRITE_END = <prod> /LOCAL_R_W & WAIT_FOR_DSACK & /DSACK0;
WAIT_WRITE_END = <prod> SLVBR;       {Wait for a write cycle to end}
STRT_READ_END = <prod> LOCAL_R_W & WAIT_FOR_DSACK & /DSACK0;
```

Appendix C. Controller/Arbiter PLD (continued)

```
WAIT_READ_END = <prod> SLVBR;       {Wait for a read cycle to end}
PCNT0 = <prod> WAIT_FOR_PROP;
PCNT1 = <prod> WAIT_FOR_PROP & PCNT0;
PCNT2 = <prod> WAIT_FOR_PROP & PCNT0 & PCNT1;
TIMEOUT = <prod> PCNT0 & PCNT1 & !PCNT2;
CNT_RESET = <prod> TIMEOUT;         {Counter can be optimized for the
                                    required propagation delay to get the
                                    address to the VIC068A before starting a
                                    cycle.  Counter resets when it reaches
                                    the selected count}

{-- Control state machine outputs ----------------------------------------}
{CTRL_PULSE is terminated when DSACK0 goes active for writes}
/CTRL_PULSE = <inv_sum> /STRT_DSACK & /WAIT_FOR_DSACK & /STRT_READ_END &
             /WAIT_READ_END;
/CAPT_ADDR = <inv_sum> /STRT_DSACK & /WAIT_FOR_DSACK & /STRT_READ_END &
             /WAIT_READ_END & /STRT_WRITE_END & /WAIT_WRITE_END;
```

Document #: BUS0005–0792

Connecting the Cypress VIC068/VAC068 to the TI TMS320C40: A Prototype Design

Introduction

The Cypress Semiconductor VIC068 VMEbus Interface Controller and its companion VAC068 VMEbus Address Controller provide a complete VMEbus interface including master and slave capability (Reference 2). As these components can be used in a wide variety of applications, it is natural to utilize the VIC068/VAC068 in a single- or multiple-TMS320C40 VMEbus card design.

This application note provides high-level as well as low-level details of interfacing VIC/VAC to TMS320C40. This allows for techniques to be implemented to minimize design time for subsequent efforts since this design has not been optimized for either size or speed. The Design Requisites section provides the design goals established prior to design as well as relevant background regarding devices involved. Hardware details, including schematics and programmable logic source code, represent the central focus of the paper. In addition, software initialization of the chip, set by the TMS320C40, is covered. Throughout this note, it is assumed that the reader is familiar with the TMS320C40 architecture (Reference 1), the basics of the VIC068A/VAC068A (Reference 2), and the VMEbus and its protocol(s) (References 5, 6).

Design Requisites

Design Goals

This project began with the development of a set of design goals for the VME interface based on our particular needs. The main focus was on a TMS320C40 card providing both (VMEbus) master and slave capability for reads, writes, read-modify-writes, write posting, and slave block transfers. In terms of the address/data capability, a design was made to the most prevalent configuration (for other cards available commercially): 24-bit address and 32-bit data (i.e., A24/D32). However, the design presented here does not preclude 32-bit addressing with minor modifications. Via the VIC068, this design also provides system controller capability. VMEbus interrupt support is not provided. The VAC068 is utilized to provide address control/mapping, two Universal Asynchronous Receiver/Transmitters (UARTs) (required for our application), and a general purpose parallel I/O. In addition to the VMEbus functionality, interface compatibility is required with both the existing TMS320C40–40, which has 50-nanosecond cycle time, and the faster TMS320C40, 40-nanosecond part.

Design Considerations

The 68020 User's Manual (Reference 7) was referenced extensively for this design, which covers a complete examination of the VIC068 and VAC068 and extends to review the 680x0 family bus signals and cycles. An examination of the VIC068 and VAC068 reveals a direct interface to the Motorola 680x0 family data, address, and control signals. The VIC068 and VAC068 are both driven with the familiar 680x0 address and data strobes (PAS*, DS*), which have an asynchronous transfer protocol implemented with the data transfer and size acknowledgment signals DSACK0* and DSACK1*. In addition to these signals, the transfer size signals, SIZ0 and SIZ1, are an integral part of the 680x0's dynamic bus sizing capability and, with the the lower ad-

dress lines, encode the size of the transfer in progress. During transfer, the function code signals (FC0–FC2) provide additional information of importance in multi-user environments. Bus arbitration is an integral part of the 680x0 via the bus request (BR*), bus grant (BG*), and bus grant acknowledge (BGACK*) signals and are used directly by the VIC068. Finally, like many other general purpose microprocessors, bus cycles for the 680x0 are several clock cycles long.

Although the VIC068 and VAC068 are driven by, and can drive, the familiar 680x0 bus signals, a quick examination of the TMS320C40 bus signals shows little similarity to the 680x0 family. The TMS320C40 provides a bus protocol common to the TMS320 floating-point DSP product line. An external ready signal allows for wait-state generation and controls the rate of transfer in a synchronous fashion (i.e., cycles can be extended an integer number of clock cycles). As described in Reference 1, the TMS320C40 provides a 32-bit address range divided into two identical 31-bit address, 32-bit data buses termed local and global. The TMS320C40 executes single cycle instructions and relies on a multistage pipeline for execution speed. Detailed bus status is provided for each cycle via the STAT lines, which provide information regarding the type of instruction and access. Individual control lines are provided for three-stating the address, data, and control bus(es). A read-modify-write signal is not provided (as it is on the 680x0). However, an instruction-driven LOCK* signal is available. Each cycle is controlled by a strobe (STRBx*) signal in conjunction with the corresponding read/write (R/Wx*) strobe. One of the TMS320C40's many features is the range of configuration options for this external interface. The TMS320C40 has evolved from its earlier floating-point counterparts and provides a truly flexible interface via the local and global bus configuration control registers.

High-Level Architecture

The high-level architecture for the card places 20-nanosecond (ns), high-density, 4-megabit (128Kx32) Cypress CMOS SRAM modules on both local and global buses of the TMS320C40. (The size of the memory array should not impact the TMS320C40-to-VIC068/VAC068 interface design.) The global side is designated as program memory and the local side as data memory for the application. It is anticipated that the local memory will be fully occupied by DMA coprocessor activity coupled with data fetches during communications-oriented DSP operations. Given this, the A24 VME spectrum is placed on the global (program) side, segmenting the local side I/O activity, the critical path for the application, from all VMEbus activity. (Note that the interface documented herein can be used on either side due to the symmetry of the global and local buses.) In addition to programming SRAM on the global side, two 128Kx16 EPROMs for embedded program store are also placed, with the boot load feature of the TMS320C40 applied.

Because the design is limited to VMEbus A24 addressing, its spectrum fits nicely anywhere in the TMS320C40 global side address spectrum, from 08000 0000h to 0FFFF FFFFh. Therefore, VMEbus master access is memory mapped into the TMS320C40 global side address range. When an access occurs in this predefined A24 range, the TMS320C40 bus signals are transformed into 680x0 bus signals. These drive the VIC068/VAC068 pair and initiate a VMEbus transfer. Global side accesses outside of this range do not generate such signals and occur at full speed (i.e., the speed appropriate for that memory or peripheral). Regarding the "endianess" (References 8, 9) of the interface, it is known that the 680x0 family maintains big-endian byte ordering (byte addressable memory organization) with little-endian bit ordering in each addressable unit. In contrast, the TMS320C40 is flat in its byte endianess (32-bit word addressable only) and little-endian bit ordered. Therefore, no swapping is done on the interface as 32-bit word transfers (between processors) maintain D0 as the least significant bit. This forces the designer to tradeoff transfer speed with a wider range of transfers (byte, word, and three byte) than inherent to the TMS320C40 (longword). Transfers are limited to D32. In order to provide transfers of all sizes, additional set-up and/or decoding would have to be done prior to/during the transfer in progress.

Hardware Description

After examining the VIC068/VAC068 interface and capabilities and comparing them with the TMS320C40, a prototype design was initiated. Based on the above discussion, the strategy is to map from the given set of TMS320C40 bus signals to a set of 680x0-like signals driving their counterparts on the VIC068 and VAC068 for master cycles. (Note: the TMS320C40 is reading from or writing to the VMEbus.) Not only can the TMS320C40 initiate VMEbus cycles as a bus master, the card can also respond to slave cycles. Most often, slave access is used in terms of access to shared memory on the (slave) card. To accomplish this on the TMS320C40-based VME card, a set of signals is required to respond to bus requests from the VIC068/VAC068 and an additional set is required to "hold off" the TMS320C40 global side during such transfers.

To accomplish this transformation of signals, programmable logic is applied. It is desirable to keep the design time to a minimum while maintaining the most flexible or programmable design. Based on this, Cypress's CY7C335 universal synchronous EPLDs (Reference 3) are used. These devices are field programmable and optimized for state machines. The 335 has 12 input macrocells, 12 output macrocells, 256 product terms, 4 buried registers, and operates to a maximum frequency of 100 megahertz (MHz). Development tools for these EPLDs include *Warp2*, a VHDL compiler from Cypress, and Data I/O's ABEL™ version 4.0 or better, using fitters available on the Cypress Bulletin Board (408–943–2954).

Reset Circuitry

The VIC068 *must* receive a global reset in order to function correctly. The global reset should occur after the power supply and the 64-MHz (U4) oscillator have stabilized and before any interaction with the chip is attempted. The VIC requires both the leading and trailing edges of IRESET*/IPL0, as shown on page 14–32 of the Users Guide and as discussed in Chapter 12.

To implement this function, an R/C network along with a pair of Schmitt Trigger inverters are used to create a power-up signal. The R/C values are not given since they will be a function of the system power supply and oscillator power-up delay time. The power-up signal is supplied to U12, a 22V10 that is programmed as a state machine to create the required waveforms. (Part of U12 is also used for address decoding.) After the power up delay is complete, the power-up signal goes High, which causes U12 to drive IRESET Low. The state machine waits for the VIC to respond by driving RESET Low. It then drives IPL0 Low for two clock cycles indicating a global reset is to take place. IPL0 is returned to a High state followed by the IRESET signal. If the VIC is configured as a VMEbus system controller, a global reset will cause the VIC to drive the SYSRESET line for 200 ms, as required by the VMEbus specification. The RESET output on the VIC068 is used to reset both the TMS320C40 as well as the VAC068 and all programmable logic onboard.

Address Bus Decoding

The VIC068/VAC068 interface, and consequently the VMEbus itself, is mapped into the TMS320C40 global side at 0D000 0000h. In this application, the global side is divided into two halves via the STRB ACTIVE field in the TMS320C40 (global) memory control register. Zero-wait state devices (fast SRAM) are placed in the lower half and slower memory (EPROM) and peripherals (the VIC068/VAC068 pair) are placed in the upper half. Therefore, the TMS320C40 addresses program memory via STRB0 and addresses the VMEbus via STRB1. As shown in the accompanying schematics, U12 is a 22V10 PAL used for $\overline{\text{STRB1}}$ address bus decoding. In particular, a Cypress PAL22V10D–7 was used. This PAL is used to decode each (global) TMS320C40 $\overline{\text{STRB1}}$ bus cycle using the TMS320C40 H1 clock. Cycle type decoding is accomplished fully via the STAT lines (versus simply using the R/W strobe) and allows for future expansion/reconfiguration if required. As shown, the $\overline{\text{STRB1}}$ range is divided into 8 distinct segments via use of GA28–GA30. (GA31 is implicitly a logic 1.) Outputs of the decoding operation are (VMEbus) master write (MWR*), master read (MRD*),

VIC068/VAC068 register write (RWR*), register read (RRD*) and EPROM read (GPROM*). As found in the VIC068/VAC068 documentation, the VAC068 is hard-wired, starting at address 0FFFD 0000h. It also provides for VIC068 selection, starting at address 0FFFC 0000h. A memory map for the global side, as decoded by the 22V10 PAL, is shown in *Table 1*. The ABEL source code is provided in the appendix.

Table 1. Global Side Memory Map

Address	Unit Addressed
08000 0000h	SRAM
0C000 0000h	EPROM
0D000 0000h	VMEbus A24
Address 00 0000h	
0FFFC 0000h	VIC068 Register Set
0FFFD 0000h	VAC068 Register Set

Bus Control

Once a cycle in the VMEbus address range is detected by the address decoding PAL, the sequencers shown provide the signals required for both master and slave cycles. U13 is the first of three sequencers and accomplishes overall bus control providing enable signals for global bus access by the TMS320C40 (GBE*), master cycle sequencer output (MOE*), slave cycle sequencer output (SOE*), VMEbus slave local bus grant (LBG*) and a ready signal for the TMS320C40 (GRDY1*). Notice that a full complement of inputs are presented to the bus control sequencer. This is done to accommodate all possible cycles and to allow reconfiguration without hardware changes. While the TMS320C40 H3 clock (20 MHz) is used here, this is not an absolute requirement as the array of sequencers operate asynchronously once a master or slave cycle begins. The use of H3 here, however, does simplify the sequencer code because the H3 clock serves as a convenient reference to the TMS320C40 cycle in progress.

A master cycle begins with U12 generating a master read or write signal or either register read or write. This enables the output of the master bus cycle signal generation sequencer U14. In fact, this signal is asserted during all bus activity other than slave cycles. Termination of a master cycle ends with the assertion of the acknowledge signals DSACK0* and DSACK1* and/or the local bus error signal (LBERR*). All are generated by the VIC068 in response to acknowledge signals provided over the VMEbus. The sequencer responds to these signals by providing the ready signal for this TMS320C40 $\overline{\text{STRB1}}$ access (RDY1*) by asserting GRDY1*. In this design, external ready signals are used exclusively, versus ANDing or ORing with internal ready, and the generation of the ready signal conforms to the second of two methods called out in Reference 1: ready is High between accesses.

Slave cycles are initiated by the assertion of local bus request (LBR*) by the VIC068. With this asserted, U13 provides bus control by first disabling the TMS320C40 global bus (deasserting GBE*), then disabling the master bus cycle generation sequencer outputs (U14 MOE*), followed by enabling the outputs on the slave bus cycle signal generation sequencer (SOE*), U15. Given that the bus has been successfully "seized", the local bus grant signal (LBG*) is asserted. Slave cycles are terminated by deassertion of the local bus request input.

Master Bus Cycle Generation

The master bus cycle generation sequencer U14 runs in tandem with the bus control seqencer U13. The sequencer code found in U13 and U14 results from one common state diagram. It is necessary to split these functions up due to limitations of the number of outputs per sequencer. Therefore, the inputs to U14 are identical to those found on U13. Master bus cycles proceed according to the appropriate cycle (read or write) definition found in Reference 7. The function code lines are driven to a supervisory state, giving the widest possible audience, supervisory data. Termination of a master cycle ends with the assertion of the acknowledge signals DSACK0* and DSACK1* and/or the local bus error signal (LBERR*) as described above. Note that VIC068/VAC068 register accesses are also master accesses in the address range(s) specified above. While the sequencer code does not initiate read-modify-write cycles, it is easy to see how the

use of the TMS320C40 GLOCK* input could be used to accomplish this.

Slave Bus Cycle Generation

Slave cycles are initiated by the VAC068 in response to a request over the VMEbus in the selected range as determined in the appropriate VAC068 register (covered in the next section). As shown, inputs to the sequencer are the common 680x0 bus signals driven by the VIC068 for slave cycles and alternately driven by the master sequencers for master cycles. Assertion of the local bus grant signal (LBG*) by U13 indicates the absence of the TMS320C40 on the global bus, thereby allowing access of (shared) global SRAM by the VIC068/VAC068 pair. Assuming the correct transfer size, the memory strobe signals GSTRB0* and GR/W0 are driven to provide access to the shared global SRAM. Following this, acknowledgement is provided via DSACK0* and DSACK1*, ending the slave cycle. Note that while VAC068 documentation states that its DSACK signals can be three-stated on the assertion of LAEN, this was not the case with this configuration. Therefore, U8A was required to artificially three-state those signals so that the slave sequencer could control the data acknowledgement.

VIC068/VAC068 Software Initialization

The VIC068/VAC068 pair register set can be overwhelming at first glance, but very few registers require attention prior to using the pair for either master or slave operations. The VAC068 should be initialized first as this controls both master and slave address mapping. With that complete, the VIC068 is programmed. Fine tuning of the interface can be performed using the programmable delay registers for the interface after initial capability is verified. As the VIC068/VAC068 was programmed using C, vic.h and vac.h header files were developed which provide base and offset definitions for the complete register set of each device.

Before programming the VIC068/VAC068 pair, the VAC068 must be brought out of its initial "Force EPROM" mode which asserts EPROMCS* for all accesses. This is accomplished by reading from the EPROM space beginning at 0FF00 0000h. The address decoding PAL U12 does not provide for access to this range. However, a dummy access to this region can be initiated by manipulating the TMS320C40 (global) memory interface control register. The register is set to provide zero-wait state, internal ready dependent (only) accesses to the appropriate strobe (STRB1 for this case). With this set in the SWW and WTCNT fields, a read from address 0FF00 0000h is performed. Immediately following this read, the SWW field to external ready accesses is reset and a second read to the VAC068 is performed, this time at the VAC068 register base, 0FFFD 0000h. This read provides the required access to "snap" the sequencers back to their default states.

After the "Force EPROMCS" mode is exited, it is verified that the VAC068 can be addressed by reading the device ID via the VAC068 ID register. With that established, the (slave) SLSEL0 Base Address register and the SLSEL0 Address Mask register followed by the (master) A24 Base Address register are programmed. To enable the VAC068 decode and compare functions, the last step is to write to the VAC068 ID register. The VIC068 ID register is similarly polled and. following the successful read of that register, the Address Modifier Source register and the Slave Select 0 Control 0 register are set. This completes the initial programming of the pair. At this time, the SYSFAIL LED, if applicable, may be extinguished by writing to the Interprocessor Communication 7 register. The initial register settings for this application are provided in *Table 2*.

Table 2. VIC068/VAC068 Initial Register Settings

Address	Register	Size (Bits)	Setting
0FFFD 0200h	VAC SLSEL0 Base	16	0010h
0FFFD 0300h	VAC SLSEL0 Mask	16	00F0h
0FFFD 0800h	VAC A24 Base	13	0D10h
0FFFD 2900h	VAC ID	16	Write to EnableVAC
0FFFC 00B4h	VIC Address Modifier	8	03Dh
0FFFC 00C0h	VIC Slave Select 0 Control 0	8	014h

Conclusion

The development of a prototype interface between the TMS320C40 DSP and the Cypress VIC068/VAC068 was accomplished with a minimum amount of programmable logic in the form of simple PALs and sequencers. The result is a reconfigurable, programmable interface for A24/D32 VMEbus master/slave capability. While the initial transfer speed is low, speed gains can be made by increasing the sequencer's clock speed and eliminating unnecessary states in the prototype sequencer code. Read-modify-write cycles can be accomplished with the existing hardware via the use of the TMS320C40 LOCK instruction group. Ultimately, the design documented herein could be encapsulated in an FPGA for both speed and size gains.

References

1. *TMS320C4x Users Guide*. Texas Instruments, 1991.
2. *VIC068A/VAC068A Users Guide*. Cypress Semiconductor, 1992.
3. *Cypress Semiconductor CMOS/BiCMOS Data Book*. Cypress Semiconductor, 1992.
4. Siy, P. F., and W. T. Ralston, "Application of the TI TMS320C40 in Satellite Modem Technology," to be published, to be presented at the Third Annual International Conference on Signal Processing Applications and Technology, November, 1992, Boston, MA.
5. *IEEE Standard for a Versatile Backplane Bus: VMEbus*. IEEE, 1987, New York, NY: Wiley-Interscience.
6. Peterson, W. D., *The VMEbus Handbook*, Scottsdale, AZ: VFEA International Trade Association.
7. *MC68020 32-Bit Microprocessor User's Manual*. Motorola, Inc., 1984
8. Henessey, J. L., and D. A. Patterson, 1990, *Computer Architecture A Quantitative Approach*, San Mateo, CA: Morgan Kaufmann Publishers, Inc.
9. Dewar, R. B. K, and M. Smosna, 1990, *Microprocessors A Programmers View*, New York, NY: McGraw-Hill, Inc.

Appendix A. Address Bus Decoder – ABEL Source

```
module          U12
title           'Global Bus Decode
Revision        1.0
Part            Cypress PAL22V10D-7
Abel Version    4.3
Project         C40 I/O Board'

     U12 device 'P22V10';

"Inputs"
clk, reset                  pin 1,2;     "clock, reset"
gstat0,gstat1               pin 3,4 ;    "C40 status"
gstat2,gstat3               pin 5,6;     "C40 status"
ga28,ga29,ga30              pin 7,8,9;   "C40 address"
gstrb1                      pin 10;      "C40 strobe 1"
oute                        pin 11;      "output enable
pureset                     pin 13;      "output enable"

"Outputs"
ipl0,ireset,tmp1            pin 17,18,16 istype 'reg,invert';
mrd,mwr                     pin 19,20;   "master read & write"
rrd,rwr                     pin 21,22;   "register read & write"
gprom                       pin 23;      "PROM select"

"Misc"
ga31 = 1;                               "dummy var"

"Sets"
stat = [gstat3,gstat2,gstat1,gstat0];    "status"
addr = [ga31,ga30,ga29,ga28];            "ms nibble"
output = [gprom,rwr,rrd,mwr,mrd];        "output"
power_up = [ireset,ipl0,tmp1];           "reset

" state definations for power up reset sequence.
"  [ireset,ipl0,tmp1]
s0 = [1,1,1] ; "initial state
s1 = [0,1,1] ; "ireset asserted
s2 = [0,0,1] ; "ireset/ipl0 asserted
s3 = [0,0,0] ; "hold for extra clock
s4 = [0,1,0] ; "de-assert ipl0
s5 = [1,1,0] ; "de-assert ireset
s6 = [1,0,0] ; "should never get here
s7 = [1,0,1] ; "should never get here

H,L,X,C,Z = 1,0,.X.,.C.,.Z.;
```

Appendix A. Address Bus Decoder – ABEL Source (continued)

```
equations
output.c = clk;
power_up.c = clk;
output.oe = !oute;
power_up.oe = !oute;

"Master Read"
!mrd := reset & (addr == ^hd) & (stat == [1,0,X,X]) & !gstrb1;
"Master Write"
!mwr := reset & (addr == ^hd) & ((stat == [1,1,0,1]) #
(stat == [1,1,1,0])) & !gstrb1;

"Register Read"
!rrd := reset & (addr == ^hf) & (stat == [1,0,X,X]) & !gstrb1;
"Register Write"
!rwr := reset & (addr == ^hf) & ((stat == [1,1,0,1]) #
(stat == [1,1,1,0])) & !gstrb1;

"PROM Read"
!gprom := reset & (addr == ^hc) & (stat == [1,0,X,X]) & !gstrb1;

" power up reset state equations
state_diagram power_up

" Initial power up state
state s0:
if (pureset) then s1
else s0;

" assert ireset
state s1:
if (!pureset) then s0
else if (!reset) then s2
else s1;

"assert ipl0
state s2:
if (!pureset) then s0
else s3;

"keep both asserted for extra clock
state s3:
if (!pureset) then s0
else s4;
```

Appendix A. Address Bus Decoder – ABEL Source (continued)

```
"de-assert ipl0
state s4:
if (!pureset) then s0
else s5;

"de-assert ireset and stop
state s5:
if (!pureset) then s0
else s5;

"check on indeterminate states
state s6:
if (!pureset) then s0
else s6;

"check on indeterminate states
state s7:
if (!pureset) then s0
else s7;

test_vectors
([clk,reset,gstat3,gstat2,gstat1,gstat0,ga30,ga29,ga28,
gstrb1,oute] -> output)

[C,X,X,X,X,X,X,X,X,1] ->     Z;     "1 test for high-z"
[C,0,X,X,X,X,X,X,X,0] ->    ^b11111;"2 test for reset"
[C,1,1,0,X,X,1,0,1,0,0] ->   ^b11110;"3 test for master read"
[C,1,1,1,0,1,1,0,1,0,0] ->   ^b11101;"4 test for master write"
[C,1,1,1,1,0,1,0,1,0,0] ->   ^b11101;"5 test for master write"
[C,1,1,0,X,X,1,1,1,0,0] ->   ^b11011;"6 test for register read"
[C,1,1,1,0,1,1,1,1,0,0] ->   ^b10111;"7 test for register write"
[C,1,1,1,1,0,1,1,1,0,0] ->   ^b10111;"8 test for register write"
[C,1,1,0,X,X,1,0,0,0,0] ->   ^b01111;"9 test for PROM read"
[C,1,1,0,X,X,0,0,0,0,0] ->   ^b11111;"10 test bad address"
[C,1,0,0,0,0,1,1,1,0,0] ->   ^b11111;"11 test bad status"
end U12
```

Appendix B. Bus Control Sequencer – ABEL Source

```
module          U13C
title           'C40 Bus Control
Revision        1.0
Part            CY7C335
Abel Version    4.3 using Cypress fitter
Project         TMS320C40 I/O Card '

U13C device 'p335';

"Inputs"
clk, reset                  pin 1,13;          "clock, reset"
!mrd,mwr,rrd,rwr,gprom      pin 24,11,10,9,12; "decoded cycle"
mwb,lbr                     pin 7,6;           "master/slave requests"
dedlk                       pin 5;             "m/s deadlock"
dsack0,!dsack1,!lberr       pin 4,28,15;       "cycle responses"
!glock                      pin 26;            "C40 lock"
oe                          pin 14;            "output enable"

"Outputs"
lbg       pin 27      istype 'reg_RS,invert' ;"slave grant"
gbe       pin 16      istype 'reg_RS,invert' ;"C40 g bus enable"
soe,moe   pin 17,18   istype 'reg_RS,invert' ;"pls oe(s)"
grdy1     pin 19      istype 'reg_RS,invert' ;"C40 ready 1"

"Sets"
cycle  = [gprom,rwr,rrd,mwr,mrd];  "cycle request"
ack    = [dsack1,dsack0];          "acknowledge"
output = [grdy1,soe,moe,gbe,lbg];  "output"

"State Description"
P4,P3,P2,P1 node 34,33,32,31;
P0   pin  25 istype 'reg,invert';
P4,P3,P2,P1  istype 'reg';

sreg = [P4,P3,P2,P1,!P0];
 S0 = [0,0,0,0,0];
 S1 = [0,0,0,0,1];
 S2 = [0,0,0,1,0];
 S3 = [0,0,0,1,1];
 S4 = [0,0,1,0,0];
 S5 = [0,0,1,0,1];
 S6 = [0,0,1,1,0];
 S7 = [0,0,1,1,1];
 S8 = [0,1,0,0,0];
 S9 = [0,1,0,0,1];
 S10 = [0,1,0,1,0];
 S11 = [0,1,0,1,1];
```

Appendix B. Bus Control Sequencer – ABEL Source (continued)

```
 S12 = [0,1,1,0,0];
 S13 = [0,1,1,0,1];
 S14 = [0,1,1,1,0];
 S15 = [0,1,1,1,1];
 S16 = [1,0,0,0,0];
 S17 = [1,0,0,0,1];
 S18 = [1,0,0,1,0];
 S19 = [1,0,0,1,1];
 S20 = [1,0,1,0,0];
 S21 = [1,0,1,0,1];
 S22 = [1,0,1,1,0];
 S23 = [1,0,1,1,1];
 S24 = [1,1,0,0,0];
 S25 = [1,1,0,0,1];
 S26 = [1,1,0,1,0];
 S27 = [1,1,0,1,1];
 S28 = [1,1,1,0,0];
 S29 = [1,1,1,0,1];
 S30 = [1,1,1,1,0];
 S31 = [1,1,1,1,1];

"Misc"
H,L,X,C,Z = 1,0,.X.,.C.,.Z.;

equations
output.OE = !oe;          "set output enable"
output.CLK = clk;         "clock the output regs"
sreg.CLK = clk;           "and state regs"

@page
state_diagram sreg
state S0:

if !reset then S0 WITH
 lbg.S = 1; "slave disable"
 gbe.R = 1; "enable C40 global side"
 soe.S = 1; "slave disable"
 moe.R = 1; "enable master pls"
 grdy1.S = 1; "not ready"
 ENDWITH;

else if (!mrd # !mwr & lbr) then S1;     "master read/write"
else if (!rrd # !rwr & lbr) then S4;     "reg read/write"
else if (!gprom & lbr) then S8;          "EPROM read"
else if (!lbr # !dedlk) then S16 WITH    "slave request"
gbe.S = 1;                               "disable global side"
moe.S = 1;                               "and master pls"
 ENDWITH;
```

Appendix B. Bus Control Sequencer – ABEL Source (continued)

```
else S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "slave disable"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

@page
"Master Read/Write"

state S1:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else if !dedlk & ((!mwb) # (mwb)) then S16 WITH
 moe.S = 1;
 gbe.S = 1;
 ENDWITH;

else if !mwb then S2;"wait for !mwb"

else S1;

state S2:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else if !dedlk & ((!mwb) # (mwb)) then S16 WITH;
 moe.S = 1;
 gbe.S = 1;
 ENDWITH;

else if ((!dsack1 & !dsack0) # !lberr) then S3 WITH
 grdy1.R = 1;
 ENDWITH;

else S2;
```

Appendix B. Bus Control Sequencer – ABEL Source (continued)

```
state S3:
goto S0 WITH
 grdy1.S = 1;
 ENDWITH;

@page
"Register Read/Write"

state S4:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else if !dsack1 then S5 WITH
 grdy1.R = 1;
 ENDWITH;

else S4;

state S5:
goto S0 WITH
 grdy1.S = 1;
 ENDWITH;

@page
"EPROM Read, 150ns EPROMs"

state S8:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S9;
```

Appendix B. Bus Control Sequencer – ABEL Source (continued)

```
state S9:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S10;

state S10:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S11;

state S11:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S12 WITH
 grdy1.R = 1;
 ENDWITH;

state S12:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S0 WITH
 grdy1.S = 1;
 ENDWITH;
```

Appendix B. Bus Control Sequencer – ABEL Source (continued)

```
@page
"Local Bus Request"

state S16:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S17 WITH
 soe.R = 1;          "enable slave PLS"
 ENDWITH;

state S17:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S18 WITH
 lbg.R = 1;     "finally allow slave access"
 ENDWITH;

state S18:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else if lbr then goto S19 WITH
 lbg.S = 1;     "slave disable"
 ENDWITH;

else S18;

state S19:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
```

Appendix B. Bus Control Sequencer – ABEL Source (continued)

```
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S20 WITH
 soe.S =1;          "disable slave pls"
 ENDWITH;

state S20:
if !reset then S0 WITH
 lbg.S = 1;     "slave disable"
 gbe.R = 1;     "enable C40 global side"
 soe.S = 1;     "disable slave pls"
 moe.R = 1;     "enable master pls"
 grdy1.S = 1;   "not ready"
 ENDWITH;

else goto S0 WITH
 moe.R = 1;
 gbe.R = 1;
 ENDWITH;

@page
"make sure there are no undefined states"
state S21: goto S0;
state S22: goto S0;
state S23: goto S0;
state S24: goto S0;
state S25: goto S0;
state S26: goto S0;
state S27: goto S0;
state S28: goto S0;
state S29: goto S0;
state S30: goto S0;

"Power-Up"

 state S31:
goto S0 WITH
 lbg.S = 1;     "slave disable"
 lbg.R = 0;     "dummy err 6099"
 gbe.R = 1;     "enable C40 global side"
 gbe.S = 0;     "dummy err 6099"
 soe.S = 1;     "disable slave PLS"
 soe.R = 0;     "dummy err 6099"
 moe.R = 1;     "enable master pls"
 moe.S = 0;     "dummy err 6099"
 grdy1.S = 1;   "not ready"
```

Appendix B. Bus Control Sequencer – ABEL Source (continued)

```
 grdy1.R = 0;   "dummy err 6099"
 ENDWITH;

@page
" Test vectors will not work with beta version of
" Abel 4.3
test_vectors

([clk,reset,gprom,rwr,rrd,mwr,mrd,lbr,mwb,
 dsack1,dsack0,dedlk,lberr,glock,oe] ->
 [!sreg,grdy1,soe,moe,gbe,lbg])

[1,X,X,X,X,X,X,X,X,X,X,X,X,X,0]->[S31,X,X,X,X,X];"1 power up"
[0,X,X,X,X,X,X,X,X,X,X,X,X,X,0]->[S31,X,X,X,X,X];"2 power up"
[C,0,X,X,X,X,X,X,X,X,X,X,X,X,0]->[S0,1,1,0,0,1]; "3 resetstate"
[C,1,1,1,1,1,1,0,1,1,1,1,1,1,1,0]->[S1,1,1,0,0,1];"4 master read"
[C,1,1,1,1,1,1,0,1,0,1,1,1,1,1,0]->[S2,1,1,0,0,1];"5 mwb asserted"
[C,1,1,1,1,1,1,0,1,0,0,0,1,1,1,0]->[S3,0,1,0,0,1];"6 data acked"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S0,1,1,0,0,1];"7 ready for nxt"
[C,1,1,1,1,1,0,1,1,1,1,1,1,1,1,0]->[S1,1,1,0,0,1];"8 master write"
[C,1,1,1,1,1,1,0,1,0,1,1,1,1,1,0]->[S2,1,1,0,0,1];"9 mwb asserted"
[C,1,1,1,1,1,1,0,1,0,0,0,1,1,1,0]->[S3,0,1,0,0,1];"10 data acked"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S0,1,1,0,0,1];"11 rdy for nxt"
[C,1,1,1,0,1,1,1,1,1,1,1,1,1,1,0]->[S4,1,1,0,0,1];"12 reg read"
[C,1,1,1,0,1,1,1,1,1,0,1,1,1,1,0]->[S5,0,1,0,0,1];"13 data ackd"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S0,1,1,0,0,1];"14 rdy for nxt"
[C,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S8,1,1,0,0,1];"15 prom read"
[C,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S9,1,1,0,0,1];"16 prom read"
[C,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S10,1,1,0,0,1];"17 wait"
[C,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S11,1,1,0,0,1];"18 wait"
[C,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S12,0,1,0,0,1];"19 wait"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S0, 1,1,0,0,1];"20 rdy for nxt"
[C,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0]->[S16,1,1,1,1,1];"21slaverequest'
[C,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0]->[S17,1,0,1,1,1];"22 en slve pls"
[C,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0]->[S18,1,0,1,1,0];"23 slave grant"
[C,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0]->[S18,1,0,1,1,0];"24 slave aces"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S19,1,0,1,1,1];"25rescnd grant"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S20,1,1,1,1,1];"26disbl sl pls"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S0, 1,1,0,0,1];"27end sl acces"
[C,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0]->[S16,1,1,1,1,1];"29 deadlock"
[C,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0]->[S17,1,0,1,1,1];"30 en slve pls"
[C,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0]->[S18,1,0,1,1,0];"31 slave grant"
[C,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0]->[S18,1,0,1,1,0];"32 slave aces"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S19,1,0,1,1,1];"33rescnd grant"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S20,1,1,1,1,1];"34disbl sl pls"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0]->[S0, 1,1,0,0,1];"35end sl acces"

end U13C
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source

```
module          u14a
title           'C40 Bus Control
Revision        1.0
Part            CY7C335
Abel Version    4.3 using Cypress fitter
Project         TMS320C40 I/O Card '

U14a device 'p335';

"Inputs"
clk, reset                  pin 1,13;       "clock, reset"
mrd,mwr,rrd,rwr             pin 12,11,10,9; "decoded cycle"
mwb,lbr,gprom               pin 7,6,5 ;     "master/slave requests"
dedlk                       pin 4;        "m/s deadlock"
!dsack0,!dsack1,lberr       pin 28,27,2;  "cycle responses"
glock                       pin 3;        "C40 lock"
oe                          pin 14;        "output enable"

"Outputs"
pas  pin 18     istype 'invert,reg_RS';"68K address strobe"
ds   pin 16     istype 'invert,reg_RS';"68K data strobe"
rw   pin 17     istype 'invert,reg_RS';"68K read/write bar"
rmc  pin 15     istype 'invert,reg_RS';"68K read-mod-write"
siz0 pin 19     istype 'invert,reg_RS';"68K size 0"
siz1 pin 20     istype 'invert,reg_RS';"68K size 1"
fc1  pin 23     istype 'invert,reg_RS';"68K function 1"
fc2  pin 24     istype 'invert,reg_RS';"68K function 0"

"Sets"
cycle  = [gprom,rwr,rrd,mwr,mrd];  "cycle request"
ack    = [dsack1,dsack0];          "acknowledge"
output = [pas,ds,rw,rmc,siz0,siz1,fc1,fc2]; "68K ouputs"

"State Description"
P4,P3,P2,P1  node 34,33,32,31 istype 'reg';
P0           pin  25          istype 'reg,invert';
sreg = [P4,P3,P2,P1,!P0];
 S0 = [0,0,0,0,0];
 S1 = [0,0,0,0,1];
 S2 = [0,0,0,1,0];
 S3 = [0,0,0,1,1];
 S4 = [0,0,1,0,0];
 S5 = [0,0,1,0,1];
 S6 = [0,0,1,1,0];
 S7 = [0,0,1,1,1];
 S8 = [0,1,0,0,0];
 S9 = [0,1,0,0,1];
 S10 = [0,1,0,1,0];
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
 S11 = [0,1,0,1,1];
 S12 = [0,1,1,0,0];
 S13 = [0,1,1,0,1];
 S14 = [0,1,1,1,0];
 S15 = [0,1,1,1,1];
 S16 = [1,0,0,0,0];
 S17 = [1,0,0,0,1];
 S18 = [1,0,0,1,0];
 S19 = [1,0,0,1,1];
 S20 = [1,0,1,0,0];
 S21 = [1,0,1,0,1];
 S22 = [1,0,1,1,0];
 S23 = [1,0,1,1,1];
 S24 = [1,1,0,0,0];
 S25 = [1,1,0,0,1];
 S26 = [1,1,0,1,0];
 S27 = [1,1,0,1,1];
 S28 = [1,1,1,0,0];
 S29 = [1,1,1,0,1];
 S30 = [1,1,1,1,0];
 S31 = [1,1,1,1,1];

"Misc"
rwmem  pin 26 istype 'reg_RS,invert'; "r/w flag"

H,L,X,C,Z = 1,0,.X.,.C.,.Z.;

equations
output.OE = !oe;          "set output enable"
output.CLK = clk;         "clock the output regs"
sreg.CLK = clk;           "and state regs"
rwmem.CLK = clk;          "and r/w store"

@page

state_diagram sreg
state S0:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;   "flag for mem"
 rmc.S = 1;     "no rmc
 siz0.R = 1     "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for supervisory"
 fc2.S = 1;     "data access"
 ENDWITH;
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
else if (!mrd & !rwmem & lbr) then S1 WITH    "master read"
 rw.S = 1;           "assert read/write"
 rwmem.S =1;
 ENDWITH;

else if (!mrd & rwmem & lbr) then S2 WITH     "master read"
 pas.R = 1;          "assert pas"
 ds.R = 1;           "and ds"
 ENDWITH;

else if (!mwr & rwmem & lbr) then S8 WITH     "master write"
  rw.R = 1;          "assert r/w"
 rwmem.R = 1;
 ENDWITH;

else if (!mwr & !rwmem & lbr) then S9 WITH    "master write"
pas.R = 1;           "assert pas only"
 ENDWITH;

else if (!rrd & !rwmem & lbr) then S16 WITH   "reg read"
 rw.S = 1;           "assert r/w"
 rwmem.S = 1;
 ENDWITH;

else if (!rrd & rwmem & lbr) then S17 WITH    "reg read"
 pas.R = 1;          "assert pas"
 ds.R = 1;           "and ds"
 ENDWITH;

else if (!rwr & rwmem & lbr) then S24 WITH    "reg write"
rw.R = 1;
 rwmem.R = 1;
 ENDWITH;

else if (!rwr & !rwmem & lbr) then S25 WITH
 pas.R = 1;          "assert pas only"
 ENDWITH;

else S0 WITH
 pas.S = 1;      "no strobe"
 ds.S= 1;        "no strobe"
 rw.S = 1;       "read"
 rwmem.S = 1;    "flag for mem"
 rmc.S = 1;      "no rmc"
 siz0.R = 1;     "set for"
 siz1.R = 1;     "32-bit xfers"
 fc1.R = 1;      "set for supervisory"
 fc2.S = 1;      "data access"
 ENDWITH;
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
@page
"Master Read"

state S1:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;   "flag for mem"
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for super"
 fc2.S = 1;     "data access"
 ENDWITH;

else S2 WITH
 pas.R = 1;
 ds.R = 1;
 ENDWITH;

state S2:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;   "flag for mem"
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for super"
 fc2.S = 1;     "data access"
 ENDWITH;

else if !mwb then S3;          "wait for !mwb"

else S2;

state S3:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;   "flag for mem"
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for supervisory"
 fc2.S = 1;     "data access"
 ENDWITH;
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
else if ((!dsack1 & !dsack0) # !lberr) then S4
" WITH
" grdy1.R = 1"
ENDWITH;

else S3;

state S4:
goto S0 WITH
 pas.S = 1;
 ds.S = 1;
 ENDWITH;

@page
"Master Write"

state S8:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;
 rmc.S = 1; "  no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for supervisory"
 fc2.S = 1;     "data access"
 ENDWITH;

else S9 WITH
 pas.R = 1;
 ENDWITH;

state S9:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for supervisory"
 fc2.S = 1;     "data access"
 ENDWITH;

else S10 WITH
 ds.r = 1;
 ENDWITH;
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
state S10:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;      "no strobe"
 ds.S= 1;        "no strobe"
 rw.S = 1;       "read"
 rwmem.S = 1;
 rmc.S = 1;      "no rmc"
 siz0.R = 1;     "set for"
 siz1.R = 1;     "32-bit xfers"
 fc1.R = 1;      "set for super"
 fc2.S = 1;      "data access"
 ENDWITH;

else if !mwb then S11;
else S10;

state S11:
if (!reset # !dedlk) then S0 WITH
 pas.S = 1;      "no strobe"
 ds.S= 1;        "no strobe"
 rw.S = 1;       "read"
 rwmem.S = 1;
 rmc.S = 1;      "no rmc"
 siz0.R = 1;     "set for"
 siz1.R = 1;     "32-bit xfers"
 fc1.R = 1;      "set for supervisory"
 fc2.S = 1;      "data access"
 ENDWITH;

else if ((!dsack1 & !dsack0) # !lberr) then S12;

else S11;

state S12:
goto S0 WITH
 pas.S = 1;
 ds.S = 1;
 ENDWITH;

@page
"Register Read"

state S16:
if !reset then S0 WITH
 pas.S = 1;      "no strobe"
 ds.S= 1;        "no strobe"
 rw.S = 1;       "read"
 rwmem.S = 1;
 rmc.S = 1;      "no rmc"
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for super"
 fc2.S = 1;     "data access"
 ENDWITH;

else S17 WITH
 pas.R = 1;
 ds.R =1;
 ENDWITH;

state S17:
if !reset then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for super"
 fc2.S = 1;     "data access"
 ENDWITH;

else if !dsack1 then S18
"WITH
" grdy1.R = 1
" ENDWITH;

else S17;

state S18:
goto S0 WITH
 pas.S = 1;
 ds.S = 1;
 ENDWITH;

@page
"Register Write"

state S24:
if !reset then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for supervisory"
 fc2.S = 1;     "data access"
ENDWITH;

else S25 WITH
 pas.R = 1;
 ENDWITH;

state S25:
if !reset then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for supervisory"
 fc2.S = 1;     "data access"
 ENDWITH;

else S26 WITH
 ds.r = 1;
 ENDWITH;

state S26:
if !reset then S0 WITH
 pas.S = 1;     "no strobe"
 ds.S= 1;       "no strobe"
 rw.S = 1;      "read"
 rwmem.S = 1;
 rmc.S = 1;     "no rmc"
 siz0.R = 1;    "set for"
 siz1.R = 1;    "32-bit xfers"
 fc1.R = 1;     "set for supervisory"
 fc2.S = 1;     "data access"
 ENDWITH;

else if !dsack1 then S27;

else S26;

state S27:
goto S0 WITH
 pas.S = 1;
 ds.S = 1;
 ENDWITH;
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
@page
"Power-Up"

state S31:
goto S0 WITH
 pas.S = 1;     "no strobe"
 pas.R = 0;     "error 6099 fix"
 ds.S= 1;       "no strobe"
 ds.R= 0;       "error 6099 fix"
 rw.S = 1;      "read"
 rwmem.S = 1;
 rw.R = 0;      "error 6099 fix"
 rmc.S = 1;     "no rmc"
 rmc.R = 0;     "error 6099 fix"
 siz0.R = 1;    "set for"
 siz0.S = 0;    "error 6099 fix"
 siz1.R = 1;    "32-bit xfers"
 siz1.S = 0;    "error 6099 fix"
 fc1.R = 1;     "set for supervisory"
 fc1.S = 0;     "error 6099 fix"
 fc2.S = 1;     "data access"
 fc2.R = 0;     "error 6099 fix"
ENDWITH;

@page
test_vectors

([clk,reset,gprom,rwr,rrd,mwr,mrd,lbr,mwb,
 dsack1,dsack0,dedlk,lberr,glock,oe] ->
[!sreg,rwmem,fc2,fc1,siz1,siz0,rmc,rw,ds,pas])

"1 power up"
[1,X,X,X,X,X,X,X,X,X,X,X,X,0] -> [S31,X,X,X,X,X,X,X,X,X];
"2 power up"
[0,X,X,X,X,X,X,X,X,X,X,X,X,0] -> [S31,X,X,X,X,X,X,X,X,X];
"3 reset state"
[C,0,X,X,X,X,X,X,X,X,X,X,X,0] -> [S0, 1,1,0,0,0,1,1,1,1];
"4 master read"
[C,1,1,1,1,1,0,1,1,1,1,1,1,1,0] -> [S2, 1,1,0,0,0,1,1,0,0];
"5 mwb  asserted"
[C,1,1,1,1,1,0,1,0,1,1,1,1,1,0] -> [S3, 1,1,0,0,0,1,1,0,0];
"6 data acked"
[C,1,1,1,1,1,0,1,0,0,0,1,1,1,0] -> [S4, 1,1,0,0,0,1,1,0,0];
"7 ready for  nxt"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] -> [S0, 1,1,0,0,0,1,1,1,1];
"8 master  write"
[C,1,1,1,1,0,1,1,1,1,1,1,1,1,0] -> [S8, 0,1,0,0,0,1,0,1,1];
"9 assert pas"
[C,1,1,1,1,0,1,1,1,1,1,1,1,1,0] -> [S9, 0,1,0,0,0,1,0,1,0];
```

Appendix C. Master Cycle Generation Sequencer – ABEL Source (continued)

```
"10 assert ds"
[C,1,1,1,1,0,1,1,1,1,1,1,1,1,0] -> [S10,0,1,0,0,0,1,0,0,0];
"11 mwb"
[C,1,1,1,1,0,1,1,0,1,1,1,1,1,0] -> [S11,0,1,0,0,0,1,0,0,0];
"12 data ackd"
[C,1,1,1,1,0,1,1,0,0,0,1,1,1,0] -> [S12,0,1,0,0,0,1,0,0,0];
"13 ready for  next"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] -> [S0, 0,1,0,0,0,1,0,1,1];
"14 reg read"
[C,1,1,1,0,1,1,1,1,1,1,1,1,1,0] -> [S16,1,1,0,0,0,1,1,1,1];
"15 assert  strobes"
[C,1,1,1,0,1,1,1,1,1,1,1,1,1,0] -> [S17,1,1,0,0,0,1,1,0,0];
"16 data ackd"
[C,1,1,1,0,1,1,1,1,0,1,1,1,1,0] -> [S18,1,1,0,0,0,1,1,0,0];
"17 ready for  nxt"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] -> [S0, 1,1,0,0,0,1,1,1,1];
"18 reg write"
[C,1,1,0,1,1,1,1,1,1,1,1,1,1,0] -> [S24,0,1,0,0,0,1,0,1,1];
"19 assert pas"
[C,1,1,0,1,1,1,1,1,1,1,1,1,1,0] -> [S25,0,1,0,0,0,1,0,1,0];
"20 assert ds"
[C,1,1,0,1,1,1,1,1,1,1,1,1,1,0] -> [S26,0,1,0,0,0,1,0,0,0];
"21 data ackd"
[C,1,1,0,1,1,1,1,1,0,1,1,1,1,0] -> [S27,0,1,0,0,0,1,0,0,0];
"22 ready for  next"
[C,1,1,1,1,1,1,1,1,1,1,1,1,1,0] -> [S0, 0,1,0,0,0,1,0,1,1];

end u14a
```

Appendix D. Slave Cycle Generation Sequencer – ABEL Source

```
module          u15a
title           'C40 Bus Control
Revision        1.0
Part            CY7C335
Abel Version    4.30
Project         TMS320C40 I/O Card '

U15a device 'p335';

"Inputs"
clk, reset      pin 1,13;  "clock, reset"
pas,ds          pin 12,11;  "address,data strobe"
rw,rmc          pin 10,9;  "read/write strobes"
siz0,siz1       pin 7,6;  "bus sizing"
fc0,fc1,fc2     pin 5,4,3;"function codes"
lbg             pin 2;   "local bus grant"
oe              pin 14;   "output enable"

"Outputs"
dsack0    pin 15    istype 'invert,reg_RS'; "data ack 0"
dsack1    pin 17    istype 'invert,reg_RS'; "data ack 1"
lberr     pin 19    istype 'invert,reg_RS'; "bus error"
gstrb0    pin 23    istype 'invert,reg_RS'; "C40 mem strobe"
grw0      pin 25    istype 'invert,reg_RS'; "C40 read/write"

"Sets"
size   = [siz1,siz0];    "size"
func   = [fc2,fc1,fc0];  "function"
output = [grw0,gstrb0,lberr,dsack1,dsack0];

"State Description"
P3,P2,P1,P0    node 34,33,32,31    istype 'reg';
sreg = [P3,P2,P1,P0];
 S0 = [0,0,0,0];
 S1 = [0,0,0,1];
 S2 = [0,0,1,0];
 S3 = [0,0,1,1];
 S4 = [0,1,0,0];
 S5 = [0,1,0,1];
 S6 = [0,1,1,0];
 S7 = [0,1,1,1];
 S8 = [1,0,0,0];
 S9 = [1,0,0,1];
 S10 = [1,0,1,0];
 S11 = [1,0,1,1];
 S12 = [1,1,0,0];
 S13 = [1,1,0,1];
 S14 = [1,1,1,0];
 S15 = [1,1,1,1];
```

Appendix D. Slave Cycle Generation Sequencer – ABEL Source (continued)

```
"Misc"
!rwmem pin  27 istype 'reg_RS,invert'; "r/w flag"
H,L,X,C,Z = 1,0,.X.,.C.,.Z.;

equations
output.OE = !oe; "set output enable"
output.CLK = clk;"clock the output regs"
sreg.CLK = clk;  "and state regs"
rwmem.CLK = clk; "and r/w store"

@page
state_diagram sreg
state S0:

if (!reset) then S0 WITH
 dsack0.S = 1; "deassert"
 dsack1.S = 1; "all"
 lberr.S = 1;  "strobes"
 gstrb0.S = 1; "deassert C40"
 grw0.R = 1;   "strobe, read"
 rwmem.S = 1;  "set to read"
 ENDWITH;

else if (!lbg) then S1;

else S0 WITH
 dsack0.S = 1; "deassert"
 dsack1.S = 1; "all"
 lberr.S = 1;  "strobes"
 gstrb0.S = 1; "deassert C40"
 grw0.R = 1;   "strobe, read"
 rwmem.S = 1;  "set to read"
 ENDWITH;

@page
"Sort Slave Request"
state S1:

"Reset"
if (!reset) then S0 WITH
 dsack0.S = 1; "deassert"
 dsack1.S = 1; "all"
 lberr.S = 1;  "strobes"
 gstrb0.S = 1; "deassert C40"
 grw0.R = 1;   "strobe, read"
 rwmem.S = 1;  "set to read"
 ENDWITH;
```

Appendix D. Slave Cycle Generation Sequencer – ABEL Source (continued)

```
"32-Bit Read"
else if (!pas & !ds & rw & !rwmem & !siz0 & !siz1) then S2 WITH
grw0.S = 1;
 rwmem.S = 1;
 ENDWITH;

else if (!pas & !ds & rw & rwmem & !siz0 & !siz1) then S3 WITH
gstrb0.R = 1;
 ENDWITH;

"32-Bit Write"
else if (!pas & !ds & !rw & rwmem & !siz0 & !siz1) then S2 WITH
grw0.R = 1;
 rwmem.R = 1;
 ENDWITH;

else if (!pas & !ds & !rw & !rwmem & !siz0 & !siz1) then S3 WITH
gstrb0.R = 1;
 ENDWITH;

"Illegal Access (non-32 bit access)"
 else if (!pas & !ds & (rw # !rw) & (siz0 # siz1)) then S9 WITH
lberr.R = 1;
 ENDWITH;

else S1;

@page
"32-Bit Read/Write"

state S2:
goto S3 WITH
 gstrb0.R = 1;
 ENDWITH;

state S3:
goto S4 WITH
 dsack0.R = 1;
 dsack1.R = 1;
 ENDWITH;

state S4:
if pas then S0 WITH
 dsack0.S = 1;
 dsack1.S = 1;
 gstrb0.S = 1;
 ENDWITH;
```

Appendix D. Slave Cycle Generation Sequencer – ABEL Source (continued)

```
else S4;

@page
"Illegal Access"

state S9:
if pas then S0 WITH
 lberr.S = 1;
 ENDWITH

@page
"Power-Up"

state S15:
goto S0 WITH
 dsack0.S = 1; "no ack"
 dsack0.R = 0; "error 6099 fix"
 dsack1.S = 1; "no ack"
 dsack1.R = 0; "error 6099 fix"
 rwmem.S = 1;  "r/w mem"
 rwmem.R = 0;  "error 6099 fix"
 lberr.S = 1;  "no bus error"
 lberr.R = 0;  "error 6099 fix"
 gstrb0.S = 1; "no strobe"
 grw0.S = 1;   "read"
 ENDWITH;

@page
test_vectors

([clk,reset,pas,ds,rw,rmc,siz0,siz1,fc0,fc1,fc2,lbg,oe] ->
[!sreg,rwmem,dsack0,dsack1,lberr,gstrb0,grw0])

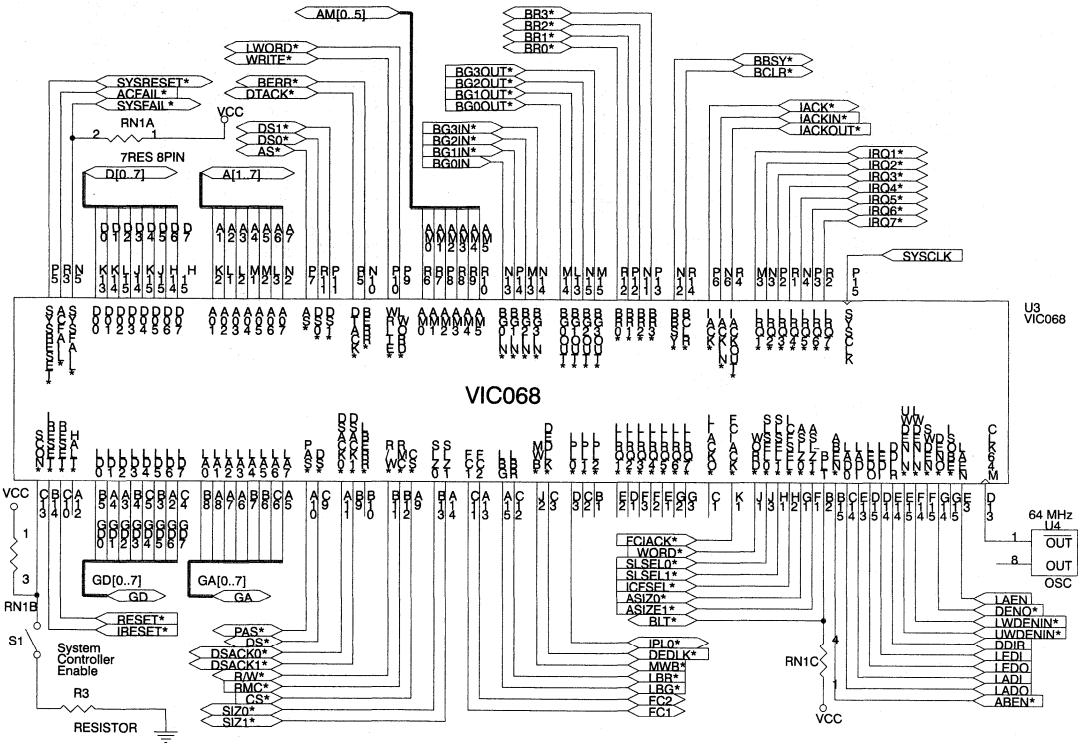
[1,X,X,X,X,X,X,X,X,X,X,X,0]->[S15,X,X,X,X,X,X];"1 power up"
[0,X,X,X,X,X,X,X,X,X,X,X,0]->[S15,X,X,X,X,X,X];"2 power up"
[C,0,X,X,X,X,X,X,X,X,X,X,0]->[S0, 1,1,1,1,1,1];"3 reset state"
[C,1,1,1,1,1,1,1,X,X,X,0,0]->[S1, 1,1,1,1,1,1];"4 slave read,lbg"
[C,1,0,1,1,1,0,0,X,X,X,0,0]->[S1, 1,1,1,1,1,1];"5 pas asserted"
[C,1,0,0,1,1,0,0,X,X,X,0,0]->[S3, 1,1,1,1,0,1];"6 and ds,strobe"
[C,1,0,0,1,1,0,0,X,X,X,0,0]->[S4, 1,0,0,1,0,1];"7 ack"
[C,1,0,0,1,1,0,0,X,X,X,0,0]->[S4, 1,0,0,1,0,1];"8 wtfor pas rel"
[C,1,1,1,1,1,1,1,X,X,X,1,0]->[S0, 1,1,1,1,1,1];"9 dne, rel gstrb"
[C,1,1,1,0,1,1,1,X,X,X,0,0]->[S1, 1,1,1,1,1,1];"10 slav wrte,lbg"
[C,1,0,1,0,1,0,0,X,X,X,0,0]->[S1, 1,1,1,1,1,1];"11 pas assert"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S2, 0,1,1,1,1,0];"12 and ds"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S3, 0,1,1,1,0,0];"13 asert strob"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S4, 0,0,0,1,0,0];"14 ack"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S4, 0,0,0,1,0,0];"15 wtfor pas rel"
```

Appendix D. Slave Cycle Generation Sequencer – ABEL Source (continued)

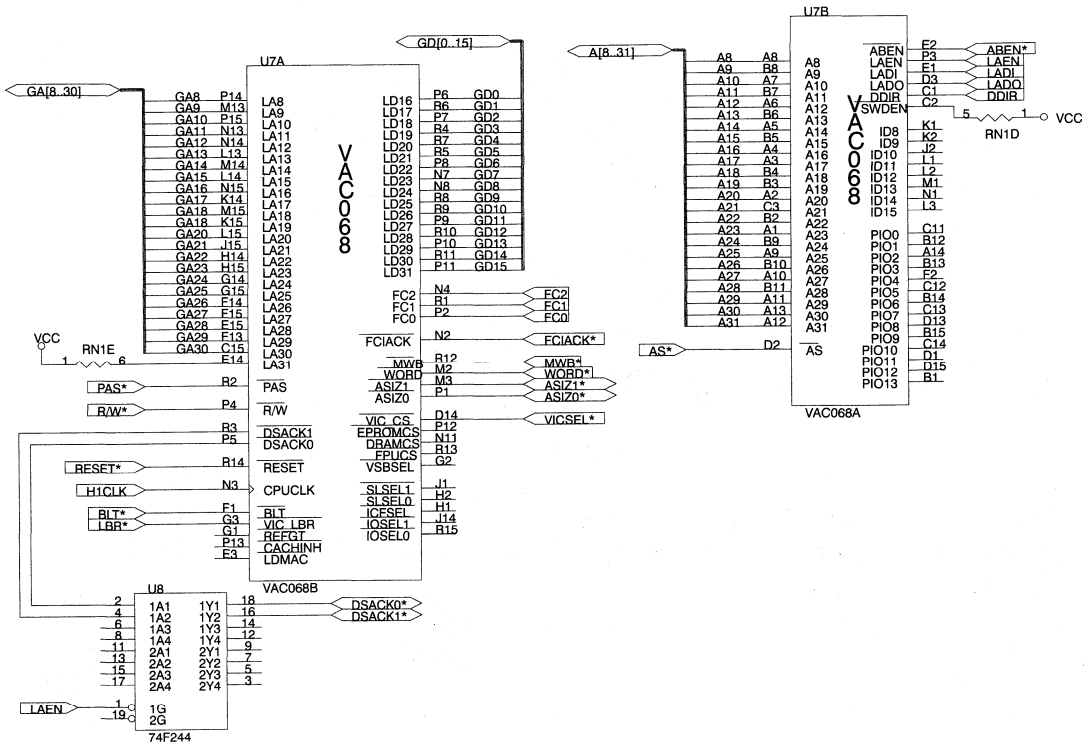
```
[C,1,1,1,1,1,1,1,X,X,X,1,0]->[S0, 0,1,1,1,1,0];"16done,rel gstrb"
[C,1,1,1,1,1,1,1,X,X,X,0,0]->[S1, 0,1,1,1,1,0];"17 slav read,lbg"
[C,1,0,1,1,1,0,0,X,X,X,0,0]->[S1, 0,1,1,1,1,0];"18 pas asserted"
[C,1,0,0,1,1,0,0,X,X,X,0,0]->[S2, 1,1,1,1,1,1];"19 & ds,r/w asrt"
[C,1,0,0,1,1,0,0,X,X,X,0,0]->[S3, 1,1,1,1,0,1];"20 and strobe"
[C,1,0,0,1,1,0,0,X,X,X,0,0]->[S4, 1,0,0,1,0,1];"21 ack"
[C,1,0,0,1,1,0,0,X,X,X,0,0]->[S4, 1,0,0,1,0,1];"22 wtfor pas rel"
[C,1,1,1,1,1,1,1,X,X,X,1,0]->[S0, 1,1,1,1,1,1];"23done,rel gstrb"
[C,1,1,1,1,1,1,1,X,X,X,0,0]->[S1, 1,1,1,1,1,1];"24 bad acess,lbg"
[C,1,0,1,1,1,0,1,X,X,X,0,0]->[S1, 1,1,1,1,1,1];"25 pas asserted"
[C,1,0,0,1,1,0,1,X,X,X,0,0]->[S9, 1,1,1,0,1,1];"26 & ds, error"
[C,1,0,0,1,1,0,1,X,X,X,0,0]->[S9, 1,1,1,0,1,1];"27 wtfor pas rel"
[C,1,1,1,1,1,1,1,X,X,X,1,0]->[S0, 1,1,1,1,1,1];"28done,rel lberr"
[C,1,1,1,0,1,1,1,X,X,X,0,0]->[S1, 1,1,1,1,1,1];"29 slv write,lbg"
[C,1,0,1,0,1,0,0,X,X,X,0,0]->[S1, 1,1,1,1,1,1];"30 pas asserted"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S2, 0,1,1,1,1,0];"31 and ds"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S3, 0,1,1,1,0,0];"32 assert strobe"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S4, 0,0,0,1,0,0];"33 ack"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S4, 0,0,0,1,0,0];"34 wtfor pas rel"
[C,1,1,1,1,1,1,1,X,X,X,1,0]->[S0, 0,1,1,1,1,0];"35done,rel gstrb"
[C,1,1,1,0,1,1,1,X,X,X,0,0]->[S1, 0,1,1,1,1,0];"36 slav wrte,lbg"
[C,1,0,1,0,1,0,0,X,X,X,0,0]->[S1, 0,1,1,1,1,0];"37 pas asserted"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S3, 0,1,1,1,0,0];"38 & ds,asrt str"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S4, 0,0,0,1,0,0];"39 ack"
[C,1,0,0,0,1,0,0,X,X,X,0,0]->[S4, 0,0,0,1,0,0];"40 wtfor pas rel"
[C,1,1,1,1,1,1,1,X,X,X,1,0]->[S0, 0,1,1,1,1,0];"41done,rel gstrb"

end u15a
```

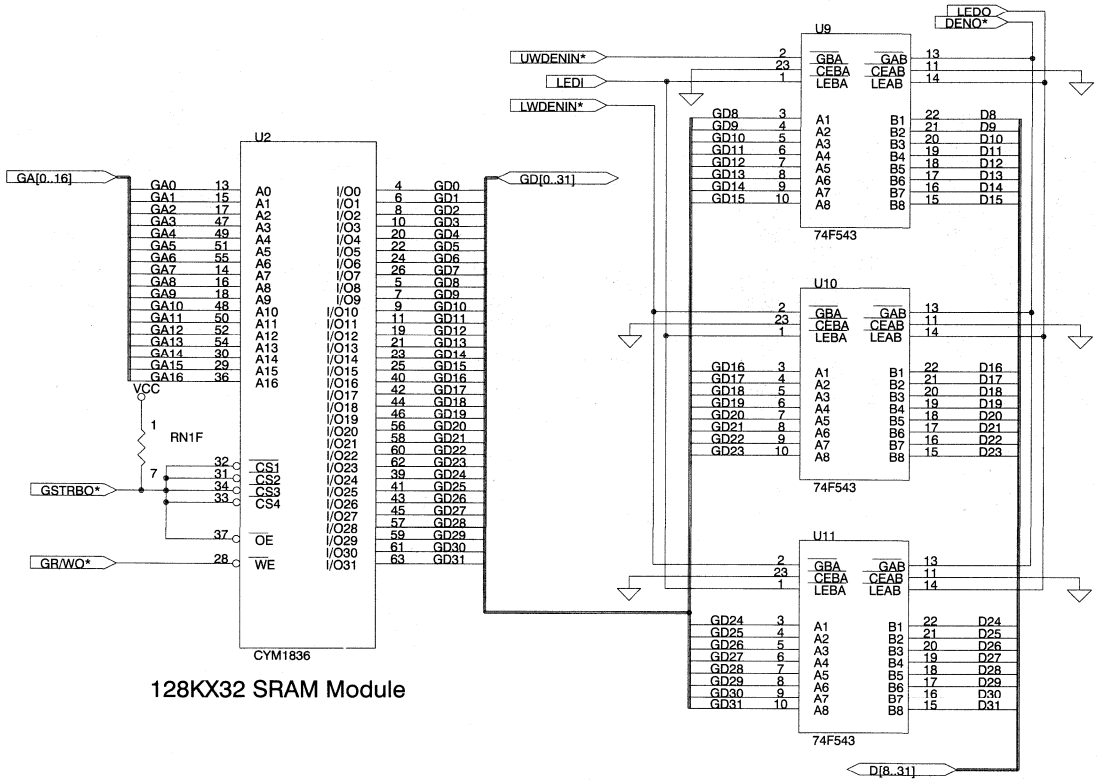
Appendix E. Schematics



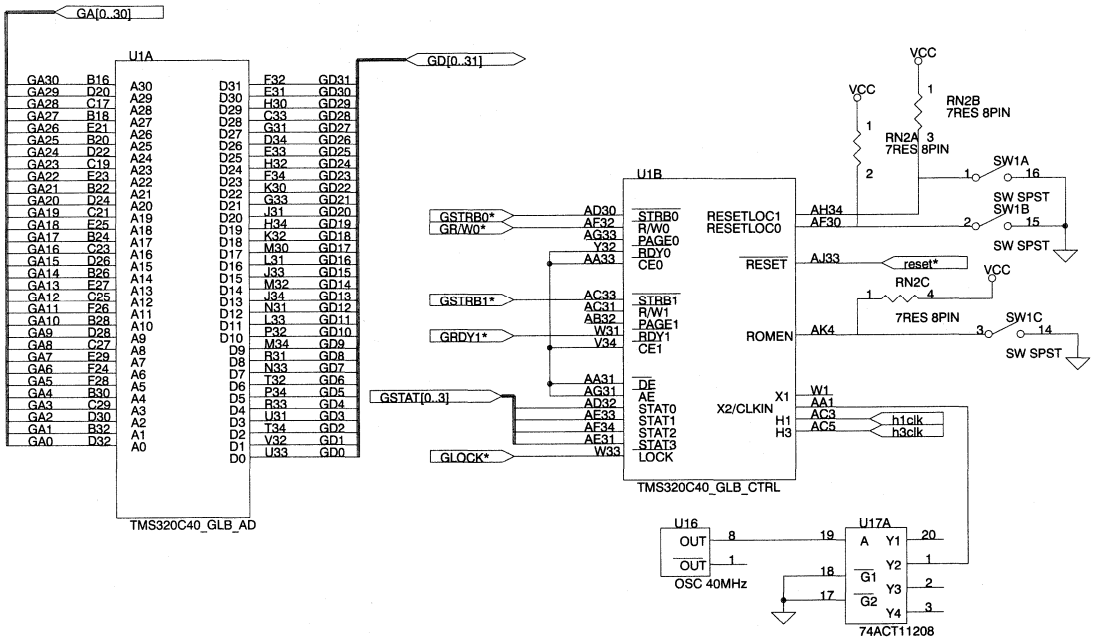
Appendix E. Schematics (continued)



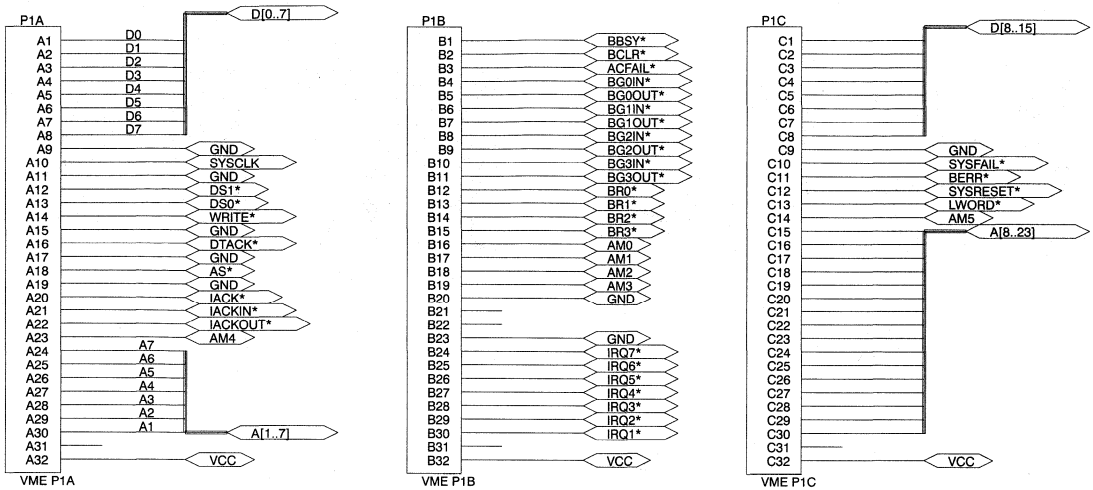
Appendix E. Schematics (continued)



Appendix E. Schematics (continued)

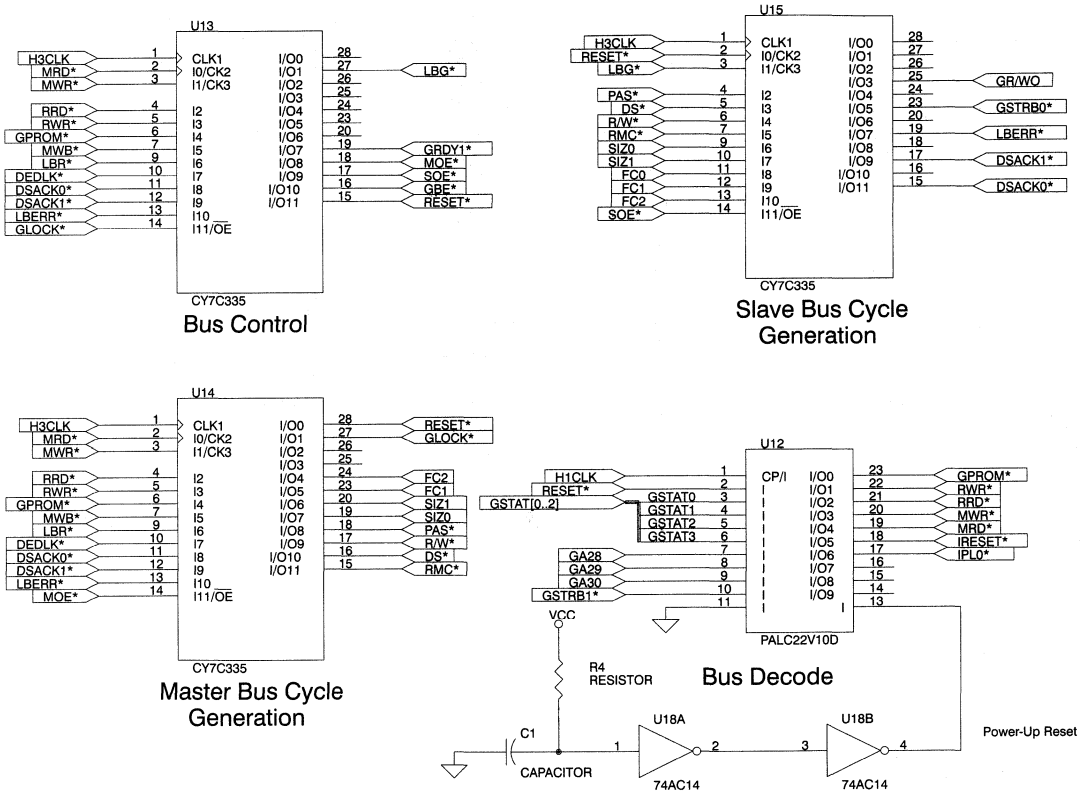


Appendix E. Schematics (continued)



VME P1 CONNECTOR

Appendix E. Schematics (continued)



Document #: BUS0001–0993

Software Considerations for the VIC64

Introduction

This application note provides the VIC64 software developer with proven tips and examples for both configuring and operating the VIC64. The software described here is based on a SPARC-based VMEbus card utilizing a VIC64. This board was developed within Cypress Semiconductor as a test/evaluation vehicle for the VIC64 and the CY7C964.

This application note also discusses the configuration of the CY7C964 VMEbus address compare functions.

Although this application note specifically addresses the VIC64, virtually everything in this application note could also be applied to the VIC068A. VIC64-only features are flagged to notify the reader of items that are not applicable to the VIC068A.

The source files vic.h, cyb675.h, and blt_cmd.c, which are described in this application note, are available through the Cypress Semiconductor BBS (Bulletin Board System).

Related Documents

The reader may also wish to consult the following documents for additional information:

- *VIC068A/VAC068A User's Guide*
- *VIC64/CY7C964 Design Notes*

These documents are available through your local Cypress Semiconductor field sales office.

Hardware Overview

The examples in this application note are based on an actual design of a SPARC-based VIC64 evaluation VMEbus board developed by Cypress Semiconductor. The following paragraphs provide background for this hardware platform. Contact your local field applications engineer regarding specific hardware information on this board.

The CYB675 Evaluation Board

The CYB675 is the unofficial name given to Cypress's VIC64 evaluation board. Features include:

- Cypress's CY7C611 embedded SPARC microprocessor
- Floating-point support using Cypress's CY7C602
- 64 Kbytes to 4 Mbytes of private SRAM
- 64 Kbytes to 2 Mbytes of shared SRAM
- 512 Kbytes of EPROM for the embedded monitor program
- Performs D64 VMEbus transfers utilizing VIC64 and CY7C964 devices
- MC68681 DUART
- 2 Kbytes of non-volatile storage
- Real-time clock

CYB675 Local Control Register (LCR)

The CYB675 contains a single 32-bit, dual-purpose control register. When read, this register provides the memory size of the SIMM sockets as shown in *Table 1*.

Table 1. LCR Read Fields

Bits	Socket
bits 0,1	SIMM socket 1 size (private)
bits 2,3	SIMM socket 2 size (private)
bits 4,5	SIMM socket 3 size (private)
bits 6,7	SIMM socket 4 size (private)
bits 8,9	SIMM socket 5 size (shared)
bits 10,11	SIMM socket 6 size (shared)

The two bits for each SIMM contain one of the codes shown in *Table 2*.

Table 2. SIMM Size Codes

Code	Size
00	1M-byte SIMM
01	256K-byte SIMM
10	64K-bytes SIMM
11	Socket empty

When written, this register provides control over the the resources shown in *Table 3*.

Table 3. LCR Write Fields

Bits	Function
bits 0–15:	LEDs (lit when bit is clear)
bits 16,17:	VIC64 reset
bits 18–28:	VMEbus address (A31:21)
bits 29,30:	VMEbus address size (ASIZ0–1)
bit 31:	VMEbus data port size (WORD*)

Bits 0–15 provide control of 16 LEDs located on the edge of the board. When a bit is cleared, the corresponding LED is lit.

Bits 16 and 17 provide control over the reset operations of the VIC64. When bit 16 is cleared, CYB675 state logic asserts the IRESET* signal of the VIC64. When bit 17 is cleared, state logic asserts the IPL0* signal of the VIC64, issuing a global reset to the VIC64.

Bits 18–28 provide control over the most-significant eleven VMEbus address lines. Prior to a VMEbus access, this bit field is loaded with the most-significant eleven address bits. An access is then made to a predefined address (VME_BASE_ADRS on the CYB675) with the least-significant 21 VMEbus address lines obtained from the physical address of the transaction.

Bits 29 and 30 control the VIC64 ASIZ0/1 signals respectively. These signals tell the VIC64 what address size to use.

Bit 31 controls the WORD* signal line. When clear, the VIC64 performs D16 VMEbus accesses; when set, D32.

Software Considerations

SPARCmon™

The embedded monitor program used on the CYB675 is *SPARCmon*. *SPARCmon* is a commercial product available from Sun Microsystems. *SPARCmon* consists of source code modules for initialization, trap handling, floating-point support, process control, remote debugging, I/O, and a main command interpreter. Board-specific code such as board initialization, test, and additional commands are incorporated into *SPARCmon* separately. This application note does not address the specifics of *SPARCmon*, only board-specific details as it relates to the CYB675 and the VIC64.

CYB675 Boot-Up

The flow of initialization for booting the CYB675 is described in the following sections.

Disable Traps

Traps are disabled until resources exist to service them.

Initialize 7C611 Window Invalid Mask (WIM) and Trap Base Register (TBR)

See the *SPARC RISC User's Guide* for details on these registers.

Reset the VIC64

This is discussed in detail later in this application note.

Test First 64 Kbytes of Private Memory

This provides us with tested memory for temporary storage to perform subsequent boot tasks.

Set Up Initial Stack Frame Pointer and Enable Traps

With the first 64 Kbytes of memory tested, we may now service traps. The trap vector table is located initially in EPROM at address $0.

Initialize I/O

This consists of setting up I/O tables, structures and the DUART itself.

Perform Board Diagnostics

The remainder of the board is checked, including

- — EPROM checksum
- — NVRAM checksum
- — Determining amount of SRAM installed
- — Testing remaining private SRAM
- — Testing the shared SRAM
- — Testing the NVRAM
- — Testing the VIC64 (discussed later)
- — Testing the DUART
- — Configuring board local memory map

Local memory map is created with regions for

- — Monitor variables (DATA)
- — Uninitialized monitor variables (BSS)
- — The relocated trap table
- — User memory area
- — User stack (STACK) area

Clear User Memory Areas

The user areas are "cleared" to a predefined value.

Relocate the Trap Table in EPROM to SRAM

This speeds up trap table accesses and makes the table modifiable. The TBR is adjusted after the table is moved.

Configure VIC64

This is discussed in detail later in this application note.

VIC64 Initialization and Test

VIC64 Register Accesses

All of the VIC64's internal registers are 8 bits wide but occupy 32 bits of address space. Specific address and size information must be presented to the VIC64 in order for the VIC64 to accept the register access.

When the VIC64 has been selected for a register access (CS*, PAS*, and DS* are asserted to the VIC64), the VIC64 checks the SIZ1/0 and LA[1:0] signals to insure proper byte orientation. This is because the VIC64 is only connected to the lower 8 data lines of the local data bus and the data must be aligned as such.

Table 4 shows the valid combinations of SIZ1/0 and LA[1:0] that must be present for the VIC64 to accept the register access. The VIC64 mimics the Motorola CISC processors in that the SIZ and LA combinations for it are the same as for the VIC64. The SIZ codes for the CY7C611 are not the same and translation circuitry is required.

Table 4. VIC64/068 D(7:0) Data Alignment

SIZ1	SIZ0	LA1	LA0	Size
0	1	1	1	Byte
1	0	1	0	Word
0	0	0	0	Longword
1	1	1	1	3-Byte

If *Table 4* is not satisfied, the VIC64 ignores the attempted cycle by not reading or writing the information and not acknowledging the cycle (does not assert DSACKi*).

VIC64 Reset

The CYB675 issues a power-on reset to the VIC64 via the LCR. The LCR contains two bits for VIC64 reset. Bit 16 controls the assertion of IRESET* for the purposes of performing a internal reset. Bit 17 controls the assertion of IPL0*, which is used in conjunction with IRESET*, to perform a global reset. The VIC64 requires that a global reset be issued at power-up. The SPARC assembler code in *Figure 1* performs a VIC64 global reset.

This routine is written in assembler language because it must be a "leaf" routine. That is, it must not use the stack in any way since no stack exists yet. Calls from a high-level language or calling an additional routine would almost certainly use the stack.

Notice that the VIC64 is reset in stages. First the IRESET* signal is asserted to the VIC64 by clearing bit 16 of the LCR. The next instruction clears bit 17 to assert IPL0*. The reason that these are performed in separate instructions is that sufficient time must be allowed for the assertion of IRESET* to switch the IPL0* from an output to an input. Next, the IPL0* signal is removed, then the IRESET* signal is removed, in separate instructions. This is done to insure that the VIC64 200-ms reset timeout is observed. If they were removed simultaneously, this timeout may not be observed and the reset would complete immediately. Refer to section 12.1 of the *VIC068/VAC068 User's Guide* for more details on VIC reset.

VIC64 Test

To determine if the VIC64 is present and has been reset properly, the VIC64 test routine performs write-read-verify cycles to the VIC64 ICR0–5 registers. At this time, the VIC64 version register is read to determine the mask revision. The mask register reads $00, and any VIC64 values above $F0 indicate a VIC068 is installed. This may or may not be acceptable for specific applications.

VIC64 Configuration

The configuration of the VIC64 is accomplished by writing the VIC64 registers to desired values. The CYB675 stores these predetermined values as a structure located in the NVRAM at boot-up. The VIC64 configuration routine reads these values and stores them into the appropriate VIC64 registers. This way, the configuration of the VIC64 is not hard-coded and may be modified by simply changing the values in NVRAM and calling a VIC64 configuration routine.

VIC64 Address Spaces

In VMEbus systems, each VMEbus board typically has its own unique address spaces within the total 4-Gbyte VMEbus addressing range. These regions may consist of various sub-regions including:

- A32, A24, and/or A16 regions
- D32 and/or D16 regions
- Interprocessor communication regions

In addition to the VMEbus address spaces, the local processor within each board works with a local address space that may include:

- Private memory
- Shared memory (shared with the VMEbus)
- UARTs
- Interrupt acknowledge
- Board control registers

```
#include <CYB675.h>                    ; Needed for LCR pointer

set LOCAL_CONTROL_BASE_ADRS, %l6       ; This symbol points to the LCR
set 0xffffffff, %l2
st %l2, [%l6]                          ; "clear" LCR
set 0xfff7ffff, %l2
st %l2, [%l6]                          ; Assert IRESET*
set 0xfffcffff, %l2
st %l2, [%l6]                          ; Assert IPL0*
set 0xfff7ffff, %l2
st %l2, [%l6]                          ; Remove IPL0*
set 0xffffffff, %l2
st %l2, [%l6]                          ; Remove IRESET*
```

Figure 1. VIC64 Reset

- The VMEbus
- Control registers (including VIC64)

These local areas may or may not be visible to other VMEbus modules. It is not uncommon for shared memory to be the only local resource available to other VMEbus modules. This is the case for the CYB675. The local addresses and the VMEbus addresses to this shared memory would almost certainly be different. Some type of secondary decode or address translation is necessary in these instances. In the examples given in this application note, the header file CYB675.h defines the local address map used for the CYB675.

The VIC64 does not directly support VMEbus accesses to their internal registers with the exception of the Interprocessor Communication registers. It is possible via external hardware to make all VIC64 registers visible to the VMEbus (see Cypress's application note titled "Using the VIC068 Without a Processor"). If a VAC068A is used, the local VIC64 (the VAC068A is not compatible with the D64 operations of the VIC64, but can be used if D64 operation are not performed) register region is fixed at addresses FFFCxxxx to FFFDxxxx. As a minimum, sufficient space must always be allotted for the 58 longwords of VIC64 registers.

VMEbus addressing through the LCR

As noted earlier, bits 18–28 of the LCR provide control over the most significant eleven VMEbus address lines. Therefore, a VMEbus access may consist of two parts: loading the LCR with the proper value, and performing the actual transfer to the VMEbus address location. This location consists of a fixed address in combination with the lower 21 bits of the VMEbus address.

As an example, assume a VMEbus A32, D32 read access is desired from the VMEbus address 0x38004000 and that the LEDs should remain clear (see *Figure 2*). A value of 0xA703FFFF should be written into the LCR. If the VMEbus address space on the local address map is 0xE00000 (VME_BASE_ADRS), the local address should be 0xE04000 (0xE00000 + least significant 21 bits of VMEbus address).

An addressing scheme of this sort makes the entire 4-Gbyte range of the VMEbus addressable by the CYB675. A disadvantage is that the LCR must be written for any VMEbus transaction is in a different 2-Mbyte address spaces from the previous VMEbus transaction. An example of a function that would return the proper address is shown in *Figure 3*.

An example function that returns the proper LCR value could be as shown in *Figure 4*.

CY7C964 Address Comparator Configuration

The CYB675 uses the CY7C964 as the VMEbus slave address comparator. The address comparator consists of two registers: the mask register and the compare register. The compare register is loaded with the base address of the slave address. The mask register is loaded with a value that determines which bits of the address should be compared with the value in the compare register. This defines the size of the address region. A zero in a bit enables the comparison of the corresponding bit in the compare register to the VMEbus address bit.

For example, if there are 4 Mbytes of shared memory and the VMEbus slave range is to start at address 0xC00000, the following values should be loaded into the CY7C964 registers:

Compare Register:	0x00C00000
Mask Register:	0x003FFFFF

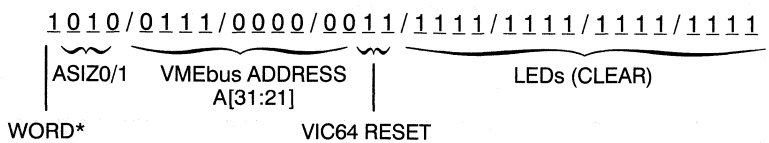


Figure 2. VMEbus A32, D32 Read Access

```
/* CYB675.h includes the following:
   typedef unsigned int WORD
   #define VME_BASE_ADRS 0xE00000 */

#include <CYB675.h>

#define VMEADRSMASK 0x001FFFFF

WORD *CalcVMEadrs (adrs)
WORD *adrs
{
WORD VMEadrs;

VMEadrs = (WORD) adrs;
VMEadrs &= VMEADRSMASK;       /* mask off upper 11 bits of address */
VMEadrs |= VME_BASE_ADRS;     /* overlay VMEbus address for CYB675 */

return ((WORD *) VMEadrs);
}
```

Figure 3. VMEbus Address Calculation

```
/* CYB675.h includes the following:
   typedef unsigned int WORD */

#include <CYB675.h>

#define LCRADRSMASK 0xFFE00000
#define LCRMASK 0xE003FFFF
#define LCRSHIFT 3

WORD *CalcLCR (adrs, LCReg)
WORD *adrs, LCReg;
{
WORD TempAdrs;

TempAdrs = (WORD) adrs;    /* convert WORD pointer to WORD */
TempAdrs &= LCRADRSMASK;   /* mask off lower 21 address bits */
TempAdrs >>= LCRSHIFT;     /* shift over by 3 */
LCReg &= LCRAMSK;          /* clear out existing address in LCReg */
LCReg |= TempAdrs;         /* overlay new address onto LCReg */

return (LCReg);
}
```

Figure 4. LCR VMEbus Address Calculation

```
/* NO!!! */
WORD *VMEadrs = (WORD *) 0x400000;

/* Yes!!! */
WORD *VMEadrs;
VMEadrs = (WORD *) 0x400000;
```

Figure 5. Proper Variable Initialization

Compiling Considerations

Because the monitor used for the CYB675 is EPROM-based, certain considerations are noted, namely:

1. All monitor sections that can be read-only are linked such that they occupy a contiguous section of EPROM. This may be done with the −R option of a UNIX *cc* compiler. The −R option merges the code segment TEXT with the initialized data segment DATA.

2. Because the DATA segment is now located in EPROM, any initialized data is now read-only and is not modifiable. This suggests that variable declarations do not initialize the variable, as shown in *Figure 5*.

3. The uninitialized data segment BSS and the stack segment STACK must be located in RAM.

Example VIC64 Software Building Blocks

The following are examples of code that were used for the VIC64-specific routines on the CYB675.

vic.h

vic.h is a header file that defines useful macros and VIC64-register-related constants. First, the macro VIC is defined, which returns an address to a VIC64 register. The argument to this macro is the number of the register. These numbers start from 0 (VIICR) and end with 57 (BTLR2) for the VIC64 (56 for the VIC068). These numbers are not the address of the register. Next, constants are defined that assign these numbers to the register names themselves. And lastly, a unique VIC64 register identifier is given to each register so that its address and contents can be obtained directly. A similar macro is defined for setting and clearing the Interprocessor Communication (IPC) switches. This IPC macro needs, as an argument, the starting address of the VMEbus IPC areas of interest.

As examples, consider the code fragment shown in *Figure 6*, which illustrates the VIC_xxx macros.

In addition, numerous other constants are defined that aid in manipulating the various bit fields within the registers themselves. These constants are separated by register. Also, the last character of the constant name may consist of a underscore (_) or lower case letters that indicate something about the constant or the bits. *Table 5* summarizes these characters.

```
#include <vic.h>              /* VIC macros located here */
#include <CYB675.h>           /* typedef for BYTE (unsigned char) */

BYTE      TempStorage;
BYTE      *TempStoragePtr;

TempStorage = *VIC_BTCR;      /* read contents of BTCR */
*VIC_SS0CR0 = TempStorage;    /* store contents of SS0CR0 */
TempStoragePtr = VIC_TTR;     /* read pointer to TTR */
ICF_ICGS0_SET (ICF_BASE);     /* set ICGS0 */
```

Figure 6. Using the "VIC" Macros

Table 5. *vic.h* Constant Preceders

Suffix	Meaning
_	Implies a bit field which is cleared
r	Implies read-only bit(s)
m	Implies a masking value for bit(s)

CYB675.h

CYB675.h is a header file that contains board-specific constants. These constants also include the local address map of the board, including those resources described in *Table 6*.

In addition, other types and constants are defined, including individual DUART registers, power-of-2 constants, byte-extraction macros, and some NVRAM macros.

A Generic Block Transfer Utility

blt_cmd is a generic, command-line driven program that enables the user to perform almost every conceivable block transfer operation using the VIC64 or the VIC068. One notable exception is allowing the VIC64 to interrupt when the block transfer is complete. blt_cmd is meant mainly to be used as a vehicle for board and code testing.

Configuration is provided by the command-line arguments outlined in *Table 7*.

Table 6. CYB675 Local Address Symbols

Memory Area	Privilege	Symbol
EPROM	Read/Write	ROM_BASE_ADDRESS
Status Register (LCR)	Read-Only	STATUS1_BASE_ADRS
Control Register (LCR)	Write-Only	LOCAL_CONTROL_BASE_ADRS
DUART	Read/Write	M68681_BASE_ADRS
NVRAM	Read/Write	NVRAM_BASE_ADRS
7C964 Mask Register	Write-Only	BILC_M_BASE_ADRS
7C964 Compare Register	Write-Only	BILC_C_BASE_ADRS
Interrupt Acknowledge	Read-Only	INT_ACK_BASE_ADRS
VIC64	Read/Write	VIC_BASE_ADRS
VMEbus	Read/Write	VME_BASE_ADRS
Private SRAM	Read/Write	BANK1_BASE_ADRS
Shared SRAM	Read/Write	BANK2_BASE_ADRS

Table 7. Command-Line Arguments

Argument	Default[1]	Function
−6		Performs D64 transfers (requires VIC64 device).
−3	√	Performs D32 transfers.
−a[address]	0xC00000	Sets local starting address for which data will be read, for VMEbus write block transfers or written for VMEbus read block transfers to *address*.
−A[value]	Disabled	Sets user-defined AM code that is to used for block transfers to *value*.
−b[value]	0x200	Sets minimum value for byte count to *value*. If the −ib value is 0 (increment byte count) the fixed byte will be set to *value*.
−B[value]	0xFFFC	Sets maximum value for byte count to *value*. Not used if −ib value is set to 0.
−cl		Enables local boundary crossing.
−cL	√	Disables local boundary crossing.
−ct		Enables 2-kbyte VMEbus boundary crossing (implies −cv).
−cT	√	Disables 2-kbyte VMEbus boundary crossing.
−cv	√	Enables VMEbus boundary crossing.
−cV		Disables VMEbus boundary crossing.
−d		Enables the dual-path option but does not perform interleave master cycles (see −p).
−D	√	Disables the dual-path option.
−e		Sets the release mode to RWD.
−E	√	Sets the release mode to ROR.
−f		Enables DRAM refresh.
−F	√	Disables DRAM refresh.
−ib[value]	0	Set the byte count increment value to: *value* * size of the operand.
−ii[value]	0	Sets the interleave increment value to *value.*
−iu[value]	0	Sets the burst count increment value to *value.*
−i[value]	0	Sets minimum value for interleave to *value*. If the −ii value is 0 (increment increment count) the fixed interleave value will be set to *value*.
−I	0xF	Sets maximum value for interleave to *value*. Not used if −ii value is set to 0.
−k	√	Enables data set-up before every block transfer and data checking after every block transfer.
−K		Disables data set-up before every block transfer and data checking after every block transfer.

Note:

1. The check mark indicates the default of two preceding arguments.

Table 7. Command-Line Arguments (continued)

Argument	Default[1]	Function
−l[value]	1	Sets the number of block transfers to perform to *value.* If *value* is set to 0, program will loop forever.
−m		Enables the clearing of the BLT enable bit (BTCR[4]) during the first interleave (VIC64 only).
−M	√	Enables the clearing of the BLT enable bit (BTCR[4]) after the block transfer is completely finished.
−p		Enables the dual-path feature and performs VMEbus master cycles during the interleave period.
−P	√	Disables the performing of interleave master VMEbus cycles. Leaves the dual-path feature enabled.
−r	√	Enables BLT reads.
−R		Disables BLT reads.
−s[value]	3	Sets the VMEbus request level to *value.*
−t	√	Enables the "enhanced" BLT turbo mode (VIC64 only).
−T		Disables the "enhanced" BLT turbo mode.
−u[value]	0	Sets minimum value for the burst count to *value.* If the −iu value is 0 (increment burst count) the fixed burst count will be set to *value.*
−U[value]	0x3F	Sets maximum value for burst count to *value.* Not used if −iu value is set to 0.
−v[value]	0xDEADC0DE	Sets the value to which destination memory will be set to *value.*
−w	√	Enables BLT writes.
−W		Disables BLT writes.
−x		Restores all options to their default states.
[address(es)]	0x200000	VMEbus starting address(es) for block transfer. Up to five may be specified.

All mutually exclusive options are shown without a divider between the options. If two mutually exclusive options are defined, the last one in the command line will take precedence. The state of these options are saved in static variables such that once a configuration is entered, the whole command string will not have to be retyped. Only those options that need to be changed will have a new option. Using the −x option will restore all options to their default state.

Unsupplied Functions

blt_cmd.c contains one function, lib_atohex(), that is not supplied. It is a library routine supplied with the SPARCmon source. Any ASCII-to-hex convert-

```
#include <atohex.h>
/* needed for lib_atohex return
   values */

lib_atohex (string, hexvalue)
char *string;
unsigned long *hexvalue;

/*
inputs:
string    character to be converted
Outputs:
hexvalue  pointer to the hex result

Return value:
SUCCEEDED       valid number
(otherwise)     illegal hex number
*/
```

Figure 7. atohex() prototype

er could be used with small modifications to blt_cmd.c. lib_atohex() is outlined in *Figure 7.*

Program Flow

Figures 8, 9, and *10* illustrate the flow of blt_cmd.c.

Example Operations

The following examples show how blt_cmd can be used to initiate a variety of block transfers.

```
blt_cmd -l0 -6 -ii1 -aC800000 D800000
```

This command line would perform D64 read and write block transfers indefinitely using local address 0xC800000 and VMEbus address 0xD800000. After each read/write block transfer, the interleave period is incremented by 1. All other options would remain at their default values.

```
blt_cmd -3 -W -iu1 D800000 E800000
```

This command line would perform D32 read block transfers indefinitely (−l0 still in effect) using local address 0xC800000 (defined last time) and VMEbus addresses 0xD800000 and 0xE800000. After each read block transfer, the burst count and the interleave period (still defined from last time) is incremented by 1. All other options would remain at their default values.

```
blt_cmd -6 -w -ib1 -K -p D800000
```

This command line would perform D64 read and write (writes are re-enabled with −w) block transfers indefinitely using local address 0xC800000 and VMEbus address 0xD800000. After each read/write block transfer, the byte count would be incremented by 8 (1 * 8 bytes/transfer). Data checking is suppressed. Master cycles are performed in the interleave period. All other options would remain at their default values.

```
blt_cmd
```

Performs block transfers using the same parameters as the last time invoked.

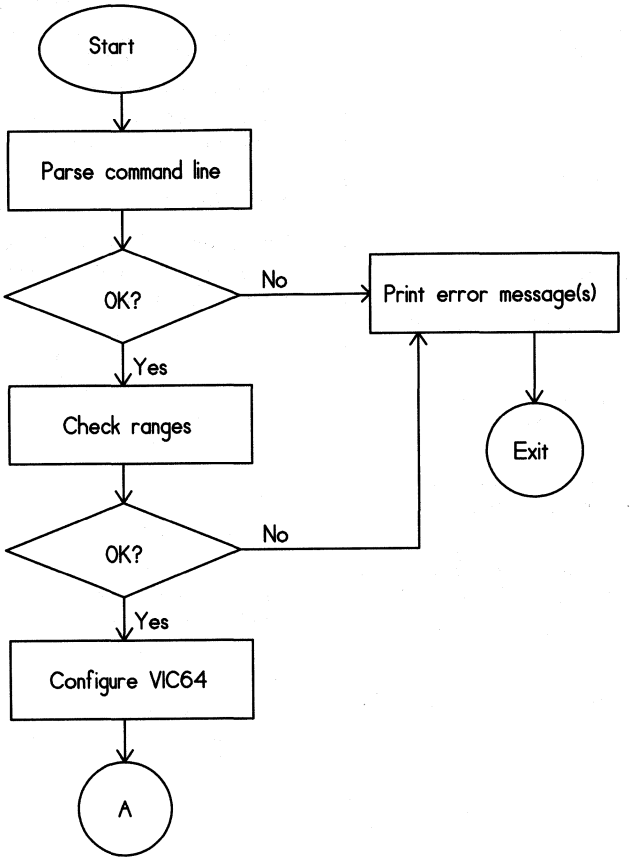


Figure 8. blt_cmd Flow

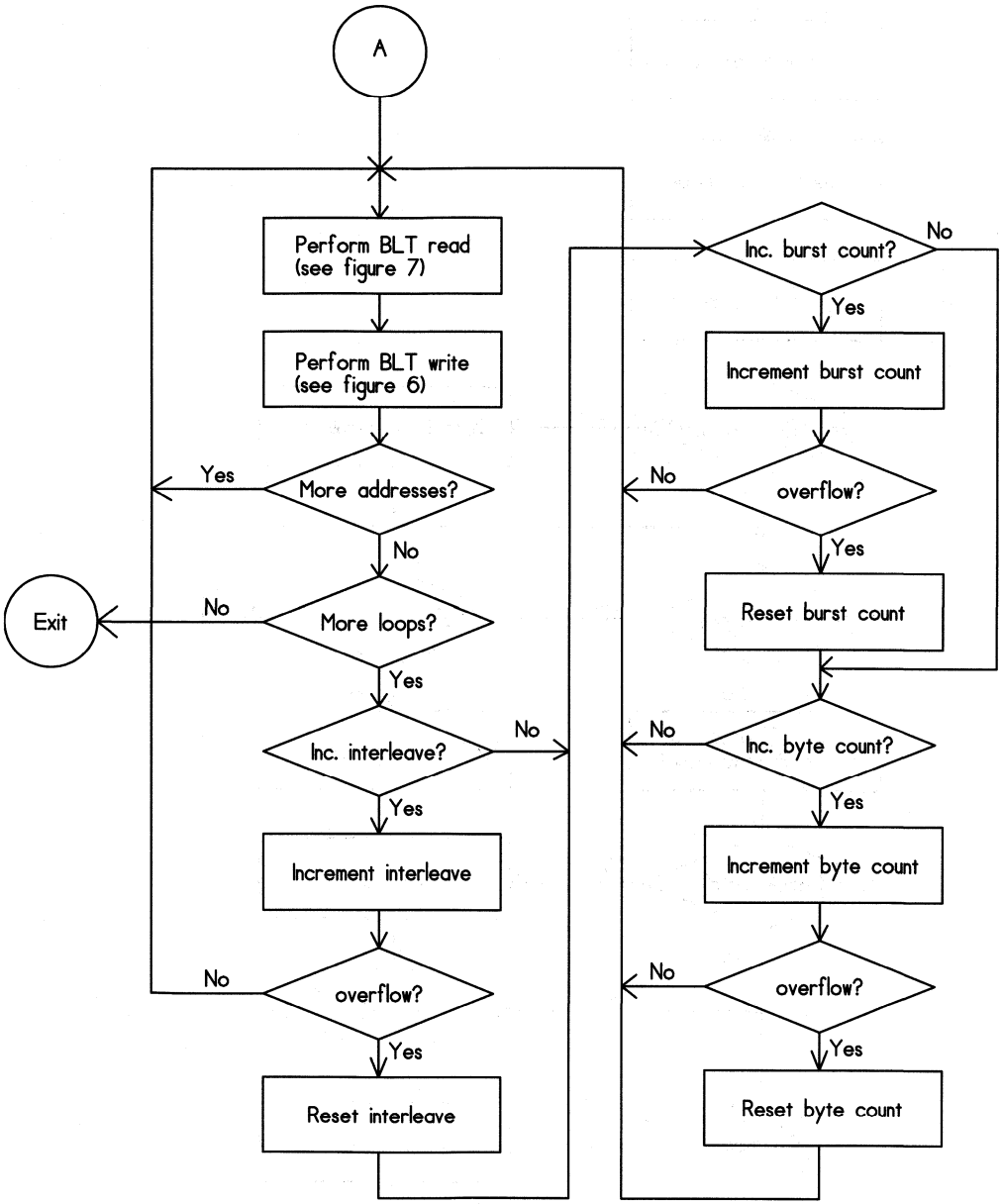


Figure 8. blt_cmd Flow (continued)

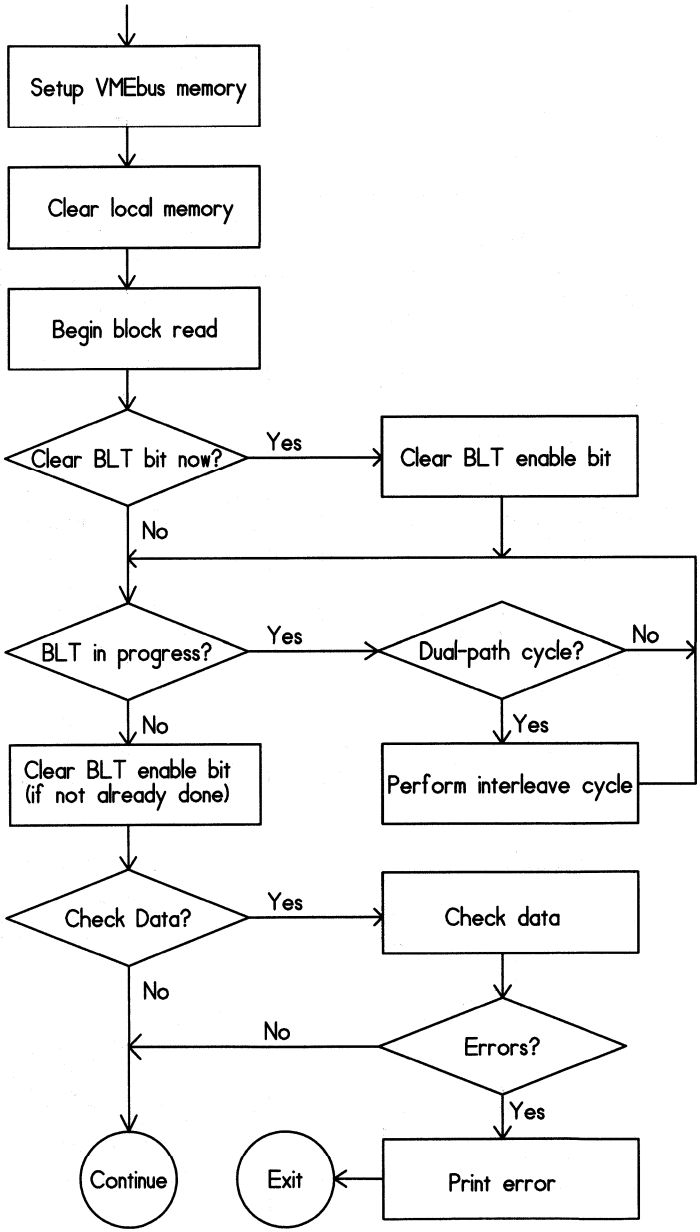


Figure 9. blt_cmd Read Flow

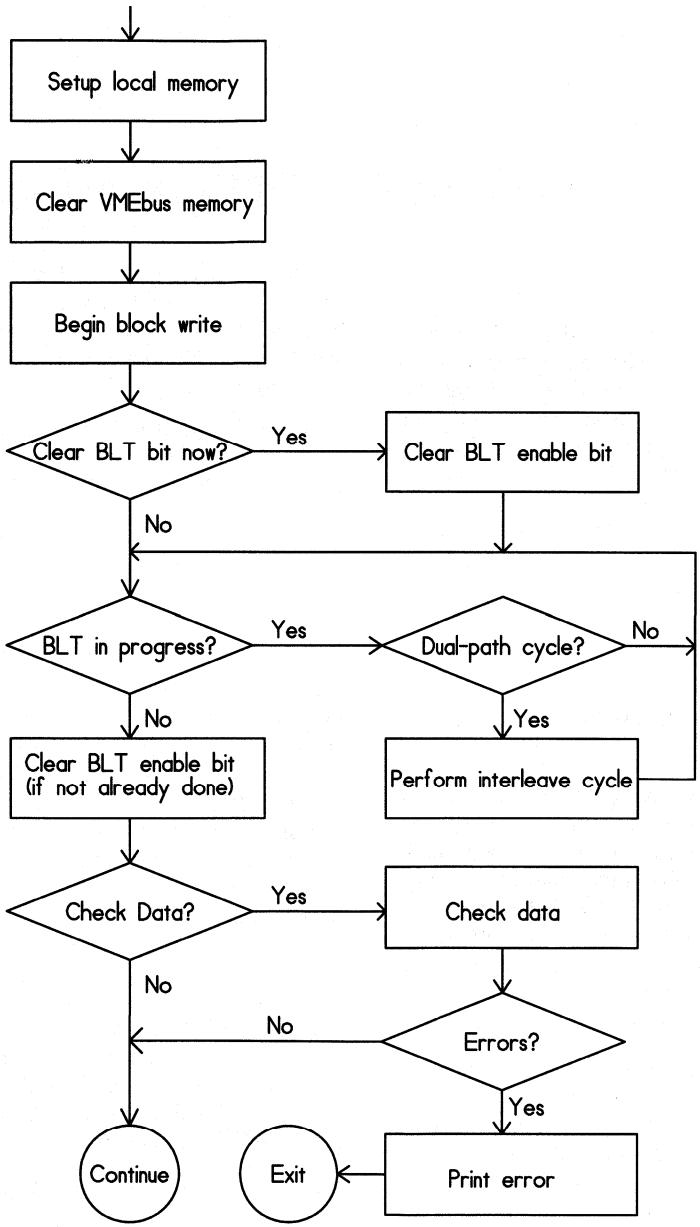


Figure 10. blt_cmd Write Flow

Document #: BUS0006–0992

VIC64 to Motorola 68040 Interface

Purpose

This application note shows how the VIC64 can be interfaced to a Motorola 68040 microprocessor operating at 40 MHz. The issues and assumptions that go into designing such an interface are considerable and complex; thus, this application note will not attempt to design a complete VME board that can do everything. It will cover some of the issues that are pertinent when designing a 68040-based VMEbus board and will focus on the circuitry required for VIC64 to 68040 interfacing.

Design Issues

Asynchronous Bus (VIC64) to Synchronous Bus (68040) Interfacing

With the 68040 microprocessor, Motorola radically changed its bus architecture. With the 68030 and prior processors, Motorola used an asynchronous bus protocol. The 68040, on the other hand, uses a synchronous bus protocol. The VIC64, being an extension of the VIC068A architecture, retains the asynchronous bus protocol that is compatible with the 68030 and prior microprocessors. This makes the VIC64 and 68040 bus protocols incompatible.

For the most part, the VIC64 is a peripheral to the 68040. The 68040 generates read and write cycles to the VIC64 and the VIC64 responds. There is only one case where the 68040 would act as a peripheral to the VIC64 and that is if the 68040's snooping capability were turned on and the 68040 was required to supply data from its internal cache for a VIC64 cycle. To simplify the snooping interface, there are memory design strategies described later in this application note that can isolate memory accessed by the VIC64 from the 68040 internal cache. Thus, whenever the VIC64 were to act as master on the bus, the 68040 would never need to respond to a VIC64 cycle. No memory area that the VIC64 can access would be cached by the 68040.

To allow the 68040 and VIC64 to communicate, the VIC64 must be synchronized to the 68040. The primary signals that undergo this synchronization are the handshaking signals, DSACK0* and DSACK1*, that the VIC64 sends to the 68040 to indicate the completion of a register transfer or a VMEbus transfer.

Putting a "Slow" VIC64 on the 68040's Bus

The 68040 synchronous bus can transfer data at a rate of 1 transfer per 2 cycles of the 40-MHz bus clock when running in single-cycle mode. The transfer can either be a byte, word, or longword in length. This translates to 1 transfer every 50 ns. The VIC64 responds to a request for a data transfer to its internal registers no quicker than 67.5 ns. When the 68040 accesses the VMEbus via the VIC64, the transfer can be considerably slower since the VMEbus slave controls the progress of the transfer. To pace the transfer without losing data, the 68040 allows a slow peripheral to hold off on asserting $\overline{TA}$ until it has its data available on a read, or can accept data on a write. The interface designed in this application note synchronizes the DSACK1* and DSACK0* signals from the VIC64 and uses them to generate $\overline{TA}$ to the 68040.

Bus Contention – Peripheral Write after Read

When designing with a high-speed processor and a slow peripheral, bus contention is always a concern. Bus contention comes into play when a slow peripheral is being read by the processor in the current bus

cycle and in the next cycle, the processor executes a write. Typically, the slow peripheral cannot be disabled off of the bus before the processor begins driving the bus. The VIC64 to 68040 interface is no exception.

Figure 1 shows the timing of the contention. The VMEbus interface used in this application note is the full functional D64 VMEbus interface using the VIC64 and 3 CY7C964s as shown in *Figure 4* of the Cypress application note titled, "Using the CY7C964 with VIC." At the end of the 68040 cycle where data is read, it takes up to 5 ns for PAS*, DS*, and CS* to deassert (using a PALC22V10D−7), up to 23 ns from DS* deasserted to ISOBE* deasserted, up to 12 ns from ISOBE* deasserted to CISOBE deasserted, and then 7.5 ns for the '245 to disable assuming a 74FCT16245T is used. The next cycle can begin and write data can be presented to the bus as early as 5.25 ns after the BCLK following the cycle when PAS*, DS*, and CS* were deasserted. This creates over 15 ns of contention!

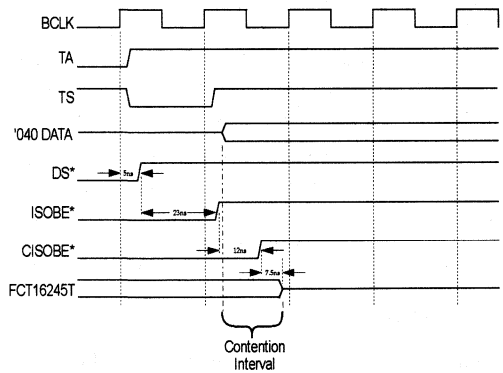


Figure 1. Contention for a Read Followed by a Write

Solving Bus Contention with Arbitration

The solution to the contention is easy considering the bus arbitration scheme of the 68040. In prior members of the 68k family, the processor also contained a bus arbiter on the same chip. Any peripheral that wanted to get access to the bus was required to request the bus from the processor. The 68040 relies on the designer to implement an external bus arbiter. All devices that can be masters on the bus must request the bus from the arbiter and the 68040 is no exception.

A way to eliminate the contention is to not allow the processor to begin a write cycle immediately after it has read the VMEbus or the VIC64 registers. The arbitration states of the 68040 make this possible. The timing of the arbitration is shown in *Figure 2*. At the beginning of the read cycle, the 68040 asserts $\overline{TS}$ along with an address that indicates either a VMEbus cycle or a VIC64 register access. The progression then is as follows:

1. The address is decoded and CS*, STROBE*, or MWB* is asserted along with PAS*.
2. The arbiter deasserts $\overline{BG}$ in response to CS*, STROBE*, or MWB* assertion. The 68040 will complete its current cycle. It is assumed that the 68040 does not want to relinquish the bus and will continue to drive $\overline{BR}$ asserted.
3. After receiving $\overline{TA}$, the 68040 is forced off the bus since the $\overline{BG}$ signal had been previously deasserted. However, when the arbiter sees that

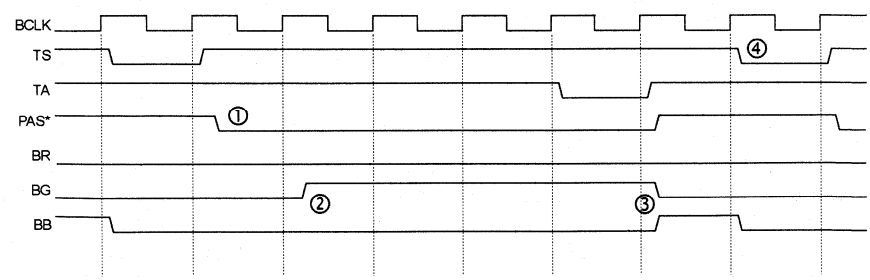


Figure 2. Arbitration Used to Eliminate Contention

$\overline{TA}$ has been asserted it grants the bus back to the 68040.

4. The 68040 on the next clock rising edge can assert a $\overline{TS}$ to begin the new cycle since $\overline{BG}$ is seen asserted.

With this method, there is no possibility of contention since 25 ns has been added to the contention resolution time.

For this method to be effective, good board layout and decoupling must be used. Taking the bus away from the 68040 causes its bus buffers to go high-impedance and then low-impedance in a single bus cycle. This can cause significant ground bounce and noise if the proper design practices are not used. Also, the signals that go high-impedance must be pulled up to V_{CC} to prevent them from floating.

Slave Access Implementation

Regardless of the memory map of the board, there are common issues pertaining to slave access of the board from the VMEbus. The slave interface is highly dependent on the function of the board. If the board is a memory array, chances are the board will primarily be accessed as a slave. However, if the board is a general purpose microprocessor, it will probably spend most of its time as a master on the VMEbus.

Since the slave interface is variable from board to board, the details of a slave interface to onboard circuitry will not be covered. The information in the *VIC068A/VAC068A User's Guide* and the *VIC64/CY7C964 Design Notes* contain ample information on using the VIC64 and CY7C964s for slave accesses. The next three sections address issues necessary for designing the slave circuitry on the board.

Bus Snooping

The 68040 can be configured to snoop cycles on its bus when it is not a master. Snooping is only a concern if the 68040 and the VIC64 share a common memory subsystem. If the VIC64 has its own dedicated memory which is gated off from the 68040's memory, snooping is not an issue (unless, of course, multiple bus masters reside on the bus with the 68040).

When the 68040 finds a cycle that requires data to be supplied from its internal cache, it will inhibit the memory subsystem and provide the requested data. The timing of this operation is synchronous to the BCLK and thus, if snooping were configured, the VIC64, when acting as a bus master, must have its signals synchronized to properly meet the 68040 timing.

Inhibiting Cache Transfers From Shared Memory

To avoid the timing difficulties that arise when snooping is enabled with a common memory subsystem, snooping can be disabled! This would also require that data areas on the board accessed by the VIC64 cannot be cached internally by the 68040. To disable caching of VIC64 register data or read VMEbus data, and disable snooping, accesses to the VIC64 and CY7C964 circuitry that generate STROBE*, CS*, and/or MWB* would cause $\overline{TCI}$, SC0, and SC1 signals to go to the 68040 in their inhibiting states. This would disallow the current cycle from being cached internal to the 68040.

To prevent the caching of data written to the board when the VIC64 is acting as a slave or a block transfer controller, the 68040's memory map decoder must assert $\overline{TCI}$ when any location the VIC64 can access when in that mode is requested by the 68040.

Memory Map Decoding and Remapping

Another design issue when implementing slave access logic is that of memory map decoding and remapping. When an address is provided from the VMEbus, it may not correspond to the same physical address on the board. Through the use of PLDs for decoding and shifting addresses, the VMEbus address can map to an on-board address.

Design Assumptions

Other than the design issues covered above, there are two assumptions that have been made in the design of the circuits herein. The first pertains to the memory system design and the second pertains to the buffer-type selection of the 68040.

Memory System Design

The goal in any memory system design is to match the performance of the memory to the masters that access it. This presents a problem in the design since the 68040 and VIC64 have vastly different bus structures. The 68040 is based on a synchronous bus and supports high-speed burst transfers as well as single-cycle transfers with all data and control signals synchronized to a common bus clock (BCLK). However, the VIC64 relies on asynchronous bus transfers that are paced by asynchronous data accesses and acknowledgements. There must be an assumption made by the board designer of one of the following memory strategies. Based on the typical application of the board, the designer can select a memory strategy to maximize data throughput. Two designs are presented here but many more are possible. In each case, the block labeled "VME Interface" contains the circuit shown in *Figure 4* of the Cypress application note titled, "Using the CY7C964 with VIC."

Two Memory Banks Architecture with No Caching of Shared Bank

Figure 3 shows a memory system design that is split into two separate banks. The first bank of memory is dedicated to the 68040 and runs synchronously. The second bank of memory is dedicated to the VIC64 and runs asynchronously. By having two separate memory banks, each can be designed to run optimally with its corresponding bus master. This would offer the best performance for the 68040 for its burst mode, and for the VIC64 for its burst mode. The gate between the two memory buses allows the 68040 access to the VIC's memory and to the VME-bus for single-cycle transfers. Access to the VIC64's memory bus is controlled by the arbiter and is granted to the 68040 when the VIC64 is not active on its bus.

Under normal operation, the gate opens when requested by the 68040, allowing the 68040 free access onto the VIC64's memory bus and onto the VME-bus. Only when the VIC64 is accessed as a slave or it is controlling burst transfers would the gate be closed. The VIC64 would request access to its bus via its LBR* signal. A memory configuration like this would allow both the VIC64 and the 68040 the most bandwidth on their respective busses. Both could be operating as bus masters at the same time. The application note titled "Interfacing the CY7C611A with the VIC64" uses this type of

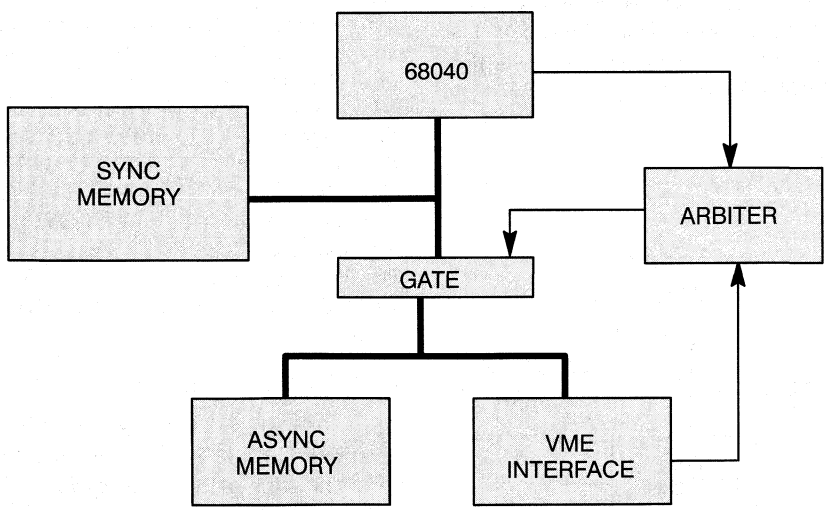


Figure 3. Two Memory Banks Architecture

memory scheme. The only caveat with this type of memory scheme is that when the 68040 is accessing the VIC64's memory, the data and acknowledgements from the VIC64 or its memory must be synchronized to the 68040's bus requirements.

Shared Memory with No Caching of VME Area

The other type of memory subsystem would be one that can act synchronously or asynchronously depending on whether the 68040 or the VIC64 was on the bus. This is illustrated in *Figure 4*. Both the 68040 and the VIC64 would share the same address and data buses and the arbiter would be used to grant access to one or the other. The arbiter could also indicate to the memory subsystem who has access to the bus. Although this simplifies the bus structure, it could complicate the memory design. It could also limit the bandwidth of the 68040 and the VIC64 to unacceptable levels.

However, this memory design might be perfectly acceptable for certain applications. The VIC068A and earlier 68K-family processors were able to share the same bus due to their compatible bus structures. Many designs allowed both the VIC64 and, for example, a 68020 to share bus bandwidth without detrimental effects. It is assumed that since this design revolves around a 68040 at 40 MHz, bus bandwidth for the processor is important! For this application note, it will be assumed that the separate memories strategy is used.

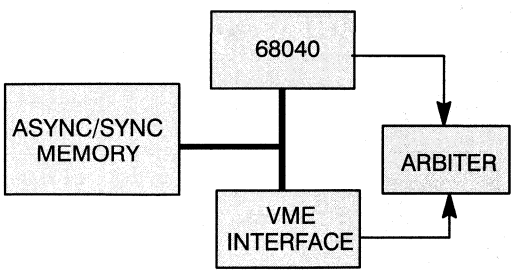


Figure 4. Single Memory Bank Architecture

68040 Configured for Large Buffer Timing Mode

To simplify the timing analysis and insure the peak performance from the design, the 68040 will be used in its Large Buffer Timing mode. This will require the careful layout of the board and the use of signal terminations to prevent adverse results from transmission line effects. Large Buffer mode is entered into during processor reset by pulling the $\overline{IPL2}$, $\overline{IPL1}$, and $\overline{IPL0}$ signals to a logic-one state.

Reset Circuitry

The reset circuitry and its routing is shown in *Figure 5*. There are three possible sources for a reset in this design. The first is a power-up or front panel pushbutton reset. The second is a reset initiated by the VIC64. The third is a 68040-initiated reset. The Reset PLD, a CY7C335-83, controls the sequencing for each of these reset types. The VHDL code describing this PLD is given in Appendix A.

Power-Up or Pushbutton Reset

The timing for the power-up or pushbutton reset is shown in *Figure 6*. While PWRUP_RST_N is LOW from either the pushbutton being depressed or the capacitor in *Figure 5* charging at power-up, BRD_RST_N_OUT is LOW. The capacitor and resistor values are chosen to guarantee that the clock and the board V_{CC} are stable when the rising edge of PWR_RST_N occurs. This insures that the VIC64 will be reset properly with a global reset. When the rising edge of PWRUP_RST_N occurs, the IRESET* signal is pulled LOW to the VIC64. The VIC64 responds with RESET* LOW, which in turn causes IPL0* to be pulled LOW, thus beginning a global reset. The IPL0* signal is then returned HIGH and, after a delay, IRESET* and BRD_RST_N_OUT are brought HIGH, ending the reset.

68040 Mode Selection

The 68040 is reset via the $\overline{RSTI}$ signal. For a valid reset to occur, the $\overline{RSTI}$ signal must be held LOW for a minimum of 10 BCLK cycles. The operation of the Reset PLD guarantees that $\overline{RSTI}$ will be held LOW for greater than this minimum amount of time. On the rising edge of $\overline{RSTI}$, the 68040 reads

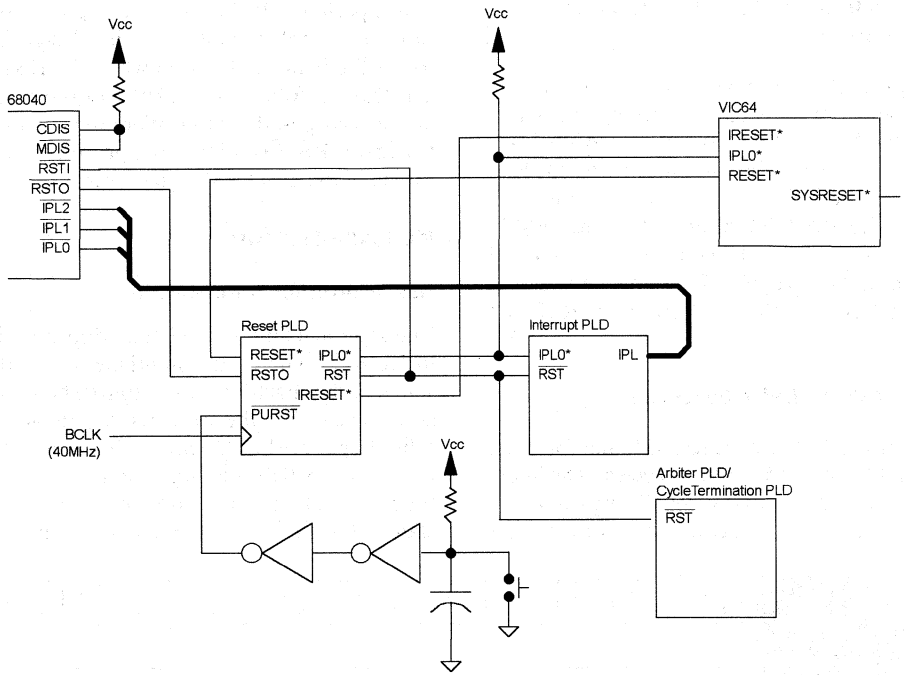


Figure 5. Reset Circuitry

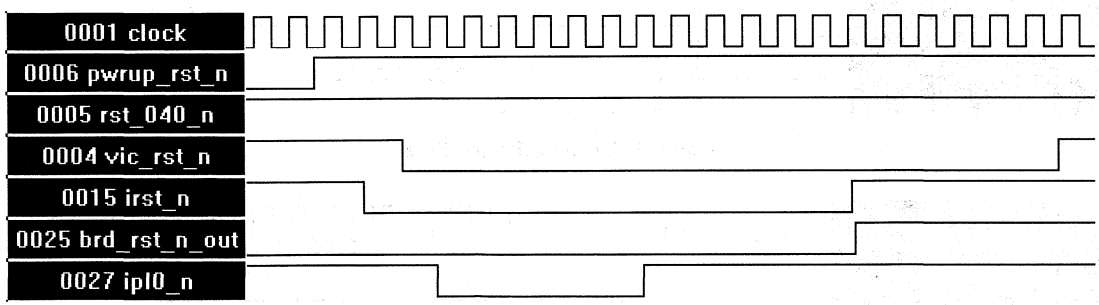


Figure 6. Power-Up or Pushbutton Reset

the current state of the $\overline{IPL0}$, $\overline{IPL1}$, $\overline{IPL2}$, $\overline{MDIS}$, and $\overline{CDIS}$ and sets the mode of operation of the 68040.

The $\overline{CDIS}$ and $\overline{MDIS}$ signals are both pulled HIGH through a resistor that, at reset, disables Multiplexed Bus mode and Data Latch Enable mode. During normal operation, pulling $\overline{CDIS}$ and $\overline{MDIS}$ HIGH enables the internal cache of the 68040 and enables its internal MMU. The $\overline{IPLx}$ signals are all pulled HIGH at reset also via the Interrupt PLD. This enables the Large Buffer Timing mode for the data, address, and control signals.

VIC-Initiated Reset (SYSRESET* Active or SRCR Written)

The timing for a VIC-initiated reset is shown in *Figure 7*. A reset from the VIC is caused by one of two events. If the SYSRESET* signal on the VMEbus is driven active, the VIC64 will respond with the RESET* driven active. The VIC64 will also issue a RESET* if the SRR ($E3) is written with a value of $F0. This will cause the Reset PLD to force a full board reset via the BRD_RST_N_OUT signal and a global reset to the VIC64 with the IPL0* and IRESET* signals.

Support for 68040 RESET Instruction

The 68040 has an instruction, RESET, that forces its $\overline{\text{RSTO}}$ signal LOW for 512 BCLK cycles. The internal state of the 68040 is unaffected during this interval, which makes this instruction good for resetting board peripherals during normal processor operation. The implementation in this design, however, forces the board to be reset when the $\overline{\text{RSTO}}$ signal is activated. When the 68040 sees the $\overline{\text{RSTI}}$ signal active during an $\overline{\text{RSTO}}$ LOW interval, it immediately negates $\overline{\text{RSTO}}$ and forces a processor reset. The timing of this reset is shown in *Figure 8*.

Bus Arbitration

Bus Arbitration State Machine

The state machine for the bus arbitration is shown in *Figure 9*. There are essentially three arbitration states in the machine with a fourth being the reset state. The task of the arbiter is to grant access to the VIC64's private bus (*Figure 3*). Thus, it will normally allow the 68040's $\overline{\text{BG}}$ signal to remain active at all times. In fact, the arbiter does not even consider the state of the $\overline{\text{BR}}$ signal from the 68040 in the arbitra-

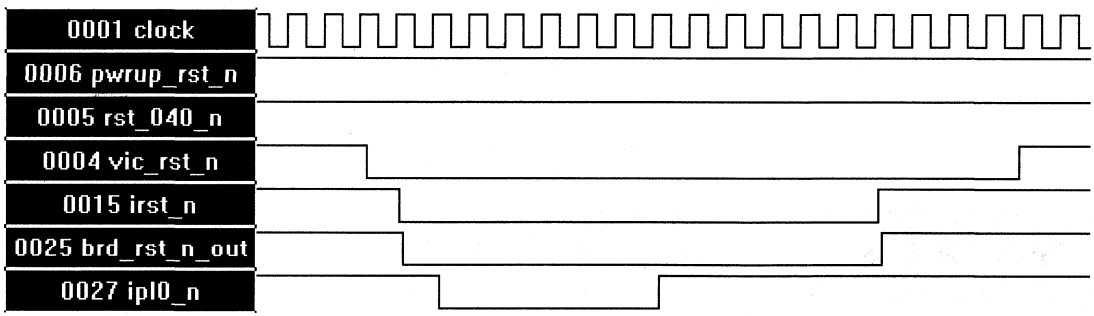


Figure 7. VIC64-Initiated Reset

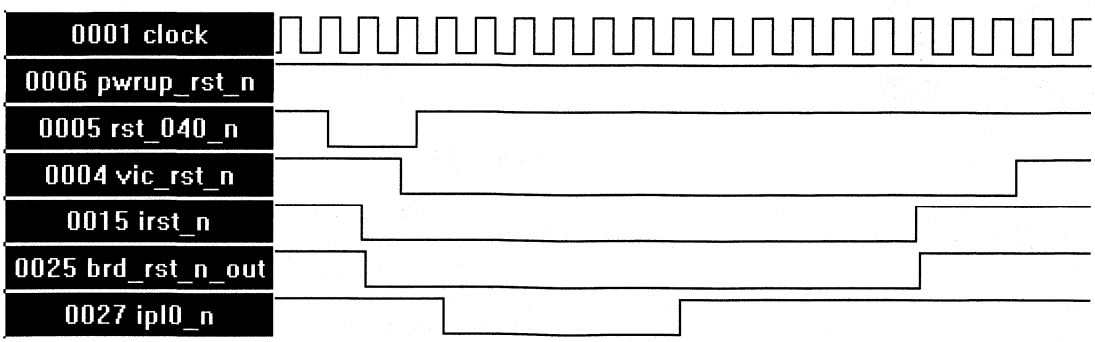


Figure 8. 68040-Initiated Reset

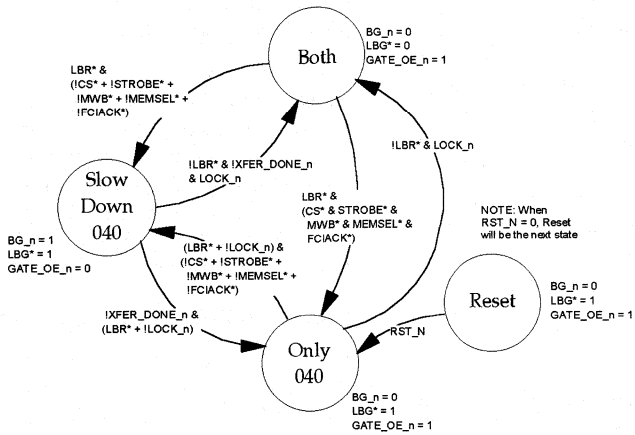


Figure 9. Bus Arbitration State Machine

tion. Rather, the state of the FCIACK*, MEMSEL*, CS*, STROBE*, or MWB* signals determine if the 68040 requires access to the VIC64's bus for either memory access, VIC64 or CY7C964 register access, or VMEbus access.

After a board reset has completed, the state machine transitions from the *Reset* state to the *Only_040* state. In this state, the 68040's $\overline{BG}$ signal is active, granting the 68040 access to its private bus. The LBG* to the VIC64 is inactive and the GATE_OE_N signal is also inactive. The GATE_OE_N signal is used to open the gate between the 68040's bus and the VIC64's bus. This "gate" consists of '245-type bidirectional drivers between the data busses and '244-type drivers for the 68040's address and control signals. It is suggested that FCT-C speed gates be used to insure that the data and address signals from the 68040 are driven to the VIC64 and/or the VMEbus with adequate set-up time to DS* and PAS*.

The bus arbitration state machine is implemented in a CY7C335–83 and is named the Bus Arbitration PLD. The VHDL code describing this PLD is in Appendix D. The PLD and its connections within the circuit are shown in the schematic in *Figure 18*.

68040 Request for VIC64 Bus Access

There are two states from which the 68040 can gain access to the VIC64's bus, the *Only_040* state and the *Both* state. From the *Only_040* state, the 68040 would attempt access to the VIC64 bus with either the FCIACK*, CS*, STROBE*, MWB*, or the MEMSEL* going active. Only one of the signals would go active in a given access cycle. If the LBR* from the VIC64 is not active, the 68040 is granted access to the VIC64's bus by transitioning to the *Slow_Down_040* state. Another possible transition into the *Slow_Down_040* state from the *Only_040* state is if the 68040 is currently in the middle of a read-modify-write cycle, indicated by the $\overline{LOCK}$ signal being active. Regardless of the state of LBR*, if $\overline{LOCK}$ is active, the read-modify-write cycle is allowed to continue before the VIC64 can gain control of its bus.

Once in the *Slow_Down_040* state, the BG to the 68040 is driven inactive (to cause the 1-cycle delay in the 68040 bus cycle as described above) and the GATE_OE_N is driven active. When the current cycle completes as indicated by the XFER_DONE_N signal going active, the state machine transitions to either the *Both* state or the *Only_040* state depending on the state of the LBR* and $\overline{LOCK}$ signals.

If the VIC64 currently has ownership of its bus and the 68040 requests the VIC64's bus, the 68040 will not be granted access until the LBR* from the VIC64 has gone inactive. This could pose a problem if the VIC64 were in the midst of a block transfer. The 68040 might not receive ownership of the VIC64 in a timely fashion. Although not implemented in this application note, a bus timeout could be implemented to cancel the 68040's attempt to access the VIC64's data bus, or a method of testing the BLT* signal before initiating a cycle could be used.

VIC64 Bus Requests

The VIC64 requests ownership of its bus via the LBR* going active. If the active state is *Only_040* and the 68040 is currently not in the middle of a read-modify-write cycle (LOCK inactive), the state machine will transition to the *Both* state. In this state, the 68040 will have access to its bus, the VIC64 will be granted access to its bus, and the gate between the two buses will be closed. If the active state is *Slow_Down_040*, indicating that the 68040 currently owns the VIC64's bus, the VIC64 will not be granted its bus until the 68040 finishes its current cycle (assuming that the cycle is not the first half of a read-modify-write cycle). When the cycle completes, the state machine will transition from the *Slow_Down_040* state to the *Both* state.

Once in the *Both* state, the state machine will not transition until the VIC64 finishes its current cycle and releases its bus by driving the LBR* signal inactive. If the 68040 is attempting access to the VIC64's bus via the CS*, STROBE*, MWB*, or MEMSEL* signals, the state machine will transition to the *Slow_Down_040* state; otherwise, it will transition to the *Only_040* state.

Sample Arbitration Timing Diagrams

Figure 10 is a sample arbitration timing diagram. As the state machine exits the *Reset* state, $\overline{BG}$ is active, and GATE_OE_N and LBG* are inactive. When the 68040 attempts access to the VIC64's bus with the MEMSEL* signal, it is granted access and the $\overline{BG}$ signal goes inactive and the GATE_OE_N goes active. During the access, the LBR* signal goes active signifying that the VIC64 wants access to its bus. It is granted access (LBG* goes LOW) after the 68040's cycle completes with the XFER_DONE_N signal pulsing active.

During the VIC64's active time on its bus, the 68040 attempts access to the VIC64's bus via the MWB* signal. The 68040's cycle does not begin until the

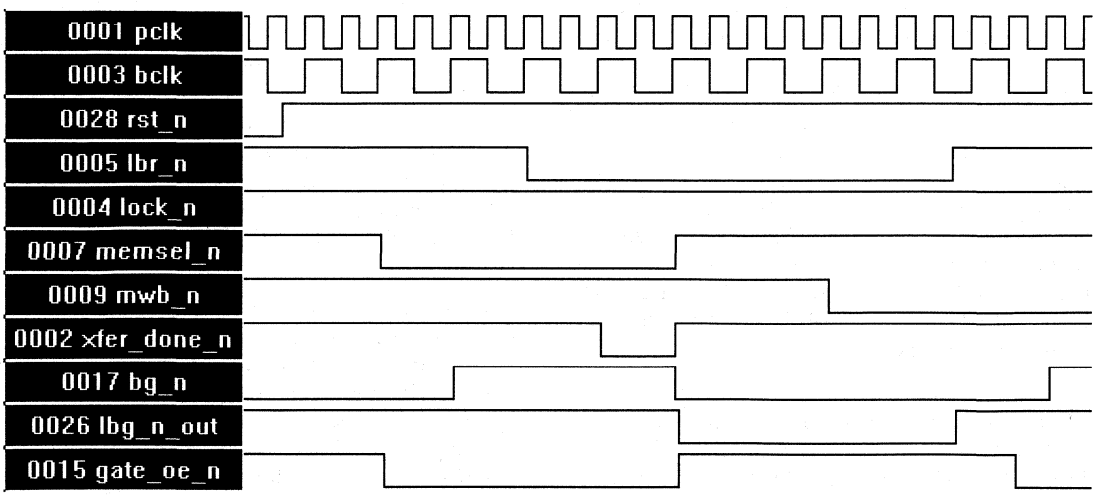


Figure 10. Arbitration Timing Diagram 1

LBR* signal is driven inactive, at which time the LBG* signal is driven inactive along with $\overline{BG}$. The *gate_oe_n* signal is driven active, allowing the 68040 onto the VIC64's bus.

Figure 11 is a continuation of *Figure 10*. The 68040 is completing its access with the MWB* signal and begins a read-modify-write cycle via the MEMSEL* signal. At the same time, the VIC64 requests access to its bus with the LBR* signal. In this case, the 68040 wins the arbitration and is allowed to complete the two cycles of the read-modify-write sequence. Once the sequence completes, the VIC64 is granted access to its bus until it deasserts the LBR* signal.

VIC64 and CY7C964 Register Access Cycles

VIC64 and CY7C964 register access cycles, as well as all other access cycles, are controlled by three PLDs. Two of the PLDs, the Address and Cycle Decode PLDs, control the initiation of a transfer. They are PALC22V10D-7s. The remaining PLD, the Cycle Termination PLD, controls the normal or abnormal completion of a cycle. This PLD is a CY7C335-83. The VHDL code for these PLDs can be found in Appendix B and Appendix C respectively. The PLDs and their connections within the circuit are shown in the schematic in *Figure 18*.

Selection of the PALC22V10D and CY7C335 Devices

The PALC22V10D and CY7C335 were chosen for a single, key reason. Both have a guarantee on their output data stability. The CY7C335 has a parameter, t_{OH}, that guarantees 2 ns of output data stability from the clock supplied to the part. The PALC22V10D-7 also guarantees a minimum on the t_{CO} specification of 2 ns. This is vital to the design because the 68040 running at 40 MHz requires that signals such as $\overline{TA}$, $\overline{TEA}$, $\overline{TBI}$, $\overline{TCI}$, etc., have a hold time of 2 ns from the rising edge of BCLK.

Selecting the VIC Registers vs. the CY7C964 Registers

Both the VIC64 registers and the CY7C964 registers are mapped into the same base address of A31–A28 = "0001." To make the determination between register sets, the lowest-order address bits, A00 and A01, are used in conjunction with the size signals, SIZ0 and SIZ1. When a byte transfer is requested and the lowest address bits are both HIGH, this is decoded as a VIC64 register access. When a longword transfer is requested and the lowest ad-

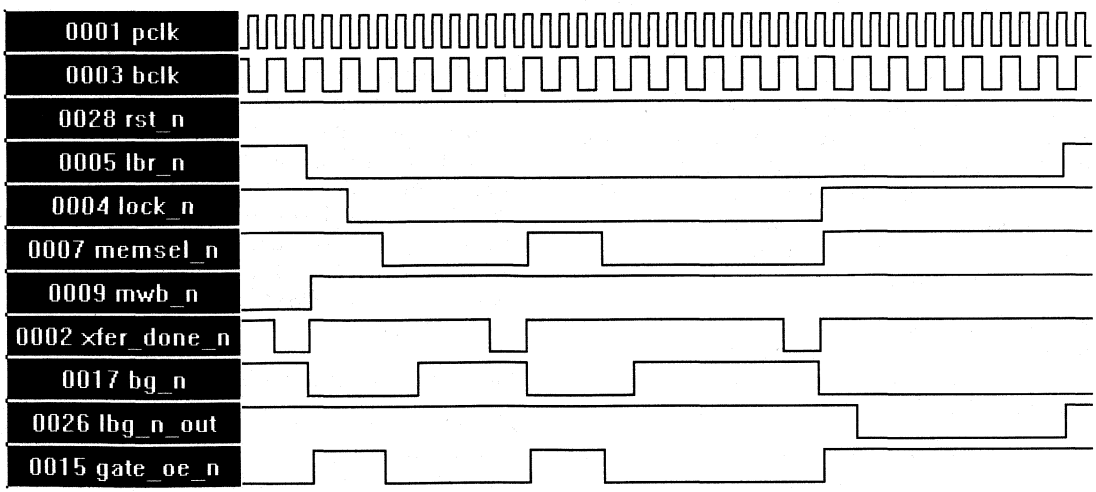


Figure 11. Arbitration Timing Diagram 2

dress bits are both LOW, this is decoded as a CY7C964 register access.

Register Access Cycle Initiation

If an address of $1xxxxxxx_{16}$ is detected when $\overline{TS}$ from the 68040 is active, a register access cycle is initiated. The address is qualified by the lowest address bits, the SIZ signals, and the transfer type from the 68040. For the CY7C964 register access, the PAS* and DS* signals are both kept inactive and only the STROBE* signal is allowed to be driven active. Data on the D31–D08 signals will be written into the CY7C964's while the data on D07–D00 is ignored. For the VIC64 register access, the PAS*, DS*, and CS* are all driven active. Data to be written to the VIC64 would be presented on D07–D00. Data read from the VIC64 would also appear on D07–D00.

To assure the proper timing on the D07–D00 signals with respect to the DS* signal, DS* is driven active on the cycle following PAS* driven active. This guarantees that, during a write cycle, data is present at the VIC64 prior to DS* becoming active.

Register Access Cycle Termination

The end of a register access cycle is indicated differently depending on whether the VIC64 registers were accessed or the CY7C964 registers were accessed. Also, the read or write status of the transfer has a bearing on how the cycle is terminated. For the VIC64 register transfers, the Cycle Termination PLD waits for either DSACK0* or DSACK1* to occur to indicate the end of the transfer. The DSACK1* and DSACK0* signals are registered as they enter the Cycle Termination PLD in order to synchronize them to the BCLK before they are used in output equations.

For the CY7C964 register transfers, the PLD counts three BCLK cycles before ending the cycle. This is because there are no external signals that indicate that the CY7C964s have received data.

In order to allow proper data hold times to the CY7C964 or VIC64, the termination of a write cycle is handled differently from the termination of the read cycle. In a read cycle, all active signals (PAS*, DS*, and CS*) are driven inactive at the same time in response to the XFER_DONE_N signal from the Cycle Termination PLD. However, for writes, an additional signal, XFER_DONE_W_N, is activated a full cycle before XFER_DONE_N. The STROBE* signal for CY7C964 register writes and the DS* for VIC64 register writes are driven inactive in response to this signal. On the subsequent BCLK cycle, the 68040 is given a $\overline{TA}$ signal and the PAS* and CS* signals are driven inactive. This insures that the rising edge of STROBE* or DS* is a cycle before the 68040 can remove data from the bus, thus guaranteeing the necessary data hold time into the CY7C964's and VIC64.

Performance of Register Access Cycles

An example of VIC64 register access is shown in *Figure 12*. An example of CY7C964 register access is shown in *Figure 13*. From these diagrams, the following performance figures are guaranteed for the different types of register access cycles.

VIC64 register write: 11 BCLK cycles assuming the slowest DSACK1/0* response time from the VIC64.

VIC64 register read: 10 BCLK cycles assuming the slowest DSACK1/0* response time from the VIC64.

CY7C964 register write: 10 BCLK cycles.

Master Read Cycles

Master Read cycles are also controlled by three PLDs, two Address and Cycle Decode PLDs and a Cycle Termination PLD. These cycles are very similar to a VIC64 Register read; however, instead of the VIC64 providing data and terminating the cycle, an addressed slave board would provide data and indicate that the data is available with the VMEbus signal, DTACK*. The VIC64 would issue DSACK1* and/or DSACK0* in response to the DTACK* signal.

Master Read Cycle Initiation

If an address of $2xxxxxxx_{16}$, $3xxxxxxx_{16}$, or $Fxxxxxxx_{16}$ is detected, along with R/$\overline{W}$ being in the HIGH state, when $\overline{TS}$ from the 68040 is active, a VMEbus master read access cycle is initiated. The address is qualified by the transfer type from the 68040.

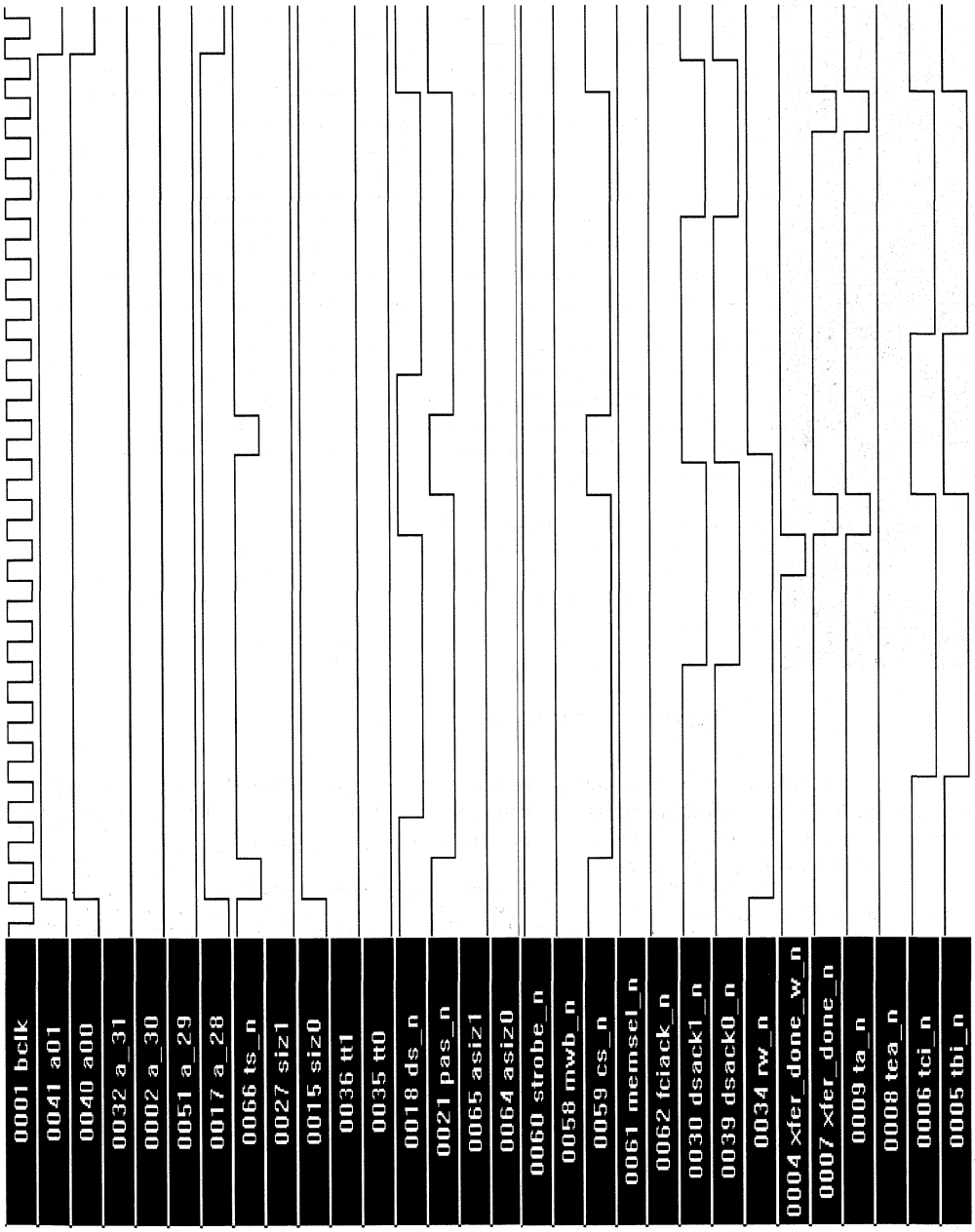


Figure 12. VIC64 Register Access

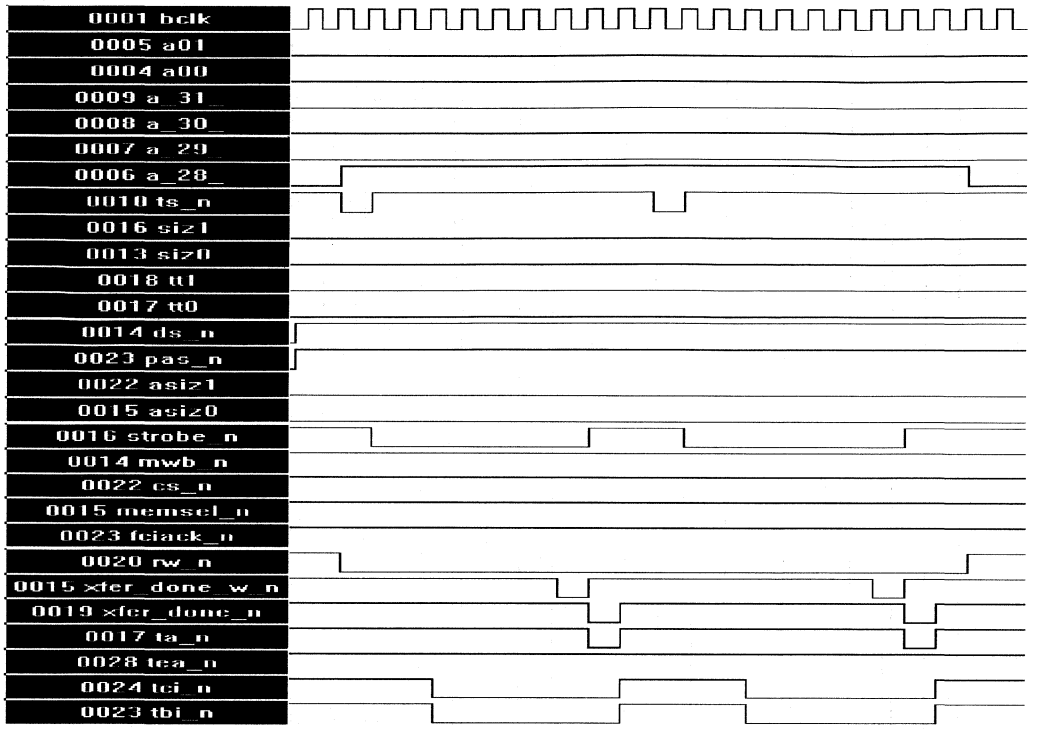


Figure 13. CY7C964 Register Access

MWB*, PAS* and DS* are driven active when the cycle is decoded and the ASIZ1 and ASIZ0 are driven based on the address from the 68040. *Table 1* shows how the address from the 68040 is decoded.

Table 1. 68040 Address Decode

68040 Address	Address Size	ASIZ1/ASIZ0
$2xxxxxxx_{16}$	A16	1/0
$3xxxxxxx_{16}$	A24	1/1
$Fxxxxxxx_{16}$	A32	0/1

The 68040 also indicates the memory space that is to be accessed with its TM2–TM0 lines. These signals are driven the the VIC64's FC2 and FC1 signals for generating AM codes on the VMEbus. The VIC64 will control the buffer control signals to the CY7C964's based on the size of the data that is to be transferred (indicated by the SIZ1 and SIZ0 signals from the 68040) and will initiate a VMEbus read with the appropriate VMEbus signals.

Master Read Cycle Termination

Once there has been a VMEbus read cycle initiated by the VIC64, there are three typical ways the cycle can be terminated. The cycle can be ended normally, be deadlocked and retried, or be terminated abnormally via a bus error.

Master Read Cycle Normal Termination

A normal master read will be terminated when the Cycle Termination PLD receives one or both of the DSACK0* or DSACK1* signals from the VIC64. These signals are driven by the VIC64 in response to a DTACK* signal from the addressed slave on the VMEbus backplane. The performance of a normal-

ly terminated cycle can vary due to the response time of the slave board being addressed and whether or not the VIC64 was granted access to the VMEbus quickly. *Figure 14* illustrates two back to back read cycles on the VMEbus that are terminated normally.

Master Read Cycle Deadlock/Retry Termination

A master read that ends in deadlock occurs when a slave cycle and a master cycle are asserted to the VIC64 at the same time. The VIC64 indicates that a deadlock has occurred by asserting the DEDLK* signal when the local side attempts access during a slave transaction. In response to the DEDLK* signal from the VIC64, the Cycle Termination PLD drives both the $\overline{\text{TEA}}$ and $\overline{\text{TA}}$ signals active to the 68040. This will cause the 68040 to end its cycle, wait one BCLK cycle (due to the Bus Arbitration PLD), and then attempt the cycle again.

The 68040 will continue to retry the cycle until the cycle is ended either with $\overline{\text{TEA}}$ or $\overline{\text{TA}}$ only. On each attempt the 68040 makes to the VIC64, the Address and Cycle Decode PLDs look at the state of the DEDLK_S signal from the Cycle Termination PLD. DEDLK_S is a double-registered version of the DEDLK* signal from the VIC64. If the DEDLK_S is active on an otherwise valid attempt to access the VIC64's private bus, the DEAD_N signal will activate instead of the normal signal (MWB*, CS*, etc.). The assertion of *DEAD*_N will not affect the Bus Arbitration state machine but will allow the Cycle Termination PLD to again cause a retry to the 68040 with $\overline{\text{TEA}}$ and $\overline{\text{TA}}$ together.

This method is used because there is a possibility that DEDLK* could go inactive *during* a 68040

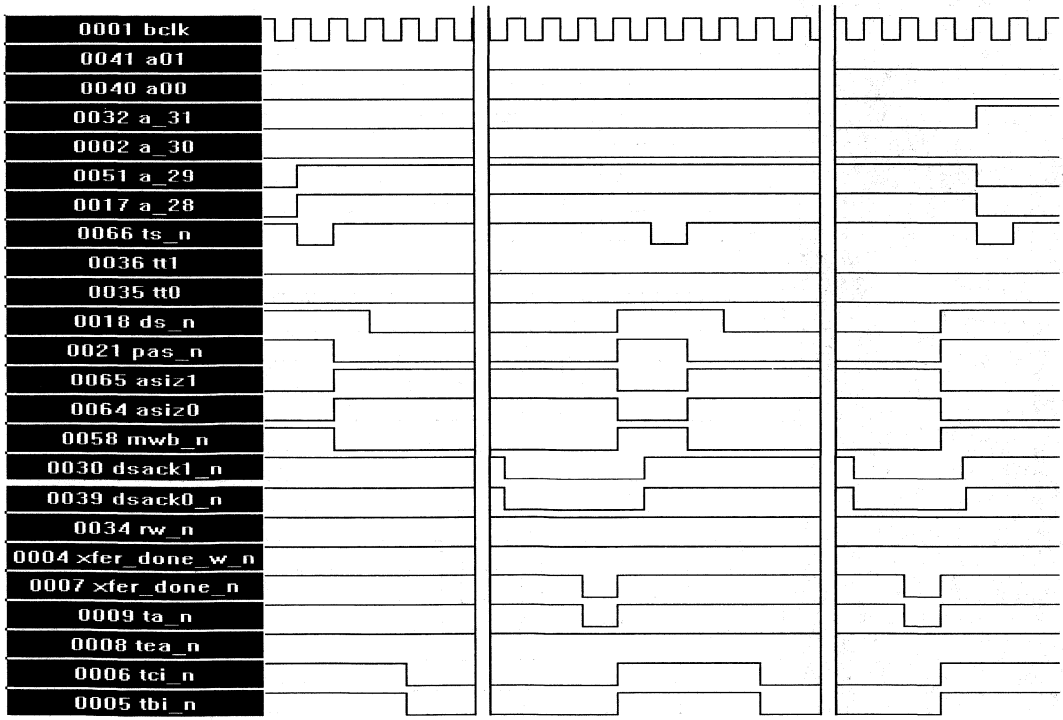


Figure 14. Master Reads

cycle. If this occurs, the Cycle Termination PLD could see DEDLK* active and terminate the cycle with $\overline{TEA}$ and $\overline{TA}$ active, thus indicating a retry to the 68040. However, the Bus Arbitration PLD may see that the Address and Cycle Decode PLD is signaling a valid cycle with LBR* inactive and an active select signal. This would cause the Cycle Termination PLD and the Bus Arbitration PLD to lose synchronization with each other. By preventing the Bus Arbitration PLD from even seeing a cycle that potentially could have a deadlock (with the *dead_n* signal), it will not arbitrate that cycle and the Cycle Termination PLD will cause a retry.

When the DEDLK* has been released by the VIC64, the 68040 will be able to finally complete the cycle that it has been retrying. The cycle will be a normal master read. *Figure 15* shows 4 cycles attempted by the 68040. The first cycle ends in retry when a DEDLK* is recognized in the middle of the

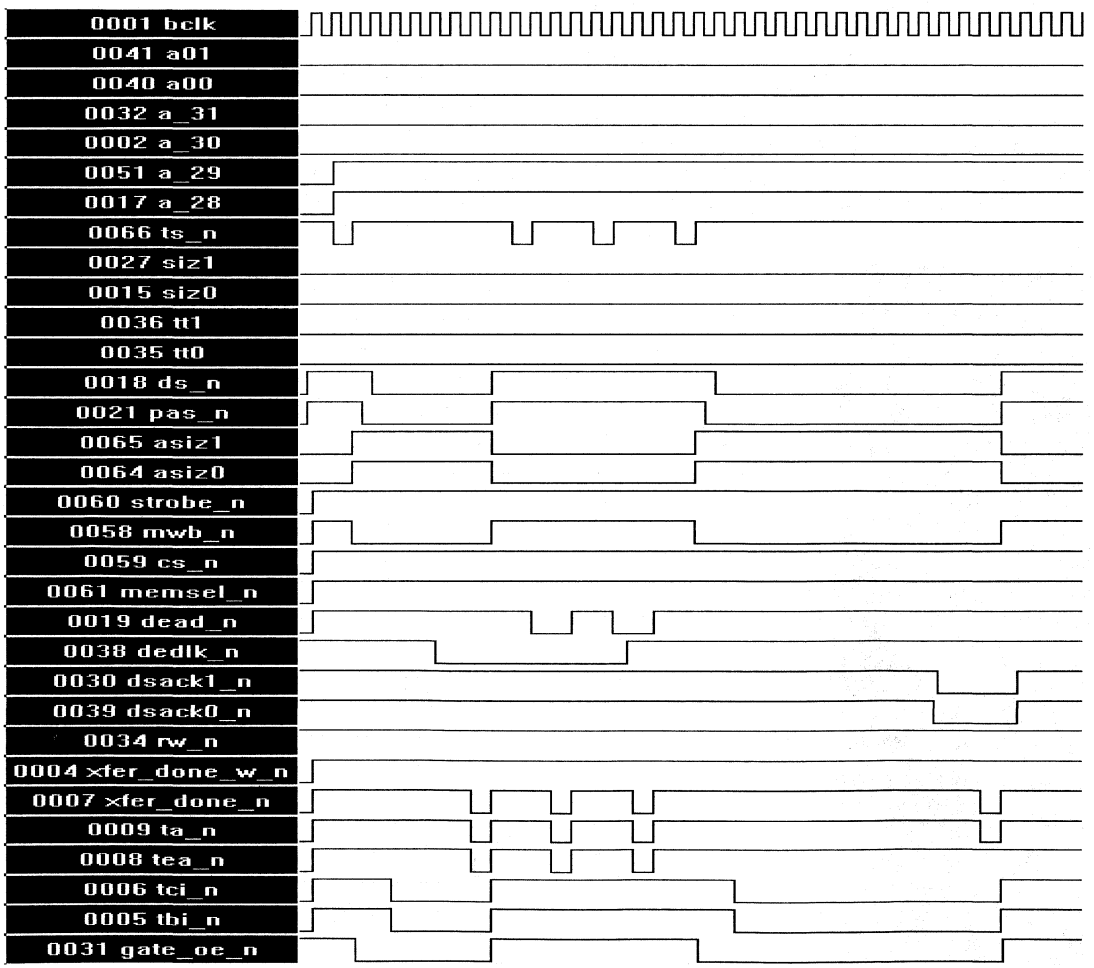


Figure 15. Master Reads with Deadlock

cycle. The next two cycles begin as deadlocked cycles and thus are immediately forced to be retried by the Cycle Termination PLD. The last cycle occurs after the deadlock and thus begins and ends as a normal master read.

Master Read Cycle Bus Error Termination

A master read will be terminated as a bus error to the 68040 when the Cycle Termination PLD receives the LBERR* signal from the VIC64. This signal is driven by the VIC64 in response to a BERR* signal from the addressed slave on the VMEbus backplane or a VMEbus timeout (based on the configuration of the TTR, register $A3 in the VIC64). A cycle terminated with a Bus Error will look similar to the timing shown in *Figure 14*. The differences will be twofold. First, the LBERR* signal will be driven by the VIC64 instead of DSACK1* and DSACK0*. Second, the $\overline{TEA}$ signal will be driven to the 68040 instead of the $\overline{TA}$ signal. Other than these differences, the cycles are equivalent.

Master Write, Writepost, and BLT Initiation Cycles

Like the Register Access cycles and Master Read cycles, Master Write cycles are also controlled by three PLDs, two Address and Cycle Decode PLDs and a Cycle Termination PLD. These cycles are very similar to a VIC64 Register write; however, instead of the VIC64 providing data and terminating the cycle, an addressed slave board would provide data and indicate that the data is available with the VMEbus signal, DTACK*. The VIC64 would issue DSACK1* and/or DSACK0* in response to the DTACK* signal.

Commonality Between the Various Write Cycles

Each of the cycles, Master Write, Writepost, and BLT initiation are subtly different. However, each shares the common trait that they are all write cycles from the 68040's perspective and all produce an MWB* signal to the VIC64. In each case however, the data is dissimilar. The Master Write and Writepost actually provide data that is transferred to an addressed slave, while the data from the BLT initiation cycle is the local address where the block transfer begins.

Write Cycle Initiation

If an address of $2xxxxxxx_{16}$, $3xxxxxxx_{16}$, or $Fxxxxxxx_{16}$ is detected, along with R/$\overline{W}$ being in the LOW state, when $\overline{TS}$ from the 68040 is active, a VMEbus master write-access cycle is initiated (or a block transfer initiation cycle if bit 6 of the BTCR is set). The address is qualified by the transfer type from the 68040. MWB*, PAS* and DS* are driven active when the cycle is decoded and the ASIZ1 and ASIZ0 are driven based on the address from the 68040. *Table 2* shows how the address from the 68040 is decoded.

Table 2. 68040 Address Decode

68040 Address	Address Size	ASIZ1/ASIZ0
$2xxxxxxx_{16}$	A16	1/0
$3xxxxxxx_{16}$	A24	1/1
$Fxxxxxxx_{16}$	A32	0/1

The 68040 also indicates the memory space that is to be accessed with its TM2–TM0 lines. These signals are driven as the VIC64's FC2 and FC1 signals for generating AM codes on the VMEbus. The VIC64 will control the buffer control signals to the CY7C964's based on the size of the data that is to be transferred (indicated by the SIZ1 and SIZ0 signals from the 68040) and will initiate a VMEbus write with the appropriate VMEbus signals.

To assure the proper timing on the D07–D00 signals with respect to the DS* signal, DS* is driven active on the cycle following PAS* driven active. This guarantees that during a write cycle data is present at the VIC64 prior to DS* active.

Write Cycle Termination

As with a VMEbus read cycle, once there has been a VMEbus write cycle initiated by the VIC64, there are three typical ways the cycle can be terminated. The cycle can be ended normally, be deadlocked and retried, or be terminated abnormally via a bus error.

Write Cycle Normal Termination

A normal master write will be terminated when the Cycle Termination PLD receives one or both of the

DSACK0* or DSACK1* signals from the VIC64. These signals are driven by the VIC64 in response to a DTACK* signal from the addressed slave on the VMEbus backplane. The performance of a normally terminated cycle can vary due to the response time of the slave board being addressed and whether or not the VIC64 was granted access to the VMEbus quickly.

In order to allow proper data hold times to the VIC64 for BLT initiation cycles and Master Write-posts, the termination of a write cycle is handled differently from the termination of the read cycle. In a read cycle, all active signals (PAS*, DS*, and MWB*) are brought inactive at the same time in response to the XFER_DONE_N signal from the Cycle Termination PLD. However, for writes, an additional signal, XFER_DONE_W_N, is activated a full cycle before XFER_DONE_N. The DS* signal is brought inactive in response to this signal. On the subsequent BCLK cycle, the 68040 is given a $\overline{TA}$ signal and the PAS* and MWB* signals are driven inactive. This insures that the rising edge of DS* is a cycle before the 68040 removes data from the bus, thus guaranteeing the necessary data hold time into the VIC64.

This timing is not an issue for Master writes since the VMEbus specification states that a slave will only issue a DTACK* after it has accepted the data written to it. Thus, hold time on the data is inherent in the delay from DTACK* on the VMEbus to DSACKx* on the local bus to $\overline{TA}$ from the cycle termination PLD. *Figure 16* illustrates two back-to-back write cycles on the VMEbus that are terminated normally.

Write Cycle Deadlock/Retry Termination

A write that ends in deadlock occurs when a slave cycle and a master cycle are asserted to the VIC64 at the same time. The VIC64 indicates that a deadlock has occurred by asserting the DEDLK* signal when the local side attempts access during a slave transaction. In response to the DEDLK* signal from the VIC64, the Cycle Termination PLD drives both the $\overline{TEA}$ and $\overline{TA}$ signals active to the 68040. This will cause the 68040 to end its cycle, wait one BCLK cycle (due to the Bus Arbitration PLD), and then attempt the cycle again.

As with deadlock on a read cycle, there is a timing relationship between the bus arbitration PLD and the cycle termination PLD that must be maintained. This timing is discussed in the Master Read Cycle Deadlock/Retry Termination section above. Timing for write cycles that deadlock is identical to the timing shown in *Figure 16*. The only difference is the relationship of DS* to both MWB* and PAS* as described above.

Write Cycle Bus Error Termination

A master write will be terminated as a bus error to the 68040 when the Cycle Termination PLD receives the LBERR* signal from the VIC64. This signal is driven by the VIC64 in response to a BERR* signal from the addressed slave on the VMEbus backplane or a VMEbus timeout (based on the configuration of the TTR, register $A3 in the VIC64). A cycle terminated with a Bus Error will look similar to the timing shown in *Figure 16*. The differences will be twofold. First, the LBERR* signal will be driven by the VIC64 instead of DSACKx*. Second, the $\overline{TEA}$ signal will be driven to the 68040 instead of the $\overline{TA}$ signal. Other than these differences, the cycles are equivalent.

Interrupt Acknowledge Cycles

Interrupt Acknowledge cycles are controlled by the Interrupt PLD, Cycle Termination PLD, and Address and Cycle Decode PLDs. Typical functionality of the Interrupt PLD is shown in *Figure 17*. Operation of the Address Decode PLDs and the Cycle Termination PLD is comparable to a Master Read Cycle except that FCIACK* is active rather than MWB*.

Operation At Reset

Although not an interrupt-related function, the Interrupt PLD controls the configuration of the 68040's buffer mode via the $\overline{IPL2}$, $\overline{IPL1}$, and $\overline{IPL0}$ signals. During a board reset, the signals are all driven to a HIGH state to configure the 68040's signals to Large Buffer mode.

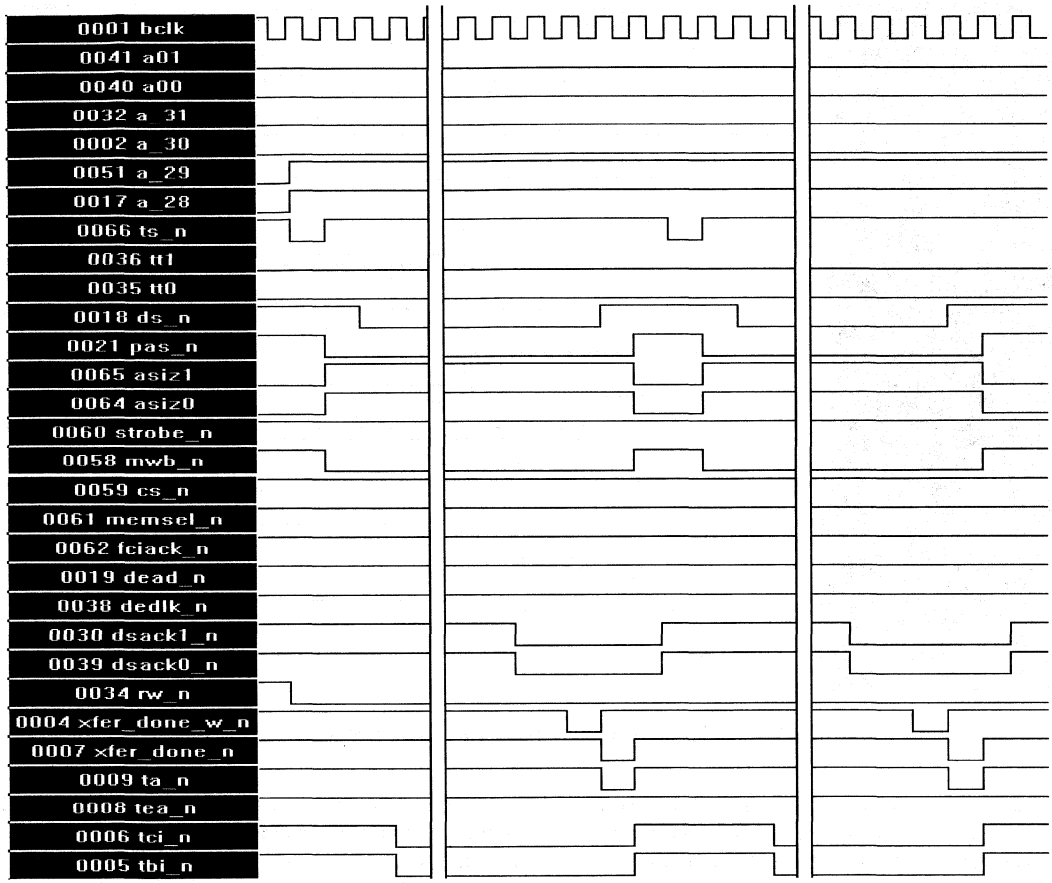


Figure 16. Master Writes

VMEbus vs. Local Interrupts

There are two possible sources for interrupts, the VMEbus and local interrupts. For VMEbus interrupts, the 68040 will only be involved if the VIC64 is configured as an interrupt handler. When the VMEbus interrupter generates an interrupt, the VIC64 will assert an interrupt to the 68040 via the IPL2*–IPL0* lines. The 68040 will respond with an interrupt acknowledge cycle. When the VIC64 sees the interrupt acknowledge cycle from the 68040, it obtains the VMEbus to request the Status/ID vector from the Interrupter. As the Status/ID vector is placed on data bus, it is passed through to the 68040 and the VIC64 terminates the cycle.

For local interrupts, a device on the board requiring service will assert an interrupt to the VIC64 via the LIRQ7*–LIRQ0* lines. The VIC64 will then assert an interrupt to the 68040 via the IPL2*–IPL0* lines. The 68040 will respond with an interrupt acknowledge cycle. There are two possible responses to the interrupt acknowledge cycle from the 68040. If the VIC64 is enabled to supply a vector for the current interrupt, it will do so and terminate the cycle with the DSACKx* signals. If the VIC64 is not

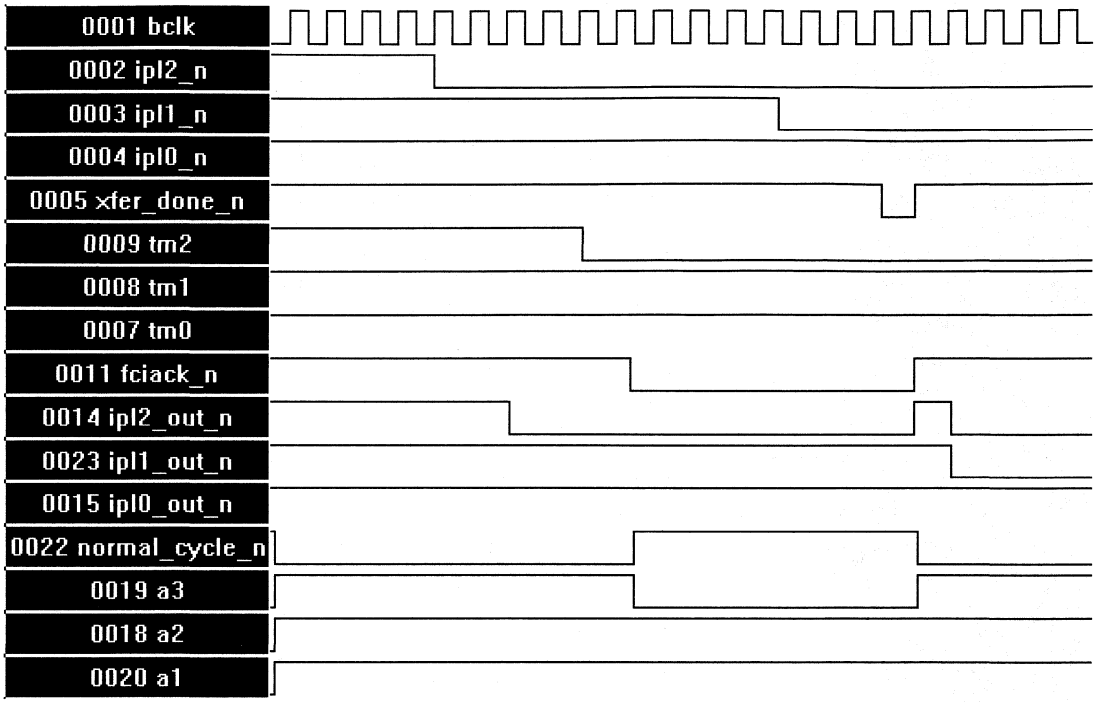


Figure 17. Interrupt Initiation and Acknowledge

enabled to provide a vector, it will assert the LIACKO* signal instead. The Cycle Termination PLD asserts AVEC to the 68040 in response to LIACKO* active and then terminates the cycle.

Another possible configuration for local interrupts is to have the LIACKO* from the VIC64 tell the interrupting device to supply a Status/ID vector to the 68040 and then terminate the cycle with a $\overline{TA}$ to the 68040.

Interrupt Initiation from the VIC64

As described above, the VIC64 issues an interrupt via its IPL2*–IPL0* signals. The IPL2*–IPL0* signals are normally in a HIGH state and are pulled LOW to request interrupt service from the 68040. When the IPL2*–IPL0* signals are pulled LOW, the Interrupt PLD synchronizes the signals before providing them to the 68040. The VIC64 may have up to 10 ns of skew in the IPL2*–IPL0* signals and that skew could be expanded to a full BCLK cycle through synchronization. However, the 68040 must see the interrupt level for two full BCLK cycles before it is considered valid so the skew is inconsequential.

Interrupt Cycle Initiation by the 68040

When the 68040 begins a bus cycle with $\overline{TS}$ active and the TT1 and TT0 signals are both HIGH, an interrupt acknowledge cycle is indicated. The SIZ1 and SIZ0 signals are also qualified to make sure they are indicating a byte-width operation. The Address Decode PLDs respond to the cycle by issuing FCIACK*, PAS*, and DS*. The VIC64 recognizes the beginning of an interrupt acknowledge cycle on the overlap of FCIACK*, PAS*, and DS* active.

Interrupt Cycle Decode

When FCIACK* goes active, the Interrupt PLD captures the current state of the $\overline{IPL2}$–$\overline{IPL0}$ signals and holds them throughout the cycle. The use of a Cypress PALC22V10D for this PLD guarantees the required hold times on the $\overline{IPL2}$–$\overline{IPL0}$ signals to the 68040 are met. When the cycle terminates, the $\overline{IPL2}$–$\overline{IPL0}$ signals are driven inactive for at least one BCLK cycle before a new interrupt level can be driven.

Another function that the Interrupt PLD performs is steering the TM2–TM0 signals from the 68040 onto the A3–A1 address lines on the VIC64. The TM2–TM0 signals from the 68040 contain the level of the interrupt being acknowledged and the VIC64 requires that information be passed on address lines A3–A1.

Interrupt Cycle Termination

The interrupt cycle is terminated in one of two ways from the VIC64. If the VIC64 is configured to supply a Status/ID vector, it will place that vector on D7–D0 and supply DSACKx* to the Cycle Termination PLD. If the VIC64 is not configured to supply a vector, it will issue a LIACKO* signal which will cause the Cycle Termination PLD to issue AVEC to the 68040 and then terminate the cycle with a $\overline{TA}$.

Summary

This application note has designed a possible VIC64 to Motorola 68040 interface. The issues and assumptions that must be addressed in the interface have been covered. The circuitry required for bus arbitration, resets, reads, writes, and interrupts has been designed. VHDL code for the PLDs used in the application note as well as timing diagrams and schematics have been provided in the following Appendices.

References

1. Cypress Semiconductor, *VIC068A/VAC068A User's Guide*, June, 1992.
2. Cypress Semiconductor, *VIC64/CY7C964 Design Notes*, October, 1992.
3. Motorola, Inc., *MC68040 Microprocessors User's Manual (M68040UM/AD)*, 1992.
4. Mazor, S., and P. Langstraat, *A Guide to VHDL*, Boston: Kluwer Academic, 1992.

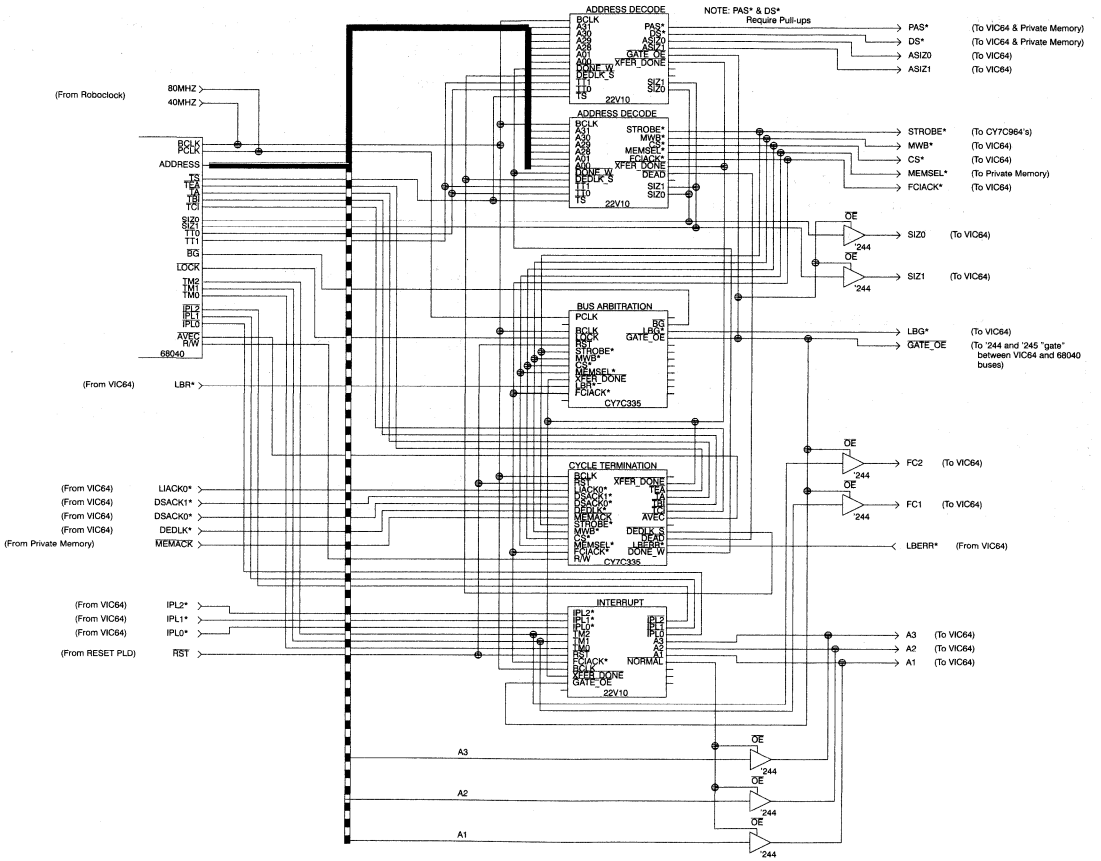


Figure 18. Schematic of Control Logic Circuitry

Appendix A. Reset Control PLD (CY7C335)

```
-- RESET PLD design
--
-- The following table is a cross reference between the PLD port names and
-- those on the schematic in the application note text.
--
--    clock            =  BCLK
--    vic_rst_n        =  RESET*
--    rst_040_n        =  RSTO "bar"
--    pwrup_rst_n      =  PURST "bar"
--    irst_n           =  IRESET*
--    brd_rst_n_out    =  RST "bar"
--    ipl0_n           =  IPL0*

ENTITY rst_ctrl IS
        PORT (clock, vic_rst_n, rst_040_n, pwrup_rst_n : in bit;
              irst_n, brd_rst_n_out : out bit;
              ipl0_n : inout x01z);
attribute part_name of rst_ctrl:entity is "c335";
attribute pin_numbers of rst_ctrl:entity is "clock:1 vic_rst_n:4 "
                                    & "rst_040_n:5 pwrup_rst_n:6";
END rst_ctrl;

USE work.rtlpkg.all;

ARCHITECTURE operation OF rst_ctrl IS
        signal pwrup_rst_n_rising:bit;
        signal pwrup_rst_n_reg:bit;
        signal pwrup_rst_n_reg1:bit;
        signal brd_rst_n:bit;
        signal start:bit;
        signal expired: bit;
        signal timer_count:integer(0 to 4);
        signal ipl0_oe: bit;
        signal ipl0_sig: bit;
        type states is (idle, rst1, rst2, rst3, wait_for_no_rst);
        signal rst_state: states;

        BEGIN

-- This process captures the power-up or pushbutton reset in two registers
-- in order to synchronize the signal and create a pulse on the rising edge
-- of the reset

        sync_rst: PROCESS BEGIN
            WAIT UNTIL clock = '1';
            pwrup_rst_n_reg <= pwrup_rst_n;
            pwrup_rst_n_reg1 <= pwrup_rst_n_reg;
        END PROCESS;

-- This concurrent assignment creates a pulse when the rising edge of the
-- power-up or pushbutton reset occurs.  This is used to begin a global reset
-- to the VIC which must be reset AFTER the Vcc and Oscillator are stable.
```

Appendix A. Reset Control PLD (CY7C335) (continued)

```
        pwrup_rst_n_rising <= '1' WHEN ((pwrup_rst_n_reg = '1') AND
                                        (pwrup_rst_n_reg1 = '0'))
                              ELSE '0';

-- This concurrent assignment guarantees that the reset out of this part will
-- be low when either the power-up/pushbutton reset is low or the brd_rst_n
-- signal generated in the process below is low.  This assures that as this
-- PLD powers up, the reset out of it will be low even if the process below
-- has not begun stable operation.

        brd_rst_n_out <= '0' WHEN ((brd_rst_n = '0') OR
                                   (pwrup_rst_n = '0'))
                         ELSE '1';

        rst: PROCESS BEGIN
            WAIT UNTIL clock = '1';

-- Do the following while the board is powering up or the pushbutton
-- is depressed

            IF (pwrup_rst_n_reg = '0') THEN
                  rst_state <= idle;
                  brd_rst_n <= '0';
                  ipl0_sig <= '1';
                  irst_n <= '1';
                  start <= '0';
            END IF;
            CASE (rst_state) IS

-- When the state machine is in the idle state, we look for the VIC to
-- initiate a reset or the 68040 to initiate a reset.  We also look for the
-- end of the powerup reset.  Remember that even though this state machine
-- is waiting for the end of power-up/pushbutton reset, the board is still
-- held in reset by the brd_rst_n_out signal above.

                  WHEN idle =>
                        IF ((pwrup_rst_n_rising = '1') OR
                            (vic_rst_n = '0') OR
                            (rst_040_n = '0')) THEN
                               rst_state <= rst1;
                               start <= '1';
                               irst_n <= '0';
                               brd_rst_n <= '0';
                        END IF;

-- In the rst1 state, we wait for one of two events.  If the VIC responds to
-- the reset from this PLD before the timer expires, we pull ipl0_n low and
-- continue to the rst2 state.  This would be the normal procedure.  If for
-- some unknown reason the VIC doesn't respond, we would wait for the timer
-- to expire and then assert ipl0_n.
```

Appendix A. Reset Control PLD (CY7C335) (continued)

```
                    WHEN rst1 =>
                            start <= '0';
                            IF (vic_rst_n = '0') THEN
                                    rst_state <= rst2;
                                    ipl0_sig <= '0';
                                    start <= '1';
                            ELSIF (expired = '1') THEN
                                    rst_state <= rst2;
                                    ipl0_sig <= '0';
                                    start <= '1';
                            END IF;

-- Just wait around in this state until the timer expires.  Remove the ipl0_n
-- signal at the end of this state.

                    WHEN rst2 =>
                            start <= '0';
                            IF ((expired = '1') AND (start = '0')) THEN
                                    rst_state <= rst3;
                                    ipl0_sig <= '1';
                                    start <= '1';
                            END IF;

-- Just wait around in this state until the timer expires.  Remove resets at
-- the end of this state.

                    WHEN rst3 =>
                            start <= '0';
                            IF (expired = '1') THEN
                                    rst_state <= wait_for_no_rst;
                                    irst_n <= '1';
                                    brd_rst_n <= '1';
                            END IF;

-- We remain in this state until the VIC comes out of reset.

                    WHEN wait_for_no_rst =>
                            IF (vic_rst_n = '1') THEN
                                    rst_state <= idle;
                            END IF;

-- Make the state machine complete.

                    WHEN others => rst_state <= idle;
            END CASE;
        END PROCESS;

-- The following makes the ipl0_sig signal a three-state signal on the
-- pin of the device.  This is required since ipl0 is normally driven by
-- the VIC64 but needs to be driven by this PLD during global reset.

        ipl0_oe <= NOT ipl0_sig;
        ipl0: bufoe port map (ipl0_sig, ipl0_oe, ipl0_n, open);
```

Appendix A. Reset Control PLD (CY7C335) (continued)

```
-- The timer process runs a counter that times how long the state machine
-- above should remain in a state.

        timer: PROCESS BEGIN
                WAIT UNTIL clock = '1';
                IF (pwrup_rst_n_reg = '0') THEN
                        timer_count <= 0;
                        expired <= '0';
                ELSE
                        IF (timer_count /= 0) THEN
                             timer_count <= timer_count + 1;
                        ELSE
                             timer_count <= 0;
                        END IF;

                    IF start = '1' THEN
                             timer_count <= 1;
                        END IF;

                        IF timer_count = 4 THEN
                             expired <= '1';
                    ELSE
                         expired <= '0';
                        END IF;

                END IF;
        END PROCESS;
END operation;
```

Appendix B. Address and Cycle Decode PLDs (PALC22V10D)

```
-- ADDRESS DECODER design 1
--
-- The following table is a cross reference between the PLD port names and
-- the signals found on the physical IC's.
--
--    bclk              =  BCLK on 68040
--    a(31) to a(28)    =  Address bus on 68040
--    ts_n              =  TS "bar" on 68040
--    tt1               =  TT1 on 68040
--    tt0               =  TT0 on 68040
--    siz1              =  SIZ1 on 68040
--    siz0              =  SIZ0 on 68040
--    xfer_done_n       =  xfer_done_n from TERMINATION PLD
--    xfer_done_w_n     =  xfer_done_w_n from TERMINATION PLD
--    gate_oe_n         =  gate_oe_n from BUS ARBITRATION PLD
--    dedlk_s           =  dedlk_s from TERMINATION PLD
--    asiz1             =  ASIZ1 to VIC64
--    asiz0             =  ASIZ0 to VIC64
--    pas_n             =  PAS* to VIC64
--    ds_n              =  DS* to VIC64

ENTITY address_decoder IS
        PORT (a01, a00, bclk, ts_n, tt1, tt0, siz1, siz0 : in bit;
              xfer_done_n, xfer_done_w_n, gate_oe_n, dedlk_s : in bit;
              a : in bit_vector(31 downto 28);
              asiz1, asiz0 : out bit;
              pas_n, ds_n : inout x01z);
attribute part_name of address_decoder:entity is "c22v10";
END address_decoder;

USE work.rtlpkg.all;

ARCHITECTURE operation OF address_decoder IS
          SIGNAL tt, siz : bit_vector(1 downto 0);
          SIGNAL pas_sig, ds_sig : bit;
          SIGNAL open_gate : bit;
          CONSTANT byte : bit_vector(1 downto 0) := "01";
          CONSTANT word : bit_vector(1 downto 0) := "10";
          CONSTANT lword : bit_vector(1 downto 0) := "00";
          CONSTANT acknow : bit_vector(1 downto 0) := "11";
          CONSTANT normal : bit_vector(1 downto 0) := "00";
BEGIN

     tt <= tt1 & tt0;
     siz <= siz1 & siz0;

     PROCESS BEGIN
          WAIT UNTIL bclk = '1';

-- At the start of a 68040 cycle, determine which signals should be activated
-- However, if the dedlk_s signal from the CYCLE TERMINATION PLD is active,
-- a transfer should not be begun to the VIC64.
```

Appendix B. Address and Cycle Decode PLDs (PALC22V10D) (continued)

```
IF (ts_n = '0') AND (dedlk_s = '1') THEN
     -- 964 Registers
     IF (a = "0001") AND (a01 & a00 = "00") AND (tt = normal) AND
       (siz = lword) THEN
          asiz1 <= '0';
          asiz0 <= '0';
          pas_sig <= '1';
     -- VIC64 Registers
     ELSIF (a = "0001") AND (a01 & a00 = "11") AND (tt = normal) AND
          (siz = byte) THEN
          asiz1 <= '0';
          asiz0 <= '0';
          pas_sig <= '0';
     -- A16 Addressing
     ELSIF (a = "0010") AND (tt = normal) THEN
          asiz1 <= '1';
          asiz0 <= '0';
          pas_sig <= '0';
     -- A24 Addressing
     ELSIF (a = "0011") AND (tt = normal) THEN
          asiz1 <= '1';
          asiz0 <= '1';
          pas_sig <= '0';
     -- A32 Addressing
     ELSIF (a = "1111") AND (tt = normal) THEN
          asiz1 <= '0';
          asiz0 <= '1';
          pas_sig <= '0';
     -- VIC64's Private Memory
     ELSIF (a = "0100") AND (tt = normal) THEN
          asiz1 <= '0';
          asiz0 <= '0';
          pas_sig <= '1';
     -- Interrupt Acknowledge
     ELSIF (tt = acknow) AND (siz = byte) THEN
          asiz1 <= '0';
          asiz0 <= '0';
          pas_sig <= '0';
     -- Not a cycle for us
     ELSE
          asiz1 <= '0';
          asiz0 <= '0';
          pas_sig <= '1';
     END IF;
END IF;

-- DS will follow whatever PAS does on the subsequent cycle

IF (pas_sig = '0') THEN
     ds_sig <= '0';
END IF;
```

Appendix B. Address and Cycle Decode PLDs (PALC22V10D) (continued)

```
-- If the cycle was a write cycle, the ds_sig must be pulled high to latch
-- data into the VIC64.  This is to assure that the local data hold time of
-- 0ns to the VIC64 is not violated.  If this were a cycle sending data
-- across the VMEbus, pulling ds_sig high before pas_n will not cause
-- problems because the slave board that is being written to would have
-- captured data when it asserted DTACK* to the VIC64.

               IF (xfer_done_w_n = '0') THEN
                    ds_sig <= '1';
               END IF;

-- When the cycle has been completed, the signals are all returned to their
-- inactive states.

               IF (xfer_done_n = '0') THEN
                    asiz1 <= '0';
                    asiz0 <= '0';
                    pas_sig <= '1';
                    ds_sig <='1';
               END IF;
          END PROCESS;

-- The pas_n and ds_n are driven by the VIC64 when it has access to its
-- private bus.  By looking at the state of the gate_oe_n signal, the owner
-- of the bus can be determined.  If the gate_oe_n signal is asserted (low),
-- the '040 has control of the bus and the pas_n and ds_n signals must be
-- active.

     open_gate <= NOT gate_oe_n;

     pas: bufoe PORT MAP (pas_sig, open_gate, pas_n, open);
     ds:  bufoe PORT MAP (ds_sig,  open_gate, ds_n,  open);

END operation;
```

Appendix B. Address and Cycle Decode PLDs (PALC22V10D) (continued)

```
-- ADDRESS DECODER design 2
--
-- The following table is a cross reference between the PLD port names and
-- the signals found on the physical IC's.
--
--    bclk              =  BCLK on 68040
--    a(31) to a(28)    =  Address bus on 68040
--    ts_n              =  TS "bar" on 68040
--    tt1               =  TT1 on 68040
--    tt0               =  TT0 on 68040
--    siz1              =  SIZ1 on 68040
--    siz0              =  SIZ0 on 68040
--    xfer_done_n       =  xfer_done_n from TERMINATION PLD
--    xfer_done_w_n     =  xfer_done_w_n from TERMINATION PLD
--    dedlk_s           =  dedlk_s from TERMINATION PLD

--    dead_n            =  dead_n to TERMINATION PLD
--    memsel_n          =  chip select for VIC64's private memory
--    strobe_n          =  STROBE* on CY7C964's
--    mwb_n             =  MWB* to VIC64
--    cs_n              =  CS* to VIC64
--    fciack_n          =  FCIACK* to VIC64

ENTITY address_decoder IS
        PORT (a01, a00, bclk, ts_n, tt1, tt0, siz1, siz0, xfer_done_n : in bit;
              xfer_done_w_n, dedlk_s : in bit;
              a : in bit_vector(31 downto 28);
              memsel_n, strobe_n, mwb_n, cs_n, fciack_n, dead_n : out bit);
attribute part_name of address_decoder:entity is "c22v10";
END address_decoder;

USE work.rtlpkg.all;

ARCHITECTURE operation OF address_decoder IS
          SIGNAL tt, siz : bit_vector(1 downto 0);
          CONSTANT byte : bit_vector(1 downto 0) := "01";
          CONSTANT word : bit_vector(1 downto 0) := "10";
          CONSTANT lword : bit_vector(1 downto 0) := "00";
          CONSTANT acknow : bit_vector(1 downto 0) := "11";
          CONSTANT normal : bit_vector(1 downto 0) := "00";

BEGIN

     tt <= tt1 & tt0;
     siz <= siz1 & siz0;

     PROCESS BEGIN
          WAIT UNTIL bclk = '1';

-- At the start of a 68040 cycle, determine which signals should be activated
-- This will be run only if we are not seeing a deadlock situation via the
-- dedlk_s signal.
```

Appendix B. Address and Cycle Decode PLDs (PALC22V10D) (continued)

```
IF (ts_n = '0') AND (dedlk_s = '1') THEN
     -- 964 Registers
     IF (a = "0001") AND (a01 & a00 = "00") AND (tt = normal) AND
       (siz = lword) THEN
          strobe_n <= '0';
          mwb_n    <= '1';
          cs_n     <= '1';
          memsel_n <= '1';
          fciack_n <= '1';
     -- VIC64 Registers
     ELSIF (a = "0001") AND (a01 & a00 = "11") AND (tt = normal) AND
          (siz = byte) THEN
          strobe_n <= '1';
          mwb_n    <= '1';
          cs_n     <= '0';
          memsel_n <= '1';
          fciack_n <= '1';
     -- A16 Addressing
     ELSIF (a = "0010") AND (tt = normal) THEN
          strobe_n <= '1';
          mwb_n    <= '0';
          cs_n     <= '1';
          memsel_n <= '1';
          fciack_n <= '1';
     -- A24 Addressing
     ELSIF (a = "0011") AND (tt = normal) THEN
          strobe_n <= '1';
          mwb_n    <= '0';
          cs_n     <= '1';
          memsel_n <= '1';
          fciack_n <= '1';
     -- A32 Addressing
     ELSIF (a = "1111") AND (tt = normal) THEN
          strobe_n <= '1';
          mwb_n    <= '0';
          cs_n     <= '1';
          memsel_n <= '1';
          fciack_n <= '1';
     -- VIC64's Private Memory
     ELSIF (a = "0100") AND (tt = normal) THEN
          strobe_n <= '1';
          mwb_n    <= '1';
          cs_n     <= '1';
          memsel_n <= '0';
          fciack_n <= '1';
     -- Interrupt Acknowledge
     ELSIF (tt = acknow) AND (siz = byte) THEN
          strobe_n <= '1';
          mwb_n    <= '1';
          cs_n     <= '1';
          memsel_n <= '1';
          fciack_n <= '0';
```

Appendix B. Address and Cycle Decode PLDs (PALC22V10D) (continued)

```
        -- Not a cycle for us
        ELSE
             strobe_n <= '1';
             mwb_n    <= '1';
             cs_n     <= '1';
             memsel_n <= '1';
             fciack_n <= '1';
        END IF;
   END IF;


-- This is the section of code that will be run if there is a deadlock.
-- If the decoded address/tt/siz information would have normally decoded
-- to a valid cycle, we send out the dead_n signal instead.  This lets the
-- TERMINATION PLD know that the 68040 is issuing a valid request to the
-- VIC64 but that the VIC64 can't be bothered cause it is currently finishing
-- a slave operation.

   IF (ts_n = '0') AND (dedlk_s = '0') THEN
        strobe_n <= '1';
        mwb_n    <= '1';
        cs_n     <= '1';
        memsel_n <= '1';
        fciack_n <= '1';

        -- 964 Registers
        IF (a = "0001") AND (a01 & a00 = "00") AND (tt = normal) AND
          (siz = lword) THEN
             dead_n <= '0';
        -- VIC64 Registers
        ELSIF (a = "0001") AND (a01 & a00 = "11") AND (tt = normal) AND
             (siz = byte) THEN
             dead_n <= '0';
        -- A16 Addressing
        ELSIF (a = "0010") AND (tt = normal) THEN
             dead_n <= '0';
        -- A24 Addressing
        ELSIF (a = "0011") AND (tt = normal) THEN
             dead_n <= '0';
        -- A32 Addressing
        ELSIF (a = "1111") AND (tt = normal) THEN
             dead_n <= '0';
        -- VIC64's Private Memory
        ELSIF (a = "0100") AND (tt = normal) THEN
             dead_n <= '0';
        -- Interrupt Acknowledge
        ELSIF (tt = acknow) AND (siz = byte) THEN
             dead_n <= '0';
        -- Not a cycle for us
        ELSE
             dead_n <= '1';
        END IF;
   END IF;
```

Appendix B. Address and Cycle Decode PLDs (PALC22V10D) (continued)

```
-- If the cycle was a write cycle, the strobe_n must be pulled high to latch
-- data into the '964's.  This is to assure that the local data hold time of
-- 5ns to the 964's is not violated.

               IF (xfer_done_w_n = '0') THEN
                    strobe_n <= '1';
               END IF;

-- When the cycle has been completed, the signals are all returned to their
-- inactive states.

               IF (xfer_done_n = '0') THEN
                    strobe_n <= '1';
                    mwb_n    <= '1';
                    cs_n     <= '1';
                    memsel_n <= '1';
                    fciack_n <= '1';
                    dead_n   <= '1';
               END IF;
          END PROCESS;

END operation;
```

Appendix C. Cycle Termination PLD (CY7C335)

```
-- CYCLE TERMINATION PLD
--
-- The following table is a cross reference between the PLD port names and
-- the signals found on the physical IC's.
--
--    bclk             =  BCLK on 68040
--    rw_n             =  R/W "bar" on 68040
--    rst_n            =  brd_rst_n_out from RESET PLD
--    liacko_n         =  LIACKO* from VIC64
--    dsack1_n         =  DSACK1* from VIC64
--    dsack0_n         =  DSACK0* from VIC64
--    lberr_n          =  LBERR* from VIC64
--    dedlk_n          =  DEDLK* from VIC64
--    memack_n         =  MEMACK* from private memory
--    memsel_n         =  MEMSEL* from ADDRESS DECODE PLD
--    strobe_n         =  STROBE* from ADDRESS DECODE PLD
--    mwb_n            =  MWB* from ADDRESS DECODE PLD
--    cs_n             =  CS* from ADDRESS DECODE PLD
--    fciack_n         =  FCIACK* from ADDRESS DECODE PLD
--    dead_n           =  dead_n from ADDRESS DECODE PLD

--    avec_n           =  AVEC "bar" to 68040
--    xfer_done_n      =  XFER_DONE "bar" to BUS ARBITRATION/ADDRESS DECODE PLD's
--    xfer_done_w_n    =  XFER_DONE_W "bar" to BUS ARBITRATION/ADDRESS DECODE PLD's
--    tea_n            =  TEA "bar" to 68040
--    ta_n             =  TA "bar" to 68040
--    tci_n            =  TCI "bar" to 68040
--    tbi_n            =  TBI "bar" to 68040
--    dedlk_s          =  Double registered (sync'ed) dedlk_n signal to
--                                ADDRESS DECODE PLDS


ENTITY cycle_termination IS
        PORT (bclk, liacko_n, dsack1_n, dsack0_n : in boolean;
              lberr_n, dedlk_n, memack_n, rst_n, rw_n : in boolean;
              dead_n : in boolean;
              memsel_n, strobe_n, mwb_n, cs_n, fciack_n : in boolean;
              avec_n, tea_n, ta_n, tci_n, tbi_n : out bit;
              dedlk_s : out bit;
              xfer_done_w_n, xfer_done_n : buffer bit);
attribute part_name of cycle_termination:entity is "c335";
END cycle_termination;

USE work.rtlpkg.all;
USE work.table_bv.all;

ARCHITECTURE operation OF cycle_termination IS
     SIGNAL any_access     : boolean;
     SIGNAL any_access_reg : boolean;
     SIGNAL cycle_end      : boolean;
     SIGNAL start, expired : bit;
     SIGNAL timer_count    : integer(0 to 7);
```

Appendix C. Cycle Termination PLD (CY7C335) (continued)

```
     SIGNAL liacko_n_reg    : boolean;
     SIGNAL dsack0_n_reg    : boolean;
     SIGNAL dsack1_n_reg    : boolean;
     SIGNAL dedlk_n_reg     : boolean;
     SIGNAL lberr_n_reg     : boolean;

BEGIN

     any_access <= NOT memsel_n OR NOT strobe_n OR NOT mwb_n OR NOT cs_n OR
                   NOT fciack_n;

     cycle_end <= (NOT memsel_n AND NOT memack_n) OR
                  (NOT strobe_n AND (timer_count = 3)) OR
                  (NOT mwb_n    AND (NOT dsack0_n_reg OR NOT dsack1_n_reg)) OR
                  (NOT fciack_n AND (NOT dsack0_n_reg OR NOT dsack1_n_reg)) OR
                  (NOT fciack_n AND (NOT liacko_n_reg)) OR
                  (NOT cs_n     AND (NOT dsack0_n_reg OR NOT dsack1_n_reg));

     controller: PROCESS BEGIN
          WAIT UNTIL bclk;
               liacko_n_reg   <= liacko_n;
               dsack0_n_reg   <= dsack0_n;
               dsack1_n_reg   <= dsack1_n;
               dedlk_n_reg    <= dedlk_n;
               IF dedlk_n_reg THEN
                    dedlk_s <= '1';
               ELSE
                    dedlk_s <= '0';
               END IF;
               lberr_n_reg    <= lberr_n;
               start          <= '0';
               xfer_done_n    <= '1';

               IF xfer_done_n = '0' THEN
                    any_access_reg <= FALSE;
               ELSE
                    any_access_reg <= any_access;
               END IF;

-- Normal beginning of a cycle starts the cycle timer and asserts the tbi_n
-- and tci_n to inhibit bursts and caching.

               IF any_access_reg THEN
                    tbi_n <= '0';
                    tci_n <= '0';
                    start <= '1';
               END IF;

-- Normal end to a write cycle will assert the xfer_done_w_n followed by an
-- assertion of xfer_done_n and ta_n.  Normal end to a read cycle is
-- xfer_done_n and ta_n asserted.

               IF cycle_end AND NOT rw_n AND (xfer_done_n = '1') THEN
                    xfer_done_w_n <= '0';
                    start <= '0';
```

Appendix C. Cycle Termination PLD (CY7C335) (continued)

```
                END IF;

                IF (cycle_end AND rw_n) OR (xfer_done_w_n = '0') THEN
                     xfer_done_w_n <= '1';
                     xfer_done_n <= '0';
                     ta_n <= '0';
                     start <= '0';
                END IF;

--  Error endings.  If dedlk_n_reg is active and an access is being attempted,
--  retry the cycle with ta_n and tea_n asserted together.  This will occur
--  only during a cycle.  If dead_n is active, we have already had an
--  initial deadlocked cycle and we are now in a sequence of retries to the
--  68040.
--  If there is a lberr_n assertion, just end the cycle with tea_n to
--  indicate an erred cycle.
--  xfer_done_n is also asserted in either case to shut off the selects in
--  the ADDRESS DECODE PLD's.

                IF (any_access_reg AND (NOT dedlk_n_reg)) OR (NOT dead_n) THEN
                     ta_n <= '0';
                     tea_n <= '0';
                     xfer_done_n <= '0';
                     start <= '0';
                END IF;

                IF any_access_reg AND (NOT lberr_n) THEN
                     tea_n <= '0';
                     xfer_done_n <= '0';
                     start <= '0';
                END IF;

--  liacko_n being asserted means that the processor should autovector the
--  current interrupt.

                IF (NOT liacko_n) THEN
                     avec_n <= '0';
                END IF;

--  Conclusion of the cycle.  xfer_done_n and all other outputs from this
--  PLD are placed in their inactive state.

                IF (xfer_done_n = '0') THEN
                     xfer_done_n <= '1';
                     tbi_n <= '1';
                     tci_n <= '1';
                     ta_n <= '1';
                     tea_n <= '1';
                     avec_n <= '1';
                     start <= '0';
                END IF;

-- Reset condition takes priority over any of the above assignments.
```

Appendix C. Cycle Termination PLD (CY7C335) (continued)

```
                IF (NOT rst_n) THEN
                     xfer_done_n <= '1';
                     xfer_done_w_n <= '1';
                     tbi_n <= '1';
                     tci_n <= '1';
                     ta_n <= '1';
                     tea_n <= '1';
                     avec_n <= '1';
                     start <= '0';
                END IF;

     END PROCESS;

     timer: PROCESS BEGIN
          WAIT UNTIL bclk;
                IF (timer_count /= 0) THEN
                     timer_count <= timer_count + 1;
                ELSE
                     timer_count <= 0;
                END IF;
                -------------------------------------
                IF start = '1' AND (timer_count = 0) THEN
                     timer_count <= 1;
                END IF;
                -------------------------------------
                IF xfer_done_n = '0' OR start = '0' OR (NOT rst_n) THEN
                     timer_count <= 0;
                END IF;
     END PROCESS;
END operation;
```

Appendix D. Bus Arbitration PLD (CY7C335)

```
-- BUS ARBITER PLD design
--
-- The following table is a cross reference between the PLD port names and
-- the signals found on the physical IC's.
--
--    pclk             =  PCLK on 68040
--    bclk             =  BCLK on 68040
--    bg_n             =  BG "bar" on 68040
--    lock_n           =  LOCK "bar" on 68040 (requires external pullup)
--    cs_n             =  cs_n from ADDRESS DECODE PLD
--    strobe_n         =  strobe_n from ADDRESS DECODE PLD
--    mwb_n            =  mwb_n from ADDRESS DECODE PLD
--    memsel_n         =  memsel_n from ADDRESS DECODE PLD
--    fciack_n         =  fciack_n from ADDRESS DECODE PLD
--    xfer_done_n      =  xfer_done_n from TERMINATION PLD
--    rst_n            =  brd_rst_n_out from RESET PLD
--    lbr_n            =  LBR* on VIC64
--    lbg_n_out        =  LBG* on VIC64
--    gate_oe_n        =  OE on GATE between 040 bus and VIC64 bus

ENTITY arbiter IS
        PORT (pclk, bclk, lock_n, cs_n, strobe_n, mwb_n : in bit;
              memsel_n, xfer_done_n, rst_n, lbr_n, fciack_n : in bit;
              bg_n, lbg_n_out, gate_oe_n : out bit);
attribute part_name of arbiter:entity is "c335";
attribute pin_numbers of arbiter:entity is "pclk:1 bclk:3";
END arbiter;

USE work.rtlpkg.all;

ARCHITECTURE operation OF arbiter IS
        signal lbr_n_reg1, lbr_n_reg2:bit;
        signal lbg_n : bit;
        signal selects:bit_vector(4 downto 0);
        type states is (reset, only040, slow_down040, both);
        signal arb_state: states;
        constant no_selects:bit_vector(4 downto 0) := "11111";
BEGIN

-- The local bus grant to the VIC64 must be removed within 1 VIC64 clock
-- cycle or the VIC64 would respond with an unsolicited bus request.

        lbg_n_out <= '0' WHEN ((lbr_n = '0') AND (lbg_n = '0')) ELSE '1';

-- This process captures the lbr_n signal from the VIC64 and double
-- registers it using the pclk signal.

        caputure_lbr: PROCESS BEGIN
            WAIT UNTIL pclk = '1';
            lbr_n_reg1 <= lbr_n;
            lbr_n_reg2 <= lbr_n_reg1;
        END PROCESS;
```

Appendix D. Bus Arbitration PLD (CY7C335) (continued)

```
-- gate_oe_n is triggered in a "Mealy" fashion to begin VIC64 cycles as
-- soon as possible.

        gate_oe_n <= '0' WHEN ((arb_state = slow_down040)
                                    OR
                               ((arb_state = only040) AND
                                   ((lock_n = '0' OR lbr_n_reg2 = '1') AND
                                    (selects /= no_selects)))
                                    OR
                               ((arb_state = both) AND
                                  ((lbr_n_reg2 = '1') AND
                                   (selects /= no_selects))))
                     ELSE '1';

        selects <= cs_n & strobe_n & mwb_n & memsel_n & fciack_n;

        arb_machine: PROCESS BEGIN
            WAIT UNTIL bclk = '1';
                CASE arb_state IS
                    WHEN reset =>
                        IF rst_n = '0' THEN
                            arb_state <= reset;
                            lbg_n <= '1';
                            bg_n <= '0';
                        ELSE
                            arb_state <= only040;
                            lbg_n <= '1';
                            bg_n <= '0';
                        END IF;
                    WHEN only040 =>
                        IF (lbr_n_reg2 = '0' AND lock_n = '1') THEN
                            arb_state <= both;
                            lbg_n <= '0';
                            bg_n <= '0';
                        ELSIF (lock_n = '0' OR lbr_n_reg2 = '1') AND
                              (selects /= no_selects) THEN
                            arb_state <= slow_down040;
                            lbg_n <= '1';
                            bg_n <= '1';
                        ELSE
                            arb_state <= only040;
                            lbg_n <= '1';
                            bg_n <= '0';
                        END IF;
                    WHEN slow_down040 =>
                        IF (xfer_done_n = '0') THEN
                            IF (lbr_n_reg2 = '0' AND lock_n = '1') THEN
                                arb_state <= both;
                                lbg_n <= '0';
                                bg_n <= '0';
                            ELSE
                                arb_state <= only040;
                                lbg_n <= '1';
```

Appendix D. Bus Arbitration PLD (CY7C335) (continued)

```
                    bg_n <= '0';
                END IF;
            ELSE
                arb_state <= slow_down040;
                lbg_n <= '1';
                bg_n <= '1';
            END IF;
        WHEN both =>
            IF (lbr_n_reg2 = '1') THEN
                IF (selects /= no_selects) THEN
                    arb_state <= slow_down040;
                    lbg_n <= '1';
                    bg_n <= '1';
                ELSE
                    arb_state <= only040;
                    lbg_n <= '1';
                    bg_n <= '0';
                END IF;
            ELSE
                arb_state <= both;
                lbg_n <= '0';
                bg_n <= '0';
            END IF;
        WHEN OTHERS => arb_state <= reset;
                       lbg_n <= '1';
                       bg_n <= '0';
    END CASE;
    IF (rst_n = '0') THEN
        arb_state <= reset;
    END IF;
END PROCESS;
END operation;
```

Appendix E. Interrupt Synchronizing PLD (22V10D)

```
-- INTERRUPT PLD
--
-- The following table is a cross reference between the PLD port names and
-- the signals found on the physical IC's.
--
--    bclk              =  BCLK on 68040
--    iplx_n            =  IPLx* from VIC64
--    tmx               =  TM2-TM0 on 68040
--    board_reset_n     =  brd_rst_n from RESET PLD
--    fciack_n          =  fciack_n from ADDRESS DECODE PLD
--    gate_oe_n         =  gate_oe_n from BUS ARBITRATION PLD
--    xfer_done_n       =  xfer_done_n from TERMINATION PLD

--    normal_cycle_n    =  OE to '244's driving A3-A1 from 68040
--    a3, a2, a1        =  a3-a1 to VIC64
--    iplx_out_n        =  IPLx "bar" on 68040

ENTITY interrupt_ctrl IS
       PORT (bclk, ipl0_n, ipl1_n, ipl2_n : in bit;
             tm2, tm1, tm0 : in bit;
             board_reset_n, fciack_n : in bit;
             gate_oe_n, xfer_done_n : in bit;
             normal_cycle_n : out bit;
             a3, a2, a1 : inout x01z;
             ipl0_out_n, ipl1_out_n, ipl2_out_n : buffer bit);
END interrupt_ctrl;

use work.cypress.all;
use work.rtlpkg.all;

ARCHITECTURE operation OF interrupt_ctrl IS
     signal addr_oe, ipl0_n_reg, ipl1_n_reg, ipl2_n_reg : bit;
BEGIN

-- Synchronize the incoming ipl signals from the VIC64 to eliminate skew
     PROCESS BEGIN
          WAIT UNTIL bclk = '1';
               ipl0_n_reg <= ipl0_n;
               ipl1_n_reg <= ipl1_n;
               ipl2_n_reg <= ipl2_n;
     END PROCESS;

-- If the board is in reset, the ipl signals must be driven high to configure
-- the driver capability in the 68040.  Otherwise, the following equations
-- will keep the ipl signals from changing to the 68040 during an acknowledge
-- cycle and will synchronize them.  When the acknowledge cycle is finished,
-- the ipl signals will return to inactive state before reading the current
-- input values from the VIC64.

     PROCESS BEGIN
          WAIT UNTIL bclk = '1';
               IF board_reset_n = '0' THEN
                    ipl0_out_n <= '1';
                    ipl1_out_n <= '1';
```

Appendix E. Interrupt Synchronizing PLD (22V10D) (continued)

```
                ipl2_out_n <= '1';
            ELSIF xfer_done_n = '0' THEN
                ipl0_out_n <= '1';
                ipl1_out_n <= '1';
                ipl2_out_n <= '1';
            ELSIF fciack_n = '0' THEN
                ipl0_out_n <= ipl0_out_n;
                ipl1_out_n <= ipl1_out_n;
                ipl2_out_n <= ipl2_out_n;
            ELSE
                ipl0_out_n <= ipl0_n_reg;
                ipl1_out_n <= ipl1_n_reg;
                ipl2_out_n <= ipl2_n_reg;
            END IF;
    END PROCESS;

-- The normal_cycle_n signal is low most of the time to enable the a3-a1
-- signals from the 68040 to the VIC64.  However, if we are in an interrupt
-- acknowledge cycle, we would steer the tm2-tm0 signals from the 68040
-- onto the a3-a1 signals on the VIC64 since the tmx signals indicate which
-- interrupt level is being acknowledged

    normal_cycle_n <= NOT (fciack_n AND (NOT gate_oe_n));

    addr_oe <= NOT fciack_n;
    a3_map: bufoe port map(tm2, addr_oe, a3, open);
    a2_map: bufoe port map(tm1, addr_oe, a2, open);
    a1_map: bufoe port map(tm0, addr_oe, a1, open);

END operation;
```

Document #: BUS0002–0194

Interfacing the CY7C611A with the VIC64

The popularity of the VMEbus and the Motorola 680x0 family of microprocessors has produced a large number of peripheral controllers with 680x0-compatible asynchronous local bus interfaces. Many of these parts are mature, proven, and inexpensive, making them attractive candidates for low-bandwidth I/O applications.

This application note describes an interface between the synchronous CY7C611A SPARC processor and asynchronous bus peripherals such as the Cypress Semiconductor VIC64 64-bit VMEbus interface chip. It is based on the design of a SPARC-based VIC64 VMEbus evaluation board developed by Cypress Semiconductor. Only the synchronous-to-asynchronous bus conversion logic is discussed within this application note; however, the full schematics of the board and all PLD design files are available from Cypress Semiconductor.

Related Documents

The reader may also wish to consult the following documents for additional information:

- *VIC068A/VAC068A User's Guide*
- *VIC64 and CY7C964 Design Notes*
- "Memory Protection and Address Exception Logic for the CY7C611A SPARC Controller" application note
- "Understanding the 361" application note
- Motorola's *MC6800 Family Reference*

With the exception of the Motorola document, these documents are available through your local Cypress Semiconductor field sales office.

Typical Asynchronous Bus Operation

Asynchronous buses operate using some type of handshake system. The processor presents or requests data from a peripheral and an acknowledge is generated by the selected device. The length of the processor cycle is determined by the performance level of the peripheral. The processor maintains a bus cycle until it receives an acknowledge.

With this type of bus, operation problems can occur if the processor attempts a cycle to an address region that does not select valid memory or peripherals. In this situation an acknowledge signal is not issued to the processor and the system operation halts.

To avoid this potential lock-up condition, most asynchronous bus protocols have a separate signal for acknowledging erroneous cycles. Assertion of this signal releases the processor from the pending bus cycle and can also be used to inform the system software that the bus cycle did not terminate properly. These cycles are typically known as bus error and memory exception cycles.

Memory Exception Cycles are Important

Asynchronous microprocessor buses are not unique in the inclusion of memory exception cycles. The CY7C601A and CY7C611A include a similar mechanism. In normal system operation, memory exceptions should not occur regularly. They can be used to furnish beneficial debug and system configuration information in some applications.

VMEbus applications where logic boards can be added and removed from systems often use the bus error mechanism to determine system configura-

tion. CPU board initialization software can *hunt* the VMEbus address regions, searching for other cards. Address regions that respond with normal acknowledge signals can then be further interrogated and initialized.

Overview of the CY7C611A Memory Interface

The CY7C611A is a 32-bit, four-stage, pipelined SPARC RISC integer processor. The processor is synchronous and, after initializing the pipeline, it can execute one instruction per clock cycle.

The CY7C611A memory interface consists of a group of signals that control memory loads/stores, pipeline control, and memory exception generation. These signals are listed in *Table 1*.

$\overline{\text{MHOLD(A/B)}}$

These two signals are logically ORed together within the CY7C611A. Asserting either of these signals (Low) freezes the processor's pipeline, causing the processor to remain on the same execution cycle. The $\overline{\text{MHOLDA}}$ and $\overline{\text{MHOLDB}}$ signals allow the processor to communicate with slow peripherals.

$\overline{\text{MDS}}$

MDS is used to strobe data or instructions into the processor after the pipeline has been frozen by the assertion of $\overline{\text{MHOLD(A/B)}}$. Asserting $\overline{\text{MDS}}$ with the pipeline frozen enables the processor to clock the information present on the external data bus into the processor. MDS is also used to strobe in the $\overline{\text{MEXC}}$ signal.

$\overline{\text{MEXC}}$

Asserting this signal (Low) informs the processor that the memory system could not supply the data or instruction requested. When the signal is asserted, either a data or instruction access trap occurs. The type of trap directly corresponds to the type of memory cycle in progress. $\overline{\text{MEXC}}$ is strobed into the processor by asserting the $\overline{\text{MDS}}$ signal.

Table 1. CY7C611A Memory Interface Signals

Name	Description	Type
$\overline{\text{MHOLD(A/B)}}$	Memory Hold A/B	Input
$\overline{\text{MDS}}$	Memory Data Strobe	Input
$\overline{\text{MEXC}}$	Memory Exception	Input
$\overline{\text{INULL}}$	Integer Unit Nullify	Output
$\overline{\text{WE}}$	Write Enable	Output
$\overline{\text{WRT}}$	Advanced Write	Output
$\overline{\text{RD}}$	Read Access	Output

INULL

The assertion of INULL (High) indicates that the memory cycle in progress is being nullified. Memory cycles are nullified when the processor determines the the current address is invalid or that the information being read is not required. This improves performance because no time is wasted communicating with slow peripherals or reloading cache line data that is not needed. INULL is asserted by the processor in the following situations:

- During the second cycle of any store operation. The same address is presented on the first and second cycle of all store operations, the second occurrence is nullified because it is not truly the *next* address being requested by the processor.
- On all traps. This nullifies the third instruction fetch after the trap is encountered, because the processor vectors to the appropriate trap handler.
- On a load with the hardware interlock active.
- On JMPL and RETT instructions.

$\overline{\text{WE}}$

Write Enable (active Low) indicates that the processor is performing a store operation. This signal is asserted in the second clock cycle of the store operation, the same cycle that the store data is presented.

WRT

Advanced Write (active High) notifies the external control logic that a store operation is in progress. The processor asserts this signal on the first cycle of the operation, before the data is available.

RD

Read Access (active High) indicates that a load cycle is in progress.

CY7C611A Load and Store Cycles

Two general bus cycles, load and store, are described at a high level of abstraction within this section. Many variations of these cycles exist.

When loading data, the processor supplies the address information on the rising edge of a the processor clock and expects the data on the next rising edge. The Read signal (RD) remains active (High) during the cycle with $\overline{WE}$ and WRT inactive (High and Low respectively).

The process for storing data is similar, but one additional clock cycle occurs before the processor presents the data. On the first cycle of the store the address is presented, RD is driven inactive (Low), and WRT is driven active (High). On the second clock cycle, the store address is again placed on the bus, WE is asserted (Low), WRT is deasserted (Low), and the data is placed on the bus. INULL becomes active after the falling edge of the second clock cycle, nullifying the second occurrence of the store address.

Figures 1 and *2* show Store Single and Load Single CY7C611A bus cycles. In general, the address execution unit operates one clock cycle ahead of the data unit. The cycles shown assume that the peripherals or memory are capable of operating at the performance level of the processor. The pipeline is never frozen using MHOLDA or MHOLDB, and both cycles terminate normally without generating memory exceptions.

Overview of the VIC64 Asynchronous Interface

The Cypress Semiconductor VIC64 is compatible with the 680x0 asynchronous microprocessor bus. It is a 64-bit VME interface chip capable of performing D16, D32, and D64 block transfers on the VMEbus at transfer rates up to 70 Mbytes/sec. The VIC64 and its associated control logic are also capable of performing Direct Memory Access (DMA) operations during VMEbus block transfers. VIC64 DMA operations generate 680x0-compatible bus cycles to transfer data to and from local memory.

The basic control signals required to communicate with a VIC64 or other generic 680x0-style peripherals are listed in *Table 2*.

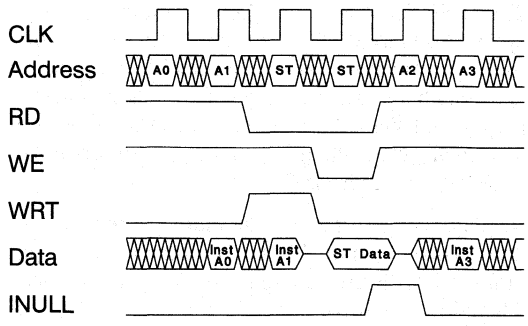


Figure 1. CY7C611A Store Single Operation

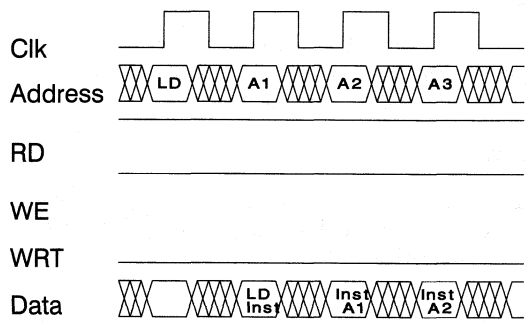


Figure 2. CY7C611A Load Single Operation

Table 2. 680x0 Basic Control Signals

Name	Description	Type
AS	Address Strobe	(Normally) Input
DS	Data Strobe	(Normally) Input
R/W	Read Write	(Normally) Input
DSACK0/1	Data Acknowledge	(Normally) Output
BERR	Bus Error	(Normally) Output

The signal types, input or output, have been referenced in a normal operating mode for dumb peripherals. Since the VIC64 is also capable of becoming a *bus master* during local DMA transfers, it can source AS, DS, and R/W as well as receive these signals. This also holds for the output signals DSACK1/0 and BERR. If the VIC64 is generating the bus cycle, these control signals become inputs.

AS

Address Strobe is asserted (Low) at the beginning of a bus cycle to indicate that a valid address is currently on the address bus. The address must remain constant while Address Strobe is active. Address Strobe remains active for the length of the bus cycle. On the VIC64 this signal is named Processor Address Strobe (PAS).

DS

The assertion of Data Strobe informs the receiving peripheral device or memory that it may place data on or extract data from the bus.

R/W

The Read/Write signal indicates the type of cycle in progress. This signal is High for read cycles and Low for write cycles.

DSACK0/1

The DSACK0/1 signals are driven by the peripheral device to tell the device performing the bus cycle that the data has been accepted or is available on the bus. Bus cycles persist until an acknowledge or BERR signal is detected. There is no limit to the length of this type of bus cycle. Many 680x0 peripheral devices have only a single acknowledge, often named *DTACK*.

VIC64 has two DSACK signals, 0 and 1, which adhere to the Motorola dynamic bus sizing convention and report the bus width, (8, 16, or 32 bits), of the peripheral acknowledging the bus cycle.

BERR

Asserting BERR terminates a pending bus cycle and forces the processor to trap to an exception handler. This signal terminates erroneous bus cycles. Many systems have bus timeout timers that monitor the length of all bus cycles and assert BERR if a cycle persists for the timeout period.

680x0 Asynchronous Read and Write Cycles

As with the corresponding section on the CY7C611A load and store cycles, the read and write cycles within this section are only described at the High level. Many variations of these cycles exists. Refer to the *VIC068A/VAC068A User's Guide* or the Motorola microprocessor documentation for more information.

Write cycles begin with an address being placed on the bus by the controlling processor or peripheral. The R/W signal is driven Low to indicate a write cycle. Address Strobe is asserted (Low), denoting the beginning of the cycle. One clock later, after the data has been placed on the bus, DS is asserted (Low). All signals remain stable in this state until a normal acknowledge, DSACK0/1, or error acknowledge is received (Low).

Reads cycles operate in a similar manner. An address is placed on the bus by the controlling peripheral and R/W is driven High to indicate a read cycle. AS and DS are driven Low simultaneously, informing the peripheral that data can be placed on the bus. These signals remain in this state until an acknowledge of some sort is received.

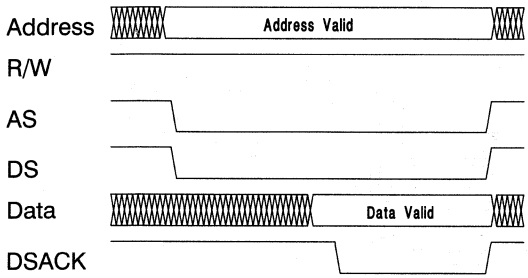


Figure 3. 680x0-Compatible Read Cycle

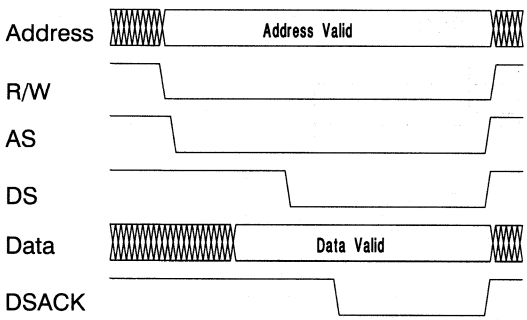


Figure 4. 680x0-Compatible Write Cycle

Figures 3 and *4* show typical bus cycles for asynchronous 680x0 peripheral devices like the VIC64.

Clear Differences in Cycle Types

As can be seen with even a cursory view of the two styles of bus cycles, interfacing between the CY7C611A and peripherals like the VIC64 can be challenging. In general, the problem is slowing the CY7C611A down to operate with the peripheral. This can be accomplished in a number of ways, each having its own set of considerations.

Pipeline Freezing Using MHOLDA/B

Per design, the CY7C611A contains control logic that allows the execution unit to be held for communication with slow memory devices or peripherals. The logic sequences required to suspend execution have some tight timing requirements. If an MHOLD signal is not asserted quickly enough, the processor advances to the next cycle. The CY7C611A, unlike the CY7C601, does not have an MAO pin. Therefore, if the processor does advance to the next cycle, there is no way to have it place the last address back on the bus. This can become a significant problem. Obviously other undesirable situations can occur when control logic does not or cannot meet necessary timing constraints.

These potential problems can be overcome by using high-performance logic like the CY7B336, CY7B337, CY7B338, and CY7B339 family.

Clock Stretching

Another method of interfacing the CY7C611A to slow memory and peripherals is a procedure known as clock stretching. The CY7C611A is a fully static microprocessor. This furnishes a simple method for slowing the processor down, simply by delaying or changing the duty cycle of the clock. The processor can be held within an execution state without asserting $\overline{\text{MHOLDA/B}}$. This technique allows execution to resume without strobing data into the processor with $\overline{\text{MDS}}$.

This procedure works well for peripherals with fixed access times. When the bus cycle begins, the clock is stretched. When the peripheral has completed the data transaction the clock is allowed to advance.

There are two subtle problems with this method of interfacing:

- Additional logic is required to operate with peripherals that are truly asynchronous in nature
- Memory exceptions cannot be generated because they require $\overline{\text{MDS}}$, $\overline{\text{MEXC}}$, and $\overline{\text{MHOLD}}$

Each of these problems becomes a significant issue when interfacing to the VIC64. Using the VIC64 to perform single-cycle processor transfers across the VMEbus has no guaranteed cycle time. The length of the cycle is directly dependant on the performance level of the slave plus the acquisition time required to obtain the VMEbus. Therefore, using a fixed clock-stretch cycle time would either be too short for slow slave boards, or a significant performance barrier when communicating with faster boards.

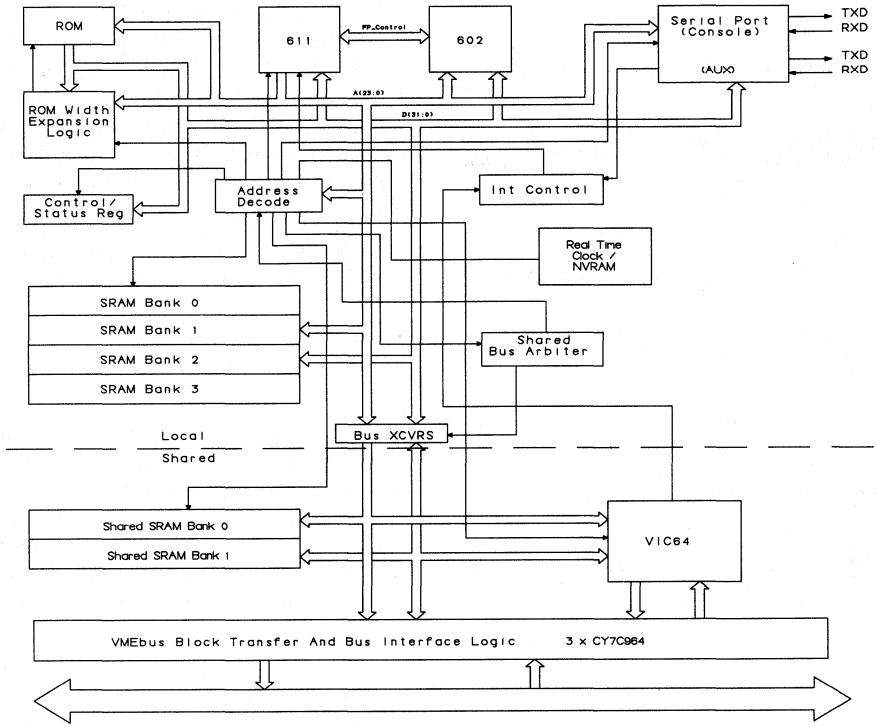


Figure 5. CY7C611A / VIC64 VMEbus Board Block Diagram

Bus errors are also an integral part of the VMEbus and the VIC64's operation. The inability to use this feature would significantly limit the functionality of many systems. Mapping this function into an interrupt is not desirable because if interrupts are disabled, or if interrupt latency is encountered because higher-priority interrupts are pending, the software's ability to determine the cycle that caused the error is hampered.

CY7C611A/VIC64 VMEbus Board

The CY7C611A/VIC64 evaluation board is a typical single-board computer with the following features:

- 25-MHz CY7C611A embedded-control SPARC RISC processor
- 25-MHz CY7C602 floating-point unit
- 64 Kbytes to 4 Mbytes of local SRAM
- 64 Kbytes to 2 Mbytes of dual-port SRAM
- 128 Kbytes to 512 Kbytes of EPROM
- MC68681 DUART
- 2 Kbytes of non-volatile storage
- Time-of-day clock calendar
- Split address and data bus for high-performance VMEbus block transfer operation

The block diagram of the board is shown in *Figure 5*. The local SRAM on the board operates at zero wait states, removing the need for an instruction or data cache. With the exception of the local SRAM and a

system control/status register, all other peripherals operate using asynchronous 680x0-style bus cycles.

Having all peripherals operate using one of the two cycle types simplifies the interface and control logic. The 680x0-style cycle is essential since the VIC64 and MC68681 DUART communicate on this type of interface. The shared SRAM also needs to operate using this type of cycle to be compatible with the VIC64 during VMEbus block transfer DMA operations. It is then simple to adapt other slow peripherals (ROM, non-volatile SRAM, and Time-of-Day clock) to the slow, 680x0-style bus cycle.

The CY7C611A-to-680x0 Bus Converter

As discussed in the previous sections, there was a strong desire to build an interface that was logically simple but preserved memory exception capability. The scheme was implemented as a hybrid technique using the CY7C611A pipeline freezing and memory exception logic along with a clock-stretching technique.

The control logic is implemented within two PLDs, a CY7C361 and a 22V10B, operating as pseudo master slave devices. The logic is split between two devices because of other functionality needed on the board, which is well suited for the CY7C361. If these other functions were removed from the CY7C361, the entire synchronous to asynchronous conversion logic could fit within the CY7C361. However, the CY7C361 on the CY7C611A/VIC64 board provides:

- Generation and control of clocks for the processor and peripherals
- Local bus arbitration for the VIC64 and CY7C611A
- Synchronization of asynchronous signals, which is needed for the slave 22V10B

This bus conversion scheme operates as follows. The processor begins execution and an address is presented, latched, and decoded. If the address region decodes to a slow 680x0-compatible cycle, the clock to the processor and control logic is stretched. If the cycle terminates normally, the clock is re-enabled to the processor and to control logic, which advances to the next execution cycle. If the cycle terminates in a memory exception or bus error, $\overline{\text{MHOLDA}}$ is asserted to the processor, freezing the pipeline, and the clock is re-enabled. With the pipeline frozen and the processor and control logic clock running, $\overline{\text{MEXC}}$ and $\overline{\text{MDS}}$ are asserted to the processor, generating the exception.

Clock Control Using The CY7C361

To simplify interface design and maximize performance, microprocessor control logic typically needs to operate at twice the clock frequency of the processor. Even with the relatively slow 25-MHz clock frequency of the CY7C611A, routing, managing skew, and operating TTL control logic at 50 MHz can significantly increase the complexity of a design.

To eliminate this problem, the CY7C361 was selected as a clock-generation device. The CY7C361 is an ultra high speed PLD that features an internal clock doubler, double input registers for metastable hardening of asynchronous inputs, and 32 general-purpose state macrocells. While this is not a typical application for a PLD, the CY7C361 has a pin-to-pin skew of 2 ns maximum.

Operating the CY7C361 at 50 MHz externally and 100 MHz internally allows the generation of three different 25-MHz clocks. While the system still requires a 50-MHz clock, the CY7C361 is the only device operating from it, simplifying routing and termination problems. Since no other device on the board operates from the 50-MHz clock, no relationship needs to be maintained between the CY7C361 clock input and output pins, removing the clock-to-output propagation delay from the timing analysis. The 25-MHz clocks operate all sequential logic on the board with the exception of the 3.68-MHz clock needed by the MC68681 DUART for baud-rate generation.

The Clock-Generation Machine

The clock-generation state machine within the CY7C361 has the following input and output signals:

NNULL (Input)

This synchronous input is a conditioned active-Low signal formed by combining the CY7C611A INULL and the CY7C602A FNULL signals. FNULL is the corresponding nullify signal from the floating-point unit. It operates in the same manner as INULL. The NNULL signal is used to filter out the nullifies that occur during every store cycle. The store nullifies were a don't care for the board's control logic since the signal is generated to nullify the second occurrence of the store address.

$$\overline{\text{NNULL}} = (\text{INULL OR FNULL}) \text{ AND LWE}$$

INULL and FNULL are active High and LWE is simply the latched WE signal from the CY7C611A. LWE is latched on the rising edge of CPUHCLK.

LWRT (Input)

This synchronous input is the latched WRT signal from the CY7C611A. This signal is latched on the rising edge of CPUHCLK.

$\overline{\text{MHOLDA}}$ (Input)

This is the synchronous CY7C611A $\overline{\text{MHOLDA}}$ signal. This signal is generated by the 22V10B that generates Motorola-style bus cycles.

DONE (Input)

A synchronous signal generated elsewhere within the CY7C361 that indicates that a Motorola bus cycle has been acknowledged. All acknowledges returned from the board are asynchronous signals. Double input registers on the CY7C361 are used to synchronize it for state logic use. DONE is active Low and is asserted if the cycle terminates normally or in a bus error.

HOLD (Input)

HOLD is a synchronous signal from the address decoding logic indicating that the selected peripheral requires a slow, 680x0-style bus cycle and that the CY7C611A and control logic clock must be stretched.

CPUCLK (Output)

This is a free running 25-MHz clock that is used for much of the sequential control logic. Although the name may imply it, this clock is not used by the CY7C611A CPU.

CPUHCLK (Output)

This is a 25-MHz stretched version of CPUCLK. It is the clock used to control the CY7C611A, CY7C602A, and address decode/latch logic. This clock is stretched by the CY7C361 if the address decoding logic reports that a slow, asynchronous cycle should be performed. When this clock is operating, it is always in phase with CPUCLK.

CPU90 (Output)

This is a 25-MHz free-running clock that lags the CPUCLK by 90 degrees (1/2 cycle). This clock, in conjunction with CPUCLK, provides the board control logic with a time base with 10 ns of resolution.

START (Output)

The assertion of START (Low) informs the slave 22V10B state machine that a 680x0 cycle should begin. This signal is not actually an output of the state machine, but of external state logic that is controlled by this state machine.

The 680x0 cycle cannot start at the beginning of the stretched clock cycle because of the latency associated with the assertion of INULL and FNULL from the CY7C611A and CY7C602A. If the NNULL signal is not asserted 40 ns after the clock stretching has started, the bus cycle is deemed valid and the START is asserted.

The state diagram of this machine is shown in *Figure 6.*

The transition equations for this machine are:

1. (State3) OR (HOLD AND /MHOLDA AND LWRT)
2. State3 AND /HOLD AND /MHOLDA AND /LWRT
3. State7 AND /DONE AND NNULL
4. (State7) OR (DONE AND /NNULL)

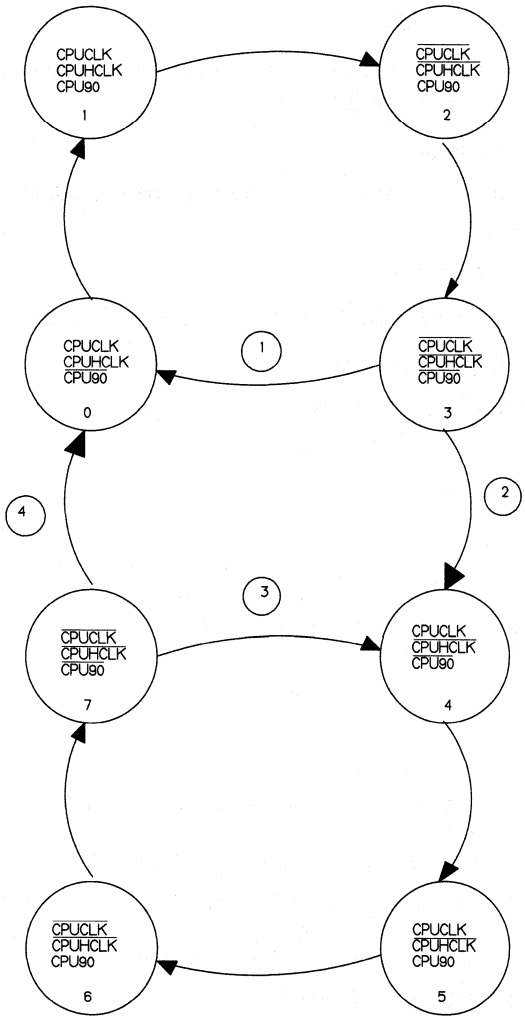


Figure 6. Clock-Generation State Machine

Clock-Stretch Machine Operation

This machine has two main paths of operation. The first is a sequence in which all three clock outputs are operating, sequencing through states 0, 1, 2, 3, and back to 0. The second is a clock-stretched path sequencing through states 4, 5, 6, 7, and back to 4. This machine enters state 0 at the deassertion of reset and therefore always begins execution generating all clocks.

While within state 3, the machine samples the HOLD, $\overline{\text{MHOLDA}}$, and LWRT (Latch Advanced Write signal). HOLD High indicates that memory address on the bus is not selecting a slow device and that the clock should not be stretched. $\overline{\text{MHOLDA}}$ Low in this circuit indicates that a memory exception has occurred, and that the processor clock should continue to operate. The clock must be re-enabled so that $\overline{\text{MDS}}$ and $\overline{\text{MEXC}}$ can strobe the memory exception condition into the device.

The third signal sampled is Latched Advanced Write (LWRT). When this signal is High it indicates that the processor is starting a store cycle. The CY7C611A does not provide data to be stored until the second clock cycle of the operation. Therefore the processor must be advanced by at least one clock to place the data on the external bus. LWRT High and MHOLDA Low cause the processor clock to continue operating even if the address decoding logic asserts HOLD Low, indicating that the cycle should be stretched.

If HOLD is asserted (Low) and neither LWRT or MHOLDA are in their active states, then the state machine moves to state 4 and the clock is disabled to the processor and control logic. The other output clocks (CPUCLK and CPU90) continue to operate as the machine sequences through states 4, 5, 6, and 7. State 7 is also a decision-making state within this machine. At this point the machine either continues stretching or re-enables the processor and control logic clock. The clock is only re-enabled if either of two conditions (NNULL or DONE are detected active (Low)) is true.

NNULL is asserted if the current cycle is not a store cycle and the CY7C611A or CY7C602A nullifies the cycle. The processor does not generate the INULL or FNULL until late in the cycle. Therefore nullified asynchronous bus cycles end up being stretched for 40 ns before this is determined. If the stretched bus cycle is nullified by the CY7C611A or CY7C602A, the NNULL is asserted before the machines samples the signal in state 7. If NNULL has not been asserted upon entering state 7 for the first time after clock stretching has begun, START is as-

serted (Low) . This signals the slave 22V10B that a 680x0 style cycle should begin.

680x0 Bus Cycle Machine

A Mealy state machine implemented in a 22V10B performs the 680x0-compatible asynchronous bus cycle and asserts $\overline{\text{MHOLDA}}$, $\overline{\text{MDS}}$, and $\overline{\text{MEXC}}$ to the CY7C611A if a bus error is detected. This machine uses the CY7C361 to synchronize all asynchronous signals and therefore operates in a totally synchronous environment. This simplifies the implementation of the machine and enhances performance.

The input and output signals for the machine are:

START (Input)

This signal is asserted (Low) by the CY7C361 clock control state machine to indicate that a Motorola bus cycle should start. This signal remains asserted until the bus cycle completes.

LRD (Input)

This is the latched Read Access (RD) signal from the CY7C611A.

SPARC_WB (Input)

This is an output from the local bus arbiter for the board. If the asynchronous bus cycle is accessing something on the shared half bus, this signal is asserted by the address decode logic. If this signal is active (Low) the bus cycle must not begin until the grant signal, SPARC_GB, is asserted, granting access to the shared bus.

SPARC_GB (Input)

This active-Low signal is the bus grant signal from the local bus arbiter.

DONE (Input)

This is the synchronous active-Low signal from the CY7C361, indicating that some form of asynchronous cycle acknowledge has been received.

BERR (Input)

An asynchronous active-Low input that is combined with DONE for synchronization.

AS (Output)

This is the active-Low 680x0 compatible address strobe.

DS (Output)

This is the active-Low 680x0 compatible data strobe

$\overline{\text{MHOLDA}}$ (Output)

This is the $\overline{\text{MHOLDA}}$ signal to freeze the pipeline of the CY7C611A and CY7C602A.

$\overline{\text{MDS}}$ (Output)

This output is the $\overline{\text{MDS}}$ and $\overline{\text{MEXC}}$ signals to the CY7C611A. Since this bus control cycle only requires $\overline{\text{MDS}}$ for memory exception cycles, it was possible to reduce these two into a single output on the machine.

The state diagram of the machine is shown in *Figure 7.*

The transition equations for this machine are:

1. (/START AND /LRD AND /SPARC_WB AND /SPARC_GB) OR (/START AND /LRD AND SPARC_WB)
2. (/START AND LRD AND /SPARC_WB AND /SPARC_GB) OR (/START AND LRD AND SPARC_WB)
3. /DONE AND /BERR
4. /DONE AND BERR

Machine Operation

This machine resets to state 0 and waits for the assertion of the START signal from the CY7C361. When this signal is active, the machine samples the states of the CY7C611A Latched Read Access signal (LRD), the Local Bus Request signal (SPARC_WB), and the Local Bus Grant signal (SPARC_GB). The state of LRD instructs the 22V10B to perform either a read or write cycle. If SPARC_WB is asserted (Low), then the

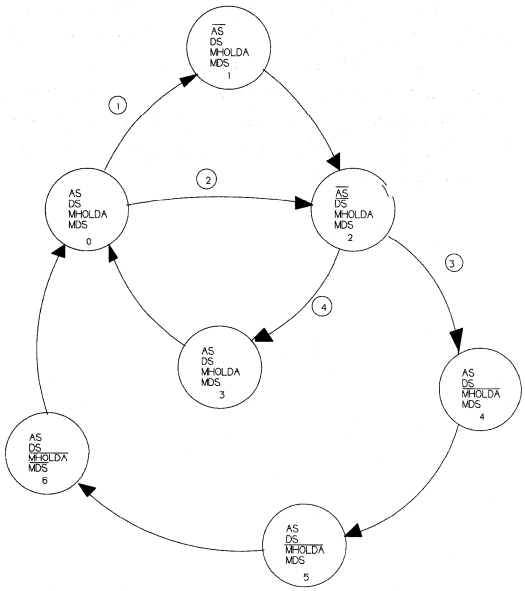


Figure 7. 680x0 Bus Cycle State Machine

peripheral device being accessed is on the shared section of the board. The bus cycle cannot start until a grant has been issued by the local bus arbiter. When SPARC_GB is asserted (Low), the shared bus has been granted to the CY7C611A. Read or write cycles that access peripherals on the local section of the card (CY7C611A access only) do not need permission from the local bus arbiter. CY7C611A local accesses can occur simultaneously with VIC64 local DMA accesses.

When the conditions have been met to start a cycle, AS strobe is asserted (Low). If the cycle is a read, DS is also asserted (Low). On write cycles, the assertion of DS is delayed one clock cycle to mimic the 680x0 cycle. This may not be necessary for many peripherals since, unlike Motorola processors, the CY7C611A has already placed the data on the data bus before the assertion of AS.

The machine moves to state 2 were it waits for the assertion of DONE (Low) from the CY7C361. DONE is generated by combining the board's asynchronous peripheral acknowledge and bus error signals. This combined signal is then run through the double input register structure of the CY7C361 to synchronize it. Double registering these asynchronous signals with the CY7C361 is the most efficient manner of synchronization as these registers are being clocked internally at 100 MHz. The entire double-register synchronization process takes only 20 ns. DONE is further qualified with the appropriate 25-MHz clock from the CY7C361 so that it can be considered completely synchronous to the 22V10B.

When the 22V10B detects DONE asserted, it samples the BERR input. If BERR is inactive (High), then the cycle terminates normally. The machine drives AS and DS inactive and advances back to through state 3 to state 0 to prepare for the next cycle. State 3, a delay state, is necessary to allow the control logic recovery time before the next cycle begins. This is a Mealy machine and removing state 3 would allow situations to occur were AS and DS would not meet the minimum High times required by slow peripherals. Refer to the waveforms on the following pages, which show the control signal sequencer for normally terminated and memory exception cycles.

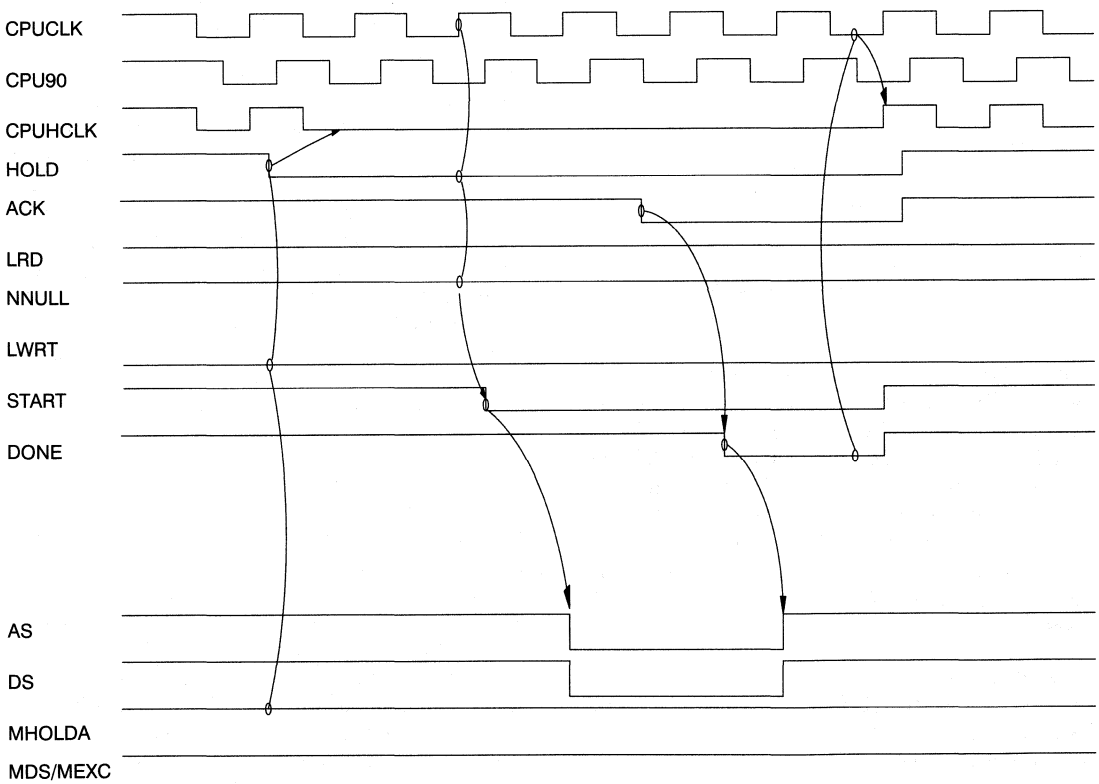
If BERR is active (Low) when DONE is asserted, the asynchronous cycle is terminated in a bus error or memory exception. The 22V10B asserts MHOLDA (Low) to the CY7C611A freezing the pipeline. The assertion of this signal informs the CY7C361 to re-enable clocking to the CY7C611A and control logic so that the exception can be strobed in using $\overline{\text{MDS}}$ and $\overline{\text{MEXC}}$. $\overline{\text{MDS}}$ and $\overline{\text{MEXC}}$ are then asserted simultaneously to the CY7C611A, indicating that a memory exception has occurred. Since memory exceptions are the only cycles that freeze the CY7C611A pipeline, $\overline{\text{MDS}}$ and $\overline{\text{MEXC}}$ are always asserted simultaneously. This allows the generation of a single signal, rather than two, freeing up an output on the PLD.

Conclusion

This hybrid bus conversion has worked well on the CY7C611A/VIC64 VMEbus board. Using the CY7C361 as a clock-generation device, allowing all of the logic on the board to operate from the relatively slow 25-MHz clocks, greatly simplified the timing analysis without sacrificing performance. The CY7C361 also provides logic functions that are not discussed within this application note. In many applications it may be possible to move the slave 680x0 bus-cycle generation state logic into the CY7C361. This would reduce the bus conversion logic to a single device.

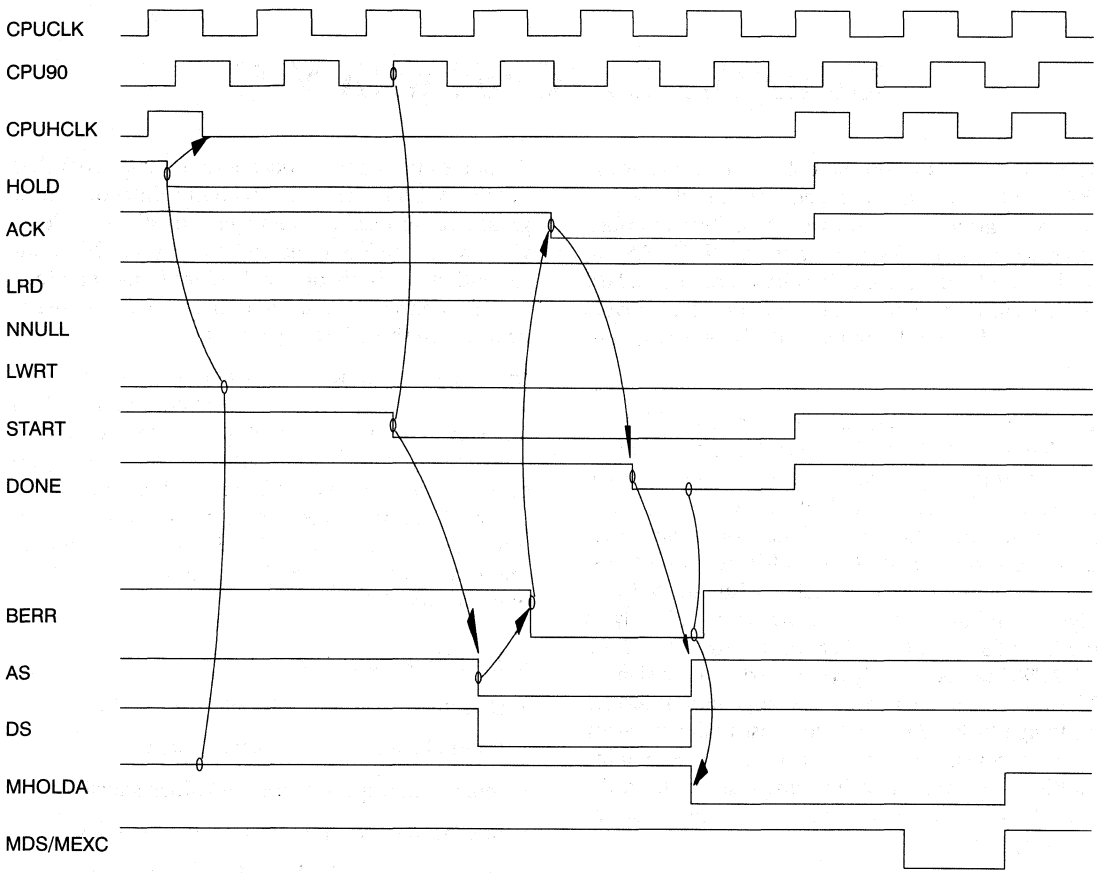
Output Waveforms

CY7C611A to 680x0 Normally Terminated Cycle



Output Waveforms (continued)

CY7C611A to 680x0 Memory Exception Cycle



Document #: BUS0007–0992

Using the CY7C964 with VIC

The CY7C964 is a flexible collection of byte-wide (8-bit) transceivers, latches, counters, multiplexers, and comparators that provide VMEbus interface designs with a low-cost alternative to PLDS, ASICs, or discrete logic devices. It is based on a standard cell design that incorporates patented line drivers for reduced ground bounce and high-noise immunity.

The CY7C964 is a companion part to the Cypress VIC068A and VIC64 VMEbus Interface Controller devices. It is compatible with all operating modes of either device, including dual address path, block transfers, block transfer initialization cycles, local DMA control, and D64 64-bit VMEbus block transfers (when used in conjunction with the VIC64). Signal naming conventions correspond directly to the VIC068A/VIC64 buffer control signals and the CY7C964 can be directly connected to these signals. The device can also be used as a generic interface building block. CY7C964s are cascadable allowing easy interfacing to any width bus. By combining multiple logic functions into one discrete part, the CY7C964 saves board space and reduces power consumption.

The CY7C964 has two main operating modes, byte width and word width. The byte-width configuration of the device has 8 local address, 8 local data, 8 VMEbus address, and 8 VMEbus data signals. In the byte-width modes, two methods are available for loading the VMEbus master block transfer address counters. This counter is loaded using the nominal VIC block transfer initiation cycle, or alternatively from the local data bus. Loading the VMEbus master block transfer counter from the local data bus decouples the board's local address map from that of the VMEbus. This allows boards that consume a great deal of local address space to still be able to view the entire VMEbus address region. More information on both of the VMEbus master block transfer address counter initialization techniques is provided in the following sections.

In word-width mode the local and VMEbus data signals change to 16-bit address or data paths. This mode expands all of the features of the device to 16 bits in width, with the exception of the D64 multiplexer. This multiplexer is disabled in word-width mode. Since protocols such as block transfer initiation cycles remain compatible in word-width mode, the device becomes useful as a 16-bit bus interface block for non-VMEbus applications.

CY7C964 Features

- Directly connects to VIC068A or VIC64
- Internal counters for block transfers
- Internal multiplexers for D64 block transfers
- Internal comparators for address decoding
- Supports VIC068A/VIC64 dual address path option
- Supports cascadable operation
- Directly drives VMEbus address and data
- Directly drives local address and data bus signals
- Reduces components for compact board design
- Low power requirements
- Available in 64-pin QFP

CY7C964 Block Diagrams

The CY7C964 is an array of optimally controlled counters, comparators, general registers, and multiplexers. A small amount of state logic is also present within the device. This state logic monitors bus cycles that are issued to the device and places the component in the appropriate mode. The state logic is implemented as an asynchronous rather than a synchronous sequential state machine. These hidden internal-state or configuration bits are set and cleared automatically by monitoring the arrival times of various input signals.

The configuration bits, and other input signals to the device, select the operating mode (byte width or word width), as well as the initialization method for the internal VMEbus master and local block transfer counters. The block diagrams in *Figures 1, 2,* and *3,* show an equivalent internal representation of the device for each operational mode.

Byte-Width Mode I

In this mode, the CY7C964 operates as a conventional byte-width slice of VIC compatible interface logic. The device conforms to the standard VIC block transfer initiation cycle and includes multiplexers for D64 block transfer operations and comparison logic for VMEbus address decoding.

Counters C1, C2, and C3, latch L8, and multiplexers S3 and S5 form the core of the block transfer address generation logic. C1 is the local master block transfer address counter, C2 is the VMEbus slave block transfer address counter, and C3 is the VMEbus

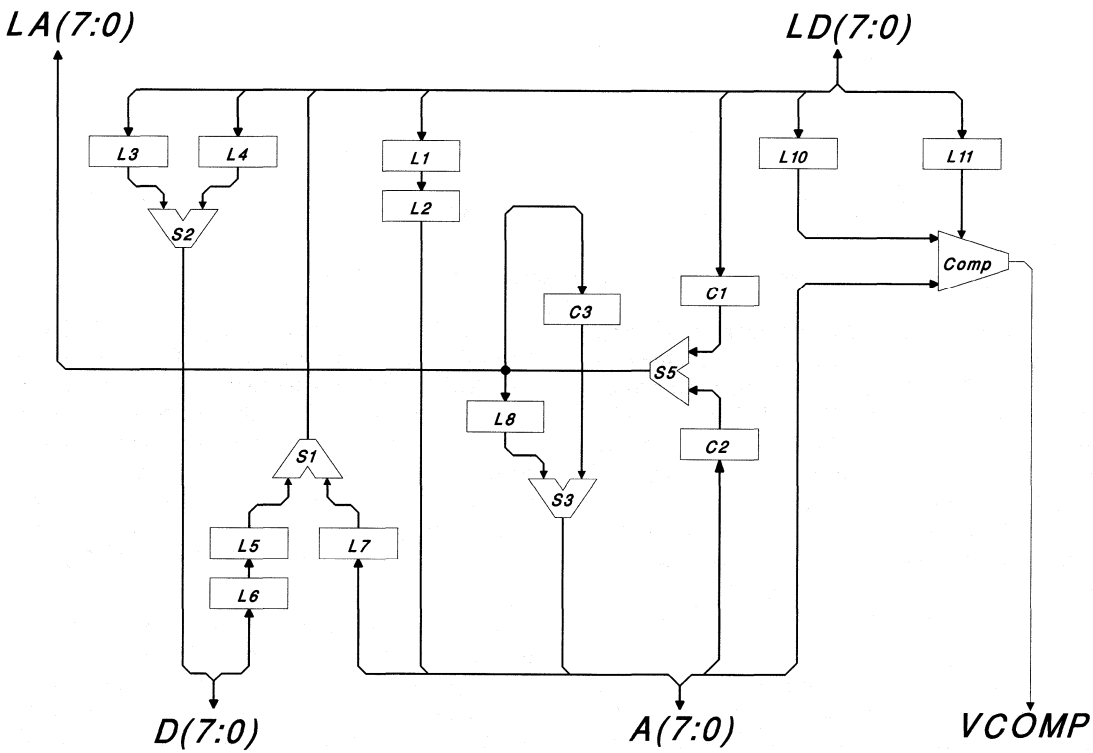


Figure 1. CY7C964 Byte-Width Mode I Block Diagram

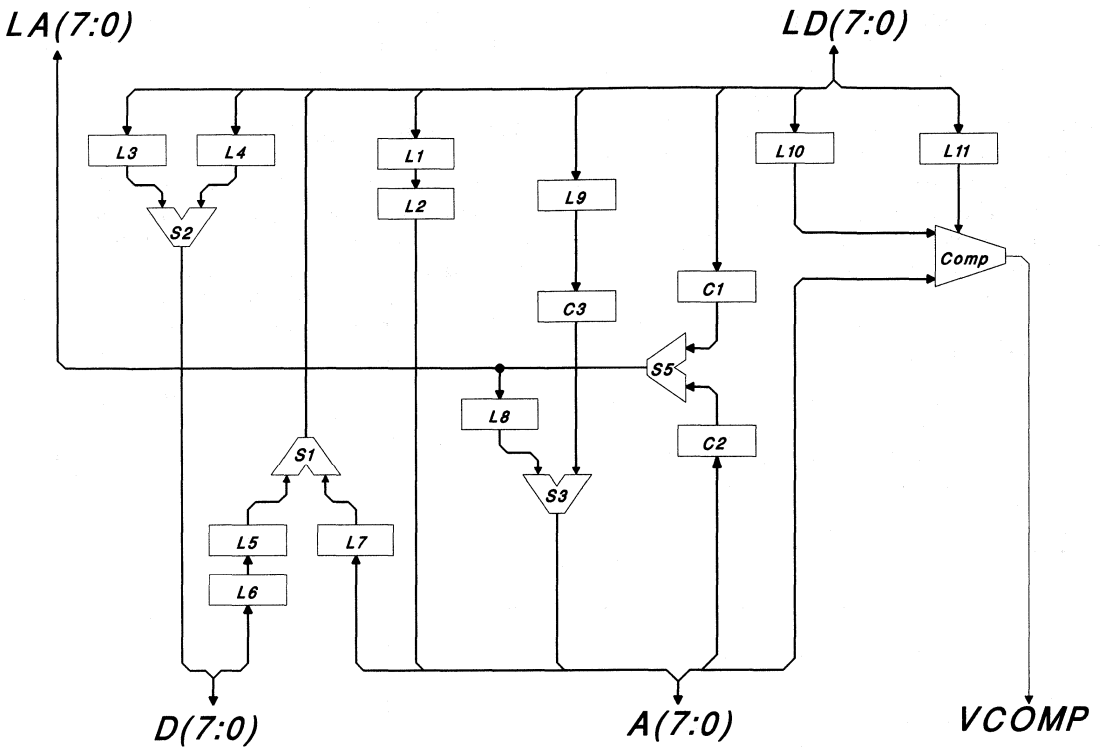


Figure 2. CY7C964 Byte-Width Mode II Block Diagram

master block transfer address counter. As shown in the block diagram, counter C1 loads from the local data bus LD[7:0], C3 loads from the local address bus LA[7:0], and counter C2 loads from the VMEbus address bus A[7:0]. Multiplexer S5 selects the source for the local address either through C2 (which is also used for single cycle operations) or C1. Latch L8 and multiplexer S3 provide the support for the Dual Address Path feature. Single cycle VMEbus master transfers can occur using L8 during the interleave periods of master block transfers without corrupting the contents of C3.

Latches L10, L11, and comparator COMP form the VMEbus address comparison logic. L11 is the Address Mask register that enables and disables bits of the address comparator. L10 is the Address Comparison register which contains an 8-bit value that is matched against A[7:0]. When the enabled bits of L10 match the corresponding signals of A[7:0], the VCOMP* output is asserted (Low). Writing 1's to all bits of L11 disables the comparison logic. In this case all values of A[7:0] match, causing the VCOMP output to be asserted continuously. Loading comparison register L10 clears and enables all bits of the mask register L11. Therefore, during system initialization, comparison register L10 must be loaded first, then bits can be disabled within mask register L11.

Latches L1, L2, L3, and L4, and multiplexer S2 combine to form a high performance D64 block transfer data pipeline and multiplexer. During D64 block transfer operations data is fetched from local memory and transferred to latch L1. A second memory fetch is required to assemble the 64-bit word. This data is stored in L3. When L3 is updated, the data in L1 is moved to L2. This allows the VIC

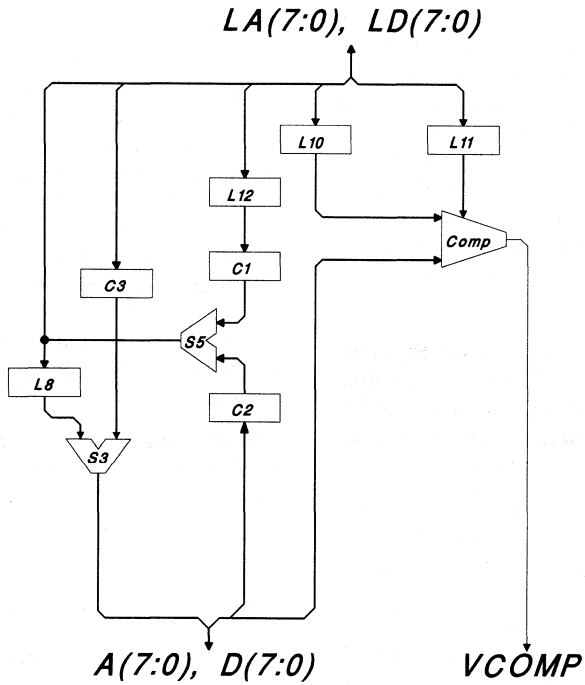


Figure 3. CY7C964 Word Width Mode Block Diagram

to prefetch the next word of information from local memory while the VMEbus D64 data transfer operation is in progress.

Latches L5, L6, L7 and multiplexer S1 form the VMEbus D64 block transfer data demultiplexer. Data is latched from the VMEbus into latches L6 and L7, simultaneously. The data is then moved to the local data bus through multiplexer S1.

This is the nominal operating mode of the CY7C964. Control signals connect to the corresponding buffer control signal on the VIC, with the exception of the DENIN* and DENIN1* inputs. Refer to the following section, Interfacing to the VIC64 and VIC068A, and to the *VIC64/CY7C964 Design Notes* for additional information on DENIN*, DENIN1*, and other control signals.

Byte Width Mode II

This mode is nearly identical to the previous byte-width mode with one exception. The VMEbus master block transfer counter, C3, loads from the local data bus LD[7:0] rather than the local address bus LA[7:0]. The main benefit of this operating mode is that the entire VMEbus address space is available for data transfers. Loading C3 from the local address bus may preclude some addresses from the VMEbus, because they are being decoded locally. For example if EPROM is located at address 0x00000000, this address may not be accessible across the VMEbus. Using the CY7C964 in this mode requires performing one additional bus cycle during the block transfer initiation.

The CY7C964 operates in byte-width mode if the BLT* and MWB* input signals on the CY7C964 are swapped. In other words, the MWB* signal on the

VIC connects to the BLT* input on the associated CY7C964s. The same rule applies for the BLT* output on the VIC. It connects to the MWB* input on the CY7C964s. The CY7C964 monitors the arrival time of these two signals and expects to load the master block transfer counter from the local data bus LD(7:0) if BLT* is asserted prior to MWB*. Swapping these two signals does not change the operation of any other feature on the device, however, there are two things to consider when using this mode.

The address decode signal that drives MWB* on the CY7C964 connects to the BLT* output on the VIC. This signal should be driven with an open collector or three-state device to allow the VIC to control the signal during block transfer operations.

Block transfer initiation cycles also change. The VMEbus master block transfer address counter loads from the local data bus one cycle before the actual block transfer initiation cycle. The subsequent cycle is a typical block transfer initiation cycle with the local data bus containing the local DMA address. The local address bus is ignored by the CY7C964s during this cycle, but not by the VIC. The low-order byte of the local address bus LA[7:0] must contain the correct VMEbus master block transfer address. This is necessary because the VIC cannot be programmed to load the VMEbus master block transfer address from the local data bus. For more information on Byte Width Mode II refer to the *VIC64/CY7C964 Design Notes*.

Word-Width Mode

The second main operating mode of the CY7C964 is the word-width mode. This mode of the device works well for VMEbus address control functions. All of the address related functions (local master block transfer counter, C1, VMEbus slave block transfer counter, C2, VMEbus master block transfer counter C3, and the address comparison logic) expand to 16 bits. The address and data buses on the part combine to form two 16-bit buses. A high-drive-strength, 16-bit bus and a medium-drive-strength bus are formed from A[7:0], D[7:0] and LA[7:0], LD[7:0] respectively. D[7:0] and LD[7:0] are the least significant sections of each of these buses.

In this mode one additional latch (L12), is located between the local address bus and local master block transfer counter, C1. This latch allows the local master block transfer counter to be loaded from the local address bus prior to loading the VMEbus master block transfer counter, C3. This is necessary since both the VMEbus master block transfer counter, C3, and the local master block transfer counter, C1, are loaded from the same local address bus. When counter C3 loads during the block transfer initiation cycle, the contents of latch L12 are moved to counter, C1.

All other functions available in this mode operate in a similar manner as in the byte width mode. For further information on this mode and detailed timing information refer to the *VIC64/CY7C964 Design Notes*.

Interfacing to the VIC64 and VIC068A

Previously, interfacing the VIC068A to the VMEbus required a significant amount of LSI and MSI devices. With the advent of 64-bit VMEbus block transfers and the VIC64, the external discrete device count for a full functional interface implementation expanded. The CY7C964 has been developed to combat this problem by incorporating the functions of much of this external logic into a single package. Use of the CY7C964 shortens system design, debug, and manufacturing cycle times. This removes the burden of performing worst-case and critical timing analysis on the VMEbus and VIC buffer control sections of a system design. Local control signals other than those directly connected to the VIC64 or VIC068A have been kept to a minimum. *Figure 4* shows a full function D64 VMEbus interface implemented using the CY7C964, VIC64 and all VMEbus interface local support logic. This example interface features:

- Dual Path Address Operation
- Slave BLT Cycles During Master BLT Interleave
- Software Programmable Slave VMEbus Address
- Write Posting
- VIC Mail Box Interrupt Messaging Support

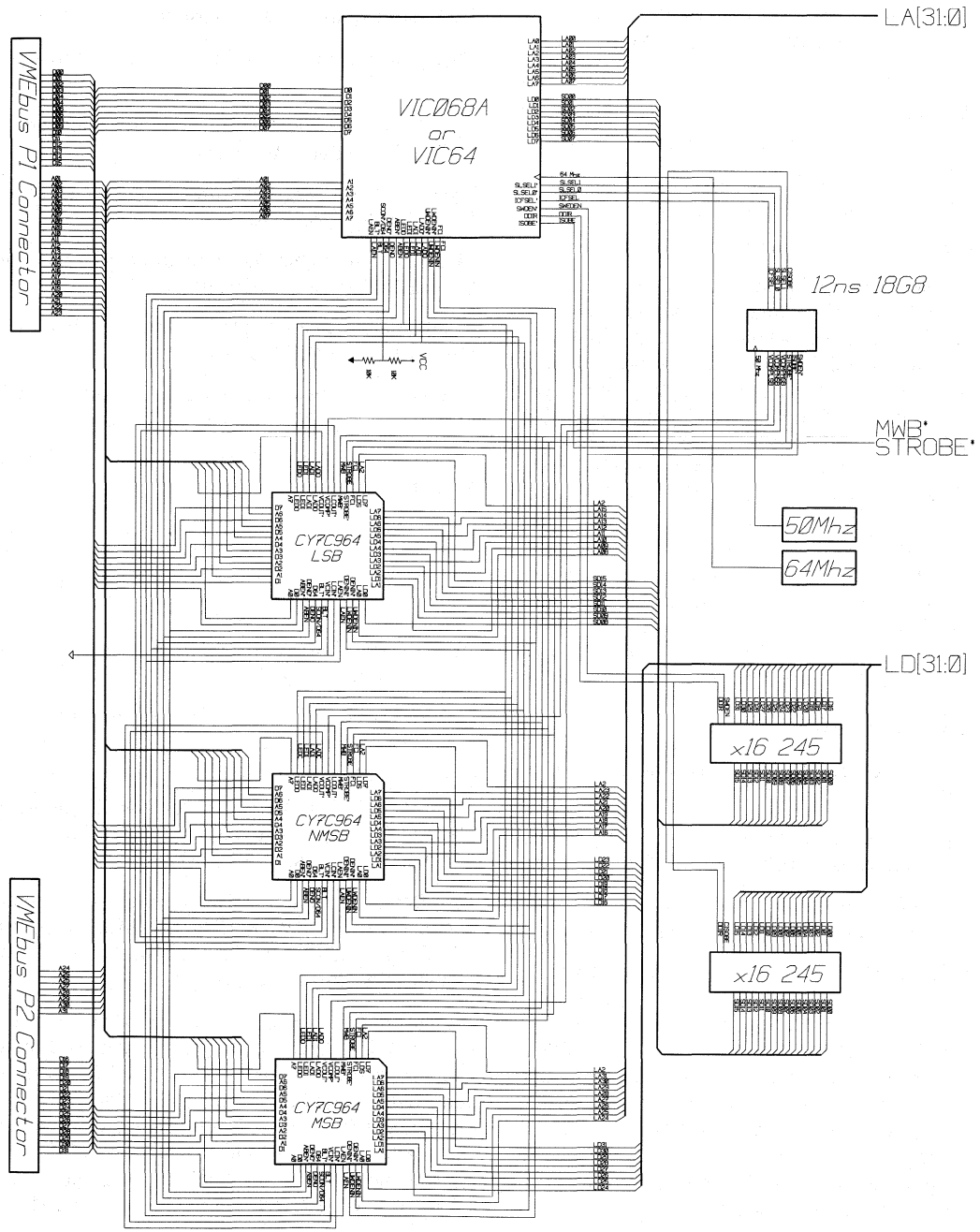


Figure 4. CY7C964/VIC VMEbus Interface

The interface can be dissected into 5 functional sections for the purpose of discussion. These sections are:

- VMEbus Signal Group
- VIC Buffer Control Signal Group
- CY7C964 Local Signal Group
- CY7C964 Address Comparison Group
- Local Data Swap Buffer Logic

The focus of this application note is the CY7C964. Each of the interface functional sections are examined from this perspective. The CY7C964s are referred to as the LSB (Least Significant Byte), NMSB (Next Most Significant Byte), and MSB (Most Significant Byte) device depending on the segment of the VMEbus that they control. The LSB controls VMEbus address and data signals [15:8], NMSB [23:16], and MSB [31:24]. This interface uses the CY7C964s in the byte width mode I as address and data controllers. All of the information contained within this section pertains to this mode of operation. For additional information on the signals described within this section consult The *VMEbus Specification (IEEE 1014)* and/or the *Cypress Semiconductor VIC068A/VAC068A User's Guide*

VMEbus Signal Group

This group includes the VMEbus address and data signals.

D[7:0]. VMEbus compatible data signals that directly connect to VMEbus P1 and P2 connectors.

A[7:0]. VMEbus compatible address signals that directly connect to VMEbus P1 and P2 connectors.

Each CY7C964 provides support for 8 bits of VMEbus address and data. Three CY7C964s are necessary for 32-bit (D32/D64) interface applications. The A[7:0] and D[7:0] transceivers on the CY7C964 furnish a high drive strength allowing direct connection to the respective address and data signals on the VMEbus backplane. With the VIC068A or VIC64 controlling the CY7C964s, all VMEbus worst case timing and drive strength requirements are met for all types of data transfer operations.

VIC Buffer Control Signal Group

This group includes all of the VIC buffer control signals.

LADO. Latch Address Out, directly connects to VIC LADO on all CY7C964s.

LADI. Latch Address In, directly connects to VIC LADI on all CY7C964s.

LEDO. Latch Enable Data Out, directly connects to VIC LEDO on all CY7C964s.

LEDI. Latch Enable Data In, directly connects to VIC LEDI on all CY7C964s.

ABEN*. VMEbus Address Bus Enable, directly connects to VIC ABEN on all CY7C964s.

DENO. Data Enable Output, directly connects to VIC DENO on all CY7C964s.

D64. D64 Block Transfer Mode Enable, directly connects to VIC64 SCON/D64 pin on all CY7C964s. This input should be tied Low on all CY7C964s when using VIC068A.

BLT*. Block Transfer Enable, directly connects to VIC BLT on all CY7C964s.

LAEN. Local Address Enable, directly connects to VIC LAEN on all CY7C964s.

DENIN*. Primary Data Enable In, directly connects to VIC UWDENIN* on NMSB and MSB CY7C964s, and directly connects to VIC LWDENIN* on LSB CY7C964.

DENIN1*. Companion Data Enable In, directly connects to VIC LWDENIN* on NMSB and MSB CY7C964s, and directly connects to VIC UWDENIN* on LSB CY7C964.

A major design-time savings is realized when using the CY7C964s because all of these signals directly connect to the VIC or are hardwired to a steady state value. The buffer control interface is simple and straight forward, with the minor exception that the connection of UWDENIN* and LWDENIN* control signals from the VIC are swapped to the DENIN* and DENIN1* on the LSB CY7C964.

CY7C964 Local Signal Group

The CY7C964 local signal group consists of the VMEbus and local block transfer counter count-enable daisy-chains.

LCIN*. Local address counter Count enable IN. On the LSB CY7C964 tie this input Low. On the NMSB device directly connect this signal to the LCOUT* of LSB device. For the MSB CY7C964 connect this input to the LCOUT* of the NMSB device.

LCOUT*. Local address counter Count enable OUT. On the LSB CY7C964, connect this output to the NMSB LCIN* input. On the NMSB CY7C964, connect this output to the MSB LCIN* input. For the MSB device do not connect this output.

VCIN*. VMEbus address counter Count enable IN. On the LSB CY7C964 tie this input Low. On the NMSB device directly connect this input to the VCOUT* of LSB device. For the MSB CY7C964 connect this input to the VCOUT* of the NMSB device.

VCOUT*. VMEbus address counter Count enable OUT. On the LSB CY7C964, connect this output to the NMSB VCIN* input. On the NMSB CY7C964, connect this output to the MSB VCIN* input. For the MSB device, do not connect this output.

These signals enable the local and VMEbus higher order address counters, two local address counters, a master block transfer, a slave block transfer, and a single VMEbus address counter. The local address counters share the LCIN*/LCOUT* count enable daisy-chain. These signals are multiplexed within the CY7C964 and enable the proper counter depending on the current state of the interface. The VCIN*/VCOUT* daisy-chain is dedicated to the VMEbus address counter on the device. When the VCIN* or LCIN* inputs are held Low, counting is enabled for the appropriate counters within the device. The VCIN* and LCIN* signals do not advance the counters; these signals just enable counting. The counters increment when these signals are active and the proper increment count control logic sequence occurs. The VIC advances the address counters at the proper time during VMEbus and local DMA block transfer operations.

CY7C964 Address Comparison Signal Group

The CY7C964 address comparison signal groups consists of the local signals that are associated with the internal VMEbus address comparators. FC1, MWB*, and LDS also are used to control other functions on the CY7C964. Refer to the *VIC64/CY7C964 Design Notes* for more information on these signals.

FC1. Function Code 1 signal, directly connects on all CY7C964s to the local signal that drives the VIC FC1.

MWB*. Module Wants VMEbus, directly connects on all CY7C964s to the local signal that drives the VIC MWB*.

LDS. Load Register Select, directly connects to LA2 for systems with 32-bit local bus. Refer to the following text for additional information.

STROBE*. Latch Register Control. A chip select like signal that selects the CY7C964 internal comparator mask and comparison registers.

VCOMP*. VMEbus Address Comparator Output. The comparator output signal from the CY7C964 address comparator. This signal asserts Low if the a pattern on address signals A[7:0] matches the programmed values of the comparison logic.

The implementation of this group of CY7C964 signals is application specific. The MWB* and FC1 signals are included in this section because they are locally generated signals required by the VIC. These two signals differ slightly from their companions on the VIC. On the CY7C964 the MWB* and FC1 signals are inputs. On the VIC, MWB* is also an input, but FC1 is a bidirectional signal that can be driven by the VIC. These signals on the CY7C964 can be directly connected to their respective local signals on VIC.

The CY7C964s contain a high-performance programmable VMEbus address equality comparator. This comparator is controlled by two internal write-only registers: a mask and a compare register. The mask register enables and disables bits of the

comparator and the compare register stores the data pattern which inputs are compared against. VCOMP* is the active Low comparator match output signal. VCOMP* is driven Low by the CY7C964 if the bit pattern on A[7:0] matches the associated enabled bits of the compare register. Loading the mask register bits with 0's enables the corresponding bits of the compare register. Loading bits of the mask register with 1s places bits of the compare register in a don't care or match-on-anything state. If all bits of the mask register are loaded with 1s, the compare register matches everything, causing VCOMP* to always be driven Low. These registers are loaded by supplying the proper data on LD(7:0) and address on MWB* and LDS signals. The STROBE* input is used to qualify the address and latch the data into the proper internal register. *Figures 5*and *6* and *Table 1* show the waveforms and timing conditions needed to load the compare and mask registers. The mask register is cleared (all bits enabled), when the compare register is loaded.

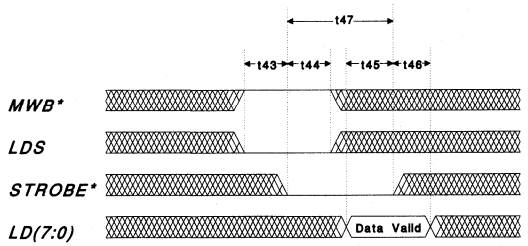


Figure 5. Mask Register Load Cycle

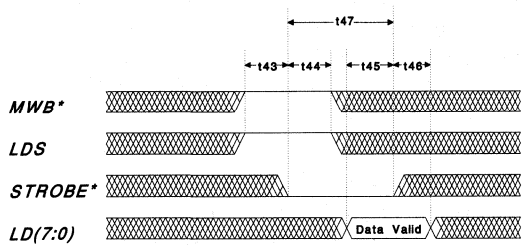


Figure 6. Compare Register Load Cycle

Table 1. Compare Register Load Cycle Times

Para	Description	Min.	Max.	Unit
t43	MWB*, LDS set-up time to STROBE* falling edge	0		ns
t44	MWB*, LDS hold time after STROBE* falling edge	5		ns
t45	LD(7:0) set-up time to STROBE* rising edge	5		ns
t46	LD(7:0) hold time after STROBE* rising edge	5		ns
t47	STROBE* minimum pulse width	10		ns

Therefore it is important to load the compare register first, then load the mask register as desired.

This load cycle operates as follows. The state of LDS and MWB* are latched on the falling edge of STROBE*. The data is loaded into the selected register on the rising edge of STROBE*. MWB* must be held inactive (High) during comparator register loading. The state of LDS selects the register to load. If LDS is High during the cycle the compare register is loaded; if LDS is Low the data is written to the mask register. This load cycle can be generated by decoding a separate address region or chip select signal for the CY7C964 comparator registers.

For applications with a 32-bit local data bus it is desirable to load all three CY7C964s in parallel by having the host processor perform a 32-bit write cycle to the address region that activates STROBE*. The select signal for the address region is connected to the STROBE* input on all three CY7C964s. The 8 bits of data on the lowest order section of the local data bus LD[7:0] do not matter to the VIC, as long as the VIC CS* signal remains inactive during this write cycle. Boards that use this style of interface should connect LDS to LA2, thereby decoding the mask register at the Base Address of the address region and the compare register at the Base Address + 4. LDS also controls the operation of the D64 block transfer data multiplexer/demultiplexer. Sys-

tems with 32-bit local data buses should connect local address signal LA2 to LDS for proper operation of the D64 data multiplexer/demultiplexer logic.

The mask and compare registers can be set to select any contiguous address region on the VMEbus. These registers do not preload and can power up in any state. It is advisable to initialize these registers as soon as possible in the system boot sequence. The CY7C964 comparator output signal VCOMP*, supplies the result from the equality compare logic. VCOMP* drives Low when the input matches the loaded comparator conditions.

The CY7C964 VCOMP* signal is not directly compatible with the VIC SLSEL0* and SLSEL1* slave select signals. The short (10 ns) address setup time to AS* active for VMEbus slave boards does not meet the worst case compare out delay of the CY7C964 VCOMP* signal. Combining this with the potential output glitching that can occur with an asynchronous comparator can cause problems for the VIC. It is recommended that the VCOMP* signal be externally filtered prior to being used with the VIC SLSEL0* or SLSEL1* signals. Most applications will require some external comparison logic to combine the VCOMP* signals from the NMSB and MSB devices, furnishing finer grained VMEbus decoding.

The interface example in *Figure 4* uses a 12-ns 18G8 to perform these functions and to disable the VMEbus slave select signals to the VIC until the CY7C964 comparator control registers have been initialized. Using this PLD allows the interface to decode three different VMEbus addresses regions:

- VMEbus A32 for local access – VIC SLSEL0*
- VMEbus A24 for local access – VIC SLSEL1*
- VMEbus A16 for mailbox interrupt – VIC ICFSEL*

Figure 7 shows the PLD ToolKit design file for this device. The two VIC slave select signals (SLSEL0* and SLSEL1*) can be used to conveniently decode two VMEbus address regions. SLSEL0* selects if the NMSB CY7C964 becomes TRUE. SLSEL1* requires both NMSB and MSB comparators to evaluate TRUE.

A 50-MHz clock and the D registers within the 18G8 are used to build a simple digital filter that removes any glitches that may occur on the three CY7C964 VCOMP* signals.

As mentioned previously, the comparators within the CY7C964s are always active and they power up in an unknown state. The PLD includes an ENABLE signal which disables the SLSEL0*, SLSEL1*, and ICFSEL* signals to the VIC until the first access is made to one of the comparator control registers. Adding the ENABLE function to this PLD guarantees that the VIC slave select signals cannot become active until the one of the comparator control registers has been initialized.

There is a potential problem that can occur when loading the CY7C964 comparator control registers. The local data bus isolation buffer, which is necessary to allow data swapping, is normally disabled by the VIC. This causes a problem during CY7C964 register initialization cycles because the VIC only enables the local data bus isolation buffers during VIC or VMEbus accesses. The PLD solves this problem by providing conditioning logic for the ISOBE* signal. In the PLD design file in *Figure 7,* a signal named CISOBE* is generated. CISOBE* asserts (Low) enabling the isolation buffer if the VIC ISOBE* is asserted or the CY7C964 STROBE* input is asserted. SWDEN* was added to the equation for completeness, however, it may not be necessary in many designs. There are obviously many other implementations for control of the VIC isolation buffers. One implementation is shown here, but the best method for control of this signal is application specific and left to the designer.

Local Data Swap Buffer Logic

Local Data Swap Buffer logic is a requirement for all 32-bit local bus designs that want to be able to perform 8- or 16-bit transfers. The swap buffer moves data to and from the lower section of the VMEbus D[15:0] to the upper segments of the local bus D[31:16]. VMEbus requires that all 8- and 16-bit data transfers be performed on the D[15:0] section of the bus. The CY7C964s work properly with the VIC controlled swap buffer.

```
C18G8;

CONFIGURE;

CLK_50Mhz(node=1),
LSBCOMP(node=2),
NSBCOMP(node=3),
MSBCOMP(node=4),
STROBE(node=5),
BD_RESET(node=6),
ISOBE(node=7),
SWDEN(node=8),

ENABLE(node=12, noreg, iop),
SEL(node=13),
DSEL(node=14),
CISOBE(node=15, noreg, iop),
SLSEL1(node=17, noreg, iop),
SLSEL0(node=18, noreg, iop),
ICFSEL(node=19, noreg, iop),

EQUATIONS;

/CISOBE     =    <OE>
                 <SUM> /ISOBE
                 +     /STROBE *  SWDEN;

/ENABLE     =    <OE>
                 <SUM>  BD_RESET * /STROBE
                 +      BD_RESET * /ENABLE;

/SEL        =    <OE>
                 <SUM>  /LSBCOMP * /ENABLE *  BD_RESET
                 +      /NSBCOMP * /ENABLE *  BD_RESET
                 +      /MSBCOMP * /ENABLE *  BD_RESET;

/DSEL       =    <OE>
                 <SUM>  /SEL * /LSBCOMP * /ENABLE *  BD_RESET
                 +      /SEL * /NSBCOMP * /ENABLE *  BD_RESET
                 +      /SEL * /MSBCOMP * /ENABLE *  BD_RESET;

/ICFSEL     =    <OE>
                 <SUM>  /DSEL * /LSBCOMP * /ENABLE * BD_RESET;

/SLSEL0     =    <OE>
                 <SUM>  /DSEL * /NSBCOMP * /ENABLE * BD_RESET;

/SLSEL1     =    <OE>
                 <SUM>  /DSEL * /MSBCOMP * /NSBCOMP * /ENABLE *  BD_RESET;
```

Figure 7. PLD ToolKit™ Design File For 18G8 PLD

Summary

The CY7C964 is a high-performance byte or word width slice of VIC compatible VMEbus logic. Using the CY7C964 in conjunction with the VIC068A or VIC64 shortens design cycle time, reduces component count, reduces interface real estate requirements, and overall design risk. For further information on the local VIC interface and more detailed timing information in the CY7C964 refer to the *VIC068A/VAC068A User's Guide* and the *VIC64/CY7C964 Design Notes.*

Document #: BUS0008–0293

PLD ToolKit is a trademark of Cypress Semiconductor, Inc.

Modules – 8



Module Section Contents and Abstracts

DRAM Accelerator: Set-Up for Basic Operation **8–1**

The CYM7232/CYM7264 is a high-performance DRAM controller designed for a wide variety of uniprocessor and multiprocessor environments. It consists of a data path for error detection and correction, FIFO buffering, and funnel multiplexing of data from the bus width to a wide DRAM access. The controller has many operating modes, so set-up may seem complicated at first.

This application note provides a bootstrap description, giving register load values and configuration connections to assist the user in getting the controller to function with a minimum of effort. Once this is accomplished, more advanced use of the controller can be explored.

DRAM Configuration and Diagnostics **8–7**

This application note describes a method to automatically determine the DRAM configurations controlled by the DRAM Accelerator. It also includes a procedure to test the ECC circuit by injecting an error into the check bits.

DRAM Interfacing **8–13**

This application note illustrates how to interface the CYM7232/CYM7264 DRAM Accelerator Module to industry standard DRAM SIMMs. Capacitive loading tables are presented for various size main memories. Timing diagrams illustrate the critical timing for 32-bit and 64-bit-wide interfaces to main memory. A thorough timing analysis is presented.

CYM7232/7264 DRAM Accelerator: Mixing 5-Volt and 3.3-Volt DRAMs **8–32**

Many high-performance systems are powered by 3.3V processors. However, many system components are not available in 3.3V. Therefore, mixing 5V and 3.3V components in the same system is common practice. This application note describes how 5V and 3.3V DRAM SIMMs can be used simultaneously with a DRAM Accelerator.

DRAM Accelerator: Set-Up for Basic Operation

Introduction

The CYM7232/CYM7264 is a high-performance DRAM controller designed for a wide variety of uniprocessor and multiprocessor environments. It consists of a data path for error detection and correction, FIFO buffering, and funnel multiplexing of data from the bus width to a wide DRAM access. The controller has many operating modes, so set-up may seem complicated at first.

This application note provides a bootstrap description, giving register load values and configuration connections to assist the user in getting the controller to function with a minimum of effort. Once this is accomplished, more advanced use of the controller can be explored.

Getting Started

This section is intended to assist the user in getting started with the CYM7232/7264 quickly. Much of the programming flexibility is bypassed, and little explanation is provided for programmed register values or their corresponding impact on system performance. More detailed information is given in the Cypress Semiconductor *High-Performance Data Book*. The basic set-up starts with a sequence of steps, register value loads, and an explanation of expected results. The user should be able to get the device functioning quickly if the instructions are followed. After basic functionality is achieved, programmed register values can be modified to enhance system performance. This description is applicable to both hardware and simulation environments. The most general set-up possible is described. Certain parameters are chosen as typical values that one might expect to find in an average system. However, the simplest operating modes are selected, minimizing start-up problems. Although many different buses are supported, this example is specific to an Intel™ 486 interface.

The values shown in *Table 1* are used for the internal register programming. Parameter names refer directly to the register field names as given in the CYM7232/7264 datasheet in the Cypress Semiconductor *High-Performance Data Book* (1993).

Table 1. Register Values for Basic Operation

Register Field Name	Suggested Value	Description
IMD (Pin M3)[1]	1	Generic bus interface (non-MBus) (IMD is programmed by an external pin)
SNWSOURCE[1, 2]	0	External snoop window source
DSBACKMODE[1, 2]	01	Early DS (1 clk), real time bus acknowledge
Command Register 0–Index 00H		
CIE (Bit 7)	0	Disable coherent invalidate (for MBus only)
RFD (Bits 6–0)	7F(h)	Refresh divisor – 7FH x 15 clock cycles between refresh requests

Table 1. Register Values for Basic Operation (continued)

Register Field Name	Suggested Value	Description
Command Register 1–Index 01H		
INIT (Bit 7)	0	Do not initialize DRAM and check bits
WC (Bit 6)	0	Use incoming data to calculate ECC check bits
AEM (Bit 5)	0	BERR asserted for one clock after an address parity error
SBS (Bit 4)	0/1	0–32 bit System Bus/1–64 bit System Bus
ES (Bit 3)	0/1	0–32 bit EDC for CYM7232/1–64 bit EDC for CYM7264
CLM (Bits 2–1)	00	x1 clock multiplier
RFT (Bit 0)	0	Refresh test mode disabled (required for proper operation)
Command Register 2–Index 02H		
BLP (Bits 7–4)	0001(b)	Block 0 populated (this is system dependent)
BLK (Bits 3–2)	00(b)	1 block populated (this is system dependent)
DFB (Bits 1–0)	01(b)	Default burst length is 32 bytes
Command Register 3–Index 03H		
RCM (Bits 7–6)	00(b)	Refresh OFF; DRAM initialization OFF
BAM (Bits 5–4)	00(b)	No Read wait states; Error Correction OFF
SEN (Bit 3)	0	Scrub disabled
CAM (Bit 2)	0	CAS[3:0] independently asserted
RSM (Bit 1)	0	RAS[3:0] staggered by one bus clock
BRM (Bit 0)	1	Bus Arbiter OFF; $\overline{BR}/\overline{FE}$ reflects FIFO status
Command Register 4–Index 04H		
PLT (Bits 7–6)	00(b)	Internal VCOs disabled
VCO (Bits 5–4)	00(b)	No VCO selected, use external MCLK
IOSWC (Bits 3–0)	0(h)	External SNW source is selected; snoop window count is not used
Command Register 5–Index 05H		
IE (Bit 7)	0	Disable interrupts to the system bus
EDP (Bit 5)	1	Enable data bus parity error interrupt, disabled by IE (Bit 7) zero
EAP (Bit 4)	1	Enable address bus parity error interrupt, disabled by IE
EME (Bit 3)	1	Enable read-modify-write multi. bit error interrupt, disabled by IE
EUE (Bit 2)	1	Enable multiple bit uncorrectable error interrupt, disabled by IE
EDE (Bit 1)	1	Enable double bit error interrupt, disabled by IE
ESE (Bit 0)	1	Enable single bit error interrupt, disabled by IE
Command Register 6–Index 06H		
SNWWRCNT (7–4)	0(h)	External SNW source is selected; this field is irrelevant
SNWRDCNT (3–0)	0(h)	External SNW source is selected; this field is irrelevant
DRAM Timing Register 0–Index 08H		
TRAM (Bits 7–4)	1	DRAM timing, RAS to address mux

Table 1. Register Values for Basic Operation (continued)

Register Field Name	Suggested Value	Description
TAR (Bits 3–0)	1	DRAM timing, address to RAS
DRAM Timing Register 1–Index 09H		
TRAS (Bits 7–4)	6	DRAM timing, RAS pulse width
TMAC (Bits 3–0)	1	DRAM timing, address mux to CAS
DRAM Timing Register 2–Index 0AH		
TCP (Bits 7–4)	1	DRAM timing, CAS precharge width
TRPR (Bits 3–0)	2	DRAM timing, RAS precharge width
DRAM Timing Register 3–Index 0BH		
TRIN (Bits 7–4)	2	DRAM timing, RAS completion during non-reflective inhibits
TDC (Bits 3–0)	1	DRAM timing, FIFO data delay to CAS assertion
DRAM Timing Register 4–Index 0CH		
TENW (Bits 7–4)	1	DRAM timing, DRAM enable delay on write
TENR (Bits 3–0)	1	DRAM timing, DRAM enable delay on read
Base Address Register–Byte Index 10H, BA[27:20] (Bits 7–0); Byte Index 11H, BA[35:28] (Bits 15–8)		
BA	0000(h)	Set the base address of memory at 0
Logical Block Displacement Register—Byte Index 12H, LBD[7:0]; Byte Index 13H, LBD[15:8]		
DS3 (Bits 10–8)	1(h)	Set-Up for 1-Meg-deep DRAMs (system dependent)
DS2 (Bits 5–3)	1(h)	Set-Up for 1-Meg-deep DRAMs (system dependent)
DS1 (Bits 2–0)	1(h)	Set-Up for 1-Meg-deep DRAMs (system dependent)
Logical Block Population Code Register–Byte Index 18H, LPCR[7:0]; Byte Index 19H, LPCR[15:8]		
LPC3 (Bits 7–6)	1(h)	Set-Up for 1-Meg-deep DRAMs (system dependent)
LPC2 (Bits 5–4)	1(h)	Set-Up for 1-Meg-deep DRAMs (system dependent)
LPC1 (Bits 3–2)	1(h)	Set-Up for 1-Meg-deep DRAMs (system dependent)
LPC0 (Bits 1–0)	1(h)	Set-Up for 1-Meg-deep DRAMs (system dependent)
LBP3 (Bit 11)	0	Block 3 unpopulated (system dependent)
LBP2 (Bit 10)	0	Block 2 unpopulated (system dependent)
LBP1 (Bit 9)	0	Block 1 unpopulated (system dependent)
LBP0 (Bit 8)	1	Block 0 populated (system dependent)
Physical/Logical Block Map Register–Index 1AH		
PBL0 (Bits 1–0)	0(h)	Map physical block 0 to logical block 0
PBL1 (Bits 3–2)	1(h)	Map physical block 1 to logical block 1
PBL2 (Bits 5–4)	2(h)	Map physical block 2 to logical block 2
PBL3 (Bits 7–6)	3(h)	Map physical block 3 to logical block 3
Logical Block Mux Position Register–Byte Index 1CH, MPR[7:0]; Byte Index 1DH, MPR[15:8]		
LBMP0 (Bits 2–0)	2(h)	DRAM A10/A9 row/column address mux split
LBMP1 (Bits 5–3)	2(h)	DRAM A10/A9 row/column address mux split

Table 1. Register Values for Basic Operation (continued)

Register Field Name	Suggested Value	Description
LBMP2 (Bits 10–8)	2(h)	DRAM A10/A9 row/column address mux split
LBMP3 (Bits 13–11)	2(h)	DRAM A10/A9 row/column address mux split
BERR Control Register–Index 31H		
BEM (Bit 1)	0	Data parity error BERR asserted for one clock
BEC (Bit 0)	0	Used to clear BERR if BEM=1; unused if BEM=0

Using these values for their respective register fields, the following register load values are obtained (assumes CYM7232 with 64-bit bus size).

CR0	{CIE, RFD}	7F(h)
CR1	{INIT, WC, AEM, SBS, ES, RFT}	10(h)
CR2	{BLP, BLK, DFB}	11(h)
CR3	{RCM, BAM, SEN, CAM, RSM, BRM}	01(h)
CR4	{PLT, VCO, IOSWC}	00(h)
CR5	{IE, 0, EDP, EAP, EME, EUE, EDE, ESE}	3F(h)
CR6	{SNWWRCNT, SNWRDCNT}	33(h)
DT0	{TRAM, TAR}	11(h)
DT1	{TRAS, TMAC}	61(h)
DT2	{TCP, TRPR}	12(h)
DT3	{TRIN, TDC}	21(h)
DT4	{TENW, TENR}	11(h)
BAR0		00(h)
BAR1		00(h)
LBD0	{00, DS2, DS1}	09(h)
LBD1	{0000000, DS3}	01(h)
LPCR0	{LPC3, LPC2, LPC1, LPC0}	55(h)
LPCR1	{0000, LBP3, LBP2, LBP1, LBP0}	01(h)
PBL	{PBL3, PBL2, PBL1, PBL0}	E4(h)
LBMPR0	{LBMP1, LBMP0}	12(h)
LBMPR1	{LBMP3, LBMP2}	12(h)
BERRC	{000000, BEM, BEC}	00(h)

Device Connection

With these selections, the device should be connected as follows. MCLK and CLK must be tied together. For convenience a x1 clock multiplier and the external MCLK source have been programmed. $\overline{SNW}$, $\overline{INH}$, and $\overline{TRC}$ input pins should be tied HIGH. This will cause the snoop window to close immediately, preventing inhibited or transformed transactions. DS should be tied LOW to simplify the transaction handshake.

The following signals should be pulled up with a resistor:

$\overline{BERR}$, $\overline{INT}$, $\overline{BACK}$[1:0], UERR, and $\overline{BB}$.

Tie TSTE to V_{DD}.

Power-On Reset

With power applied and clocks applied to CLK and MCLK pins, assert $\overline{RSTIN}$ (0), set $\overline{UERR}$ to logic 0 (SNWSOURCE), $\overline{BACK}$[1:0] to logic 01(b) (DSBACKMODE) for at least 16 clock cycles. (Simulation environments could also drive D[63:0] to 00000000000000FF (h) and DP[7:0] to 00 (h) at this time. This will initialize the data bus holding buffers and allow the first read of the index register to be easily recognized.) After the RSTIN interval, three-state the drivers for $\overline{UERR}$ and $\overline{BACK}$[1:0]. $\overline{RSTIN}$, $\overline{UERR}$, and $\overline{BACK}$[1:0] may be changed in the same clock cycle; do not release $\overline{UERR}$ or $\overline{BACK}$[1:0] prior to deasserting $\overline{RSTIN}$.

Register Programming

Sixteen clock cycles after $\overline{RSTIN}$ is deasserted, I/O transactions may begin to set the internal registers. The index register points to Command Register 0 after system reset (Index Register Address 00). System verification should begin with a read of Command Register 0 (initialized contents of 7F(H)). Subsequently writing and reading the index register immediately followed by the associated internal control register will verify I/O read-write transactions.

The primary indication of a successful transaction is the assertion of $\overline{BACK}$[1]. For I/O operations,

$\overline{\text{BACK}}$[1] will occur within ten clock cycles after the address strobe is asserted. The address has no effect on I/O operations in the generic mode. These transactions are controlled through the ID inputs. A $\overline{\text{BACK}}$[1] response with no read data suggests a possible mix-up in the endian definition in the bus master. Look for valid data on D[7:0]. Data for the I/O registers will always appear on D[7:0] regardless of the endian selection. (Some registers will supply data in other portions of the data bus as well.) It is best to make the first attempts with address and data parity OFF to eliminate parity errors as a factor in the debug (PMD[2:0] = 000).

Preinit and INIT

After initializing all internal registers, rewrite Command Register 1 with bit 7 set to 1. This action sets the INIT bit starting the Preinit cycle (16 RAS only cycles to properly power-up the DRAM components). An Init of the entire DRAM array may optionally follow the Preinit phase. The DRAM data bits can be cleared to 0 with the corresponding check bits preset to the associated value. During preliminary debug Init should be avoided to save time (Command Register 3, Bit 4: BAM0 = 0). Firmware should periodically examine the IC bit in the Interrupt Status register to determine when initialization is completed. Debugging may be simplified by waiting about 250 clock cycles after the INIT bit is set, guaranteeing completion of the initialization. DRAM operations may now commence.

When using the internal clock multiplier PLLs, Command Register 4 should be loaded last in the sequence. The start-up example does not use the internal PLLs, so the register order is irrelevant.

Preparing for DRAM Access

The DRAM array will power up randomly. After the Preinit phase described above, the data and check bits will not correspond (as a result of skipping the Init portion). During hardware debug, the error correction and scrubbing should be turned OFF (Command Register 3, Bit 4: BAM0=0 and Command Register 3, bit 3: SEN=0)). Interrupts should be disabled (Command Register 5, bit 7: IE=0). This will prevent error indications in read operations even though the DRAM check bits are initially incorrect. Writes will cause the correct error check bits to be written into DRAM.

Write and Read Operations

Start the debug of DRAM with a write of 8 bytes (see recommended patterns below) or 4 bytes for a 32-bit system bus. Let SIZE[3:0] = 3, TYPE[5:0] = 00, A[35:0] = 0. The module will respond by asserting $\overline{\text{BACK}}$[1] for one clock cycle to acknowledge the write transaction into its FIFOs. These are non-burst, non-read-modify-write operations and are least likely to have complications. Monitor the RAS[0], CAS[0], and R/W[1:0] (only R/W[0] for CYM7264) signals. Check that the timing of these signals is reasonable and check the data lines to verify the little/big endian convention. Monitor the check bits to see if they match the table and perform an 8-byte read to the same address location. Again, verify that the RAS[0], CAS[0] and the R/W signals make sense. Check that the DRAMs' output data is valid during CAS assertion. The data read by the system bus master must be correct during the $\overline{\text{BACK}}$[1] assertion cycle. At this point, the basic communication with the CYM7232/ CYM7264 is established. The user may now fine-tune the register values to give optimum system performance. The internal PLLs (or an externally multiplied MCLK) may also be used to adjust the DRAM timing parameters for higher resolution.

Some Recommended Patterns

CYM7232		CYM7264	
DATA	CHECK BITS	DATA	CHECK BITS
MSB <– LSB	MSB <– LSB	MSB <– LSB	MSB <– LSB
01234567	66	8F7F6F5F4F3F2F1F	03
FEDCBA98	62	708090A0B0C00D0E0	07
55555555	78	11FF22EE33DD44CC	E4
AAAAAAAA	7C	0123456789ABCDEF	F3

Notes

1. External Programming—IMD, SNWSOURCE, and DSBACKMODE are not directly accessible in registers.
2. SNWSOURCE is programmed by driving UERR during reset and DSBACKMODE is programmed by driving BACK[1:0] during reset.

Document #: MOD0004–1293

Intel is a trademark of Intel Corporation.

DRAM Configuration and Diagnostics

Introduction

The CYM7232/CYM7264 is a high-performance DRAM controller designed for a wide variety of single and multiprocessor environments. It consists of a data path for error detection and correction, FIFO buffering, and funnel multiplexing of data from the bus width to a wide DRAM access. The controller is very flexible. Set-up for various operating modes can seem complicated at first.

This application note provides:

1. A method for determining the configuration of DRAM SIMMs after power up without user intervention
2. A recommended diagnostic sequence for checking the DRAM
3. The check bit polynomial generation algorithm

An Algorithm for Determining the DRAM Configuration

Field upgrades of the DRAM can be facilitated by an algorithm that automatically determines the memory configuration in each block after power-up. The user simply inserts the DRAM modules into the sockets and turns on the system. The system BIOS will execute the following algorithm to interrogate the type and size of the DRAM modules and then configure the DRAM Accelerator registers accordingly.

The algorithm is based on a process of elimination. The technique requires choosing a combination of addresses which provide a unique access for a particular DRAM type and an aliased address for all other DRAM types. The row/column address split points for various symmetrical DRAM configurations is given in *Table 1*.

Table 1. Row/Column Address Splits

DRAM	Column	Row
256k	0:8	9:17
1M	0:9	10:19
4M	0:10	11:21
16M	0:11	12:23

Certain DRAMs have an asymmetrical configuration (e.g., 12 row addresses, 10 column addresses). The described algorithm assumes a symmetrical address arrangement, but can be easily extended to cover most of the asymmetrical configurations.

When the user has finished inserting new DRAM modules and turned on the power, the system BIOS will program all the DRAM Accelerator's internal registers to reflect the following configurations. Refresh is enabled, scrubbing is disabled, and DRAM initialization (clearing of all bits) is disabled. All relevant registers are programmed assuming 16M deep DRAMs (split, displacement, etc.). All four logical blocks are populated and the logical-to-physical mapping is programmed such that the logical block = the physical block. After the complete register set is programmed, including DRAM timing, the INIT bit in Command Register 1 is set to start the preinit cycle and enable access to the DRAM (note that because DRAM initialization is turned off by setting RCM1 in Command Register 3 to zero, the setting of the INIT bit only causes preinit to occur). The algorithm assumes the base address is 0. Otherwise, the base address must be added to the numbers given below.

For a selected block, the procedure is as follows:

1. Write a selected pattern (Pattern 1) to DRAM address 0.

2. Write a new pattern (Pattern 2) distinct from the first to DRAM address 800000 (h). For non-16M DRAMs, this overwrites address 0.

3. Write a new pattern (Pattern 3) distinct from the 1 and 2 to DRAM address 80 (h). This operation precharges the DRAM data bus to a unique state. The block must be populated with DRAMs to override the precharge during the next read.

4. Read DRAM address 0. When Pattern 1 is returned the block is populated with 16M DRAMs. When Pattern 2 is returned the block is populated with shallower DRAMs. When Pattern 3 is returned, the block is not populated. Reading DRAM address 800000 (h) and receiving Pattern 2 in return confirms 16M population.

5. After the 16M deep DRAM is eliminated, the algorithm tests for 4M DRAM. Steps 1 through 4 are repeated except that the write in step 2 and the confirmation read in step 4 is made to DRAM address 200000 (h). Make sure that the DRAM row/column split remains set for 16M DRAMs.

6. Step 5 is repeated for 1M and 256k DRAMs with DRAM addresses of 80000 (h) and 20000 (h) respectively.

7. The process is then repeated for each of the four memory blocks.

The addresses used throughout the procedure are given in *Table 2*. The Bus Address is given assuming a base address of 0.

Table 2. Test Addresses for Sensing Block Content

DRAM Size	DRAM Address	Bus Address
Block 0		
16M	800000 (h)	8000000 (h)
4M	200000 (h)	2000000 (h)
1M	080000 (h)	0800000 (h)
256k	020000 (h)	0200000 (h)
Block 1		
16M	800000 (h)	18000000 (h)
4M	200000 (h)	12000000 (h)
1M	080000 (h)	10800000 (h)
256k	020000 (h)	10200000 (h)
Block 2		
16M	800000 (h)	28000000 (h)
4M	200000 (h)	22000000 (h)
1M	080000 (h)	20800000 (h)
256k	020000 (h)	20200000 (h)
Block 3		
16M	800000 (h)	38000000 (h)
4M	200000 (h)	3200000 (h)
1M	080000 (h)	30800000 (h)
256k	020000 (h)	30200000 (h)

DRAM Diagnostic Sequence

DRAM array diagnostic routines can use the Check Bit registers to replace the internally generated error polynomials. During these diagnostics, the refresh should be disabled and error correction should be enabled. The WC bit in Command Register 1 must be set so that the content of the Diagnostic Check Bit registers instead of the check word calculated from the incoming data is stored into the DRAMs during a write. The diagnostic sequence is as follows:

1. Perform a write to a particular DRAM location.

2. Read back the same location. The incoming Check Bits are logged in the CB fields of the two EDC registers.

3. To insert an error in the data portion, invert one data bit. Copy the check bits in the EDC registers into the Diagnostic Check Bit registers. Write the new pattern together with the content of the Diagnostic Check Bit registers into the DRAMs.

   To insert an error in the check bit fields, invert one check bit and write the modified check bits into the Diagnostic Check Bit registers. Write

the original data together with the contents of the Diagnostic Check Bit register into the DRAMs.

4. Read back the same DRAM location. The modified data or the modified Check Bits will cause a single bit error. $\overline{INT}$ will be asserted and the location of the error will be logged in the Error Location register.

5. The Syndrome bits in the EDC registers can be examined to determine whether the error bit position matches the expected location.

6. Repeat the process until diagnostics are complete. Remember to clear the WC bit in Command Register 1 before resuming normal operations.

The diagnostic instruction which causes the DRAM Accelerator to read out the syndrome bits must not reside in the DRAMs because reading anything from the DRAMs will overwrite the contents of the EDC register, thus destroying the previously stored error status. The diagnostic program must therefore remain in EPROM or an instruction cache.

Error Correction Polynomial Generation—CYM7232

Table 3 illustrates how the check bits are generated in the CYM7232. The check bits CB[6:0] are generated by seven 32-bit parity generators. The parity is computed on data shown in each column marked by an X. For example, to generate check bit 0 (CB0), data bits 0, 4, 6, 7, 8, 9, 11, 14, 17, 18, 19, 21, 26, 28, 29, and 31 are used. Because CB0 is odd parity, the specified data bits are Exclusive-ORed together. In other words, CB0 is set if the selected data bits contain an odd numbers of ones. Check Bits which are marked as even (e.g., CB3) gate their selected data bits together with Exclusive-NOR gates. For example, CB3 is set if its group of data bits contain all zeros or a even number of ones.

Table 3. 32-Bit Error Detection-Correction Calculation

Data	CB0 Odd	CB1 Odd	CB2 Odd	CB3 Even	CB4 Even	CB5 Even	CB6 Even
D0	X	X	X	X			X
D1		X		X			X
D2		X			X		X
D3			X		X		X
D4	X	X	X		X		X
D5				X	X		X
D6	X	X		X	X		X
D7	X		X	X	X		X
D8	X	X				X	
D9	X		X			X	
D10		X	X			X	
D11	X			X		X	
D12		X		X		X	
D13			X	X		X	
D14	X				X	X	
D15			X		X	X	
D16		X	X	X			
D17	X	X		X			
D18	X	X			X		
D19	X		X		X		
D20		X	X		X		
D21	X			X	X		
D22		X		X	X		
D23			X	X	X		
D24		X				X	X
D25			X			X	X
D26	X	X	X			X	X
D27				X		X	X
D28	X	X		X		X	X
D29	X		X	X		X	X
D30					X	X	X
D31	X		X		X	X	X

Error Correction Polynomial Generation—CYM7264

The check bits CB[7:0] are generated by eight 64-bit parity generators using the algorithm shown in *Table 4*.

Table 4. 64-Bit Error Detection-Correction Calculation

Data	CB0 Even	CB1 Even	CB2 Odd	CB3 Odd	CB4 Even	CB5 Even	CB6 Even	CB7 Even
D0		X	X	X			X	X
D1	X	X		X			X	X
D2	X	X			X		X	X
D3	X		X		X		X	X
D4		X	X		X		X	X
D5	X			X	X		X	X
D6		X		X	X		X	X
D7			X	X	X		X	X
D8	X	X				X		
D9	X		X			X		
D10		X	X			X		
D11	X			X		X		
D12		X		X		X		
D13			X	X		X		
D14	X				X	X		
D15			X		X	X		
D16		X	X	X				
D17	X	X		X				
D18	X	X			X			
D19	X		X		X			
D20		X	X		X			
D21	X			X	X			
D22		X		X	X			
D23			X	X	X			
D24	X	X				X	X	X
D25	X		X			X	X	X
D26		X	X			X	X	X
D27	X			X		X	X	X
D28		X		X		X	X	X
D29			X	X		X	X	X
D30	X				X	X	X	X
D31			X		X	X	X	X

Table 4. 64-Bit Error Detection-Correction Calculation (continued)

Data	CB0 Even	CB1 Even	CB2 Odd	CB3 Odd	CB4 Even	CB5 Even	CB6 Even	CB7 Even
D32	X	X	X	X			X	
D33		X		X			X	
D34		X			X		X	
D35			X		X		X	
D36	X	X	X		X		X	
D37				X	X		X	
D38	X	X		X	X		X	
D39	X		X	X	X		X	
D40		X				X		X
D41			X			X		X
D42	X	X	X			X		X
D43				X		X		X
D44	X	X		X		X		X
D45	X		X	X		X		X
D46					X	X		X
D47	X		X		X	X		X
D48	X	X	X	X				X
D49		X		X				X
D50		X			X			X
D51			X		X			X
D52	X	X	X		X			X
D53				X	X			X
D54	X	X		X	X			X
D55	X		X	X	X			X
D56		X				X	X	
D57			X			X	X	
D58	X	X	X			X	X	
D59				X		X	X	
D60	X	X		X		X	X	
D61	X		X	X		X	X	
D62					X	X	X	
D63	X		X		X	X	X	

Document #: MOD0003–1293

DRAM Interfacing

Introduction

The CYM7232/CYM7264 is a high-performance DRAM controller designed for a wide variety of single and multiprocessor environments. It consists of a data path for error detection and correction, FIFO buffering, and funnel multiplexing of data from the bus width to a wide DRAM access. The controller is very flexible. It allows user programmable DRAM timing generation and supports a wide variety of SIMM types. This application note will discuss the CYM72XX to DRAM interface.

Timing Selections

Given a particular memory configuration, the designer selects suitable buffers to receive the controller's RAS, CAS, DRAM address, and Read/Write control signals and drive the DRAM array. The buffer delays depend on device type and total DRAM capacitive load. Appropriate values are chosen for the DRAM timing values to be programmed into registers DT0 through DT4 based on buffer delay, DRAM speed and total load capacitance. This section presents an approach to choosing the timing parameters.

DRAM Configurations and Capacitive Loading

DRAM SIMM modules are available in several varieties. The capacitance loading (pF) for typical DRAM SIMM modules (both single sided—S.S. and double sided—D.S.) configured for use with the CYM7232/CYM7264 is given in *Table 1.*

Each DRAM Accelerator module supports from 1 to 4 blocks of memory. A block consists of 128 bits of simultaneously accessed DRAM data RAMs and their associated error check bits. Diagrams of the various DRAM SIMMs configured for use with the CYM7232/CYM7264 are shown in *Figures 1* and *2.*

The DRAM control lines require buffers to drive the large capacitive load presented by multiple blocks of DRAM. The buffer delays depend on the intrinsic driver delay, capacitive load, and the driver array configuration. Since the signals may be loaded differently, there may be a slight skew in the arrival of various signals at the DRAM. Skew can be minimized by driving all DRAMs in a particular SIMM from the same driver. Sixteen-bit drivers are recommended (12 for the address and 1 or 2 each for RAS and CAS). The delays and skews must be included in timing calculations for the values programmed into registers DT0 through DT4.

Table 1. Capacitive Loading of Signals in pF

SIMM CONFIG	RAS (pF)	CAS (pF)	ADRS (pF)	WR (pF)	CYM7232	CYM7264
1M,4M x 36 S.S.	114	144	88	104	–	4/1 Block
1M,4M x 40 S.S.	90	90	90	90	4/1 Block	–
2M,8M x 36 D.S.	124	124	161	193	–	4/2 Blocks
2M,8M x 40 D.S.	90	90	140	160	4/2 Blocks	–

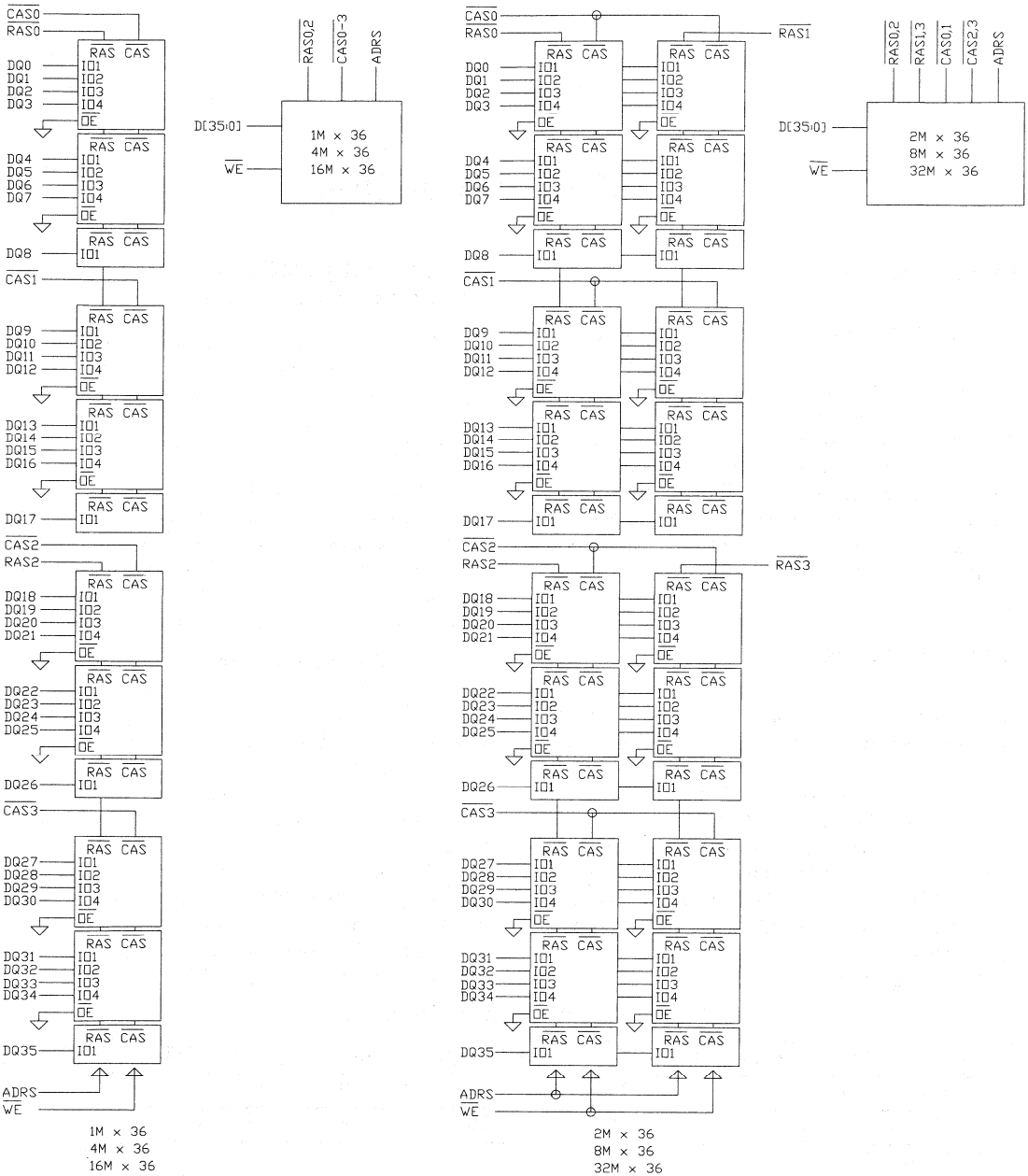


Figure 1. CYM7264 Compatible DRAM SIMMs

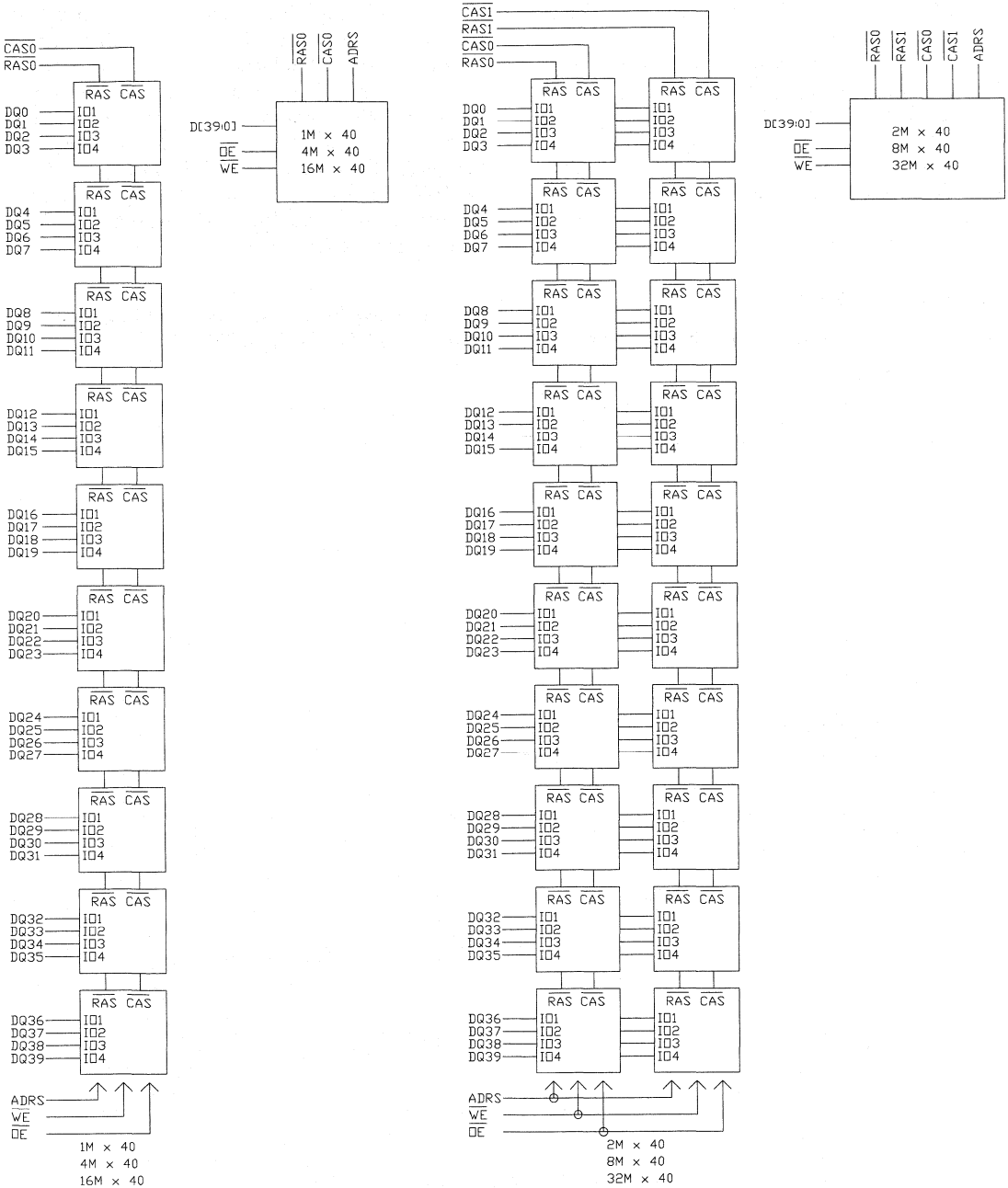


Figure 2. CYM7232 Compatible DRAM SIMMs

Configuration examples for the CYM7232 (CYM7264) are shown in *Figure 3 (4)*. Blocks 0 and 1 in both examples are each populated with 4 single sided SIMMs. Blocks 2 and 3 of each figure share 4 double-sided SIMMs. Double-sided SIMMs may be legally placed in Blocks 0 (occupying Blocks 0 and 1) and/or 2 (occupying Blocks 2 and 3). Single-sided SIMMs may be placed in any block. Single- and double-sided SIMMs may be mixed as long as these restrictions are met.

Recommended Drivers

DRAM array drivers must be fast buffers capable of driving large capacitive loads while minimizing ground bounce. Optimal drivers will have small packages (low lead inductance), series damping resistors (25 ohms is standard) and reduced output swing (less di/dt when discharging the load capacitance). The propagation delays computed below use the Quality Semiconductor, QS74FCT162244CTs and QS74FCT162373CTs, which have many of the characteristics of an ideal driver. Other, similar buffers also meet the requirements.

The capacitive loading for buffer outputs shown in *Figures 3* and *4* is shown in *Table 2.*

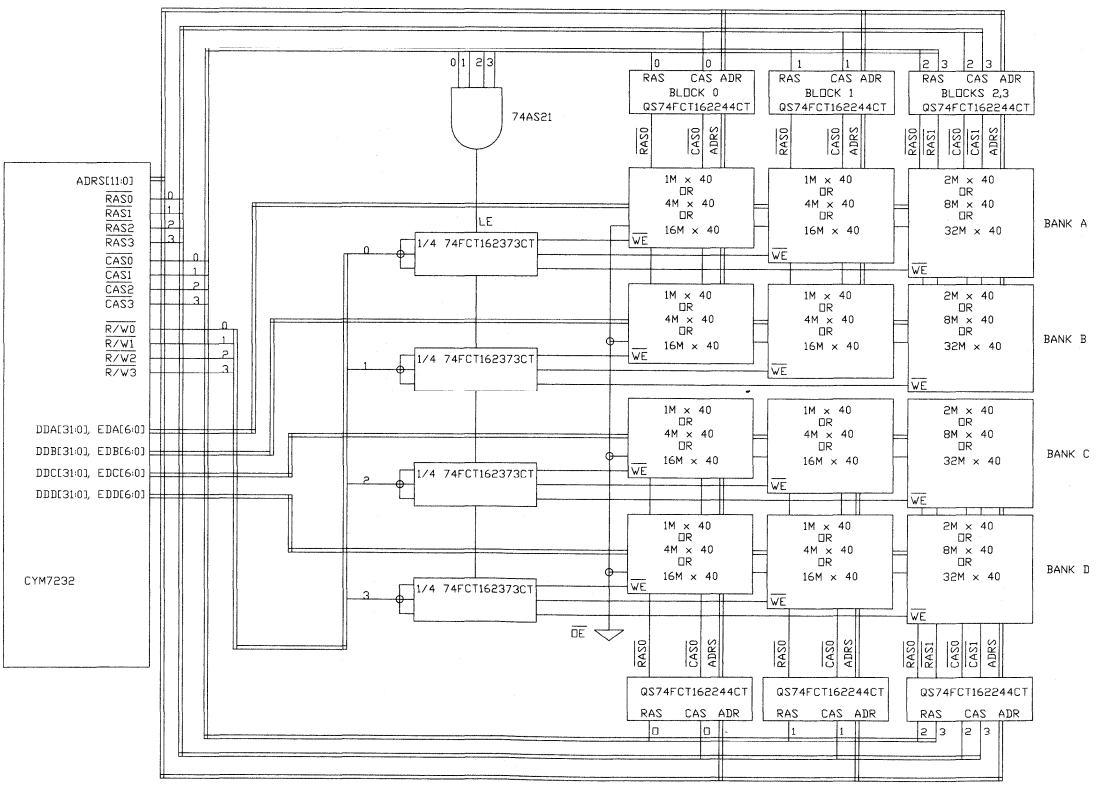


Figure 3. CYM7232 DRAM Array

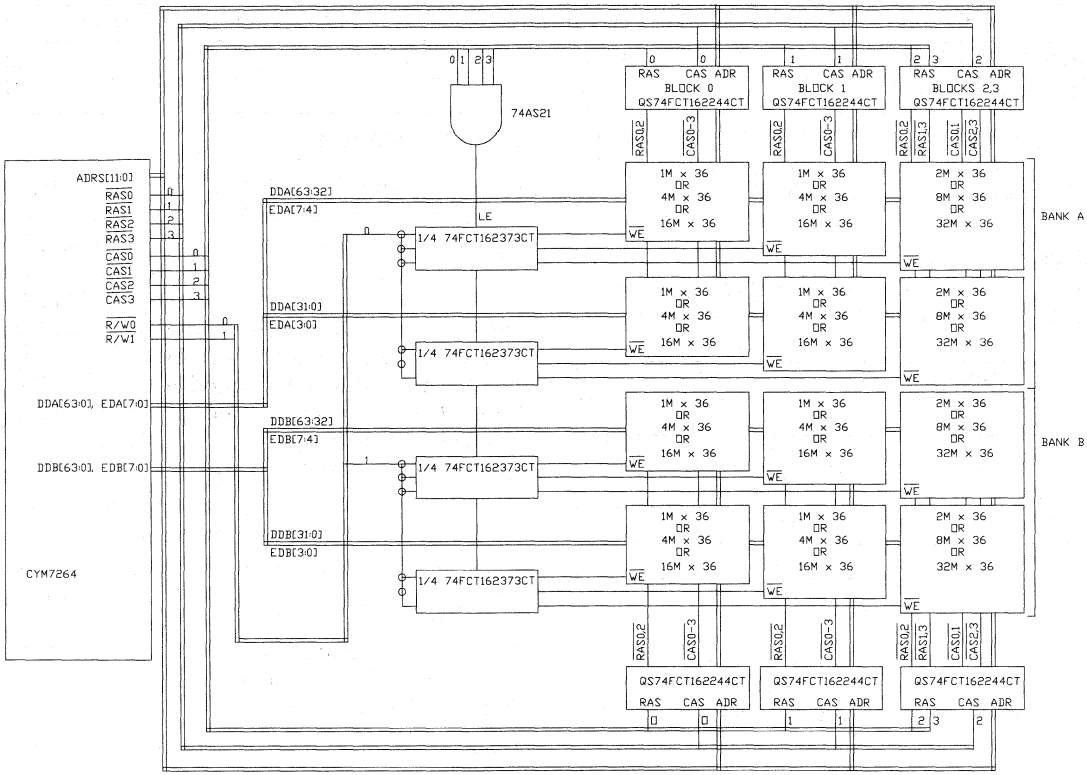


Figure 4. CYM7264 DRAM Array

Table 2. Capacitive Loading of Signals in pF

CYM7232	RAS (162244)	CAS (162244)	ADRS (162244)	WR (162373)
Blocks 0,1	180	180	180	90
Blocks 2,3	180	180	280	160
CYM7264	**RAS (162244)**	**CAS (162244)**	**ADRS (162244)**	**WR (162373)**
Blocks 0,1	228	288	176	104
Blocks 2,3	248	248	332	193

Each buffer output is not allowed to drive more than 20 (24) DRAM loads for the CYM7232 (CYM7264).

The datasheets for the drivers give the delay characteristics at 50 pF load over the commercial voltage and temperature range:

DRIVER	DELAY (1 output switches)	DELAY (all outputs switch)
QS74FCT162244CT	4.1 ns	4.6 ns
QS74FCT162273CT	4.7 ns	5.2 ns

The propagation delay for all outputs switching depends heavily on package type (lead inductance). The values given assume a QSOP.

A graph of propagation delay versus load capacitance (normalized to 50 pF loading) is given in the Quality Semiconductor Data Book, June 1993.

Cap. (pF)	Delay
50 pF	0
100 pF	1.6 ns
150 pF	2.75 ns
200 pF	3.6 ns
250 pF	4.0 ns

Interpolating (and extrapolating for loads exceeding 250 pF), we can expect the delays shown in *Table 3* based on loading for the drivers in *Figures 3* and *4*.

The maximum dispersion of delays for the selected buffer configurations is less than 3 ns for both the CYM7232 and CYM7264. A maximum delay of 10 ns will be used to compute margin for the design examples shown below.

Read Operations

Figure 5 shows the detailed timing sequence for a 64-bit system-bus burst-read operation using the arrays shown in *Figures 3* or *4*. 8−1−2−1 performance (i.e., a 32-byte cache line is transferred in 12 clock cycles − 300 ns) is achieved using a 256-Mbyte DRAM array fully populated with 70-ns SIMMs. The example assumes a ratio of 2 MCLK periods to 1 CLK period, with a CLK period of 25 ns (40-MHz System Bus), the most common application. At an 80-MHz MCLK frequency, the DRAM timing pulses can be resolved to 12.5 ns.

Every timing propagation delay can be divided into three parts: (1) the CYM7232/64 internal programmed time delay, (2) the internal CYM7232/64 propagation delay from multiplied clock (MCLK) to output (This is the AC timing value given in the AC Timing Characteristics section of the datasheet.), and (3) the external propagation delay through the driver array.

The read cycle begins with the assertion of $\overline{AS}$. Internally, the CYM7232(64) requires at least 2 MCLKs (3 MCLKs when the frequency ratio MCLK/CLK = 3) to determine the block mapping and row/column split. Referring to *Figure 5*, the DRAM read operation is timed as shown in *Table 4*.

Table 3. Delay in Signals due to Loading of *Table 2*

CYM7232	RAS (162244)	CAS (162244)	ADRS (162244)	WR (162373)
Blocks 0,1	7.9 ns	7.9 ns	7.9 ns	6.6 ns
Blocks 2,3	7.9 ns	7.9 ns	9.0 ns	8.0 ns
CYM7264	**RAS (162244)**	**CAS (162244)**	**ADRS (162244)**	**WR (162373)**
Blocks 0,1	8.4 ns	9.0 ns	7.4 ns	6.7 ns
Blocks 2,3	8.6 ns	8.6 ns	9.6 ns	8.7 ns

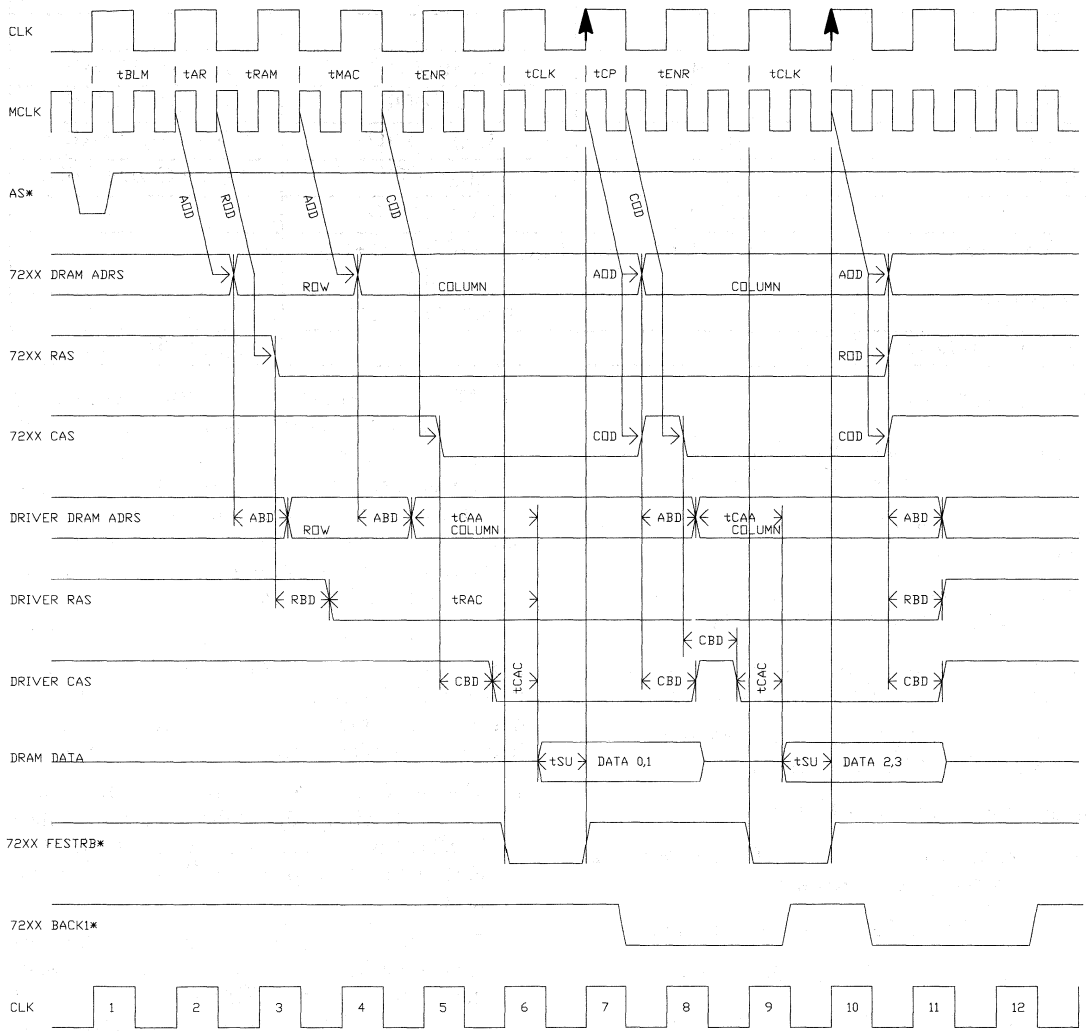


Figure 5. DRAM Read Operation—32 Bytes, 64-Bit Bus, 40 MHz

Table 4. Timing for DRAM Read Operation

Event	Internal	CYM72XX Out	At DRAM
$\overline{AS}$	0		
Row Adrs	t_{BLM}	t_{BLM}+AOD	t_{BLM}+AOD+ABD
RAS	t_{AR}	t_{AR}+ROD	t_{AR}+ROD+RBD
Column Adrs	t_{RAM}	t_{RAM}+AOD	t_{RAM}+AOD+ABD
CAS	t_{MAC}	t_{MAC}+COD	t_{MAC}+COD+CBD
	At DRAM	**CYM72XX In**	
DRAM Read Data	t_{MAC}+COD+CBD+t_{AA}	t_{MAC}+COD+CBD+t_{AA}+t_{SU}	
$\overline{FESTRB}$		t_{ENR}+align+t_{CLK}	
System Bus Data			t_{ENR}+align+2xt_{CLK}

Definition of Terms

t_{BLM}: Number of internal MCLKs to Row Address (2 for X2, X4 MCLK/CLK ratio, 3 for X3 ratio.

t_{AR}: Number of internal MCLKs from Row Address to RAS assertion.

t_{RAM}: Number of internal MCLKs from RAS assertion to Address Mux (to Column Address).

t_{MAC}: Number of internal MCLKs from Address Mux (to Column Address) and CAS assertion.

t_{ENR}: t_{ENR} + align + t_{CLK} = CAS pulse width [0<=align<= (MCLK/CLK)−1].

t_{CLK}: System Bus Clock Period

t_{CP}: Number of internal MCLKs for CAS precharge before accessing next DRAM page.

t_{SU}: CYM72XX DRAM Data Set-Up Time

AOD: CYM72XX ADRS[11:0] output delay

ROD: CYM72XX RAS[3:0] output delay

COD: CYM72XX CAS[3:0] output delay

ABD: CYM72XX ADRS[11:0] buffer delay

RBD: CYM72XX RAS[3:0] buffer delay

CBD: CYM72XX CAS[3:0] buffer delay

There is an internal signal ($\overline{FESTRB}$) which is asserted after the programmable delay t_{ENR}. $\overline{FESTRB}$ is one bus clock wide. The assertion and deassertion of $\overline{FESTRB}$ is aligned to the bus clock boundary. Whenever t_{ENR} does not end on a bus clock boundary, a number of MCLK cycles (align) is added to achieve alignment. Correct programming of the timing registers can force t_{ENR} to end on a bus clock boundary, optimizing performance. See the examples in *Figures 5* and *6*.

DRAM read data is clocked into the CYM72XX pipeline register on the rising edge of the bus clock in which $\overline{FESTRB}$ is asserted. The DRAM read data must meet the CYM72XX set-up time as specified in the datasheet. There are three constraints.

1. The time from RAS assertion at the DRAM devices to the rising edge of the bus clock in which $\overline{FESTRB}$ is asserted must exceed the DRAM RAS access time plus the CYM72XX set-up time.
2. The time from multiplexed column address arrival at the DRAM devices to the rising edge of the bus clock in which $\overline{FESTRB}$ is asserted must exceed the DRAM column address access time plus the CYM72XX set-up time.
3. The time from CAS assertion at the DRAM devices to the rising edge of the bus clock in which $\overline{FESTRB}$ is asserted must exceed the DRAM CAS access time plus the CYM72XX set-up time.

The following notation is used for DRAM and 72XX datasheet parameters:

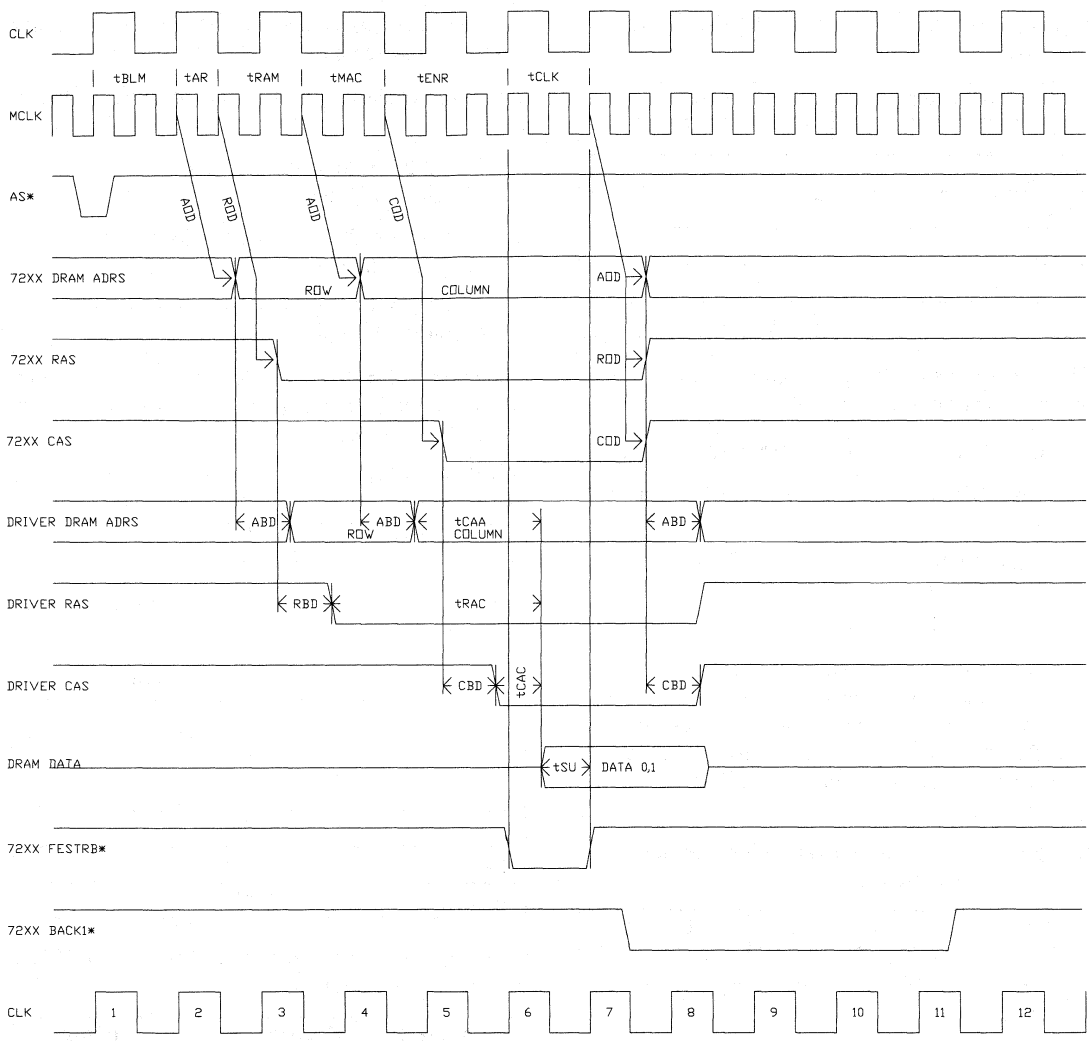


Figure 6. DRAM Read Operation—16 Bytes, 32-Bit Bus, 40 MHz

t_{RAC}: RAS access time (from the DRAM datasheet)

t_{CAA}: CAS address access time (from the DRAM datasheet)

t_{CAC}: CAS access time (from the DRAM datasheet)

t_{CP}: CAS precharge (from the DRAM datasheet)

t_{RPR}: RAS precharge (from the DRAM datasheet)

t_{RAS}: RAS pulse width (from the DRAM datasheet)

t_{SU}: data to clock set-up time (CYM72XX datasheet)

Key timings from typical 70-ns DRAM SIMMs are given below:

Parameter	Value
t_{RAC}	70 ns
t_{CAA}	35 ns
t_{CAC}	20 ns
t_{CP}	10 ns
t_{RPR}	60 ns
t_{RAS}	70 ns

The constraints on the programmed timing values are summarized as follows:

1. $t_{ENR} + t_{CLK} > t_{COD} + t_{CBD} + t_{CAC} + t_{SU}$
2. $t_{MAC} + t_{ENR} + t_{CLK} > t_{AOD} + t_{ABD} + t_{CAA} + t_{SU}$
3. $t_{RAM} + t_{MAC} + t_{ENR} + t_{CLK} > t_{ROD} + t_{RBD} + t_{RAC} + t_{SU}$
4. $t_{RAM} + t_{MAC} + t_{RIN} >= t_{RAS}$

The last equation can be used to automatically determine t_{RIN}. When a write is inhibited, CAS is not asserted and the read cycle is aborted. RAS continues after the abort for t_{RIN} to insure that the minimum RAS pulse width meets the DRAM device specifications.

Using *Figure 5*, we can set values for internal 72XX read timing parameters:

Parameter	Register Value	Timing
t_{AR}	0	12.5 ns
t_{RAM}	1	25 ns
t_{MAC}	1	25 ns
t_{ENR}	2	37.5 ns
t_{CP}	0	12.5 ns
t_{RIN}	1	25 ns
t_{RAS}	5	75 ns
t_{RPR}	4	62.5 ns

The value t_{RAM}=1 gives ample Row Address hold time. Setting t_{MAC}=1 and t_{ENR}=0 forces the optimal alignment constraint (align=0). The values for t_{RAS} (used during refresh operations) and t_{RPR} are taken directly from the DRAM datasheet.

The three constraint equations are satisfied as follows:

1. 37.5 + 25 > 16 + 10 + 20 + 4
   Margin = 12.5 ns
2. 25 + 37.5 + 25 > 16 + 10 + 35 + 4
   Margin = 22.5 ns
3. 25 + 25 + 37.5 + 25 > 16 + 10 + 70 + 4
   Margin = 12.5 ns

The margins for this example are substantial. Only two remaining timing parameters (for writes) are left undetermined.

Another read timing example is shown in *Figure 7*. This example uses 60-ns DRAM SIMMs with a 33-MHz system clock and 66-MHz DRAM clock to achieve 7–1–2–1 performance.

The relevant 60-ns DRAM SIMM timings are given below:

Parameter	Value
t_{RAC}	60 ns
t_{CAA}	35 ns
t_{CAC}	20 ns
t_{CP}	10 ns
t_{RPR}	50 ns
t_{RAS}	60 ns

Using *Figure 7* we can set values for internal 72XX read timing parameters:

Parameter	Register Value	Timing
t_{AR}	0	15 ns
t_{RAM}	0	15 ns
t_{MAC}	0	15 ns
t_{ENR}	2	45 ns
t_{CP}	0	15 ns
t_{RIN}	1	30 ns
t_{RAS}	3	60 ns
t_{RPR}	3	60 ns

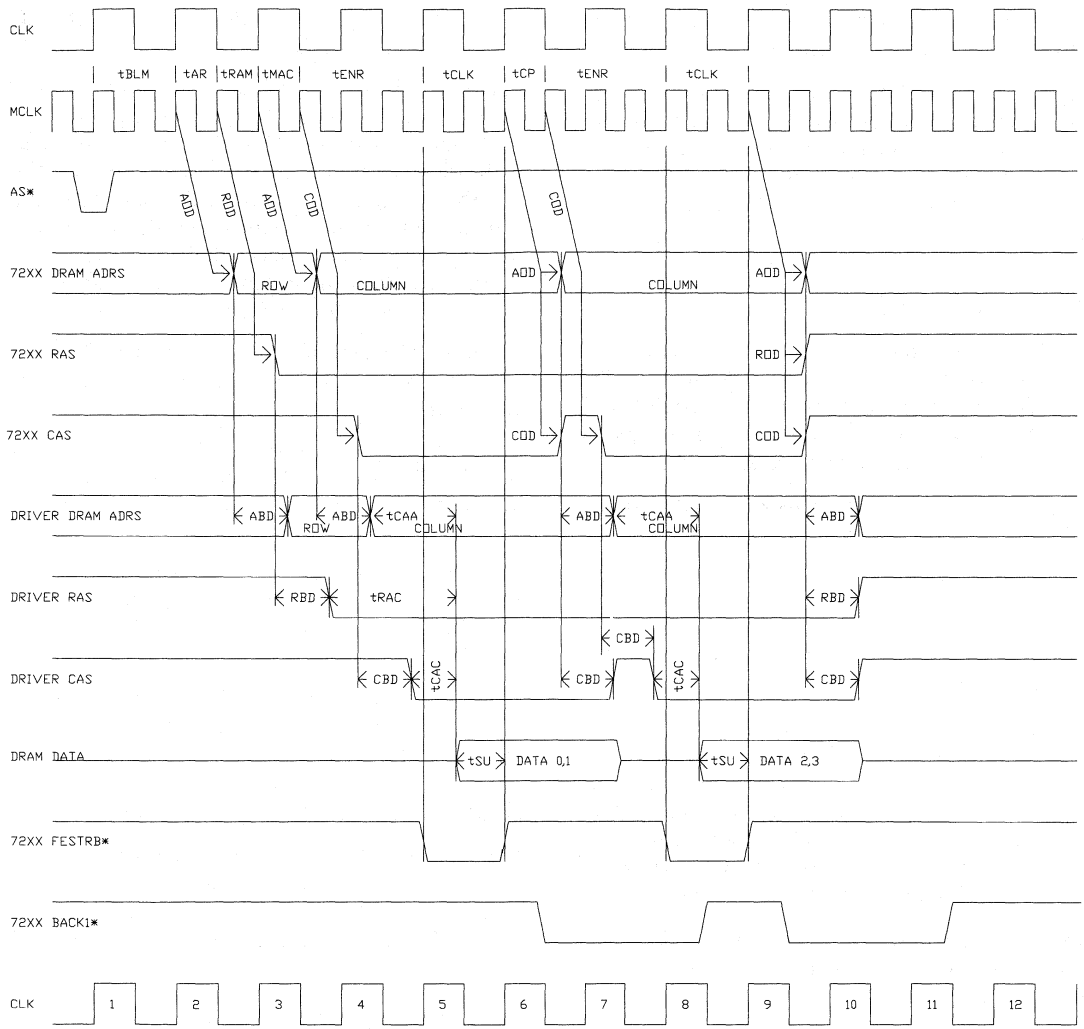


Figure 7. DRAM Read Operation—32 Bytes, 64-Bit Bus, 33 MHz

The constraints on the programmed timing values are now computed with the AC timing characteristics for the 33-MHz version of the CYM72XX:

1. $t_{ENR} + t_{CLK} > t_{COD} + t_{CBD} + t_{CAC} + t_{SU}$
   22-ns margin
2. $t_{MAC} + t_{ENR} + t_{CLK} > t_{AOD} + t_{ABD} + t_{CAA} + t_{SU}$
   22-ns margin
3. $t_{RAM} + t_{MAC} + t_{ENR} + t_{CLK} > t_{ROD} + t_{RBD} + t_{RAC} + t_{SU}$
   12-ns margin

Equation 3 places a limit on the RAS access time of the DRAM SIMMs. A margin of 12 ns suggests that

70-ns SIMMs could almost be acceptable in this example (leaving a 2-ns margin).

Figure 6 shows a 16-byte read operation (i.e., 486 or 68040 line fill) using a CYM7232 in a 32-bit bus application. The timings are the same as in *Figure 5*. The burst is contiguous (only one 128-bit DRAM access is required). 8–1–1–1 performance is achieved.

Write Operations

A 32-byte write burst operation (cache-line purge over a 64-bit bus) is shown in *Figure 8*. The CYM72XX waits for 128 bits to accumulate in its FIFOs before writing the first of two DRAM pages. Additional variables are defined as follows:

t_{DOH}: CYM72XX DDA, DDB, DDC, DDD, EDA, EDB, EDC, EDD, DRAM Data hold times.

t_{DH}: DRAM data hold time after CAS fall during early write cycle (from DRAM spec).

WBD: Write latch delay when transparent (i.e., 74FCT162373CT).

Note: The preliminary datasheet has an error for t_{DOH} parameter which will be corrected when the final datasheet is issued. The actual minimum value for t_{DOH} is 20 ns. This represents the minimum time required to read the next value from the write FIFOs and append the error polynomial. The typical DRAM SIMM hold requirement is 15 ns.

Two remaining parameters can now easily be established (the register values appear in parentheses after the timing value):

1. CAS pulse width for writes:
   $t_{ENW} + t_{CLK} > t_{CAC}$
   t_{ENW} =12.5 ns (0)
2. Data set-up to CAS fall:
   $t_{DC} + COD + CBD > DOD + WBD$
   t_{DC} = 25 ns (1)
3. Data hold after CAS fall:
   $t_{CLK} + t_{DOH} > t_{COD} + t_{CBD} + t_{DH}$

Equation 1 gives a margin for CAS pulse width of 17.5 ns. The CYM72XX constrains the falling edge of CAS to be aligned with the rising edge of CLK during write operations.

Equation 2 gives extraordinary margin for data set-up time to CAS falling edge (23 ns). The write enable signals R/W0 and R/W1 originate from the data path with the same propagation delays as the DRAM data signals. These write signals form the actual critical set-up path to DRAM during an early write cycle operation. Since the CAS falling edge is forced to a clock boundary the minimum t_{DC} is forced to 25 ns by the alignment constraint.

Equation 3 establishes the requirement for data hold time from CAS falling edge. The margin is 4 ns at 40 MHz

Caution: Substantial noise can be generated when the CYM7232 switches 156 data bits simultaneously into highly capacitive loads (as much as 100 pF for 4 blocks of memory) on writes.

Figure 9 shows the timing diagram for write bursts at 33 MHz. The same constraints apply to t_{ENW} and t_{DC}. For this case t_{ENW} = 15 ns (0) and t_{DC} = 30 ns (1). The t_{DH} margin is 7 ns.

Figure 10 shows the timing diagram for a DRAM write operation of 16 Bytes over a 32-bit-wide (4 bytes) bus at a CLK frequency of 40 MHz.

Read-Modify-Write Operation

Figure 11 shows the timing for a Read-Modify-Write operation. The data read is sampled on the rising edge of CLK during cycle 7 and written to the DRAM by the falling edge of CAS driving CLK cycle 9.

DRAM Refresh Operations

Figure 12 shows the timing for a normal DRAM refresh operation. The memory array refresh period is programmable, but should be at least every two milliseconds.

Figure 13 shows the timing for a DRAM refresh operation with scrubbing. If either a single-bit error is detected and corrected or a double-bit error occurs, a scrub operation is performed and an interrupt is generated so that the intelligence at the next higher level of control can take the appropriate action.

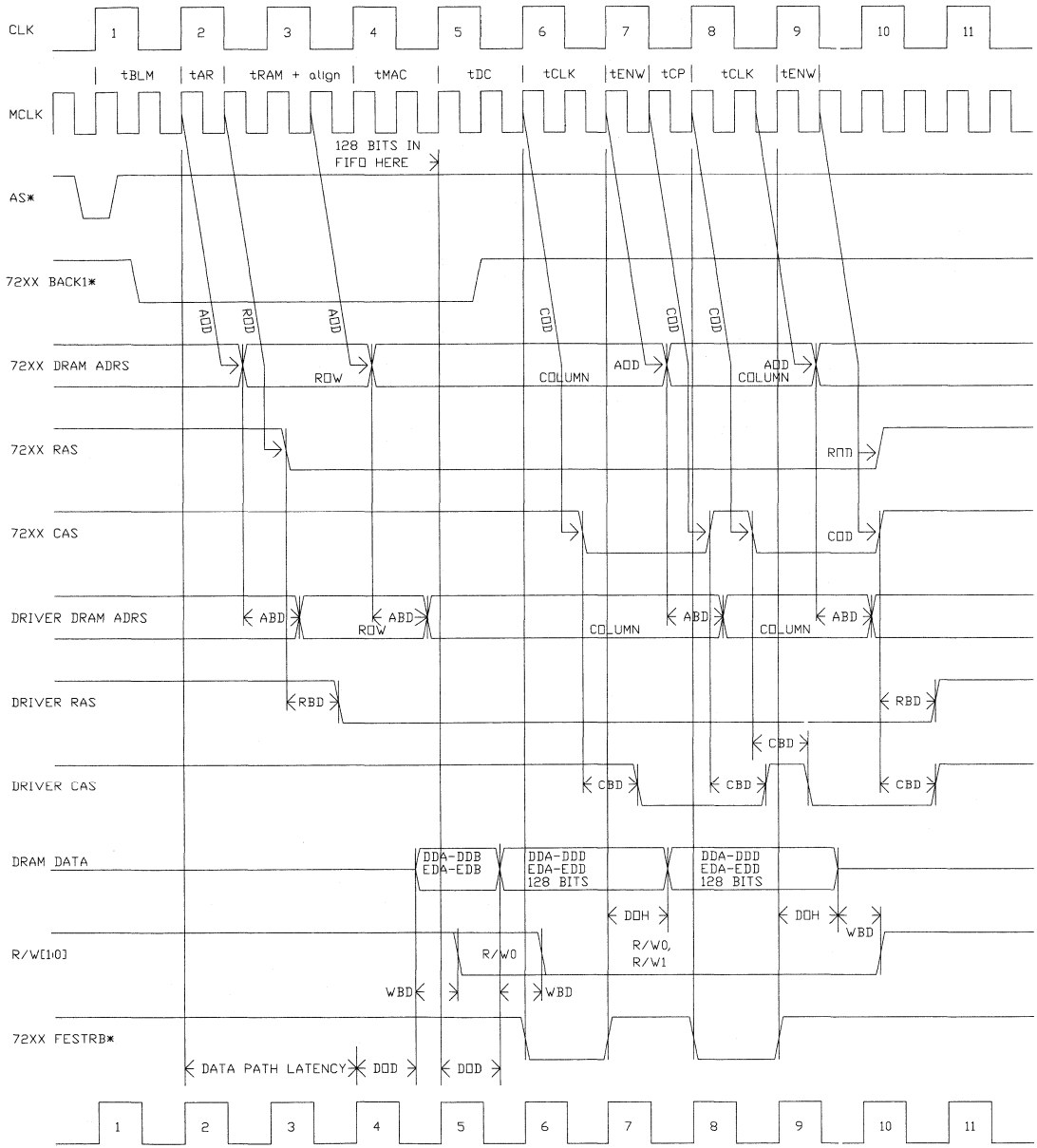


Figure 8. DRAM Write Operation—32 Bytes, 64-Bit Bus, 40 MHz

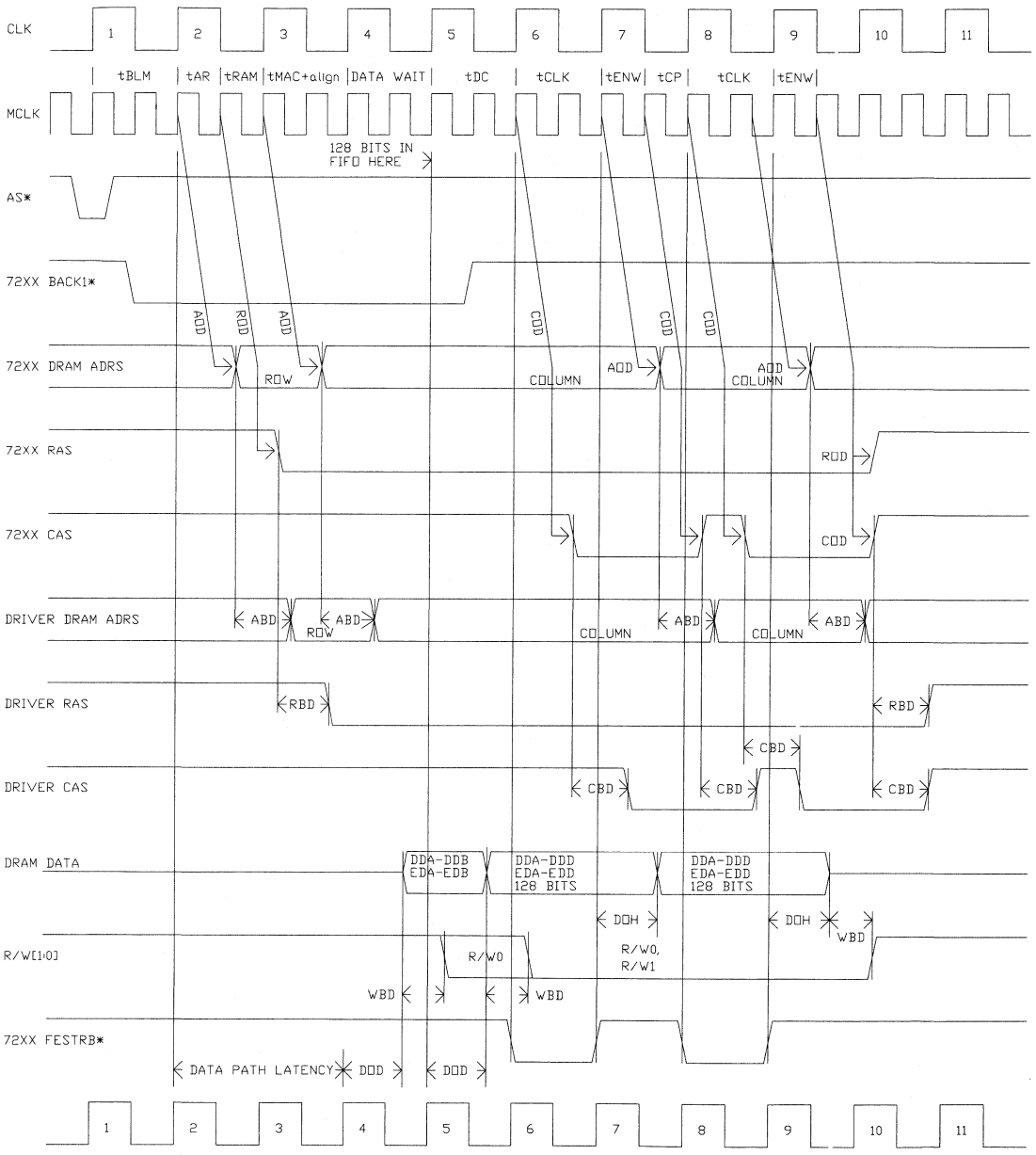


Figure 9. DRAM Write Operation—32 Bytes, 64-Bit Bus, 33 MHz

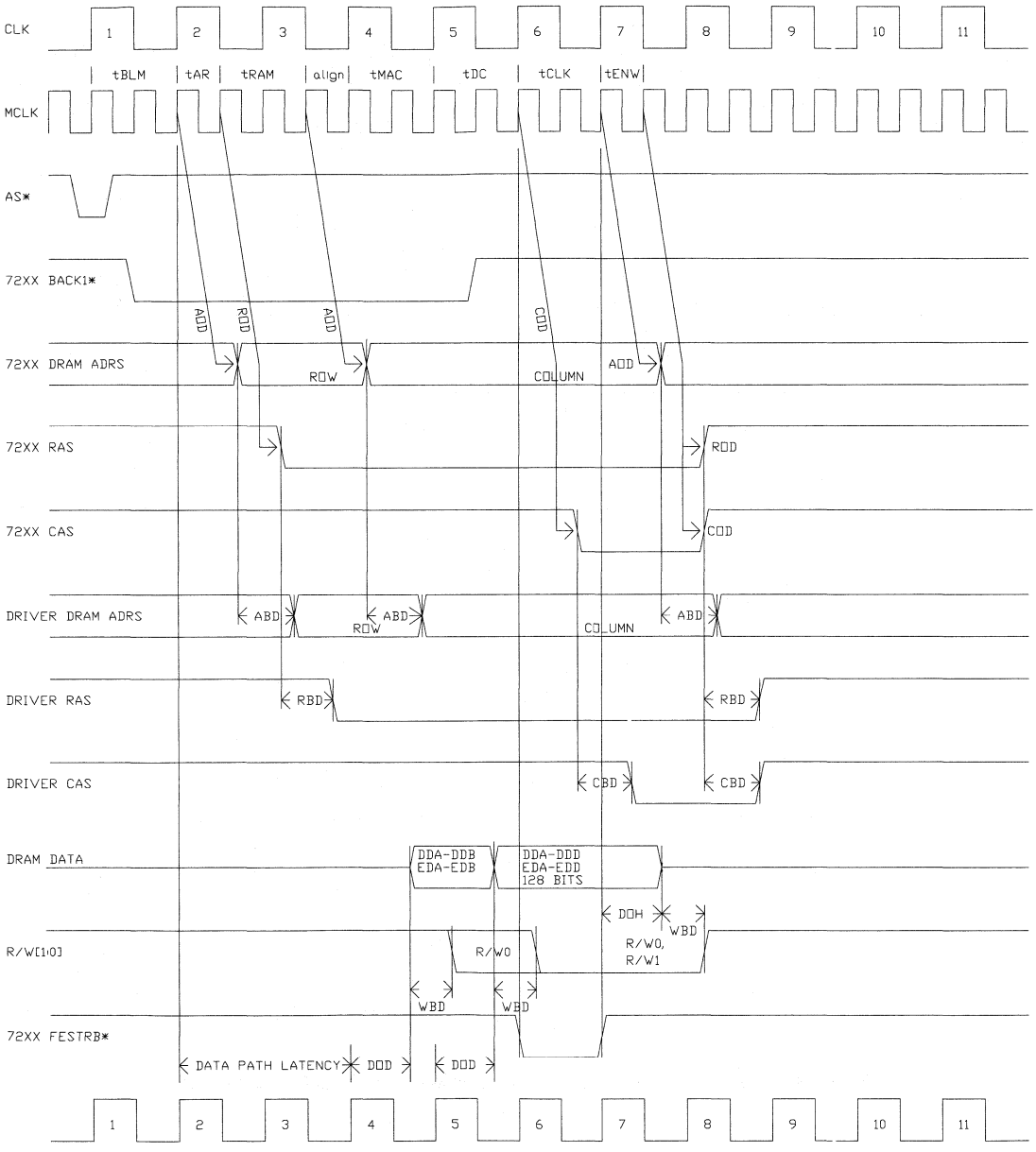


Figure 10. DRAM Write Operation—16 Bytes, 32-Bit Bus, 40 MHz

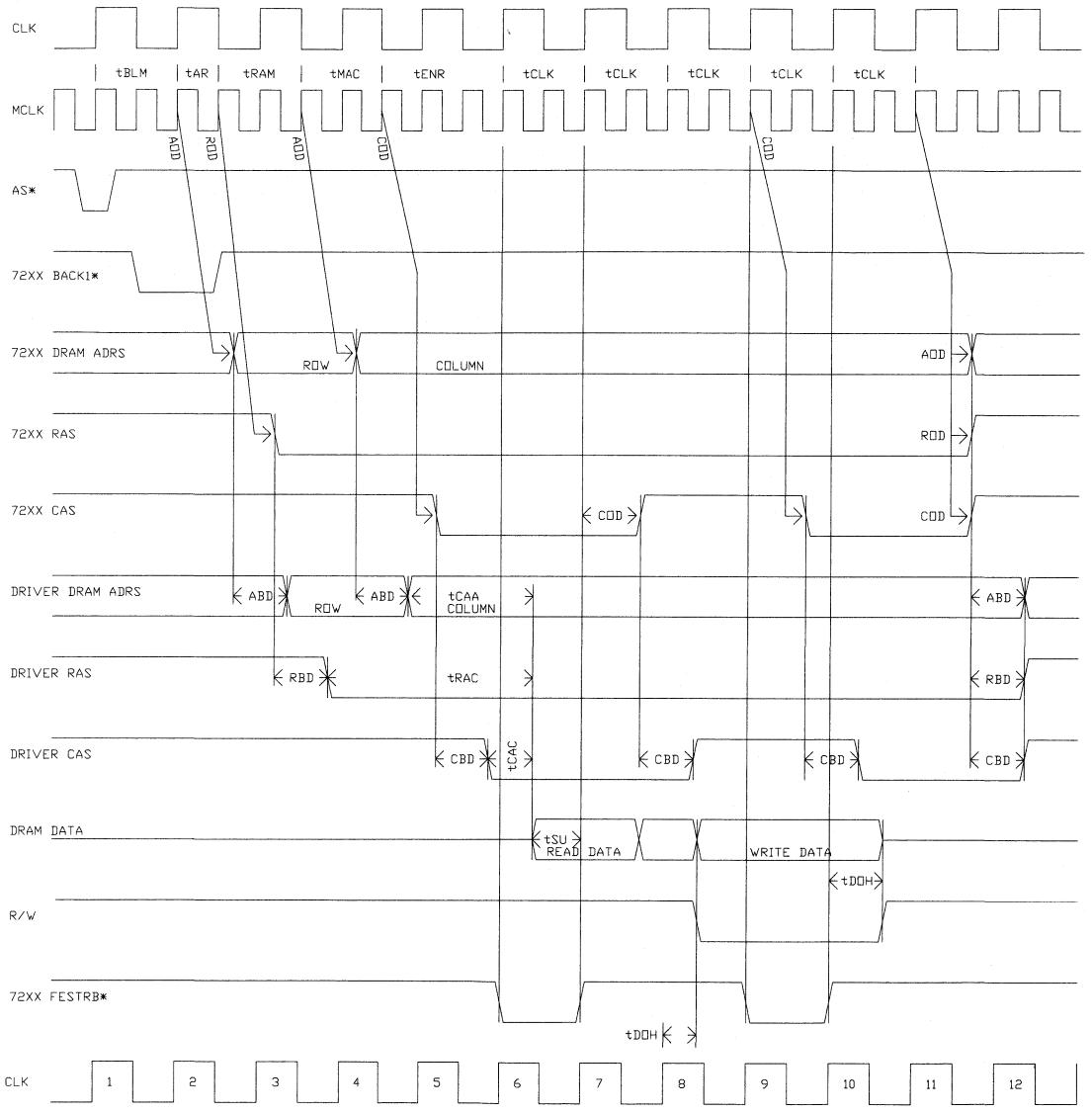


Figure 11. DRAM Read-Modify-Write Operation—32-/64-Bit Bus, 40 MHz

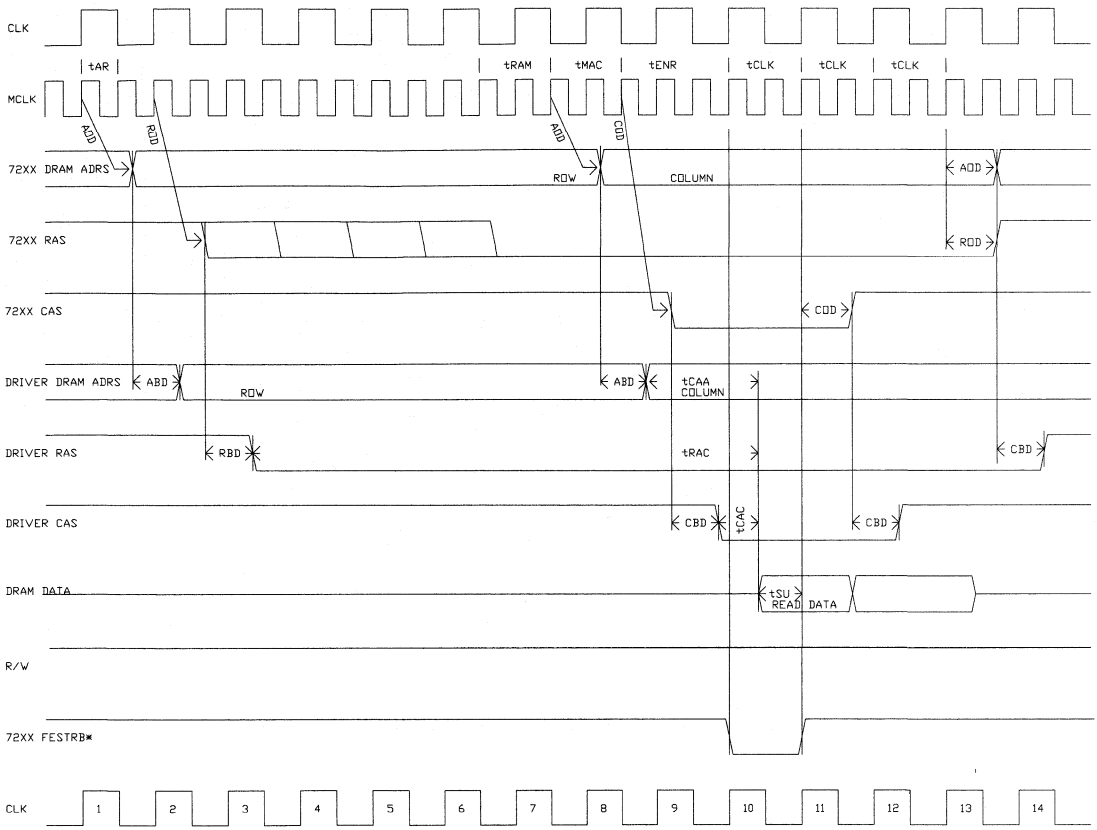


Figure 12. DRAM Refresh Operation—40 MHz

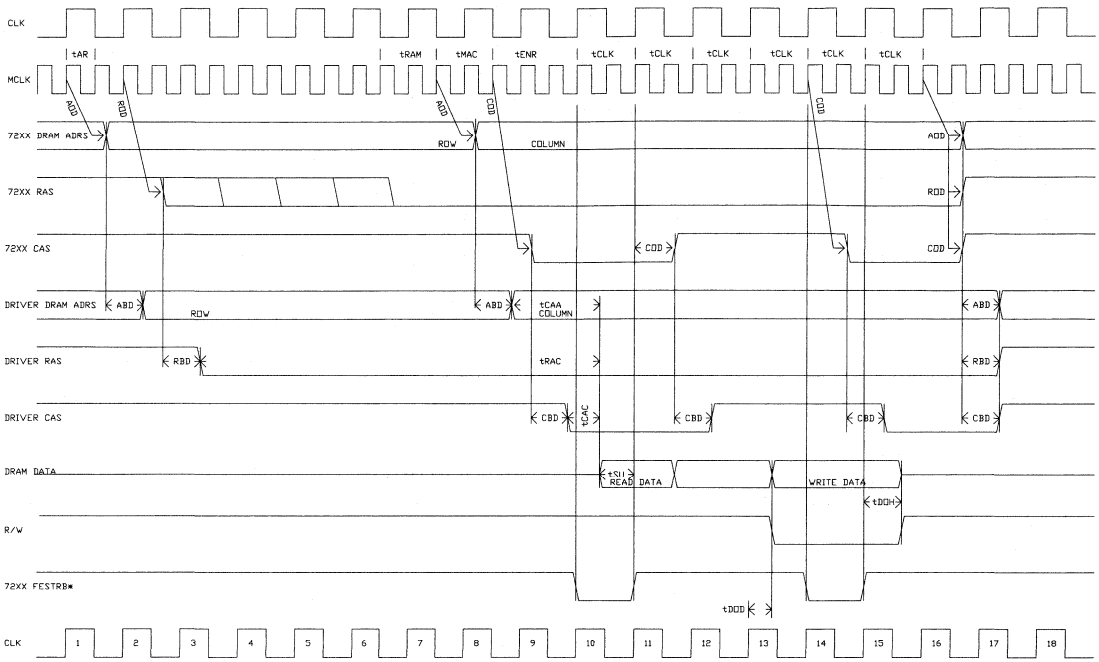


Figure 13. DRAM Refresh Operation with Scrub—40 MHz

DRAM Configurations: 32-Bit EDC

DRAM Type	DRAM Bank	DRAM Block	Capacity per Block (Mbytes)	Number of Blocks	Capacity per System (Mbytes)
256Kx1	256Kx39	4x256Kx39	4	1 – 4	4 – 16
1Mx1	1Mx39	4x1Mx39	16	1 – 4	16 – 64
4Mx1	4Mx39	4x4Mx39	64	1 – 4	64 – 256
16Mx1	16Mx39	4x16Mx39	256	1 – 4	256 – 1024
256Kx4	256Kx40	4x256Kx40	4	1 – 4	4 – 16
1Mx4	1Mx40	4x1Mx40	16	1 – 4	16 – 64
4Mx4	4Mx40	4x4Mx40	64	1 – 4	64 – 256
16Mx4	16Mx40	4x16Mx40	256	1 – 4	256 – 1024

DRAM Configurations: 64-Bit EDC

DRAM Type	DRAM Bank	DRAM Block	Capacity per Block (Mbytes)	Number of Blocks	Capacity per System (Mbytes)
256Kx1	256Kx72	2x256Kx72	4	1 – 4	4 – 16
1Mx1	1Mx72	2x1Mx72	16	1 – 4	16 – 64
4Mx1	4Mx72	2x4Mx72	64	1 – 4	64 – 256
16Mx1	16Mx72	2x16Mx72	256	1 – 4	256 – 1024
256Kx4	256Kx72	2x256Kx72	4	1 – 4	4 – 16
1Mx4	1Mx72	2x1Mx72	16	1 – 4	16 – 64
4Mx4	4Mx72	2x4Mx72	64	1 – 4	64 – 256
16Mx4	16Mx72	2x16Mx72	256	1 – 4	256 – 1024

Document #: MOD0002–1293

CYM7232/7264 DRAM Accelerator: Mixing 5-Volt and 3.3-Volt DRAMs

Introduction

The CYM7232/7264 is a high-performance DRAM controller designed for a wide variety of single and multiprocessor environments. It consists of a data path for error detection and correction, FIFO buffering, and funnel multiplexing of data from the bus width to a wide DRAM access. The controller is very flexible and may easily be interfaced to many different processors.

This application note describes a method for mixing 5-volt and 3.3-volt DRAM SIMMs in one system. Presently, all available DRAM SIMMs (using 4-MB and 16-MB technologies) require 5-volt power supplies and have TTL-compatible interfaces. Future generations of DRAM SIMMs (using 64-MB technology) will probably use 3.3-volt power supplies. The CYM7232/64 DRAM Accelerator can support both types of SIMMs in a single system.

5-Volt DRAM Interface

A complete discussion of the 5-volt DRAM interface is contained in the Cypress Semiconductor CYM7232/64 Application Note titled "DRAM Interfacing." The document describes the problems associated with capacitive loading, buffering, and DRAM timing for a variety of bus speeds and 5-volt memory access times.

3.3-Volt DRAM Interface

5-volt to 3.3-volt translators are required to interface with 3.3-volt-only DRAM SIMMs. Bidirectional translators are used for the data lines and unidirectional buffer translators are needed to drive the $\overline{RAS}$ signals, the $\overline{CAS}$ signals, the DRAM addresses, and the read/write signal R/$\overline{W}$. A schematic of the CYM7264 interface with four 128-bit DRAM blocks[1] arranged as two banks[1] is shown in *Figure 1*. The first two blocks are constructed using 36-bit-wide, 5-volt-only single-sided DRAM SIMMs. The other two blocks of DRAM are formed from 72-bit-wide, 3.3-volt-only dual-sided DRAM SIMMs. Because each dual-sided SIMM supports 72 bits of data, it is shown as a single unit in *Figure 1*.

Data and Check Bits Translation

The 128 data lines and 16 error check lines (64 data bits and 8 error check bits from each bank) of the four 3.3V DRAM SIMMs are connected to the 5-volt-only DRAM SIMMs via bidirectional translators such as the 16-bit 74FCT163245ST. Naturally, the DRAM timing calculations should include the translator delay.

Control Signal Translation

The $\overline{RAS}$, $\overline{CAS}$, and address lines are buffered by 3.3V non-inverting buffers such as the 16-bit 74FCT163244. The write signals, on the other hand, are buffered by transparent latches such as the 74FCT163373. In *Figure 1*, all write signals to the DRAMs (both 5V and 3.3V devices) are buffered by the same set of 3.3V latches. Driving the write inputs with a 3.3V device is acceptable because the 5V DRAMs have TTL-compatible I/Os.

The two R/$\overline{W}$ signals from the latch are also used as the direction controls for each bank of the bidirectional data translators. Series terminating resistors

are advised, as shown, to reduce ringing when driving highly capacitive loads with fast signal edges.

Document #: MOD0005–1293

Notes

1. For the definition of banks and blocks please refer to the CYM7232/64 datasheet.

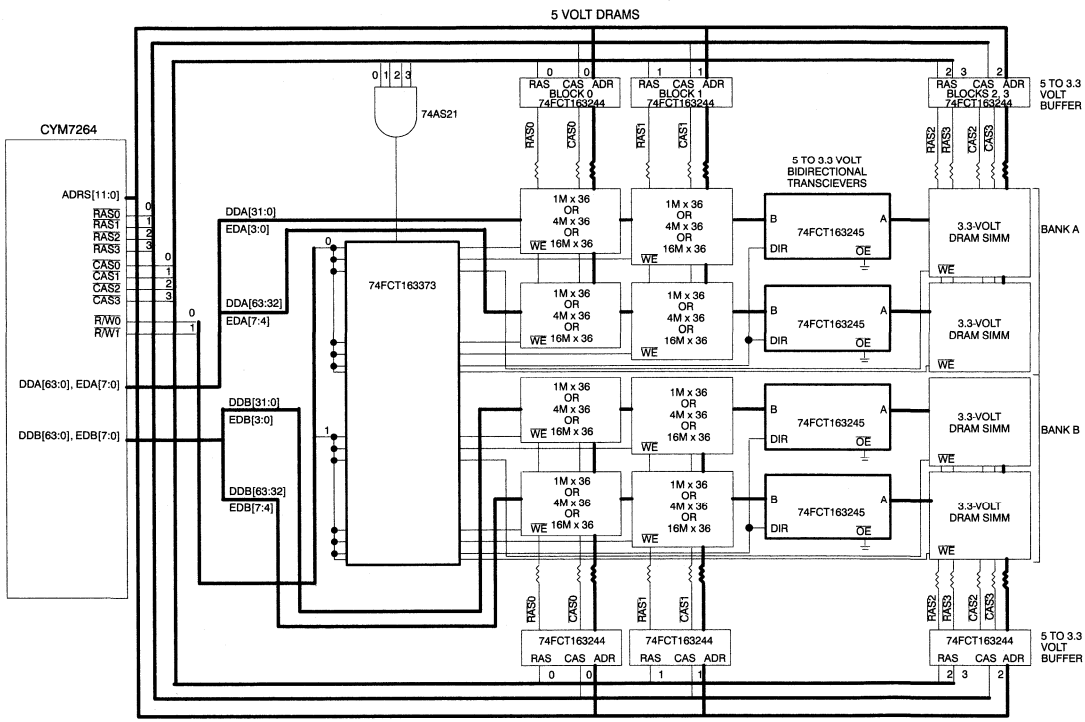


Figure 1. DRAM Connection to the CYM7264

CAM-based Products – 9





CAM-based Products Section Contents and Abstracts

Introduction to the SmartCAM **9–1**

This application note introduces the Cypress SmartCAM™. First, the concept of CAMs is introduced to the reader. Next, the architectural features of the SmartCAM, along with its modes of operation, are described. Finally, some applications well suited to the SmartCAM are described, along with the hardware and software support required to develop these applications.

SmartCAM Hardware Design Considerations **9–12**

This application note is intended to describe the design considerations that apply to a SmartCAM™ hardware design. The first section, "Timing Considerations," focuses on issues that affect any SmartCAM design. The second section, "Cascading the SmartCAM," discusses the technical aspects of building a SmartCAM array.

Pattern Matching with the SmartCAM **9–21**

With its simultaneous search property, the SmartCAM is well suited for recognizing images patterns using the straight-forward template matching technique or the more practical fuzzy template approach. Optical character recognition is used as an example to illustrate the template method. The SmartCAM is also useful in motion estimation between two sequential video frames. It can quickly determine where and 8 by 8 pixel block has moved from one frame to another with a 16-pixel by 16-pixel content.

Introduction to SmartCAM™

Overview

This application note introduces the Cypress SmartCAM™. First, the concept of CAMs is introduced to the reader. Next, the architectural features of the SmartCAM, along with its modes of operation, are described. Finally, some applications well suited to the SmartCAM are described, along with the hardware and software support required to develop these applications.

What is a CAM?

A CAM (Content-Addressable Memory) is defined as a memory that is capable of determining whether a given datum—the search word—is located at one of its addresses[1].

A CAP (Content-Addressable Processor) is a CAM with additional on-chip logic to perform sophisticated data transformations on its contents.

How Are CAMs Different from Conventional Memories ?

As indicated above, a CAM is accessed on the basis of its contents. Conventional memories (SRAM, DRAM, PROM) are accessed based on the location. Hence, while an address is presented to conventional memories, a data pattern is presented to a CAM to access its stored data.

A compare operation is the most basic operation that can be performed on a CAM because data is presented to a CAM. A pure CAM performs this compare operation in parallel inside the device, but this feature can also be emulated slowly in software. In addition, since a search is the most basic CAM operation, a CAM is most ideally suited for pattern recognition and search-intensive applications.

Finally, it must be emphasized that an equality-match search-comparison is not the only mode supported by CAMs. Other searches such as not equal to, partial match, greater than, less than, minimum, and maximum, may also be supported.

What is SmartCAM ?

SmartCAM is a Single Instruction Multiple Data (SIMD) device based on a CAM core and an array of high-speed processors. It falls into the category of content-addressable processors, rather than a pure content-addressable memory.

The CAM array inside SmartCAM is organized as 1024 words, each 42 bits wide. An array of 1024 single-bit on-chip ALUs processes search results generated by the CAM core. In addition, registers controlled by an on-chip state machine are provided to support data manipulation.

The SmartCAM communicates with external logic via a memory-mapped peripheral type interface. Both on-chip registers and CAM entries can be accessed via the 32-bit data port and the control signals. Commands issued to the SmartCAM are interpreted by an on-chip state machine that performs the specified operation. Finally, a handshake with external logic is provided by the Ready output, which is asserted when an operation is completed.

If a search pattern is longer than 42 bits, adjacent entries in the array can be concatenated to form a longer search string. Cascading of devices in 1K-entry increments is also supported. In this case, the open-drain $\overline{\text{MATCH}}$ outputs of various devices can be connected together with no additional delay.

Target applications for SmartCAM include pattern recognition, digital noise filtering, database accelerators, graphics processing and neural networks. In all of these search-intensive applications, SmartCAM provides a hardware solution for searching. Since a hardware search is inherently faster than a software search, using SmartCAMs in such systems can increase system performance considerably.

The next section discusses the architecture of SmartCAM in more detail.

Architecture of SmartCAM

SmartCAM is a high-performance CAM-based associative processor targeted towards search-intensive applications. This device, when used in hardware systems, provides a significant performance increase over conventional (software) methods of searching.

Figure 1 shows the logic block diagram for SmartCAM. The SmartCAM logic can be functionally divided into:

- CAM array
- Interface and Control registers
- ALUs and Row logic
- State machine

Each of these functional blocks is discussed in greater detail in the following sections.

CAM Array

The CAM array is organized as 1042 words with 42 bits per word. The 42-bit word serves two important purposes: (1) it provides a wider search pattern, and (2) the upper 10-bits of a word can be used as an address identifier for that word. The CAM array is used solely for storing data. Patterns that need to be matched with the CAM contents can be loaded into the $Data_{IN}$ register or supplied directly from the 32 data input pins. A Match operation is completed in 4 clock cycles and, if a match is found, is indicated by the assertion of the $\overline{MATCH}$ output signal.

Interface and Control Registers

The Interface and Control registers are accessed via the 32-bit data port (D31–D0) using access codes

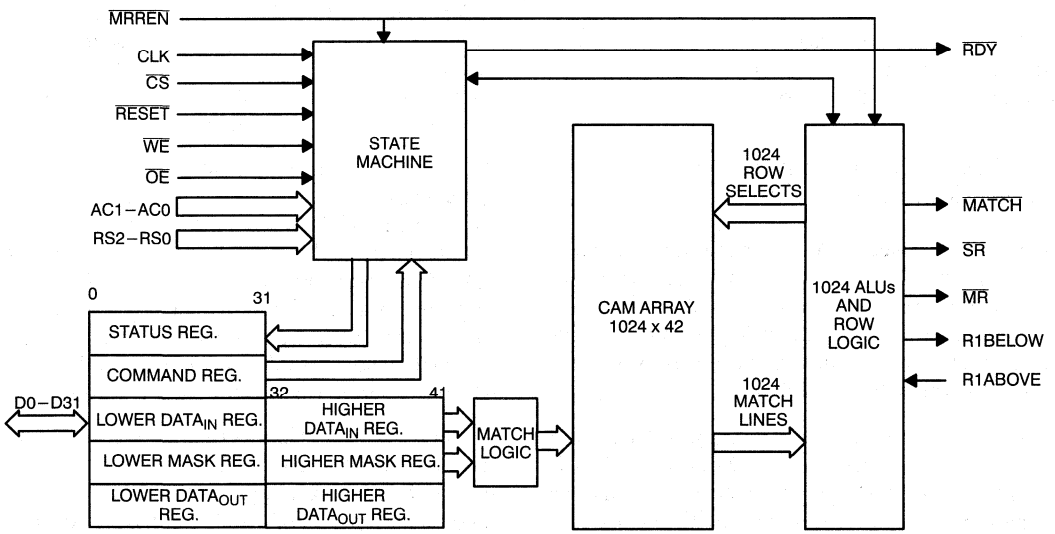


Figure 1. Logic Block Diagram of the SmartCAM

(AC1–AC0), the register select inputs (RS2–RS0), the chip select input ($\overline{CS}$), and the MRR enable input ($\overline{MRREN}$).

SmartCAM contains five interface and control registers: $Data_{IN}$, $Data_{OUT}$, Mask, Command, and Control/Status.

$Data_{IN}$ Register

The $Data_{IN}$ register is 42 bits wide. It is loaded with the pattern to be compared against the contents of the CAM array. As the data port is only 32 bits wide, this register is loaded in two steps: the lower 32 bits and the upper 10 bits.

$Data_{OUT}$ Register

The $Data_{OUT}$ register is 42 bits wide. It is used to store the CAM word obtained by the READ instructions.

Mask Register

The Mask register is 42 bits wide. If a bit position in this register is set, the corresponding bit position in the $Data_{IN}$ register will be disqualified from the match process or the Write operation. During CAM WRITE instructions, all column positions marked by a 1 in the Mask register will not be updated. Mask patterns can be loaded externally or generated internally under program control.

Command Register

The Command register is 32 bits wide. It holds one of the twelve instructions supported by SmartCAM. This register is loaded via the 32-bit data port.

Control/Status Register

The Control/Status register is 32 bits wide. It holds control and status information for the device. The status information includes the state of the $\overline{SR}$, $\overline{MR}$, $\overline{MATCH}$, $R1A_{OUT}$ and $R1B_{OUT}$ outputs, the $\overline{MFF}$ ($\overline{MRREN}$_FF) value, and the 10-bit CAM Address field. *Figure 2* shows the organization of this register.

$\overline{SR}$, $\overline{MR}$, $\overline{MATCH}$, $R1A_{OUT}$ and $R1B_{OUT}$ indicate Some Response, Multiple Response, Match, Top bit in R1 and Bottom bit in R1 respectively. These signals corresponding bits in the Command/Status register are updated at the end of every instruction. All have values corresponding to the output pins of the same name.

When a READ, READSHIFT, or READSNEXT instruction is executed, the state of the $\overline{MRREN}$ input is loaded into the $\overline{MFF}$ bit location on the first rising edge of clock. This bit is used during register bank read accesses to the $Data_{OUT}$ register when the access code is 11.

The control information of the Control/Status register includes a Wait State field and a Device ID field. The Wait State field is 3 bits in length and provides a programmable means of inserting 0–7 smart wait states in a SmartCAM operation to allow daisychain settling in cascaded systems. The Device ID field is used to hold the unique ID of a particular device in a cascaded array of SmartCAMs. The 10-bit Address field contains the 10-bit address of the CAM entry pointed to by the output of the MRR. Hence, the device ID field and 10-bit address are sufficient to locate a matched entry in an array of SmartCAMs.

The RES field consists of 3 bits that are reserved for internal use. During a read, these bits are returned as zeros.

ALUs and Row Logic

This functional block consists of a General Purpose Logic Block (GPLB), which contains 1024 single-bit ALUs, a set of Response Registers, a Multiple Response Resolver (MRR), and a Match Latch (ML). The three response registers are connected to the input of the GPLB. A block diagram of the ALUs and Row Logic is shown in *Figure 3.*

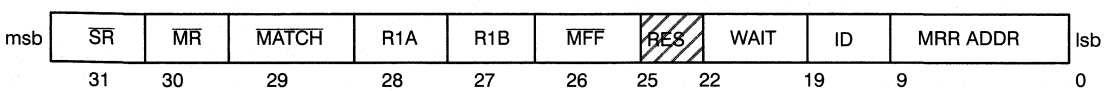


Figure 2. SmartCAM Control/Status Register Format

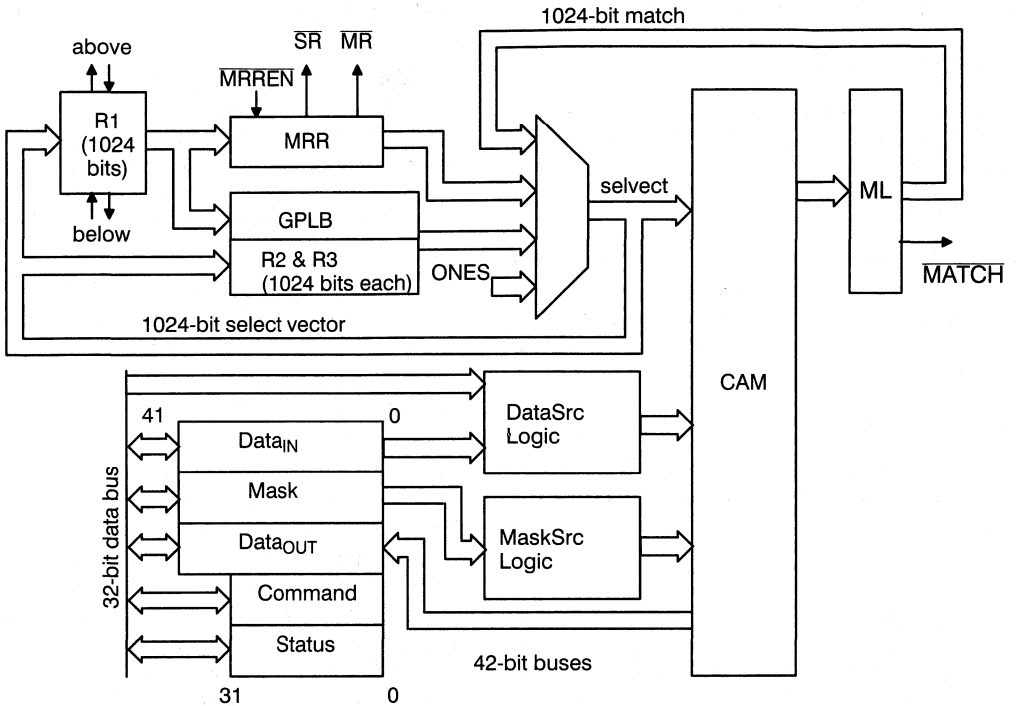


Figure 3. ALUs and Row Logic Block Diagram

GPLB

The GPLB is capable of performing 1 of 256 Boolean operations on the three sets of inputs and stores the result back to one of the three response registers. The Boolean operation is specified by an 8-bit field in the instruction.

Response Register R1

R1 is a 1024-bit register with one bit position associated with each word in the CAM array. This register is multi-functional:

- It can be used to store the result of a Match operation. The contents of R1 are automatically prioritized by the Multiple Response Resolver (MRR) to select the matching CAM entry with the lowest address.
- It can be the source and/or the destination of a Logic operation via the GPLB.
- It is a shift register. Under software control, the contents of R1 can be shifted up or down by one bit position.
- Cascading of SmartCAMs can be achieved by tying the top and bottom bits of R1 to the upper and lower devices, respectively.

R1 cannot be accessed directly by external logic. However, its contents can be modified by the output of the General Purpose Logic Block. R1 is cleared by both the power-on reset and a Reset operation initiated by the $\overline{\text{RESET}}$ input.

Response Registers R2 and R3

R2 and R3 are each 1024-bit registers with one bit position associated with each word in the CAM array. Each register can be used to store the result of a Match operation and act as the source and/or destination of a Logic operation (except multiword operation). R2 and R3 cannot be accessed directly

by external logic. However, their contents can be modified by the output of the General Purpose Logic Block.

Match Latch

The Match Latch is 1024 bits long. It holds the result of the most recent Match operation. This latch cannot be accessed by external logic. The Match Latch is cleared by both the power-on reset and a Reset operation initiated by the $\overline{\text{RESET}}$ input. The contents of the Match Latch are encoded to drive the $\overline{\text{MATCH}}$ output.

Multiple Response Resolver (MRR)

The Multiple Response Resolver prioritizes the match result stored in Response Register R1. Note that when a Match operation occurs, and a match is found in one or more words of the CAM array, the corresponding bit in R1 is set. If more than one match is found, multiple bits in R1 are set. The MRR prioritizes this match result, and provides the following outputs:

- A 1024-bit vector, which points to the match entry with the highest priority (lowest address).
- The Some Response ($\overline{\text{SR}}$) output, which is asserted if at least one bit in R1 is set.
- The Multiple Response ($\overline{\text{MR}}$) output, which is asserted if more than one bit in R1 is set.

The MRR can be disabled by deasserting the $\overline{\text{MRREN}}$ input to the SmartCAM. In this case, the MRR vector will contain all zeros.

State Machine

The state machine is required to decode and execute instructions presented to the device. The state machine is the controller in the SmartCAM processor.

Operation of the SmartCAM

This section describes

- Modes of accessing the CAM array
- Match operations
- The instruction set

Modes of Access

External logic can access the CAM via the 32-bit data port by executing Read or Write operations. Since the SmartCAM has no address decoding logic, the CAM array cannot be accessed in a RAM-like fashion. Instead, four types of select vectors are provided to choose one or more entries in the CAM array. These select vectors are

- Output of the MRR
- Contents of the Match Latch
- Output of the GPLB
- All Ones (selecting all CAM entries).

The last select vector, All Ones, merits further explanation. If the select vector is All Ones, all entries of the CAM will be written to during a Write operation, and "ones" will be written to the response register during a Move operation. Note that a Read operation does not need a select vector, since the output of the MRR is used to access the CAM array.

Match Operations

The SmartCAM supports two types of Match operations, bit match and quad match.

Bit Match

In Bit Match operations, a data pattern, qualified by a mask pattern, is compared with the contents of the CAM array. Bit positions marked by the mask pattern are not compared. The result of the Match operation is stored into one of the three Response registers. The MATCH instruction is used to execute this operation. The instruction is described in detail in the next section.

Quad Match

In Quad Match operations adjacent bits of each 42-bit CAM entry are paired up to form 21 quad values. Since each quad value consists of two bits, it can have four states: Zero (0), One (1), Don't Care (X) and Never Match (N). Values of these four states for CAM encoding are listed in *Table 1*.

Table 1. CAM Quad Encoding

First CAM Bit	Second CAM Bit	Quad Value
0	0	Zero (0)
1	1	One (1)
1	0	Don't Care (X)
0	1	Never Match (N)

The data pattern (to be matched) needs to be qualified by the mask pattern and then encoded. Since there are two data patterns here, the Match quad encoding is done using two bits from the data pattern and two bits from the mask pattern. Values for Match Quad encoding are shown in *Table 2.*

Table 2. Match Quad Encoding

First Bit	Second Bit	Quad Value
Mask Bit = 1 Data Bit = X	Mask Bit = 0 Data Bit = 0	Zero (0)
Mask Bit = 0 Data Bit = 1	Mask Bit = 1 Data Bit = X	One (1)
Mask Bit = 1 Data Bit = X	Mask Bit = 1 Data Bit = X	Don't Care (X)
Mask Bit = 0 Data Bit = 1	Mask Bit = 0 Data Bit = 0	Never Match (N)

To actually perform a Match operation, the first bit in the CAM quad encoding is compared with the first bit in the Match quad encoding. The second bit in the CAM quad encoding is compared with the second bit of the Match quad encoding. The possible results of this comparison are shown in *Table 3.*

As an example, let us take the comparison of a CAM quad value of 0 with a Match quad value of 0. The Match quad value of 0 consists of two mask-data pairs. The first has a mask value of 1 and a data value of X, while the second has a mask value of 0 and a data value of 0. Since the first match quad value has a 1 in the mask bit, the data bit is disqualified from the match process. This means that the first bit of the CAM quad encoding will be a match regardless of its actual value. The second mask-data pair has a value of 0, since the mask bit is 0 and the data bit is also 0. Hence, if the second bit of the CAM quad value contains a 0, a match will occur. In other words, all CAM quad values with the second bit equal to 0 will match the Match quad value having a 0. This corresponds to a CAM quad value of 0 and Don't Care. *Table 3* shows all the possible results obtained when trying to perform a quad match.

The reader may now ask, "Why is a quad match useful?" Consider that the reader is trying to determine whether the CAM contains any values between 3 and 15. The binary values for data between 3 and 15 are shown in *Figure 4.* Note that values 4 through 7 can be encoded as 01XX, while values 8 through 15 can be encoded as 1XXX. Hence, instead of storing 13 values in the CAM, the user ends up storing only three values. Now, when a data pattern of 0101 (corresponding to a decimal value of 5) is presented to the CAM, the CAM can detect that this value is within the range of the second stored value, namely 01XX. Hence, a match will be reported.

The advantage of using this method is twofold. First, more data values can be encoded and stored into the CAM array, which has the advantage of reducing the number of times the SmartCAM is reloaded with data. Second, this method of finding a match is faster than actually finding all values greater than 3, then finding all values less than 15, and then filtering out the intersection of the two sets.

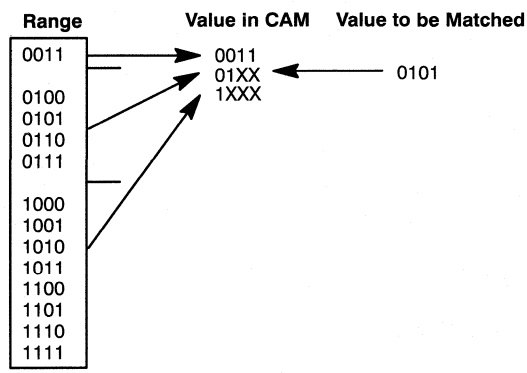


Figure 4. Procedure for Matching Range of Values

Table 3. Quad Match Results

CAM Value \ Match Value			0		1		X		N	
			M=1 D=X	M=0 D=0	M=0 D=1	M=1 D=X	M=1 D=X	M=1 D=X	M=0 D=1	M=0 D=0
0	0	0	Match		No Match		Match		No Match	
1	1	1	No Match		Match		Match		No Match	
X	1	0	Match		Match		Match		Match	
N	0	1	No Match		No Match		No Match		No Match	

The disadvantage of using this method is that the software needs to take care of first encoding the table into quad values, and then actually translating these quad values into values that can be stored into the CAM array. However, in most cases, the advantages will far outweigh the disadvantages.

Finally, it must be noted that the SmartCAM does not have any special hardware or instruction to perform a quad match. The mechanism described above is a software mechanism, and to perform a quad match, the software needs to put the proper encoded values in the CAM array along with the data and mask sources. Once this is accomplished, the bit match matching mechanism is used to perform the match.

SmartCAM Instruction Set

The SmartCAM supports twelve instructions. The instruction word is 32-bits wide. Instructions are presented to the state machine, which decodes and executes the corresponding operations. A brief description of the SmartCAM instructions follows. The instruction arguments are explained in Appendix A.

NOP

The device remains idle during this instruction cycle. All states are unchanged. The $\overline{\text{RDY}}$ output is asserted during the last cycle of the instruction. This instruction is executed in 2 cycles.

SHIFT dir

The contents of R1 are shifted once in the direction specified by *dir*. This instruction is executed in 2 cycles.

SNEXT

The topmost non-zero entry in R1 is reset to zero. The instruction is useful when multiple matches occur and the user needs to sequence through all the match locations and process the match locations/data. This instruction is executed in 2 cycles.

MOVE respreg, selvect [,GPLBterms]

The contents of the response register are replaced by the data pointed to by the *selvect* (select vector). The select vector can be the Match Latch, the MRR output, the GPLB output or All Ones. This instruction is executed in 2 cycles.

READ

The CAM word that the output of the MRR points to is read, and the contents are stored in the Data_{OUT} register. This instruction is executed in 3 cycles.

READSHIFT dir

The CAM word that the output of the MRR points to is read, and the contents are stored in the Data_{OUT} Register. The contents of R1 are then shifted once in the direction specified by *dir*. This instruction is useful in performing non-destructive reads of the CAM array, after a Match operation. This instruction is executed in 3 cycles.

READSNEXT

The CAM word that the output of the MRR points to is read, and the contents are stored in the Data_{OUT} register. The topmost non-zero entry in Response register R1 is reset to 0. This instruction is useful in performing destructive reads of the CAM array, after a Match operation. This instruction is executed in 3 cycles.

WRITE selvect, datasrc, msksrc [, GPLBterms] [, mpat, mpatbit, bitpos]

The contents of the CAM entry selected by selvect are written with the data information specified by *datasrc* (the data source), and mask information specified by *msksrc*. The data source can be the $Data_{IN}$ register, the data lines ($D_{31}-D_0$), all ones, or all zeros. The mask source can be the Mask Register, a MARK or FILL pattern generated internally. The GPLBterms are required if GPLB is selected as the select vector and *mpat*, *mpatbit*, and *bitpos* are required if an internally generated mask pattern is to be used. This instruction is executed in 3 cycles if the *selvect* is MRR, otherwise it takes 9 cycles.

WRITESHIFT selvect, datasrc, msksrc, dir [, GPLBterms] [, mpat, mpatbit, bitpos]

The contents of the CAM entry selected by *selvect* are written with data information specified by *datasrc*, and mask information specified by *msksrc*. After the Write operation is completed, the contents of R1 are shifted once in the direction specified by *dir*. This instruction is executed in 3 cycles if the *selvect* is MRR, otherwise it takes 9 cycles.

WRITESNEXTselvect, datasrc, msksrc [, GPLBterms] [, mpat, mpatbit, bitpos]

The contents of the CAM entry selected by *selvect* are written with data information specified by *datasrc*, and mask information specified by *msksrc*. After the Write is completed, the topmost non-zero entry in R1 is reset to zero. This instruction is executed in 3 cycles if the selvect is MRR, otherwise it takes 9 cycles.

WRITECOL selvect, bitpos [, GPLBterms]

The column of the CAM entry selected by *bitpos* (a number between 0 and 41) is loaded with the contents of *selvect*. This instruction is executed in 3 cycles.

MATCH respreg, datasrc, msksrc [, mpat, mpatbit, bitpos]

Data from the *datasrc* is compared with all CAM words in the array according to the mask source specified by *msksrc*. The result is stored in the Match Latch and the *respreg*. This instruction is executed in 4 cycles.

Error Handling in SmartCAM

An internal error is generated in SmartCAM in two cases: an invalid register bank access or an invalid instruction execution attempt. For both of these errors, nothing is executed, the $D_{31}-D_0$ lines are disabled, and the $\overline{RDY}$ output is never asserted to terminate the bus cycle. Since the $\overline{RDY}$ output is not asserted, the system will hang waiting for a response from the CAM until an external watchdog timeout terminates the access. This error can be cleared by writing to the Control/Status register with the access code (AC1–AC0) set to 00. On doing so, the $\overline{RDY}$ signal is asserted. This register bank access will not alter the contents of the Control/Status register. For more information on error handling, please refer to the application note "SmartCAM Hardware Design Considerations" in this book.

Applications

As mentioned before, the SmartCAM is ideal for search-intensive applications. Database joins and pattern recognition are two very popular applications that fall in this category. The procedure of a database join is illustrated below.

Database Join

Consider a simple relational database system with two tables, Supplier and Project_Part. The Supplier table contains two columns, Supplier# and Part#. The Project_Part contains two columns, Project# and Part#. Assume that the tables are populated with data shown in *Table 4* and *Table 5*.

Any query presented to a database system usually involves the "joining" of the database tables specified in the query. In the above case, the cardinality (number of rows) of each table is 3. If a query specifies "Which supplier supplies parts for which project?", the database system will step through each row in *Table 4*, and compare the Part# values of that table to those of *Table 5* to see if there is a valid match. The pseudocode for this join is shown in *Figure 5*.

Table 4. Supplier Table

Supplier#	Part#
S-00	Pa-00
S-01	Pa-01
S-02	Pa-02

Table 5. Project_Part

Project#	Part#
Pr-0	Pa-00
Pr-1	Pa-01
Pr-2	Pa-02

There are a total of nine comparisons in the above case. Hence, the number of comparisons is directly proportional to the product of the cardinality of each table. Additionally, performing the comparisons in software is a serial operation.

Remember that the SmartCAM performs searches in parallel. Hence, if the user were to load the CAM once, with data from either table, and then match the data from the other table, a match could be obtained in 3 searches. Additionally, since the match is performed in hardware, it is inherently faster than software. The pseudocode in this case will look like that listed in *Figure 6*.

Hence, using the SmartCAM in a database system speeds up the search mechanism considerably.

Image Cleanup

An image cleanup application falls under the category of pattern-recognition applications. In such applications, templates (of binary data) corresponding to patterns to be recognized are created and stored in the SmartCAM along with its identification. Next, the pattern to be recognized is presented to the SmartCAM, and any matching templates are noted.

For an image cleanup demo, let us define that all data less than 5 pixels in size is noise. Hence, when

```
int i,j;
for (i=0;i<Cardinality_supplier;i++)
{
 for (j=0;j<Cardinality_project;j++)
 {
  if (Project.part# == Supplier.Part#)
  print "Project.Project#, Supplier.Supplier#);
 }
}
```

Figure 5. Pseudocode for Conventional Database Join

```
int i;
CamloadData() ;   /* Load Data of table Project_Part into CAM */
for (i=0;i<Cardinality_supplier;i++)
{
 CamMatch(Supplier.Part#);
 while (match_found)     /* A match was found, indicated by SR and MR */
 {
  print "Project.Project#, Supplier.Supplier#);
 }
}
```

Figure 6. Pseudocode for CAM Database Join

binary information of an image is presented to the SmartCAM, all data less than 5 pixels in size needs to be filtered out, and the rest will be output. Obviously, it is much easier (and quicker) to perform such filtering in hardware than in software, and the SmartCAM is ideally suited to this application.

Hardware Support

Cypress Semiconductor has built two boards for demonstrating system design with SmartCAM as well as demonstrating a couple of its applications. The two boards are:

- SBus board that fits into an SBus slot on a SUN Sparcstation
- PC-AT bus board, which can fit into the AT bus slot of a PC.

Application notes explaining the design of these boards, as well as their target applications, are available from Cypress Semiconductor.

Software Support

Cypress Semiconductor is providing the following software tools compatible with the SmartCAM.

- A library of fundamental routines to communicate with the SmartCAM, as well as commonly used routines. The user can embed these routines in their host code and link in the SmartCAM library at compile time.
- Assembler with built-in debugger. The assembler allows users to write their own routines in SmartCAM assembly language, and optimize these routines for their application. The assembler will generate C host code. The debugger is provided to facilitate users to debug their code.
- Simulator. The SmartCAM simulator is a standalone package that allows users to run their SmartCAM code, and have an opportunity to see how the SmartCAM will respond, without actually building a system.
- VHDL, C, and LMC™ models to allow users to simulate the SmartCAM in a system.

Conclusion

The purpose of this application note is to introduce SmartCAM to the reader and to provide a list of its basic features. Additionally, the application note also demonstrates, by means of an example, how SmartCAM is useful in search intensive applications by providing a big performance boost to the user's software.

References

1. Oxford Reference, "Dictionary of Computing", 3rd Edition, Oxford University Press, 1990.

Appendix A. Instruction Arguments

dir. Specifies direction in which the contents of R1 are shifted.

respreg. Specifies the Response register. Can be R1, R2, R3.

selvect. Specifies the select vector. Can be Match Latch, MRR output, GPLB output, or All Ones.

datasrc. The Data Source. Can be the $Data_{IN}$ register, the data bus [D0–D31], All Ones, or All Zeros.

msksrc. The Mask Source. Can be the Mask register, a MARK or FILL pattern, or an Internal Pattern.

GPLBterms. Specify the Boolean operation to be performed on the contents of Response register.

mpat. Specifies the type of internally generated mask pattern, MARK or FILL.

mpatbit. Value of the internally generated mask pattern.

bitpos. Specifies upper bit position for "mpatbits" to be inserted in FILL pattern or the bit position of the "mark" bit in MARK pattern.

Document #: CAM0001–1193

LMC is a trademark of Logic Modeling Corporation
SmartCAM is a trademark of Cypress Semiconductor Corportation.

SmartCAM™ Hardware Design Considerations

Overview

This application note is intended to describe the design considerations that apply to a SmartCAM™ hardware design. The first section, "Timing Considerations," focuses on issues that affect any SmartCAM design. The second section, "Cascading the SmartCAM," discusses the technical aspects of building a SmartCAM array.

Timing Considerations

Using a Watchdog Timer for Error Recovery

The SmartCAM is designed to support applications that require maintaining data integrity in the presence of errors. When an error is detected, the SmartCAM responds to the error by not asserting the ready output ($\overline{RDY}$) to the host system. The effect is to stall the host system bus waiting for assertion of the $\overline{RDY}$ signal to terminate the bus cycle that caused the error.

A watchdog timer circuit is recommended that will either complete or abort the bus cycle and interrupt the host processor to report an error condition. The host processor will then proceed with error recovery processing tailored to the requirements of the application.

The error condition in the SmartCAM is cleared by writing to the Control/Status register with an Access Code of 00 (register access based on chip select).

There are two types of errors the SmartCAM can detect that will cause this behavior:

- An illegal system write (Access Code=11) to the $Data_{IN}$, $Data_{OUT}$, Mask, or Command registers. The Control/Status register is the only legal destination for a system write (AC=11).

  A SmartCAM array is treated as a large CAM with a single set of $Data_{IN}$, $Data_{OUT}$, Mask, and Command registers. As an example, writing data to any $Data_{IN}$ register requires writing the same data to every $Data_{IN}$ register in the array to maintain consistency. A system write (Access Code=11) can only write to one register in a SmartCAM array and is thus illegal to the $Data_{IN}$, $Data_{OUT}$, Mask, or Command registers because the effect is to leave the array in an undefined/illegal state.

  The exception is the Control/Status register (CSR) that is written on an individual basis during initialization. The CSR contains an ID field that uniquely identifies each SmartCAM in an array.

- The second error is attempting to execute an illegal instruction. There are four types of illegal instructions.
  - The first type of illegal instruction occurs when the Access Code (AC) is 01 (execute the command stored in the Command register) in a read bus cycle and the instruction in the Command register is a SHIFT, SNEXT, MOVE, MATCH, WRITE, WRITESHIFT, WRITESNEXT, or WRITECOL. None of these instructions return data to the data bus to satisfy the read bus cycle.
  - The second type happens when the Access Code is 01 for a write cycle and the instruction in the Command register is a READ, READSHIFT, or READSNEXT. The read data cannot be driven onto the data bus during a write cycle.

— The third type is an Access Code of 10 (execute the instruction on the data bus) for any read bus cycle. This combination is illegal because the instruction is supposed to be supplied by the host processor via the data bus.
— The fourth type of illegal instruction is an undefined opcode.

Adding an Idle State when a Write follows a Read

There is a potential problem with data bus latency if a read cycle as shown in *Figure 1* is followed immediately by a write cycle as shown in *Figure 2*.

The SmartCAM output buffers driving read data on the bus may not release the bus before the host processor buffers begin driving write data. The result would be a buffer conflict that will generate noise and could cause the SmartCAM to fail.

The recommended solution is to ensure by design that an idle clock cycle is inserted when a read is followed immediately by a write.

The simplest solution is to ensure the host system software cannot generate a read from the SmartCAM followed by a write in the next clock cycle. This is a software-only solution that does not require any hardware.

For most designs, bus latency will not be a problem. The main reason bus latency is not a problem is the bus timing at the host processor includes set-up and hold times and buffer delays because the processor itself has a common I/O data bus. The bus interface on most host processors is usually designed to add a clock cycle when a write follows a read to allow the buffers to turn around.

The other reason bus latency is not usually a problem is that the host processor may be unable to queue a write to a peripheral before a read from the peripheral completes.

If the host processor can generate back to back a read cycle followed by a write cycle, another alternative is to design a state machine (see *Figure 3*) into the SmartCAM interface that inserts an idle state (adds a wait state) when this condition occurs.

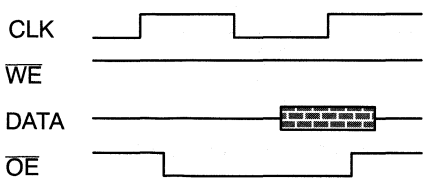


Figure 1. Read a Register

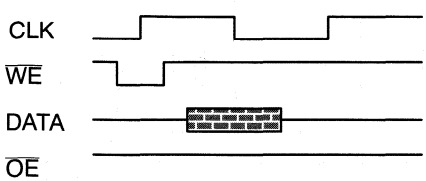


Figure 2. Write a Register

Cascading the SmartCAM

The following section focuses on the design considerations of a SmartCAM array.

The R1 Daisy Chain

R1 is a response register with one bit assigned to each CAM word in the cascaded array. To form the "system R1" register, the individual R1 registers are cascaded together by daisy chain signals as shown in *Figure 4*. There are four daisy chain signals: R1 Above In, R1 Above Out, R1 Below In, and R1 Below Out.

The R1 Above In and R1 Above Out signals connect a SmartCAM R1 to the next higher priority SmartCAM R1. The R1 Below In and R1 Below Out signals connect a SmartCAM R1 to the next lower priority SmartCAM R1.

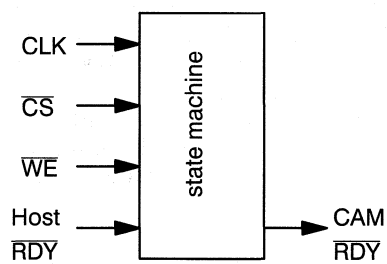


Figure 3. State Machine

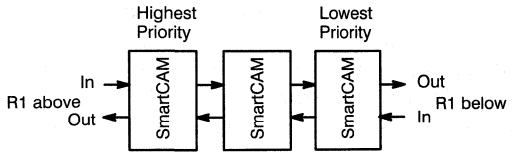


Figure 4. R1 Cascading

The R1 Above In signal of the highest priority CAM and the R1 Below In signal of the lowest priority CAM are usually driven by an external latch under software control. The R1 Above Out signal of the highest priority CAM and the R1 Below Out signal of the lowest priority CAM are usually monitored by the software via an external buffer. This hardware arrangement allows the user to shift in a ONE or ZERO from either direction (UP/DOWN). The user can monitor R1 Below Out and drive R1 Above In under software control. Similarly, the user can monitor R1 Above Out and drive R1 Below In under software control. The effect is to connect the R1 signals of the lowest priority CAM to the R1 signals of the highest priority CAM. This configuration allows the user to recirculate the R1 register during SHIFT, READSHIFT, and WRITESHIFT instructions.

Monitoring the $\overline{\text{RDY}}$ Output

The SmartCAM is controlled by an internal state machine. In a cascaded array, the SmartCAM state machines operate in a synchronous, lockstep fashion.

With the exception of skewing effects, every SmartCAM $\overline{\text{RDY}}$ output should be exactly the same. The skewing effects will be due to clock skew and normal variation in propagation delays from one part to another. The clock skew across a SmartCAM array can be minimized to near zero by incorporating a programmable skew clock buffer such as a CY7B991.

For design purposes, this means the $\overline{\text{RDY}}$ output from any SmartCAM in the array can be used as the $\overline{\text{RDY}}$ output for the entire SmartCAM array.

Generating System Outputs

There are three "system" output signals that can be generated and monitored as shown in *Figure 5*: $\overline{\text{MATCH}}$, $\overline{\text{SR}}$, and $\overline{\text{MR}}$. The match output ($\overline{\text{MATCH}}$) is asserted LOW after a MATCH instruction finds one or more CAM words in the cascaded array that match the search criteria. The some response ($\overline{\text{SR}}$) output is asserted LOW when there is one or more bits set in "system" Response register R1. The multiple response ($\overline{\text{MR}}$) output is asserted LOW when there is more than one bit set in "system" Response register R1. In other words, if the system $\overline{\text{SR}}$ output is active (LOW) and the system $\overline{\text{MR}}$ output is inactive (HIGH), there is only one matching entry (one bit set in R1) in the entire CAM array.

Match ($\overline{\text{MATCH}}$)

The $\overline{\text{MATCH}}$ is an open drain output from the SmartCAM. The individual $\overline{\text{MATCH}}$ outputs are tied together with a pull-up resistor to provide a system $\overline{\text{MATCH}}$ output as shown in *Figure 6*. This works fairly well for "small" systems consisting of 8 or less SmartCAMs.

The minimum value of the pull-up resistor is determined by dividing the required voltage drop by the

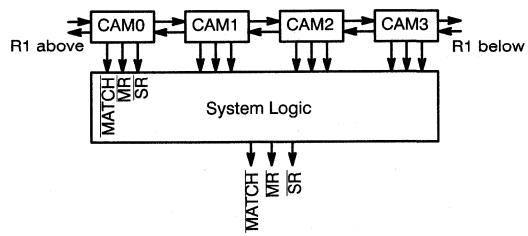


Figure 5. System Outputs

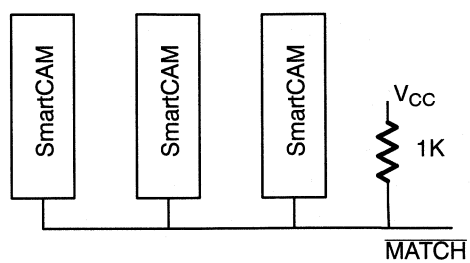


Figure 6. $\overline{\text{MATCH}}$ Output

specified output low current (I_{OL}) of 6 mA. The required voltage drop (4.7V) is the difference between the maximum V_{CC} (5.50V) and the maximum input low voltage V_{IL} (0.8V). The minimum pull-up resistor value in a 5 volt system is therefore 783 ohms.

The next question is how many CAMs can share one minimum value pull-up resistor. The following equation can be used to calculate response time as a function of the number of CAMs, assuming each CAM represents a capacitive load of 10 pF:

$V_{IH} = (V_{CC}-0.2)*(1-e(-t/RC)) + 0.2$

V_{IH} = 2.2V minimum input high voltage

V_{CC} = 4.5V minimum V_{CC}

0.2 is the assumed minimum output low voltage

t = response time in seconds

R = 783 ohms pull-up resistor

C = n*10 pF capacitive loading due to CAMs

Rearranging and solving for response time (t) as a function of the number (n) of CAMs yields:

t = n * 4.9E−9

which is approximately 5 ns per CAM.

CAMs	ns
2	10
4	20
8	40
16	80

It is fairly obvious from these calculations that the $\overline{\text{MATCH}}$ output should drive a schmitt trigger input or equivalent logic as the rise and fall times of the $\overline{\text{MATCH}}$ output can be relatively slow.

Assuming the $\overline{\text{MATCH}}$ output is sampled on the rising edge of the clock as shown in *Figure 7*, the maximum response time to the first clock rising edge after a comparison is 12 ns @50 MHz:

$t\Delta = t_{CYC}+t_{CL}-t_{MV}-t_{SU}$.

t_{CYC} (20 ns) is the clock period

t_{CL} (8 ns) is the min. clock LOW time

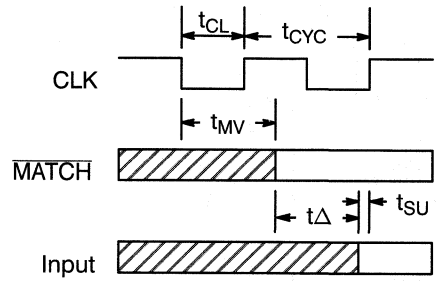


Figure 7. $\overline{\text{MATCH}}$ Timing

t_{MV} (14 ns) is max. $\overline{\text{MATCH}}$ valid delay

t_{SU} (2 ns) is min. set-up time

The number of CAMs that can share a pull-up resistor depends on the operating frequency and when the $\overline{\text{MATCH}}$ output is sampled. At 50 MHz, the recommended logic is one pull-up resistor per four CAMs for a reasonable $\overline{\text{MATCH}}$ response time.

Some Response ($\overline{\text{SR}}$)

The Some Response Output ($\overline{\text{SRO}}$) for a cascaded system is asserted LOW if any $\overline{\text{SR}}$ outputs are asserted by the individual chips in an array. Let's assume an array size of eight CAMs. Then the $\overline{\text{SRO}}$ signal can be generated within a PLD with the following equation (ABEL notation):

```
!SRO = !SR7 # !SR6 # !SR5 # !SR4
     # !SR3 # !SR2 # !SR1 # !SR0;
```

Now suppose the array size is larger than 8 CAMs. As an example, consider an array of 64 CAMs as shown in *Figure 8*. In that case, the $\overline{\text{SRO}}$ output for the cascaded array consists of two layers of logic. The first layer has 64 $\overline{\text{SRn}}$ inputs (n in 0..63) and 8 $\overline{\text{SROm}}$ outputs (m in 0..7). The second logic layer consists of one PLD with 8 inputs and one $\overline{\text{SRO}}$ output. Please note the unused inputs of this PLD will be pulled up to indicate no response.

This approach can be expanded to generate a system Some Response Output ($\overline{\text{SRO}}$) for any size array.

Multiple Response ($\overline{\text{MR}}$)

A system Multiple Response Output ($\overline{\text{MRO}}$) is a little trickier. The system $\overline{\text{MRO}}$ output should be

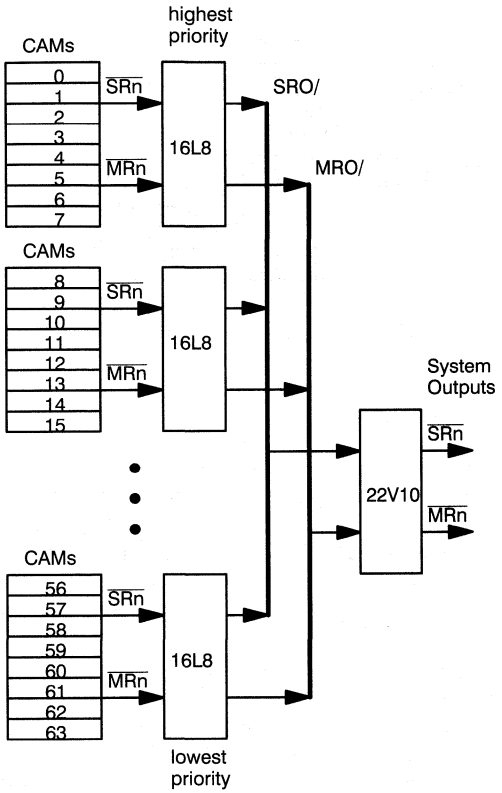


Figure 8. System $\overline{SR}$ and $\overline{MR}$

asserted if any chip $\overline{MR}$ output is active or if two or more chip $\overline{SR}$ outputs are active. Again assuming an array size of 8 CAMs, the $\overline{MRO}$ output can be generated with a PLD as shown with the following equation (ABEL notation):

```
!MRO =
   !( MR0 & MR1 & MR2 & MR3
   & MR4 & MR5 & MR6 & MR7
   & ( SR7 & SR6 & SR5 & SR4 & SR3 & SR2 & SR1
   #  SR7 & SR6 & SR5 & SR4 & SR3 & SR2 & SR0
   #  SR7 & SR6 & SR5 & SR4 & SR3 & SR1 & SR0
   #  SR7 & SR6 & SR5 & SR4 & SR2 & SR1 & SR0
   #  SR7 & SR6 & SR5 & SR3 & SR2 & SR1 & SR0
   #  SR7 & SR6 & SR4 & SR3 & SR2 & SR1 & SR0
   #  SR7 & SR5 & SR4 & SR3 & SR2 & SR1 & SR0
   #  SR6 & SR5 & SR4 & SR3 & SR2 & SR1 & SR0 ) );
```

As long as all of the $\overline{MR}$ inputs are inactive HIGH and at least seven of the eight $\overline{SR}$ inputs are inactive HIGH, the $\overline{MRO}$ output will be inactive HIGH. Gated in this manner, the above function uses one product term per CAM chip.

This approach can also be extended to handle larger arrays when used with the $\overline{SRO}$ logic noted previously. Assume an array size of 64 again. This function can be implemented with two layers of logic. The first layer has 64 $\overline{SRn}$ and 64 $\overline{MRn}$ (n in 0..63) inputs and generates 8 $\overline{MROm}$ outputs (m in 0..7). The second layer has 16 inputs and one system $\overline{MRO}$ output. The unused PLD inputs are pulled inactive HIGH.

Generating System Inputs

There are two "system" input signals in addition to the R1 daisy chain signals described previously. The "system" input signals are the chip selects ($\overline{CS}$) and the Multiple Response Resolver Enable ($\overline{MRREN}$) inputs as shown in *Figure 9*.

Chip Select (CS/)

There are two methods of generating the chip select inputs to a SmartCAM array. The two methods tradeoff decreased number of components versus decreased initialization time.

The first method ties all the SmartCAM chip select inputs to ground (active LOW) and relies on the $\overline{SR}$ to $\overline{MRREN}$ logic to select an individual chip as necessary. This method reduces the number of components required to support the CAM array and increases the initialization time.

During initialization, the user should write a unique 10 bit ID to the Status/Control register of each CAM. With the chip select inputs grounded, an individual CAM is selected for access by the $\overline{SR}$ to $\overline{MRREN}$ logic based on the contents of the "system" R1 Response register. The initialization could be performed as follows:

1. Write all ones to R1 with a MOVE instruction.
2. Write an ID to the Status/Control register.
3. Execute 1024 SNEXT instructions to change the CAM selection pointer to the next CAM
4. If $\overline{SR}$ is active LOW then the last CAM has not been initialized yet. Repeat steps (2) to (4).

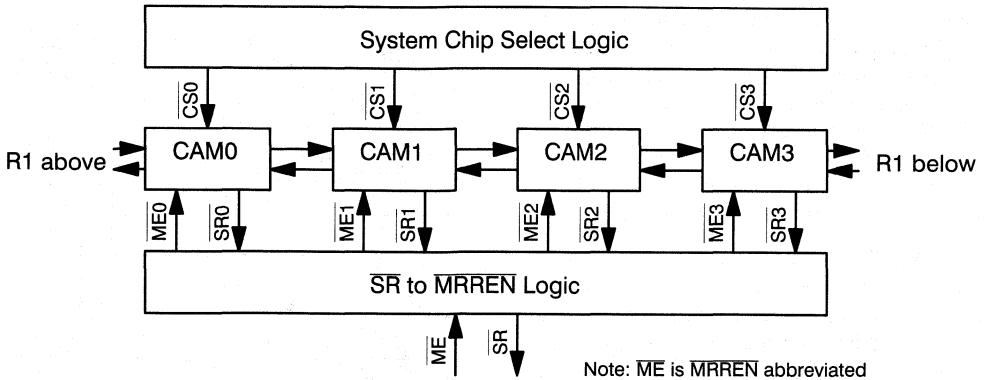


Figure 9. System Inputs

The second method of generating the chip select (Individual Chip Select) adds hardware to reduce the initialization time. The idea is to decode an address supplied by the host and the access code to generate chip selects to the SmartCAM array on an individual basis.

For all Access Codes except 00 (01,10,11), the Chip Select (CS) logic **must** assert the chip select inputs to all CAMs in the array. For Access Code 00, the CS logic decodes the address to decide whether to assert one $\overline{CS}$ input to select one CAM or assert all $\overline{CS}$ inputs to select all of the CAMs for a broadcast write. A typical host address bit assignments for a 16 CAM array might look like the following:

aa s cccc rrr

where "aa" are address bits that drive the Access Code inputs to the array. If the AC is not 00, then all chip selects are asserted. If AC is 00, then the chip selects depend on "s" and "cccc." "s" is an address bit that selects whether this access should assert an individual chip select or all chip selects (e.g., broadcast write). If "s" indicates an individual chip select, the "cccc" bits identify which CAM will get an active LOW chip select. The "rrr" are address bits that drive the Register Select inputs to the array.

A few examples of this addressing scheme for a 16 CAM array are:

- Write to 001xxxx111 would be a broadcast write to the Control/Status register with all chip selects asserted.
- Read from 0001000111 would be a read from CAM8 Control/Status register.
- Read from 11xxxxx111 would be a system read from the Control/Status register. All chip selects are asserted and the CAM with an active $\overline{MRREN}$ input would respond by driving the data bus.

Initializing the array with Individual Chip Select is just a matter of writing to the selected Control/Status registers, once per SmartCAM.

Multiple Response Resolver Enable ($\overline{MRREN}$)

The $\overline{SR}$ to $\overline{MRREN}$ logic selects a CAM based on the contents of the "system" R1 Response register. If there is more than one bit set in R1, the $\overline{SR}$ to $\overline{MRREN}$ logic selects the highest priority CAM. If there are no bits set in R1, the $\overline{SR}$ to $\overline{MRREN}$ logic selects the lowest priority CAM. In a small SmartCAM array with four CAMs, the CAMs could be numbered from CAM0 to CAM3 where CAM0 would be the highest priority CAM. The equations for the $\overline{MRREN}$ outputs in this array would look like the following (ABEL):

```
!MRREN0 = !MRREN & !SR0;
!MRREN1 = !MRREN & !SR1 & SR0;
!MRREN2 = !MRREN & !SR2 & SR1 & SR0;
!MRREN3 = !MRREN &  SR2 & SR1 & SR0;
```

The !MRREN input facilitates building larger arrays. The first equation selects the highest priority CAM (CAM0) when it asserts Some Response ($\overline{SR0}$). The second equation selects the second CAM (CAM1) when it asserts $\overline{SR1}$ and the first CAM deasserts $\overline{SR0}$. The last equation selects the lowest priority CAM (CAM3) when none of the higher priority CAMs assert their $\overline{SR}$ outputs.

The $\overline{SRn}$ to $\overline{MRRENn}$ propagation delay is one of the key timing parameters of a SmartCAM design. This delay determines the number of "Smart" wait states that should be programmed into the Status/ Control register of CAMs in the array to allow the array to stabilize after R1 is altered by an instruction.

Assuming the basic building block is a 16L8, the $\overline{SR}$ to $\overline{MRREN}$ logic for a small system would be as shown in *Figure 10*.

Each CAM provides an $\overline{SR}$ input to the PLD. The PLD provides an $\overline{MRREN}$ input to each CAM. The propagation delay is the same as t_{PD} for the PLD.

Larger systems can be built with ripple carry type logic as shown in *Figure 11*.

Each 16L8 can provide one $\overline{SR}$ output to the next PLD in the carry chain and 7 $\overline{MRREN}$ outputs to CAMs. The ripple carry can be expanded for larger arrays by designing in one PLD per 7 CAMs. The $\overline{SR}$ ripples through the PLDs until a System $\overline{SR}$ is generated. The $\overline{MRREN}$ is applied to each PLD rather than rippling back through the chain.

The equations for all the PLDs except the one driving the System $\overline{SR}$ output are given below (ABEL):

```
!SRO = !SR6 # !SR5 # !SR4 # !SR3 # !SR2 # !SR1 # !SR0;
```

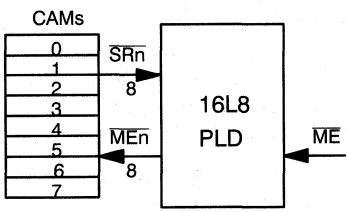


Figure 10. (1–8) CAMs

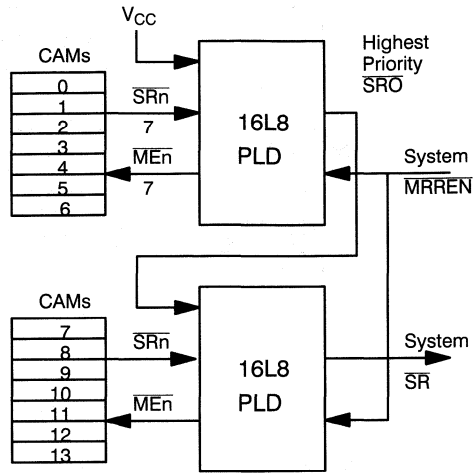


Figure 11. (9–14) CAMs

```
!MRREN0 = !MRREN & !SR0;
!MRREN1 = !MRREN & !SR1 & SR0;
!MRREN2 = !MRREN & !SR2 & SR1 & SR0;
!MRREN3 = !MRREN & !SR3 & SR2 & SR1 & SR0;
!MRREN4 = !MRREN & !SR4 & SR3 & SR2 & SR1 & SR0;
!MRREN5 = !MRREN & !SR5 & SR4 & SR3 & SR2 & SR1
          & SR0;
!MRREN6 = !MRREN & !SR6 & SR5 & SR4 & SR3 & SR2
          & SR1 & SR0;
```

The PLD driving the System $\overline{SR}$ output also has the $\overline{MRREN}$ output to the lowest priority CAM. The MRREN6 equation is therefore slightly different.

```
!MRREN6 = !MRREN & SR5 & SR4 & SR3 & SR2  & SR1
          & SR0;
```

The worst case propagation delay from $\overline{SR}$ to $\overline{MRREN}$ in a ripple carry system is $(n+1)*t_{PD}$ where n is the number of PLDs in the ripple chain and t_{PD} is the propagation delay of the PLD.

A better method for large SmartCAM arrays is equivalent to a carry lookahead as shown in *Figure 12*:

The worst case propagation in this circuit is dependent on the number of layers. The delay will be $(2*n-1)*t_{PD}$ where n is the number of layers. The number of layers required for a CAM array where

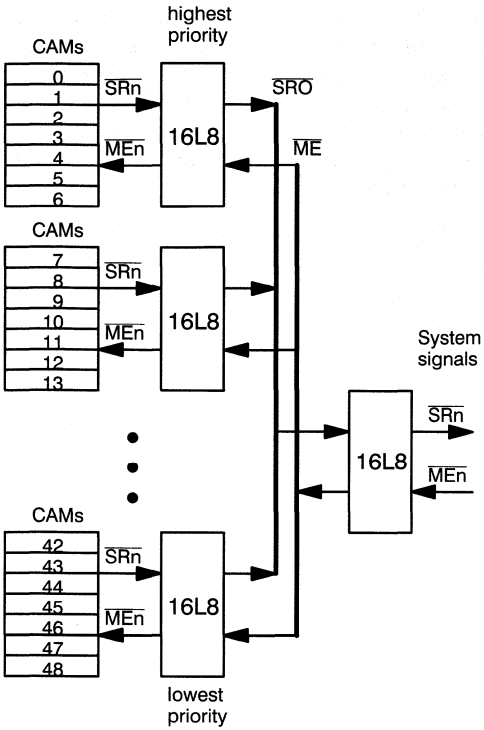


Figure 12. (15 or more) CAMs

x is the number of CAMs is given by n=ln(x)/ln(7). Note the number 7 comes from the limited outputs in each 16L8 PLD where one output drives group $\overline{SR}$ to the next higher layer and the other 7 outputs drive $\overline{MRREN}$ inputs to the next lower layer. A larger PLD with more I/O capability can support more CAMs per PLD.

Initializing a SmartCAM Array

There are two configuration parameters in a SmartCAM that should be written during reset processing: the number of "Smart" wait states and a unique "ID" for each CAM.

Programming "Smart" Wait States

The SmartCAM will add a programmable number of wait states to instructions that alter response register R1 such as SHIFT, SNEXT, etc. The wait states allow the external $\overline{SR}$ to $\overline{MRREN}$ logic to stabilize

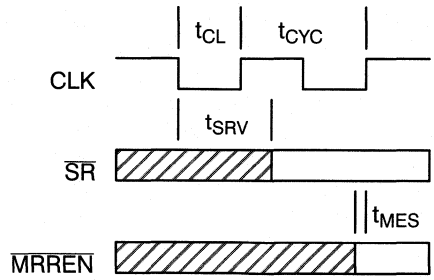


Figure 13. Zero Wait Timing

before another instruction can be executed. This feature is referred to as "Smart" wait states because the CAM only adds wait states when they are necessary for correct operation.

The number of wait states that are needed depend on the external $\overline{SR}$ to $\overline{MRREN}$ logic. As shown in *Figure 13*, the amount of time (t) available for $\overline{SR}$ to $\overline{MRREN}$ propagation with zero wait states is 12 ns @50 MHz:

$t = t_{CYC} + t_{CL} - t_{MES} - t_{SRV}$

t_{CYC} is the clock period

t_{CL} is the minimum clock LOW time

t_{MES} is minimum $\overline{MRREN}$ set-up time

t_{SRV} is maximum $\overline{SR}$ valid delay

If the $\overline{SR}$ to $\overline{MRREN}$ dely exceeds 12 ns, additional wait states must be added. Every wait state adds one clock period (t_{CYC}) to the amount of time allowed for $\overline{SR}$ to $\overline{MRREN}$ logic to propagate.

The number of Smart wait states (0–7) are controlled by a writeable three bit field in the Control/Status register. The number of Smart wait states should be programmed by a broadcast write to all CAMs simultaneously. A broadcast write is performed with all of the chip select inputs asserted and an Access Code of 00.

Assigning Unique IDs

A unique "ID" for each SmartCAM allows the user to easily identify when a match is found which CAM had the match and an offset address that indicates which stored CAM word matched.

The IDs can be assigned such that the highest priority CAM has an ID=0, the next highest priority CAM has an ID=1, etc. until the lowest priority CAM has an ID=(n−1) where n is the number of CAMs. If the IDs are assigned in this manner, the ID and offset address fields in the system (AC=11) Control/Status register form a 20 bit address when a match is found. This address can be used as an index into an external RAM array.

The idea is to store "keys" in the CAM with either data or pointers in corresponding locations in an external RAM. After input data is compared to the stored keys, the user can read the system Status Register (AC=11) to determine if a match was found. If there was a match, the ID and address fields of the Status Register contain a 20 bit index into the data stored in the external RAM.

Document #: CAM0002−1293

Pattern Matching with the SmartCAM™

The architecture of the SmartCAM™ lends itself to a variety of applications that use pattern matching. These applications include optical character recognition, machine vision, target recognition, and video data compression. Pattern matching is well suited to the SmartCAM because the core is comprised of a large content-addressable memory, or CAM. The CAM is a special type of memory device capable of performing searches on all locations simultaneously, instead of serially as a conventional processor must do. Pattern-matching applications tend to use search-based algorithms and the SmartCAM can perform these searches much faster than conventional processors. The SmartCAM contains a total of 1024, 42-bit-wide CAM entries. If more CAM is required, multiple devices can easily be cascaded.

The SmartCAM has several other features that suit it to pattern-matching applications. For example, the general-purpose logic block and Response registers allow arithmetic, logical, and shift manipulation of data or patterns that match a given search criteria. Again, these manipulations occur in parallel to all entries that match the search criteria. Another very useful feature is the ability to mask off selected portions of the CAM word via the Mask register, so that they are ignored during searches. As we'll see, the 42-bit CAM word is typically partitioned into several fields. Some of these fields must be masked off, since they are not part of the search criteria.

This application note illustrates three examples of pattern matching: fast text string searches, optical character recognition using template matching, and motion estimation. The latter application is used widely in emerging video compression standards, including P X 64 and MPEG. The techniques covered can serve as a boiler plate on which the reader can base his or her own specific application.

This application note focuses on the high-level algorithms and data structures required to implement the above mentioned applications. It is left as an exercise to the reader to implement these algorithms using the SmartCAM instruction set. Additionally, the host interface is not specified in any of these examples. The efficiency and speed of the host interface will effect the overall performance of the system. For example, if the algorithm requires the SmartCAM to be repeatedly loaded with different data from external memory, a dedicated DMA engine, along with fast SRAM, will yield higher performance than a microcontroller and 100-ns DRAM.

General knowledge of the features and architecture of this device is a prerequisite to understanding this application note. This can be obtained by reading the datasheet for the SmartCAM and the application note entitled "Introduction to the SmartCAM.

General

While the SmartCAM does have its own instruction set and can intelligently process data, it has not been designed to serve as a system's central control processor. Instead, it is intended to be used as a coprocessor optimized for the execution of search-based algorithms that can be implemented in a SIMD (single instruction multiple data) fashion. Hence, the SmartCAM utilizes a peripheral-style interface and must be fed instructions and data from an outside source. Similarly, the results of SmartCAM data searches and manipulations must be read and interpreted by an external entity. This entity would typically be a general-purpose microprocessor, but could also be a sophisticated sequencer or bit-slice machine. In the examples below, we will call this external entity the host.

Application #1: Text String Searches

Overview

Let's start out with a simple example that illustrates how the SmartCAM could be used to quickly find text strings of interest located within a larger body of text. A potential use of this application is to search through several different magazine articles with a list of key words and phrases that are of interest to the reader. The system could then respond with only the articles that contained the words and phrases given. Additionally, a grading system could be devised, such that articles in which more matches were found would be prioritized. Thus, the reader is saved from having to manually sort through large amounts of information to find the data of interest.

In this application, the host loads a large body of text into the SmartCAM. It then presents the text string(s) of interest as the search criteria to be matched. Part of the information stored in the SmartCAM are pointers that indicate the location of the matched string within the larger body of text.

If the body of text is too large to fit into a single device, two approaches are possible. Multiple SmartCAMs can be cascaded to provide more CAM, or the text can be partitioned so that searches will occur on one section of text at a time. Once a given section of text has been searched, the next section can be loaded and searched. Combining these two techniques should yield the optimum price/performance tradeoff.

CAM Data Structure

The CAM array must be partitioned such that different parts of the array are used for different functions. Since text is typically represented using eight-bit ASCII code, we will allocate eight bits for each alphanumeric character. The first letter in each word stored must start in the far left field. Letters are then stored in successive eight-bit fields. A maximum of five letters can be stored in the 42-bit CAM word with two bits left over. The last bit in the CAM line is defined as the New Word bit. This bit is set if this CAM entry is the first entry for a word, otherwise, the bit is reset. Consecutive CAM entries are used to store longer words. All entries for a word except the first have the New Word bit reset. The last CAM entry for a word also contains a 17-bit Tag field. This field contains a pointer that indicates where within the larger body of text this word is located. The length of this field is arbitrary and can be made longer or shorter for a given application.

Figure 1 shows the data structure for the CAM array for the words paddle, cat, and dictionary. For ease of understanding, we show the actual characters instead of their ASCII representation.

Operation

In operation, the SmartCAM is first loaded with the body of text to be searched, following the data structure described above. One technique for doing this is to execute a MATCH instruction with all data bits masked off. This causes every entry in the CAM to generate a Match and set an internal match pointer pointing to the first entry of the CAM array. Start filling the CAM by writing text data to the $Data_{IN}$ register. Then store the data into the CAM with the WRITESNEXT instruction, which will automatically move the match pointer to the next CAM entry at the end of the Write operation. Continue until all data has been loaded into the SmartCAM. The New Word bit is set for the first entry of each word and reset for all other entries. Along with each word, a tag is entered that corresponds to the word's location or address within the body of text.

After the SmartCAM has been loaded, text searches can begin. The first few letters of the word to be compared are loaded into the $Data_{IN}$ register. Fields within the $Data_{IN}$ register must be aligned with fields within the CAM array. The New Word field is set to one, indicating that this is the start of a new word. This ensures that the SmartCAM ignores all entries that may match the search criteria, but are not the beginning of the word. Any unused fields, including the Tag field (if the word occupies only one CAM entry) are masked off by setting the corresponding bits in the Mask register to 1. Next, the match operation is performed, specifying a destination of R1 for the match status. If the word fits into a single CAM entry, the $\overline{MATCH}$ pin is now monitored to determine whether the query resulted in a match. If so, the $\overline{MR}$ pin is polled to determine

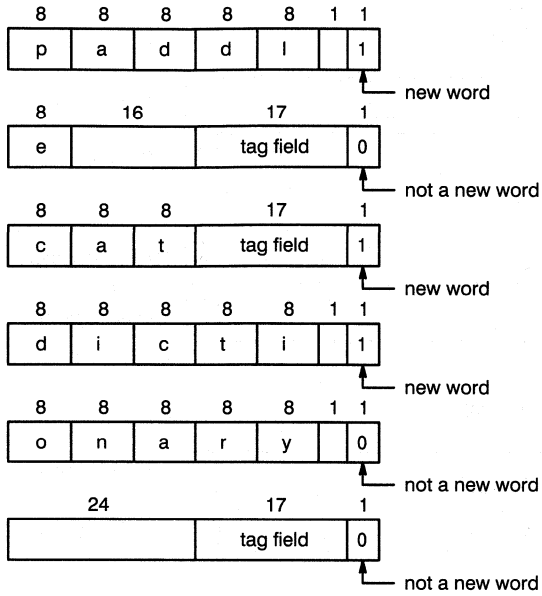


Figure 1. CAM Data Structure for Text String Searches

whether multiple entries matched the query. If at least one match resulted, the READ or READSNEXT commands can be used to load matching entries into the $Data_{OUT}$ register, so that the host can read their pointers and quickly index into the body of text.

If the word is longer than three letters, it must occupy multiple CAM entries within the array. In this case, multiple queries must be used to match the entire word. After the Match operation for the first part of the word, the next one to five letters are loaded into the $Data_{IN}$ register and the New Word field is set to zero to indicate that this is not the beginning of a new word. Again, unused fields are ignored by the Mask register. The Match operation is performed and the ALU is set to logically AND the results of this match with R1 from the CAM entry immediately above the current entry. The results are then stored in R1 of the current CAM entry. Thus, R1 of the current entry will only be set if all fields from the previous entry and all fields from the current entry match. This process continues until all letters in the word have been compared.

Figure 2 shows the values of the $Data_{IN}$ and Mask registers for several words. Notice that, for the word "dictionary," an extra Compare operation is executed with all fields except the New Word field masked off. This is done in order to select the CAM entry containing the Tag field, so that subsequent READ or READSNEXT operations can read it out. Again, after all letters have been compared, the $\overline{MATCH}$ and $\overline{MR}$ pins are monitored to determine the match status.

Matching Text Strings

The above algorithm can be easily extended to match entire strings of text. In this case, the match result of the first Match operation of the second word in the string is ANDed with R1 from the CAM entry above. The result is then stored in R1 of the present CAM entry. In this way, R1 will only be set if the current CAM line is a match and the previous word also matched. This process continues until all words in the string have been compared. The $\overline{MATCH}$ and $\overline{MR}$ pins are polled. If matches exist, the READ or READSNEXT instructions are used to access the Tag field of all matched strings.

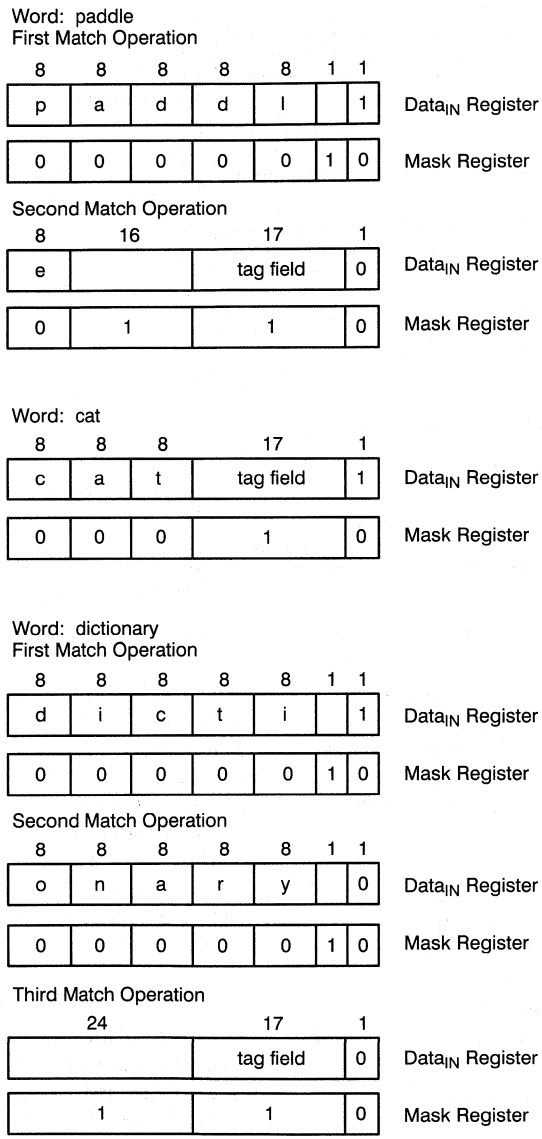
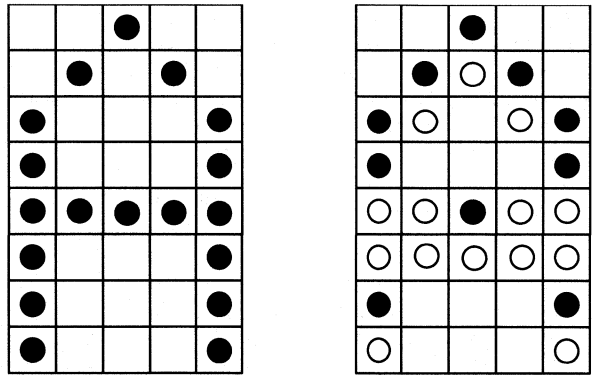


Figure 2. Data$_{IN}$ and Mask Register Values for Several Queries



A B

Character "A" coded with (A), an exact template and (B), a fuzzy template. Filled-in circles and no circles indicate exact matches. Empty circles indicate "don't cares."

Figure 3. SmartCAM Templates

Application #2: Optical Character Recognition

Overview

The SmartCAM can also be used to quickly recognize typed characters on a page, so that they can be coded into ASCII, or some other computer-coded format. To do this, several templates are loaded into the SmartCAM; one for each possible character. Each template has a Tag field containing the ASCII code for the character. The SmartCAM is then presented with a bit map of the character in question. If it matches any of the loaded templates, the Tag field of the matched entry is read out to determine the ASCII code for that character.

In this example, we will assume a 5 by 8 pixel template for each character. An exact template, such as that shown in *Figure 3A* could be used in this operation. However, in the real world, characters scanned from the outside world will seldom look exactly like the original. Therefore, a "fuzzy" template, such as that shown in *Figure 3B* is more appropriate. This template only requires an exact match in areas of the template that uniquely define the character. This template also makes use of "don't care" values in several parts of the template that are not critical to character identification. "Don't cares" will match either a 1 or a 0 value. In this way, scanned characters with slight irregularities will still match the template.

Quad Encoding

Storing "don't care" values in the template requires the SmartCAM's ability to implement Quad Encoding. This mode codes two adjacent bits in the SmartCAM into one of four values: 0, 1, don't care, or never match. Since two bits are required to encode one of four values, each 42-bit CAM line can represent up to 21 quad values. *Table 1* shows the coding for this sequence.

Table 1. CAM Quad Encoding

1st CAM bit	2nd CAM bit	Quad Value
0	0	Zero
1	1	One
1	0	Don't Care
0	1	Never Match

These values are compared against the 21 quad values formed by the contents of the $Data_{IN}$ and Mask registers. Unlike the quad values coded in the CAM array, which use two bits in the same CAM entry to encode one of four possible states, the comparator uses a total of four bits for its encoding: two from the $Data_{IN}$ register and two from the Mask register. *Table 2* illustrates the truth table for the compare word.

In this example, the compare word will be composed of only 1s and 0s. A 1 in the compare word will match a 1 or a "don't care" in the CAM array. A 0 will match a 0 or "don't care" in the CAM array.

Table 2. Compare Word Quad Encoding

1st bit	2nd bit	Quad Value
Mask bit= 1	Mask bit=0	Zero
Data bit=X	Data bit=0	
Mask bit=0	Mask bit=1	One
Data bit=1	Data bit=X	
Mask bit=1	Mask bit=1	Don't Care
Data bit=X	Data bit=X	
Mask bit=0	Mask bit=0	Never Match
Data bit=1	Data bit=0	

SmartCAM Data Structure

Our 5 x 8 pixel templates will require a total of 40 quads of storage within the array. We will also require an 8-bit tag to uniquely identify each ASCII character. Since this data structure occupies a total of three CAM lines for each template, we also need a field identifying which of the three CAM lines we are trying to match. We therefore define the last two bits of each CAM line as the Line Identifier. This data structure is shown in *Figure 4.*

Operation

The operation of this application is very similar to that of the text string search example. First, the SmartCAM is loaded with the various templates, along with their corresponding tags. Since the maximum number of templates that can be uniquely identified by an 8-bit tag is 256 and each template occupies three CAM lines, a maximum of 768 CAM lines are required. Unused lines should be loaded with the binary value 10 in the last two bit fields of the line (the Line Identifier field). This will ensure that these lines are never inadvertently selected for a match.

Once the templates have been initialized inside the SmartCAM, template matching operations can begin. The top half of the character to be matched (i.e., the first 20 pixels) is encoded into the first 40 bits of the $Data_{IN}$ and Mask registers. The last two bits of both registers are loaded with 0s so that only the first line of each template is eligible for a match. (Notice that the SmartCAM allows the user to mix quad and binary encoding on the same line!) Now, the Match operation is performed with the destination of the match result selected as R1. Next, the lower half of the character is encoded into the $Data_{IN}$ and Mask registers. This time, the last two bits of the $Data_{IN}$ register are set to the binary value 01, while the corresponding bits in the Mask register remain 00. This qualifies only the second CAM line of each template for the Match operation. The Match operation is performed and the ALU is set to logically AND the results of this match with R1 from the CAM entry immediately above. The results are then stored in R1 of the current CAM line. At this point, the $\overline{MATCH}$ signal should be monitored. If active, the pattern presented matched at least one of the templates. If this signal is not active, no match was found.

If $\overline{MATCH}$ is asserted, the next step is to read out the Tag field(s) corresponding to the matched tem-

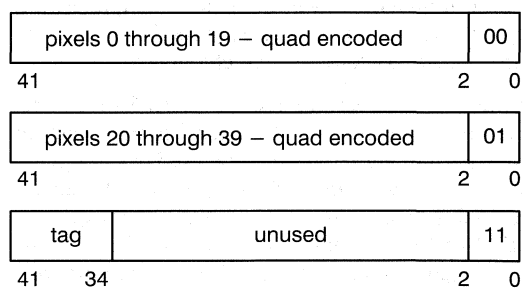


Figure 4. CAM Data Structure for 5x8 Pixel Template

plate(s). To do this, the Line Identifier field of the $Data_{IN}$ register is loaded with 11 to select the third CAM line in the template. The corresponding bits in the Mask register remain 00, but all other bits in this register are set to 1. This masks off all but the last two bits from comparison. The Match operation is again performed with the ALU configured to logically AND the results of this match with R1 from the CAM line immediately above. At this point, the $\overline{MR}$ signal can be polled to determine whether the match is unique or if multiple templates matched the query. Finally, the Read or ReadSnext operations are performed to read out the tag fields for all matched templates.

Application #3: Motion Estimation

Motion estimation is an integral part of video compression algorithms based on the discrete cosine transform, or DCT. This includes emerging compression standards, such as P X 64 and MPEG. Essentially, motion estimation is a technique used to determine whether information from the previous picture frame has moved in the current frame, and if so, where. This allows the encoding of full motion video to be much more efficient than that of still-frame encoding techniques, such as JPEG.

In motion estimation, a small block of pixels from the previous frame is compared to an area surrounding that block in the current frame to determine if that block has moved. The best match between the pixel block and all possible blocks in the area of comparison is determined. Subsequently, a motion vector is encoded into the bitstream. This comparison must be done for each block in the picture frame. Using conventional processing techniques, this would involve billions of calculations. The time required to perform this task would be prohibitive, even with the fastest digital signal processors. However, the SmartCAM can be used to dramatically improve the performance of this application.

In our example, we will use blocks that are 8-by-8 pixels in size and compare them within the surrounding area 16-by-16 pixels in size within the new picture frame. *Figure 5* illustrates this. The algorithm we are using finds the best fit within the area

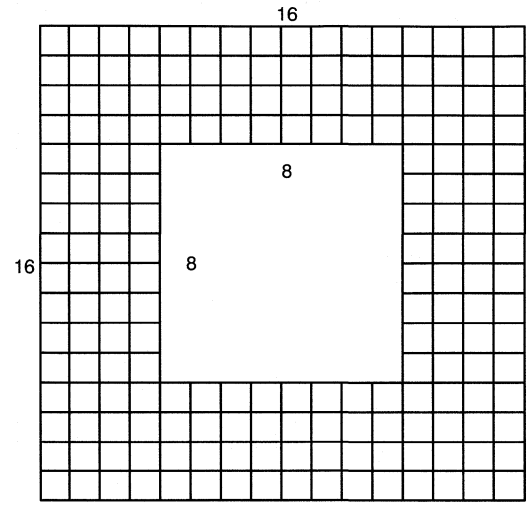


Figure 5. Coding the SmartCAM

of comparison, even for inexact matches. Therefore, if the object has not only moved, but also changed somewhat, a match can still be determined. An example of this would be a bird flying across the screen. Since the bird is flapping its wings, its representation will change slightly from frame to frame.

Now let's take a look at the data structures required to implement this example. We will assume a four-bit value to represent pixel intensity. Therefore, each pixel can take on one of 16 intensity values. The 16-by-16 pixel comparison area is partitioned into multiple, overlapping 8-by-8 pixel blocks. This can be envisioned by starting with the block at the upper left most portion of the area and naming this block 0. Then, shift the 8-by-8 block right by one pixel and call this block 1. Continue shifting right, one pixel at a time until the right edge of the 8-by-8 block meets up with the right edge of the area of comparison. This is block 8. Now return to the left side of the area and shift down by one pixel. This is block 9. Continue this process of moving to the right and down until the 8-by-8 block borders the lower right side of the 16-by-16 area. This is block 80. *Figure 6* illustrates the locations of the four non-overlapping blocks within the area. The job of the SmartCAM is to determine which of these 81 blocks from the

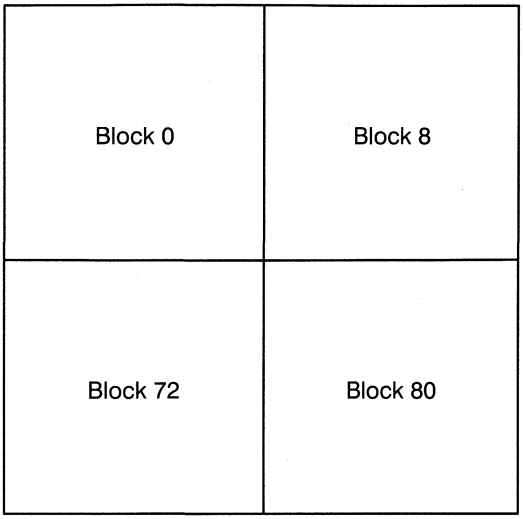


Figure 6. Four Non-Overlapping Blocks within the SmartCAM

count	row	pixels
	0	p7 p6 p5 p4 p3 p2 p1 p0
	1	p15 p14 p13 p12 p11 p10 p9 p8
	2	p23 p22 p21 p20 p19 p18 p17 p16
	3	p31 p30 p29 p28 p27 p26 p25 p24
	4	p39 p38 p37 p36 p35 p34 p33 p32
	5	p47 p46 p45 p44 p43 p42 p41 p40
	6	p55 p54 p53 p52 p51 p50 p49 p48
	7	p63 p62 p61 p60 p59 p58 p57 p56
41	34	31 0

Figure 7. Pixel Block Coding within SmartCAM

current picture frame yields the best match with the 8-by 8-block from the previous frame that was at the center of this 16-by-16 field.

Figure 7 shows how an 8-by-8 pixel block is stored into 8 words of CAM. There are 8 pixels at 4 bits per pixel (32 bits) in each CAM word. The remaining 10 bits are split between a 3-bit row number and a 7-bit counter field. Subsequent CAM entries are coded in a similar fashion, with row 1 containing pixels 8 through 15 and so forth. Note the maximum number of matching pixels for any 8-by-8 pixel block is 64, which can be represented with seven bits.

The 81 pixel blocks are stored in CAM in numerical order with block 0 in the first 8 CAM words, block 1 in the second 8 CAM words, and so on. As a result, the position within the CAM of a given pixel block can be used to calculate a block number (0 to 80).

Each pixel block takes up eight entries in the SmartCAM. Since there are a total of 81 blocks in the 16-by-16 area, the entire area can be easily coded into a single SmartCAM requiring a total of 648 CAM words.

Operation

The motion detection algorithm consists of four parts: load the CAM, count matching pixels per row, calculate matching pixels per block, and identify the block with the largest matching pixel count. The position (address) of the block with the largest matching pixel count can then be divided by eight to calculate the desired block number (0 to 80). Each of these steps is described below.

Load the 16-by-16 Pixel Block into CAM.

The first problem is loading the 16-by-16 pixel block representation (81 8-by-8 pixel blocks) into CAM and initializing the row and the counter fields.

We start by writing zeros to the counter field in the entire array by executing a WRITE instruction with the select vector all ones, the data source all zeros, and a fill mask pattern. The fill pattern has 35 ones (bits 0 to 34) and 7 zeros (bits 35 to 41). This single instruction will clear the counter field for every word in the CAM array. For the first part of the algorithm, we will use the most significant bit of the counter field (bit 41) as a tag bit to exclude the unused portion of the CAM from the pixel counting comparisons. To set that tag bit to one (invalid), a WRITECOL instruction is executed with a select vector of all ones to bit position 41. This will mark every CAM word as invalid initially.

Now we are ready to write the 81 pixel blocks from the current frame into CAM. A MOVE instruction with select vector all ones is executed to set R1 to all

ones. The first eight pixels (32-bits) from block 0 are written into the Lower $Data_{IN}$ register. Row number zero is written into the Upper $Data_{IN}$ register (counter field is zero). A WRITESNEXT instruction is executed next with the MRR (Multiple Response Resolver) as the select vector. This writes the first word in CAM. Note that a count of zero (including bit 41) indicates this word is valid. The next eight pixels from block 0 are written into the Lower $Data_{IN}$ register. The row number is incremented by one modulo eight and written into the Upper $Data_{IN}$ register. And the WRITESNEXT instruction is repeated. This sequence is repeated until all 81 pixel blocks have been stored in the CAM array.

Count Matching Pixels for Each Row.

We need one comparison per pixel (64) to count the number of matching pixels. The algorithm counts the number of matching pixels per row and stores the 4-bit counts in the least significant bits (bits 35 to 38) of the counter field for each row. At the end of this step, we have matching pixel counts for each row.

Let's take a more detailed look at how this works. The first eight pixels in the block from the current frame are loaded into the Lower $Data_{IN}$ register. The Upper $Data_{IN}$ register is loaded with a row count of zero (bits 0 to 2) and a zero in the valid tag (bit 9). 0xfffffff0 is written into the Lower Mask register to enable pixel 0 comparison. 0x1f8 is written into the Upper Mask register. The effect is to limit comparisons to the first pixel of the first row of every valid pixel block in CAM.

The pixel comparison operation is a MATCH instruction that writes the comparison results into R1. At this point, all entries for row zero whose least significant pixel matches the least significant pixel in the $Data_{IN}$ register will have a 1 in R1. Now an ALU operation is performed which adds the value in R1 to the count field. This will cause all blocks that matched pixel zero to have their counts incremented by one. All blocks that did not match will maintain their present count. Note the ALU operation is a sequence of instructions that performs a bit serial addition.

The second pixel in the input pixel block is compared and counted in a similar fashion. The Lower Mask register is set to 0xfffffff0f to enable a pixel one comparison. Then a Match is performed and the matching pixels are counted with an ALU operation. This process is repeated until all eight pixels in the first row have been compared and counted. Then, the next eight pixels are loaded into the Lower $Data_{IN}$ register and a row count is set to 1. Again, after the Match operation, the results are stored in R1 and an ALU operation is performed to increment the count of all entries that matched. The process continues until all 64 pixels have been compared and counted.

Calculate Matching Pixels per Pixel Block.

At this point, we have matching pixel counts for each row and we need to add them together to calculate the number of matching pixels per block. The idea is to use the SmartCAM to perform three parallel additions with bit serial arithmetic.

The first addition adds the 4-bit even and odd row counts together into a 5-bit result stored in the count field of the even rows. The second addition adds rows 0 and 2 into row 0. And adds rows 4 and 6 into row 4. At this point, the result is 6-bits long. The third addition adds rows 0 and 4 together into row 0. This is the final 7-bit result that writes over the valid tag bit.

Identify the Pixel Block with the Largest Count.

Finally, the block with the best fit, or highest number of matching pixels must be selected. This is done by implementing a successive approximation algorithm that iteratively queries the count field with values until the highest count value with at least one match is identified. This is implemented by masking off all fields except the count and row fields. Set the row field to zero to select the row that contains the count, and load the count field with the query value. Now, perform a Match operation and monitor the $\overline{\text{MATCH}}$ signal. Continue querying the SmartCAM with different count values until the largest count value is identified. Then perform a comparison with the largest count value to determine which block(s) from the previous frame had the best fit with the block from the current frame.

A lower threshold should also be established, such that if no stored blocks have more matching pixels than the threshold value, then there are no suitable matches within the area compared. For example, if the threshold is determined to be at least 70 percent, the best match must have at least 0.7 x 64 or 45 matching pixels. Match values below this threshold are ignored.

Summary of Motion Detection

The operation described must be performed for every 8-by-8 pixel block within the picture. In a 640 x 480 display, there are a total of 4,800 8-by-8 pixel blocks. (The algorithm is simplified slightly for blocks around the edge of the picture, but the same principles apply.) The motion detection algorithm can be extended by either sequentially performing the calculations for each of the 4,800 blocks within a single SmartCAM, or by partitioning the problem into multiple SmartCAMs.

For example, with 16 SmartCAMs operating in parallel, each SmartCAM must only process 300 pixel blocks. Since, the SmartCAMs operate on different data, the SmartCAMs would be arranged as 16 separate arrays rather than one single array. With 48 SmartCAMs, each SmartCAM must only process 100 blocks. The number of SmartCAMs required for a particular application is determined by cost constraints, the number of frames processed in a specific period of time, the number of pixel blocks per frame, and the amount of time it takes to process one pixel block.

For example, suppose we want to process a 640 x 480 display at 30 frames per second. Let's also assume that it takes about 110 microseconds (conservative estimate) to process an individual pixel block. At 30 frames per second, each frame must be processed in no more than 33.3 milliseconds. Each SmartCAM can process a total of 300 pixel blocks in that time. To calculate the number of SmartCAMs required to maintain this frame rate, divide the total number of blocks in the picture by the number of blocks that each SmartCAM can process within one frame time and round up (4,800/300). This results in a total of 16 SmartCAMs operating in parallel.

Document #: CAM0003–0193

SmartCAM is a trademark of Cypress Semiconductor Corporation.

Domestic Direct Sales Offices

Corporate Headquarters

Cypress Semiconductor
3901 N. First Street
San Jose, CA 95134
(408) 943–2600
Telex: 821032 CYPRESS SNJ UD
TWX: 910 997 0753
FAX: (408) 943–2741

IC Designs Division
12020–113th Ave. N.E.
Kirkland, WA 98034
(206) 821–9202
FAX: (206) 820–8959

Alabama

Cypress Semiconductor
555 Sparkman Drive, Ste. 1212
Huntsville, AL 35816
(205) 721–9500
FAX: (205) 721–0230

California

Northwest Sales Office
Cypress Semiconductor
100 Century Center Court
Suite 340
San Jose, CA 95112
(408) 943–4867
FAX: (408) 943–6860

Cypress Semiconductor
23586 Calabasas Rd., Ste. 201
Calabasas, CA 91302
(818) 222–3800
FAX: (818) 222–3810

Cypress Semiconductor
2 Venture Plaza, Suite 460
Irvine, CA 92718
(714) 753–5800
FAX: (714) 753–5808

Cypress Semiconductor
12526 High Bluff Dr., Ste. 300
San Diego, CA 92130
(619) 755–1976
FAX: (619) 755–1969

Canada

Cypress Semiconductor
701 Evans Avenue
Suite 312
Toronto, Ontario M9C 1A3
(416) 620–7276
FAX: (416) 620–7279

Colorado

Cypress Semiconductor
4704 Harlan St., Suite 360
Denver, CO 80212
(303) 433–4889
FAX: (303) 433–0398

Florida

Cypress Semiconductor
10014 N. Dale Mabry Hwy. 101
Tampa, FL 33618
(813) 968–1504
FAX: (813) 968–8474

Cypress Semiconductor
255 South Orange Avenue
Suite 1255
Orlando, FL 32801
(407) 422–0734
FAX: (407) 422–1976

Georgia

Cypress Semiconductor
1080 Holcomb Bridge Rd.
Building 100, Ste. 300
Roswell, GA 30076
(404) 998–0491
FAX (404) 998–2172

Illinois

Cypress Semiconductor
1530 E. Dundee Rd., Ste. 190
Palatine, IL 60067
(708) 934–3144
FAX: (708) 934–7364

Maryland

Cypress Semiconductor
8850 Stanford Blvd., Suite 1600
Columbia, MD 21045
(410) 312–2911
FAX: (410) 290–1808

Minnesota

Cypress Semiconductor
14525 Hwy. 7, Ste. 360
Minnetonka, MN 55345
(612) 935–7747
FAX: (612) 935–6982

New Hampshire

Cypress Semiconductor
61 Spit Brook Road, Ste. 110
Nashua, NH 03060
(603) 891–2655
FAX: (603) 891–2676

New Jersey

Cypress Semiconductor
100 Metro Park South
3rd Floor
Laurence Harbor, NJ 08878
(908) 583–9008
FAX (908) 583–8810

New York

Cypress Semiconductor
244 Hooker Ave., Ste. B
Poughkeepsie, NY 12603
(914) 485–6375
FAX: (914) 485–7103

North Carolina

Cypress Semiconductor
7500 Six Forks Rd., Suite G
Raleigh, NC 27615
(919) 870–0880
FAX: (919) 870–0881

Oregon

Cypress Semiconductor
8196 S.W. Hall Blvd. Suite 100
Beaverton, OR 97005
(503) 626–6622
FAX: (503) 626–6688

Pennsylvania

Cypress Semiconductor
Two Neshaminy Interplex, Ste. 206
Trevose, PA 19053
(215) 639–6663
FAX: (215) 639–9024

Texas

Cypress Semiconductor
333 West Campbell Rd., Ste. 240
Richardson, TX 75080
(214) 437–0496
FAX: (214) 644–4839

Cypress Semiconductor
Great Hills Plaza
9600 Great Hills Trail, Ste. 150W
Austin, TX 78759
(512) 502–3023
FAX: (512) 338–0865

Cypress Semiconductor
20405 SH 249, Ste. 216
Houston, TX 77070
(713) 370–0221
FAX: (713) 370–0222

Virginia

Cypress Semiconductor
3151C Anchorway Court
Falls Church, VA 22042
(703) 849–1733
FAX: (703) 849–1734

Domestic Sales Representatives

Alabama

Giesting & Associates
4835 University Square
Suite 15
Huntsville, AL 35816
(205) 830–4554
FAX: (205) 830–4699

Arizona

Thom Luke Sales, Inc.
9700 North 91st St., Suite A–200
Scottsdale, AZ 85258
(602) 451–5400
FAX: (602) 451–0172

California

TAARCOM
451 N. Shoreline Blvd.
Mountain View, CA 94043
(415) 960–1550
FAX: (415) 960–1999

TAARCOM
735 Sunrise Ave., Suite 200–4
Roseville, CA 95661
(916) 782–1776
FAX: (916) 782–1786

Canada

bbd Electronics, Inc.
6685–1 Millcreek Dr.
Mississauga, Ontario L5N 5M5
(416) 821–7800
FAX: (416) 821–4541

bbd Electronics, Inc.
298 Lakeshore Rd., Ste. 203
Pointe Claire, Quebec H9S 4L3
(514) 697–0801
FAX: (514) 697–0277

bbd Electronics, Inc. — Ottawa
(613) 564–0014
FAX: (416) 821–4092

bbd Electronics, Inc. — Winnipeg
(204) 942–2977
FAX: (416) 821–4092

Western Canada

Microwe Electronics Corporation
5330 Wallace Avenue
Delta, British Columbia V4M 1A1
(604) 943–5020
FAX: (604) 943–8184

Connecticut

HLM
3 Pembroke Rd.
Danbury, CT 06810
(203) 791–1878
FAX: (203) 791–1876

Florida

CM Marketing
252 Springs Colony Circle, Unit 382
Altamonte Springs, FL 32714
(407) 682–7709
FAX: (407) 682–7995

CM Marketing
202 West Bearss Avenue
Suite 220
Tampa, FL 33613
(813) 264–9707
FAX: (813) 962–1445

CM Marketing
664 Hollows Circle
Deerfield Beach, FL 33442
(305) 429–8626
FAX: (305) 429–3440

Georgia

Giesting & Associates
2434 Highway 120
Suite 108
Duluth, GA 30136
(404) 476–0025
FAX: (404) 476–2405

Illinois

Micro Sales Inc.
901 W. Hawthorn Drive
Itasca, IL 60143
(708) 285–1000
FAX: (708) 285–1008

Indiana

Technology Mktg. Corp.
1526 East Greyhound Pass
Carmel, IN 46032
(317) 844–8462
FAX: (317) 573–5472

Technology Mktg. Corp.
4630–10 W. Jefferson Blvd.
Ft. Wayne, IN 46804
(219) 432–5553
FAX: (219) 432–5555

Technology Marketing Corp.
1214 Appletree Lane
Kokomo, IN 46902
(317) 459–5152
FAX: (317) 457–3822

Iowa

Midwest Technical Sales
463 Northland Ave., N.E.
Suite 101
Cedar Rapids, IA 52402
(319) 377–1688
FAX: (319) 377–2029

Kansas

Midwest Technical Sales
13 Woodward Dr.
Augusta, KS 67010
(316) 775–2565
FAX: (316) 775–3577

Midwest Technical Sales
15301 W. 87 Parkway, Ste. 200
Lenexa, KS 66219
(913) 888–5100
FAX: (913) 888–1103

Kentucky

Technology Marketing Corp.
718 Amhurst Place
Louisville, KY 40223
(502) 245–7411
FAX: (502) 245–4818

Michigan

Techrep
2200 North Canton Center Rd.
Suite 110
Canton, MI 48187
(313) 981–1950
FAX: (313) 981–2006

Missouri

Midwest Technical Sales
514 Earth City Expwy., #239
Earth City, MO 63045
(314) 298–8787
FAX: (314) 298–9843

Nevada

TAARCOM
735 Sunrise Ave.
Suite 200–4
Roseville, CA 95661
(916) 782–1776
FAX: (916) 782–1786

New Jersey

HLM
333 Littleton Rd.
Parsippany, NJ 07054
(201) 263–1535
FAX: (201) 263–0914

New York

HLM
64 Mariners Lane
P.O. Box 328
Northport, NY 11768
(516) 757–1606
FAX: (516) 757–1636

Reagan/Compar
96 W. Forest Dr.
Rochester, NY 14624
(716) 271–2230
FAX: (716) 381–2840

Reagan/Compar
214 Dorchester Ave., #3C
Syracuse, NY 13203
(315) 432–8232
FAX: (315) 432–8238

Reagan/Compar
3301 Country Club Road
Ste. 2211
P.O. Box 8635
Endwell, NY 13760
(607) 754–2171
FAX: (607) 754–4270

Domestic Sales Representatives (continued)

Ohio

KW Electronic Sales, Inc.
8514 North Main Street
Dayton, OH 45415
(513) 890–2150
FAX: (513) 890–5408

KW Electronic Sales, Inc.
3645 Warrensville Center Rd. #244
Shaker Heights, OH 44122
(216) 491–9177
FAX: (216) 491–9102

Oregon

Northwest Marketing Associates
6975 S. W. Sandburg Rd, Ste. 330
Beaverton, OR 97223
(503) 620–0441
FAX: (503) 684–2541

Pennsylvania

L. D. Lowery
2801 West Chester Pike
Broomall, PA 19008
(215) 356–5300
FAX: (215) 356–8710

KW Electronic Sales, Inc.
4068 Mt. Royal Blvd., Ste. 110
Allison Park, PA 15101
(412) 492–0777
FAX: (412) 492–0780

Puerto Rico

Electronic Technical Sales
P.O. Box 10758
Caparra Heights Station
San Juan, P.R. 00922
(809) 798–1300
FAX: (809) 798–3661

Utah

Sierra Technical Sales
1192 E. Draper Parkway
Suite 103
Draper, UT 84020
(801) 571–8195
FAX: (801) 571–8194

Washington

Northwest Marketing Associates
12835 Bellevue-Redmond, Ste. 330N
Bellevue, WA 98005
(206) 455–5846
FAX: (206) 451–1130

Wisconsin

Micro Sales Inc.
210 Regency Court
Suite L101
Waukesha, WI 53186
(414) 786–1403
FAX: (414) 786–1813

International Direct Sales Offices

Cypress Semiconductor International—Europe

Avenue Ernest Solvay, 7
B–1310 La Hulpe, Belgium
Tel: (32) 2–652–0270
Telex: 64677 CYPINT B
FAX: (32) 2–652–1504

France

Cypress Semiconductor France
Miniparc Bât. no 8
Avenue des Andes, 6
Z.A. de Courtaboeuf
91952 Les Ulis Cedex, France
Tel: (33) 1–69–07–55–46
FAX: (33) 1–69–07–55–71

Germany

Cypress Semiconductor GmbH
Munchner Str. 15A
W-8011, Zorneding, Germany
Tel: (49) 81–06–2855
FAX: (49) 81–06–20087

Cypress Semiconductor GmbH
Büro Nord
Matthias-Claudius-Str. 17
W–2359 Henstedt-Ulzburg, Germany
Tel: (49) 4193-77217
FAX: (49) 4193-78259

Italy

Cypress Semiconductor
Via del Poggio Laurentino 118
00144 Rome, Italy
Tel: (39) 65–920–723
FAX: (39) 69–496–888

Cypress Semiconductor
Interporto di Torino
Proma Strada n. 5/B
10043 Orbassano, Italy
Tel: (39) 11–397–57–98
or (39) 11–397–57–57
FAX: (39) 11–397–58–10

Japan

Cypress Semiconductor Japan K.K.
Fuchu-Minami Bldg., 2F
10-3, 1-Chome, Fuchu-machi,
Fuchu-shi, Tokyo, Japan 183
Tel: (81) 423–69–82–11
FAX: (81) 423–69–82–10

Sweden

Cypress Semiconductor
Scandinavia AB
Täby Centrum, Ingang S
S–18311 Täby, Sweden
Tel: (46) 8 638 0100
FAX: (46) 8 792 1560

United Kingdom

Cypress Semiconductor U.K., Ltd.
3, Blackhorse Lane, Hitchin,
Hertfordshire, U.K., SG4 9EE
Tel: (44) 462–42–05–66
FAX: (44) 462–42–19–69

Cypress Semiconductor Manchester
27 Saville Rd. Cheadle
Gatley, Cheshire, U.K.
Tel: (44) 614–28–22–08
FAX: (44) 614–28–0746

International Sales Representatives

Australia

Braemac Pty. Ltd.
Unit 6, 111 Moore St.
Leichhardt, N.S.W. 2040, Australia
Tel: (61) 2–564–1211
FAX: (61) 2–564–2789

Braemac Pty. Ltd.
10–12 Prospect Street, Box Hill
Melbourne, Victoria, 3128, Australia
Tel: (61) 3–899–1272
FAX: (61) 3–899–1276

Austria

Hitronik Vertriebsge GmbH
St. Veitgasse 51
A–1130 Wien, Austria
Tel: (43) 1–877–4199
Telex: 133404 HIT A
FAX: (43) 1–876–55–72

Belgium

Sonetech
Limburg Stirum 243
1810 Wemmel, Limburg
Tel: (32) 2–460–0707
FAX: (32) 2–460–1200

Denmark

ScandComp Denmark A/S
Rentemestervej 69A
DK–2400 Copenhagen NV
Denmark
Tel: (45) 31 19 44 33
FAX: (45) 31 10 12 87

Finland

ScandComp Finland OY
Asemakuja 2
SF–02 770 ESPOO
Finland
Tel: 358–0–859–3099
FAX: 358–0–859–2120

France

Arrow Electronics
73/79, Rue des Solets
Silic 585
94653 Rungis Cedex
Tel: (33) 1 49 78 49 00
FAX: (33) 1 49 78 05 99

Arrow Electronics
Les Jardins d'Entreprises
Betiment B3
213, Rue Gerland
69007 Lyon
Tel: (33) 78 72 79 42
FAX: (33) 78 72 80 24

Arrow Electronics
Centreda
Avenue Didier Daurat
31700 Blagnac
Tel: (33) 61 15 75 18
FAX: (33) 61 30 01 93

Arrow Electronics
Immeuble St. Christophe
Rue de la Frebardiere
Zi Sud Est
35135 Chantepie
Tel: (33) 99 41 70 44
FAX: (33) 99 50 11 28

Newtek
Rue de L'Esterel, 8, Silic 583
F–94663 Rungis Cedex, France
Tel: (33) 1–46–87–22–00
Telex: 263046 F
FAX: (33) 1–46–87–80–49

Newtek
Rue de l'Europe, 4
Zac Font–Ratel
38640 Claix, France
Tel: (33) 16–76–98–56–01
FAX: (33) 16–76–98–16–04

Scaib, SA
80 Rue d'Arcueil Silic 137
9 4523 Rungis, Cedex, France
Tel: (33) 1–46–87–23–13
FAX: (33) 1–45–60–55–49

Germany

API Electronik GmbH
Lorenz–Brarenstr 32
W–8062 Markt, Indersdorf
Germany
Tel: (49) 8136 7092
Telex: 527 0505
FAX: (49) 8136 7398

International Sales Representatives (continued)

Metronik GmbH
Leonhardsweg 2, Postfach 1328
W–8025 Unterhaching,
Germany
Tel: (49) 89 611080
Telex: 17 897434 METRO D
FAX: (49) 89 6116468

Metronik GmbH
Laufamholzstrasse 118
W–8500 Nürnberg,
Germany
Tel: (49) 911 544966
Telex: 6 26 205
FAX: (49) 911 542936

Metronik GmbH
Löewenstrasse 37
W–7000 Stuttgart 70
Germany
Tel: (49) 711 764033
Telex: 7–255–228
FAX: (49) 711 7655181

Metronik GmbH
Siemensstrasse 4–6
W–6805 Heddesheim, Germany
Tel: (49) 6203 4701
Telex: 465 035
FAX: (49) 6203 45543

Metronik GmbH
Zum Lonnenhohl 38
W–4600 Dortmund 13, Germany
Tel: (49) 231 217041
FAX: (49) 231 210799

Metronik Halle
Thalmannplatz 16/0904
O–4020 Halle, Germany

SASCO GmbH
Hermann-Oberth-Str. 16
8011 Putzbrunn, Germany
Tel: (089) 4611–211
Telex: 529 504 sasco d
FAX: (089) 4611–271

SASCO GmbH
Gibitzenhofstr. 62
8500 Nurnberg 70, Germany
Tel: (0911) 42 10 65
Telex: 623097
FAX: (0911) 42 57 94

SASCO GmbH
Stafflenbergstr. 24
7000 Stuttgart 1, Germany
Tel: (0711) 24 45 21
Telex: 723936
FAX: (0711) 23 39 63

SASCO GmbH
Am Gansacker 26
7801 Umkirch bei Freiburg
Tel: (07665) 70 18
Telex: 7722945
FAX: (07665) 87 78

SASCO GmbH
Hainer Weg 48
6000 Frankfurt 70, Germany
Tel: (069) 61 03 91
Telex: 414435
FAX: (069) 61 88 24

SASCO GmbH
Beratgerstr. 36
4600 Dortmund 1, Germany
Tel: (0231) 17 97 91
Telex: 8227826
FAX: (0231) 17 29 91

SASCO GmbH
Am Uhrturm 7
3000 Hannover 81, Germany
Tel: (0511) 83 90 20
Telex: 921123
FAX: (0511) 8 43 76 18

SASCO GmbH
Europaallee 3
2000 Norderstedt, Germany
Tel: (040) 5 23 20 13
Telex: 2165623
FAX: (040) 5 23 23 78

Hong Kong

Tekcomp Electronics, Ltd.
913–4 Bank Centre
636, Nathan Road, Mongkok
Kowloon, Hong Kong
Tel: (852) 3–880–629
Telex: 38513 TEKHL
FAX: (852) 7–805-871

India

Spectra Innovations Inc.
Manipal Centre, Unit No. S–822
47, Dickenson Rd.
Bangalore–560,042
Karnataka, India
Tel: 80–588–323
Telex: 845 2696 or 8055
(Attn: ICTP–705)
FAX: 80–586–872

Israel

Talviton Electronics
P.O. Box 21104, 9 Biltmore Street
Tel Aviv 61 210, Israel
Tel: (972) 3–544–2430
Telex: 33400 VITKO
FAX: (972) 3–544–2085

Italy

Dott. Ing. Guiseppe De Mico s.p.a.
V. Le Vittorio Veneto, 8
I–20060 Cassina d'Pechi
Milano, Italy
Tel: (39) 29–53–43–600
Telex: 330869 DEMICO I
FAX: (39) 29–52–19–12

Silverstar Ltd. SPA
Viale Fulvio Testi, 280
20126 Milano, Italy
Tel: (39) 2 661251
Teles: 33 2189 SIL 7I
FAX: (39) 2 66101359

Japan

Tomen Electronics Corp.
2-1-1 Uchisaiwai-Cho, Chiyoda-Ku
Tokyo, 100 Japan
Tel: (81) 3–3506–3673
Telex: 23548 TMELCA
FAX: (81) 3–3506–3497

Inno Micor Corporation
KM Dai-ichi, Bldg. 2-13-13, Shinyokohama
Kouhoku-ku, Yokohama-shi
Kanagawa, Japan 222
Tel: (81) 45–476–7502
FAX: (81) 45–476–7516

Fuji Electronics Co., Ltd.
Ochanomizu Center Bldg.
3–2–12 Hongo, Bunkyo–Ku
Tokyo, 113 Japan
Tel: (81) 3–3814–1411
Telex: J28603 FUJITRON
FAX: (81) 3–3814–1414

Ryoyo Electro Corporation
Knowa Bldg., 1–12–22 Tsukiji,
Chuo–ku, Tokyo 104 Japan
Tel: (81) 3–5565–1531
FAX: (81) 3–5565–1546

Korea

Logicom Inc.
1634–9 Bongchun-Dong
Kwanak–ku
Seoul, Korea 151–061
Tel: (822) 888–2858
FAX: (822) 888–7040

superCHIP Inc.
6th Floor, KyungJin Bldg. 161–17
Sansung-dong, Kangnam-ku
Seoul, Korea
Tel: (822) 558–5559
FAX: (822) 558–1875

International Sales Representatives (continued)

Netherlands

Sonetech B.V.
Gulberg 33, NL–5674
Te Nuenen
The Netherlands
Tel: (31) 40–83–70–75
Telex: 59418 INTRA NL
FAX: (31) 40–83–23–00

Norway

ScandComp Norway A/S
Aslakveien 20 F
N–0753 OSLO 7
Norway
Tel: (47) 22 50 06 50
FAX: (47) 22 50 27 77

Singapore

Electec PTE Ltd.
Block 50, Kallang Bahru
#04–21, Singapore 1233
Tel: (65) 294–8389
FAX: (65) 294–7623

Spain

ATD Electronica
Avda. de la Industria No. 32
Nave 17, 2B, 28100 Alcobendas
Madrid, Spain
Tel: (34) 1–66–16–551
FAX: (34) 1–66–16–300

Sweden

ScandComp Sweden AB
Box 8303 Domnarvsgatan 33
S–163 08 SPANGA
Sweden
Tel: (46) 8–761–73–00
FAX: (46) 8–760–46–69

Switzerland

Basix für Elektronik A. G.
Hardturmstrasse 181
CH–8010 Zurich, Switzerland
Tel: (41) 1–276–11–11
Telex: 822762 BAEZ CH
FAX: (41) 1–276–14–48

Taiwan R.O.C.

Prospect Technology Corp.
5F, No. 348, Section 7
Cheng-Teh Rd.
Taipei, Taiwan
Tel: (886) 2–820–5353
Telex: 14391 PROSTECH
FAX: (886) 2–820–5731

United Kingdom

Ambar Components Ltd.
17 Thame Park Road
Thame, Oxfordshire
England, OX9 3XD
Tel: (44) 844–26–11–44
Telex: 837427
FAX: (44) 844–26–17–89

Arrow Electronics (UK) Ltd.
St. martins Business Centre
Cambridge Road
Bedford MK42 OLF, U.K.
Tel: (44) 234 270272
FAX: (44) 234 214674

Pronto Electronic System Ltd.
City Gate House
Eastern Avenue, 399–425
Gants Hill, Ilford,
Essex, U. K. IG2 6LR
Tel: (44) 81–554–62–22
Telex: 8954213 PRONTO G
FAX: (44) 81–518–32–22

Distributors

Alliance Electronics:

Scottsdale, AZ 85254
1-800-608-9494
FAX: (602) 443-3898

Santa Clarita, CA 91350
(805) 297-6204
FAX: (805) 297-6205

Shelton, CT 06484
(203) 926-0087
FAX: (203) 926-1850

Tampa, FL 33611
(813) 831-7972
FAX: (813) 831-8297

Vernon Hills, IL 60061
(708) 949-9890
FAX: (708) 949-1162

Boston, MA 02110
1-800-854-2393
FAX: (617) 261-7987

Winchester, MA 01890
1-800-888-6627
FAX: (617) 756-1226

Albuquerque, NM 87123
1-800-955-3360
FAX: (505) 275-6392

Binghamton, NY 13901
(607) 648-8833

Huntington, NY 11743
(516) 673-1930
FAX: (516) 673-1934

Dayton, OH 45459
(513) 433-7700
FAX: (513) 433-3147

Carrolton, TX 75007
(214) 492-6700
FAX: (214) 492-5474

Anthem Electronics, Inc.:

Tempe, AZ 85281
(602) 966-6600

Chatsworth, CA 91311
(818) 775-1333

East Irvine, CA 92718
(714) 768-4444

Rocklin, CA 95677
(916) 624-9744

San Jose, CA 95131
(408) 453-1200

San Diego, CA 92121
(619) 453-9005

Englewood, CO 80112
(303) 790-4500

Waterbury, CT 06705
(203) 575-1575

Altamonte Springs, FL 32701
(407) 831-0007

Schaumburg, IL 60173
(708) 884-0200

Wilmington, MA 01887
(508) 657-5170

Columbia, MD 21046
(301) 995-6640

Eden Prairie, MN 55344
(612) 944-5454

Pine Brook, NJ 07058
(201) 227-7960

Commack, NY 11725
(516) 864-6600

Beaverton, OR 97005
(503) 643-1114

Horsham, PA 19044
(215) 443-5150

Richardson, TX 75081
(214) 238-7100

Salt Lake City, UT 84119
(801) 973-8555

Bothel, WA 98011
(206) 483-1700

Arrow Electronics:

Alabama
Huntsville, AL 35816
(205) 837-6955

Arizona
Tempe, AZ 85282
(602) 431-0030

California
Calabasas, CA 91302
(818) 880-9686

San Diego, CA 92123
(619) 565-4800

San Jose, CA 95131
(408) 441-9700

San Jose, CA 95134

Tustin, CA 92680
(714) 587-0404

Canada
Mississauga, Ontario L5T 1MA
(416) 670-7769

Dorval, Quebec H9P 2T5
(514) 421-7411

Neapean, Ontario K2E 7W5
(613) 226-6903

Quebec City, Quebec G2E 5RN
(418) 871-7500

Burnaby, British Columbia V5A 4T8
(604) 421-2333

Colorado
Englewood, CO 80112
(303) 799-0258

Connecticut
Wallingford, CT 06492
(203) 265-7741

Florida
Deerfield Beach, FL 33441
(305) 429-8200

Florida (continued)
Lake Mary, FL 32746
(407) 333-9300

Georgia
Deluth, GA 30071
(404) 497-1300

Illinois
Itasca, IL 60143
(708) 250-0500

Indiana
Indianapolis, IN 46268
(317) 299-2071

Kansas
Lenexa, KS 66214
(913) 541-9542

Distributors (continued)

Arrow Electronics: (cont.)

Maryland
Columbia, MD 21046
(410) 596–7800

Gathersburg, MD
(301) 596–7800

Massachusetts
Wilmington, MA 01887
(617) 658–0900

Michigan
Livonia, MI 48152
(313) 462–2290

Minnesota
Eden Prairie, MS 55344
(612) 941–5280

Missouri
St. Louis, MO 63146
(314) 567–6888

New Jersey
Marlton, NJ 08053
(609) 596–8000

Pinebrook, NJ 07058
(201) 227–7880

New York
Rochester, NY 14623
(716) 427–0300

Hauppauge, NY 11788
(516) 231–1000

North Carolina
Raleigh, NC 27604
(919) 876–3132

Ohio
Centerville, OH 45458
(513) 435–5563

Solon, OH 44139
(216) 248–3990

Oklahoma
Tulsa, OK 74146
(918) 252–7537

Oregon
Beaverton, OR 97006–7312
(503) 629–8090

Pennsylvania
Pittsburgh, PA 15238
(412) 963–6807

Texas
Austin, TX 78758
(512) 835–4180

Carrollton, TX 75006
(214) 380–6464

Houston, TX 77099
(713) 530–4700

Washington
Bellevue, WA 98007
(206) 643–9992

Spokane, WA 99206–6606
(509) 924–9500

Wisconsin
Brookfield, WI 53045
(414) 792–0150

Distributors (continued)

Marshall Industries:

Alabama
Huntsville, AL 35801
(205) 881–9235

Arizona
Phoenix, AZ 85044
(602) 496–0290

California
Marshall Industries, Corp. Headquarters
El Monte, CA 91731–3004
(818) 307–6000

Irvine, CA 92718
(714) 458–5301

Calabasas, CA 91302
(818) 878–7000

Rancho Cordova, CA 95670
(916) 635–9700

San Diego, CA 92123
(619) 627–4140

Milpitas, CA 95035
(408) 942–4600

Canada
Brampton, Ontario L6T 5G3
(416) 458–8046

Ottawa, Ontario
(613) 564–0166

Pointe Claire, Quebec H9R 5P9
(514) 694–8142

Colorado
Thornton, CO 80241
(303) 451–8383

Connecticut
Wallingford, CT 06492–0200
(203) 265–3822

Florida
Ft. Lauderdale, FL 33309
(305) 977–4880

Florida (continued)
Altamonte Springs, FL 32701
(407) 767–8585

St. Petersburg, FL 33716
(813) 573–1399

Georgia
Norcross, GA 30093
(404) 923–5750

Illinois
Schaumbrug, IL 60173
(708) 490–0155

Indiana
Indianapolis, IN 46278
(317) 297–0483

Kansas
Lenexa, KS 66214
(913) 492–3121

Maryland
Silver Springs, MD 20904
(301) 622–1118

Massachusetts
Wilmington, MA 01887
(508) 658–0810

Michigan
Livonia, MI 48150
(313) 525–5850

Minnesota
Plymouth, MN 55447
(612) 559–2211

Missouri
Bridgeton, MO 63044
(314) 291–4650

New Jersey
Fairfield, NJ 07006
(201) 882–0320

Mt. Laurel, NJ 08054
(609) 234–9100

New York
Endicott, NY 13760
(607) 785–2345

Hauppage, NY 11788
(516) 273–2695

Rochester, NY 14624
(716) 235–7620

North Carolina
Raleigh, NC 27604
(919) 878–9882

Ohio
Solon, OH 44139
(216) 248–1788

Dayton, OH 45414
(513) 898–4480

Oregon
Beaverton, OR 97005
(503) 644–5050

Pennsylvania
Mt. Laurel, NJ 08054
(609) 234–9100

Texas
Austin, TX 78754
(512) 837–1991

Richardson, TX 75081
(214) 705–0600

Houston, TX 77043
(713) 467–1666

Utah
Salt Lake City, UT 84119
(801) 973–2288

Washington
Bothell, WA 98011
(206) 486–5747

Wisconsin
Waukesha, WI 53186
(414) 797–8400

Distributors (continued)

Semad:

Calgary
Calgary, Alberta T2H 2S8
(403) 252–5664
FAX: (800) 565–9779

Montreal
Pointe Claire, Quebec H9R 427
(514) 694–0860
1–800–361–6558
FAX: (514) 694–0965

Ottawa
Ottawa, Ontario K1B 1A7
(613) 526–4866
FAX: (613) 523–4372

Toronto
Markham, Ontario L3R 4Z4
(416) 475–3922
FAX: (416) 475–4158

Vancouver
Burnaby, British Columbia V5G 4M1
(604) 451–3444
1–800–663–8956
FAX: (604) 451–3445

Zeus Electronics:

Yorba Linda, CA 92686
(714) 921–9000

San Jose, CA 95131
(408) 629–4789

Lake Mary, FL 32746
(407) 333-3055

Wilmington, MA 01887
(508) 658–4776

Port Chester, NY 10573
(914) 937–7400

Carrollton, TX 75006
(214) 380–4330

Glossary

8B/10B code: A patented coding method that converts "raw-data" to a form more suitable for transmission over a high speed serial interconnect link. This particular code insures a high transition density with a perfect DC balance.

Abel-HDL: Proprietary Hardware Description Language (HDL) from Data I/O Corp. Created as text input design language for their PLD/FPGA development software.

access code: The access code is two signals that indicate to a SmartCAM what type of bus cycle when the chip select input is active LOW.

AHDL (Advanced Hardware Description Language): A high-level, modular language used to create logic designs for MAX EPLDs.

Alias SYNC: An unintentional SYNC character that occurs when transmission errors corrupt the serial data stream. It is possible to create a bit pattern that matches the SYNC character, but which is not correctly aligned with the serial byte boundaries. This Alias SYNC can make it impossible to correctly recover the data.

AND: The and logic gate.

ANSI X3T9.3: The name of a data communication standard sponsored by the American National Standards Institute. X3T9.3 is the specific standard describing Fibre Channel.

arbitration: The process of deciding which one of two or more competing entities will be allocated a resource.

architecture: As pertains to VHDL logic synthesis, the declaration that specifies the behavior or structure of an entity. Entities and Architectures are always paired in VHDL descriptions.

artwork: The graphic materials generated for use in production of printed circuit boards, containing a representation of the copper circuit patterns in computer, mylar, or glass form.

associativity: The number of lines per set in a cache.

asynchronous: The way a system or signal is described when the rising edge of a clock pulse is not used as a reference signal.

Asynchronous Bus Protocol: A method of transferring data between a processor and peripheral which does not derive or rely on any timing parameters linked to a synchronous clock.

asynchronous: Operations which do not occur simultaneously with a specific time interval.

bandwidth: (1) the absolute difference between the upper and lower frequency limits of operation. (2) The points in a spectrum where the circuit response is 3 dB down from nominal.

baseline wander: A low frequency variation in the relative threshold position at the receive end of a transmission line.

Behavioral VHDL: A description of how a design should operate or its behavior, as opposed to its structure. This is the highest level of abstraction for a VHDL description.

BER: Bit Error Rate is the ratio of corrupted data to correctly received data. This ratio is typically small and expressed as an exponent (i.e., $1X10^{-12}$, one error in 10^{12} bits). BER may be expressed in either bits in error or Bytes in error.

bias: The DC component of an AC signal or the DC level of a signal dictated by a resistor divider.

BiCMOS: Bipolar Complementary Metal Oxide Semiconductor. An advanced silicon process technology that combines the best features of bipolar technology (high speed) and CMOS technology (low power, high density), but at a cost penalty relative to pure CMOS.

BiFAMOS (bipolar floating avalanched MOS): BiCMOS technology, combined with EPROM (dubbed FAMOS™ in the industry). Provides the world's fastest reprogrammable technology. See EPROM.

BIFO (Bidirectional FIFO): A FIFO, e.g., CY7C439, whose two sets of data pins can both be configured as either inputs or outputs, allowing transmission of data in both directions (though not simultaneously).

bipolar: A widely commercialized, silicon integrated circuit (IC) technology. Bipolar technologies create the highest performance silicon integrated circuits, but at the expense of high power consumption and the inability to make very large chips economically.

BIST: Built-In Self-Test is logic included in the chip that allows it to generate patterns and test them without external hardware or software intervention.

bit-cell: The nominal time period of a single bit in a serial data stream.

burst sequence: The sequence of addresses followed when multiple locations in a memory are being consecutively accessed in a single operation.

bus contention: A period in time when a common data or address bus may have more than one active driver on the bus at a given time.

cache: A small, fast memory located between the CPU and main memory. A cache's purpose is to store copies of the instructions and/or data the CPU is most likely to need in the near future so that the CPU can access them more quickly than if they were stored only in main memory.

cache hit: An access to main memory that is found in and serviced by the cache memory.

cache lock: A method, e.g. using a status bit, for ensuring that specific lines in the cache do not get replaced Users can lock critical programs in the cache to ensure that performance on these programs is high and deterministic.

cache miss: An access to the main memory that is not found in the cache memory, and therefore must be serviced by the main memory.

cache tag: A table of the current contents of a cache. The tag itself is made up of a varying number of address bits that uniquely identifies each line in the cache as coming from a specific main memory line.

carrier: A signal whose presence is necessary to allow communications.

CAS (Column Address Strobe): In dynamic RAMs, this signal is asserted to strobe the column address of the current access into the device after the row has been input.

cascading: (1) Connecting several smaller parts, usually SRAMs, Dual-Ports, or FIFOs, together in such a way that it creates an effective memory that is deeper or wider. (2) Cascading is the method used to connect multiple SmartCAMs into a SmartCAM array. A cascaded system connects the R1 input and output pins into a daisy chain, generates chip selects for the array, and has $\overline{SR}$ to $\overline{MRREN}$ logic.

clean: The status of a cache line when it contains the same data as the copy in main memory. See *dirty*.

clock generator: A circuit that is used to generate a clock signal used to trigger digital logic circuits.

clock stability: The stability of a clock signal with respect to frequency, pulse width, and amplitude.

CMOS levels: There are two sets of CMOS specifications: HC and HCT. The older HC devices are generally not TTL compatible, and the newer HCT (also FACT, FCT, etc.) are TTL compatible. Because the minimum V_{OH} level for TTL is 2.4V, TTL is not guaranteed to drive an HC input high.. A 4Ω 10,000Ω pull-up resistor to V_{CC} at the TTL device's output enables the device to achieve the HC V_{IH} level of 3.5V.

coax (coaxial cable): Copper cable consisting of a single central conductor, surrounded by a dielectric

spacer, having an overall cable shield that is isolated from the central conductor.

combinatorial: A logic function which does not involve any synchronous elements.

component: A component is a VHDL design unit that may be instantiated in other VHDL design units. Before it can be instantiated, it must be declared using the COMPONENT declaration which specifies the name of the component and lists its local signal names.

Concurrent Statement: As pertains to VHDL logic synthesis, it is a statement in an Architecture that executes or is modeled concurrently (simultaneously) with all other statements in the architecture.

converting ABEL: A technique whereby ABEL hardware descriptions are converted to VHDL.

CPLD: complex programmable logic device

CRC: Acronym for Cyclic Redundancy Check or Cyclic Redundancy Code. Used for error detection on serial data communication channels.

effective access time: A cache performance metric giving the average time required to service a memory reference.

coherency (consistency): Agreement between shared contents of members of the memory system.

crosstalk: The temporal change in either the magnetic field or the electric field of a signal on one conductor that results in an unwanted signal being coupled to other conductors.

Cyclic Redundancy Check (CRC): An error control mechanism based on use of an error-detecting code. The code can be described as follows. Given a k-bit message, the transmitter generates an n-bit sequence, know as a check sequence, so that the resulting message, consisting of $k + n$ bits is exactly divisible by some predetermined number. The receiver then divides the incoming message by the same number, and if the remainder is zero, assumes there was no error. The CRC codes are often expressed as polynomials.

daisy-chain: A method of making connections serially, from each point to each next point, in one continuous sequence (as in PCB layout).

data bus latency: This refers to the amount of time the data bus is driven after a given bus cycle terminates.

DCD: Duty Cycle Distortion is deterministic jitter that is typically caused by mismatches within the serial transmission line interface. It causes rising edges to be misplaced in one direction and falling edges to be misplaced in the opposite direction (typically an identical offset).

DDJ: Data Dependent Jitter is a deterministic jitter that is a function of characteristics of a particular serial interconnect media and the content of the serial data stream. It causes edges (either rising or falling) to be misplaced by a distance that varies as a function of their distance from preceding transitions. Variations in pulse width inherent in the data stream cause variations in the magnitude of the misplacement of the data transitions.

deadlock: The condition in which two or more processes that share resources halt because no process can obtain all the resources it needs to continue.

Dhrystone: A measurement of PC or microprocessor performance taken while running a benchmark program that consists of a loop of simple integer operations.

differential: Mode of communication where two complementary signals are compared to each other to determine the logical state of the signal. Also known as a balanced connection.

dirty: The status of a cache line that has been modified and now contains data different than the copy in main memory.

dispersion: Widening of a pulse as it travels down a transmission line due to characteristics of both the pulse and the media.

DMA (Direct Memory Access): A design technique that offloads some of the I/O processing from the CPU. A DMA controller allows the CPU to continue operation while the controller controls block transfers between I/O and memory or between separate memories in a multiprocessor system.

DRAM (Dynamic Random Access Memory): The main memory in almost all computers. The highest-volume chip manufactured. See *SRAM*.

dual-port RAM: An SRAM that can process two different accesses simultaneously.

duplex: Capable of bidirectional operation and having multiple sources and destinations.

duty cycle: The relationship of a clock pulse HIGH time to its LOW time—expressed as a percent.

ECC: Error correction code, used to ensure that data is correctly stored or transmitted.

ECL (Emitter-Coupled Logic): A convention for "one" and "zero" voltage reference levels in one integrated circuit family. ECL "one" and "zero" voltage levels are very small and, therefore, are able to be sent into and out of integrated circuit packages very quickly. ECL logic is used in the fastest available computers, such as those from Cray and Convex. ECL circuits are fabricated in Bipolar or BiCMOS technology. See *TTL*.

EDC: Error detection/correction. Hardware or software used to generate/check ECC bits. Upon single-bit error detection, the EDC will also correct the faulty bit.

Entity: As pertains to VHDL logic synthesis, the declaration which lists or describes the ports (the interfaces to the outside) of the design. An Entity describes the names, directions, and data types of each port.

error-free window: The widest possible area within which a transition can occur and be correctly interpreted by the receiving circuit. This is a measure of jitter tolerance.

ESCON: A protocol used to interconnect IBM compatible computers at data rates of 20 MByte/sec.

expanders: Extra product terms in MAX EPLDs that are available to be used and shared by all macrocells in a LAB.

eye pattern: Method of examining a data stream that compares the stable versus unstable portions of a bit-cell.

fall time: The amount of time it takes a digital logic signal to transition from a logic HIGH to a logic LOW.

FDDI: Acronym for Fiber Data Distribution Interface. A high-speed local-area network, using a pair of fiber optic links in a dual token ring topology. Data rates of 10 Mbytes per second are supported.

fiberoptic: Those components whose primary mode of operation is through the use of optical rather than electrical energy.

Fibre Channel: An ANSI standard data communications interface for computers and peripherals. A high-performance computer interconnect standard that describes a method of interconnecting computers and peripherals at specified data rates between 13 and 100 MByte/sec.

FIFO – First In First Out Memory: A memory device where data is accessed from memory in the order that it was written into memory.

finite state machine: A synchronous sequential circuit whose current state is defined by the contents of its flip-flops and whose next state is determined by its current state and the inputs to the circuit.

Fourier transform: A mathematical operation used to convert time-domain expressions into equivalent frequency-domain expressions.

FPGA: field programmable gate arrays.

framer: The internal logic included in the HOTLink Receiver that examines the serial bit stream and looks for the SYNC character. When it is found, this logic aligns the deserializer with the transmitted byte boundaries.

Framing: (As it applies to the CY7B933 HOTLink Receiver) The process of determining what the proper byte boundaries are in a serial bit data stream.

full duplex: A device or system that can transmit information in two directions simultaneously. See simplex, half duplex.

ground bounce: When many outputs of a device change from HIGH to LOW there is a rush of current into the output drivers. If the inductance to

ground is sufficient, the virtual ground level is raised due to this inductance. The voltage spike caused by this phenomenon is called ground bounce.

half duplex: A device or system that can transmit information in two directions, but not simultaneously. The Cypress CY7C439 BIFO is a half-duplex device. See simplex, full duplex.

HIPPI: Acronym for HIgh Performance Peripheral Interface. A standard way of interconnecting high performance peripheral devices to medium and large scale computers. The interface is characterized by a parallel bus using ECL logic levels, and is capable of data transfer rates of several hundred Mbytes per second over relatively limited distances

HOTLink: The name for the High-Speed Optical Transceiver Link chip set.

Hysteresis: Relating to a condition where the the path or level being observed varies depending on an increasing or decreasing situation.

IPI: Acronym for Intelligent Peripheral Interface. Originally defined in IPI1 as a controller interface for high performance disk drives, the standard has evolved in IPI3 to a relatively complete channel interface, intended for general purpose high speed I/O in medium to large scale computer systems.

jitter: Jitter is a typical form of corruption that occurs on serial data streams. It is a misplacement of the timing of a transition from its ideal position. The two basic types of jitter are Random and Deterministic. Deterministic jitter is further divided into DCD and DDJ.

jitter tolerance: Jitter tolerance is the ability of the deserializer to recover data from a corrupted serial data stream. This specification indicates tolerance to misplaced transitions within the expected bit window. This tolerance may be expressed in time (i.e., nanoseconds) or as a percentage of a bit time (i.e., ±45% of a bit time). See error-free window.

K28.5: A special character that is defined in the 8B/10B code. This character is typically used as an idle or fill character when no data is to be transmitted on the serial media. Sometimes referred to as a Sync Character. See SYNC.

LAB (Logic Array Block): In Cypress MAX PLD devices, the LAB represents a separate functional block in the device. Each type of MAX PLD has a different number of LABs.

latch-up: A regenerative phenomenon that occurs when the voltage at an input pin or an output pin is either raised above the power-supply voltage potential or lowered below the substrate voltage potential, which is usually ground.

LFSR: A Linear Feedback Shift Register that is used to generate a pseudo random sequence of characters. The LFSR in the HOTLink is used to generate and check the BIST sequence.

library: A library is a logical storage facility for design units. Before a component can be instantiated in a higher-level design unit, its package must be compiled into a library that is visible to that design unit, usually the current work library.

line (block): The basic unit of information exchange between a cache and main memory or between a parent cache and its child(ren) cache(s).

line size: The number of bytes or words in one cache/main memory line. In a cache system, a line is the quantum of data identified by the cache tag and is the smallest quantum of data that can be transferred between the cache and main memory. Whenever a new entry is placed into the cache, one line is transferred. Common line sizes are 16 and 32 bytes.

Linear Feedback Shift Register (LFSR): A shift register using XOR gates and feedback to implement cyclic redundancy check polynomials.

MACH: The trademark for Advanced Micro Devices' family of complex programmable logic devices.

macrocell: A low-level block of logic in programmable logic devices. This block can include one or more registers, along with configurable feedback and/or output paths.

master device: A device that controls the timing for data exchanges between two devices. Or when devices are cascaded in width, the master device is the one that controls the timing for data exchanges between the cascaded devices and an external interface. The controlled device is called the slave device.

MAX7000: the trademark for Altera's family of complex programmable logic devices

Mealy machine: A state machine in which outputs depend on the current state and current inputs.

metastable: A condition where neither a logic zero or a logic one can be guaranteed due to a timing violation to a synchronous logic element.

Moore machine: A state machine in which the outputs depend only on the current state.

MTBF (Mean Time Before Failure): The average length of time a system or component will continuously operate between failures, given a certain set of operating conditions.

multimode: Fiberoptic communication where light propagates in one or more modes through the optical media.

multiprocessing: A computer architecture in which two or more processing units are coupled together to run different programs simultaneously, while sharing the same computer frame and memory. The problem: several processors sharing the same memory without corrupting each other's data.

Non-Return-to-Zero-Invert (NRZI): A method of encoding a serial bit stream. A transition indicates a 1 and no transition indicates a 0, hence the term non-return-to-zero. The waveform doesn't return to zero to indicate a bit value.

NTSC (National Television Systems Committee): A video format used in the USA.

Number Representations: Required VHDL syntax for binary, octal, decimal, and hexadecimal numbers.

OLC – Optical Link Card: The OLC is a LED/laser based data communications adapter card. These cards are based on the Fibre Channel standard.

optical module: A device capable of bidirectional conversion of electrical signals to optical signals for use in communicating over fiberoptic cables.

OR: The or logic gate.

oscillator: A circuit that is generally crystal controlled and is used to generate a clock frequency.

overshoot: The amount by which the amplitude of a signal transitions above its final value on a low-to-high transition.

package: A package is a collection of VHDL declarations that can be used by other VHDL descriptions. For the purpose of creating hierarchical designs, a package consists of one or more components. However, a package may also include other types of declarations.

PAL (Phase Alienation by Line): A video format used in Europe and the Far East.

Parity: An error detection scheme where a status flag is saved indicating that the number of "on" bits is even or odd.

PECL: A variation of ECL often referred to as Positive-ECL or Pseudo-ECL where the devices are operating off of a positive power supply instead of 0 volts to −5.0 volts .

photodiode: Optoelectric device capable of converting changes in received light amplitude into changes in current.

Physical Layer: The devices and components that attach directly to the physical media. These include drivers, shifters, filters, etc. that are needed to implement the physical requirements of the communication protocol. The Physical Layer is usually the lowest layer of a communication protocol stack.

PIA (Programmable Interconnect Array): In Cypress MAX devices, the PIA is the routing path between separate logic array blocks (LABs). The PIA is automatically routed and provides uniform timing throughout the devices.

PIM: programmable interconnect matrix

PLD (Programmable Logic Device): An integrated circuit that is shipped blank to customers and can be field programmed into a custom logic circuit, such as a counter, an adder, or a state machine.

PLL: (Phase-locked loop.) Hardware used to minimize clock skews by keeping them in phase with respect to a reference clock. Or, used to generate a clock that is a multiple frequency of the reference clock.

Polarity Conventions: Rule of thumb for assigning and interpreting polarity in VHDL.

priority: In a SmartCAM array with multiple locations responding, the responder with the highest priority is the responding SmartCAM in the R1 daisy chain closest to the system R1 Below In signal. Within a SmartCAM, the highest priority responder is the one with the lowest physical address.

process: As pertains to VHDL logic synthesis, a collection of Sequential Statements appearing in a design Architecture. The Process itself is evaluated concurrently within the Architecture.

product term: A Boolean AND of all the inputs to the PLD array.

PROM (Programmable Read-Only Memory): Memory in which the data is fixed even when the power is turned off. Programmable ROMs are shipped blank to customers and customized in their facilities.

random jitter: Random jitter is a measure of edge misplacement that is uncorrelated with either the interconnect media or the serial data stream. It is usually caused by random effects in the interconnect system or by thermal effects in the high gain amplifiers used to translate between optical and electrical information.

RAS (Row Address Strobe): In dynamic RAMs, this signal is asserted to strobe the row address into the device; the address inputs are time-multiplexed.

recursion: see *recursion*.

reference: A request by the processor to read or write a memory location.

reframe: Reframe is the function of determining and aligning the deserialization logic with correct byte boundaries, so that the data can be decoded correctly.

refresh: Transactions used to periodically replenish the charge on storage capacitors used in DRAM cells.

rise time: The amount of time it takes a digital logic signal to transition from a logic LOW to a logic HIGH.

RTL (Register Transfer Level): A level of description in hardware design languages that consists of operations being described in terms of register- and gate-level structures.

run length: Run length can be either the distance between transitions (i.e., the maximum number of adjacent ones or the maximum number of adjacent zeros) in a serial data stream; or the length of time that an error will propagate after an error event. In the first case, the 8B/10B code rules allow a run-length of five (5) bits. In the second case, a single error event can occur within a single byte, and be terminated at the next one, or in the case of a running disparity error (or a framing error) the effect of the error can continue for an indeterminate time.

running disparity: Running disparity is a concept included in the 8B/10B code that allows it to insure a perfect DC balance. It is a measure of difference between the number of 1s (high-bits) and number of 0s (low-bits) and is automatically managed by logic which selects alternative codes from the possible code tables to assure a perfect match.

running disparity error: A type of error in a serial data bit stream in which there are too many consecutive bits at a single logic level for the data received to be valid.

SBCCS: Acronym for Single Byte Command Code Set. A command set defined as a Fibre Channel level 4 protocol. The set is characterized by having, in all cases to command defined in the first byte, and all subsequent bytes providing only parametric information relating to the command.

scrub: A DRAM Accelerator operation where data errors are detected and corrected during refresh cycles.

SCSI: Acronym for Small Computer Systems Interface. A standard way of interconnecting peripheral devices, such as disk and tape to small to medium sized computers. It is specified in a document from the ANSI committee X3.31. Up to 7 storage devices can be attached to a single computer using a single SCSI network.

SECAM (Systeme Sequentiel a Memoire): A video format used in France and Europe.

semaphore: A software technique for providing explicit mutual synchronization of parallel sequential (software) processes. Semaphores are initialized with the value zero or one before the processes are started. After initialization, the processes access the semaphores only via two specific operations—the so-called synchronizing primitives. The operations carried out on semaphores are referred to as P and V, which are the first letters of the Dutch words corresponding to WAIT and SIGNAL, respectively.

Sequential Statement: As pertains to VHDL logic synthesis, it is a statement appearing within a Process. All statements within a Process are executed or modeled in order, similar to programming languages such as C or Pascal.

set: A collection of cache locations in which a line may reside.

set associativity: A property that allows a cache to be divided into sets, each of which contains one or more lines. This property enables a line of main memory to map to more than one line in the cache; the line of main memory can map to one line in each of the sets. When searching the cache, the tags of one line from each of the sets are compared to the reference tag concurrently to determine to which set, if any, the main memory line was mapped.

shielded twisted pair: Copper cable consisting of two insulated conductors twisted together in a controlled fashion, having an overall cable shield that is isolated from both conductors.

simplex: A device or system that can only transmit data in one direction. See half duplex, full duplex.

single-ended: Mode of communication where a received signal is compared to an internal or external fixed reference to determine the logical state of the signal. Also known as an unbalanced connection.

single mode: Fiberoptic communication where light propagates in only one mode through the optical media.

slave device: A device that allows another device to control the timing for data exchanges between the two devices. Or when devices are cascaded in width, the slave device is the one that allows another device to control the timing for data exchanges between the cascaded devices and an external interface. The controlling device is called the master device.

SmartCAM array: A SmartCAM array is one or more SmartCAMs connected together with necessary external logic. A SmartCAM array is modeled as a single, large SmartCAM chip.

snooping: A method used in multimaster applications in which one or more of the masters contain data or instruction cache. Cache coherency and maintenance operations occur when the active master requests an operation on data that happens to be contained in a non-active master's cache. The non-active master can intervene, and depending on the type of transfer, maintain its cache accordingly and possibly supply its cached data to the active master. The act of monitoring the bus address and data while not the active master is considered "bus snooping."

spectrum analyzer: Frequency domain oscilloscope.

SRAM (Static Random Access Memory): A Random Access Memory allows the user to store and retrieve data at a high rate of speed. The term "static" means that as long as the power is on, the memory will retain its data. This feature contrasts with Dynamic Random Access Memories (DRAMs) whose data is stored in a temporary medium, which allows the data to fade away every few milliseconds. DRAMs must have their data refreshed continuously, even when the power is on, but they provide greater density at lower costs than SRAMs.

starvation: The condition in which one process that shares resources with other processes halts due to the fact that it can never obtain the resource(s) it needs to continue.

Structural VHDL: A description of how the various components that make up a design are connected. This is the lowest level of abstraction for a VHDL description.

sum of products: A Boolean algebra construct where inputs are logic ANDed, and then the outputs of the AND gates are ORed together. This is how most PLDs are constructed.

SYNC: The special character included in the 8B/10B code that allows the serial data stream to be properly decoded. This character (K28.5) contains a unique sequence of bits that can never occur with any combination of legal data bytes in an undamaged data stream.

synchronous: The way a system or signal is described when the rising edge of a clock pulse is used as a reference signal.

system read: A system read drives data read from a single SmartCAM onto the data bus. All chip selects in a SmartCAM array are asserted for this bus cycle and a single SmartCAM is selected by the $\overline{SR}$ to $\overline{MRREN}$ logic.

system write: A system write causes the data on the data bus to be written into the corresponding register of every CAM in an array. All chip selects in a SmartCAM array are asserted for this bus cycle.

termination: A technique of matching the impedance of a driver to a line or a line to a load.

thevenin: A type of circuit used to terminate a transmission line.

three-state: A signal that can be at a logic level high, low, or high impedance.

token passing: (as applied to state machines) A design-methodology in which an n-bit state machine is built with n 1-bit registers, instead of with $\lceil \log_2(n) \rceil$ registers. In a token-passing state machine, the state is indicated by which specific 1-bit register contains the only "1," and state transitions are accomplished by passing the "1" (i.e., the token) from one register to another.

transformed transaction: A transaction that is changed from its original intent. A read becomes a write and a write becomes a read.

transimpedance amplifier: Amplifier designed to convert a small change in current into a large change in voltage.

transparent write: Possible only on separate I/O RAMs. During a transparent write, the data appears at the outputs as the data is written into the array.

transmitter: A circuit used to send information or digital data.

TTL (Transistor Transistor Logic): The dominant convention for "one" and "zero" voltage reference levels in integrated circuits. TTL circuits are pervasive in most electronics applications, including personal computers, workstations, and consumer electronics. See *ECL*.

twinax (twinaxial cable): Copper cable consisting of two insulated conductors assembled parallel to each other, having an overall cable shield that is isolated from both conductors.

uniprocessing: A computer architecture in which one processing unit runs all of the programs.

VCO: Voltage controlled oscillator; a clock generator that uses input voltage levels to vary the clock frequency.

VHDL (VHSIC–Very High Speed Integrated Circuit–Hardware Description Language): A powerful, standard (IEEE 1076) software language for describing and simulating hardware designs, from transistor level up to full-system level. It is the language used in Cypress' *Warp* PLD design tools.

watchdog timer: A watchdog timer limits the amount of time a system will wait for a bus cycle termination signal (e.g., $\overline{RDY}$). If the watchdog timer completes, the system assumes that an error has occurred and responds appropriately.

XOR: The exclusive or logic gate

Index

An italicized page number means the reference is to a figure.

.ABL, converting to VHDL, 3–180

.ABL to VHDL
  conversion, pitfalls, 3–191
  conversion approach, 3–180
  conversion preparation, 3–180

.DOC file, 3–181

68020, and the VIC068A, 7–6 to 7–12

68040 processor, 2–37 to 2–39

680x0
  asynchronous read and write cycles, 7–128 to 7–129
  bus cycle machine, 7–134

74FCT244, 6–53, 7–17

74FCT245, 7–14, 7–17

74FCT373, 7–17
  switching characteristics, 6–74

74FCT543, 7–14

74HC40103, 2–8

80386 processor, 2–34
  See also AMD, 386

80486 processor, 2–36
  with CY7B991/2, 6–53 to 6–76

88100 processor, 2–45

8B/10B, 5–63

8B/10B encoder, 5–1

960CA processor, 2–39

ABEL, 2–25 to 2–29, 3–180
  comparator PROM, source code, *2–28*
  generating LE and IRDY for Am29000, source code, 2–50
  PALC22V10 cycle decoding, source code, *7–11*

Abel–HDL, 3–217
  vs. VHDL, 3–219

AC impedance, 1–4

AC termination, 1–20

ACFAIL, 7–4

active power, 1–42 to 1–45

adder, 3–191

address
  left port camped on in dual-port RAMs, 4–9
  right and left equal simultaneously in dual-port RAMs, 4–9
  transition detection, 4–12 to 4–13
    sequence, *4–13*
  unequal in dual-port RAMs, 4–9

ADSP2100A, 2–53, 2–54
  DSP to memory interface, 2–53
  initialization, 2–53
  timing, 2–54
    external program memory, *2–54*

alias SYNC, 5–85

ALU, combinatorial, 4–1

ALUs, designing with PROMs, 2–59 to 2–65

Am29000 processor, 2–41 to 2–43

AM7968/7969–175, 5–93

AMD, 386 processor, 2–34, 2–35

AND-OR logic, 3–138

ANSI/IEEE Standard 1014, 7–1

arbitration logic, in dual-port RAM, 4–8 to 4–9

architecture
  CPLD, 3–102
  CY7C335, 3–73
  pipeline, 3–77

serial decoder, 3–82
Architecture section, 3–220
architecture, 3–81
comparator, 3–79
multiplexer, 3–79
array based interconnect, 3–103, *3–104*
ASCII binary PROM programming file format, 2–20
ASCII–HEX PROM programming file format, 2–20
asynchronous
preset and reset product term, 3–106
preset/reset, 3–112
automatic test vector, 3–156
baseline wander, 5–34
behavioral descriptions, 3–73
behavioral logic description, 3–150
behavioral VHDL, 3–154
bias generator, *1–57*
BiCMOS, products, 1–58 to 1–59
bipolar ICs
history of, 1–49
replacing with CMOS, 1–1
BIST, 5–211
See also HOTLink, built-in self-test
bit-slice CPU control
execution in state machines, 3–28
inactive states, 3–32
INTERRUPT mode, 3–29
NONPIPELINED RUN mode, 3–28
PIELINED RUN mode, 3–28
REPEAT INSTRUCTION mode, 3–29
SINGLE STEP mode, 3–28
STOP mode, 3–28
WAIT mode, 3–28
blanking, 2–5
blanking signals, *2–7*
Boolean equations, 3–73
Boolean logic description, 3–150
bottom-up approach, 3–150
buffers
core, 1–37
for communication between systems, 4–2 to 4–3
input, 1–36, 1–55
output, 1–37
bufoe component, 3–81, 3–111, 3–183, 3–209
Built-In-Self-Test mode, 4–49, 4–53, 4–57, 4–63
buried registers, 3–75, 3–111
bus interface
products, 1–61
signals, 7–14
specialized logic, 7–17
bus lines, connecting, 7–8
buses
bidirectional, 1–18
interface products, 1–61
size on VMEbus board, 7–21
BUSY signal, in dual-port RAMs, 4–10
cables, coaxial, 1–16
calculations, obtaining values, 1–46
CAM
array, 9–2
defined, 9–1
capacitance, for ideal case, 1–21 to 1–22
capacitors
decoupling, 1–31
filter
high-frequency, 1–31 to 1–32
low-frequency, 1–33
paralleling, 1–33
impedance vs. frequency, *1–32*
CD (carrier detect), 5–222
channel resistance, 1–18
characteristic impedance, 1–4
CISC, and CY7C270, 2–34
clamping diodes, 1–49 to 1–50, 1–51
input, 1–2
clock
control using CY7C361, 7–131
distribution, 6–2 to 6–4, 6–53 to 6–54
driver, with programmable skew, *6–56*
generation, 7–131 to 7–132

generator
implementation, 3–36 to 3–41
inputs and outputs, 3–29 to 3–30
stretching, 7–129 to 7–130

CML, 1–61

CMOS
characteristics of Cypress CMOS, 1–55
high-current, 1–54
history of, 1–51
input buffers, 1–55

coarse-grain logic cell, 3–138

coaxial cable, 1–16

Code Rule Violation, 5–90

coefficients, reflection, 1–6

combinatorial logic equations, 3–222

comments, 3–181

comparator, 3–190
designing with VHDL, 3–78 to 3–79
Warp2 report file excerpt, 3–90
Warp2 source code, 3–89

compiler, VHDL, 3–73, 3–77

concurrent statements, 3–224

Configuration statement, 3–220

constants, 3–181

converter, CY7C611A to 680x0 bus, 7–131

copper signal characteristics, 5–32
8B/10B code, running disparity, 5–33
8B/10B code dependencies, 5–32
8B/10B data, frequency characteristics, 5–37
baseline wander, 5–34
jitter, 5–34
NRZ modulation, 5–32
transmission line effects, serial data, 5–33

core buffers, 1–37

counter, 3–190
on-chip programmable, 2–33

CPLD, 3–137

CPLDs, overview, 3–102 to 3–103

creating files, using high-level languages, 2–25

crosstalk, 1–2, 5–13

crystal oscillator, 6–81
accuracy considerations for time-keeping in PC design, 6–81
parallel vs. serial resonant oscillator circuits, 6–81
selecting a high-accuracy reference crystal, 6–82
specifying a custom parallel-resonant crystal, 6–81

current
circuit conventions, 1–52
input, 1–55–1–56
measurement, 1–37
operating supply
vs. frequency characteristic, 1–41–1–42
worst-case, 1–41

CY27H010, 2–60, 2–61

CY7B166, switching characteristics, 6–73

CY7B46X, interface to CY7B923, 4–46 to 4–48

CY7B923, 5–2, 5–135
as ECL clock source, 5–135
block diagram, 5–135
clock generator, 5–135
encoder, 5–136
input register, 5–136
output, 5–136
shifter, 5–136
test logic, 5–136
clock issues, 5–137
clock skew, 5–140
device packaging, 5–140
drive capability, 5–140
duty cycle stability, 5–138
HOTLink transmitter printed circuit layout, 5–140
jitter, 5–138
power supply current, 5–140
rise and fall time, 5–139
termination, 5–139
frequency range, 5–136
fulfilling the requirements, 5–136
HOTLink transmitter
block diagram, *5–2*
clock generator, 5–3
description, 5–2
encoder, 5–2
Input register, 5–2
outputs, 5–3

shifter, 5–3
test logic, 5–3
HOTLink transmitter features and specifications, 5–135
ideal clock circuit, 5–135
interface to CY7C42X/46X, 4–46 to 4–48
interface to wide data clocked FIFO, 4–57 to 4–66
interfacing to clocked FIFOs, 4–49 to 4–56
test circuit, 5–137

CY7B923/933, 5–128

CY7B933, 5–4, 5–92
HOTLink receiver
Clock Sync, 5–4
Decode register, 5–5
decoder, 5–5
description, 5–4
ECL-TTL translator, 5–4
framer, 5–5
logic block diagram, *5–4*
Output register, 5–5
serial data inputs, 5–4
shifter, 5–5
test logic, 5–6
interface to wide data clocked FIFO, 4–57–4–66
interfacing to clocked FIFOs, 4–49 to 4–56

CY7B9331, 5–214

CY7B9331 HOTLink-OLC system interface, *5–214*

CY7B9331 OLC receiver
CLK0 pin, 5–215
ENSYNC pin, 5–215
introduction to, 5–214
loop pin, 5–214
STATUS pin, 5–214
SYNC and CLK0 stretching, 5–217
SYNC in BIST mode, 5–218
BIST loop, 5–219
entering BIST mode, 5–218
leaving BIST, 5–219
reframing while in BIST, 5–219
start of BIST, 5–219
SYNC in bypass mode, 5–215
entering framing, 5–215
leaving framing, 5–216
normal operation, 5–215
SYNC in encoded mode, 5–216
entering framing, 5–217
leaving framing, 5–217
normal operation, 5–217
SYNC pin, 5–215
normal SYNC pin timing, 5–215

CY7B951, 5–221

CY7B991/2, 6–1 to 6–41
AC characterization, 6–37 to 6–40
derating calculations, 6–71
functional description, 6–54 to 6–65
implementations, 6–27 to 6–31
logic block diagram, *6–1*, *6–41*
Output Selection Block, 6–61
skew configurations, 6–78
Test mode, 6–77 to 6–80
with 80486 and R3000, 6–53 to 6–76

CY7C132
used in master standalone operations, 4–13
used in slave word-width expansion, 4–13

CY7C142, used in slave word-width expansion, 4–13

CY7C169, power calculation example, 1–37

CY7C245A, 2–4, 2–19

CY7C258, 2–60

CY7C258/9, 2–1 to 2–19
features, 2–3 to 2–4

CY7C259, 2–61

CY7C270, 2–32 to 2–48
architectural features, 2–32 to 2–33
benefits of using, 2–33
interfacing to CISC processor, 2–34

CY7C276, 2–60, 2–61
interfacing to DSPs, 2–51 to 2–58
introduction, 2–51

CY7C335, 3–219, 3–223
block diagram, *3–74*
described as state machine using *Warp2*, 3–236 to 3–268
designing with, 3–73 to 3–101
hidden macrocell, *3–76*
input clocking scheme, 3–76
input macrocell, *3–74*
input/output macrocell, *3–75*
overview, 3–73 to 3–76

CY7C342, 3–213

CY7C361
for clock control, 7–131
implementation as state machine, 3–39
input and output signals, 7–131
on a board without a processor, 7–17

CY7C370, using *Warp* to design with, 3–110–3–120

CY7C371, 3–121
speed considerations, 3–130
using for FIFO dipstick, 4–39
utilization, 3–130

CY7C373, 3–213

CY7C380 Family, 3–144
I/O cells, 3–147
logic cells, 3–147
performance and timing model, 3–148
routing, 3–145

CY7C380 family, power consumption, 3–172

CY7C380 Family architectures, explained, 3–144

CY7C381, 3–178
equivalent capacitance, 1–45

CY7C382, 3–176, 3–178, 3–213
equivalent capacitance, 1–45

CY7C383, 3–178

CY7C384, 3–178

CY7C385, 3–178

CY7C386, 3–178

CY7C429
decoupling capacitor example, 1–31
in unterminated line example, 1–23

CY7C42X, 4–19, 5–2
interface to CY7B923, 4–46 to 4–48

CY7C43X, 4–19

CY7C44X, 5–2

CY7C45X, programming, 4–34

CY7C611A, 2–43, 3–207
interfacing with the VIC64, 7–125 to 7–137
load and store cycles, 7–127
memory interface signals, 7–126
overview, 7–126 to 7–127

CY7C901, dual-port memory operation, 4–1 to 4–2

CY7C964, 7–93
address comparator configuration, 7–73
address comparison signals, 7–145
byte-width mode, 7–139
features, 7–138
local data swap buffer, 7–147
local signals, 7–145
used with VIC64 and VIC068A, 7–138 to 7–149
word-width mode, 7–142

CY9266, 5–65, 5–141

CY9266 parts list, 5–183

CY9266–T parts list, 5–189

CYB675
boot-up, 7–70
evaluation board, 7–69
local address symbols, 7–76
local control register, 7–69

CYM4210/20, 4–19

CYM7232 DRAM array, *8–16*

CYM7232/CYM7264, 8–1, 8–7, 8–32

CYM7264 DRAM array, *8–17*

data, ownership, 4–3

data bus, size, 7–21

DATA signal, 3–207

DC power, 1–35

DCD, 5–36, 5–72

DDJ, 5–35, 5–72

deadly embrace, 4–4 to 4–5

DEC binary PROM programming file format, 2–21

decode logic, 3–129

decoder, 3–80, 3–190
VHDL source code, 3–93
Warp2 report file excerpt, 3–94

decoupling capacitor, calculations, 1–31

delay, propagation, 1–5

design
fitting, 3–231 to 3–235
philosophy, 1–34

state machine for FIFO dipstick, 4–40
tools, 2–25 to 2–30
Warp, 3–231
design and I/O declarations, 3–219
design decomposition, 3–152
detailed architecture, FPGAs, 3–137
dielectric constant, 1–6
Dijkstra, E. W., 4–18
diodes
clamping, 1–60
input clamping, 1–49, 1–51
output clamping, 1–50
PN junction, 1–2, 1–51
Schottky, 1–23
zener diode protection, 1–30
discontinuities, voltage reflections due to, 1–9
discrete logic, 2–59
DMA, 3–207
DMA (direct memory access), 5–128
HOTLink, 5–128
DMA transfers, unattended, 7–22
dot extension, 3–182
DRAM, 3–150
DRAM Accelerator, 8–1, 8–32
3.3-volt DRAM interface, 8–32
32-bit error detection-correction calculation, 8–10
5-volt DRAM interface, 8–32
64-bit error detection-correction calculation, 8–11
algorithm for determining DRAM configuration, 8–7
configuration and diagnostics, 8–7
control signal translation, 8–32
data check bits translation, 8–32
device connection, 8–4
DRAM diagnostic sequence, 8–8
DRAM refresh operations, 8–24
error correction polynomial generation, 8–9
interfacing, 8–13
DRAM configurations and capacitive loading, 8–13
introduction, 8–13
timing selections, 8–13
introduction, 8–7
power-on reset, 8–4
preinit and INIT, 8–5
preparing for DRAM Access, 8–5
read operation, 8–18
read-modify-write operation, 8–24
recommended drivers, 8–16
register programming, 8–4
scrub disabled, 8–2, 8–7
scrub operation, 8–5
set-up for basic operation, 8–1
getting started, 8–1
write and read operations, 8–5
write operations, 8–24
drivers, programmable-skew clock driver, *6–56*
DSACK lines, connecting, 7–9
DSP, and the CY7C270, 2–47
DSP1616, 2–51–2–53
DSP to memory interface, 2–52
initialization, 2–52
memory maps, 2–52
timing, 2–53
external program memory, *2–53*
DSP56000, 2–54–2–56
DSP to memory interface, 2–55
initialization, 2–54
memory maps, 2–55
timing, 2–55
external program memory, *2–56*
DSPs, interfacing high-speed PROMs, 2–51–2–58
DTL, families, 1–50
dual-port RAMs, 4–1–4–19
arbitration logic, 4–8
BUSY signal, 4–10
cell history, 4–4
Cypress family, 4–5
design example, 4–15–4–18
interrupt logic, 4–7
left port camped on an address, 4–9
mailbox signaling, 7–3 to 7–4
operation, 4–1 to 4–2, 4–6 to 4–7
right and left addresses equal simultaneously, 4–9
standalone operation of, 4–13
unequal addresses, 4–9
using single-port RAMs, 4–2

dual-port RAMs, in VIC068A, 7–12

ECL, 5–3, 5–6, 6–83
  advantages, 6–83
  history of, 1–51
  I/O characteristics, 1–60
  input levels, 1–61
  logic levels, 6–83
  output levels, 1–61
  output routing and board layout issues, 6–84
  outputs, 6–83
  pad structure, 6–83
  power supplies, 6–83
  switching characteristics, 1–61
  terminating resistor values, 6–84
  transfer curves and specification points, *1–62*

ECL logic families, 5–10
  100K ECL, 5–12
  10K ECL, 5–11
  10KH ECL, 5–11
  HOTLink ECL inputs, 5–14
  HOTLink ECL outputs, 5–12
    differential connections, 5–13
    single-ended connections, 5–12
  mixing, 5–14
    differential connections, 5–16
    single-ended connections, 5–16

ECL output biasing, 5–17
  matched loading, 5–19
  output drivers, AC characteristics, 5–20
  shunt bias to V_{EE}, 5–18
  shunt bias to V_{TT}, 5–17
  Thevenin bias to V_{TT}, 5–18
  Y-bias to V_{TT}, 5–18

ECL to TTL translator, 5–101

ECLiPSE family, 1–51

effective time constant, 1–15

electrical characteristics, scan path, 3–178

energy considerations, for driving transmission lines, 1–7

ENIAC, 3–2

Entity section, 3–219, 3–220

Entity statement, 3–207, 3–213

equivalent capacitance, pASIC380, 1–45

error
  running disparity, 3–123
  undefined character, 3–123

error-free-window, test, 5–73

errors, soft, 1–2

ESCON, 5–3, 5–63, 5–83, 5–92

ESD
  protection, 1–56
  protection circuitry, 1–2

evaluation board, for VIC64, 7–69

Exorcisor PROM programming file format, 2–21–2–22

Exormax PROM programming file format, 2–22

Fast Fourier Transform, 5–38

FCT. *See* 74FCT

FCT-T logic family, 1–60
  clamping diodes
    input, 1–60
    output, 1–60
  CMOS process technology, 1–60
  FCT-T, 1–60
  FCT2-T, 1–60

feedback, internal, in state machine PROM, 2–3 to 2–4

Fibre Channel, 5–3, 5–63, 5–83, 5–111
  data service types, 5–111
    class 1 data service, 5–111
    class 2 data service, 5–111
    class 3 data service, 5–111
  design example, 5–121
    frame assembly, 5–126
    frame disassembly, 5–125
    input data path receiver state machine, 5–121
    input data path serial code, 5–121
    ordered set decoder, 5–122
    ordered set encoder, 5–126
    output data path serial encode, 5–126
    output data path transmitter state machine, 5–126
    port state machine, 5–126
    summary, 5–126
  frame structure, 5–113
    CRC, 5–115
    data field, 5–115
    end of frame, 5–115
    frame header, 5–115

start of frame, 5–113
layer 1 receiver state machine, 5–116
loss of sync state, 5–116
not operational states, 5–119
on–line and off-line condition, 5–119
sync acquired state, 5–116
layer 1 transmitter state machine, 5–119
failure state, 5–121
not enabled state, 5–121
open fibre state, 5–121
working state, 5–121
ordered sets, 5–112
frame delimiters, 5–113
primitive sequence, 5–113
primitive signals, 5–113
overview, 5–111
primitive link protocol, 5–115
port state machine, 5–116
protocol layers, 5–112
layer 0, 5–112
layer 1, 5–112
layer 2, 5–112
layer 3, 5–112
layer 4, 5–112
protocol layer summary, 5–112

FIFOs
applications, 4–20
asynchronous ports, 4–40
clocked, 4–29 to 4–38
depth expansion, 4–35 to 4–36
interfacing to CY7B923 and CY7B933, 4–49 to 4–56
resetting and programming, 4–33
using as standard FIFO, 4–36 to 4–38
width expansion, 4–36
configurations, 4–21 to 4–23
corrupted or repetitive data, 4–24 to 4–25
dipstick, 4–39
architecture, 4–41
differences from programmable FIFOs, 4–42
state machine design, 4–40
Warp2 implementation, 4–40
generic interface to CY7B923, 4–46
large, 4–19 to 4–29
locking up, 4–25
missing data, 4–25
out-of-sequence data, 4–25
problems with, 4–24
reading to and writing from, 4–19 to 4–20
reads, 4–30 to 4–31
resetting, 4–58
resetting and programming, 4–53, 4–63
synchronous ports, 4–39
wide data clocked, 4–57 to 4–66
writes, 4–30

filter analysis, low-pass, 1–20

filtering, high-frequency, 1–31

fine-grain logic cell, 3–138

flags
boundary, 4–32
in clocked FIFOs, 4–31

FLASH370, 3–213
designing with *Warp2*, 3–102 to 3–120
family members, 3–104
features, 3–103 to 3–109

FLASH371, 3–180

flip-flops, triggering modes, *3–2*

FOTO, 5–17, 5–73

Fourier series expansion, 1–3

Fourier transform, 1–32

FPGAs, 5–108
architecture and technologies, 3–137
architecture issues, 3–137
comparison to CPLDs, 3–144
design example, 3–156
designing with, 3–149
detailed architecture, 3–137
global architecture, *3–138*
I/O cells, 3–147
logic cells, 3–138
programmability, 3–138

frames, in video signal generator, 2–5

framing, 5–208, 5–215

frequency, vs. operating supply current characteristic, 1–41 to 1–42

function attributes, 3–187

fuse technology
characteristics, 3–142
CY7C380 Family, 3–144

Gate Array ASIC, 3–137

generator
clock, 3–29 to 3–30, 3–36 to 3–41
using CY7C361, 7–131 to 7–132
interrupt, 7–4
substrate bias, 1–2, 1–58
global synchronous set, 3–220, 3–223
graphical user interface, 3–73
groups, 3–189
gss, 3–220, 3–223
hardware, semaphores, 4–11 to 4–12
HBLANK signal, 2–7
HC CMOS, 1–54
driving TTL, 1–55
HCT CMOS, 1–54–1–55
Hewlett-Packard, HSMS–2822 Schottky diode, 1–23
hierarchical designs, 3–77
high-level architecture, VIC068/VAC068, 7–32
higher-level controller, 3–122
Horstmann, Jens U., 3–5
HOTLink, 5–1, 5–92, 5–105, 5–128, 5–208, 5–214
See also CY7B923 and CY7B933
BIST
auto-abort and restart, 5–71
tests using, 5–71
receiver jitter tolerance, 5–72
transmission line length, 5–71
built-in self-test, 5–62–5–78
Bypass mode, 5–64, 5–66
components, 5–40
bypass capacitors, 5–41
guidelines, 5–44, 5–45
receiver power pins, 5–42
transmitter power pins, 5–41
types, 5–42
clock oscillators, 5–40
requirements, 5–41
copper cables, 5–54
ANSI Fibre Channel requirements, 5–55
coaxial cable, 5–54
connectors, 5–56
shielded twisted pair, 5–54
twinaxial cable, 5–54
fiberoptic cables, 5–52
ANSI Fibre Channel requirements, 5–53
multimode fiber, 5–52
optical pulse dispersion, 5–53
single-mode fiber, 5–52
fiberoptic detectors (receivers), 5–49
power distribution requirements, 5–49
fiberoptic emitters (drivers), 5–47
ANSI Fibre Channel requirements, 5–47
power distribution requirements, 5–48
optical modules, 5–50
pulse transformers, 5–51
ANSI Fibre Channel specifications, 5–51
core materials, 5–51
resistors, 5–46
design consideration, 5–1
direct memory access model, 5–131
DMA protocol definition, 5–131
ECL inputs, 5–14
ECL outputs, 5–12
differential connections, 5–13
single-ended connections, 5–12
Encoded mode, 5–64, 5–66
features, 5–1
functional description, 5–1
high-speed serial links, 5–128
I/O space model, 5–130
implementing a data link, 5–129
introduction, 5–92
normal RDY timing, 5–208
parallel cyclic redundancy check, 5–105
CRC–16 and CRC–32, 5–105
CRC–16 parallel algorithm
description, 5–107
implementation, 5–107
CRC–32 parallel algorithm
description, 5–108
implementation, 5–109
development of parallel algorithm, 5–106
parity, 5–105
parallel interface
receiver, 5–129
transmitter, 5–129
RDY and CKR stretching, 5–210
RDY in BIST mode, 5–211
BIST loop, 5–211
entering BIST mode, 5–211
framing while in BIST, 5–212
leaving BIST, 5–211
start of BIST, 5–211

RDY in bypass mode, 5–209
entering framing, 5–209
leaving framing, 5–210
normal operation, 5–209
RDY in encoded mode, 5–208
entering framing, 5–208
leaving framing, 5–209
normal operation, 5–208
RDY pin description, 5–208
receiver, 3–121
biasing, 5–29
PECL optical modules, 5–29
receiving from copper media, 5–30
standard ECL optical modules, 5–30
BIST comparator, 5–68
block diagram, *5–63*
ECL inputs, 5–28
controlling A/B from TTL, 5–29
levels, 5–29
error-free-window test, 5–73
offset frequency, 5–77
run-length tolerance test, 5–74
serial interfaces, 5–129
serial signal characteristics, 5–6
ECL basic switch, 5–6
ECL signal level reference, 5–6
ECL signal levels, 5–7
inputs, 5–7
outputs, 5–7
emitter–follower, 5–7
sample ECL waveforms, 5–10
viewing ECL signals, 5–8
oscilloscope bandwidth, 5–8
probe grounding, 5–9
probing from V_{CC}, 5–9
probing from V_{EE}, 5–10
scope probes, 5–8
shared memory I/O model, 5–134
simplifying your system with, 5–81
built-in self-test, 5–85
DC specification, 5–87
ECL-to-TTL translator, 5–87
higher operating frequency, 5–85
more flexible command codes, 5–82
more inputs, 5–81
more outputs, 5–81
multiplexed command and data, 5–81
output enable considerations, 5–88
parallel interface, 5–87
reframing, 5–84
sending violations, 5–87
status indication, 5–89
transmitter
BIST generator, 5–66
block diagram, *5–62*
connections, 5–16
FOTO control, 5–17
power saving mode, 5–16
upgrade your TAXI–275, 5–79

HOTLink demo board, 5–141
assembly and options, 5–176
BIST support hardware, 5–176
JP2, 5–176
OLC-compatibility registers, 5–176
optical module, 5–176
oscillator, 5–176
transmitter, 5–176
voltage monitor, 5–176
BIST support, 5–142
block diagram, 5–141
board connector, 5–141
clock signals, 5–153
coaxial cables, 5–142
configuration settings, 5–154
JP1 jumper block, 5–154
receiver reference clock source select, 5–155
receiver source loopback select, 5–155
transmitter clock source select, 5–155
transmitter FOTO source select, 5–155
transmitter mode select, 5–155
S1 option select switch, 5–156
byte sync polarity, 5–157
carrier detect polarity, 5–157
enable next parallel transmitter data, 5–156
enable parallel transmitter data, 5–156
encoder mode select, 5–156
receiver A/B port select, 5–157
receiver BIST enable, 5–157
receiver mode select, 5–157
transmitter BIST enable, 5–156
transmitter FOTO enable, 5–157
connector pin numbering, 5–143
JP2–58-position pin-header, 5–143
JP3–60-position board-edge, 5–144
JP4-OLC-Compatibility connector, 5–144
connector pinouts, 5–145
control signals, 5–153

copper cable connectors, 5–177
CY9266 schematic, 5–158
data signals, 5–154
Design criteria, 5–142
external serial interface connections, 5–171
  CY9266-F serial interface connections, 5–171
    fiberoptic connector, 5–172
    optical modules, 5–171
  CY9266-C serial interface connections, 5–173
    coaxial board connectors, 5–173
    coaxial cable connectors, 5–173
  CY9266-T serial interface connections, 5–173
    STP board connectors, 5–174
    STP cable connectors, 5–174
OLC mode configuration, 5–175
  JP1 settings, 5–175
  S1 settings, 5–175
optical modules, 5–142
overview of, 5–141
power signals, 5–153
redesign capability, 5–177
shielded-pair cables, 5–142
signal descriptions, 5–146
signal naming conventions, 5–146
status signals, 5–153
switched control signals, 5–153
theory of operation, 5–158
  BIST and support hardware, 5–170
    BIST state machine, 5–170
    error display, 5–171
    other BIST PLD functions, 5–171
  copper to receiver serial interface, 5–164
    copper carrier detect, 5–165
  optical module to receiver interface, optical module signals, 5–163
  optical module to receiver serial interface, 5–163
    receiver port select, 5–163
  optical module to receiver signal interface, receiver data inputs, 5–163
  receiver parallel interface, 5–165
    OLC compatibility registers, 5–166
    receiver clocks, 5–167
    receiver control inputs, 5–168
    receiver parallel data output, 5–165
    receiver status outputs, 5–169
  transmitter parallel interface, 5–158
    transmit data bus, 5–158
    transmitter clocks, 5–160
    transmitter control signals, 5–159
  transmitter to copper cable serial interface, 5–162
  transmitter to optical module serial interface, 5–161
HOTLink transmitter, terminating ECL signals, 5–23
  driving copper media, 5–26
    capacitor coupled, 5–27
    direct coupled, 5–26
    transformer coupled, 5–27
  driving optical modules, 5–24
    PECL, 5–24
    standard ECL, 5–25
  OUTA and OUTB, 5–24
  OUTC, 5–23
HSYNC signal, 2–7
ICGS, 7–3
ICMS, 7–3
identifiers, 3–188
idle decoder, 4–61
impedance
  AC, 1–4
  input or characteristic, 1–4
  mismatch, 1–2
  surge, 1–4
in-circuit testing, using scan mode, 3–177 to 3–179
input
  buffers, 1–36, 1–55
  characteristics of Cypress circuits, 1–49 to 1–62
  clamping diodes, 1–2, 1–49 to 1–50, 1–51
  current, 1–55 to 1–56
  impedance, 1–4
  sensitivity, 1–1
input clocking scheme, 3–76
input macrocell, *3–74*
input register logic definition, 3–223
input/output macrocell, *3–75*
instruction set, SmartCAM, 9–7
Integrated Device Technology, slave companion part to dual-port family, 4–4
Intel
  80386 processor, 2–34

80486 processor, 2–36
with CY7B991/2, 6–53 to 6–76
960CA processor, 2–39
Intellec 8/MDS PROM programming file format, 2–22 to 2–23
Intellec 86 PROM programming file format, 2–23 to 2–24
interconnect, advantages and weaknesses, 3–143
interface
for VIC068A, 7–4
products for buses, 1–61
interlacing, 2–5
internal feedback, in state machine PROM, 2–3 to 2–4
internal signal declarations, 3–221
interrupt controller, 3–207
features, 3–207
interrupt generator, 7–4
interrupt register, block diagram, *3–209*
interrupt register block, 3–207
interrupts, 3–29
in VIC068A, 7–12
logic in dual-port RAM, 4–7
IS_TYPE attribute, 3–183
JEDEC file, 3–76
jitter, 5–34, 5–72
deterministic, 5–34, 5–72
data dependent, 5–35, 5–72
duty cycle distortion, 5–36, 5–72
random, 5–36
jitter tolerance, receiver, 5–72
K' MOS circuit design parameter, 1–1
K28.5, 5–3, 5–68, 5–209
keyword, 3–181, 3–185
LAB, 3–102
architectural components, 3–102
latch option, 3–111
latch-up, 1–2
causes, 1–57
conditions for, 1–57
definition of, 1–56
prevention of, 1–58
latches, on CY7C270, 2–33
lead inductance, 1–31
LFI (link fault indicator), 5–222
LFSR, 5–5, 5–66, 5–68
library, 3–152, 3–220
line voltage, for a step function, 1–7–1–9
load
capacitance, estimating, 1–6
multiple, 1–18
lockvariable, 4–3
lockword, 4–3
LOG/iC, 2–9, 2–30
clock state machine, source code, 3–42
comparator PROM, source code, *2–30*
horizontal state machine, source code, 2–13
vertical state machine, source code, 2–16
Logic
DTL and TTL families, 1–50
product characteristics, 1–40
logic cell
in FPGAs, 3–137
position designator, 3–177
logic cells, advantages and weaknesses, 3–143
logic synthesis, 3–192
low-pass filter analysis, 1–20 to 1–21
Lubkin, S., 3–2
MACH, 3–180
macrocell, 3–103
buried, 3–103
dedicated, 3–103
hidden, 3–76
input, *3–74*
input/output, *3–75*
mailbox signaling, in dual-port RAMs, 7–3 to 7–4
mask logic, 3–211
mask register, 3–210
block diagram, *3–210*
master, standalone operation of dual-port RAMs, 4–13
master device, 7–31 to 7–34

master read, 7–33
master sequencer, 7–35
master write, 7–33
MC68020, 7–4
  See also see 68020
MCS86 PROM programming file format, 2–23–2–24
Mealy machine, 3–82, 3–222
Mealy machines, 2–2, 2–4, 3–29, 3–238
  Binding Architecture file, 3–254
    using state table method, 3–265
  combining Moore and Mealy outputs, 3–239
  Design Entity file, 3–250
    using state table method, 3–263
MECL families, 1–51
memory
  dual-port. *See* dual-port RAMs
  exception cycles, 7–125–7–126
  implicit, in VHDL, 3–231
  multi-port, history of, 4–1
  pipelined, state machine PROM as, 2–5
message passing, 4–3
metastability, 3–1 to 3–24
  attacking, 3–4 to 3–5
  causes of, 3–3
  characteristics, of Cypress PLDs, 3–17
  characterization, 3–9
  circuit analysis, 3–5 to 3–7
  data on, 3–8
  definition of, 3–1
  explanation of, 3–2 to 3–3
  graphs of Cypress devices, 3–19, 3–20
  information from manufacturers, 3–9 to 3–10
  statistical analysis, 3–7 to 3–8
  testing
    of Cypress parts, 3–10 to 3–16
    PLD equations for, 3–14, 3–15
microprocessor, typical 8-bit, *4–14*
microstrip line PCB construction, 1–16 to 1–17, 6–8
MIPS, R3000 processor, with CY7B991/2, 6–53 to 6–76
mixed mode, 3–151
Moore machine, 3–222
Moore machines, 2–2, 2–4, 3–29, 3–236 to 3–237
  Binding Architecture file, 3–245
    for compact machine, 3–249
    using state table method, 3–262
  combining Moore and Mealy outputs, 3–239
  Design Entity file, 3–241
    for compact machine, 3–246
    using state table method, 3–260
Motorola
  68020, and the VIC068A, 7–6 to 7–12
  68040 processor, 2–37 to 2–39
  88100 processor, 2–45 to 2–47
  ECLiPSE family, 1–51
  Exorcisor PROM programming file format, 2–21 to 2–22
  Exormax PROM programming file format, 2–22
  MBD101, MBD102 Schottky diodes, 1–23
  MECL families, 1–51
Motorola 68040, 7–84
MTBF, 4–40
multi-port, memories, 4–1
multiplexer, 3–191
  designing with VHDL, 3–79 to 3–80
  Warp2 report file excerpt, 3–92
  Warp2 source code, 3–91
multipliers, designing with PROMs, 2–59 to 2–65
mux based interconnect, 3–103, *3–105*
negative undershoot safety margin, 1–2
networks, RC, 1–20
NMOS, output pull-up transistors, 1–58
NMOS ICs, replacing with CMOS, 1–1
nodes, bidirectional, 1–18
noise immunity, definition of, 1–52
NONPIPELINED RUN mode, 3–28
NRZ, 5–37, 5–83
NTSC TV systems, 2–5
number representations, 3–188
OLC receiver, 5–214
one-hot, 3–222
operating supply current
  vs. frequency characteristic, 1–41

worst-case, 1–41
operator, 3–184
operators, 3–181
optimization, 3–154, 3–155
output
buffers, 1–37
characteristics of Cypress circuits, 1–49 to 1–62
clamping diodes, 1–50
combining Moore and Mealy outputs, 3–239
pins, 1–58
output macrocell, 3–106
ownership, of data, 4–3
package, 3–220
packer, 3–155
PAL TV systems, 2–5
PAL16L8, on a VIC068A board without a processor, 7–18
PAL22V10
cycle decoding, *7–11*
fitting a clock state machine into, 3–38
in CY7C611A interface, 7–131
MTBF calculation, 3–8
on a VIC068A board without a processor, 7–17
PAL22V10C, 7–17
using CY7B991 to adjust clocks to, 6–56
PALC16L8, in unterminated line example, 1–23
PALs
difference from PLAs, 2–19
product characteristics, 1–39
parallel AC termination, 1–20
pASIC, 3–152
pASIC380, 3–177, 3–213
calculating power for, 1–42 to 1–48
PCBs
component placement, 1–1
construction
microstrip lines, 1–16, 6–8
strip lines, 1–17, 6–9
wire over ground, 1–16
modern, 1–17
trace inductance and current-starving, 1–31
traces, 1–2
transmission lines, 1–3
using ground or power planes, 1–2
PECL, 5–1
phase changes in received data, tolerance to, 5–77
phase-locked loop, operation, 6–3, 6–17
PIM, 3–102, 3–103
pin definitions, scan path, 3–178
pin-to-pin propagation delay, 3–142
pipelined buffer
designing with VHDL, 3–77 to 3–78
VHDL source code, 3–87
Warp2 report file excerpt, 3–88
PIPELINED RUN mode, 3–28
pipelines
freezing, 7–129
NONPIPELINED RUN mode, 3–28
nonpipelined states, 3–32
pipeline register to interface CY7B923, 4–52
PIPELINED RUN mode, 3–28
pipelined states, 3–32
registers, 4–62
state machine PROM used as pipelined memory, 2–5
place and route, 3–155
PLAs, difference from PALs, 2–19
PLD ToolKit
18G8 design file, source code, *7–148*
control PLD, source code, 7–25, 7–26
controller/arbiter PLD, source code, 7–27, 7–28, 7–29, 7–30
metastability testing, source code, *3–14*, *3–15*
selection PLD, source code, 7–23, 7–24
PLDs
design tools, 2–25 to 2–30
metastability, 3–1 to 3–24
characteristics, 3–17
PLL, 3–121, 5–1
out of lock condition, 3–122
PN junction diodes, 1–2
forward voltage threshold, 1–51
polarity conventions, 3–188
ports
asynchronous, 4–40
synchronous, 4–39

power
  active, 1–42 to 1–45
  calculating, obtaining values, 1–46
  calculating for pASIC380, 1–42 to 1–48
  DC, 1–35
  dissipation
    model, 1–35 to 1–37
    sources, 1–34
  estimation, 1–45 to 1–46
  frequency-dependent, 1–34 to 1–35
  static, 1–35, 1–42
  transient, 1–35
power consumption, calculation of, 3–172
power-down options, 1–35
predefined attributes, 3–187
priority encoder, 3–213
processors
  386 DXL, 2–35
  68020, 7–6–7–12
  68040, 2–37 to 2–39
  80386, 2–34
  80486, 2–36
    with CY7B991/2, 6–53 to 6–76
  960CA, 2–39
  Am29000, 2–41 to 2–43
  CY7C611A, 2–43
  CY7C88100, 2–45
  R3000, with CY7B991/2, 6–53 to 6–76
  with user-programmable interface, 2–32
product term
  sharing, 3–107
  steering, 3–107
product term allocator, 3–102, 3–107
  CY7C370, 3–107
  MACH, 3–107
  MAX7000, 3–108
product term array, 3–102
programmable
  initialize word, in state machine PROM, 2–4
  logic elements, 2–19
programmable connections, 3–140
programmablility, FPGAs, 3–138
PROMs, 1–59 to 1–60
  as video signal generator, 2–5
  designing with, 2–59 to 2–65
  generating programming files, 2–19 to 2–31
  high-speed asynchronous, 2–5
  output-registered, 2–4
  processor-intelligent, 2–32 to 2–48
  product characteristics, 1–39
  programmers, compatibility, 2–20
  programming file formats
    ASCII Binary, 2–20
    DEC, 2–21
    Exorcisor, 2–21 to 2–22
    Exormax, 2–22
    Intellec 8/MDS, 2–22 to 2–23
    Intellec 86, 2–23 to 2–24
    simple Binary, 2–20
    TEK HEX, 2–24
    XTEK, 2–24–2–25
  used as state machines, 2–2, 3–38
    See also CY7C258/9
    advantages, 2–2
    architectural features, 2–3 to 2–4
    versatility, 2–4 to 2–5
propagation delays, ALU, 2–59
propagation velocity and delay, 1–5
pull-down, output pull-down devices, 1–54
pull-up
  NMOS output pull-up transistors, 1–58
  output pull-up characteristics, 1–54
  terminations, 1–19
pull-down, terminations, 1–19
pulse response, 1–9
R3000 processor, with CY7B991/2, 6–53 to 6–76
RAM
  Cypress dual-port family, 4–5
  dual-port, 4–1 to 4–19
    applications, 4–2 to 4–4
    operation, 4–6 to 4–7
  dual-port RAM cell history, 4–4
  product characteristics, 1–38
  single-port, 4–2
  virtual dual-port, 4–2 to 4–3
range attributes, 3–187
RC networks, 1–20
RCA 4000 series, 1–51
RDY pin, 5–208

real-world converted designs, 3–192

receiver, HOTLink. *See* CY7B933

reference crystal
  changing requirements, 6–82
  crystal specifics, 6–82
    Epson America #MA-506 14.313M-B, 6–82
    Fox Crystal #FPX143–20, 6–82
  drive level precautions, 6–82
  error budget analysis, 6–82
  high-accuracy, 6–82

reflection
  coefficients, 1–6
  conditions for, 1–5
  due to discontinuities, 1–1 to 1–2, 1–9, *1–11*, 1–11, 1–14 to 1–15
  multiple, 1–14 to 1–15

reframe, CKR stretch, 5–76

reframe controller, 3–121
  additional functionality, 3–122 to 3–123
  counters, 3–125
  decoding function, 3–123
  design and implementation, 3–123
  inputs, 3–123
  interface, 3–123
  outputs, 3–124
  receiver system, 3–123

Reframe input, 4–51, 4–61

reframing, why necessary, 3–121 to 3–122

registers, 9–2
  bringing registers on-chip, 4–1
  CYB675 local control, 7–69
  exclusive state, 3–36
  on CY7C270, 2–33
  pipeline, 4–62
  semaphore, 4–4
  SmartCAM, 9–2

repeat instruction, 3–29

repetitive logic, 3–191

RESET signal, 3–207

resets and presets, 3–188

retransmit feature, 4–23

RISC, and CY7C270, 2–39

rise time
  effect on waveforms, 1–13
  finite, effects, 1–11

RoboClock, 6–41
  See also CY7B991
  configuration methodologies, 6–42
    using one small table, 6–42
    using three tables for multiple outputs, 6–43
  overview, 6–41
  using in resolution enhancement of a laser printer, 6–44
    background, 6–44
    configuring, 6–46
    design analysis, 6–46
    design implementation, 6–45

routing, 3–145
  signal wires supported, 3–145

RTL, 3–73, 3–217
  circuit fabrication, 1–49
  transfer characteristic, 1–52

running disparity, 5–71, 5–83
  error, 3–123

RVS, 3–122, 3–125, 5–71, 5–85

S records. *See* Exorcisor PROM programming file format

SC/D, 5–92

scan mode, using on pASIC380, 3–177

scan path
  electrical characteristics, 3–178
  pin definitions, 3–178
  timing diagram, 3–179

scanning, 2–5

schematic entry, 3–151, 3–154

Schottky diode termination, 1–22

SCLK, 3–177

sea-of-gates, 3–137, 3–139
  ASIC, 3–139

SECAM TV systems, 2–5

Seeds, Dr. Bob, 1–50

semaphores
  latch cell, *4–12*
  hardware, 4–11 to 4–12
  registers, 4–4

sequential statements, 3–224

serial decoder
  designing with VHDL, 3–82
  VHDL source code, 3–98
serial scan path, 3–177
Series 7400, 1–51
series damping, 1–18 to 1–19
set-up and hold time mismatches, 6–56
shared input multiplexer, 3–80
shift register, 3–191
SI, 3–177
signal characteristics, copper, 5–32
signal skew, 5–13
signals
  680x0 basic control, 7–128
  blanking, *2–7*
  bus interface, 7–14
  BUSY, in dual-port RAMs, 4–10
  CY7C361, input and output, 7–131
  CY7C611A, memory interface, 7–126
  CY7C964
    address comparison, 7–145
    local, 7–145
  HBLANK, 2–7
  HSYNC, 2–7
  synchronizing, *2–7*
  transition times, 1–6
  VBLANK, 2–7
  VIC64 control, 7–127, 7–144
  VMEbus control, 7–17, 7–144
  VSYNC, 2–7
simple binary PROM programming file format, 2–20
single-port, RAM for dual-port memory, 4–2
skew
  extrinsic, 6–3
  intrinsic, 6–2
slave
  standalone operation of dual-port RAMs, 4–15
  word-width expansion, 4–13
slave device, 7–31 to 7–35
SmartCAM, 9–1
  ALUs and row logic, 9–3
    GPLB, 9–4
    match latch, 9–5
    multiple response resolver (MRR), 9–5
    response register R1, 9–4
    response register R2 and R3, 9–4
  applications, 9–8
    database join, 9–8
    image cleanup, 9–9
  architecture, 9–2
  block diagram, 9–2
  cascading, 9–13
    generating system inputs, 9–16
    generating system outputs, 9–14
    initializing a SmartCAM array, 9–19
    monitoring the RDY output, 9–14
    R1 daisy chain, 9–13
  defined, 9–1
  difference from conventional memories, 9–1
  error handling, 9–8
  hardware support, 9–10
  instruction set, 9–7
    MATCH, 9–8
    MOVE, 9–7
    NOP, 9–7
    READ, 9–7
    READSHIFT, 9–7
    READSNEXT, 9–7
    SHIFT, 9–7
    SNEXT, 9–7
    WRITE, 9–8
    WRITECOL, 9–8
    WRITESHIFT, 9–8
    WRITESNEXT, 9–8
  interface and control registers, 9–2
    command register, 9–3
    control/status register, 9–3
    data in register, 9–3
    data out register, 9–3
    mask register, 9–3
  modes of access, 9–5
  operation, 9–5
    match operations, 9–5
      bit match, 9–5
      quad match, 9–5
  overview, 9–1
  pattern matching, 9–21
    general information, 9–21
    motion estimation application, 9–27
      operation, 9–28
      summary, 9–30
    optical character recognition, 9–25

operation, 9–26
overview, 9–25
quad encoding, 9–25
SmartCAM data structure, 9–26
text string application, 9–22
CAM data structure, 9–22
matching text strings, 9–23
operation, 9–22
overview, 9–22
software support, 9–10
state machine, 9–5
timing considerations, 9–12
adding idle state when a write follows a read, 9–13
using watch dog timer for error recovery, 9–12

SO, 3–177

soft errors, 1–2

SONET serial transceiver, 5–221
carrier detect and link fault indicator, 5–222
interfacing IgT WAC-013, 5–226
interfacing with PM5345, 5–223
introduction to, 5–221
loop back testing, 5–223
operating frequency, 5–221
power-down modes, 5–223
receive functions, 5–222
transmit functions, 5–222

SONET/SDH, 5–221

source code
ABEL
comparator PROM, *2–28*
generating LE and IRDY for Am29000, 2–50
PALC22V10 cycle decoding, *7–11*
LOG/iC
clock state machine, 3–42
comparator PROM, *2–30*
horizontal state machine, 2–13
vertical state machine, 2–16
PLD ToolKit
18G8 design file, *7–148*
control PLD, 7–25, 7–26
controller/arbiter PLD, 7–27, 7–28, 7–29, 7–30
metastability testing, *3–14*, *3–15*
selection PLD, 7–23, 7–24
VHDL
compact Moore machine, 3–246
Mealy machine, 3–255
using state table method, 3–263, 3–266
Moore machine, 3–241, 3–250, 3–255
using state table method, 3–260, 3–266

source level design verification, 3–152

SPARCmon, 7–70

SpDE path analyzer, 3–163, 3–165

spectrum analyzer, 5–37

SST, 5–221

standalone operation of dual-port RAMs
master, 4–13
slave, 4–15

state machines, 3–27, 3–125, 3–158, 3–190, 3–217, 3–224
as interface controller for CY7B923, 4–50
as receivers, 4–54–4–55
clock generation, 7–131
CPU inactive states, 3–32
CY7C361 implementation, 3–39
D flip-flop implementation, 3–37
entry methods, 3–27
exclusive registers, 3–36
horizontal, 2–8, *2–10*, 2–12, 2–13, 2–14, 2–15
LOG/iC PLD source code, 3–42
naming states, 3–31
partitioning, 3–31
parts of, 2–1 to 2–2
pipelined and nonpipelined states, 3–32
PLD implementation, 3–38
PROM implementation, 2–2, 3–38
requirements of, 2–1 to 2–2
state definitions, 3–222
synchronous vs. asynchronous, 3–29
T flip-flop implementation, 3–37 to 3–38
terms used, 3–27
unique states, 3–32
vertical, 2–8, *2–11*, 2–12, 2–16 to 2–21

state macrocell, 3–106

state tables, 3–27, 3–73

state vector, 2–1 to 2–2

static power, 1–35, 1–42

step function
determining line voltage for, 1–7 to 1–9
negative step function response, 1–21

positive step function response, 1–21
response for various terminations, *1–10*
response of ideal line, 1–9
strip lines, 1–17–1–18, 6–9
strobe, shortening considerations, 1–27 to 1–29
structural logic description, 3–150
substrate bias generator, 1–2
supervisor mode, decoding on the VMEbus, 7–10
surge impedance, 1–4
switches
ICGS, 7–3
ICMS, 7–3
switching characteristics, ECL, 1–61–1–63
SY2130, 4–4
SYNC, 5–3, 5–84, 5–90, 5–209
synchronization
in video signal generators, 2–5
two-stage, 3–16
synchronizing signals, *2–7*
Synertek, 4–4
synthesis, 3–155
SYSFAIL generation, 7–4
tag. *See* cache-tag RAMs
TAXI–1 devices
description of, 5–93
modifying the system for HOTLink, 5–101
asynchronous strobing, 5–102
data words longer than 8 bits, 5–102
mapping command codes, 5–102
multiplexed command and data, 5–101
operating frequency, 5–102
replacement of, 5–93
simplifying with HOTLink, 5–93
ability to send violations, 5–99
ability to turn-off serial output stream, 5–100
better DC specifications, 5–99
built-in self-test, 5–97
ECL to TTL translator, 5–101
higher operating frequency, 5–97
loop-back testing capabilities, 5–99
more robust reframing capabilities, 5–96
more serial inputs, 5–94
more serial outputs, 5–94
multiplexed command and data, 5–94
simplified synchronous interface, 5–99
superior data encoding, 5–96
TAXI–125, 5–92
TAXI–175, 5–92
TAXI–275
receiver, block diagram, *5–81*
transmitter, block diagram, *5–80*
upgrade with HOTLink, 5–79, 5–80
brief explanation, 5–80
TAXIchip, 5–92
technology mapper, 3–155
TEK HEX PROM programming file format, 2–24
terminations
pull-up/pull-down, 1–19–1–20
Schottky diode, 1–22
types of, 1–18–1–20
Test and Set instruction, 4–3–4–4
Texas Instruments
SN74S1050/52/56 Schottky diodes, 1–23
SN74S1051/53 Schottky diodes, 1–23
TMS320C30, 2–47 to 2–48
timing model, 3–142
TMS320C30, 2–47 to 2–48
TMS320C40, 7–31
architecture, 7–31
TMS320C50, memory maps, 2–57
TMS320C5X, 2–56 to 2–58
DSP to memory interface, 2–57
initialization, 2–56
timing, 2–57
external program memory, *2–57*
top-down design methodology, 3–150, 3–207
traces, most critical, 1–1
transient power, 1–35
transistors, NMOS output pull-up, 1–58
transmission line termination, 5–22
parallel termination, 5–23
series termination, 5–23
transmission lines

coaxial cable, 1–16
effects, 6–10
energy considerations for driving, 1–7
ideal, 1–3–1–4, 1–7
microstrip lines, 1–16, 6–8
model, *1–3*
pulse response, 1–9
strip lines, 1–17, 6–9
termination, 6–12
termination strategies, 1–18
theory of, 1–3
twisted pair, 1–16
types of, 1–16 to 1–17
types of terminations, 1–18
unterminated, 1–23 to 1–24
when to terminate, 1–17
wire over ground, 1–16

transmitter, HOTLink. *See* CY7B923

triout components, 3–78, 3–111

truth table, 3–213, 3–222

TTL, 1–53, 5–29
driving HC CMOS, 1–55
families, 1–50
operation of TTL gate, 1–53
output characteristics, 1–54
problems and performance, 1–53 to 1–54
voltage levels, 1–54

twisted pair PCB construction, 1–16

type attributes, 3–187

up/down counter
designing with VHDL, 3–80 to 3–82
Warp2 report file excerpt, 3–97
Warp2 source code, 3–95

user mode, decoding on the VMEbus, 7–10

value attributes, 3–187

VBLANK signal, 2–7

VHDL, 3–73, 3–77, 3–121, 3–130, 3–150, 3–180, 3–217, 4–39
compact Moore machine, source code, 3–246
component, 3–269
configurable components, 3–270
hierarchical design, 3–269
library, 3–269
Mealy machine
source code, 3–250, 3–255
using state table method, source code, 3–263, 3–266
Moore machine
source code, 3–241, 3–255
using state table method, source code, 3–260, 3–266
multiplexed dual counter design, 3–270
multiplexed quad counter design, 3–271
package, 3–269
source level debugger, 3–161
special type conversion, 3–189
techniques for design fitting, 3–231 to 3–235
to describe state machines with *Warp2*, 3–236 to 3–268
top-down design methodology, 3–207
vs. Abel–HDL, 3–219

VHDL–ABEL
dot extension, 3–182
special constants, 3–181

vialink, 3–144

VIC068/VAC068, 7–31
interfacing to TMS320C40, 7–31
design requisites, 7–31
design goals, 7–31
high-level architecture, 7–32
hardware description, 7–33
address bus decoding, 7–33
bus control, 7–34
master bus cycle generation, 7–34
reset circuitry, 7–33
slave bus cycle generation, 7–35
VIC068/VAC068 software initialization, 7–35

VIC068A
and the MC68020, 7–6 to 7–12
capabilities, 7–13 to 7–14
features, 7–1 to 7–5
interfacing, 7–4
interrupts, 7–12
reset operations, 7–6
slave access to R/W, 7–20
used with CY7C964, 7–138 to 7–149
without a microprocessor, 7–13 to 7–30

VIC64, 7–84
address spaces, 7–72 to 7–73
architecture, 7–84, 7–87
asynchronous bus protocol, 7–84
configuration, 7–72
control signals, 7–127

CYB675 evaluation board, 7–69
CYB675 local control register, 7–69
deadlock, 7–96, 7–97, 7–98, 7–99, 7–100
initialization, 7–71 to 7–75
interfacing with the CY7C611A, 7–125 to 7–137
Motorola interface to, 7–84
  bus arbitration
    68040 request for VIC64 bus access, 7–91
    bus arbitration state machine, 7–90
    sample arbitration timing diagrams, 7–92
    VIC64 bus requests, 7–92
  design assumptions, 7–86
    68040 configured for large buffer timing mode, 7–88
    memory system design, 7–87
    shared memory, 7–88
    two memory banks architecture, 7–87
  design issues, 7–84
    asynchronous bus to synchronous bus interfacing, 7–84
    bus contention, 7–84
    putting VIC64 on 68040's bus, 7–84
    slave access implementation, 7–86
    solving bus contention with arbitration, 7–85
  Interrupt acknowledge cycles, 7–100
    interrupt cycle decode, 7–103
    interrupt cycle initiation by the 68040, 7–102
    interrupt cycle termination, 7–103
    interrupt initiation from the VIC64, 7–102
    operation at reset, 7–100
    VMEbus vs. local interrupts, 7–101
  master read cycles, 7–94
    master read cycle bus error termination, 7–99
    master read cycle deadlock/retry termination, 7–97
    master read cycle initiation, 7–94
    master read cycle normal termination, 7–96
    master read cycle termination, 7–96
  master write, writepost and BLT initiation cycles, 7–99
    commonality between the various write cycles, 7–99
    write cycle bus error termination, 7–100
    write cycle deadlock/retry termination, 7–100
    write cycle initiation, 7–99
    write cycle normal termination, 7–99
    write cycle termination, 7–99
  reset circuitry, 7–88
    68040 mode selection, 7–88
    power-up or pushbutton reset, 7–88
    support for 68040 RESET instruction, 7–90
    VIC-initiated reset, 7–90
  VIC64 and CY7C964 register access cycles, 7–93
    performance of register access cycles, 7–94
    register access cycle initiation, 7–94
    register access cycle termination, 7–94
    selection of the CY7C335, 7–93
    selection of the PALC22V10, 7–93
    VIC registers vs CY7C964 registers, 7–93
  overview, 7–127 to 7–128
  reset, 7–71
  slave access implementation
    bus snooping, 7–86
    inhibiting cache transfers from shared memory, 7–86
  memory map decoding and remapping, 7–86
  software considerations, 7–69 to 7–83
  test, 7–71
  used with CY7C964, 7–138 to 7–149
video signal generator, 2–5
  schematic, *2–8*
VMEbus
  addressing, 7–73
  board with CY7C611A/VIC64, 7–130
  control signals, 7–17
  master operation, 7–9 to 7–10
  slave operation, 7–10
  support, 7–4
voltage
  definition of, 1–8
  ESD, 1–56
  line voltage for step function, 1–7 to 1–9
  reflection, 1–1 to 1–2
    coefficients, 1–6 to 1–7
    conditions for, 1–5 to 1–6
    due to discontinuities, 1–9, *1–11*, 1–14 to 1–15
VSYNC signal, 2–7
wait state requirements, 2–58
Warp,
  describing state machines, 3–236 to 3–268

designing with the CY7C370, 3–110 to 3–120
Warp2, 3–110
design flow, *3–77*
designing with, 3–73 to 3–101, 3–102 to 3–120
implementation for FIFO dipstick, 4–40
overview, 3–76 to 3–77
using for FIFO dipstick, 4–39
Warp3, 3–110, 3–149, 3–177, *5–108*
design development, 3–149
physical view, 3–177
waveforms, effect of rise time, 1–13
wire over ground PCB construction, 1–16
word-width, expansion, 4–13
write, strobe, delaying, 4–13
XTEK PROM programming file format, 2–24–2–25
zener diode, 1–30