Technical notes on using Analog Devices DSPs, processors and development tools
Contact our technical support at processor.support@analog.com and dsptools.support@analog.com
Or visit our on-line resources http://www.analog.com/ee-notes and http://www.analog.com/processors

Using Cache Memory on Blackfin® Processors

Contributed by Kunal Singh

Rev 1 – June 13, 2005

Introduction

This application note discusses cache memory management for Blackfin® processors. The document introduces popular cache schemes and then discusses the Blackfin instruction cache and the data cache in detail. Example code is provided with this application note to demonstrate cache memory management on the ADSP-BF533 Blackfin processor. These features are available on all Blackfin processors. This document assumes that the reader is familiar with basic cache terminology.

Cache Memory Concepts

This section discusses the cache memory model in general. The actual Blackfin memory model is discussed in the next section.

Memory Configuration

Systems that require huge amounts of memory generally employ a memory configuration with different memory levels. Memory at the highest level (*L1 memory*) provides the highest performance at the highest cost. Lower-level memories have multiple cycle access times, but cost less.

Cache is a high-level memory available under the control of cache controller. It cannot be accessed directly by the user. The cache controller allows larger instruction and data sections to exist in low-level memory, and code and data that are used most frequently are brought into cache (by the cache controller) and thus, are available for single-cycle access, just as though it was in L1 memory. The cache architecture is based on the fact that processor memory space has been sub-divided in to a number of fixed-size blocks (referred to as cache-lines). A block is considered to be the smallest unit of memory to be transferred from external memory to the cache memory as a result of a cache miss.

Any reference to the memory is identified as a reference to a particular block. Figure 1 depicts a memory configuration in which external memory space has been sub-divided into twenty-four memory blocks and the cache memory is divided into six blocks. (This is an example of a general memory configuration. The number of memory blocks is actually different for the Blackfin memory model.) The block size for the external memory and the cache memory is the same. Since the cache memory has a size of six blocks (in this particular example), at any time, a maximum of six data blocks of the main memory is available in the cache memory.

Block Placement

When the processor references a memory block (in the main memory) that is already available in the cache, the processor will access the data from the cache memory. This is known as a *cache hit*.

When the processor references a memory block that is not available in the cache, it is called a *cache miss*. Upon a cache miss, the cache

controller moves the referenced memory block from lower level memory to the cache memory.

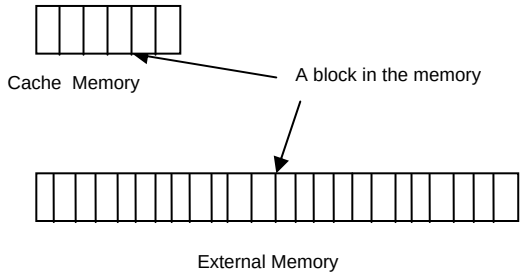


Figure 1. Memory Arranges as Fixed-size Blocks

Three schemes are used commonly for deciding the location where the incoming block may be placed in the cache memory.

Direct Mapped Cache

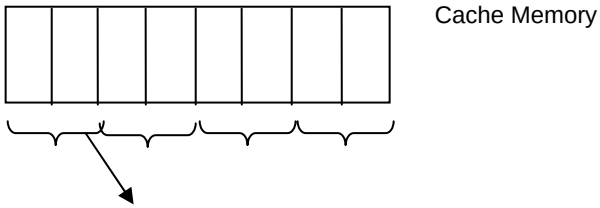
Every block in the lower-level memory has only a fixed destination block in the cache. There exists a one-to-one mapping from lower-level memory to the cache memory. This mapping is based on the address of the block in the lower-level memory.

Fully Associative Cache

A block in the main memory can replace any block in the cache memory. The mapping is completely random.

Set Associative Cache

Cache memory is arranged as sets. A set consists of a number of blocks. Any block in the lower-level memory has a fixed destination set (in which it can be placed) in the cache. The incoming block may replace any of the blocks within this fixed set. If there are *m* blocks in a set, the cache configuration is called an m-way set associative. Figure 2 depicts a 2-way set associate memory.



Set Associative Memory: A block from the external memory can be placed only in to a particular set of the cache. The block can be placed anywhere within the set.

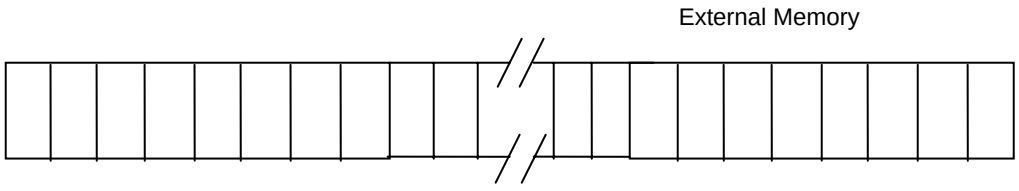


Figure 2. Configuration for a 2-Way Set Associative Cache Memory

Block Replacement

In direct mapped cache, there is always a fixed location at which a block from lower-level memory can be placed. However, with fully associative or set-associative placement, many blocks may be chosen on a cache-miss. The blocks that can possibly be replaced are called *participating blocks*. Following are some of the strategies primarily employed for selecting the block to be replaced:

(a) Random replacement: The destination block is randomly selected out of participating blocks. A pseudo-random number generator may be used to select the destination block. This is the simplest, but least efficient implementation.

(b) First in, first out (FIFO) replacement: The incoming block replaces the oldest participating block.

(c) Least-recently used (LRU) replacement: Under this scheme, all accesses to blocks are recorded. The replaced block is the one that has been unused for the longest time. LRU relies on a corollary of locality: If recently used blocks are likely to be used again, a good candidate for disposal is the least-recently used block.

(d) Modified LRU replacement: Under this scheme, every block is assigned a low priority or a high priority.

If the incoming block is a low-priority block, only low-priority blocks can participate in the cache replacement.

If the incoming block is of high priority, all low-priority blocks will participate in the cache replacement. If there are no low-priority blocks, the high-priority blocks can participate in the replacement policy.

LRU policy is used to choose the victim block among participating blocks.

Block Identification

Not all the cache blocks may have valid information contents. For example, at system startup, cache memory will contain no valid data. The cache blocks will be filled whenever a cache-miss is encountered. A mechanism must identify whether a cache block has a valid data entry in it. To identify this, a valid bit is associated with each cache block. When a cache block is filled with valid data, the corresponding valid bit is set

In addition to a valid bit, every block in the cache also has an address tag associated with it. This address tag provides the physical address of cached block in external memory. When the external memory block is referenced, the cache controller compares external address with cache address tags (with a valid bit), for the set that might contain this block. If the block address matches one of the cache address tags, it results in a cache hit.

Under a set-associative cache configuration, each memory address can be viewed as a combination of three fields. These fields are shown in Table 1. The first division is between the block address and the block offset. The block (frame) address can be further divided into the tag field and the index field.

Block Address		**Block Offset**
Tag Field	**Index Field**	
Used by the cache controller to determine a cache-hit or a cache-miss	Used to map a given bock to a particular set	Used to select a word within the given block

Table 1. Address Partitioning

The block offset field selects the desired data from the block. The index field selects the set, and the tag field is compared against it for a hit.

Write Strategies

The following section discusses two different issues with the memory write operations associated with data cache.

Write Operations with a Cache Hit

There are two basic options when writing to a cache:

- Write-Through: The information is written to both the block in the cache and to the block in the source memory.
- Write-Back: The information is written only to the block in the cache. The modified cache block is written to the main memory only when it is replaced.

To reduce the frequency of writing back blocks on replacement, a feature called dirty bit is commonly used. This status bit indicates whether the block is dirty (modified while in the cache) or clean (not modified). If the block is clean, it is not written back during replacement. Although it is application dependent, write-back mode yields about a 10-15% improvement over write-through mode. Write-through mode is best when coherency must be maintained between more than one resource (for example, the DMA controller and the processor).

Write Operations with a Cache Miss

Since the data are not needed on a write, there are two options on a write miss:

- Write Allocate: The block is allocated on a write miss, followed by the write-hit actions described above. In this scheme, write misses act like read misses.
- No-Write Allocate: Under this option, write-misses do not affect the cache. The block is modified only in lower-level memory and no caching takes place.

Blackfin Cache Model

Blackfin processors have three levels of memory, providing a trade-off between capacity and performance. Level 1 memory (also known as L1 memory) provides the highest performance with the lowest capacity. Level 2 (L2) and Level 3 (L3) memory provide much larger memory sizes, but typically have multiple cycle access times.

Blackfin processors have on-chip data and instruction memory banks, which can be independently configured as SRAM or cache. When the memory is configured as cache, the DMA controller cannot access it. All Blackfin processors have a similar cache configuration, but as a reference, the following discussion is based on ADSP-BF533 devices.

Blackfin Instruction Cache Configuration

Cache Organization

The ADSP-BF533 processor has 80 Kbytes of on-chip instruction memory. 64 Kbytes of instruction memory is available as instruction SRAM. The remaining 16 Kbytes of memory can be configured as instruction cache or can be used as instruction SRAM. Figure 3 shows the memory map of the ADSP-BF533 processor. It depicts the instruction bank configurable as instruction cache/SRAM.

When enabled as cache, the instruction memory works as a 4-way set associative memory. Each of the four ways can be locked independently. The instruction cache-controller can be configured to use the modified LRU policy or the LRU policy for block replacement.

The 16-Kbyte cache is arranged as four 4-Kbyte sub-banks. Each sub-bank consists of 32 sets (each set is four blocks). The block size (to be read on a cache miss) is 32 bytes. The external read data port (for cache controller) is 64 bits (8 bytes) wide. Hence, the complete block is read (by the cache controller) as a burst of four eight-byte-wide chunks of data.

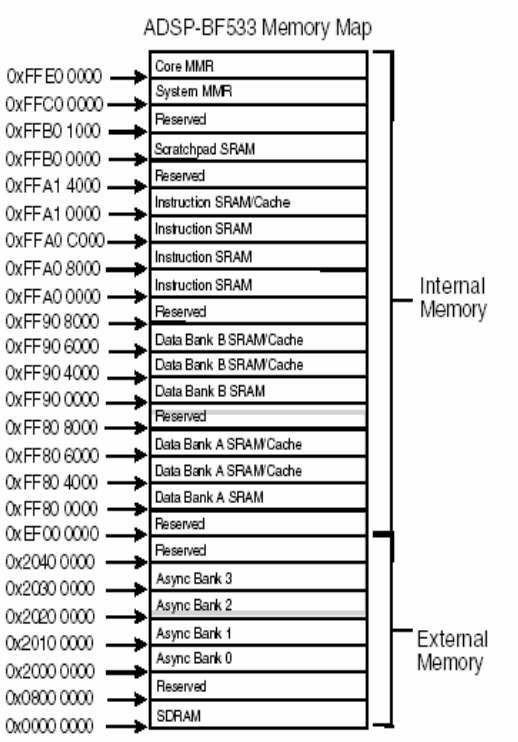


Figure 3. ADSP-BF533 Memory Map

Each block has a tag portion associated with it. The tag portion consists of four parts – Address tag, LRU state, LRU priority, and valid bit. Figure 4 depicts the bit fields of a tag section in a cache block. In the following discussion, the words *cache-block* and *cache-line* are used interchangeably.

Figure 5 shows how the 32-bit address space is mapped to cache memory space.

Figure 6 shows how a 4-Kbyte sub-bank in the instruction cache is arranged.

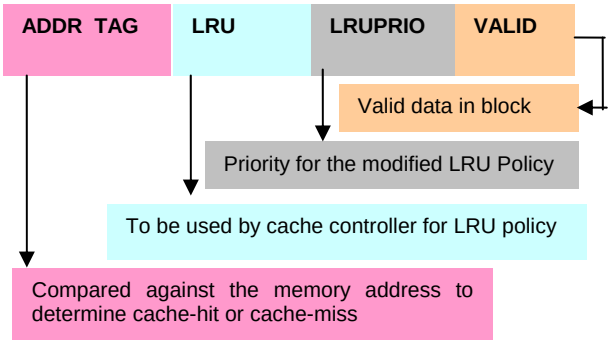


Figure 4. Tag Section of an Instruction Cache Block

Cache Hits/Misses and Cache Line Replacement

A cache hit is determined by comparing the upper 18 bits and bits 11 and 10 of the instruction fetch address to the address tag of valid lines currently stored in a cache set. If the address-tag compare operation results in a match, a cache hit occurs. If the address-tag compare operation does not result in a match, a cache miss occurs.

Consider an access to the address `0x10374956`. This address is mapped to set-10 of sub-bank 0. The upper 18 bits and bits 11-10 of this address (this forms the tag word) will hence be compared against all the valid tags of set-10.

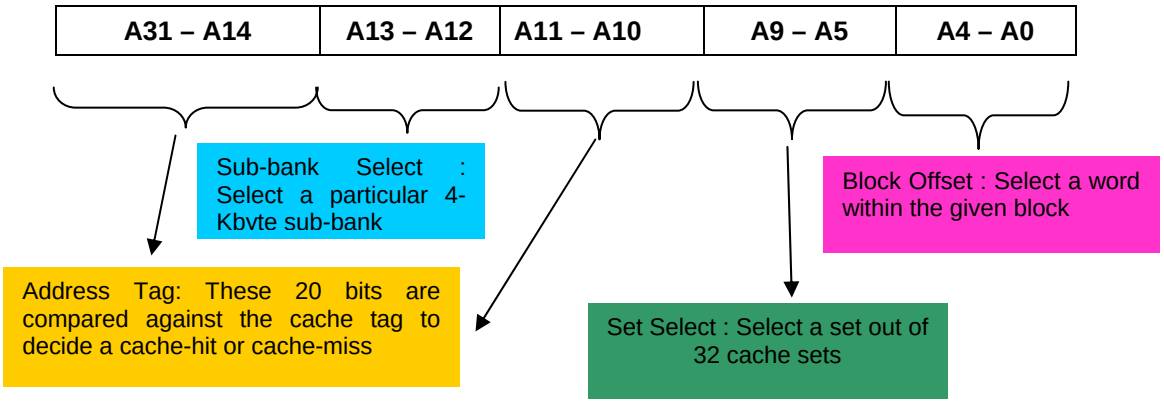


Figure 5. Mapping of the External Memory Address Space on the Instruction Cache Memory Space

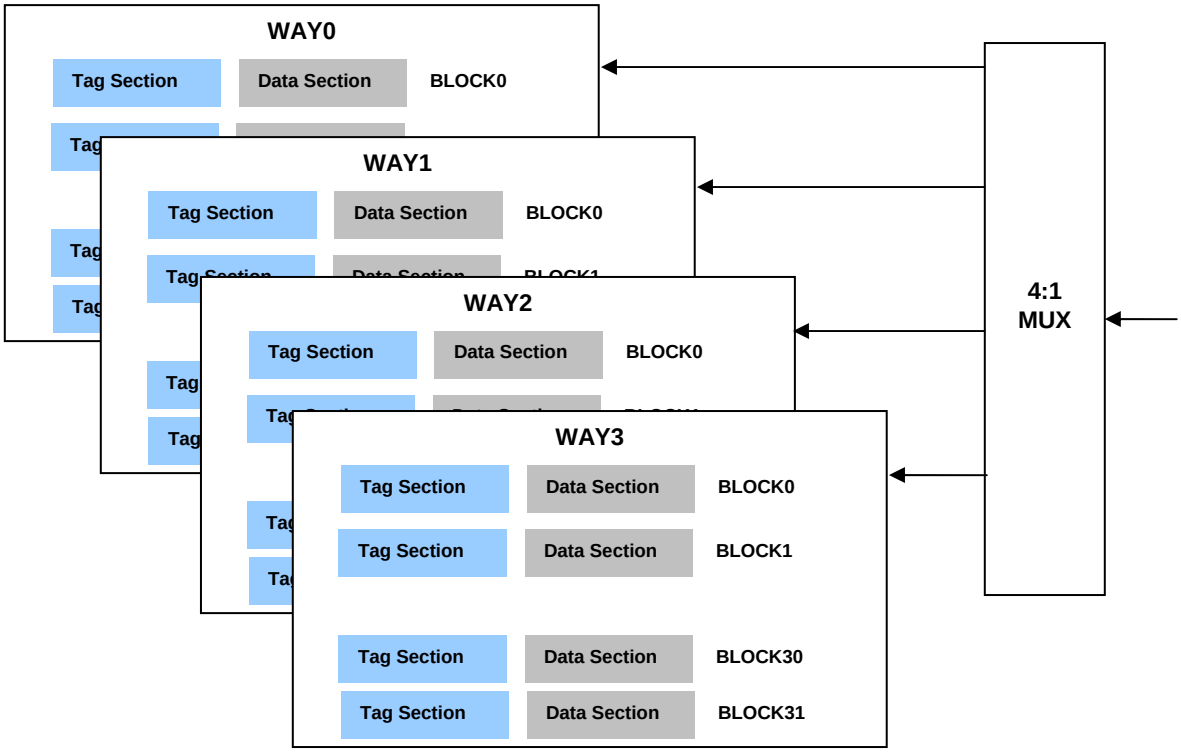


Figure 6. Blackfin Instruction Cache Configuration – 4-Way Set Associative

On a cache miss, the instruction memory unit generates a cache line fill access to retrieve the missing cache block from the external memory. The core halts until the target instruction word is returned from external memory.

The address for the external memory access is the address of the target instruction word. The cache line replacement unit uses the `Valid` and `LRU` bits (of unlocked ways) to determine which block (in the given set) is to be used for the new cache line. Figure 7 shows how the cache line replacement policy is selected.

Only One Invalid Way in the Set	Incoming block would replace this block
More than one Invalid Way in the Cache	The ways would be replaced in the following order: Way0 first, then Way1, then Way2, Way3
No Invalid Ways in the Cache	Least Recently Used (LRU) way would be replaced. **For Modified LRU policy** : Way with high priority would not be replaced if a low-priority way exists in the given set. A low-priority block cannot replace a high-priority block. If all ways are high priority, the low-priority way cannot be cached.

Figure 7. Cache Line Replacement Policy for the Blackfin Instruction Cache

The cache line fill access consists of fetching 32 bytes (one block) of data from memory. The address for the read transfer is the address of the target instruction word. When responding to a line-read request from the instruction memory unit, the external memory returns the target instruction word first. The next three words are fetched in sequential address order as depicted in Table 2.

Target Word	Fetching order for next three words
WD0	WD0, WD1, WD2, WD3
WD1	WD1, WD2, WD3, WD0
WD2	WD2, WD3, WD0, WD1
WD3	WD3, WD0, WD1, WD2

Table 2. Cache Line Word Fetching Order

When the cache block is retrieved from the external memory, each 64-bit word is buffered in a four-entry line fill buffer before it is written to a 4-Kbyte memory bank. The line fill buffer allows the core to access the data from the new cache line as the line is being retrieved from external memory, rather than having to wait until the line has been written in to the cache.

Instruction Cache Locking by Way

The instruction cache has four independent lock bits (these bits are available in the Instruction Memory Control register), which can be used to lock any of the four ways independently.

When a particular way is locked (the corresponding bit in Instruction Memory Control register is set), it does not participate in the block replacement. The cached instructions from a locked way can be removed only with an `IFLUSH` instruction.

The code-2 example (see associated ZIP file) demonstrates how the more frequently used functions (in the external memory) can be cached and locked such that they would not be replaced. The scheme consists of locking Way1, Way2, Way3 (Way0 unlocked), and making a dummy call to the functions of interest. The functions will be cached to Way0 (as all other ways are locked). Now, Way0 can be locked (and Way1, Way2, Way3 can be unlocked). Any subsequent cache miss can replace blocks in Ways 1-3 (Way0 is locked) only.

Blackfin Data Cache Configuration

Cache Organization

The ADSP-BF533 has 64 Kbytes of on-chip data memory. 32 Kbytes of data memory is available as data SRAM. The remaining 32 Kbytes of memory is available as two independent 16-Kbyte memory banks, which can be configured either as data cache or as data SRAM.

When enabled, the data cache works as 2-way set associative memory. The data cache-controller uses LRU policy for block replacement (it can not use modified LRU policy as is the case with instruction cache). Each 16-Kbyte cache bank is arranged as four 4-Kbyte sub-banks. A sub-bank consists of 64 sets (each set is four blocks). The block size is 32 bytes. Depending on the number of memory banks configured as cache, the data cache is either 16 Kbytes or 32 Kbytes.

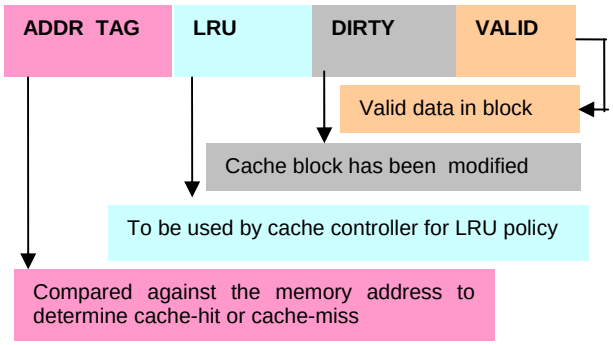


Figure 8. Tag Section of Data Cache Block

Similar to the instruction cache, each block has a tag portion associated with it. Figure 8 depicts the tag section in a data cache block. Figure 9 shows how a 4-Kbyte sub-bank in the instruction cache is arranged.

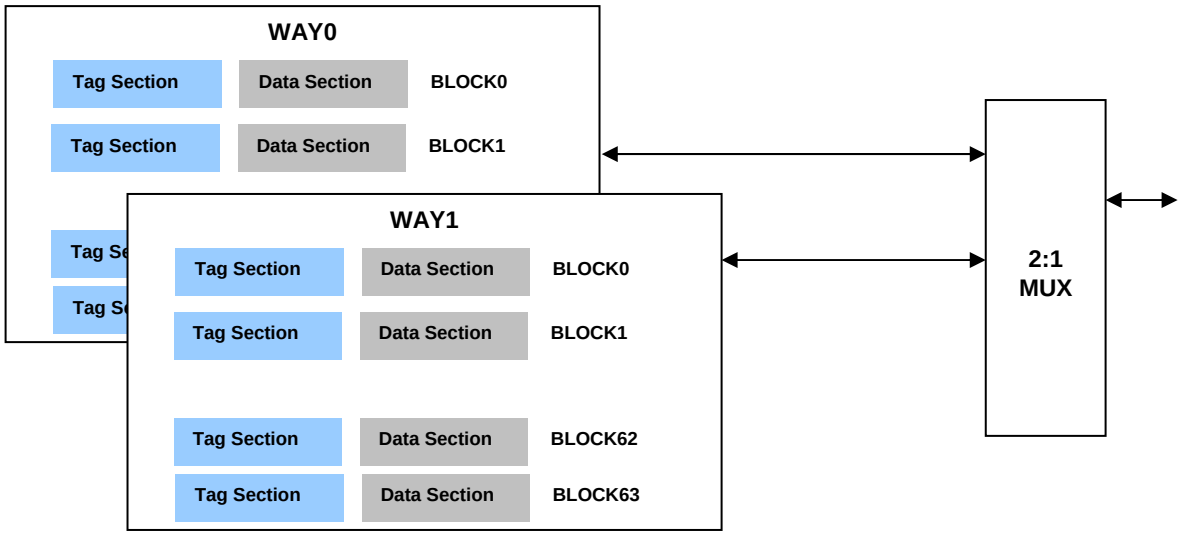


Figure 9. Blackfin Data Cache Configuration – 2-Way Set Associative

Figure 10 shows how the 32-bit address space is mapped to cache memory space. When both data banks are enabled as cache, depending on the state of DCBS bit, either bit 14 or bit 23 of the address space is used to select one of 16-Kbyte data banks.

Cache Write Methods

The external memory is divided into different pages (defined by Data Cache Protection Lookaside Buffers — DCPLB registers). The attributes for each page can be configured independently. As discussed in the next section, the memory pages can be:

- Configured either in the write-back mode or in the write-through mode
- Configured to allocate the cache lines either on reads only or on reads and writes

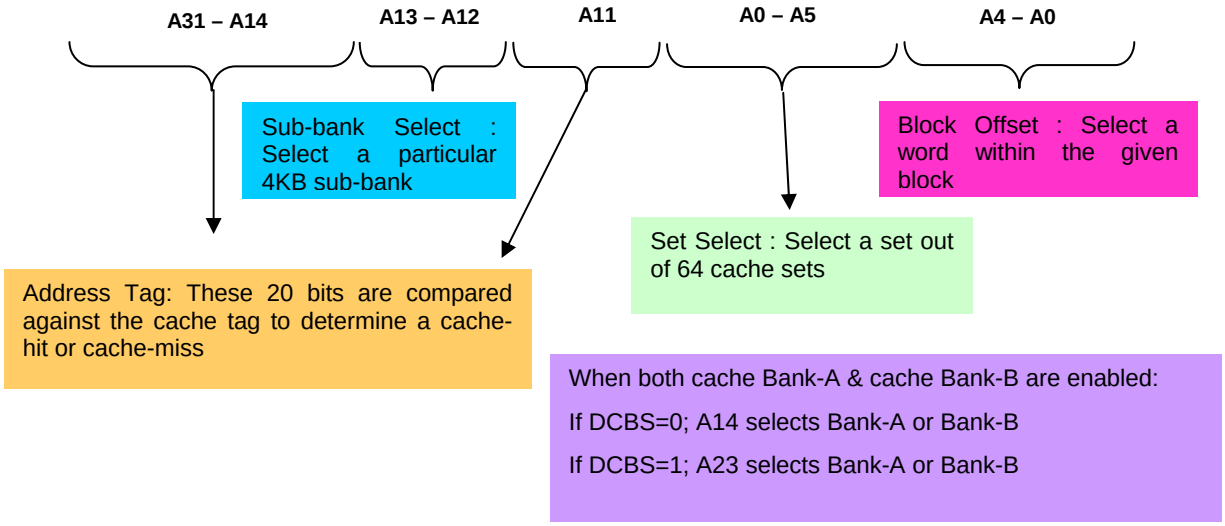


Figure 10. Mapping of the External Memory Address Space on the Data Cache Memory Space

Memory Management Unit (MMU)

The Blackfin processor contains a page-based MMU, which provides control over cacheablility of memory ranges and management of protection attributes at a page level. The MMU is implemented as two 16-entry Content Addressable Memory (CAM) blocks. Each entry is referred to as a cacheability protection lookaside buffer (CPLB) descriptor.

Cacheability Protection Lookaside Buffers (CPLBs)

Each entry to the CPLB descriptors defines cacheability and protection attributes for the given memory page. The CPLB entries are divided between instruction and data CPLBs. 16 CPLB entries (called ICPLBs) are used for instruction fetch requests. Another 16 CPLB entries (called DCPLBs) are used for data transactions. Setting the appropriate bits in the Instruction Memory Control (`IMEM_CONTROL`) and Data Memory Control (`DMEM_CONTROL`) registers enable the ICPLBs and DCPLBs. Each CPLB entry consists of a pair of 32-bit values.

For Instruction Fetches

`ICPLB_ADDR[n]` defines the start address of the page described by the CPLB descriptor.

`ICPLB_DATA[n]` defines the properties of the page described by the CPLB descriptor. Figure 11 depicts various bit-fields and their functionality in the `ICPLB_DATA` register.

For Data Operations

`DCPLB_ADDR[m]` defines the start address of the page described by the CPLB descriptor.

`DCPLB_DATA[m]` defines the properties of the page described by the CPLB descriptor. Figure 12 depicts various bit-fields and their functionality in the `DCPLB_DATA` register.

Memory Pages and Page Attributes

Each CPLB entry corresponds to a valid memory page. Every address within a page shares the attributes defined for that page. The address descriptor `xCPLB_ADDR[n]` provides the base address of the page in memory. The property descriptor word `xCPLB_DATA[n]` specifies size and attributes for the page.

Page Size

The Blackfin memory architecture supports four different page sizes – 1 Kbyte, 4 Kbyte, 1 Mbyte, or 4 Mbyte. Pages must be aligned on page boundaries that are an integer multiple of their size.

Cacheable/Non-cacheable Flag

If a page is defined as non-cacheable, access to this page bypasses the cache. The memory pages may need to be defined as cacheable when:

- An I/O device is mapped to the address
- The code residing in the page is in-frequently called (the user may not want it to be cached)

Write-Through/Write-Back Flag

Present on data memory only, this attribute defines the write-back mode for the data cache.

Dirty/Modified Flag

Present on data memory only, this attribute is valid only when the page is defined as cacheable in write-back mode. This bit should be set prior to store accesses to this page.

Write Access Permission Flags

Data memory CPLBs feature two flags that enable/disable write accesses to the corresponding page for supervisor mode and user mode, individually.

User Read Access Permission Flag

This attribute enables/disables reads from this page in user mode.

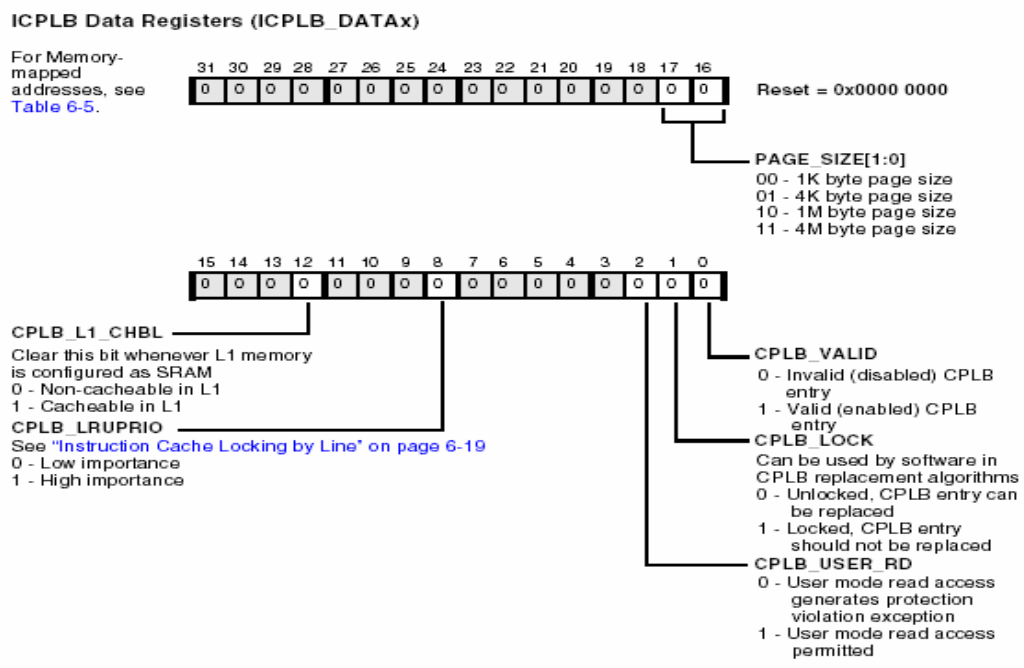


Figure 11. Bit Fields and Their Functionality for the ICPLB_DATA Register

Lock Flag

Locks a CPLB entry. This attribute is useful for dynamic memory management. When a CPLB entry is locked, the exception handler for CPLB miss will not consider it for replacement.

LRU Priority

This attribute is available on instruction memory CPLBs only. It defines LRU priority (low/high) for the given page. This is used for the modified LRU policy.

The page attributes related to "Read/Write Permission" deal with the memory protection. It may be required in a real-time application in which the entire application is partitioned between OS code and user code. The user code may have different threads, with each thread having its own memory resources, which are not accessible to the other threads. However, the OS kernel can access all the memory resources. This task can be achieved by having different CPLB configurations in different threads.

When the CPLBs are enabled, a valid CPLB entry should exist in the CPLB table for every address to which an access is made. If there is no valid CPLB entry for the referenced address, a CPLB exception is generated.

Page Descriptor Table

Generally, a memory-based data structure called a Page Descriptor table is used for CPLB management. All the potentially required CPLB entries can be stored in the Page Descriptor table (which is generally located in the internal SRAM). When the CPLBs need to be configured, the application code can pick up the CPLB entries from Page Descriptor table and fill them into the CPLB descriptors.

For small/simple memory models it may be possible to define a set of CPLB descriptors that fit into 32 CPLB entries (16 ICPLBs and 16 DCPLBs). This type of definition is referred to as a *static memory management model*. Example1 uses a static memory management model.

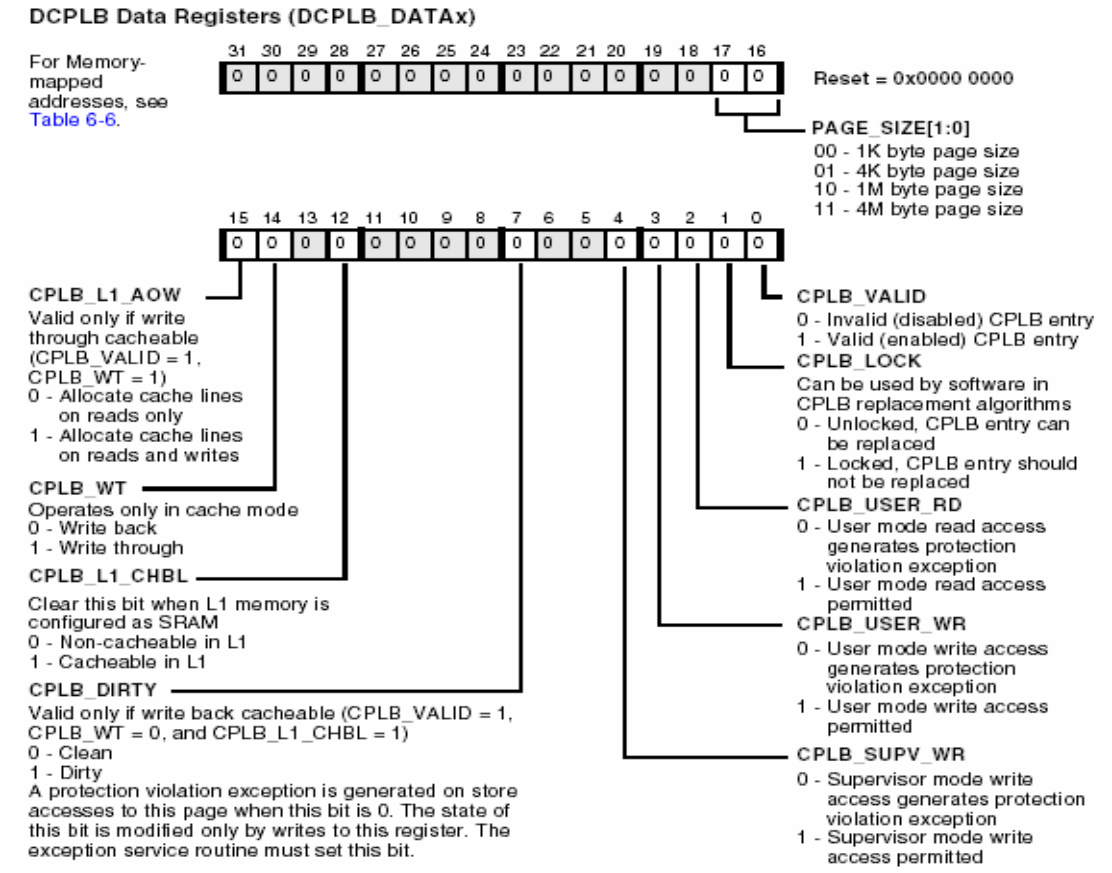


Figure 12. Bit Fields and Their Functionality for the ICPLB_DATA Register

For complex memory models, the Page Descriptor table may have CPLB entries, that do not can't fit into the available 16 CPLB registers. Under such conditions, the CPLBs can be configured first with any 16 entries. When the processor references a memory location, which is not defined by the CPLBs, an exception is generated and the address of faulting memory location is stored in the Fault Address register (`xCPLB_FAULT_ADDR`). The exception handler can use the faulting address to search for required entry in the CPLB table. One of the existing CPLB entries can be replaced by this new CPLB entry.

The CPLB replacement policy can be simple or complex, depending upon the system requirement. It is possible that more than one memory reference are made to the addresses for which there are no valid entry in the CPLB descriptors. Under such a condition, the exceptions are prioritized and serviced in this order:

- Instruction page misses
- Page misses on DAG0
- Page misses on DAG1

The code in example3 provides exception handler for DCPLB miss. It uses a round-robin scheduling method for the DCPLB replacement.

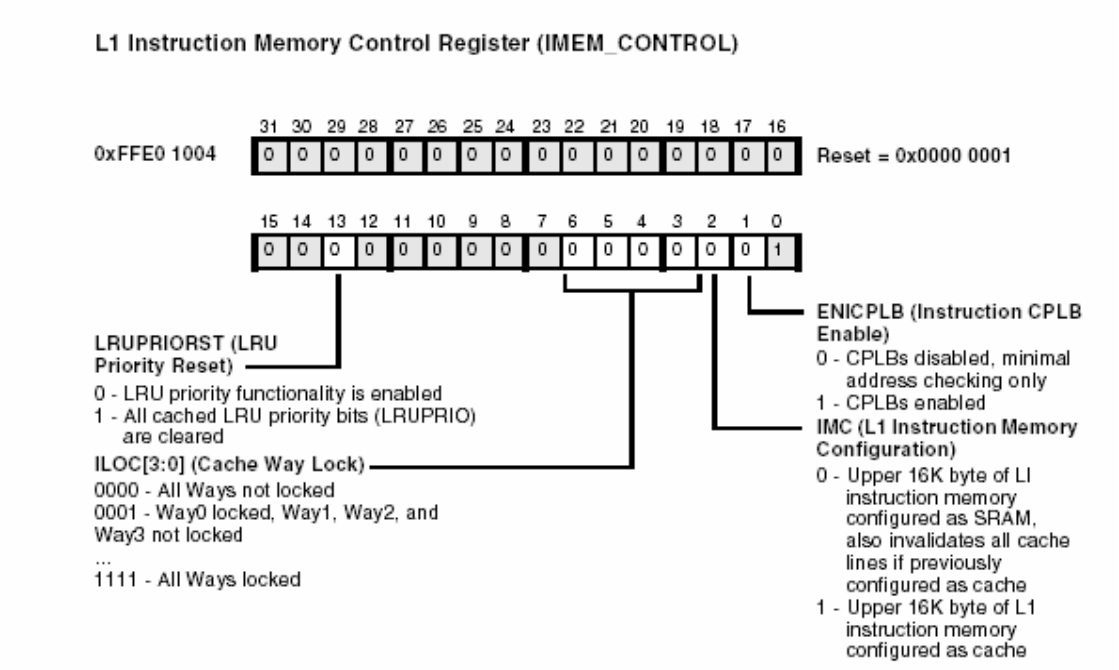


Figure 13. Bit Fields and Their Functionalities for the IMEM_CONTROL Register

Handling the Instruction and Data Cache

Enabling the Cache

The instruction and data caches can be enabled/disabled independently by configuring the `IMEM_CONTROL` and `DMEM_CONTROL` registers appropriately. The example demonstrates how the data/instruction caches can be enabled.

- Before enabling the cache, valid CPLB descriptors must be configured and enabled.
- When the memory is configured as cache, it cannot be accessed directly (neither through core, nor through DMA).

Instruction Memory Control Register (IMEM_CONTROL)

Figure 13 depicts various bit-fields and their functionality in the `IMEM_CONTROL` register.

Data Memory Control Register (DMEM_CONTROL)

Figure 14 depicts various bit-fields and their functionality in the `DMEM_CONTROL` register.

Instruction Cache Invalidation

There are three schemes for invalidating the instruction cache:

(a) The `IFLUSH` instruction can be used to invalidate a specific address in the memory map. When the instruction `IFLUSH [p2];` is executed, if the memory address pointed by `p2` has been brought in to the cache, the corresponding cache line will be invalidated after execution of the above instruction. When the instruction is of the form `IFLUSH [p2++];` the pointer increments by the size of a cache-line (e.g. 32 bytes).

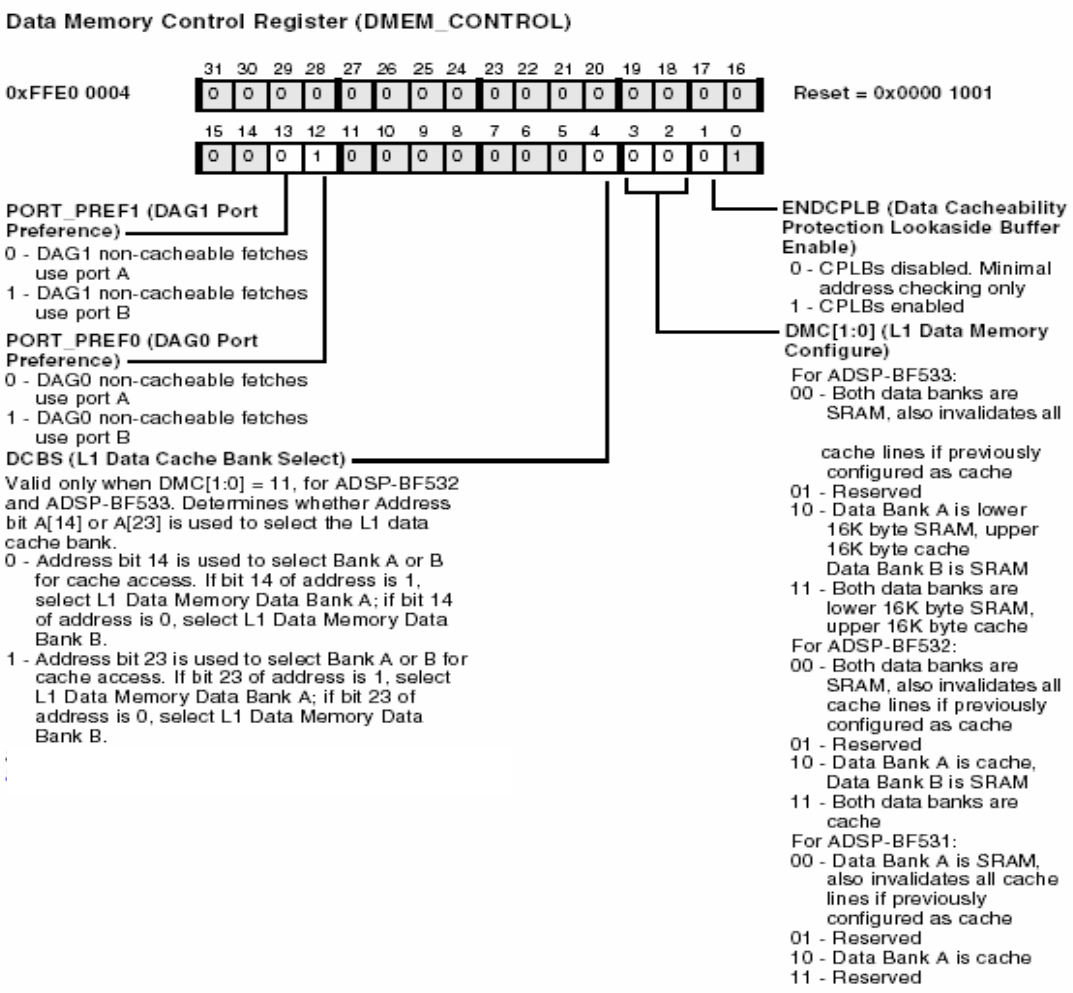


Figure 14. Bit Fields and Their Functionalities for the DMEM_CONTROL Register

(b) The `Valid` bit of the tag section for any block in the cache can be cleared explicitly by writing a one to the tag section. The value to the tag section can be written using the `ITEST_COMMAND` register. This is discussed in detail in the next section.

(c) In order to invalidate the entire cache, the `IMC` bit in the `IMEM_CONTROL` register can be cleared. This clears the valid bit for all tag sections in the cache. The `IMC` bit can be set to enable the cache again.

Data Cache Control Instructions

(a) The `PREFETCH` instruction can be used to allocate a line into the L1 cache.

(b) The `FLUSH` instruction causes the data cache to synchronize the specified cache line with the external memory. If the cached data line is dirty, the instruction writes the line out and marks the line clean in the data cache.

(c) The `FLUSHINV` instruction causes the data cache to perform the same function as the `FLUSH` instruction and then invalidates the specified line in the cache. If the location is not dirty, no flush will occur. In this case, only the invalidate step takes place.

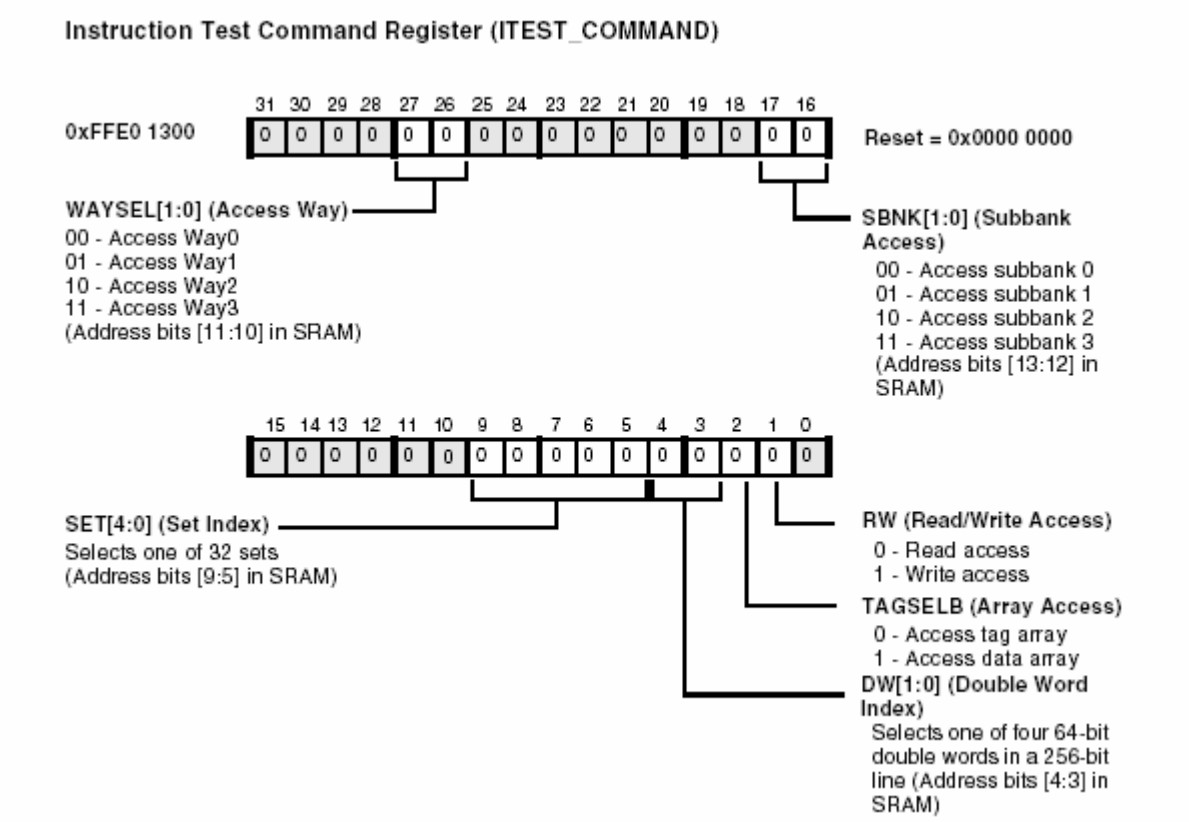


Figure 15. Bit Fields and Their Functionalities for the ITEST_COMMAND Register

Accessing the Cache Memory

When configured as cache, the L1 memory bank cannot be accessed directly by the core/DMA. Read/write operations can be performed onto the cache space using the `ITEST_COMMAND` and `DTEST_COMMAND` registers. The `DTESET_COMMAND` register can also be used to access the instruction SRAM banks. Figure 15 shows the bit fields for the `ITEST_COMMAND` register. Figure 16 shows bit fields for the `DTEST_COMMAND` register.

Accessing the Instruction Cache

The `ITEST_COMMAND` register can be used to access the data or tag sections of the instruction cache blocks.

A cache block is divided in to four 64-bit words. Any of the four words can be selected for access. While reading the cache, the data value is read into the `ITEST_DATA[1:0]` register set. While writing to the cache the value from the `ITEST_DATA[1:0]` register set are written to the cache.

When a tag section is being accessed, the 32-bit tag value is transferred to/from the `ITEST_DATA0` register.

Consider an example where value `0x0C010360` is written in to the `ITEST_COMMAND` register. This instruction will read the tag section from the way-3, set-27, subbank-1 and transfer it to the `ITEST_DATA0` register. Similarly, writing `0x0C010362` transfers the contents of `ITEST_DATA0` register to the tag section of the block located in way-3, set-27 and subbank-1. While accessing the tag section (read or write) bit 3 and bit 4 of the `ITEST_COMMAND` register are reserved.

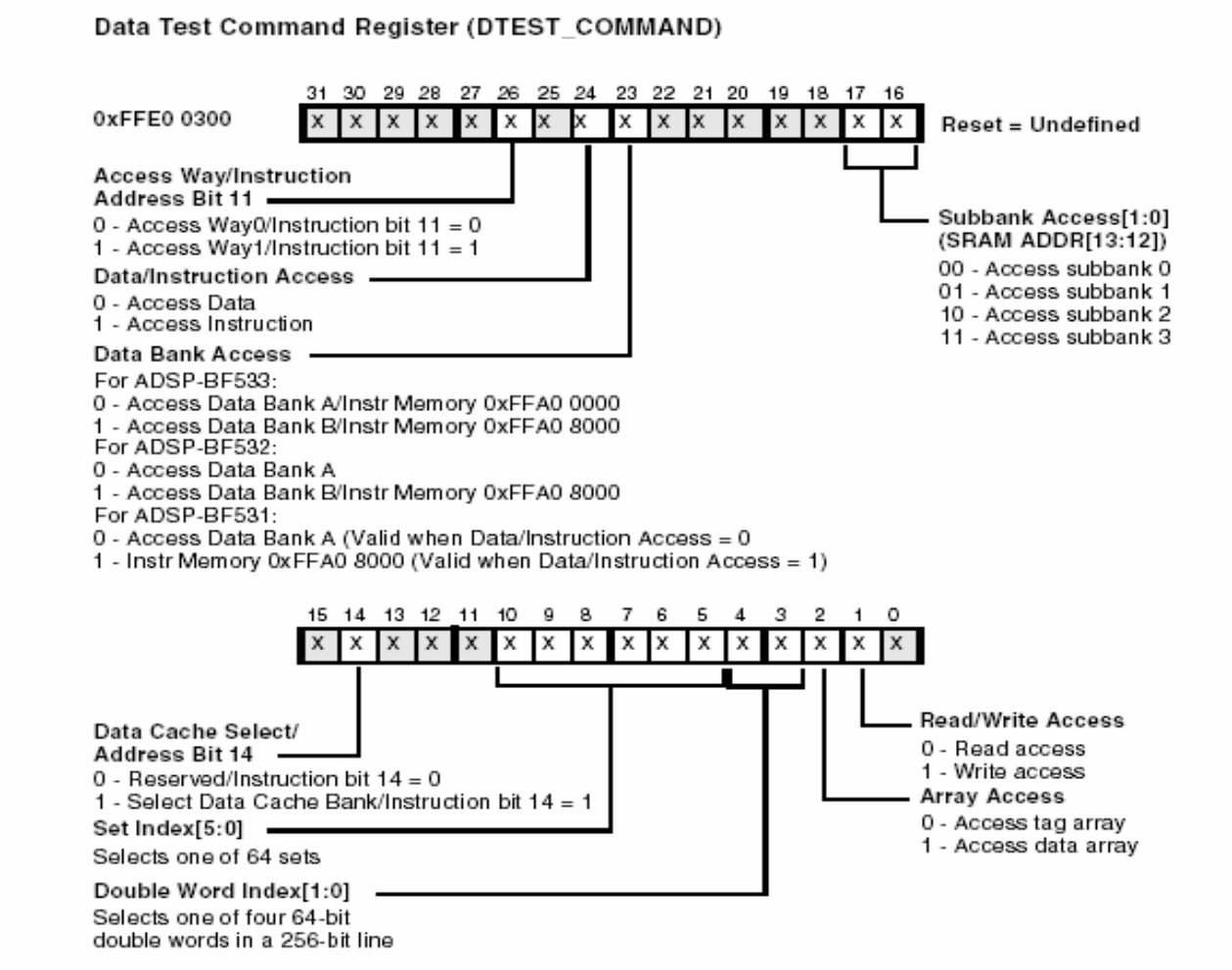


Figure 16. Bit Fields and Their Functionalities for the DTEST_COMMAND Register

Writing a value of `0x0C010374` will read the second word of the cache block located at way-3, set-27, subbank-1 and transfer it to the `ITEST_DATA[1:0]` register set. Writing a value of `0x0C010366` to the `ITEST_COMMAND` register transfers the value of the `ITEST_DATA[1:0]` register set to word-0 of the cache block located at way-3, set-27, subbank-1 of the instruction cache. While writing to the cache, the `ITEST_DATA[1:0]` register test must be loaded before the `ITEST_COMMAND` register is written.

Accessing the Data Cache

When bit 24 of the `DTEST_COMMAND` register is cleared, the `DTEST_COMMAND` register can be used to access the data or tag sections of data cache blocks. A word from the data section of cache block can be transferred to/from the `DTEST_DATA[1:0]` register set.

While accessing the tag section, the 32-bit tag value is transferred to/from the `DTEST_DATA0` register.

Consider a case where the values of the `DTEST_DATA[1:0]` register set needs to be written to word-0 of the data block in way-1, set-39, subbank-0, data cache bank-A. Then the `DTEST_COMMAND` register can be written with a value of `0x040004E5`. The `DTEST_DATA[1:0]` register set must be loaded before writing to the

DTEST_COMMAND register. Bit 14 of the DTEST_COMMAND register is reserved while accessing the data cache space (bit 24 =0).

Accessing the Instruction SRAM

When bit 24 of the DTEST_COMMAND register is set, the DTEST_COMMAND register can be used to access the instruction SRAM. A 64-bit word can be transferred to/from the DTEST_DATA[1:0] register set to/from the instruction SRAM. Thus, memory can be accessed eight bytes at a time.

Bit 2 of the DTEST_COMMAND register must be set while working in this mode. Bits 3-10 must be assigned with bits 3-10 of the address being accessed. Consider a case where the byte from address 0xFFA07935 has to be read from the instruction memory. This address lies in the bank-1. While accessing the above byte an entire block addressed by (0xFFA07930 – 0xFFA07937) will be accessed. The control value that must be loaded to the DTEST_COMMAND register will be 0x05034134.

VisualDSP++ Compiler Support

The VisualDSP++ tools support cache memory management. Some features are discussed below. Detailed information can be found in the *VisualDSP++ Compiler Manual for Blackfin Processors.* Instruction and data caches can be enabled together or separately, and the memory spaces they cache are configured separately.

CPLB Control

CPLB support is controlled through a global integer variable, ___cplb_ctrl. Its C name has two leading underscore characters, and its assembler name has three. The value of this variable determines whether the startup code enables the CPLB system. By default, the variable has the value zero, indicating that CPLBs should not be enabled. The pragma retain_name should be used with __cplb_ctrl, such that this variable is not eliminated by the compiler when optimization is enabled.

The value of ___cplb_ctrl may be changed in several ways:

- The variable may be defined as a new global with an initialization value. This definition supersedes the definition in the library.
- The linked-in version of the variable may be altered in a debugger, after loading the application but before running it, so that the startup code sees a different value.

When enabling caches using ___cplb_ctrl, it is imperative that USE_CACHE also be specified.

CPLB Installation

When ___cplb_ctrl indicates that CPLBs are to be enabled, the startup code calls the routine _cplb_init. This routine sets up instruction and data CPLBs from a table, and enables the memory protection hardware. The default configuration tables are defined in files called cplbtab*n*.s in VisualDSP\Blackfin\lib\src\libc\crt, where *n* is the part number of the Blackfin processor.

When the cache is enabled, default CPLB configuration defined in above file is installed. However, you can modify the given files to define your own CPLB configuration. The given file must be included in the project files in order for the changes be effective. The project “example5” demonstrates how the CPBL configuration table can be modified.

Exception Handling

As discussed earlier, in a complex memory model there may need to be more CPLBs than can be active at once. In such systems, there will eventually come a time when the application attempts to access memory that is not covered by one of the active CPLBs. This will raise a CPLB miss exception.

The VisualDSP++ library includes a CPLB management routine for these occasions, called `_cplb_mgr`. This routine should be called from an exception handler that has determined that a CPLB miss has occurred, regardless whether it is a data miss or an instruction miss. `_cplb_mgr` identifies the inactive CPLB that needs to be installed to resolve the access, and replaces one of the active CPLBs with this one. If CPLBs are to be enabled, the default startup code installs a default exception handler, called `_cplb_hdr`; this does nothing except test for CPLB miss exceptions, which it delegates to `_cplb_mgr`. It is expected that users will have their own exception handlers to deal with additional events.

Summary of MMRs

The following memory-map registers are used for the memory management on ADSP-BF533:

IMEM_CONTROL	DMEM_CONTROL
ITEST_COMMAND	DTEST_COMMAND
ITEST_DATA [1:0]	DTEST_DATA [1:0]
ICPLB_DATA [15:0]	DCPLB_DATA [15:0]
ICPLB_ADDR [15:0]	DCPLB_ADDR [15:0]
ICPLB_STATUS	DCPLB_STATUS
ICPLB_FAULT_ADDR	DCPLB_FAULT_ADDR

Table 3. CPLB Memory-Mapped Registers

Cache Configuration for Blackfin Derivatives

The preceding section discussed cache configuration and cache control on ADSP-BF533 processors. The same discussion also applies to ADSP-BF531 and ADSP-BF532 processors. The example code supplied with this application note can be used with ADSP-BF531 and ADSP-BF532 processors.

The amount of data and instruction memory configurable as cache on ADSP-BF535 processors is same as on ADSP-BF533 processors. However, the amount of memory available on ADSP-BF561 processors as cache is double of that of ADSP-BF533 processors. The instruction/data cache configuration and control on ADSP-BF535 and ADSP-BF561 is similar to that on ADSP-BF533. However, some MMR addresses for MMRs and bit fields definition differ. This must be considered while configuring the cache.

Conclusion

This document discussed the instruction and data cache configuration on Blackfin processors. The address mappings to the cache blocks has also been discussed. The example code provided with this application note demonstrate how to set up CPLB descriptors for instruction memory and data memory, how to enable/disable the instruction/data cache, and how to handle the CPLB exceptions and locking the instruction cache by way. Discussion on accessing the instruction/data cache by core through `xTEST_COMMAND` has also been included.

Appendix

A ZIP file is associated with this document. It contains the following code examples:

(a) Example code for configuring the CPLB descriptors and instruction/data cache

(b) Example code for CPLB exception handling

(c) Example code for locking the instruction cache by way

(d) Example codes demonstrating the Data cache control instructions

(e) Example code demonstrating VisualDSP++ compiler support for Blackfin Cache.

References

[1] *ADSP-BF533 Blackfin Hardware Reference.* Rev 3.1, May 2005. Analog Devices, Inc.

[2] *Computes Architecture A Quantitative Approach.* Second Edition, 2000 David A. Patterson and John L. Hennessy

Document History

Revision	Description
Rev 1 – June 13, 2005 by Kunal Singh	Initial Release