JOURNAL OF SPACECRAFT AND ROCKETS
Vol. 48, No. 4, July–August 2011

Improving the LEON Spacecraft Computer Processor for Real-Time Performance Analysis

David Guzmán,[*] Manuel Prieto,[†] Sebastián Sánchez,[‡] Javier Almena,[§]
Oscar Rodríguez,[¶] and Daniel Meziat[**]
Universidad de Alcalá, 28805 Alcalá de Henares, Spain

DOI: 10.2514/1.50209

This paper presents an enhanced version of the LEON architecture with a performance-monitoring unit, including performance counters for different events and a trace buffer. Through this enhancement the user or system engineer is able to obtain information about the cache behavior and the execution path of a whole program or a fragment of code. This development introduces several advantages over simulators, such as allowing tests to be performed directly on the real target in a nonintrusive mode. Taking advantage of this enhancement, a suite of tests for cache characterization and evaluation has been developed. The results of this study can contribute to modeling the behavior of the LEON architecture in areas such as worst-case execution-time estimation and probabilistic analysis. The new features can be very useful during the validation and verification phase of the real-time software, supporting code coverage techniques and reducing the execution-time variability due to hardware effects such as cache and pipelines. The field-programmable gate-array resources employed by the performance-monitoring unit developed in this paper are low, on the order of 3–5%, with the exception of an increase of 37% in the use of embedded memory blocks.

Nomenclature

N_{int}	=	number of interruptions that has taken place
T_{flush}	=	overall execution time with flush operations, s
$T_{\mathrm{flushpenalty}}$	=	difference of time caused by flush operations, s
$T_{\mathrm{int}}(\mathrm{flush})$	=	time taken by the interrupt service routine with flush, s
T_{original}	=	native bench execution time, s

I. Introduction

MODERN processors include architectural features such as pipelines, caches, and branch-prediction units that can result in significant performance improvements. However, these enhancing features introduce a high level of unpredictability and uncertainty into the workload behavior. To obtain a better understanding of these techniques and to allow the development of architectural improvements, most of these processors include hardware support for nonintrusive monitoring of a variety of system events. Commonly referred to as hardware performance counters, this capability is very useful to both applications developers and computer architects. Through these counters we can collect information about program behavior, analyze performance, and identify the causes of possible bottlenecks. These counters are an integral part of the performance-monitoring unit (PMU). Commonly, the performance counters provide information about the number of instructions executed, data- and instruction-cache hit rates, the number of branch mispredictions, and the number of interrupts handled. Usually, methods for implementing code coverage analysis tools are based on program instrumentation. This approach presents the disadvantage of introducing an elevated overhead, due to the insertion and execution of instrumentation code, and these schemes are typically not portable to many software environments. PMUs attempt to lower this overhead by leveraging the number of performance counters and events that can be managed [1–4].

The availability of PMUs in processors for space applications is usually related to their origin. On one hand, these processors can be adapted versions of commercial general-purpose microprocessors. In this case, their technologies are modified to be radiation-tolerant. However, their architectures are not modified, because it can produce a great increment of the price or reliability loss. Therefore, these processors will have a PMU if it is present in their commercial version, such as the processors based on PowerPC architecture (RAD6000 and RAD750 from BAE systems or hardware-embedded processors in Xilinx devices). On the other hand, PMUs are not commonly available in space-borne microprocessors such as HX1750, MAS281, and the LEON architecture; this can be mainly for two reasons. First is the fact that it involves validation cost and device resource consumption. Second, only specific real-time applications need a PMU. LEON architecture is widely employed for spacecraft applications with tight real-time execution-time guarantees and is being used on missions such as BepiColombo. This limitation can be solved, due to the availability of its Very High Speed Integrated Circuit Hardware Description Language (VHDL) source code. New performance counters can be included at a very small hardware cost, due to the low number of required resources. Moreover, the processor performance is not degraded, and no critical timing paths are included in the system, due to the nonintrusive behavior of the new feature.

In real-time systems, due to the criticality of the tasks that are executed and the need to ensure that the system will always perform its required functionality within the specified deadlines, it is mandatory to have a precise knowledge of the system behavior. For instance, obtaining accurate information about the longest time that a portion of a program or software can take to run, termed the worst-case execution time (WCET) [5,6], is a key issue to ensure that time constraints are met and that real-time systems operate correctly. WCET determination becomes crucial in critical applications in

Received 6 April 2010; revision received 23 March 2011; accepted for publication 27 March 2011. Copyright © 2011 by the American Institute of Aeronautics and Astronautics, Inc. All rights reserved. Copies of this paper may be made for personal or internal use, on condition that the copier pay the $10.00 per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923; include the code 0022-4650/11 and $10.00 in correspondence with the CCC.

[*]Ph.D. Candidate, Department of Computing Engineering, Space Research Group; currently Electrical Engineer, NASA Goddard Space Flight Center, 8800 Greenbelt Road, Greenbelt, MD 20771; dguzman@srg.aut.uah.es.

[†]Associate Professor, Department of Computing Engineering, Space Research Group, Campus Universitario; mpm@srg.aut.uah.es.

[‡]Associate Professor, Department of Computing Engineering, Space Research Group, Campus Universitario; chan@srg.aut.uah.es.

[§]Research Assistant, Department of Computing Engineering, Space Research Group, Campus Universitario; jalmena@srg.aut.uah.es.

[¶]Assistant Professor, Department of Computing Engineering, Space Research Group, Campus Universitario; opolo@srg.aut.uah.es.

[**]Professor, Department of Computing Engineering, Space Research Group, Campus Universitario; meziat@srg.aut.uah.es.

which there is a risk when a deadline is missed, such as aircraft and space systems [7]. Some of the tasks performed in these systems must be done in real time in order to avoid system failures or even mission loss [8]. Cache memories have not been extensively used in space and avionic applications due to the problems of calculating the WCET. The state of the cache, if it is full or not (need of a flush operation) and the effect of asynchronous events (interruptions) have a great impact over the system, producing a less deterministic system. If the system is less deterministic, the software is more difficult to validate and there are more combinations of possible cases and configurations to be tested [9]. To guarantee the execution times in these systems, a common solution has been to disable the cache. This solution implies the underuse of its capabilities. For instance, on the Airbus A380 fly-by-wire computer, it was necessary to limit the use of the cache memory of the PowerPC MPC755 microprocessor from Motorola and to manage part of it statically, so that only part remains usable dynamically [10].

For now, the only way to obtain this performance information in systems based on the LEON architecture is through the LEON simulator named TSIM. In this paper a performance-monitoring unit for the LEON architecture is presented. The main goal of this research work is improving the LEON architecture in order to provide computer architects and application developers with a better knowledge of the workload behavior. By means of our development, statistics of part, or all, of the execution path of a program can be collected. Functions for starting and stopping the tracing (`start_tracing()` and `stop_tracing()` are provided to the user, aimed at analyzing fragments of code. The results provided by our solution offer more comprehensive information than that obtained solely by simulation. This is due to the fact that the system is stimulated by real input signals, whose behaviors are not always easy to simulate, as well as the fact that it is not easy to simulate the whole test environment. Another advantage of performance counters in comparison with simulators is the fact that simulator requires computationally intensive processing. Thus, the results of the simulation may not be readily available after the simulation has started and, as a consequence, an event that may occur instantaneously in the real world can actually take hours to mimic in a simulated environment. Although there are several tools available for the evaluation and study of the WCET (such as the Bound-T tool of Tidorum, the Heptane tool of IRISA, or SWEET from Malärdalen University), at this moment, to the best of our knowledge, only two tools have support for this architecture, aiT from AbsInt and RapiTime from Rapita Systems [11,12]. Regarding the WCET calculation in the LEON architecture, although deep studies about cache performance have been carried out in other microprocessors, few works can be found about the LEON processor [13], perhaps due to its relatively limited field of application. As part of this work, data- and instruction-cache characterizations are also provided in order to determine the WCET in a suite of test programs. To validate our work, we have characterized the LEON cache behavior, comparing the results with previous tests [14] and determining if cache miss penalties have a significant influence on the execution time for typical space applications. The results of this study can contribute to modeling the behavior of the LEON microprocessor for the purposes of WCET static and probabilistic analysis. Therefore, our development is a low-cost solution to approach these issues and could also be a complement for these tools, offering the possibility to perform the testing on the real target.

The remainder of the paper is organized as follows: In Sec. II the LEON architecture is summarized, focusing on the parts of relevance for the performance-monitoring unit. In Sec. III the modifications included in the LEON architecture and the implementation details are described, in this case, focusing on the LEON3. Section IV covers the test environment. Section V describes the test carried out and its results. Conclusions are summarized in Sec. VI.

II. LEON3 Architecture

Although our development is available for both LEON2 and LEON3, in this paper we will focus on the LEON3 microprocessor as an example of the LEON architecture. The LEON3 is a 32-bit reduced-instruction-set-computer high-performance processor conforming to the SPARC version 8 standard. It has been specifically designed for embedded space applications and has been developed by Aeroflex Gaisler Research under ESA contract. Most of the source code is available under the GNU General Public License (GPL), and it can be easily implemented in several field-programmable gate-array (FPGA) prototype boards. The exception is the floating-point unit and the SpaceWire core, which are only available under commercial license. The system is organized around the advanced microcontroller bus architecture (AMBA). The AMBA bus is composed of two buses, the advanced high-performance bus (AHB) and the advanced peripheral bus (APB). The system uses the AHB bus to connect the LEON3 processor to memory the controller and other high-bandwidth devices such as SpaceWire Links, Ethernet 10/100 Mbit media access control, and controller area network 2.0 interface. Serial and JTAG debug interfaces are also connected to this bus. By default, the processor is the only master on the bus, while at least two slaves are provided: memory controller and APB bridge. The memory controller provides access to three types of memories: PROM, SRAM, and SDRAM. The APB bridge is connected to the AHB bus as a slave and acts as the master on the APB bus. It is used to access on-chip registers in the peripheral functions, such as universal asynchronous receiver/transmitter interfaces, interrupt controller, timers, and a general-purpose I/O port. New modules can be easily added to design using the on-chip AMBA AHB/APB buses. A block diagram of LEON3 architecture is shown in Fig. 1.

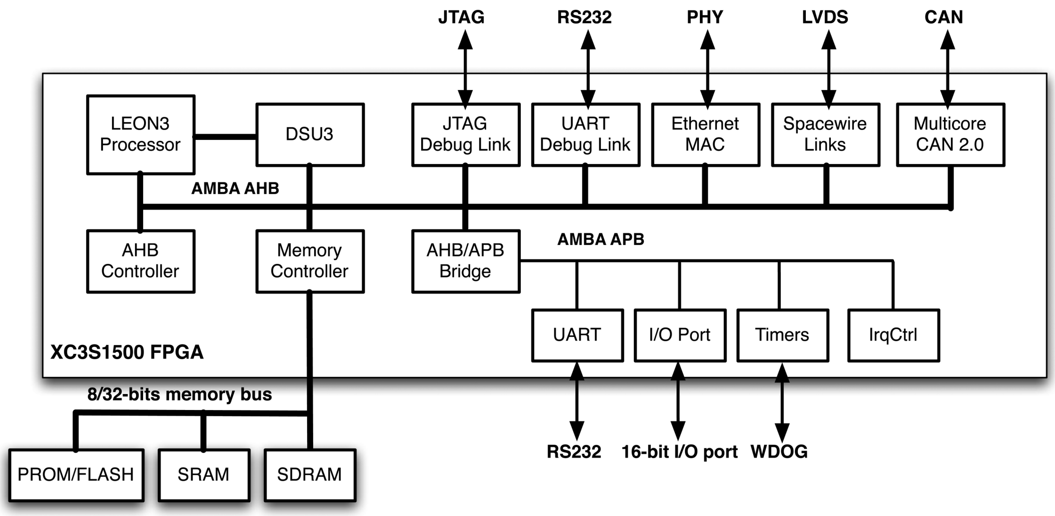


Fig. 1 Block diagram of the LEON3 architecture.

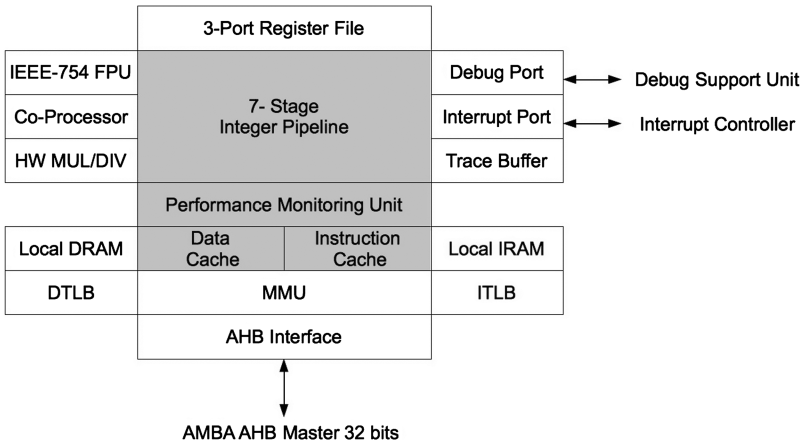


Fig. 2 LEON3 processor block diagram.

The LEON3 processor has the following main characteristics: seven-stage pipeline with Harvard architecture, separate instruction and data caches, hardware multiplier and divider, on-chip debug support, and multiprocessor extensions.

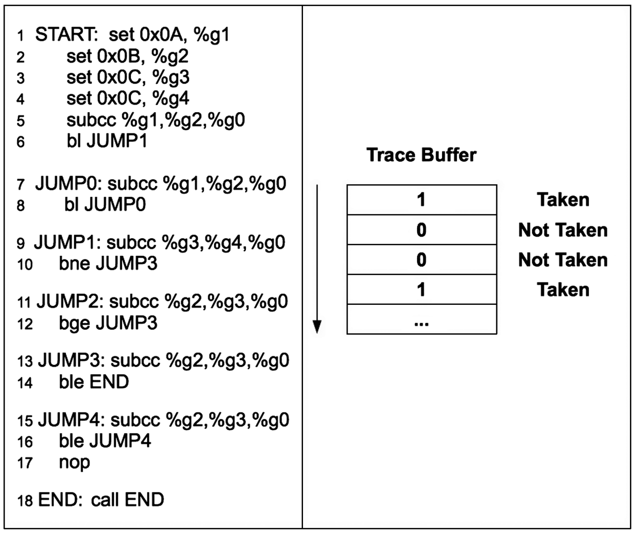
The design is highly testable through the debug support unit (DSU). The DSU can be used to aid software debugging on target hardware. Through the DSU, the user can set instruction breakpoints and perform single stepping. The LEON3 integer unit (IU) implements the full SPARC version 8 standard, including hardware multiply-and-divide instructions. The number of register windows is configurable within the limit of the SPARC standard (2–32), with a default setting of 8. The LEON3 processor has separate buses for instruction and data. Instruction and data buses are connected to two independent cache controllers. Both, instruction- and data-cache controllers can be separately configured to implement a direct-mapped cache or a multiset cache with 2–4 set associativity and maximum size of 64 kbytes. Cache lines can be either 16 or 32 bytes. The instruction cache uses streaming during line refill to minimize refill latency. The data cache uses write-through policy and implements a double-word write buffer. A local scratch-pad RAM can be added to both the instruction- and data-cache controllers to allow zero-wait-state access memory without data writeback.

III. Architecture Modifications

The LEON3 processor modules to be monitored are the IU and the cache subsystem. To include mechanisms to analyze the behavior of these units, the corresponding VHDL modules have been modified and 16 new 32-bit registers have been added. No critical timing paths into the processor core have been introduced, so that performance is not degraded by their inclusion. Furthermore, they work in a nonintrusive mode. The PMU also includes a trace buffer, which stores information about the branches. All these elements are mapped into the memory space of the APB bus through an APB slave called the performance-monitoring unit.[††] The new block diagram of the LEON3 processor, including the performance-monitoring unit and the modified modules, is shown in Fig. 2.

A. Integer Unit

This module has been modified, and eight new 32-bit registers and a trace buffer have been added. These eight new registers are divided into sets of two 32-bit registers. Every set of two registers is combined in a single 64-bit register used for accounting for different types of events. Therefore, four new registers have been added to the IU: the executed-instruction register (EIR), the clock-cycle register (CCR), the branch-instruction register (BIR), and the taken-branches register (TBR). EIR counts the number of instructions executed. CCR contains the number of clock cycles spent in the program execution. Application developers and system architects can use the new EIR and CCR registers to calculate the cycles per instruction (CPI) of the whole program execution or only a fragment of code of the program. BIR contains the number of branch instructions and TBR counts the number of taken branches. These new capabilities are very interesting due to the fact that they allow the user to perform the code coverage for the whole program execution or for only a predetermined fragment of code in a nonintrusive way. Previously, the only way to perform the code coverage was through the GCOV utility [15], with the disadvantage that it is an intrusive tool. The trace buffer complements the functionality of BIR and TBR storing information related to the branches. It stores a logic 1 when a branch is taken and 0 when a branch is not taken. In this manner the application developer can follow the program execution path and obtain more information about the workload behavior. The trace buffer is included in the FPGA embedded memory. Its size is configurable and is only limited by the number of FPGA memory blocks employed by the LEON3 processor. The trace buffer can be enabled to store the execution path of a program during a time interval or during the whole program execution. Sometimes, the size of the trace buffer will be too small to store the information related to the execution path of a whole program or a fragment of it. To solve this limitation, the PMU includes a mechanism to send the trace-buffer information through a serial line once it reaches a predetermined threshold. In Fig. 3 we show an example of how the PMU stores the information in the trace buffer. This example shows a


```
1  START:  set 0x0A, %g1
2          set 0x0B, %g2
3          set 0x0C, %g3
4          set 0x0C, %g4
5          subcc %g1,%g2,%g0
6          bl JUMP1

7  JUMP0: subcc %g1,%g2,%g0
8         bl JUMP0

9  JUMP1: subcc %g3,%g4,%g0
10        bne JUMP3

11 JUMP2: subcc %g2,%g3,%g0
12        bge JUMP3

13 JUMP3: subcc %g2,%g3,%g0
14        ble END

15 JUMP4: subcc %g2,%g3,%g0
16        ble JUMP4
17        nop

18 END: call END
```

Fig. 3 Instance of trace buffer working.

[††]All the modifications and new modules presented in this paper are available in our Web site (https://srg.aut.uah.es/joomla/index.php?option=com_content&task=view&id=86&Itemid=80&lang=en) under GNU GPL license.

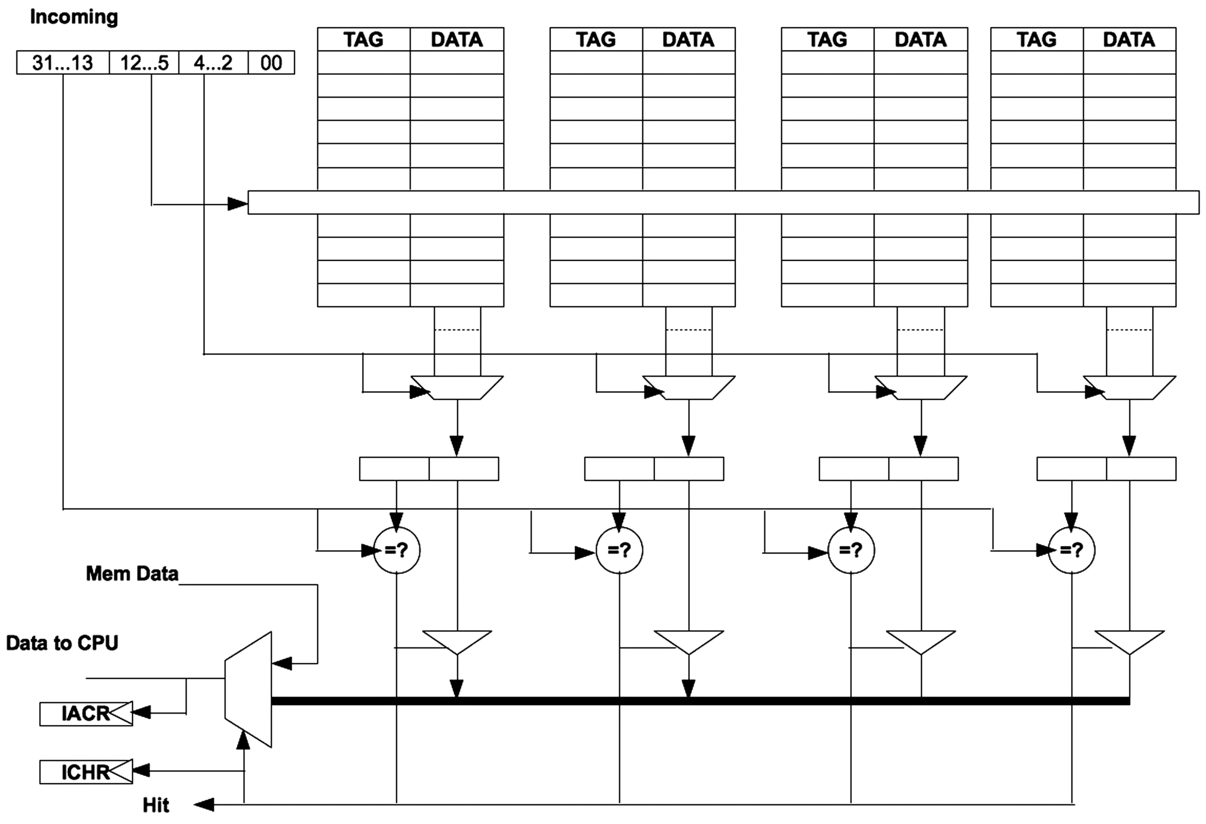


Fig. 4 Instruction-cache performance registers.

sample of code, where several values are stored into registers in order to do a set of comparisons with them. Through the trace buffer the user is able to easily follow and understand the program execution path. This can be done by examining the trace generated by the code execution while observing the result of the branch operation that is stored in the trace buffer. In this example, through the results presented by the PMU, the user knows that the first branch occurred on line 6. The two following branches that occurred on lines 12 and 14 are not taken, while the last one located on line 16 is taken. This enhancement can be very useful in order to follow more complex algorithms or fragments of code.

B. Cache Subsystem

Data- and instruction-cache VHDL modules have been modified and four new 64-bit registers, following the same approach described for the IU, have been added. The instruction access counter register counts the total number of accesses to instruction cache, both misses and hits, while the instruction-cache hit-rate register counts only the hits. Two similar registers can be found in the data-cache module. By means of both registers, the hit rates of instruction and data caches can be calculated, and a better understanding of the LEON3 cache can also be inferred. This approach contributes to the modeling of the cache and, moreover, statistical behavior of the cache can be obtained. In Fig. 4, a scheme of the approach under study for the instruction cache is shown.

C. Performance-Monitoring Unit

This module is in charge of mapping in memory the performance-counter registers that are implemented in the IU and cache subsystem. The performance-monitoring unit implements a 32-bit register named performance control register (PCR) which controls the performance registers. Through the PCR, the performance registers can be enabled, disabled and restarted, and the user can choose the method to manage the trace-buffer information. Within this module all the mechanisms needed to provide the user with the capability to perform the analysis in the whole program, or only in a portion of the code, are implemented. This is done by means of a write operation over the PCR. This fact takes special relevance when it is necessary to perform a deep study of the code coverage of a fragment of code.

D. Synthesis Results

Because of the simple and modular design, the resource consumption of the PMU is very low. As previously described, the size of the trace buffer is configurable. This value depends on the LEON3 design resource consumption, and for testing purposes, data and instruction caches used a configuration with one set of 4 kbytes and 32 bytes per line, with a random replacement algorithm, while the PMU trace buffer employed 16 kbytes. Synthesis and power estimation were performed using Xilinx ISE Design Suite 12. The

Table 1 FPGA resource and power consumption

Type of resource	LEON	LEON + PMU	Resource increments
Number of slices	6609 out of 13,312 (49%)	7141 out of 13,312 (52%)	532 out of 13,312 (3%)
Number of flip-flops	8044 out of 26,624 (30%)	8961 out of 26,624 (33%)	917 out of 26,624 (3%)
Number of four-input LUTs	12,527 out of 26,624 (47%)	13,359 out of 26,624 (50%)	832 out of 26,624 (3%)
Number of block RAM	18 out of 32 (56%)	30 out of 32 (93%)	12 out of 32 (37.5%)[a]
Total power consumption	577 mW	604 mW	27 mW
Quiescent power	221 mW	230 mW	9 mW
Dynamic power	356 mW	374 mW	18 mW

[a]We have used as much block RAM as possible.

Table 2 Timing path and fan-out report for clock signal on Xilinx XC3S-1500 FPGA at 40 MHz

Parameter	LEON	LEON + PMU
Fan-out	2765	3080
Max delay, ns	1.262	1.287
Period requirement, ns	25	25
Minimum period, ns	24.533	24.817
Maximum frequency, MHz	40.761	40.295
Paths analyzed	7,124,581	7,946,407
Connections	54,333	57,114

obtained results related to resource consumption and power dissipation are shown in Table 1. Regarding the power consumption, the most significant increment is focused on dynamic power consumption, and this is due to the use of most of the embedded memory blocks.

As previously mentioned, the inclusion of the PMU did not produce any critical timing paths into the processor, and its performance is not degraded (tests carried out and its results are presented in detail in Sec. V). Clock distribution is a key issue in embedded systems and has vital importance in synchronous systems such as microprocessors and microcontrollers. The introduction of any delay, exceeding system constraints, results in the clock signal not reaching all the points at the same instant. Consequently, the system performance is degraded. Table 2 shows that all physical constraints of the architecture related to the clock were met.

Moreover, in Table 2 we can observe that the inclusion of the PMU produced a slight increment of fan-out and the maximum delay. The maximum frequency decreased but still meets with the system requirement.

IV. Test Environment

Before the exposition of the tests and the obtained results, a brief description of the development environment where the tests were carried out is presented. The target is the GR-XC3S-1500 board from Pender/Gaisler Research.

The core of this board consists of a XC3S1500 FPGA device from Xilinx Spartan 3 family (1.5 million gates), running at 40 MHz. This FPGA is where the LEON3 core is implemented. The operating system employed is RTEMS (Real-Time Executive for Multiprocessor Systems), a real-time operating system freely available. RTEMS is an open-source deterministic real-time operating system that is specifically aimed at embedded systems. It supports a wide variety of target platforms, such as PowerPC, Intel, SPARC, among others. RTEMS does not come alone; a development environment is also included. This environment is the RTEMS cross-compilation system that consists of a GNU cross-compilation C/C++ compiler, GNU binary utilities, GNU cross debugger, and bootprom utilities. The selection of RTEMS is rooted first by its support by ESA related to the RTEMS Centre [16] and second by its use in several ESA missions (such as Herschel/Planck, BepiColombo, or Gaia, to name a few) that make use of RTEMS on SPARC architecture onboard computers. The version used in this study is RTEMS version 4.6.

Part of the development/testing environment employed in this work is the TSIM instruction-level simulator. This simulator is capable of emulating LEON-based computer systems. TSIM provides several unique features, such as accurate and cycle-true emulation of LEON processors, 64-bit time for unlimited simulation periods, instruction trace buffer, nonintrusive execution-time profiling, and code coverage monitoring. TSIM can be configured to provide a simulation platform that is almost identical to the real hardware in such aspects as memory size, cache characteristics, I/O, etc. For every test performed in this study, a script was created to configure the simulator as similarly to the Pender board configuration as possible. TSIM and its trace capability may be very useful to count the number of instructions executed and to count load/store operations, among others, but TSIM is not useful to measure real-time performance. TSIM is a good tool to simulate the LEON3 architecture, but if we want to know the real results of a program execution, we need to execute it in the real platform. In the scope of our work, TSIM will provide the trace program execution results to be compared with the PMU results.

The program execution in the Pender board is controlled by the GRMON debug monitor for the LEON processors. GRMON communicates with the LEON DSU and allows nonintrusive debugging of the complete target system. GRMON accepts commands in form of a batch file. This feature is very useful to automate the tests.

V. System Testing

In this section we describe the tests performed in this study and its results. The tests have been divided in two parts. In the first part we check and validate the PMU; for this purpose, a set of standard tests was simulated and carried out. In the second part we take advantage of the designed PMU to check a real case of satellite software, such as the attitude and orbit control system (AOCS); this kind of software is usually one of the most complex and restrictive in space systems.

A. PMU Tests

The test suite consists of several typical benchmarks that cover most of the typical structure found in software for space applications. Most of these programs are extracted from the Stanford test suite.[‡‡] The instruction- and data-cache configurations for all the tests were fixed as one set of 4 kbytes and 32 bytes per line, with a random replacement algorithm. The tests had two objectives: first, to verify the nonintrusive behavior of PMU; second, to determine the system performance of both data- and instruction-cache hit rates. System performance was not degraded due to either the IU or cache system execution paths having been modified. To verify this fact, most of the test suite was simulated with Modelsim (functional and postlayout) in the original system and then including the PMU. During simulations we verified that executions were identical in both systems, including the number of instructions executed and time employed. Moreover, we compared execution time of all programs in a board with the PMU and in a board without the PMU, obtaining identical results. A brief description of these programs and their execution times with and without the PMU are shown in Table 3. Instruction- and data-cache hit-rate results are shown in Figs. 5 and 6 respectively. According to the results obtained, TSIM is very close to the PMU results and is therefore very close to those obtained in the real platform, which are the most important and difficult to get. In both cases, the difference between TSIM and PMU results were always under 5% difference, which is generally accepted as a typical value for TSIM tolerance.

B. Practical Test Bench

As previously described, the AOCS subsystem is one of the most critical components in a spacecraft. Its operations are carried out onboard, and in most cases, they must be performed with hard real-time restrictions. The software division of our research group has been involved in the development of an AOCS for a couple of satellites [17]. It was necessary to determine the most efficient configuration for both data and instruction caches. All the supported cache configurations were tested; the data- and instruction-cache hit rates are shown in Figs. 7 and 8.

The best results for the instruction cache were obtained with a configuration with 2 sets of 4 kbytes and 32 bytes per line, with LRU as replacement algorithm. For the data cache the configuration chosen was the same as that in the instruction-cache case, but the line size was modified to 16 bytes. This cache characterization has a vital importance in hard real-time systems. Through this method, we were able to reduce execution-time variability due to cache flush. Keeping down the number of cache misses produces a decrease on the number of cache flushes. Therefore, the system is able to execute the AOCS software with minimum interference due to cache flushing.

[‡‡]Private communication with J. Hennessy, June 1988.

Table 3 Benchmark test suite

Program	Description	Execution time in Pender board without PMU at 40 MHz	Execution time in Pender board with PMU at 40 MHz
bubble	Bubble sort method	3230 cycles ($\cong$ 80.75 ms)	3230 cycles ($\cong$ 80.75 ms)
data_loop_r	Multiple read access to memory	6010 cycles ($\cong$ 150.25 ms)	6010 cycles ($\cong$ 150.25 ms)
data_loop_w	Multiple write access to memory	7010 cycles ($\cong$ 175.25 ms)	7010 cycles ($\cong$ 175.25 ms)
fft	FFT calculation	43,470,070 cycles ($\cong 1,086,751.75$ ms)	43,470,070 cycles ($\cong 1,086,751.75$ ms)
fmn	Floating point matrix multiplication	308,890 cycles ($\cong$ 7722.25 ms)	308,890 cycles ($\cong$ 7722.25 ms)
imn	Integer matrix multiplication	17,170 cycles ($\cong$ 429.25 ms)	17,170 cycles ($\cong$ 429.25 ms)
perm	Data permutation	1,569,830 cycles ($\cong 39,245.75$ ms	1,569,830 cycles ($\cong 39,245.75$ ms)
prime	Prime number calculation	58,700 cycles ($\cong$ 1467.5 ms)	58,700 cycles ($\cong$ 1467.5 ms)
quick	Quick sort method	1330 cycles ($\cong$ 33.25 ms)	1330 cycles ($\cong$ 33.25 ms)
tak	Tak algorithm; highly recursive	$\cong 31,727.5$ ms	1,269,100 cycles ($\cong 31,727.5$ ms)

Cache-flush operations are also due to context switching. In most of the cases, a cache-flush operation is done when a context switch in the operating system takes place. The objective of this test was to determine the time penalty introduced by the interleaving of flush operations in the normal execution of a reference and to obtain accurate data- and instruction-cache characterizations in order to determine the WCET. In this manner, the context-switch impact is evaluated in the worst case. To determine this time, the following expressions have been applied:

$$T_{\text{flushpenalty}} = T_{\text{flush}} - T_{\text{original}} - N_{\text{int}} \times t_{\text{int(flush)}} \qquad (1)$$

$$N_{\text{int}} = \frac{T_{\text{flush}}}{\text{period}} \qquad (2)$$

The obtained data are those corresponding to the time used by RTEMS to handle the interruption when flush takes place (T_{int}(flush)). Having these data, it is easy to calculate the flush penalty. The data obtained are presented in Fig. 9. The data used in expressions (1) and (2) are circled.

From the results presented, it can be inferred that the longest period used the least difference in execution time. There are fewer flushes, and therefore the difference in execution time is lower. It is close to 1% for the RTEMS typical period of 10 ms. This conclusion is shown in Fig. 10. Moreover, for short periods (50 μs), although there are more flushes, they have less effect. This is due to the fact that the cache is disabled during cache flushing. For short periods, it is similar to having the cache disabled. That is why the execution time increases dramatically. For the longest period, the flush operation has

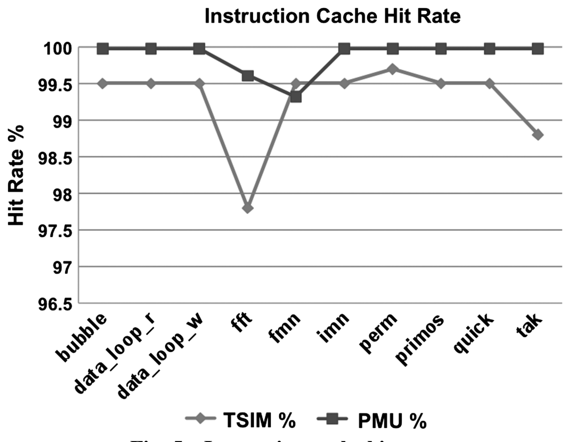


Fig. 5 Instruction-cache hit rate.

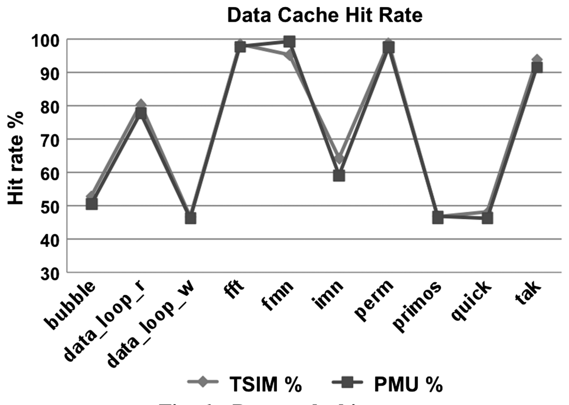


Fig. 6 Data-cache hit rate.

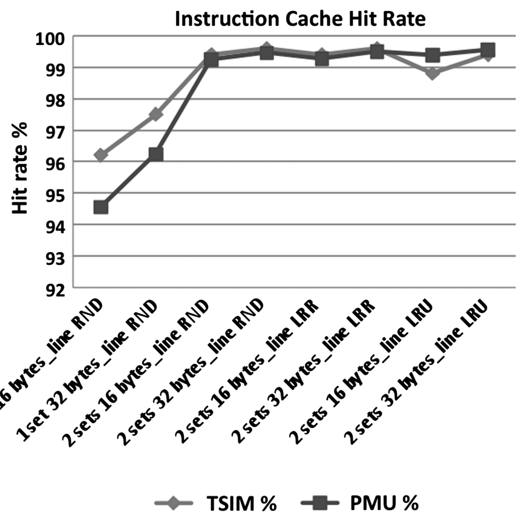


Fig. 7 Instruction-cache hit rate.

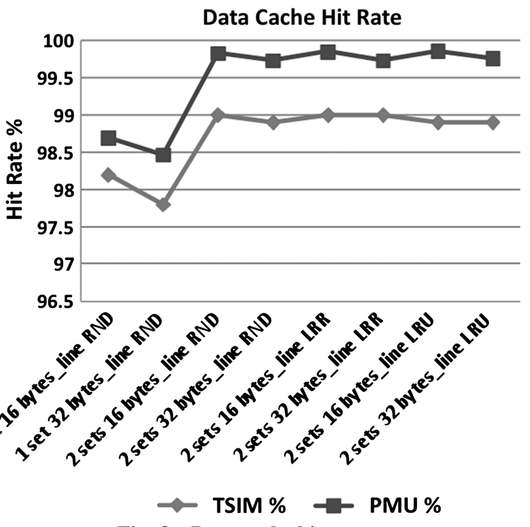


Fig. 8 Data-cache hit rate.

Bench execution time without interruptions (25 ns/cycle)	$T_{original}$ = 10038804 cycles (≈ 251 ms)					
Period (us)	50	200	500	1000	5000	10000
Period (cycles)	2000	8000	20000	40000	200000	400000
Interrupt Service Routine no flush						
Time taken (cycles)	251	251	250	250	247	242
Interrupt Service Routine with flush						
T_{int}(flush) (cycles)	808	809	791	795	816	845
Bench with flush						
T_{flush}	31040300	16351304	12289594	11144873	10259928	10155100
$T_{flushpenalty}$	8483960	4658904	1764325	884264	178692	94326
N_{int}	15492	2044	615	279	52	26
Cycles/Interruption	1356	3089	3661	3964	4252	4473

Fig. 9 AOCS bench results.

higher relative influence, as shown in Fig. 11. There are more items in the cache, and therefore every flush has more effect on execution.

Moreover, the CPI also was calculated through the PMU; its results are shown in Fig. 12. There are significant differences between the results obtained by the PMU and by the TSIM in this test. They can be due to the fact that the AOCS test uses I/O, and the TSIM is a simulator of a standard LEON architecture and probably does not properly take into account all the I/O operations delays.

The PMU was very useful to application developers to obtain more information about the program behavior. It provides accurate information about the number of taken branches and also about the execution path thanks to the branch trace buffer included. The PMU can provide user application developers a mechanism to complete their previous work and to obtain the most efficient configuration for the LEON3 processor.

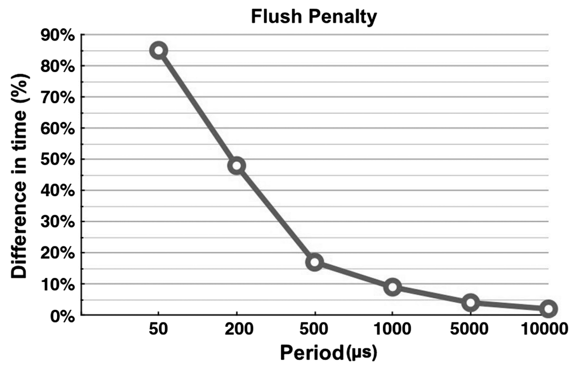


Fig. 10 Flush penalty.

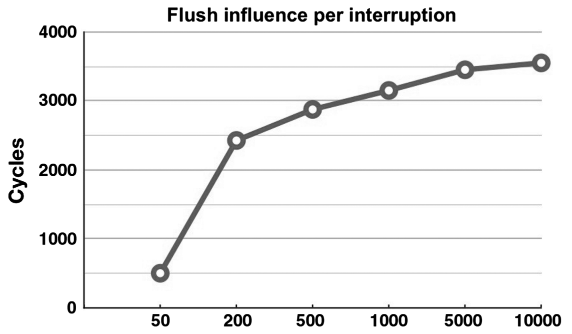


Fig. 11 Flush penalty per interruption.

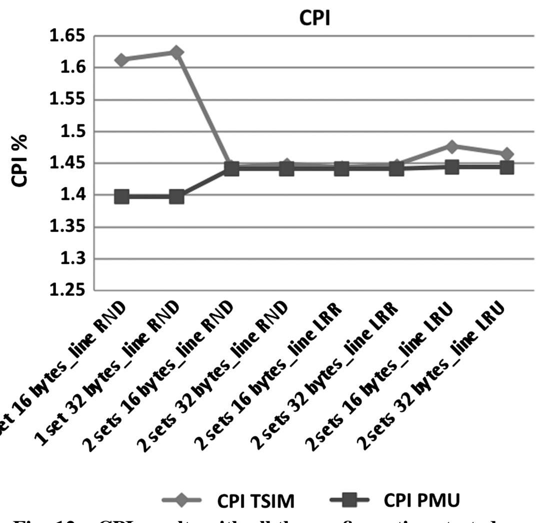


Fig. 12 CPI results with all the configurations tested.

VI. Conclusions

In this paper a performance-monitoring unit for the LEON architecture is presented. This development covers one of the most important missing features of this architecture. The enhancement is composed of performance counters for different events and a trace buffer. It provides a large amount of information related to the microprocessor behavior, including both instruction- and data-cache hit rates, number of instructions executed, number of branch instructions, and taken branches. It also allows the execution trace of a program or a piece of code to be obtained with minimal code instrumentation. With these new features, the final user can get much more information about the application behavior. This information could be used to perform full code coverage or to calculate the worst-case execution time in real-time systems, but is not limited to this; the system user can employ the new features as a new general-purpose service. Moreover, the processor performance is not degraded, due to the nonintrusive behavior of the new feature. The field-programmable gate-array resources employed by performance-monitoring unit is really low: 3% of slices, flip-flops, and lookup tables (LUTs). The number of embedded memory blocks used is bigger (37%), in order to take advantage of the field-programmable gate-array resources to obtain the biggest trace buffer possible. The power consumption increment is only 5% in comparison with the original architecture without the performance-monitoring unit.

With this improvement, users can collect information directly from the real target, which can be also complemented and checked with TSIM. The results provided by this solution offer more comprehensive information than that obtained solely by simulation. This is due to the fact that the system is stimulated by real input signals, whose behaviors are not always easy to simulate, as well as the fact that it is not easy to simulate the whole test environment. Unlike in the conservative solution in which the cache is disabled, by means of this enhancement and with the support of simulators, execution times in the LEON architecture are guaranteed, running real-time applications employing cache. To extend and reinforce this study, important efforts toward modifying the LEON architecture are being made, providing information related to wider kinds of events, such as number of context switches, traps managed, and advanced microcontroller bus architecture utilization. Moreover, the current version of the performance-monitoring unit is being improved, aimed at obtaining a better integration in the integer unit.

Acknowledgments

This work has been supported by the Ministerio de Ciencia e Innovación of Spain (grant AYA2009-13478-C02-02). We would like to express our gratitude to all of our teammates of the OL23 laboratory for their support. We would also like to thank Adolfo Figueroa and Doug Rowland for their reviews and suggestions. The authors are also grateful to the anonymous reviewers for their insightful and helpful comments.

References

[1] Tikir, M. M., and Hollingsworth, J. K., "Efficient Instrumentation for Code Coverage Testing," *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, Rome, 2002, pp. 86–96.

[2] Lu, S., Zhou, P., Liu, W., Zhou, Y., and Torrellas, J., "Path Expander: Architectural Support for Increasing the Path Coverage of Dynamic Bug Detection," *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Orlando, FL, 2006, pp. 38–52.
doi:10.1109/MICRO.2006.40

[3] Tran, A., Smith, M., and Miller, J., "A Hardware-Assisted Tool for Fast, Full Code Coverage Analysis," *Proceedings of the 19th International Symposium on Software Reliability Engineering*, Seattle, WA, 2008.
doi:10.1109/ISSRE.2008.22

[4] Shye, A., Iyer, M., Reddi, V. J., and Connors, D. A., "Code Coverage Testing Using Hardware Performance Monitoring Support," *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging*, Monterey, CA, 2005, pp. 159–163.

[5] Puschner, P., and Burns, A., "A Review of Worst Case Execution Time Analysis," *Real-Time Systems*, Vol. 18, Nos. 2–3, 2000, pp. 115–128.
doi:10.1023/A:1008119029962

[6] George, L., and Hermant, J. F., "Characterization of the Space of Feasible Worst-Case Execution Times for Earliest-Deadline-First Scheduling," *Journal of Aerospace Computing, Information, and Communication*, Vol. 6, No. 11, Nov. 2009, pp. 604–623.
doi:10.2514/1.44721

[7] Heckmann, R., Langenbach, M., Thesing, S., and Wilhelm, R., "The Influence of the Processor Architecture on the Design and the Results of WCET Tool," *Proceedings of the IEEE*, Vol. 91, No. 7, July 2003, pp. 1038–1054.
doi:10.1109/JPROC.2003.814618

[8] Rodriguez, M., Silva, N., Estives, J., Henriques, L., Costa, D., Holsti, N., and Hjortnaes, K., "Challenges in Calculating the WCET of a Complex On-Board Satellite Application," *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis, WCET 2003*, Porto, Portugal, 2003, pp. 11–15.

[9] Pellizzoni, R., and Caccamo, M., "Impact of Peripheral-Processor Interference on WCET Analysis of Real-Time Embedded Systems," *IEEE Transactions on Computers*, Vol. 59, No. 3, March 2010, pp. 400–415.
doi:10.1109/TC.2009.156

[10] Souyris, J., LePavec, E., Himbert, G., Jègu, V., Borios, G., and Heckmann, R., "Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation," *Proceedings of the 5th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007, pp. 21–24.

[11] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., et al., , "The Worst-Case Execution Time Problem-Overview of Methods and Survey of Tools," *ACM Transactions on Embedded Computing Systems*, Vol. 7, No. 3, April 2008, Paper 36.
doi:10.1145/1347375.1347389

[12] Bernat, G., Colin, A., and Petters, S. M., "pWCET: A Tool for Probabilistic Worst Case Execution Time Analysis of Real Time Systems," Department of Computer Science, Univ. of York, TR YCS353 2003, York, England, U.K., April 2003.
doi:10.1.1.14.8807

[13] Patrik, S., "Caches in Real-Time Applications—Benefiting from Caches in Timing Analysis Systems Using Next Generation Processors," *Proceedings of Data Systems in AeroSpace (DASIA)*, SP-630, ESA, 2006.

[14] Prieto, M., Guzmán, D., Meziat, D., Sánchez, S., and Planche, L., "LEON2 Cache Characterization, a Contribution to WCET Determination," *Proceedings of the IEEE International Symposium on Intelligent Signal Processing*, Alcalá de Henares, Spain, 2007.
doi:10.1109/WISP.2007.4447578

[15] Best, S., "Analyzing Code Coverage with GCOV," *Linux Magazine*, July 2003, pp. 43–50.

[16] Silva, H., Constantino, A., Freitas, D., Coutinho, M., Faustino, S., Mota, M., et al., "RTEMS Centre—Support and Maintenace Centre to RTEMS Operating System," *Proceedings of Data Systems in AeroSpace (DASIA)*, SP-669, ESA, 2009.

[17] Angulo, M., Mi, J. M., de Vicente, P., Prieto, M., Rodriguez, O., de la Fuente, E., and Palau, J., "Development of the MicroSat Programme at INTA," *Small Satellites for Earth Observation*, Springer, New York, 2008, pp. 41–54.
doi:10.1007/978-1-4020-6943-7

O. de Weck
Associate Editor