Methodology for timing characterisation of a LEON3 Numerical Emulator

William ARROUY [(1)], Christophe DUVERNOIS [(1)]

[(1)]*AIRBUS Defence & Space*
31 Rue des cosmonautes
31402 Toulouse Cedex 4
william.arrouy@astrium.eads.net
christophe.duvernois@astrium.eads.net

ABSTRACT

Nowadays, simulation based on processor numerical emulation executing real on-board software is widely used in order to prepare or validate spacecraft operations. On top of full functional processor instructions fidelity, current trend for such satellite simulation is mainly focusing on improving the execution speed taking benefits of technology such as multi-threaded or Just-In-Time execution. However, the major consequence is degrading the processor emulation timing fidelity, thus considering for example instruction timing based on statistics, or partial simulation of processor mechanisms; such as caches or instructions interactions. In the frame of on-board software validation on numerical bench such kind of deal between execution speed and timing fidelity degradation may be acceptable. However, when emulated software load is high, requires realistic timing fidelity for tasks sequencing or gets tight timing constraints, it becomes mandatory considering as a key element improvement of emulation timing fidelity as well as execution speed.

Based on the AIRBUS D&S LEON3 processor numerical emulator SimLEON, a timing characterisation phase has been led using a StarKit SCOC3 board. Execution results from same software suite execution on both the LEON3 hardware board and numerical emulator are compared. When a discrepancy is found, SimLEON is updated accordingly with the aim of improving its timing fidelity but also keeping in mind still having good execution speed in order not to impair its usability.

The objective of this paper is first to present the methodology used to perform this task, then its achievement in terms of SimLEON timing/execution speed performances and finally lessons learned for characterisation of next generation processors.

INTRODUCTION

Embedded software is now generally developed, tested and validated by AIRBUS D&S using numerical simulators. For obvious reasons, numerical simulators are relatively easy to deploy and low cost compare to hardware platform. It also proposes advanced functions such as debugging services that does not exist on hardware. Among the numerical simulator components, a key element is the processor emulator. On one hand, the current trend in processor emulator development is to focus on optimising its execution speed using various techniques (Instruction Cycles based on statistics, Just In Time, …) in order to be able to run simulations at several times real-time and consequently saving software testing time. On the other hand, embedded software are now more and more performing tasks leading to high Central Processing Unit (CPU) load and tight tasking constraints which should also been taken into account. Consequently, timing fidelity of the numerical processor emulator is now becoming as important as execution speed.

Modern processors are complex thus using specific hardware mechanisms to improve overall performances with at the end the purpose of increasing the number of instructions that can be executed by seconds. The usage of memory cache mechanisms, instruction pipelining, or branch prediction are examples of such processor features that improve the instructions throughput. Processor emulator must model those components and especially their effects on instruction cycle count in order to reach a good timing fidelity compared to hardware. As a consequence, it is necessary to fully understand the behaviour of these mechanisms. However, those functionalities are sometimes poorly documented and reverse engineering is required. This paper is then presenting the methodology which has been used to improve the timing fidelity of the AIRBUS D&S SimLEON LEON3 numerical emulator.

METHODOLOGY

This chapter describes in deep the methodology to characterise and reach good timing fidelity on processor numerical emulation. It especially describes how to use reverse-engineering in order to understand undocumented and hidden mechanisms of the processor including pipelines, caches and their sub-components as well as attached functions such as a Floating Point Unit (FPU) or I/O peripherals; all in an iterative and efficient way. It can be easily transposed and adapted to any processor family, either starting from scratch (e.g. with a simple model that counts one cycle per instruction, no matter what) or ignoring the existing documentation (which is sometimes not accurate enough) to finally build a model based on hardware observations.

The methodology was used (and refined) to characterize a SCOC3 numerical model developed by AIRBUS D&S which embeds a LEON3 emulator (SPARC v8 architecture [1]), using a SCOC3 StarKit board [2] as hardware reference. The debug facility on the hardware board is first ensured by the Debug Serial Unit (DSU) module which allows displaying accurate dated instructions and memory bus accesses and secondly thanks to a high precision timer counter which indeed gives; when read; the elapsed processor cycles. On the numerical side, a first version of SimLEON product is available and fully functionally representative of a LEON3 processor. The cache, pipelines and instruction timing are modelled based on the information available on the processor documentation [3] but with restrictions for some mechanisms or instruction combinations thus degrading its timing fidelity compared to hardware board.

First of all, the method requires setting up a test environment. It is composed of a large set of tests programs / benchmarks that covers a wide range and type of algorithms in order to reach a good set of the processor instructions combinations that could happen with real application software (for example algorithms that mainly uses the cache but often with some miss or others fully in the cache, …). It is better not to write such kind of programs but better to rely on a large set of benchmarks available on the market (test suite like Stanford, Powerstone, etc…) and adapt them to the target platform. Moreover, those benchmarks generally regroup a set of algorithms that can be easily split into several sub-programs or functions with the benefit of reducing the scope of investigation of a timing deviation.

Then, for each algorithm in the benchmark set, a specific software is written in order to run the benchmark function several times in a loop (let's consider 10 times). The number of cycles spent by each benchmark function call is measured and displayed. If such cycle counter feature is not available on the target platform, the most available precise timer must be used instead. The iteration process covers for the first iteration loop, the loading of the software to the instruction cache and for the following ones execution with only instructions cache hit. Indeed, to ease and simplify the instructions trace processing, it should be taken in to account that benchmark code is small enough to fully fit in the instruction cache. The reason for more than 2 iterations (one not in instruction cache, and one in) is to get an average timing value due to non-constant execution time on the hardware (e.g. because of SDRAM refresh impact). The benchmarks are first run on the hardware platform to get reference time and then on the emulator. A set of ten values is now available representing the number of CPU cycles needed to execute the same software on both hardware and numerical platforms. The error percentage is computed making the difference between the numbers of cycles spent on the hardware and emulation using the following formulae:

$$error\ \% = \frac{(cycles_{emulator} - cycles_{hardware})}{cycles_{hardware}} * 100 \quad (1)$$

Unless the emulator timing model is pessimistic (some instructions or memory access have extra cycles penalties), a negative value is generally output for almost all test case. On top of that it is more interesting to have negative values (emulator faster than hardware) than positive because it is generally easier to find missing cycles and update the emulator impacted algorithm that the other way…

Before starting the tests execution analysis; care shall be put on processing them in order to separate each complex component under characterisation. It is easier to first check a simple function than a complex one. As a result, in the first steps, the floating point algorithms are kept aside in order to focus on the pure integer ones. Indeed, the Floating Point Unit is a complex component that often comes with its own pipeline and that interacts with the Integer Unit (IU) one.

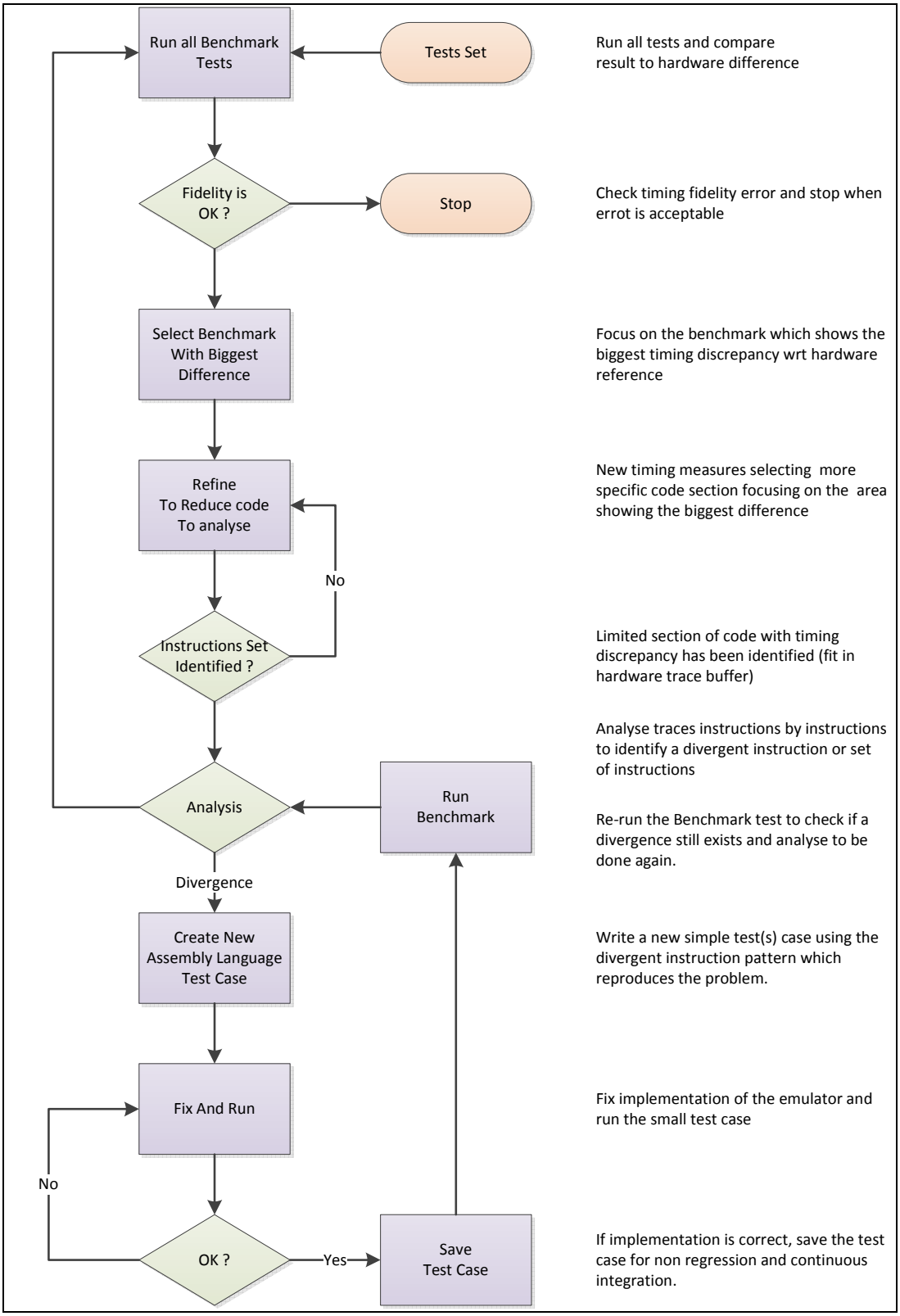

Fig. 1. Emulator Fidelity Enhancement Flowchart

Refining this same concept, the main idea is then to first check the IU focussing on the pipeline behaviour, followed by the data cache with the associated load/store mechanisms and finally with the instruction cache behaviour. More in details, it could be considered in a first batch, benchmarks without load/store instructions; then only pipeline mechanism(s) can affect the timing. From these it can easily be deduced the instructions and pipeline behaviour, that is to say detecting and understanding pipeline locks, optimisations or extra cycles due to combination of instructions (branches, jumps, …). In a second step, memory accesses (load/store) are introduced to check the timing model mainly focusing on the data cache and write buffer. Finally, as a last step, the instruction cache is considered focussing on the timings generated by the first iteration of the benchmarks (instruction cache misses). This is the best and theoretically ideal way to do but in real-world it is not so straight forward since those components are not completely independent.

In practice, it is helpful to split tests by functions (at least to isolate FPU ones), but it then quickly becomes more efficient to select the tests which are showing the most important deviation(s). If they are several, it is also more efficient and obvious to first select the one with the simplest and shortest algorithm. Indeed, as a benchmark may represent a huge amount of instructions executed by the CPU, it is necessary to reduce the scope of investigation. The first operation to perform is to display the number of cycles spent by each benchmark internal functions (and/or loop or code snippet) in addition to the benchmark time. The refining process must be continued (dichotomy method) iteratively. That is to say for a function or code section showing cycles count difference, the same process is applied measuring cycle counts of more and more limited code area until the amount of executed instructions showing a difference becomes a few hundred. This phase may additionally requires modifying some benchmark algorithm(s); for instance to reduce a loop iteration. The purpose of this “zoom” ; which leads to a reduced set of executed instructions; is to cope with the fact that the amount of instruction and bus traces the hardware can output is generally small. It is then important, to get relevant information on hardware trace which implies starting and stopping the instruction execution tracing at the right location (i.e. close to where lies the timing discrepancy). Finally, the emulator and hardware outputs are compared instructions by instructions. It should be mentioned that, to ease the identification of differences it is also helpful to get the emulator generating the same trace format as the hardware, thus allowing processing by automatic tools.

Now, the instruction or the set of executed instructions causing the timing difference has been identified. The next phase is to extract the pattern, to copy it to a small assembly language (small = a few tenths of instructions) test case in order to easily reproduce and investigate the timing issue (it should be pay attention that the assembly test case must be executed two times to avoid effect of instruction cache miss). This assembly test case purpose is to understand the timing issue root cause. As such, it can be modified, derived and extended to help finding the non-simulated mechanism or discovering new ones. As soon as the problem is understood and the correction performed on the emulator, the assembly test case is kept as reference (to be executed as non-regression). It should be paid attention to the fact that it is very important to make a clever and understood implementation and not to naively adapt the emulator to match the test result. Moreover, it is also important to keep in mind that fixing a timing issue only to get the test execution successful without understanding the underlying phenomena may incorrectly improve or degrade the fidelity of other tests with the consequence of making the convergence impossible. Additionally, a valid fix can also degrade the timing fidelity of other tests showing that there is still some timing inaccuracy to fix. From this point, it is recommended to re-run all the assembly test cases and benchmarks to check for non-regression after having updated the emulator implementation.

This process must be re-iterated on each benchmark up to getting all tests converging to high timing fidelity as described by the method flow chart (Fig. 1.).

After having tuned and fixed the emulator for the IU block, same process is applied to the remaining the floating point instructions. The focus shall be put on understanding how the FPU is interacting with the IU pipeline especially on memory/registers accesses. This may be a difficult task because the FPU is generally seen as a black box with little documentation on its internal behaviour. On top of that, an important number of cases shall be considered depending of the existence of a parallel or serial FPU pipeline, instructions parallelism inside the FPU or parameter dependant instructions execution cycle time. However, applying the same procedure as for the IU, that is to say, refining the cycle count difference display up to identifying a reduced set of instructions; allows finding all the information about the FPU pipeline, instructions and mechanisms.

APPLICATION

This method was applied to improve the fidelity of the SimLEON3 emulator especially focussing on the FPU [4] unit but also on the IU. The fidelity improvement task duration was quick. Before this improvement phase, the emulator had 92% fidelity compared to hardware reference (i.e error% = +/- 8%) using as reference benchmark the Stanford test.

At the end, we achieve very good performance reaching more than 99.5% (error 0.5% - worst case i.e. greater deviation compared to hardware results) fidelity for pure Integer Unit tests and more than 98.4% (error 1.6% - worst case) when the FPU is used as shown by (Fig. 2). Here, the error % is displayed per benchmark showing; the first execution (instruction cache loading) and for the 9 following executions (software fully in instruction cache) the worst (greater deviation compared to hardware of the 10 runs) and average (mean value of the 10 executions). Detailed results are also presented hereafter in (Tab. 1.)

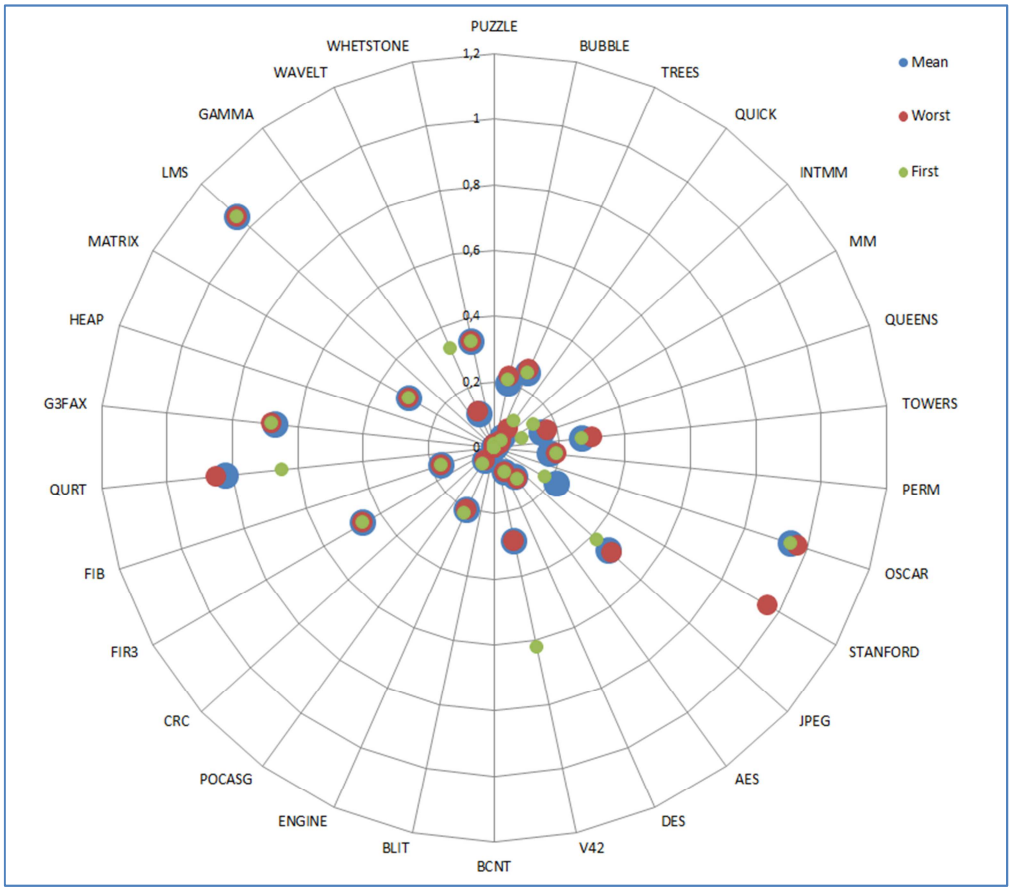

Fig. 2. Timing Dispersion (error %) Overview per Benchmark

The 100% fidelity is not reached due to three main elements: first, timings are also not constants on hardware (for example impacts of SDRAM refresh on load instructions); secondly, division and square root FPU instructions timing are operands dependant, then an average value is used by the emulator timing model; and finally because has been encountered very rare and complex phenomenon based on a large combination of instructions we do not model. For these latter ones, it is deliberately chosen not to implement them because they are quite complex to model with significant impacts on the performances (speed) and poor fidelity improvement (estimated to less than 0.02%).

The following table (tab. 1.) gives an extract of some used benchmark results showing the test name, the average error (over 10 iterations), the worst error case (worst of the 10 iterations), the first iteration error (instruction cache miss), the test total duration in seconds (time the benchmark execution lasts) and finally a test description summary. The two main

contributors to deviations are the SDRAM refresh impact on load instructions (up to 1% for the IU and FPU) and floating point division / square root instructions which are modelled using an average number of cycles.

Tab. 1. Results of some benchmark tests comparison between SimLEON3 and StarKit board

Test	Avg %	Worst %	First %	Duration (s)	Description
PUZZLE	0,00	-0,01	-0,01	4,241	Puzzle
BUBBLE	-0,20	-0,22	-0,21	1,147	Bubble Sort
TREES	-0,25	-0,26	-0,25	6,642	Tree Sort (Dynamic Allocation and Linked List)
QUICK	-0,04	-0,07	-0,10	0,937	Quick Sort
INTMM	-0,01	-0,03	-0,03	0,908	Integer Matrix Multiplication
MM	0,00	-0,02	-0,14	1,029	Real Matrix Multiplication
QUEENS	-0,15	-0,17	-0,09	0,942	9 Queens Problem
TOWERS	-0,27	-0,30	-0,27	1,306	Hanoi Towers
PERM	-0,17	-0,19	-0,19	1,015	Permutation (recursive)
OSCAR	-0,95	-0,97	-0,95	1,208	FFT / Cosine
STANFORD	-0,22	-0,96	-0,18	15,041	Stanford Benchmark
JPEG	-0,47	-0,48	-0,42	1,76	JPEG 24-bit image decompression
AES	-0,11	-0,12	-0,12	13,045	Advance Encryption Standard
DES	-0,08	-0,08	-0,08	1,677	Data Encryption Standard
V42	-0,29	-0,29	-0,62	1,278	Modem Encoding/Decoding Compression
BCNT	0,00	0,00	0,00	2,063	Bit shifting & anding through 1K array
BLIT	0,00	0,00	0,00	8,576	Graphics Application
ENGINE	-0,21	-0,21	-0,22	14,135	Engine Control Application
POCASG	-0,05	-0,05	-0,06	1,669	POCASG paging communication protocols
CRC	0,00	0,00	0,00	6,791	Cyclic Redundancy Check
FIR3	-0,46	-0,46	-0,46	8,776	Integer FIR Filter
FIB	-0,17	-0,17	-0,17	10,183	Fibonacci (recursive)
QURT	0,82	0,85	0,65	0,117	Square Root Calculation using Floating Point
G3FAX	-0,67	-0,68	-0,68	0,595	Group 3 fax Decode
HEAP	0,00	0,00	0,00	17,899	Heap Sort
MATRIX	-0,30	-0,30	-0,30	29,051	Integer Matrix Multiplication
LMS	-1,05	-1,05	-1,05	16,268	LMS Filter Algorithm (data arrays set larger than data cache can contain)
GAMMA	0,00	0,00	0,00	12,402	Gamma Function
WAVELT	-0,11	-0,12	-0,33	0,173	Wavelet
WHETSTONE	-0,33	-0,33	-0,33	5,722	Whetstone Benchmark
COS	-0,75	-0,75	-1,26	0,118	Cosine (loop on some values)
SIN	-0,22	-0,22	-0,61	0,115	Sinus (loop on some values)
ACOS	-0,3	-0,31	-0,35	0,202	Arc cosine (loop on some values)
EXP	-1,45	-1,45	-1,46	1,068	Exponential (loop on some values)
LOG	-1,6	-1,6	-1,61	0,989	Logarithm (loop on some values)
POW	-1,35	-1,36	-1,46	0,19	Power function (loop on some values)
SQRT	-0,58	-0,58	-0,6	0,096	Square root (loop on some values)

The other important point is the emulation performance; that is to say how fast the emulation can be executed compared to real-time. Despite of the fidelity improvement we also manage to slightly improve it compared to previous SimLEON version. The execution of the Stanford Test; as a reference; on Intel® Xeon® X5860 at 3.33GHz emulating a 32MHz LEON3 processor reaches a factor 11.5(i.e. tests execution duration is 11.5 times faster on SimLEON than on hardware board). This figure is also confirmed by execution of AIRBUS D&S SeoSat Satellite Simulator (including all equipments and physical models) in full operation mode; with full redundancy; emulating unmodified the SeoSAT Central Software.

CONCLUSION

A full characterisation of processor timing mechanisms requires a lot of reverse-engineering and investigations at the lowest level of the processor: instruction traces, spying the memory bus, cache inspection, generation of low level assembly languages tests … This task can be quite long and complex but it is mandatory in order to get or increase confidence on software validation based on numerical benches especially when CPU load or tight tasking constraints are to be tested. It has also shown the importance of selecting a wide range of benchmarks for the tests set in order to cover most of the hardware phenomenon. Despite, this process looks empirical it has been proved very efficient and re-usable for any type of processors

The methodology presented in this paper has been successfully implemented to complete a LEON3 (including its FPU and one I/O Bus) characterisation reaching high level of fidelity; -0.96% error – worst case on Stanford Reference Benchmark; thus keeping good execution performances with a real-time execution factor of 11.5 on this Reference Benchmark.

REFERENCES

[1] The SPARC Architecture Manual Version 8 – Revision SAV080SI9308

[2] AIRBUS Defence & Space - SCOC3 StarKit: http://www.space-airbusds.com/en/equipment/scoc3.html

[3] Aeroflex - UT699 LEON 3FT/SPARC V8 MicroProcessor™ Functional Manual

[4] Aeroflex Gailser - IEEE-STD-754 Floating Point Unit GRFPU/GRFPU-FT Companion Core Data Sheet