FPGA-based platform for fast accurate evaluation of Ultra Low Power SoC

Guillaume Patrigeon, Pascal Benoit, Lionel Torres

FPGA-based platform for fast accurate evaluation of Ultra Low Power SoC

Guillaume Patrigeon, Pascal Benoit, Lionel Torres
LIRMM
University of Montpellier, CNRS
Montpellier, France
firstname.lastname@lirmm.fr

***Abstract*—Accurate evaluation of Ultra Low Power Systems on Chip (ULP SoC) is a huge challenge for designers and developers. In embedded applications, especially for Internet of Things end-node devices, ULP SoCs have to interact with their environment and need self-management. For this kind of applications, modelling a complete SoC, including processor(s), memories, all the peripherals components, their interaction and low-power policies, can be very complex in terms of developments and benchmarking. In order to cope with this challenge, an approach is to implement the desired system on FPGA with a monitoring infrastructure dedicated to fast and accurate performance evaluation. In this paper, we propose a set of different tools used during the evaluation step that can also be easily implemented on the final product and used by the system itself for self-assessment to enable adaptive behaviour. Illustrated by a simple architecture implemented on an FPGA-based platform, this method brings flexible, cycle accurate, fast and reliable performance evaluation and self-evaluation, with the possibility to use the platform for low-cost prototyping.**

***Index Terms*—FPGA, Cortex-M0, performance evaluation, cycle accurate, self-monitoring**

I. Introduction and Related Works

Nowadays, embedded systems are widely used in various domains, and many applications set constraints for designers and developers in terms of performance, energy consumption and security. The permanent need of a smaller, energy efficient SoCs leads to explore new technologies, architectures and techniques at every levels of the development of these systems. To perform good design space exploration with these new technologies and methodologies, it is very important to make fine-grained accurate evaluations of these new designs. However, most of the current available tools for design space exploration target high performances systems.

There are different methods and tools used for performance evaluation: RTL simulation, Instruction Set Simulator (ISS), Cycle Accurate Model (CAM), hardware evaluation. RTL simulation allows following each signal of a design and so to make a fine-grained evaluation of it, unfortunately, this method is slow. ISS is faster than RTL simulation, but it only reproduces the design in a functional way: it is not accurate [1] and not exempt from bugs. CAM is a good solution for software evaluation of processors. It is faster than RTL simulation and more accurate than ISS. However, there are still some issues by using these tools: it requires that a current CAM exists for the targeted processor (for example, ARM does not provide a CAM for its Cortex-M0 processor yet), and some signals that could be useful for design space exploration are not available at the CAM output. When performing design simulation, the more accurate the solution is, the slower it is [2]. Moreover, in an embedded application context (especially for IoT applications), the final product has to interact with its environment (directly or via other components) and these interactions may be hard to reproduce by software simulation.

Some of the processors designed for ULP SoCs include trace modules. They are powerful tools used for code analysis and optimization. Trace tools are accurate and are able to stream out executed instructions, which can be used for activity analysis. [3] shows how to use the embedded trace module of a Cortex-M4 based product to retrieve accurate performance data and check the accuracy of an emulator. However, every processor does not always include such tools, and even if the processor vendor offers this option, the manufacturer may include it or not. In addition, these tools are centred on the processor and dedicated to software purposes.

When looking at high performance processors, some of them include a Performance Monitoring Unit (PMU) used for performance analysis. A PMU includes a set of counters that can be programmed to capture events. The counters values are then read by software to make a performance analysis [4]. If PMUs are great tools for software optimization, they can also be used for adaptive behaviours. [5] demonstrates how it is also possible to create a power monitor by using a PMU and a power model of the processor. However, the number of hardware counters is limited and the available events are centred on the processor.

Another way to perform evaluation of a design is to implement it on FPGA including monitoring probes: [6] shows an implementation of a PMU in an FPGA-based LEON3 platform, but the presented solution is also centred on the processor.

Using an FPGA-based platform, every signal of the implemented design is reachable, like RTL simulation but while running at higher speed. Monitoring tools can be designed and dedicated to the studied parts, and custom modules can be added in the design. Also, a lot of ULP processors run at a low frequency (generally below 200 MHz), so it is possible to run applications at real time and enable interactions with external products (like sensors or radio modules for IoT applications).

That is why we decided to use an FPGA for design space exploration of ULP SoC.

The work we present in this paper is part of a project that aims the exploration of new technologies for ULP SoC. Section II introduces our design space exploration flow using an FPGA-based platform. To illustrate our work, we have implemented a ARM Cortex-M0 based architecture, presented in Section III, which includes an activity monitor that will capture the events related to the system memory. In Section IV we show how this activity monitor is controlled by software to obtain data about a selected portion of code. Finally, we present in Section V an example of energy consumption estimation of a 28-nm FDSOI SRAM with the EEMBC CoreMark and a set of encryption algorithms.

II. Evaluation Flow

Fig. 1 presents the exploration flow used in this work. The specifications of the final system define the software and hardware needs. From the hardware definition, we create an RTL design of the desired system that will run on an FPGA-based platform. Simulations are done for unitary validations to ensure that the behaviour of each block matches the one desired.

Power consumption models are generated from hardware simulation results and/or real hardware implementation measurements. As some events are more representative of the energy consumption of a hardware block than others, they will be captured by a dedicated activity monitor included in the system. The energy consumption is then estimated from the report of this activity monitor and power consumption models of each studied block. The signals that will be captured by the monitor are identified and selected regarding their function and their correlation with the energy consumption. They can be identified by using data mining methods.

As the activity monitor is included in the system, the models can be integrated in the application code to enable self-evaluation by the device itself, and then provide adaptive behaviours.

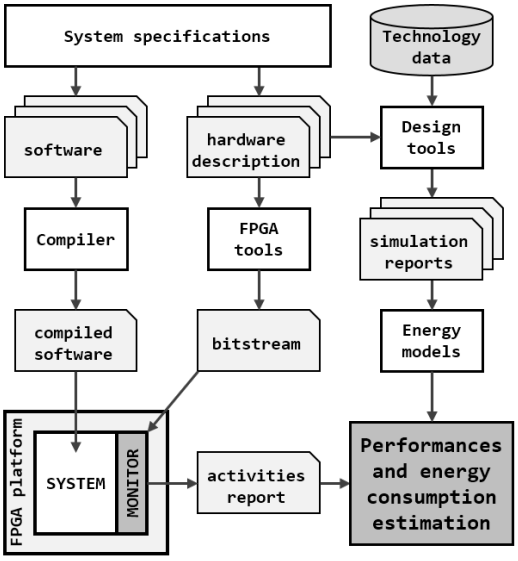

Fig. 1. Evaluation flow

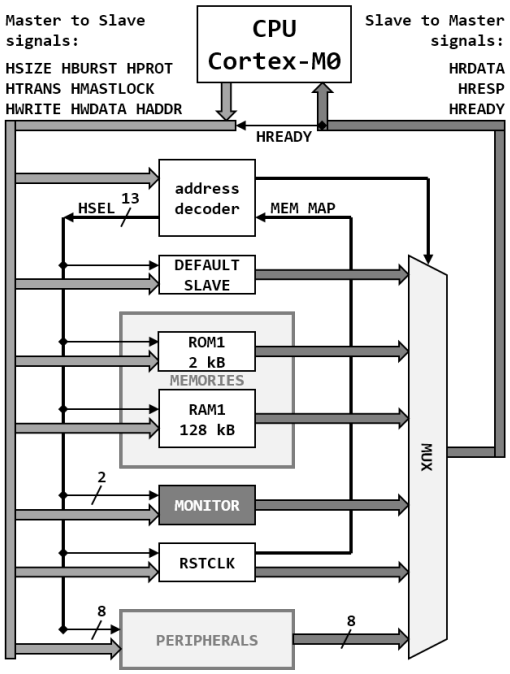

Fig. 2. Block diagram of the architecture

III. Architecture Overview

The benchmarks used for this evaluation (presented in Section V) do not need any specific hardware features other than a CPU and a memory (for code and data), that is why the system implemented on the FPGA board is simple.

As the ARM Cortex-M processors are widely used in ULP SoC, we decided to use one as the processor of our system so we can use existing products as references for performance evaluation comparison. In addition, Cortex-M0 and Cortex-M3 are available with the ARM DesignStart program.

The architecture used here in Fig. 2 is composed of the ARM DesignStart Cortex-M0 r1p0 processor, a 2 kB ROM (containing a bootloader code), a 128 kB RAM (used for code and data), some peripherals for Input/Output control and serial communication (UART), a Reset and Clock control block and the minimal AMBA3 AHB-Lite single-master system configuration as described in [7]: a default slave, an address decoder and a bus multiplexor.

The peripheral called *Monitor* is used to report events and will serve as basis for the design exploration flow.

A. ARM Cortex-M0 processor

The Cortex-M0 is a 3-stage, 32-bit RISC processor which implements the ARMv6-M ISA. It includes a single AHB-Lite interface, 32 interruption lines (r1p0 only), 1 Non-Maskable Interrupt (NMI) and a single-cycle multiplier (r1p0 only).

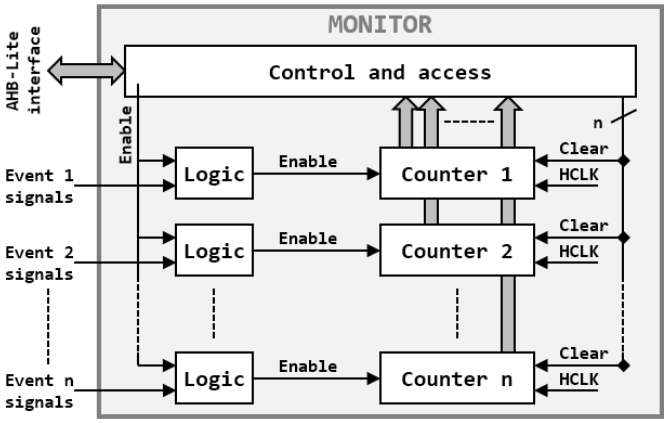

Fig. 3. Monitor block diagram

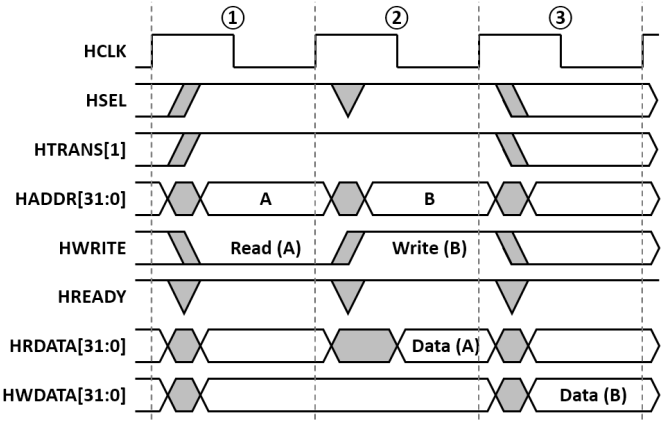

Fig. 4. AHB-Lite read and write sequences. Cycle (1): address phase A. Cycle (2): data phase A, address phase B. Cycle (3): data phase B

The ARM DesignStart Cortex-M0 r1p0 is a fixed configuration of the Cortex-M0 processor, it is delivered as a netlist synthesizable on FPGA. However, this version does not include debug modules nor low power mode.

B. Activity Monitor

The activity monitor is a set of counters used to capture events. Its architecture is based on the principle of PMU, as described in Fig. 3.

The activity monitor is designed to capture the following events related to the memory:

- Number of cycles
- Number of executed instructions
- Number of instruction fetches
- Number of RAM read accesses
- Number of RAM write accesses

The activity monitor is connected to the AHB-Lite bus system, and can be accessed by the processor as a peripheral. By connecting it to the main bus, it is possible to start, stop and reset the counters by software using a control register. Activity monitoring can be performed for a selected portion of code without adding external control hardware.

For cycle counting, a simple counter, always enabled, is used. The instruction counter is incremented each time the program counter (PC) changes. The RAM counters are incremented when a valid RAM access is detected. As it uses the same physical bus as the RAM, the Activity Monitor use its own bus interface to detect RAM operations (Instruction fetches, data reads and writes with different sizes). If the monitor was on a different bus (which could have been the case in a multi-master architecture), the corresponding RAM bus interface signals would have been simply routed to the monitor.

RAM accesses can be detected when the following logic condition is respected:

$$RAMAccess = HSEL[RAM] \cdot HREADY \cdot HTRANS[1]$$

Where *HSEL[RAM]* is the select line assigned to the RAM (active high), *HREADY*, when high, indicates that the previous transfer is completed, and *HTRANS[1]* indicates a transfer request when high.

There are counters used for each RAM transfer type: instruction fetch counter, data read counter, and data write counters. To determine the type of a transfer, the monitor look at two signals: *HPROT[0]*, indicating if the transfer is an instruction fetch (low) or a data transfer (high), and *HWRITE*, which indicates the transfer direction (read when low, write when high). Fig. 4 shows an example of the AHB-Lite protocol, with a read transfer followed by a write transfer.

An instruction fetch occurs when the following logic condition is respected:

$$RAMFetch = RAMAccess \cdot \overline{HPROT[0]} \cdot \overline{HWRITE}$$

A RAM read access is defined by:

$$RAMRead = RAMAccess \cdot HPROT[0] \cdot \overline{HWRITE}$$

A RAM write access is defined by:

$$RAMWrite = RAMAccess \cdot HPROT[0] \cdot HWRITE$$

As the Cortex-M0 supports 8, 16 and 32-bit write operations, 3 counters can be used. The *HSIZE[2:0]* vector indicate the size of the transfer.

$$RAM8bitWrite = RAMWrite \cdot (HSIZE = \text{“000”})$$
$$RAM16bitWrite = RAMWrite \cdot (HSIZE = \text{“001”})$$
$$RAM32bitWrite = RAMWrite \cdot (HSIZE = \text{“010”})$$

The other values of HSIZE, from “011” (64-bit transfer) to “111” (1024-bit transfer) are allowed by the protocol, but are not supported by this system.

Each counter is 32-bit wide. The size of the counters depends on the desired use. 32-bit counters will overflow after 86 seconds at 50 MHz (ARM gives 50 MHz as the maximum frequency supported by the Cortex-M0 processor).

C. Extra peripherals

Some extra peripherals were added to the system: Reset and Clock management, Input/Output control blocks (*General Purpose Inputs and Outputs* (GPIO) and *Peripheral Pin Selection*

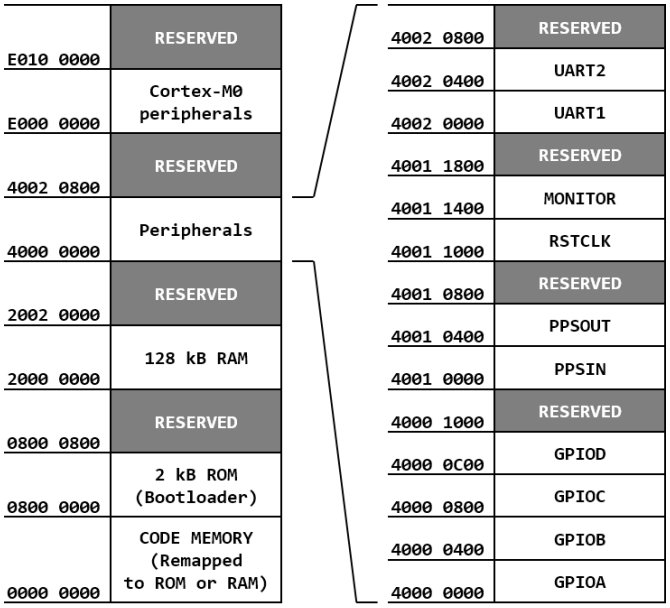

Fig. 5. Memory mapping

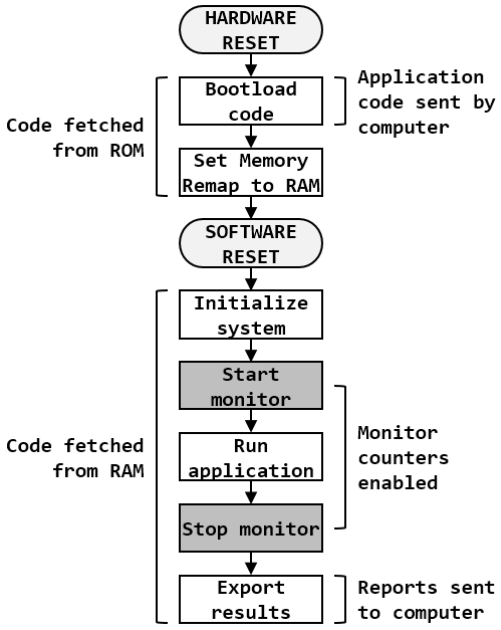

Fig. 6. Basic system initialization and run sequence

(PPS)) and serial interfaces (*UART*) to allow communications between the system and a computer.

More peripherals could be added to create a full prototyping platform (Timers, *Inter-Integrated Circuit* bus (I^2C), *Serial Peripheral Interface* (SPI) for example).

D. Memory organization

To make accessible and manageable the monitor by the software, the monitor is connected to the system bus with a standard AHB-Lite interface, and is viewed as a classic peripheral. A memory region is assigned to the monitor to access the control register and the counters value.

To simplify application code loading on the board, a bootloader code is present in a small ROM in the FPGA. The boot memory region at address 0000 0000h is remapped to ROM or RAM regarding a configuration register of the Reset and Clock Control peripheral (*RSTCLK*).

IV. Software Interface

The memory mapping of the system is presented in Fig. 5. The bootloader code inside ROM is used for loading the application code without extra hardware nor regenerating the bitstream. The application code is sent to the platform by the computer with a software written in Python. The bootloader code changes the memory mapping after writing the application code inside the memory and then performs a software reset. The boot memory mapping is selected by using the Reset and Clock Control block.

If a hardware reset occurs when the application is running, the system will restart with the bootloader code. If a code was previously loaded, it restarts with this code after 1 second if no new code is sent to the bootloader.

When running, the application code initializes the platform, runs code and performs monitoring by using the activity monitor. Then, it retrieves the monitor's counter values and can share them or process the results.

Fig. 6 shows the basic sequence to perform activity monitoring on a portion of code.

As previously described in Section III, the software can control the activity monitor by writing into its control register, like the others peripherals. However, depending the implementation and the software optimization, this operation may add few extra cycles, instructions and RAM accesses to the counters.

V. Experimental Results

A. Resources

The system presented in Section III is not a very complex one. The Cortex-M0 is the most important part of this design (in term of size) and this is still a small processor. The Cortex-M0 also set the (theoretical) maximum frequency to 50 MHz.

We choose a small, low-cost FPGA-based circuit, large enough for our system and its future evolutions, and which can easily be put on breadboards or on printed circuits to create a low-cost prototyping platform for ULP embedded devices: the Digilent Cmod A7.

The Digilent Cmod A7 is a small, 48-pin DIP form factor board built around a Xilinx Artix-7 FPGA. The Cmod A7 35T use a Xilinx XC7A35T-1CPG236C FPGA, which has 20 800 LUTs, 41 600 Flip-Flops, 225 kB of RAM and 1 MSPS ADC. The board includes a USB-JTAG programming circuit, a USB-UART bridge, a 12 MHz clock source, a 512 kB CSRAM and a 4 MB Quad-SPI Flash. As it uses a single USB port for both FPGA programming and UART bridge, no extra hardware is required for communication.

Synthesis is done using Vivado 2017.4 Synthesis tool, with Vivado default pre-set. Implementation is also done is done using Vivado 2017.4 Implementation tool, still with Vivado default pre-set. The Table I shows the resources utilization for the whole implementation and the resources utilization of the Cortex-M0 and the Activity Monitor.

TABLE I
RESOURCES UTILIZATION

Full system			
Site Type	**Used**	**Available**	**Ratio**
Slice LUTs	4 030	20 800	19.38 %
Slice registers	1 992	41 600	4.79 %
Block RAM Tile	32.5	50	65.00 %
Cortex-M0			
Site Type	**Used**	**Available**	**Ratio**
Slice LUTs	2 997	20 800	14.41 %
Slice registers	890	41 600	2.14 %
Block RAM Tile	0	50	0.00 %
Activity Monitor			
Site Type	**Used**	**Available**	**Ratio**
Slice LUTs	200	20 800	0.96 %
Slice registers	306	41 600	0.74 %
Block RAM Tile	0	50	0.00 %

B. Hardware reference

To validate the designed system and to ensure that future evaluations using this platform are reliable, a comparison to a commercial circuit was made. The STM32F072 Nucleo-64 board from STMicroelectronics is used as hardware reference. The STM32F072RBT6 microcontroller of this board includes a Cortex-M0, a 128 kB Flash and a 16 kB RAM.

For this microcontroller, the Flash memory has no wait state under 24 MHz [8]. Above 24 MHz, the Flash is not able to provide data on time and one latency cycle must be added to ensure that timings constraints are respected. As this wait state will have an impact on time execution, the test will be done at a frequency where the Flash memory can be read as fast as the RAM (12 MHz for this test). Even if some accelerators are available to fetch code from Flash faster, wait states may still be inserted during branches and constant data reads. Another way to avoid this issue is to execute code directly from RAM.

For timing measurement, an internal timer is used. It is configured to provide time without generating interrupts during the code execution, but the measure may be impacted when retrieving time (at the beginning and the end of the test code), as the timers do not have the same implementation on the STM32F072 and our system on FPGA.

C. Runtime verification

Both platforms (Cmod A7 and STM32F072 Nucle-64) run the test applications at 12 MHz. Both codes are compiled with *gcc-arm-none-eabi 6.3.1 build 20170620* compiler, with optimization flag *-O3*. Some code is added before and after each test code to initialize the platforms and send information.

Four application codes were tested to validate the implementation of the system: EEMBC CoreMark, and software implementation of AES, GIFT ([9]) and PRESENT ([10]) encryption algorithms. For EEMBC CoreMark, ARM recommends a minimum execution time of 30 seconds [11] to obtain valid results, which can be reached with 700 iterations in this configuration.

Table II presents the results for each test code.

TABLE II
RUNTIME RESULTS

Settings		
Platform	Cmod A7-35T	STM32F072 Nucleo-64
Hardware	XC7A35T-1CPG236C	STM32F072RBT6
CPU	DesignStart Cortex-M0	Cortex-M0
Frequency	12 MHz	12 MHz
Compiler	GCC 6.3.1 20170620	GCC 6.3.1 20170620
Flags	-O3	-O3
Code location	RAM	Flash
CoreMark		
Iterations	700	700
Runtime (ms)	31 740	31 740
CoreMark	22.054	22.054
CoreMark/MHz	1.838	1.838
AES - 128-bit key, 128-bit data, 11 rounds		
Iterations	10 000	10 000
Runtime (ms)	7 278	7 278
GIFT - 128-bit key, 64-bit data, 28 rounds		
Iterations	1 000	1 000
Runtime (ms)	4 262	4 262
PRESENT - 128-bit key, 128-bit data, 31 rounds		
Iterations	1 000	1 000
Runtime (ms)	4 870	4 870

Both platforms run each software (CoreMark, AES, GIFT and PRESENT) in the same time, In addition they have the same score of 1.838 CoreMark/MHz. This test validates that the Cortex-M0 based system is correctly implemented on the Cmod A7 board.

D. Activity Monitor reports

The activity monitor starts after system initialization, just before starting test code iterations (when calling the *start_time* function for CoreMark code), and stops after the last iteration (when calling the *stop_time* function for CoreMark code). As example, Table III shows the results obtained after returning from the CoreMark main function.

The following formula verifies that the number of cycles matches the execution time:

$$NumberOfCycles/Frequency = RunTimeInSecs$$
$$380883509/12000000 = 31.740292$$

In this report, the number of instruction is not equal to the number of cycles (44.9 % more cycles). This is due to multi-cycles instructions (loads, stores, branches...).

TABLE III
REPORT FOR COREMARK

	Value after 700 iterations	Average for 1 iteration
Cycles	380 883 509	544 119
Instructions	262 804 036	375 434
Fetches	176 819 289	252 599
Reads	44 484 235	63 549
Writes (total)	12 839 653	18 342
8-bit Writes	243 600	348
16-bit Writes	504 791	721
32-bit Writes	12 091 262	17 273

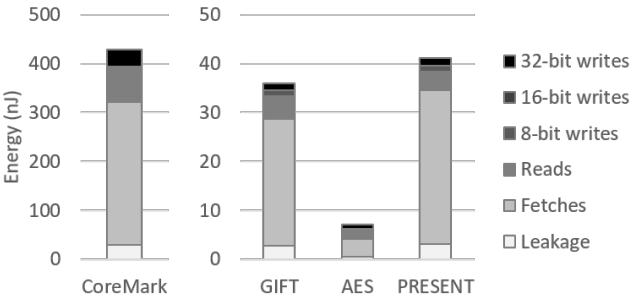

Fig. 8. Memory energy consumption repartition (average for 1 iteration)

The Cortex-M0 uses mostly 16-bit instructions from the Thumb 2 ISA, and uses few 32-bit instructions. In addition, when a branch occurs, the previously fetched code waiting to be executed is lost. That is why the number of fetches does not match the half of the number of executed instructions (as there are mostly 16-bit instructions, 2 instructions can be fetched in one read).

These results also show that a cycle accurate model which takes account of all penalties (wait states, multi-cycles instructions, code fetched lost after branches, etc...) is required for accurate performance evaluation (of a Cortex-M0 based system in this case).

We can now use these results to make a comparison of the different encryption algorithms (Fig. 7). For example, Fig. 8 shows an energy consumption estimation of the memory, using values of a 16 kB SRAM in 28-nm FDSOI from [12].

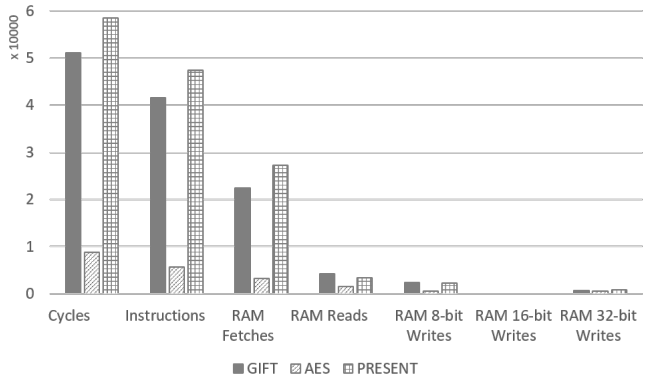

Fig. 7. Results for software implementation of AES, GIFT and PRESENT (average for 1 iteration)

VI. CONCLUSION AND PERSPECTIVES

In this paper, we have presented an FPGA-based platform for fast, real-time, accurate and low cost evaluation of ULP SoC. We have demonstrated, on CoreMark and ciphers benchmarks, that this approach was fully reliable for the given purpose as the obtained results are comparable to commercial products: the performance assessment is done in real-time with an accuracy close to 100 %. Furthermore, the activity monitor can be exploited for architecture optimization and exploration purposes, as demonstrated with RAM energy consumption estimations. By allowing interactivity with its environment, the platform used here can serve as a controller for embedded system prototyping. If the activity monitor we presented is designed for RAM evaluation, it can be easily customized and adapted as wished. The flexibility provided by the FPGA makes possible to capture events that could not be available in a software model, with an accuracy similar to RTL simulations but with real time execution.

ACKNOWLEDGMENT

This work has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 687973 (GREAT project) and the French National Research Agency under grant ANR-15-CE24-0033-01 (MASTA project).

REFERENCES

[1] S. Fontaine, L. Filion, and G. Bois, "Exploring ISS Abstractions for Embedded Software Design." IEEE, 2008, pp. 651–655.
[2] T. Rissa, A. Donlin, and W. Luk, "Evaluation of SystemC Modelling of Reconfigurable Embedded Systems." IEEE, 2005, pp. 253–258.
[3] J. Bauer and F. Freiling, "Towards Cycle-Accurate Emulation of Cortex-M Code to Detect Timing Side Channels." IEEE, Aug. 2016, pp. 49–58. [Online]. Available: http://ieeexplore.ieee.org/document/7784555/
[4] B. Sprunt, "The basics of performance-monitoring hardware," *IEEE Micro*, vol. 22, no. 4, pp. 64–71, Jul. 2002.
[5] M. E. Ahmad, M. Najem, P. Benoit, G. Sassatelli, and L. Torres, "Adaptive Power monitoring for self-aware embedded systems." IEEE, Oct. 2015, pp. 1–4.
[6] N. Ho, P. Kaufmann, and M. Platzner, "A hardware/software infrastructure for performance monitoring on LEON3 multicore platforms." IEEE, Sep. 2014, pp. 1–4.
[7] ARM Limited, "AMBA 3 AHB-Lite Protocol v1.0," Jun. 2006.
[8] STMircoelectronics, "STM32f072x8 STM32f072xb Specifications," Jan. 2017.
[9] S. Banik, S. K. Pandey, T. Peyrin, Y. Sasaki, S. M. Sim, and Y. Todo, "GIFT: A Small Present," in *Cryptographic Hardware and Embedded Systems CHES 2017*, W. Fischer and N. Homma, Eds. Cham: Springer International Publishing, 2017, vol. 10529, pp. 321–345.
[10] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An Ultra-Lightweight Block Cipher," in *Cryptographic Hardware and Embedded Systems - CHES 2007*, P. Paillier and I. Verbauwhede, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4727, pp. 450–466.
[11] ARM Limited, "CoreMark Benchmarking for ARM Cortex Processors," Jul. 2013.
[12] B. Mohammadi, O. Andersson, J. Nguyen, L. Ciampolini, A. Cathelin, and J. N. Rodrigues, "A 128 kb 7t SRAM Using a Single-Cycle Boosting Mechanism in 28-nm FDSOI," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 4, pp. 1257–1268, Apr. 2018.