Léo Poitrimolt

Master Nanotechnologies for Integrated Circuits

Year 2019/20

Company - IC'Alps

33 Boulevard des Alpes, 38240 Meylan

High Level Platform for Application Specific Integrated Circuit architectures modeling

---

Date : February 2nd to July 31st 2020

Supervisor : Baptiste Roux, IC Designer

baptiste.roux@icalps.com

Phelma Tutor : Laurent Fesquet, Associate Professor

laurent.fesquet@univ-grenoble-alpes.fr

Abstract

Per definition, embedded systems have limited power resources, and their autonomy represents a major challenge. This is even truer with implanted medical devices. In fact, those devices induce extra cost as well as invasive surgery for power battery refill or replacement. Therefore, the demand for Low Power electronics, in this field of application, increases continuously. In order to keep this effort as reasonable as possible, high-level modeling can be a very efficient approach. Indeed, it enables fast exploration of system architecture and implementation in the first stages of the design. In this report, we will explore the world of high-level modeling targeting Low Power designs investigation.

Résumé

Les systèmes embarqués ont, par définition, des ressources en énergie limitées et leur autonomie représente un enjeu majeur. Ceci est d'autant plus vrai pour les dispositifs médicaux implantables. Un rechargement ou un remplacement de la batterie engendre pour ces dispositifs des surcoûts importants ainsi que des opérations chirurgicales invasives. Ainsi, la demande concernant l'électronique basse consommation ne cesse d'augmenter. Pour maintenir la complexité de conception aussi raisonable que possible, la modélisation haut-niveau peut se montrer d'une aide précieuse. Elle permet notamment une exploration rapide d'architectures système et d'implémentations au cours des premières étapes de conception. Dans ce rapport, nous étudierons le domaine de la modélisation haut-niveau pour l'analyse de systèmes basse consommation.

Riassunto

I sistemi integrati, per loro stessa natura, hanno risorse limitate in potenza e l'incremento della loro autonomia è oggetto di considerevole sfida. Questo è maggiormente vero per dispositivi medicali impiantati. Infatti questi dispositivi comportano costi supplementari in aggiunta ad interventi chirurgici invasivi per la ricarica o la sostituzione della batteria principale. Di conseguenza, la domanda di elettronica a basso consumo in questo specifico settore è in costante aumento. Al fine di contenere questi sforzi entro un limite ragionevole, dei modelli ad alto livello possono costituire un approccio efficiente verso questo specifico problema. Sicuramente essi rendono possibile un'esplorazione rapida dell'architettura del sistema già nelle fasi preliminari di progetto. In questo resoconto, presenteremo il mondo della modellistica ad alto livello con una prospettiva specialmente dedicata alla progettazione a basso consumo.

Key words : High Level Modeling, Emulation Platform, System on Chip, Low Power

Acknowledgements

I would like to deeply thank my supervisor, Baptiste Roux, for his patience and willingness to teach me as much as possible, despite the unusual circumstances of my internship.

I warmly thank Lucille Engels and Jean-Luc Triouleyre for their trust and confidence in my work, and the whole team of IC'Alps for making me feel one of them. Special greetings for the members of the digital team : thank you Gauthier, Arthur, Bruno and Didier for the enjoyable working environment.

Contents

1 Presenting IC'Alps **2**
- 1.1 Benefits of an ASIC . . . 3
- 1.2 Business Model . . . 4

2 Project Topics and Objectives **5**
- 2.1 Interest of High Level Modeling . . . 6
- 2.2 MOPIC Project . . . 7

3 Programming Tools **9**
- 3.1 Description of SystemC . . . 9
- 3.2 Description of TLM 2.0 . . . 10
- 3.3 Virtual Components Modeling Library . . . 12
- 3.4 Consumption Model . . . 13
  - 3.4.1 Time consumption . . . 13
  - 3.4.2 Energy consumption . . . 13

4 Emulation Platform **16**
- 4.1 Application . . . 16
- 4.2 Architecture and blocks . . . 17
- 4.3 Configuration File . . . 22

5 Case Study **23**
- 5.1 Sofware or Hardware FIR Implementation . . . 24
- 5.2 Power Domains Strategy . . . 26
- 5.3 Processing Strategy . . . 29
- 5.4 Limitations and improvements . . . 31

6 Cost Estimation **32**

7 Conclusion **34**

Chapter 1

Presenting IC'Alps

Established in 2018 in Grenoble's area, IC'Alps is a fabless semiconductor specialist designing full custom ASIC and SoC. The company was co-founded by Lucille Engels and Jean-Luc Triouleyre. IC'Alps is a subsidiary of a larger group, Doliam, promoting innovative technologies for medical applications. IC'Alps's team keeps getting larger since its establishment and counts today over 30 employees. The company's offer is centered around the design and supply of ASIC, migration from FPGA to ASIC, or the implementation of custom blocks. Those services can be performed on different technologies (Bulk, SOI, FDSOI, etc) with nodes going from 350nm down to 22nm.

To realise such services, IC'Alps's expertise relies in various domains such as

- Analog and mixed signal design
- Advanced multi-core architecture
- Low Power, low noise, high resolution designs
- Security

IC'Alps counts now 3 certifications. EN 9100 for aeronautics and spatial applications, ISO 13485 for medical devices and ISO 9001 for the quality of the management. Furthermore, since 2019 the company has been approved ARM Design Partner. These elements are a guarantee of quality of IC'Alps's products.

IC'Alps's ambition is to target mainly the world of healthcare devices. ASICs can find their place in monitoring, imaging, diagnostics and many more. Ideally, 50% of the projects carried on are medical oriented. The other half is dedicated to various fields of application such as aeronautics, transport, IoT, etc. An example of IC'Alps's client is the company SentinHealth which develops innovative solutions for people suffering from chronic cardiac diseases.

Figure 1.1: Certification logos

1.1 Benefits of an ASIC

Designing an ASIC or moving from a FPGA to an ASIC can be done for numerous reasons :

1. The most obvious one concerns the dimensions. An ASIC can drastically reduce the size of an integrated circuit. This can be a major asset in medical applications, especially for implanted devices. A reduced size has a positive impact on the patient's comfort.
2. The second asset involves the power consumption. In fact, an ASIC can offer a better power management control, an improved functionality integration and less external active components. All these notions result in low power consumption, allowing long life battery operation.
3. An ASIC is often chosen for the performances it can achieve. Performances can be highly optimized and new functionalities can be added.
4. Reliability of an ASIC is one of its main asset. The number of electronic components as well as interconnects being diminished, reliability can only increase. In the same way, as being smaller, mechanical constraints are also scaled down.
5. Finally, having a system fully integrated, the security and IP protection are both enhanced.

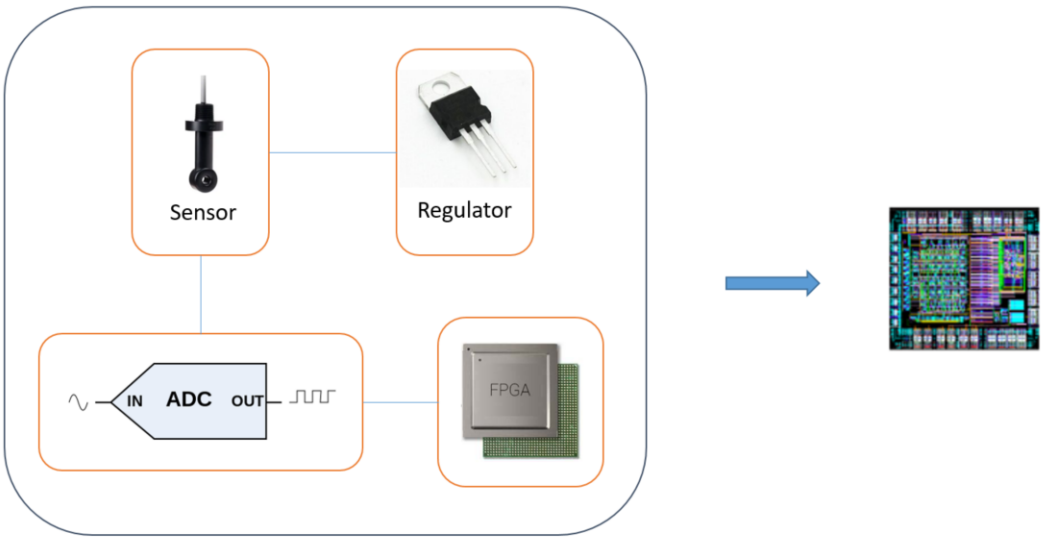


Figure 1.2: Moving from FPGA to ASIC solution

As shown in the sketch above, an ASIC can integrate digital algorithms and different blocks around it such as sensors, regulators and analog-to-digital converters.

1.2 Business Model

IC'Alps is able to provide three types of different services. The most significant one is to provide a customer with a full turnkey solution. This means IC'Alps takes in charge the whole chain from the definition of a product all the way to the industrial production. In between, this includes the design and verification, prototyping, on-chip validation and finally industrialization.

The other two services are links of this global chain : it can be back-end and design services or circuit industrialization. Regardless the size of the project, the service provided can be very flexible and adapted to a customer's needs.

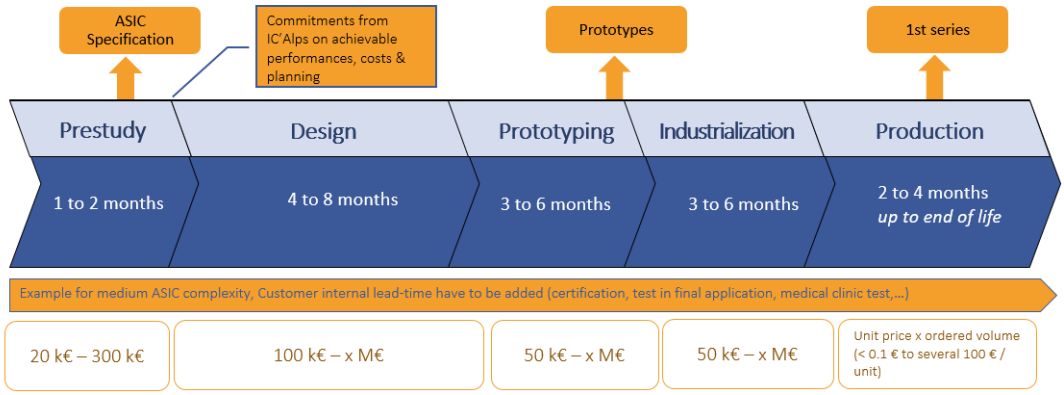


Figure 1.3: Example of planning and costs for a medium ASIC complexity

In this timeline are found all the major steps to follow when developing an ASIC, from pre-study all the way to fabrication on silicon. Ranges of time duration as well are prices are given. To illustrate the complexity of an ASIC, if we take the lower bounds of both categories we obtain : 13 months of development with a cost of 220 000 €. When the complexity is maximum, it can take up to two years.

Chapter 2

Project Topics and Objectives

IC'Alps strategy is to ease the access to custom integrated circuits by developing a ASIC platform for monitoring vital signs in medical applications. The use of this platform will help future customers to perform their proof of concept up to low-volume production and to go further by defining the integrated circuit customized per application. The objective is to build a modular ASIC platform with a Micro-Controller Unit for data processing, multiple power domains, several analog readouts, and some security functions. Section 2.2 will provide more details about this project.

The main characteristic of such a product is to provide a very high modularity and genericity. Each application has different needs and each client is different. This means for each demand, the architecture of the circuit would be changed : size and type of the memory, specific hardware IPs, etc. For each application and its specifications, plenty of solutions are available. The task of a designer is to select the best one in order to achieve the desired performance while keeping the power consumption as low as possible. But of course the decision has to be taken in a reasonable amount of time. As mentioned previously in section 1.3 , the overall pre-study takes usually between one and two months for a complete ASIC. This means a designer needs to be very efficient in his architecture exploration and obtain a satisfying solution as fast as possible.

To fasten this phase of exploration, IC'Alps wanted to have its own emulation platform. An emulation platform is *"the process of imitating the behavior of one or more pieces of hardware"* [2]. The project of my internship was thus to develop such an emulation platform. Still in this idea of quickness of exploration, this platform should be developed using high-level modeling tools. Further details about high-level modeling are provided in the following section 2.1.

The guidelines of my project have been defined by the specifications of the emulation platform. The platform has to fulfill various objectives. The main ones are :

- Exploration in advance of ASIC architectures for medical devices
- High level of modularity
- Development of generic blocks
- Hardware//Software partitioning exploration
- Development of the embedded software
- Assessment of power consumption

In the following chapters, I will detail how I have planned to construct this emulation platform, provide a use-case example, and analyse the results in order to assess the utility of this platform.

2.1 Interest of High Level Modeling

ASICs are every-time more complex and highly time consuming in terms of design. EDA (Electronic Design Automation) tools are now mandatory in the field of digital design but do not solve every problems and are still time consuming to use.

In system design, high level modeling can be a very interesting alternative to investigate different architectural or behavioral possibilities. High level modeling consists of simplistic models abstracting some features and behaviors of a system.
The main advantages of high level modeling can be listed as :

- Validate swiftly the functional behavior of a system
- Reduced time of simulation
- Ease in the description of Hardware IPs
- Early development of the embedded software application

The common tools used to perform high level modeling for complex systems are : SystemC Library and TLM 2.0 (Transaction Level Modeling). I have been using them throughout my entire project. A more precise description will be done in the following chapter 4.

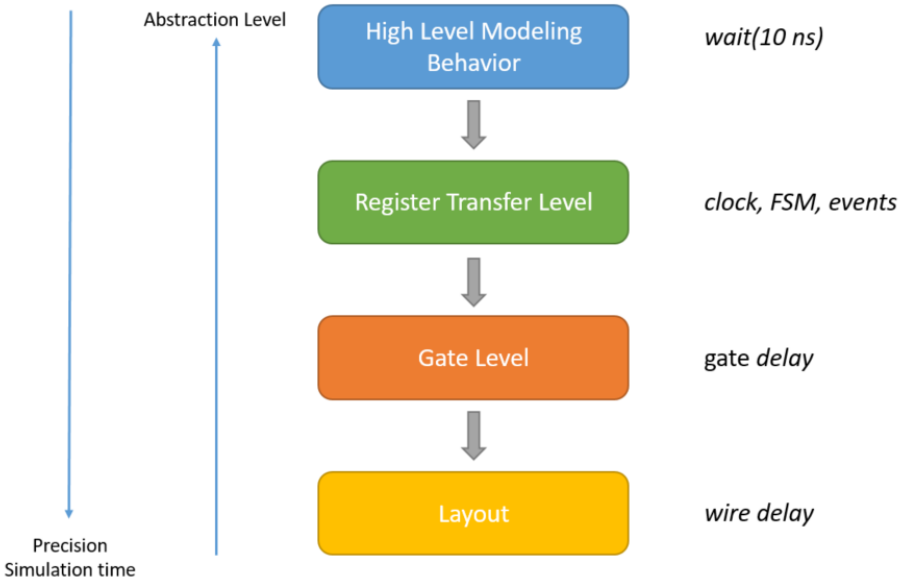


Figure 2.1: Levels of abstraction in system design

The sketch above presents the four main levels of abstraction when considering a design. Of course, the level of abstraction is conversely proportional to the precision of the model. Thus, high level modeling has a very poor precision and timing annotations are precised as *wait*.

2.2 MOPIC Project

As introduced at the beginning of the chapter, an ASIC platform targeting medical applications is being designed within IC'Alps. This platform would be a functional starting point for new customers in order to improve the definition of their product, before moving towards a more custom and specific ASIC.

This project targets applications with physiological sensors. Typical physiological measures are : electrocardiogram (ECG), electroencephalogram (EEG) or Bioelectrical Impedance Analysis (BIA). ECG measures the cardiac activity, EEG the brain one, and BIA refers the impedance of the body. This value can be used in many ways : body analysis (body fat and muscle mass) or cardiology.

Some basic bricks should be found in this platform. A micro-processor should lead analog sensors and the processing of the acquired data. A dedicated block would take in charge the strategy of power management. Security functions should also be found as well as communication ports with the outside. The sketch 2.2 below summarizes those 6 main facets of a MOPIC architecture.

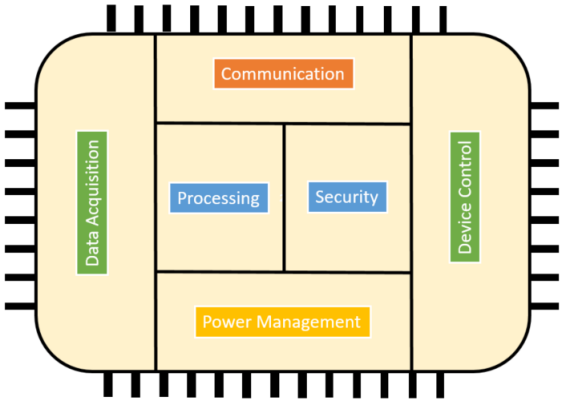


Figure 2.2: Global structure of a MOPIC architecture

Concerning the micro-processor, IC'Alps being ARM Design Partner, the choice of an ARM processor was not taken randomly. More precisely, an ARM Cortex M0+, based on the Armv6-M instruction set architecture has been selected. This micro-processor is characterized by its very low energy consumption while keeping a veritable level of performance. Its area is highly reduced too.

To reduce its impact on the power consumed, the M0+ includes two sleep modes : WFI (Wait For Interrupt) and WFE (Wait For Event).

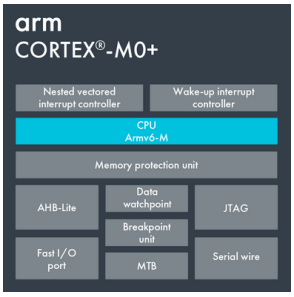


Figure 2.3: Internal structure of the micro-processor ARM Cortex M0+

For the power management, a dedicated block named PMC (Power Management Controller) drives the system into different modes. The architecture is split into Power Domains, where each domain can be set to a pre-defined state. When a given Power Domain is not active, it can be switched off to reduce its power consumption.

To notify each Power Domain in which state they should be in, the PMC gathers information from its environment. These signals can come from the Interrupt Controller of the micro-processor or other blocks such as a Timer for example. For obvious reasons, the PMC would find itself in a Power Domain defined as *Always ON*.

Processing of the data is the most application-specific part of the architecture. Plenty of possibilities are available depending on what kind of information does the client want to investigate. But before the processing stage, one or more steps of filtering are typically required to obtain a clean and exploitable signal. Thus, it can re-defined in a second version of the ASIC when the application is better defined.

Security matter is for sure a non-negligible issue when dealing with physiological measures of patients. Data have to be well protected within the system but even more if being sent to the outside world. Two main security primitives have been chosen for the MOPIC Project : The Physically Unclonable Function (PUF) and the True Random Number Generators (TRNG). Both security algorithms are innovative and being developed in collaboration with the TIMA Laboratory based in Grenoble.

Finally, communication with the outside can also vary depending on the application and the type of data to be transferred. For example the Bluetooth Low Energy (knows as BLE), defined by Nokia in 2006, is an option which could suit many applications.

The topic of my internship fits within the frame of this MOPIC project. The aim of the platform is to explore a typical MOPIC architecture, targeting medical applications. Exploration concerning the PMC can be highly facilitated thanks to high-level modeling, as power consumption remains the highest concern.

Chapter 3

Programming Tools

3.1 Description of SystemC

SystemC is a library of the programming language C++, composed of plenty of different classes. This library is open-source and can be downloaded freely on Accelera. Developped by the Open SystemC Initiative (OSCI), it was first standardized in 2005 as IEEE 1666-2005. Nowadays, Accelera Systems Initiative is in charge of its evolution.

The aim of SystemC is to imitate common Hardware Description Language such as VHDL or Verilog using a very popular programming language : C++. Hence, it enables developing systems which are composed of both hardware and software features, keeping high simulation performances. SystemC creates a bridge between these two worlds.

Accelera Systems Initiative proposes a proof of concept simulation kernel for SystemC. To create the illusion of concurrency found in Hardware Description Language, the mono-threaded kernel operates under 2 major phases : *elaboration* and *execution*. The *elaboration* phase initializes the data structures and links the different components found in the architecture description. Secondly, the *execution* phase schedules the different processes giving the impression of concurrency. All simulation processes are launched at once and their output values updated. This cycle is known as a *delta cycle*. At each *delta cycle*, the simulation time is incremented, and new processes can start if an event is notified among their sensitivity lists.

Thanks to C++ object oriented features like Templates, class inheritance or metaprogramming, SystemC contains plenty of classes such as the *sc_module class* which allows describing hardware IP containing parallel processes and events. C++ has been broaden with new object types better suited to describe hardware components. SystemC elements can be separated into two categories : structural elements such as modules and ports and communication elements such as events and channels.

Below is an example of a D Flip-Flop described using SystemC library. We recognize the class *sc_module* and the process *sample* defined thanks to the *SC_METHOD class*. As in other hardware oriented language, we can declare a list of sensitivity. Here the process is sensible to the positive edge of a clock signal [4].

```
#include "systemc.h"

SC_MODULE(dff){
    sc_in<bool>     data_in;
    sc_in<bool>     clock;
    sc_out<bool>    data_out;

    void sample() {
        data_out = data_in;
    }

    SC_CTOR(dff) {
        SC_METHOD(sample);
        sensitive << clock.pos();
    }
};
```

Listing 3.1: Exemple of D Flip-Flop described in SystemC

3.2 Description of TLM 2.0

TLM 2.0 (Transaction Level Modeling) is a standard also developed by Accelera Systems Initiative, which enables blocks to communicate via read or write commands. TLM takes in charge communication between SystemC processes, thanks to function calls. TLM has been developed specifically for on-chip memory-mapped-busses. SystemC library is extended with additional primitives such as Initiator (master) or Target (slave) sockets, as shown in the figure 3.1 below. TLM enhances simulation performances thanks to those primitives.

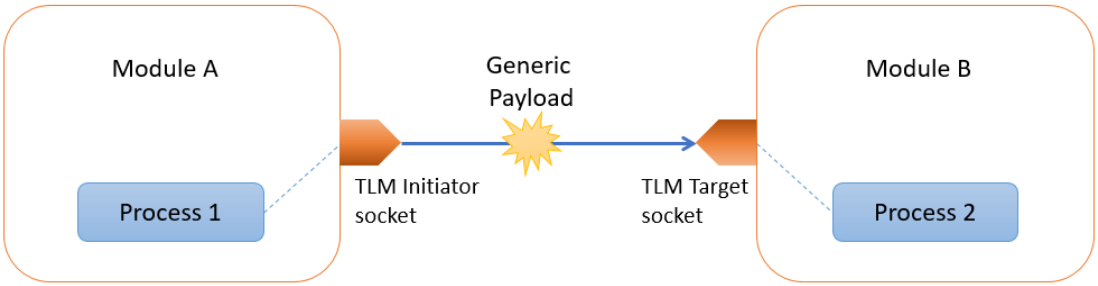


Figure 3.1: Schematic of TLM sockets and payload

TLM transactions are considered as *Generic Payloads* having a dozen of intrinsic attributes which can be set or read. The main ones are :

- Command - Write or Read instruction
- Address - Physical address in memory
- Data - Through a pointer
- Size - Number of bytes of the transaction
- Width - In case of burst transaction
- Response - Status of the response, feedback towards master

Different types of transports are possible. The most common one is the blocking transport interface (known as b_transport). Transaction's arguments are passed as reference. A timing annotation is also added to the transaction. This time is added to the simulation time. Below is an example of how the b_transport function can be implemented :

```
#include "systemc.h"
#include "tlm.h"

void b_transport( tlm::tlm_generic_payload& trans, sc_time& delay )
{
  tlm::tlm_command cmd = trans.get_command();
  sc_dt::uint64    adr = trans.get_address();
  unsigned char*   ptr = trans.get_data_ptr();
  unsigned int     len = trans.get_data_length();
  unsigned int     wid = trans.get_streaming_width();

  if (adr >= uint64(SIZE) ||  wid < len)
    SC_REPORT_ERROR("TLM-2",
                "Target does not support given generic payload transaction");

  if ( cmd == tlm::TLM_READ_COMMAND )
    memcpy(ptr, &mem[adr], len);
  else if ( cmd == tlm::TLM_WRITE_COMMAND )
    memcpy(&mem[adr], ptr, len);

  trans.set_response_status( tlm::TLM_OK_RESPONSE );
};
```

Listing 3.2: Exemple of b_transport function in SystemC [6]

Depending on the command type, *TLM_READ_COMMAND* or *TLM_WRITE_COMMAND*, arguments are not passed to the function *memcpy* in the same order. If the payload as been treated correctly, a feedback is given to the master through the payload's attribute *response_status*, setting it to *TLM_OK_RESPONSE*.

TLM 2.0 standard supports two levels or timing details :

1. Loosely Timed : A transaction, responding to a read or write command, is considered as one unit of time;
2. Approximately Timed : A transaction is broken down into different phases, being closer to real hardware protocols such as the AHB protocol used by ARM processors for example.

Commonly, Loosely timed models are associated with blocking transport interface and Approximately timed with non-blocking transport interface. In my model, I have chosen the first combination since for the platform, speed is preferred to precision.

3.3 Virtual Components Modeling Library

As a starting point for the development of the platform, with the advice of my supervisor, I have decided to reuse the work of Jan Henrik Weinstock, senior Simulation Architect at Synopsys Inc [3]. His library is open-source under Apache License and can be freely downloaded on GitHub. It contains a set of SystemC and TLM modeling primitives and component models. VCML facilitates the construction of a system level simulator, specifically for embedded systems.

It has been a great opportunity for me to get more familiar with high level modeling and the associated programming languages. Understanding how this library has been developed was to me an efficient learning process. I could expand the existing models by created new components and assemble them into a simulation platform. Different classes are defined in order to create new blocks. The *mother-class* is the *Component* one; then the class *Peripheral* inherits from it.

```
#include <vcml.h>

class my_component: public vcml::component {
public:
    vcml::slave_socket IN;
    vcml::master_socket OUT;

    my_component(sc_core::sc_module_name& nm):
        vcml::component(nm), IN("IN"), OUT("OUT") {
    }
};
```

Listing 3.3: Template provided by VCML to create a new *Component*

The abstraction level in VCML is even greater than the one in SystemC/TLM. Communications between components is smoother and programming becomes more easy-going.

Numerous primitives are defined to facilitate the work of users : TLM ports, interruption ports, I/O peripherals, components, registers, etc. Generic components are also ready to used as is. In the list we can find : UARTs, SPI, Bus, and more.

For objects of the class *Peripheral*, registers are implemented and ready to be used. Users can define its name, its address offset and its initial value. Reading and writing commands can be allowed or not, and functions associated to these commands can be overwritten.

```
class my_peripheral: public vcml::peripheral {
public:
    vcml::slave_socket IN;
    vcml::register<my_peripheral, uint32_t> REG;

    my_peripheral(const sc_core::sc_module_name& nm):
        vcml::peripheral(nm),
        IN("IN")
        REG("REG", 0x100, 0){   // name, address, value
        REG.allow_read_write(); // grant access
    }
};
```

Listing 3.4: Example of register implementation

An extra key feature of this Library is the possibility to define parameters which can be set post-compilation. Parameters can be set through different means. The most convenient one is a configuration file that is given as an option during execution of the compiled code.

VCML offers a highly configurable logging system that records all the events which occur during a simulation. The level of verbosity of this global record can differ from a component to another. This record can be saved in a file or prompted in a terminal, also with different level of verbosity.

Finally, VCML allows the plugging of different peripheral backends, like TTY session for example, for real-time interaction with the platform. This can be truly helpful to observe output values of the platform for example.

For all those reasons, using this Virtual Component Modeling Library has been a great learning process and made me gain a significant time.

3.4 Consumption Model

The main goal of the platform is to enable the exploration of architectures and the splitting and strategy of Power Domains. The measure which allows comparing between different possibilities is the power consumption. As explained in previous chapters, the main concern when considering medical devices is the consumption rather than the area. Of course the area has to stay reasonable in order to respect comfort standards.

Therefore, an important task of my internship was to establish a rather simplistic but realistic consumption model, extending the Virtual Components Modeling Library. The aim of this model is to provide clear assessments for each tested scenario. Two types of consumption needed to be modeled : time consumption in seconds and energy consumption in Joules.

3.4.1 Time consumption

High level modeling is very efficient in many ways but one should always keep in mind that in real circuits every event takes time to be completed : a transfer from a place to another, an operation, etc.

Thus, to compare more accurately different scenarios, 2 timing components have been added :

1. Delay of transactions : each time a TLM transaction routes through the interconnect a timing delay is added thanks to a *wait()* statement. The duration of this statement is customizable through a configuration file.

2. Computation time : each time an operation is done, the global simulation time is incremented. Up to now there is now distinction between all the different operations that a CPU can done. A point of improvement would be to implement a finer adjustment between operations. For example a *move* operation is different from a *logic* operation in terms of timing consumption. As previously, the timing increment is configurable.

3.4.2 Energy consumption

Three energy contributions have been listed and participate to the global energy consumption :

1. Static Contribution : As long as a device is switched on, it contributes to the consumption due to leakages. Transistors are not perfect and leakage current is unavoidable. The static power

can be defined as :

$$\begin{aligned} P_{stat} &= \int V_{DD} \times I_{leak} \quad dt \\ with \quad I_{leak} &= I_0 \times e^{\frac{-V_{th}+\lambda_{DS}V_{DD}}{v_t \times n}} \\ and \quad V_{DD} &\text{ being the nominal voltage} \end{aligned} \tag{3.1}$$

And the energy is defined as the time integral of the power.

Without going into the details of I_{leak}, the aim of my model is to remain rather generic and simplistic. As the initial formula is a integral of time, the approach chosen is the following :

$$E_{stat\_model} = \int_0^t C_{block} \quad dt \tag{3.2}$$

with C_{block} a constant specific to a block and t the time that the considered block is switched on.

2. Dynamic Contribution : each time an operation is done, hundreds of transistors are switched on and off performing logic and arithmetic operations. Those switching operations contribute to the dynamic power. The dynamic contribution of a transistor can be computed as :

$$P_{dyn} = \frac{\alpha}{2} \times C_L \times V_{DD}^2 \times f_{clk} \tag{3.3}$$

with

α : Activity factor of a transistor, usually around 10%

C_L : Load capacitance seen by the transistor

V_{DD} : Nominal voltage of the circuit

f_{clk} : Frequency of the clock

However, a such precise evaluation is not relevant within a platform designed for high level modeling. As for the static component, a simpler approach has been adopted. The dynamic component will follow a linear function :

$$E_{dyn\_model} = C_{block}^A \times x + C_{block}^B \tag{3.4}$$

with C_{block}^A and C_{block}^B being constants specific to each block and x being the number of iteration of the hardware core. This function is called each time an operation is being performed by the CPU or any hardware block described in SystemC.

3. Wake up Contribution : Lastly, as explained formerly, Power Domains can wake up at given times. This change of mode is also energy consuming. Each time a switching on operation occurs, the *Inrush current*, corresponding to the maximal instantaneous input current of a device when first turned on, engenders an important demand of energy. This spike in current can be estimated with the equivalent capacitance of a device. This equivalent capacitance can be approximated thanks to the *Gate Equivalent Model*. Unfortunately, this *Gate Equivalent* information comes out from synthesis results. As I could not have access to this information, another more simplistic approach has been chosen.

The wake up contribution has been approximated as being equal to the static consumption during 200ms of the given Power Domain.

$$E_{wake\_up} = \sum_{block\ in\ PD} \int_0^{200ms} C_{block}\ dt \tag{3.5}$$

In practice, an additional block named *Aggregator* takes place within the architecture. As its name suggests, its function is to gather every information concerning energy consumption during the simulation. The *Aggregator* is not hooked up to the interconnect, but all blocks participating to the global consumption own a TLM master socket directly connected to it. Thus, the *addr* component of a TLM payload isn't necessary to route a payload to right address anymore and can be used in a different way. I took advantage of it by placing the nature of the power consumption (static, dynamic, wake up) in this field of the transaction.

At regular intervals, the *Aggregator* writes into a CSV file containing all information about energy consumption over the simulation. In a second time, a external script written with Python programming language parses the CSV file and generates customizable assessments of a given simulation.

Chapter 4

Emulation Platform

The construction of the architecture of the platform has evolved throughout my internship. Towards the end, we imagined an application in order to have an application-oriented architecture. In the following sections, I'll present the application chosen and the corresponding architecture.

4.1 Application

The aim of the platform is to reproduce a typical architecture design of a medical device. Thus, together with my supervisor, we have imagined a story-line for a specific application. The broad idea is following :

- A sensor measures the cardiac activity via an electrical signal
- The signal is sampled in order to be treated numerically
- The signal is filtered and converted to the frequency domain
- The signal is encrypted for security reasons
- Finally, the signal is sent out through a serial peripheral (ex: SPI)

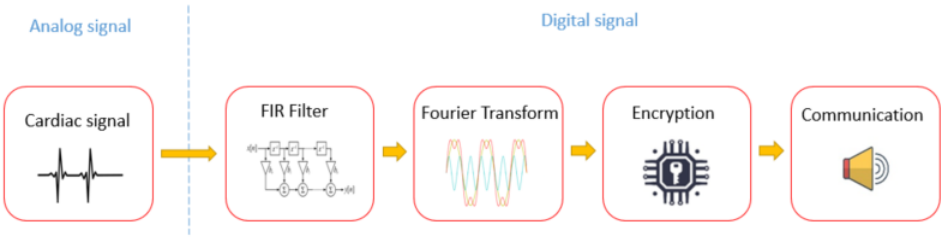


Figure 4.1: Sketch of our imagined scenario of application

With this application in mind, I knew which block should constitute my final architecture. Each block will be detailed in a following section 4.2.

As in most of medical devices, the system should not operate full time. The system alternates between phases of acquisition, rest and processing. To respect this feature of most medical devices, I have defined a strategy illustrating these different states of the system. So, my application follows the given pattern :

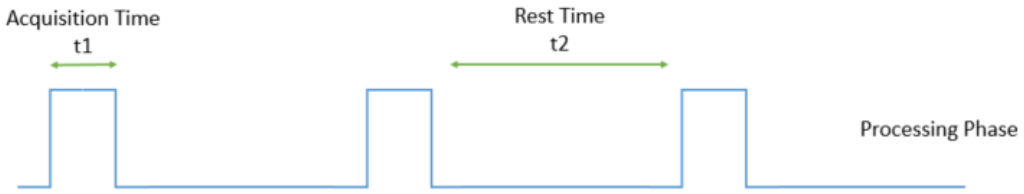


Figure 4.2: Timeline of the application

As mentioned previously, the main concern in medical apparatus is the energy consumption. By looking at the above sketch 4.2, a first idea comes to mind : when the system is in a rest phase, some blocks should be switched off in order to reduce their consumption. This is the role of the PMC mentioned in the section 2.2.

4.2 Architecture and blocks

The architecture is composed of different blocks : some are directly taken from VCML, some adapted from other resources and some are written by myself. Here is the presentation of the components found in the final architecture :

- Central Processing Unit - In order to integrate the micro-processor ARM Cortex M0+ of the MOPIC Project into the platform, adaptations were required. Two options were available :

  1. A quasi direct translation of the source code of the processor in Verilog or SystemVerilog to SystemC through the tool Verilator. This tool takes as input a synthesizable code in Verilog or SystemVerilog, performs code-quality checks and compiles it into SystemC. The precision of the obtained SystemC model is very accurate, but simulation are quite slow. Furthermore, the model obtained is seen as a *black box* and results were very inconvenient for debugging.
  2. Use of a Unicorn model, subset of the QEMU Instruction Set Simulator, translating the instruction set of the Arm Cortex M0+ into the instruction set of the platform, x86. More features are added to the model, like the correct number of registers found in the processor for example. Then, a wrapper integrates the model generated into the VCM Library, extending the *processor* component. This option is more demanding in terms of integration but results in a more manageable model. The observability is highly increased compared to the first one and debugging is thus facilitated.

  The second option with Unicorn was chosen for its speed but mainly for its observability and easiness in debugging.

- Memory - A memory unit is of course mandatory within an architecture. A SystemC model was existent is the VCML Library, miming a RAM memory of tunable size. In our case, its size was set to 128 MBytes. This parameter can be easily changed through a configuration file.

- Interconnect - VCML library also had a SystemC model of an interconnect. This interconnect is a memory-mapped bus, which shunt TLM payloads towards the right block thanks to the *addr* component of a payload (see 3.2).
- Timer - As specified previously, most medical applications have 2 main modes : a sleeping mode and an operating mode. Thus a timer is required to switch between modes. Time values can be set through registers. This has been designed by myself.
- Power Management Control (PMC) - As addressed in section 2.2, PMC's role is to set each Power Domain in the correct state. States of the architecture are defined through a Look Up Table as the following one :

	Power Domain 1	Power Domain 2	Power Domain 3
Mode ON	ON	ON	ON
Mode STANDBY	ON	ON	OFF
Mode REST	ON	OFF	OFF

Table 4.1: Example of a Look Up Table implemented in the PMC

The PMC also contains registers : the CPU may also change the mode of a Power Domain through these registers. The address map of the register is defined in this way :

Name of register	Address offset	Read and Write enable
PD_1	$0x00$	R/W
PD_2	$0x08$	R/W
PD_3	$0x10$	R/W
PD_4	$0x18$	R/W

Table 4.2: Address map of the registers of the PMC

The distribution of blocks in Power Domains as well as the total number of Power Domains can be set as wished through a configuration file.

This block has been designed by myself and is entirely customizable.

- Analog Front End (AFE) - This block takes as input a CSV file and a sampling frequency. Through the CSV file, the AFE can parse values of any kind of analog sensors. The sampling frequency should match with the frequency at which data has been stored into the CSV file. This approach provides a high level of genericity. The name of the CSV file as well as the sampling frequency are given as parameters of the AFE. For any new application to be simulated, we only need to store the output values of a given sensor in a CSV file and change this parameter of the AFE.
- Finite Impulse Response (FIR) filter - the first digital block of the processing unit is a FIR filter. I have chosen the discrete-time version of the filter. The depth of the filter can be set by user. In the general formula, the depth is set to N. The output of the filter is defined as follows [8] :

$$y[n] = \sum_{i=0}^{N} b_i x[n-i] \tag{4.1}$$

with :

- $x[n]$ the input signal
- $y[n]$ the output signal
- N the depth of the filter
- b_i the coefficients of the filter

The main advantages of this filter is that it requires no feedback and it is intrinsically stable. This absence of feedback guarantees a finite response. The parameters of FIR filter, depth and coefficients, ensure the tuning of frequency passband and stopband. In my case, the filter is used to cut off extreme values resulting from a potential dysfunction of the sensor. The deeper the depth of the filter, the finer is the filter, but more calculations are required.

As mentioned in chapter 2, the platform should enable the prediction of partition between hardware and software implementation. I've selected the FIR filter to illustrate this functionality. This means the FIR filter has both hardware (dedicated block written in SystemC integrated within the architecture) and software (function defined in source code of the application, written in C language) implementations.

- Fast Fourier Transform (FFT) - FFT is an algorithm which computes the Discrete Fourier Transform (DFT). Fourier transform enable transforming a signal from time or space domain to frequency domain and vice-versa. In my case I have implemented the direct Fourier transform, from time domain to frequency domain. I have adapted an open-source model published on Rosetta Code website. It exploits the recursive approach of the algorithm developed by Cooley and Tukey in the 1960s [1]. My task was to make this model consistent with the primitives of the platform.

  Fourier transform is usually applied for signal filtering. Some mathematical operations are simpler and thus less consuming when being performed in the frequency domain. For example a convolution in the time domain is reduced to a multiplication in the frequency domain. Applying a FFT to the signal coming out from the FIR, we obtain the figure 4.3 below. Two main peaks are observable : the first one around 1Hz contains information on the input signal (cardiac frequency is found around 1Hz) and the second one is right at 1kHz which corresponds to the sampling frequency.

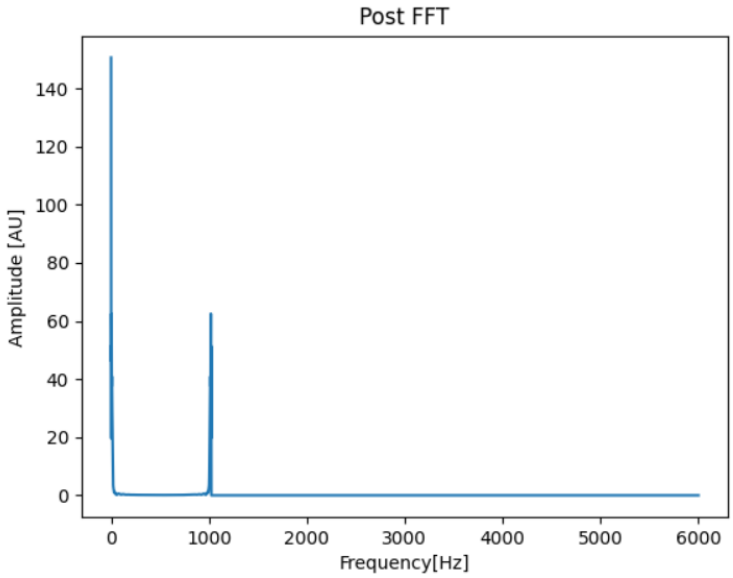


Figure 4.3: Signal in the frequency domain obtained post FFT

- Advanced Encryption Standard (AES) - AES, also known under its initial name Rijndael, is an encryption standard founded in 2001 by the American National Institute for Standards and Technology (NIST). AES is defined as a symmetric-key algorithm. This implies that a unique key is employed to encrypt and decrypt the data. This key can be specified on 3 different lengths : 128, 192 or 256 bits. A chain of 128 bits (16 bytes) can be encrypted at once. The encryption methodology is based on substitution and permutation.

  The model used is adapted from an open-source resource published on GitHub, named MachSuite. MachSuite is a collection of benchmarks for research centered on hardware accelerators. My task was to convert this model from C language to SystemC and making it compatible with the existing primitives.

- Direct Memory Access (DMA) - In the idea of keeping the consumption as low as possible, a Direct Memory Access free the CPU of a highly consuming task : copying data from one place to another. This means the CPU can stay in rest mode during longer times. In my case, one channel is defined within the DMA, copying data from the output FIFO (First In First Out) of the FIR filter to the memory.

- Copy CSV - This last block was created for convenience matters. Output values of each processing block are stored in CSV files thanks to this block. It doesn't serve any function within the architecture but guarantees an easy access to intermediate values of the processing chain.

In this way, combining all these blocks all together we obtain the following architecture 4.4 and thus can simulate the imagined story-line from an analog sensor all the way to processed and encrypted digital values.

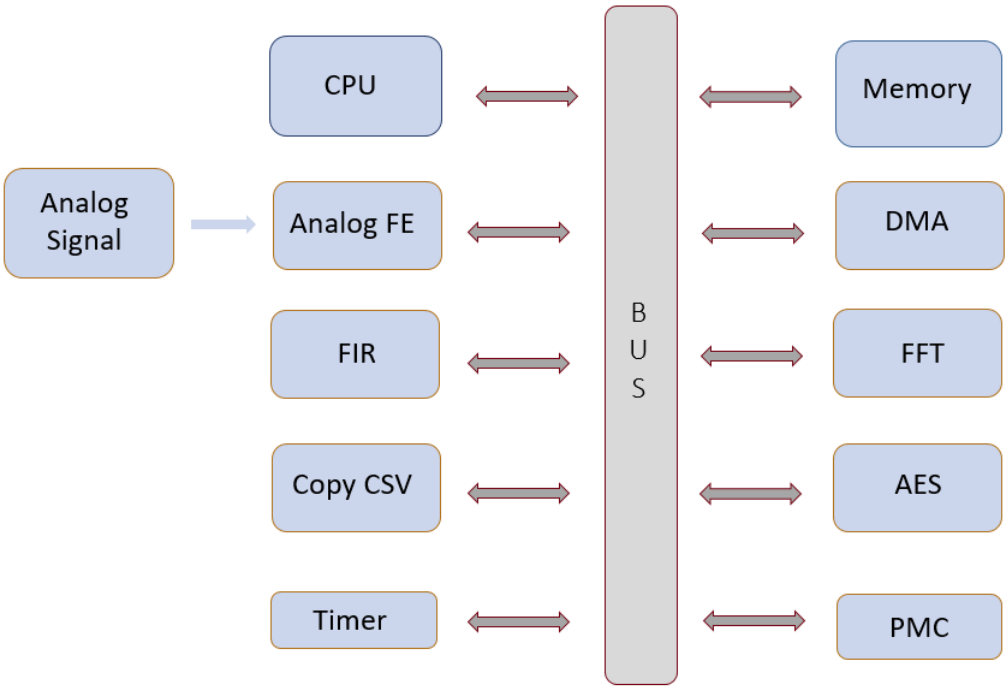


Figure 4.4: Global architecture of the platform

Up to now no block is responsible for the communication of the output data to the outside. The final stage is the output of the AES block. VCM Library contains SPI or UART models which could be integrated into the platform but it has not been done up to now for timing reasons. I judged that the added value was not so relevant for the time allotted.

4.3 Configuration File

As mentioned various times when describing the architecture, a configuration file adjoins the emulation platform. This goes along the line of genericity and modularity of the platform. Plenty of parameters can be set through this file at run time. This means no re-compilation is required between 2 simulations with different parameters. It allows a higher degree of flexibility and makes the user gain a significant time.

Parameters which can be set through this configuration file are for example the number of Power Domains, the assignment of each block to a given Power Domain, the size of the memory, etc.

```
[Source]
soc.mem.size = 0x08000000 # 128MB

[PD]
soc.pdx = 4

soc.cpu.pd_id       = 1
soc.bus.pd_id       = 0
...
soc.fft.pd_id       = 2
soc.aes.pd_id       = 2

[FIR]
soc.fir.depth_prop = 5
soc.fir.coeff_prop = 3,5,7,4,2

...
```

Listing 4.1: Fragment of configuration file

Chapter 5

Case Study

In this chapter, I will present the results obtained on the case study presented in the previous chapter and show the different assets of this tool. Thanks to the assessments generated, choices will taken in order to tend towards a more optimized solution.

The goal of my project is to set the simulation environment with generic parameters. Thus, constants C_{block}, C^{A}_{block} and C^{B}_{block} from formulas 3.2 and 3.4 have not been determined from synthesis results of an RTL (Register Transfer Level) description. Constants have chosen coherently between blocks but shouldn't not be considered intrinsically. Constants can be easily modified thanks to the configuration file 4.3.

Coherent choice of the constants between the two types of FIR implementation :
Depth of the filter = N
Number of operations = N multiplications + N additions
Cost for CPU = 2 x N instructions
Dynamic energy cost for CPU = 2 x N
Thus I set the dynamic energy cost of the hardware IP also equal to 2 x N.

The major added value of the platform relies in this energy consumption model, as it is the critical point of interest. For simplicity reason, all energy values will be considered as unit-less and will be expressed as [AU] for Arbitrary Unit. In the same idea, all the numerical results which will be presented in this chapter will have to be considered very carefully. The aim of proposing such values is more to demonstrate what the platform along with the consumption model can offer to a user having a true application with realistic energy estimations.

To imitate the analog signal coming from a cardiac sensor, an electrical potential measured between 2 electrodes, I've decided to integrate a model defined and published by PhysioNet [5], a *Research Resource for Complex Physiologic Signals*.

The model ECGSYN combine cardiovascular system variables and electrocardiograms (ECG). I was mainly interested by the realistic ECG waveform generator. I could use this model freely since it is open-source. Below is a graph of the obtained signal. To integrate this model into the platform, output values of potential (mV) are stored in a CSV file with a pre-defined sampling frequency set in the source code of the model.

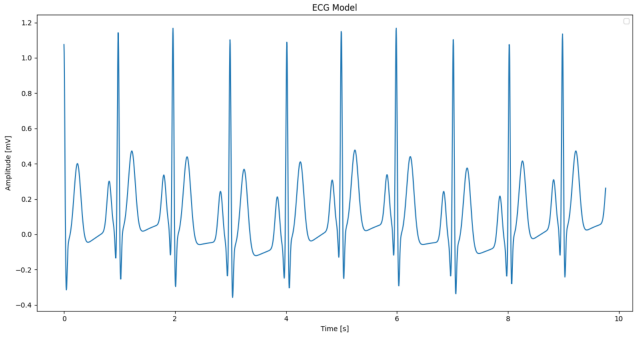


Figure 5.1: ECG signal from ECGSYN model provided by PhysioNet [5]

5.1 Sofware or Hardware FIR Implementation

The first spot of exploration is the type of implementation between hardware and software. To analyze this question, I have selected the FIR filter. Two implementations of the same filtering structure have been designed in order to compare them.

The parameters of the filter will be the followings :

Depth = 5, coefficients : $b_0 = 3$, $b_1 = 5$, $b_2 = 7$, $b_3 = 4$, $b_4 = 2$.

The output value is subjected to a normalization to maintain a unitary gain.

For the software implementation, a dedicated function is located within the source code of the simulated application; for the hardware one these parameters are set by the CPU thanks to registers. Splitting between acquisition and rest times is set as presented in the sketch below :

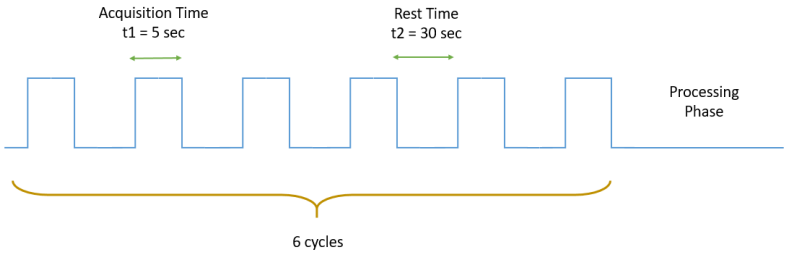


Figure 5.2: General parameters of the simulation

For this test, a very simple strategy regarding the Power Domains is applied since it is not the point of interest. The distribution is done as follows :

	Power Domain 1	Power Domain 2
Components	CPU, Memory, PMC, Timer	AnalogFE, DMA, FIR, FFT, AES
Activity	Always ON	Switched OFF during rest time

Table 5.1: Power Domains distribution for the Hardware/Software exploration

	Hardware Implementation	Software Implementation
Total Power [AU]	22 536 281	73 214 268
Simulated time [sec]	215	268

Table 5.2: General results for both Hardware/Software implementations

At first glance, the hardware implementation looks far more interesting both in terms of energy consumption and simulated time. As explained in introduction of the chapter, energy values should not be taken factually.

Let's look at another energy assessment generated thanks to the Python script.

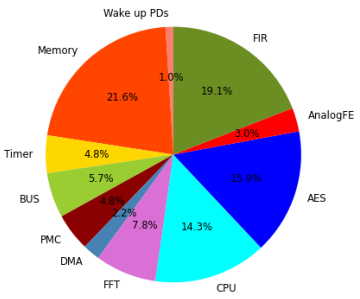


Figure 5.3: Hardware

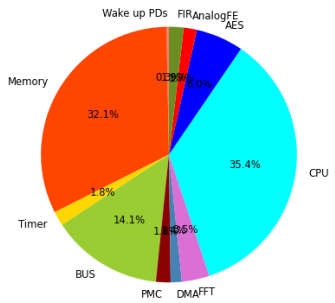


Figure 5.4: Software

Figure 5.5: Distribution of the total power

Those two pies show the distribution between all the components contributing to the total energy. For the software implementation, we can observe that 3 components monopolize the total energy : Memory (32.9%), CPU (33.7%) and the Bus (14.4%). Calculations and access to the memory through the bus from the CPU to do the FIR calculations are responsible for most of the consumption. For hardware implementation, the distribution is much more balanced between blocks.

5.2 Power Domains Strategy

The second main asset of the platform to be verified is the exploration of different Power Domains strategies. As seen in the previous section 5.1, the hardware implementation seems to be the most effective one. Thus, in this section I will be keeping this implementation for the FIR filter. The same timing parameters and number of cycles will be chosen.

Assessments will be analyzed in order to explore the most efficient power domain strategy. The starting point will be exactly the same one as in the previous analysis : 2 Power Domains with components distributed as in table 5.3

The main benefit of Power Domains is to scale down the power consumption, and more precisely the static component. Ideally, a hardware IP should be switched on only when computing or treating any data. If the following figures, we can observe that the static consumption is responsible for more than 2/3 of the overall consumption of Power Domain 1 and more than 1/5 for Power Domain 2. For the Power Domain 1, considered as the *Always ON* Power Domain, not much can be done. However for the Power Domain 2, improvement can be done since the blocks inside are not always used.

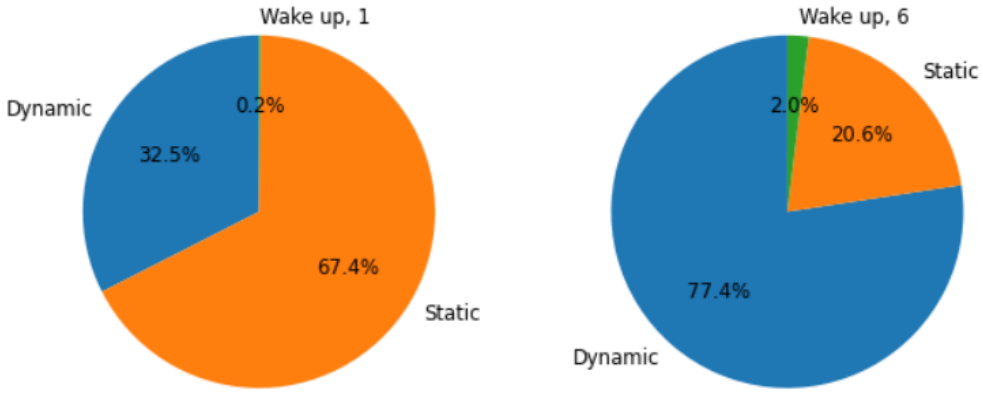


Figure 5.6: Distribution of energy components within Power Domain 1 (left) and Power Domain 2 (right)

An interesting way to start is to observe the splitting between static and dynamic components for each block with respect to time. It will bring precious information on how to regroup components in different Power Domains.

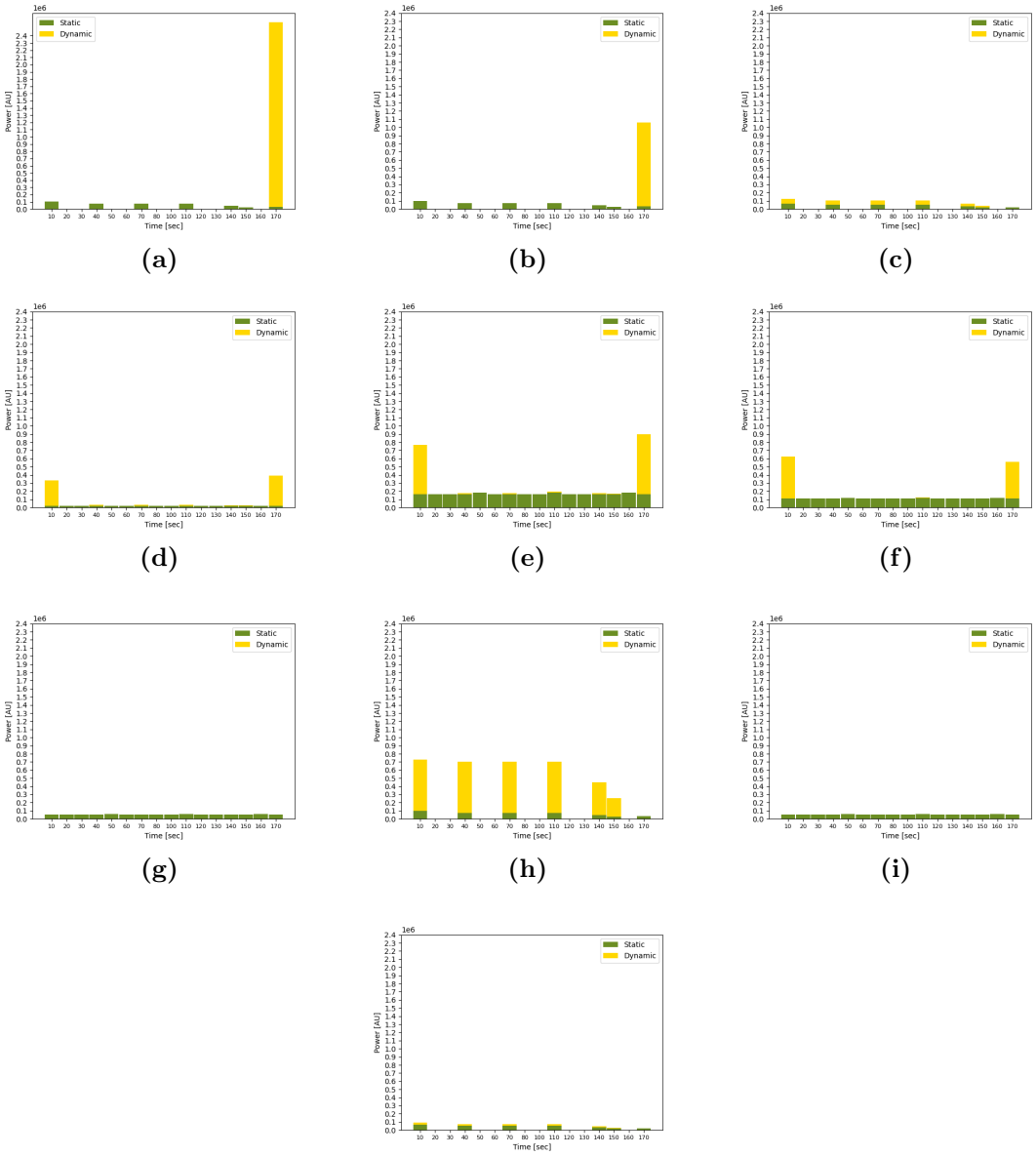


Figure 5.7: Splitting of static (green) and dynamic (yellow) components with respect to time for **(a)** AES, **(b)** FFT, **(c)** AnalogFE, **(d)** Bus, **(e)** Memory, **(f)** CPU, **(g)** Timer, **(h)** FIR, **(i)** PMC and **(j)** DMA

Two main conclusions can be taken from these graphs. Firstly, AES **(a)** and FFT **(b)** should clearly be regrouped within a specific Power Domain, since both blocks are only active towards the end of the simulation. Secondly, as in most Systems on Chip, the most consuming components are the Memory and the CPU. However, the memory is still very slightly called upon all along the simulation. Concerning the CPU, it can be moved out from the *Always ON* Power Domain.

A new distribution in Power Domains can be tested, adding a third one :

	Power Domain 1	Power Domain 2	Power Domain 3
Components	Memory, PMC, Timer	CPU, AnalogFE, DMA, FIR	FFT, AES
Activity	Always ON	Switched OFF during rest time	Switched ON only towards the end

Table 5.3: More complex Power Domains distribution

Let's compare the 2 Power Domain strategies and verify if a gain in power consumption has been acquired. To name them, I will call PD Version 1 the first distribution and PD version 2 the more elaborated one.

	PD Version 1	PD Version 2
Total Energy [AU]	22 536 281	18 310 797
Static Energy [AU]	10 199 435	7 427 790
Static Proportion [%]	45.3	40.5

Table 5.4: General results for both Hardware/Software implementations

A significant reduction of static energy is achieved with the finer Power Domain distribution. It represents a very significant improvement, since in both cases we can observe that the static component stands for about 40% of the global consumption. As mentioned few times already, all the numerical results have to be taken prudently.

Let's observe the final distribution between static and dynamic components for all the blocks outside the *Always ON* Power Domain :

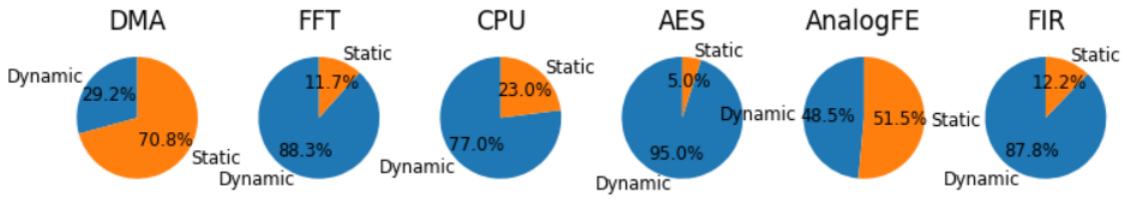


Figure 5.8: Final distribution of energy components for PD Version 2

The four most consuming components : FFT (7%), CPU (7.1%), AES (16%) and FIR (21.5%) have a clear dominant component : the dynamic one. Static component is maximal for the CPU with less than 1/4 of its global consumption. The DMA and AnalogFE are not as well optimized but are both relatively negligible compared to the global consumption. Their respective loads are 2.5 and 3.4%.

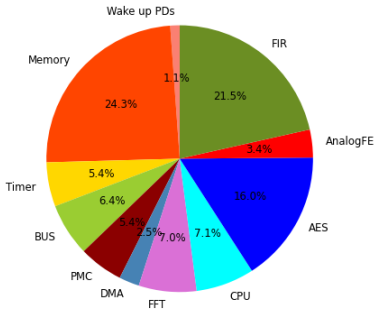


Figure 5.9: Global energy distribution between components

Power Domains can have a powerful impact on the global consumption and a wise distribution can offer significant improvements. It may seems that the more Power Domains we define, the better the results. Nonetheless, the greater the number the Power Domains, the higher the complexity for designers. Complexity can be observed in various levels such as the design, the verification of the latter and the physical implementation.

In my approach I did not consider isolation cells and cases where a request is sent towards a block being switched off are avoided, even if covered. If it happens, TLM payloads are simply not considered and requests are rejected. Three Power Domains looks to be a satisfying trade-off between consumption savings and complexity. I will thus consider this version of distribution as satisfactory.

5.3 Processing Strategy

Up to now, I have considered that the processing phase (FFT + AES) should occur at the end of the 6 cycles of data acquisition and filtering. However this choice is questionable : is it the most efficient way to process the data in terms of power consumed? This question which will be discussed in this section.

To address this point, I have imagined 3 different schemes represented in the sketches below :

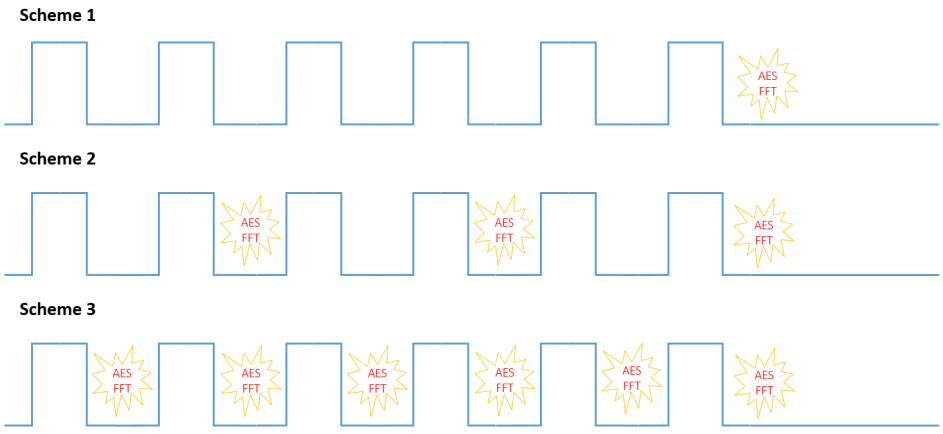


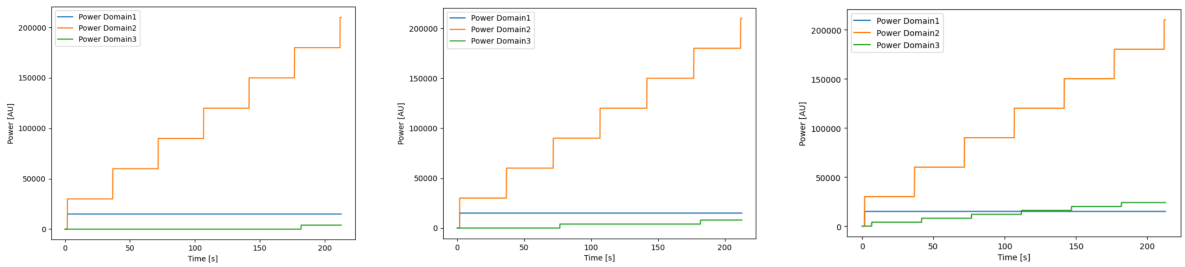
Figure 5.10: Sketch of the 3 possible schemes to compare processing strategies

Scheme 1 is the scenario which has been adopted up to now. Thus, only *Scheme 2* and *3* need to be simulated. Taking into account the results obtained in previous sections, the FIR filter will still be implemented in its hardware form and the Power Domains distribution will stay in the most elaborated form as described in Table 5.3. All the power constants are of course kept identical.

Let's compare the strategies and check if a gain in power is acquired.

	Scheme 1	Scheme 2	Scheme 3
Total Energy [AU]	18 310 797	18 255 535	18 258 452
Simulated time [sec]	212.80	212.80	212.80

Table 5.5: Comparison between 3 Schemes of processing strategy



Scheme 1 **Scheme 2** **Scheme 3**

Figure 5.11: Contribution of Wake up power for each schemes with respect to time

The second Scheme seems to be the most effective one in terms of energy consumed. The *Power Domain 3* has wake up twice more than in *Scheme 1* but it is most likely that the number of values which the FFT and the AES have to treat matches better with their respective length. The FFT deals with 1024 values each time, where as the AES can encrypt only 4 words of 32 bits at once.

In this chapter, through a single and simplistic imaginary application, many features of the platform have been exploited. Thanks to the consumption model, plenty comparisons can be achieved. In this fashion, we reach the following conclusions :

- FIR filter should be implemented with a hardware description to scale down both static and dynamic power consumption;

- Three Power Domains with the mentioned distribution seems to be an efficient and reasonable choice to limit the static consumption;

- Performing the data processing once every two cycles of acquisition looks to reduce the overall energy consumption.

5.4 Limitations and improvements

Naturally, today's state of the platform presents a certain number of limitations. All these limitations can for sure serve as points of improvement for the future. Listing all of them would very exhaustive, but here comes the features which seem to me as the most important ones :

1. Enhance the power consumption model, especially for the CPU. Up to now, all the operations performed by the CPU contribute equally to the power consumed. But we know that this assumption is not realistic and deserves a real attention for the next version of the platform. Research on this topic has already been published in literature. For example, the work of Mr. Vasilakis [7] on instruction level energy characterization of arm processors would a truly helpful resource.

2. Refine the possible states of a given Power Domain. Up to now, the mode is either ON or OFF. But in real circuits at least 2 more states are defined : *Clock OFF* and *Retention* modes. To add these features, the clock system should be brighten up with Power Domains clocks which implies re-synchronization with the system clock and retention cells to store data. This would allow more precision and flexibility in the design architecture.

3. Bufferization of the data when addressing a block being in a *OFF state*. Adding isolation cells at the interfaces of blocks to have a more realistic behavior of the relationship between Power Domains.

Additional work should be done on the platform to make it more *user-friendly* in order to deliver it to a client. While the RTL design is being implemented, a client could use the platform to improve the embedded software application.

Chapter 6

Cost Estimation

During my internship, the different sources of cost have not been numerous. Costs can be split into 3 categories : Wage, Computing Tools and Use of servers. The first 2 are direct costs and easy to provide. For the use of servers, I have supposed that they are equally used between all colleagues and the spreading in cost is done within 6 years. This means my cost during 6 months of usage can be calculated as :

$$Annual\ cost\ per\ employee = \frac{Initial\ Cost}{Number\ of\ employees} \times \frac{1}{6}$$
$$My\ cost = 0.5 \times Annual\ cost\ per\ employee$$

All the programming tools (languages + libraries) used over internship are open-source and did not engender any extra cost. Extra charges could be added such as the electricity bill, rent of offices, coffee, etc. but the main costs are presented in details in the table below. Costs are expressed in Euros.

Servers		Spreading in cost
Virtualization Servers - Initial cost	50 000	6 years
Virtualization Servers - My cost	140	-
Production Severs - Initial cost	30 000	6 years
Production Severs - My cost	85	-
My total cost for servers	225	-

Computing tools		Spreading in cost
Computer Dell Latitude 5501	1 200	3 years
Screen, Keyboard, Mouse	300	3 years
Dock	230	3 years
Office License 2016 + Zimbra License	400	1 year
Antivirus	15	1 year
Phone + License	200	1 year
My total cost for Computing Tools	596	-

Wage	
Monthly gross wage	1 200
Monthly charges	300
Total wage	9 000
Total Cost	**9 821**

Table 6.1: Listing of costs

Chapter 7

Conclusion

Throughout my internship, I had the opportunity to face plenty of new challenges. The first one was to apprehend formally the different programming languages which were new to me. I wasn't familiar with object-oriented programming and and real effort has been made to progressively obtain a sufficient comprehension and use it as a tool. After 6 months of continuous learning, I am delighted to say that I feel more at ease with these tools. In the future I'll for sure be exposed to new programming languages in the future and this experience taught me largely on how to deal with it.

Secondly, the world of ASIC and system architecture were totally unknown to me a few months ago. The fact of working on high-level modeling was highly beneficial for many reasons. I had a global overview of what do we usually find within a ASIC architecture and could manipulate these objects quite easily thanks a high level of abstraction. Thanks to the development of the platform, I am now able to understand the assets of high-level modeling.

By the very end of my internship, I had the occasion to work on RTL Design with some colleagues of the Digital team and I could measure the level of complexity encountered with this lower level of abstraction. After this experience I've realized even more the advantages that can offer high-level exploration and the benefits which come out of it.

Concerning the heart of project, the platform, most of the goals have been met. Through the Study Case (5), many features of the platform have been highlighted and answer to the requirements defined in Objectives chapter (2). Even though a lot of room is left for improvement, it validates most of the pre-defined objectives : software validation can be done, the platform is highly modular, development of new SystemC components is relatively easy and the power model allows plenty of different assessments and graphs.

I hope that in the future this work will be helpful and contribute to the success of MOPIC Project.

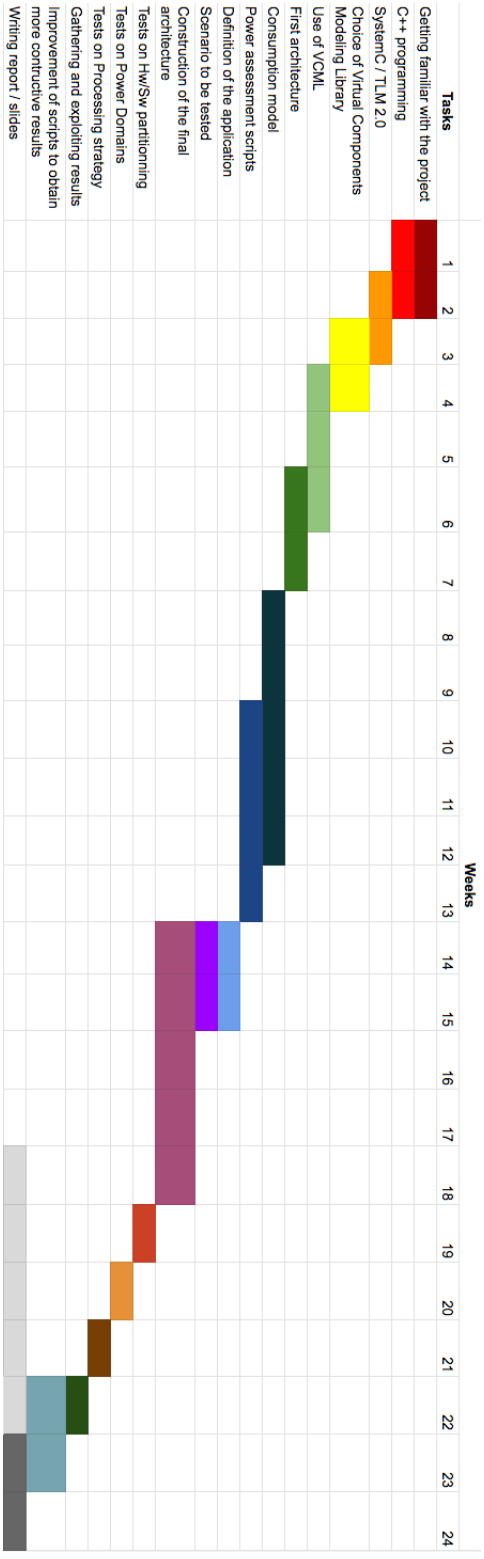


Figure 7.1: GANTT Diagram

Bibliography

[1] Cooley-Tukey FFT algorithm Wikipedia. https://en.wikipedia.org/wiki/cooley-tukey_fft_algorithm.

[2] Hardware Emulation. https://en.wikipedia.org/wiki/hardware_emulation.

[3] Virtual Components Modeling Library. https://github.com/janweinstock/vcml.

[4] Telecom ParisTech. http://hdl.telecom-paristech.fr/sc_exemples.html.

[5] PhysioNet. https://physionet.org/content/ecgsyn/1.0.0/.

[6] Doulos Tutorials. https://www.doulos.com/knowhow/systemc/tlm2/tutorial__1/.

[7] Evangelos Vasilakis. An instruction level energy characterization of arm processors. *Foundation of Research and Technology Hellas, Inst. of Computer Science, Tech. Rep. FORTH-ICS/TR-450*, 2015.

[8] FIR Filter Wikipedia. https://en.wikipedia.org/wiki/finite_impulse_response.