# NOTE TO USERS

uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Ricardo de Jesus Villalobos Guevara
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.A.Sc. (Electrical Engineering)
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Hardware/Software Co-Design Environment for BIST Applications

TITRE DE LA THÈSE / TITLE OF THESIS

Prof. V. Groza
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Prof. A. Nayak

Prof. A. Chan

Gary W. Slater
Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

# Hardware/Software Co-Design Environment for BIST Applications

by

Ricardo de Jesús Villalobos Guevara

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the M.A.Sc. degree in
Electrical and Computer Engineering

School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa

*Your file Votre référence*
*ISBN: 978-0-494-49287-1*
*Our file Notre référence*
*ISBN: 978-0-494-49287-1*

# Abstract

Current computer systems have evolved following a near exponential increase in performance of the different hardware components that they are based on. But problems are appearing in different fronts. There are issues related to the production of semiconductors and others related to the tools used to design new electronics systems.

This research presents a solution that provides a new paradigm in design of hardware system which uses proved methods found in the fields of software design. An overview of the current technologies will be presented as well as the computing paradigms used in the present day.

There will be a discussion about the reasons and causes of the problems faced by the industry and the possible solutions. This will be done in a general and a reconfigurable-centric manner. The architecture of the proposed system will be presented with an in-depth explanations of some of its most importants modules.

**Acknowledgements**

I would like to thank Dr. Voicu Groza for his ongoing support, watchful eye, and assistance. I would also like to thank Dr. Rami Abielmona for this help and contagious passion for research and hockey. Their help has been tremendous and can not be overstated, without it none of this would be possible. Thanks also to everybody that has supported me knowingly or not.

Special thanks go to the Canadian Microelectronics Corporation (CMC) for their hardware and software donations, and technical support.

Lastly, I would like to thank my family for their support and patience.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Most electronic devices available nowadays are based on the conventional single processor which has a predefined set of instructions that are exploited by computer programs in order for the device to provide the intended service. The use of conventional processors in most devices continues to this day but not without intensive research into other computing methods that may provided other benefits such as better performance, power usage, etc.

This thesis presents the contributions made to the *Hardware Operating Systems* (HOSes). HOSes is a co-design environment used to create applications that are composed of hardware (*i.e.*, circuits) and software elements. This is possible thanks to the ongoing research in *reconfigurable computing* (RC). RC technology permits the creation of logic circuits that can be modified after the semiconductor has been produced and packaged.

## 1.1 Motivations

Current hardware devices are based on conventional processors that have a fixed set of instructions. This type of processors execute a sequence of instructions in order to run an application. Examples of conventional processors are the Pentium III and PowerPC, produced by Intel and Motorola/IBM respectively. RC research presents a new paradigm that uses new devices that are different than the conventional processor. The RC field is receiving a considerable amount of attention. The monthly publication of IEEE's *Computer* dedicates the March 2007 edition to RC. This publication targeted the specific application of *high-performance reconfigurable computing* (HPRC) which looks into methods on how to benefit applications which require powerful computing system. These types of applications are benefiting from the possibility of parallelizing the intensive computing task which for now are somewhat limited as pointed out by Buel *et al.* [1].

The different articles published in the March 2007 edition of *Computer* that deal with the field of RC use a particular technology which has become the most widely used because of its popularity and history in the industrial market. These RC devices are named *field programmable gate arrays* (FPGA) and they were introduced in the mid-1980s by Xilinx which were then utilized as glue logic [1]. FPGAs are not the only technology available for RC. Hartenstein in [2] shows several technologies used in reconfigurable applications. He "*surveys a decade of R&D*" on different types of reconfigurable devices and their *computer aided design* (CAD) tools.

RC is a relatively young field that lacks well established development processes, tools, algorithms, etc. A snapshot of the current state can be seen in the same March 2007 edition of IEEE's *Computer* which present articles that tackle new tools, design techniques and implementation of algorithms that exploit the high parallelism of a reconfigurable device. Tripp *et al.* [3] presents Trident which is an "*open source compiler that translates*

*C code to a hardware circuit description.*" The RC field is a relatively new technology and as such it requires better tools than what is available now. Trident is a compiler that transforms C code into circuit description. This allows a whole set of designers, that are not versed in circuit design, to create circuit description. The article also presents a summary [3, p. 29] of some technologies that are used or are being researched for synthesizing circuits. There are other methods being studied in order to attract software designers to the RC field such as proposing a software library available to application developers. These libraries provide a relatively simple *application programming interface* (API) and internally they utilize the reconfigurable devices made available to them. Such an implementation is presented in [4] by *Moore et al.*.

There is lots of research being performed in order to exploit the parallel capabilities given by RC. Morris and Prasanna [5] provide an in-depth look into an implementation of algorithms that tackle double-precision floating-point sparse matrix iterative-linear-equation solvers while taking advantage of the FPGA platform. Another application is presented by Alam *et al.* [6]. They developed an "*implementation of a molecular dynamics simulation method*" on a FPGA. This article provides some insight on the challenges that the researchers had to overcome in order to implement algorithms used to accelerate biomolecular simulations. These publications are signs that the RC community is very active and that there is much work still left to be done before these systems can be introduced to a bigger market.

The RC field is still in its infancy and much work must be still be performed before it becomes a valid option for designers everywhere. Before this happens, it is necessary that better tools appear in the market at reasonable cost and given the apparently small size of the RC community, this may take some time. Also, RC benefits from the advances that the FPGA technology makes since any RC project is directly affected by the nature

of the reconfigurable devices along side with the available design tools. Santi [7] provides a snapshot of the current thinking in some *application specific integrated circuit* (ASIC) design circles which are starting to use the FPGA in roles that were reserved to ASIC before. This article shows the different problems encountered by the designers and their solutions when replacing ASICs with FPGAs. It is mentioned in the article how the tools are important and immature when compared to ASIC tools.

## 1.2 Contributions

RC research is advancing continuously and since it is relatively new, development tools are still being developed as we can see from some of the articles cited above. HOSes brings a new paradigm in development that tackles the run-time reconfiguration (RTR) which is a subfield of RC. This thesis presents the HOSes architecture along with the integration of its disjointed components as an entire system. The innovations of this work in the field of reconfigurable computing are presented in the following list:

**Hardware abstraction** We conceived the HOSes architecture such that it abstracts the hardware platforms. Therefore, it presents to its users a general and a standard, yet powerful, interface and model of configurable devices.

**New BIST development environment** I devised and implemented a new development environment which provides a platform that allows BIST (see section 2.3) researchers to develop new methodologies. This is done by exploiting the run-time reconfiguration (RTR) capabilities of reconfigurable devices in order to modify circuits characteristics with a speed until then unmatched.

**High parallelism** HOSes permits the creation of highly parallel hardware applications. While this is offered by all hardware development tools, HOSes provides in addition

a simple but powerful development environment.

The technical aspect of the research focuses more on the implementation of HOSes as a complete system. This thesis objective is to present a system that is based in reconfigurable devices and which provides a new platform for development of BIST methodologies. The most important contributions are listed below.

- I designed the bridging layer which performs key functions such as controlling lower layers and transferring information between the upper and lower layers. See section 4.2 for more information on HOSes' layers.
- I have designed and implemented parts of the architectural and physical layers which abstract the hardware platform.
- I have updated the design and implementation of the physical layer so that it can be used with the new bridging layer.

This thesis serves as a consolidating body of work for HOSes, presenting the development, implementation, testing and analysis of such a system in the reconfigurable computing field in general, and in run-time environment (RTE) field in particular. Within HOSes, the architectural, bridging and physical layers are disclosed in detailed form, and the application layer is presented in summary form.

## 1.3 Outline of the Thesis

This thesis introductory section summarizes the current status of the RC research and the project that aims to provide a new kind of tool for this nascent field. Chapter 2 gives a background introduction on run-time reconfigurable systems, reconfigurable architectures, run-time environments, partial reconfigurable devices, built-in self test, co-design

development and JBits. Chapter 3 describes the different issues affecting the computing industry which in turn provide the reasons in the undertaking of a project such as HOSes. This includes a broad discussion on current conditions that the industry is experiencing, but also the current need for a development platform of tests and methodologies used to verify integrated circuits; which would allow for the development of faster and cheaper semiconductors. Chapter 4 presents HOSes which is a solution that tackles the challenges listed in chapter 3. The final chapter, chapter 5, presents conclusions, alternatives to HOSes, and a general discussion on what the future may hold for HOSes and the industry in general.

# Chapter 2

# State of the Art

This chapter gives an introduction to the different concepts and technologies that this thesis project (*i.e.*, HOSes) is based on. The first section will touch upon the computing paradigms that govern this new technology which is HOSes. The second section will explain the hardware platform used which permits the development of reconfigurable systems. The third section will present the application used to test the system. The fourth section touches upon the design process used when designing applications targeted to reconfigurable devices. The final section presents the specialized tool provided by the manufacturer of the hardware platform and which is essential for the realization of this project.

## 2.1 Computing Paradigms

In order to better understand the challenges, decisions, and implementation for this research, it is important to review the concepts that are used in the project. This research is based on a relatively new form of computing that uses technologies that may not be in the vocabulary of those not immersed in the field. For this reason and in

order to better present the different concepts and technologies, a classification of the different computing paradigms that this research deals with are presented below. The classification is done by comparing the configuration capabilities that are available for different technologies that are currently available. The classification is as follows:

- Non-Configurable Computing
- Configurable Computing
- Reconfigurable Computing

As stated previously, the classification is performed by comparing the configuration capabilities of the different devices and/or system currently available. We can define *configuration* as the ability of an electronic system to change its internal arrangement and functions by the modification of parameters possessed by said system. This will become clear in the following sections.

We must keep in mind that this classification targets not only *Integrated Circuits* (IC) but also the systems that are created using one or more discrete elements that themselves may not posses the studied characteristics. Also, despite the fact that this section tries to not show a preference for a particular technology, it is important to know that this research utilizes FPGAs which are studied in more detail in section 2.2. Therefore, a bias may be detected when reading this section. This should not be taken as an indication that this type of device is the only one available. Hartenstein in [2] provides an excellent overview of several other technologies used in this field. The term used for any of these types of devices is a *programmable logic device* (PLD) and the concepts presented here can theoretically be applied to any PLD.

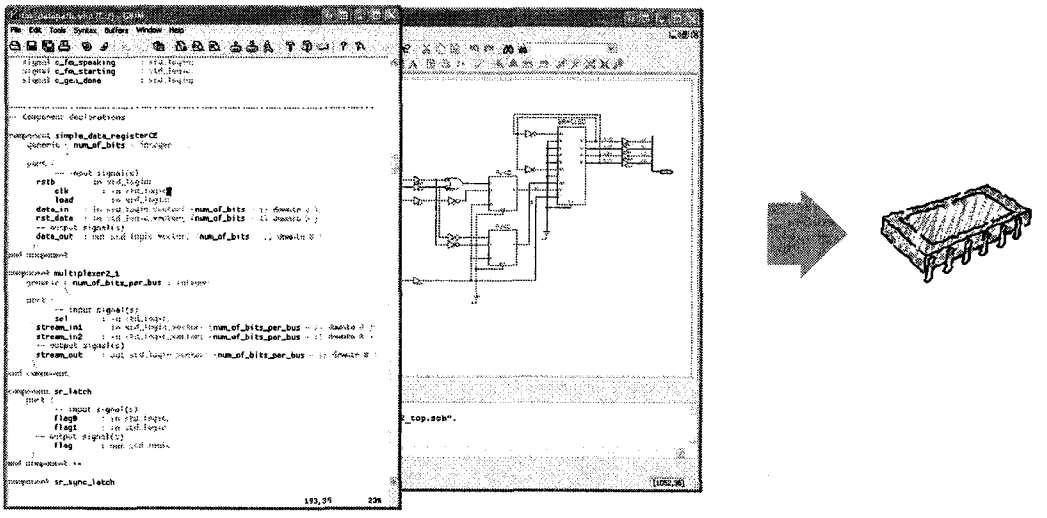


**Figure 2.1:** *To create an ASIC a designer will use several tools such as schematic entry and/or a hardware description language (HDL).*

### 2.1.1 Non-Configurable Computing

A *Non-Configurable* system is such that the hardware that composes it can not be modified after it has been fabricated or assembled. An *Application Specific Integrated Circuit* (ASIC) is an example of a Non-Configurable system. However, this does not limit the capabilities of such a system. For example, the ubiquitous personal computer (PC) is normally made using different fixed (*e.g.*, ASIC) components while providing general purpose functions. This capability is obtained by changing the software while keeping the same hardware components.

To better understand the Non-Configuration aspect and subsequently the Configuration paradigm it is necessary to look at the way an ASIC is designed and created. The figure 2.1 depicts the fact that a circuit is created by using different applications and that the final output of this process is a physical device that has been created in a special *fabrication facility*, usually referred as a "fab", whose sole role is to produce

semiconductors.

A non-exhaustive list with the advantages and disadvantages of non-configurable systems is shown below.

Advantages

- Usually optimized for particular function or requirement such as power consumption, computing speed, size, etc.
- Cost-effective when mass produced when compared to a configurable system with the same functionality, since its internals have been tuned to the specific application, which likely will require less resources.

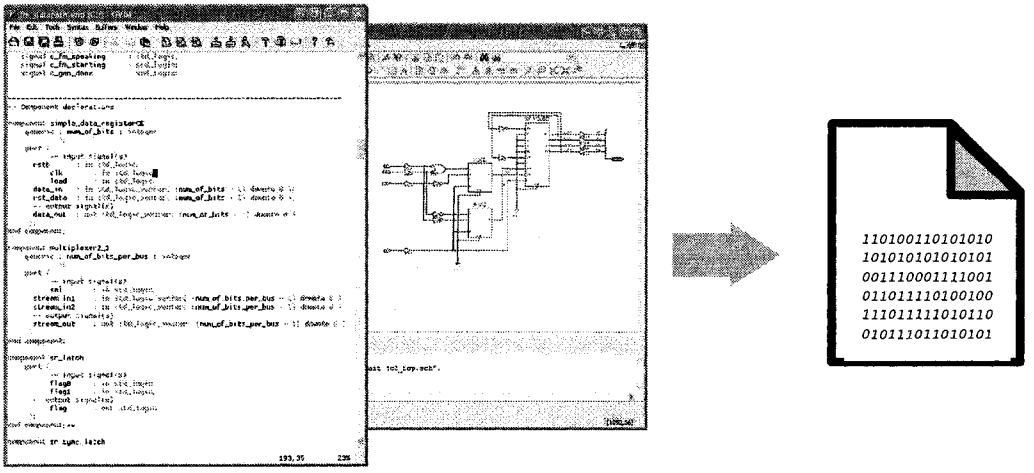
Disadvantages

- High development cost.
- Hardwire, thus, no room for rework in case of an error or improvement.
- Long product turnaround time since it most be produced in a fab.

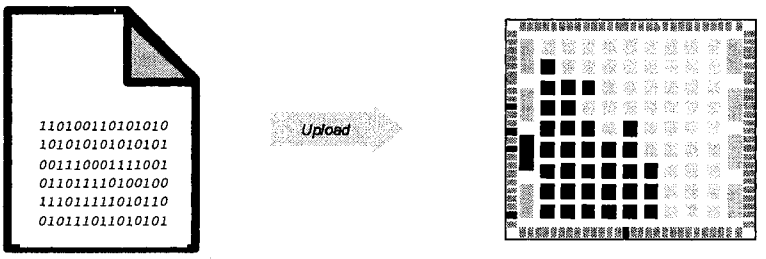
### 2.1.2 Configurable Computing

In this category we find systems that have the property, among others, of being capable of modifying their internal electric paths and logic gates in order to meet a particular set of requirements. These devices are consequently on one end of the spectrum while ASICs, for example, are found on the other end because their internals are etched permanently when fabricated. The creation and configuration process is shown in a simplistic form in figure 2.2.

The bitstream shown on the right of figure 2.2(a), is a computer file that is used to program a configurable device (see section 2.2). It is created using different programs




(a) Bitstream generation




(b) Device configuration

**Figure 2.2:** *The two main steps to create a configurable design are shown in this diagram.*

such as a schematic capture, much like when creating an ASIC. To program a configurable device a bitstream is uploaded into the device (figure 2.2(b)).

The are many advantages found in a configurable system as we can imagine. But we will also find disadvantages that are not found in non-configurable systems.

In order to simplify the classification we will place all the system that can be configured only when the system is idle or not active in the Configurable field. All other system that can modify their hardware characteristics without stopping execution will be classified as *reconfigurable* system which are studied in the next sections.

Advantages

- In most cases it is cost-effective when compared to Non-Configurable systems (*i.e.*, ASICs)..
- Quick turnaround time when compared to non-configurable systems.
- Flexible and ease in design process since it is not necessary to use a fab to create the system.

Disadvantages

- Inefficient use of hardware resources since the configurable device is usually not designed with a specific application in mind.
- System must be stopped when reconfiguring.
- Usually a host must be present in order to perform the configuration.

### 2.1.3 Reconfigurable Computing

*Reconfigurable Computing* (RC) is the relatively new field of computation that uses a special characteristic of some configurable devices. This characteristic is the ability to modify only a small part of the device while the rest remains untouched. This is referred as partial reconfiguration. One of the main goals of RC is to provide a system that brings the benefits of software flexibility to a hardware platform.

*Partial reconfiguration* (PR) is the ability of a system to accept modifications for only a part of the configurable elements and not to the complete system. This ability permits the designers to use only a subset of the logic elements inside a configurable device. The designed circuit, which does not use the entire device but only a part of it, is referred as a *hardware block* (HB). By using this technology, the designers is able to focus on the design of a HB instead of a complete system. While we will refer to a HB as the partial

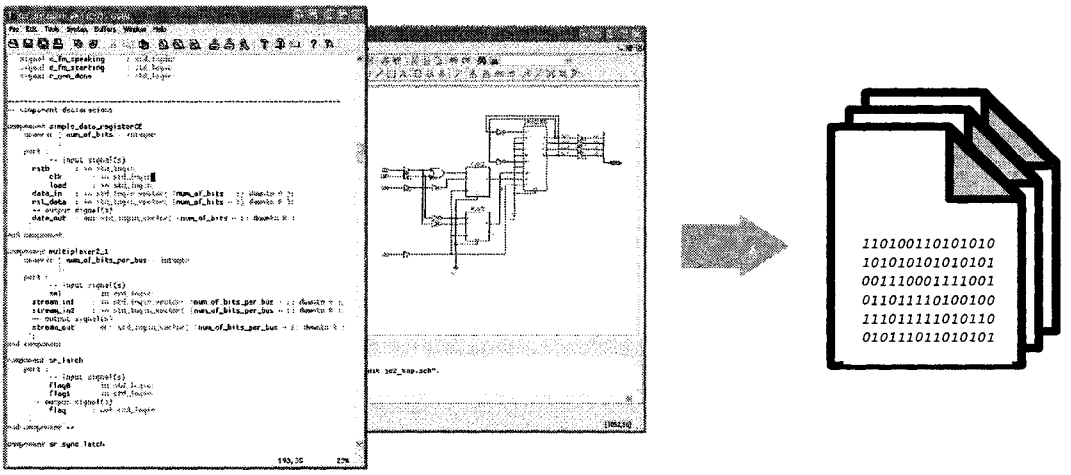



**Figure 2.3:** *To create a reconfigurable system the designer will create different files for each hardware block (HB) that will be moved in and out of the device. The system must support partial reconfiguration for this to work.*

bitstream that is loaded into a configurable device in order to complement the current system, the term will take a slightly different meaning in the "Solution" section 4.2. In that section it will be seen that a HB represent the smallest hardware element that is used to build an FPGA-based hardware application.

A system that supports the use of HBs must supply several services in order to provide a working infrastructure required by the HBs. Because we are dealing with hardware it is imperative to devise an efficient method for routing all the electrical paths such that HBs can be swapped in and out. Another important element of a RC system are the elements that manage the reconfiguration. As we will see later the controlling elements do not need to be physically or logically separated from the reconfigurable device. In fact, the logic the manages the reconfiguration can reside inside the same device.

Below is a list of advantages and disadvantages for RC systems.

Advantages

- Reconfigurable systems benefit from the same advantages as configurable systems.

- Reconfiguration on the fly.
- More efficient use of the hardware resources when compared to configurable systems since the device can be configured at any point in time.
- Better power consumption since not all hardware is utilized at the same time

Disadvantages

- Heavy overhead in order to support routing between block when swapping logic in and out of the system which could affect heavily the performance of the system.

In the following sections we will see that the result or output of reconfigurable computing is a *reconfigurable architecture* which is studied in more depth in section 2.1.3.1. We will also see how the development of the applications that take advantage of the reconfigurable capabilities is undertaken.

#### 2.1.3.1 Reconfigurable Architecture

A *Reconfigurable Architecture* (RA) is PLD-based system that implements the concepts of RC. Such a system can be partially reconfigured in order to meet the demands of the application being executed. This is a benefit when the hardware resources are not sufficient for a simultaneous use of all HBs.

The internal structure of a RA will vary greatly depending on the technology used and the target application. But one element that will always be present is the *reconfigurable processing unit* (RPU) which is the reconfigurable entity in the system. It is normal to find multiple RPUs in a given RA. Cardoso et Véstias [8] classify RAs depending on how the RPU is coupled to the rest of the system. This classification is shown in figure 2.4. This classification is as follows:

- **RPU coupled to the I/O bus**. In this setup the RPU is connected to the system through the I/O bus. An example of this will be a PCI card connected to a motherboard of a modern computer. See figure 2.4(a).

- **RPU coupled to the local bus**. With this setup the RPU will connect to the system through the local bus. Following the idea of the previous example, the RPU will be found in the motherboard of a PC and will use the said bus instead of the slower I/O bus. See figure 2.4(b).

- **RPU coupled to the CPU**. This setup uses the RPU like a co-processor. See figure 2.4(c).

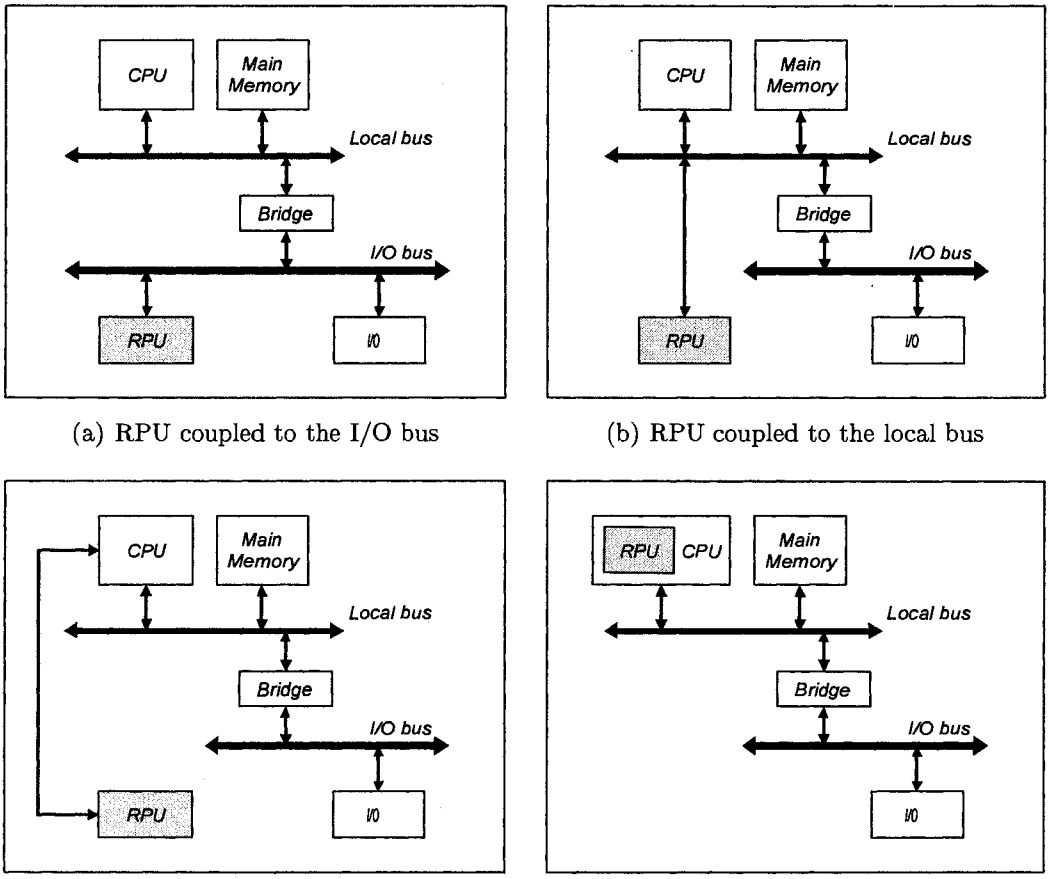
- **RPU coupled integrated in the CPU**. In this setup we find the RPU acting just as a extended datapath of the CPU. See figure 2.4(d).

Different configurable devices are designed with different goals in mind. This will dictate the actual architecture of the system, specially the dimensions, in bits, of the configurable elements of the device. This characteristic is referred as *granularity*. A loose and non-official classification of granularity is shown below.

- **Fine-grained**: Systems containing configurable elements with paths that are only a few bits wide. Usually one or two bits.

- **Medium-grained**: These are devices with paths of 4 bits wide give or take one or two bits.

- **Coarse-grained**: Classification given to devices with paths of 8 or more bits wide.




(a) RPU coupled to the I/O bus

(b) RPU coupled to the local bus

(c) RPU coupled to the CPU

(d) RPU integrated in the CPU

**Figure 2.4:** *Classification of RA system with respect to the coupling of the RPU to the host.*

#### 2.1.3.2 Run-Time Reconfiguration

*Run-time reconfiguration* (RTR) is the ability of the direct manipulation of the internal hardware resources without halting the execution of the different task that are running simultaneously in other parts of the same system. The key aspect is the ability of reconfiguring only a part of the system *while* the rest is actively pursuing its duties.

RTR allows for new and exciting functionalities such as *virtual hardware* where the actual logic circuit is only "loaded" as needed, hence, freeing the logic resources for other

HBs. As we may expect, implementing RTR is quite complex since many aspects that are relatively old in the software field have to be implemented for hardware entities that impose more challenges than software development.

#### 2.1.3.3 Run-Time Environment

The main purpose of a *Run-Time Environment* (RTE) is to provide an abstraction of the hardware to designers in order to provide a generic model that frees the developers from a particular platform. By abstracting a RA, the designer is given more freedom from the idiosyncrasies of the design tools and the particular characteristics of the platform that are encountered when designing a complete system. Hence, an important benefit obtained by providing an RTE is the widening of the pool of designers thanks to the abstraction provided by it.

Also we find development environments and co-design languages. While these are not considered RTEs they do provide hardware abstraction and libraries which give more freedom to the designer. These are referred as co-design languages environments given the fact that they help to tackle hardware and software challenges simultaneously. Examples of such environments are Handel-C, Single Assignment C, System C, and Matlab. The paper by Dr. Groza [9] provides a more in-depth information regarding these languages and environments while showcasing the research in such a co-design environment at the GEMS lab (see section 4.2 for more information on GEMS lab).

Below is a list of different projects that have as a goal to provide designers with tools that will allow them develop RC system while abstracting as much as possible.

**FOCA** [10]. FPGA Operating system for Circuit Automation divides a FPGA into equal reconfigurable slots where different tasks can be placed.

**Run-Time Reconfiguration Manager** [11]. This system monitors the different

tasks and decides how to place them in the available slots while efficiently managing the time allocated to each task.

**UCS/HTS** [12]. This project extends the Linux operating system and the GNU applications such that it can manage applications that are composed of software and hardware parts.

**QuickFlex** [13] is a company that developed the Runtime QuickFlex Resource Manager which extended an OS such as Windows, Linux or VxWorks into being able to manage reconfigurable hardware. There seems to be little development lately and there is not much information available.

**Trident** [3]. This is an open source compiler that translates C source code into hardware circuit description. Designer can therefore work in a single system and the tool will generate an application with both the hardware and software elements.

**Vforce** [4] is a framework that abstracts hardware resources into application program interfaces (API) which allows to develop applications that take full advantage of hardware components without explicitly being programmed to do so.

#### 2.1.3.4 Run-Time Environment Specific Features

To bring to life an RTE, one can expect many challenges. To have an idea what type of challenges are to be overcome, we can look how software *operating systems* (OS) are designed since an RTE will have to manage HBs much like an OS manages software tasks. Diessel and Wigley [14] identified several characteristic that an RTE must hold in addition to the services provide by the OS used in the same system. These additional RTE functions are presented in table 2.1.

| Functions |
|---|
| Task sizing |
| Task placement |
| Location independence |
| Swapping and caching |
| Input and output |
| Inter-task communications |
| Allocation and scheduling |
| Security |

**Table 2.1:** *Breakdown of the functions that an RTE must provide in addition to normal OS services.*

The following paragraphs will give a somewhat detailed explanation of the different RTE functions shown in table 2.1.

**Task Sizing** A RTE must deal with the physical characteristics of the PLD being used. This has the effect of restricting how the task's HB is handled. The RTE is in charge of efficiently use the available free space and this one will handle the different HBs depending on how the partitioning of the free area is performed. If the partitioning is *fixed*, then the HB is placed in a empty block that is big enough to accept it. The disadvantage of this approach is that there will be unallocated space since it is unlikely that HB will fit perfectly in a given partition. The second option is to implement *variable partitioning* where the RTE is in charge of modifying the spacial characteristics of the HB in order to fit it into the free space. This allows to use the device space more efficiently, but at the cost of RTE complexity and probable performance loss.

**Task Placement** In a RTE the different tasks that are swapped in and out must be placed in empty areas of the configurable device. Task placement algorithms are in charge of placing these tasks. The complexity of the placement task will depend on

the partitioning selected for the system. See "Task Sizing" above for a discussion on partitioning.

**Location Independence** By designing the RTE tasks that are independent of the location in the reconfigurable device, the system has more freedom and therefore more efficient in placing the HBs.

**Task Swapping, Caching, and Pre-loading** To better support system Multitasking, an RTE can implement *time-sharing* in the configurable device. Implementing time-sharing between task or HBs, the RTE must be able to swap the HBs, cache the context information and pre-load of soon to be used HBs.

**Input and Output** Any RTE that supports multiple HBs must coordinate I/O stream. Since configurable devices have limited pin and routing resources, the RTE must efficiently manage the stream that the different HBs require in order to communicate with the outside world.

**Intertask Communications** To effectively provide a RTE that can handle multiple HBs, this one must manage in an efficient manner the communication between HBs or tasks.

**Allocation and Scheduling** Depending on the goal selected when designing the RTE, this one must try to allocate the right number of HBs that are going to be placed at any given time. Also, the RTE must try to efficiently line up the different task to be run over time, hence affecting the scheduling process.

**Security** This refers to the ability of the RTE to protect the functioning and stability of the system. It also refers to the need to restrict access to users depending on their permission levels.

## 2.2 Field Programmable Gate Array

A *field programmable gate array* (FPGA) is an integrated circuit IC that does not have a particular logic function defined by the manufacturer when fabricating the die. Instead the user can configure (*i.e.*, program) the desired functionality into the FPGA by using special development software and loading the resulting configuration parameters into the device. Hence, the FPGA is a configurable device.

In the following sections we will find the terminology, process, and technology used in the world of FPGA development. This section tries to be device agnostic but because the project uses the Virtex technology which is made available by Xilinx Inc., it will not be devoid of bias toward that particular technology. See figure 2.6 for an overview of the FPGA architecture.

### 2.2.1 Design Flow

The sequence of steps and techniques used to design an FPGA or ASIC is called a *design flow*. In this thesis we will concentrate on the FPGA design flow only. There are different types of design flows. For example, there are flows that use schematic capture where the design engineers will use *computer aided design* (CAD) programs to draw the schematic of the desired circuit. Others will use a *hardware description language* (HDL) like VHDL (Very High Speed Integrated Circuit HDL) or Verilog for example. These techniques are not mutually exclusive, they can be used in combination. The project requirements will often dictate the steps to take when designing a new configuration. The typical design flow used by the designers, to develop FPGA-based applications, is shown in figure 2.5.

The output of a design flow is a *bitstream* . This bitstream holds the information used to configure the FPGA. Figure 2.2 shows that the bitstream is created using different applications which are usually supplied by the same manufacturer of the physical device.

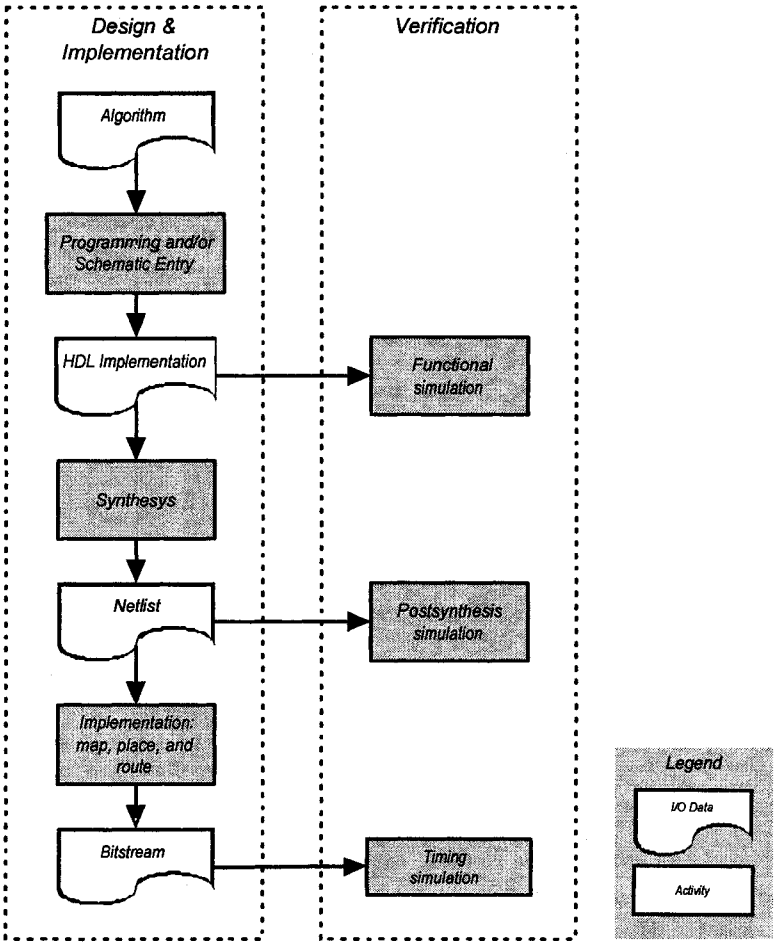



**Figure 2.5:** *Typical FPGA design flow which produces, as its final output, a bitstream. This is a modified version of the diagram that appears in [1, p. 25].*

There is also *partial-bitstreams* that are used to configure only a part of the chip and not the entire fabric. This has been covered in section 2.1.3.

### 2.2.2 Configuration

The FPGA is, as stated earlier, an IC that does not have a particular function etched in its fabric. The FPGA must be programmed in order to use it for a particular function. This is achieved by loading digital information into the device that specifies how the different

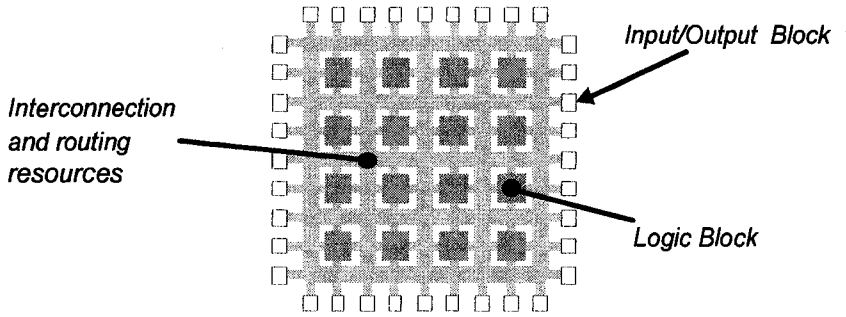



**Figure 2.6:** *This diagram depicts a generalization of the internal structure of an FPGA. Actual devices are more complex and usually contain other types of elements such as ALU and RAM blocks.*

internal logic elements behave and are wired together. This process of programming the FPGA is referred as *configuration*.

### 2.2.3 FPGA Internals

Different FPGA architectures can be categorized as fine-, medium-, and coarse-grained as seen in section 2.1.3.1. The Virtex family of configurable devices is considered fine-grained since the smallest bit path is one bit wide.

Internally not all FPGAs are equal. Different makers will use different technologies between them, but also between models. But regardless on the particular technology used, all the devices must have an architecture that implements logic blocks which are configured by the application designer using a flow. FPGAs also possess *I/O blocks* (IOB) that are also configurable. The goal of IOBs is to provide access to the outside world through the different pins found on the chip. These configurable resources need to communicate between each other; as such, an infrastructure of electrical paths and multiplexers must be provided in order to complete the device. A depiction of the FPGA architecture is presented in figure 2.6.

The Virtex family of FPGAs are SRAM-based. In this type of FPGA, the configu-

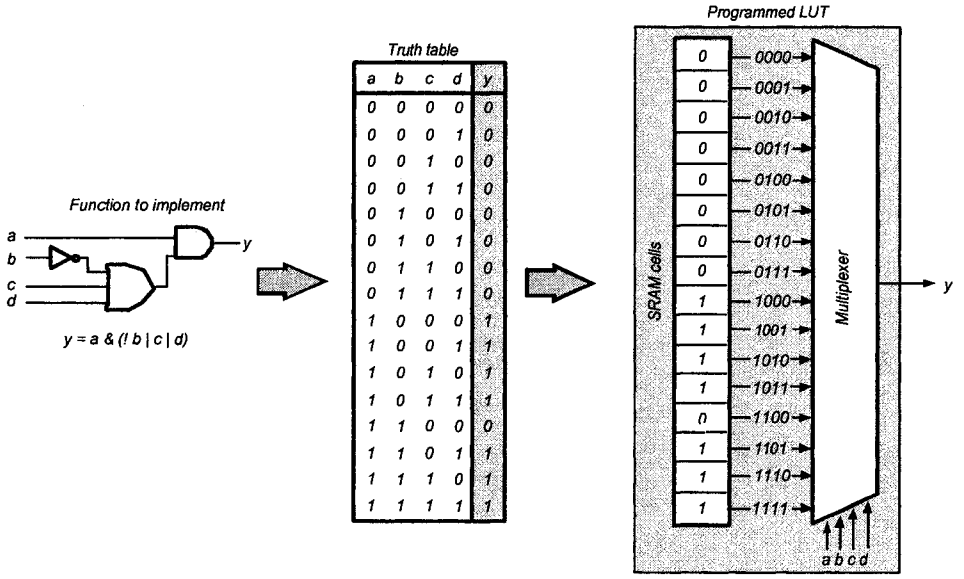



**Figure 2.7:** *This diagram shows how a simple circuit is implemented using a* look-up table *(LUT). This is a modified example from [15, p. 51].*

ration data are stored in the on-chip SRAM. Because SRAM looses its contents when it is powered off, the device needs to load its configuration parameters every time it is powered up. The advantage of using SRAM technology is that the reconfiguration is fast since SRAM technology is faster than FLASH or EEPROM technologies which are also used in reconfigurable devices. As expected, fast configuration is an important characteristic for a reconfigurable application.

#### 2.2.3.1 Look-Up Tables

The role of a *look-up tables* (LUT) is to provide the same functionality as a circuit composed of a few gates. This is accomplished by creating a table with all the possible inputs and the corresponding outputs of the circuit to implement. Then a multiplexer is used to select the right output given the received inputs. This way the LUT behaves as it was the original circuit. This is shown in figure 2.7.

A LUT is composed of memory cells and a multiplexer. In the Virtex family and all Xilinx products SRAM is used to store the configuration bits in the LUT. The number of cells and the multiplexer size depends on the number of inputs for the given LUT. For example, to implement a circuit of four inputs and one output, the array of memory cells must have a length of sixteen bits and the multiplexer should have a ratio of 16 to 1. This is the technology used in the Virtex devices. This traduces to the use of a 16 SRAM cells and a 16:1 multiplexer.

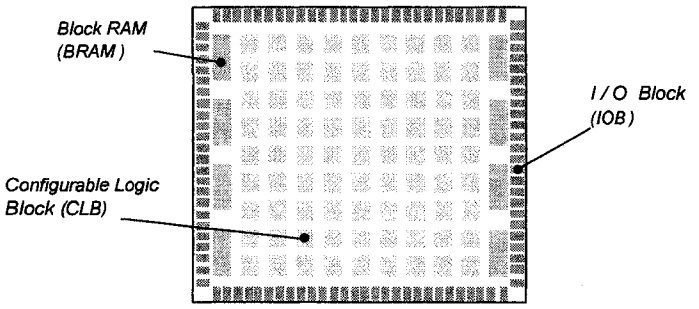
### 2.2.4 Virtex FPGA

Xilinx introduced the Virtex family of FPGAs in the late 1980s. The first devices were formed of a grid of logic elements that formed a square in the middle of the chip. *Block RAM* (BRAM) was found on two sides of the square opposite to each other. IOBs are found in the outside on all four edges of the die. Figure 2.8(a) shows the chip layout. For an in depth description see the Virtex data sheet [16].

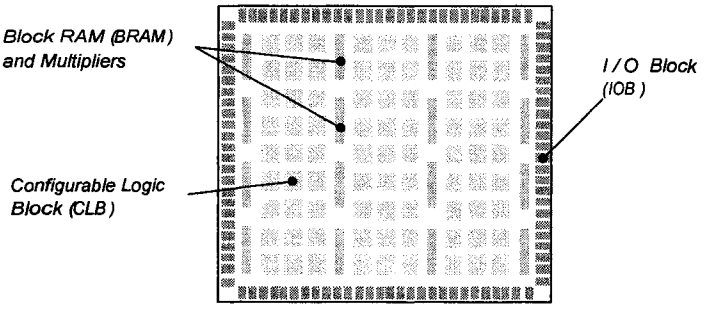
Later devices such as Virtex-II, Virtex-II Pro, Virtex 4 and 5 are more complex. The layout in these devices differ from the earlier generation. In this devices the logic is arranged in columns that are separated by rows of BRAM, multipliers and other logic. This is depicted in figure 2.8(b). See section 2.2.4.2 for more information about new generation chips.

#### 2.2.4.1 Configurable Logic Block

A *configurable logic block* (CLB) is the highest in the hierarchy of logic elements found in Xilinx FPGAs. CLBs contains the different elements that help implement the target circuits defined in the bitstream. CLBs are then divided into similar elements called *slices*. Each slice will contain multiple LUTs which in older generation of Virtex FPGAs




(a) Earlier generation Virtex architecture




(b) Column-based architecture for newer Virtex devices

**Figure 2.8:** *These diagrams show the internal architecture of the different Virtex FPGAs. Note the column arrangement of the logic elements in figure 2.8(b).*

each LUT was contained inside a *logic cell* (LC). This LC implements the LUTs and its required glue and helper logic [16]. LCs are found to be the smallest reconfigurable unit in these older generation of devices. A functional view of a CLB found in modern Virtex devices is shown in figure 2.9 which depicts two LCs per slice event though Xilinx does not use LCs in newer chips since the Virtex-II was introduced to the market. LCs are shown here in order to simplify the diagram.

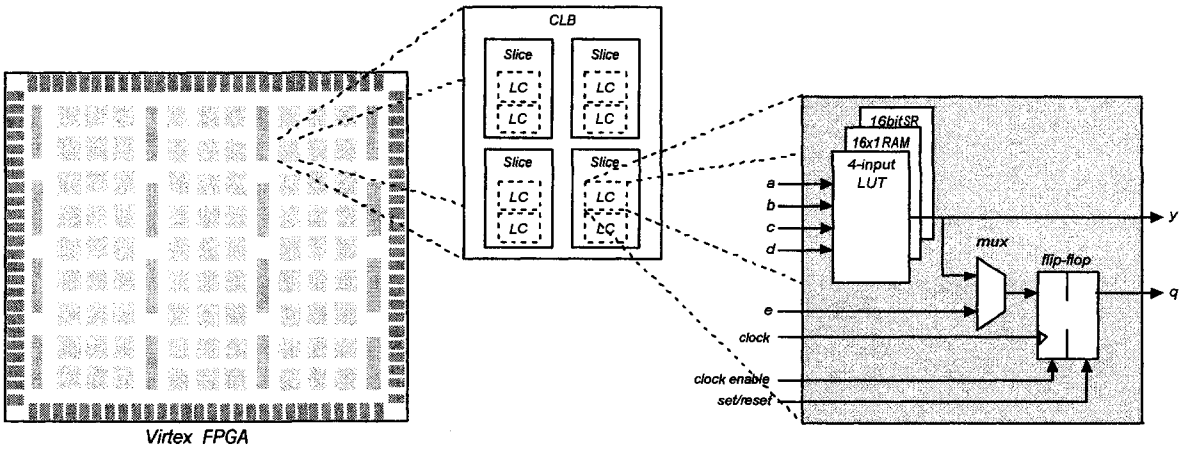



**Figure 2.9:** *This is a simplified view of the breakdown of a Xilinx Virtex FPGA. Note that Xilinx does not use LCs in its documentation for its newer chips, but it is used here for simplification.*

#### 2.2.4.2 Newer Virtex Chips

Newer Xilinx products use a technology based on columns of configurable logic including the Virtex-II, Virtex 4 and Virtex 5 families. These columns are separated by RAM, multipliers, and other components. CLB are found in all devices but the number of slices will differ from model to model. For example while the Virtex-II and Virtex-II Pro models contain four slices per CLB, the Virtex-5 contains two slices per CLB. See the Virtex-II data sheet [17] at page 35 for a complete CLB and slice description. Also see [18] for an overview on the newest chip, the Virtex-5.

A simplified diagram that shows the different components in a Virtex device is shown in figure 2.9.

The column-based layout of the modern Virtex FPGA raises new challenges for partial reconfiguration designs since the algorithms must account for the overhead that this causes such as efficiently placing the HBs in the given partition which spans for more than one column.

## 2.3 Built-In Self Test

Computer system are not exempt of defects in production and or design. Therefore it is important to have methods that allow to test every element in the system including hardware and software components. *Built-in self test* (BIST) is a technique used to test for faults in an electronic circuit. This research project uses BIST as a target application since the GEMS lab (see section 4.2 for more information about GEMS) at the University of Ottawa with the collaboration of other researchers have used BIST in order to advance RC research. The following three papers are a sample of publications made by this group and collaborators: [19], [20], and [21].

The paper by Khalaf *et al.* [19] presents an implementation of BIST methods that exploit the dynamic partial reconfiguration abilities of a reconfigurable device. Assaf *et al.* [20] discusses an architecture that uses JBits (see section 2.5) in order to implement BIST techniques. A final example of the collaboration is the paper by Assaf *et al.* [21] which looks into utilizing also JBits for BIST research.

BIST provides the mechanisms on how to perform test on a circuit. For this reason, the research in BIST methodologies is an important area since the test used in the target circuit are highly dependent on the circuit nature such as its size for example. Hence, BIST designers need better tools, and this research project provides a platform that researchers can use to test their BIST methods. Because of this, HOSes is an important addition to the researchers' tool belt, thanks in part to the use of reconfigurable concepts exploited by it.

In the following sections an introduction is given on how BIST is used in order to discover errors in hardware designs. It is important to note that this section will not cover all aspects of BIST, but only those that are relevant to this research.

### 2.3.1 Fault Injection

While there are a myriad of testing methods for discovering errors in digital circuits, we will focus on one in particular referred as *fault injections*. Hsueh *et al.* [22] provide a good deal of information about this methodology and mentions the following regarding fault injection: "Fault injection tests fault detection, fault isolation, and reconfiguration and recovery capabilities." A fault injection is in its most simple form, a change in the logic level in a particular electrical path in the circuit being tested. If the logic level injected is high, then it is referred as *stuck-at-1*, and if the logic level is low we refer to the injection as a *stuck-at-0* signal. As these terms indicate, the fault is in fact an arbitrary logic level injected in a specific point in the circuit which is then stuck into that particular logic value.

### 2.3.2 Circuit Under Test

A *circuit under test* (CUT) is the target circuit where the fault injection is being performed by applying or injection of a logic signal into a given electrical path in order to create a fault. This permits the designers to evaluate the failsafe mechanisms that are added into the circuit in order to verify the correct functioning of the device. For example, a crude way of performing fault injection in a circuit is to physically invade a CUT with the right equipment and inject a new voltage level into it. As one can expect, this method will prove to be difficult and in most cases impossible given the size of current ICs.

### 2.3.3 Test Pattern Generators

There is extensive research being perform in order to find better ways to test for faults in electronic circuits. This also applies to how those test are performed. This includes the

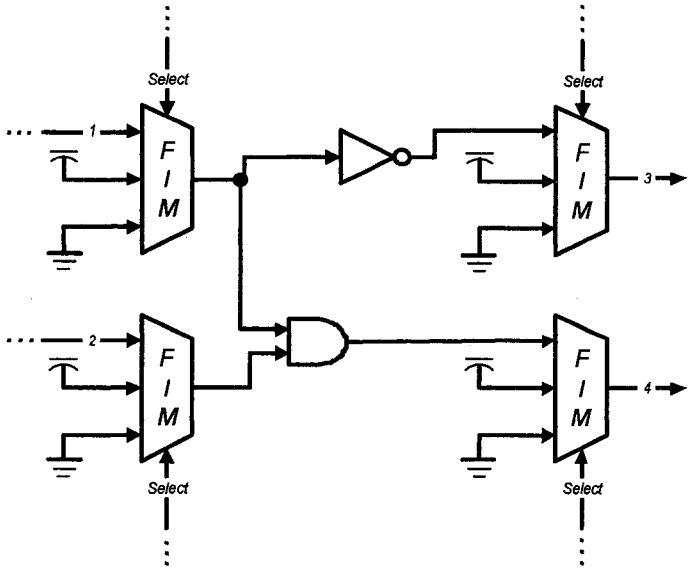



**Figure 2.10:** *This diagram shows the BIST hardware infrastructure in a given CUT. The injection of a fault is done by selecting the desired output in a given fault-injection multiplexer (FIM).*

parameters used to apply the different faults into the CUT. The techniques used to create the *test patterns*, which are the actual injection signal information, are implemented in *test pattern generators* (TPG). Researchers are constantly working in order to find better ways to generate the test patterns which drives the design of better TPGs.

### 2.3.4 Test Methods

Test that are applied to a CUT can be performed in two different ways, usually this will depend on the size and complexity of the circuit in question. One method is to apply a *deterministic* test which applies a known test pattern into the CUT which will produce the corresponding output. The other method is to apply a *pseudo-random* method that generates a test pattern that while being almost random, will test the different target areas of interest in a CUT.

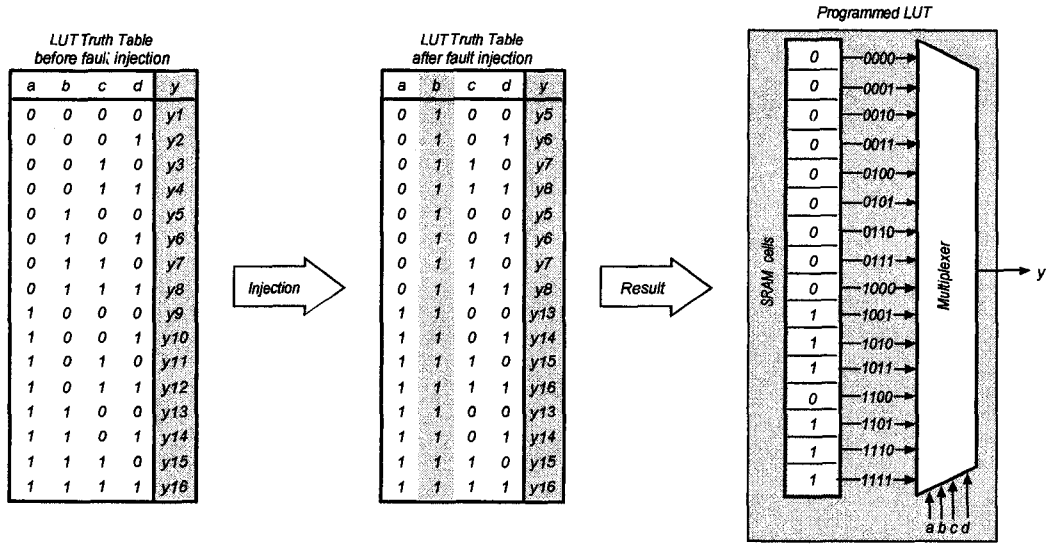



**Figure 2.11:** *This figure shows the effects on a LUT when a stuck-at-1 signal is injected in input* ***b****. See figure 2.7 for a diagram of the circuit that this LUT implements.*

GEMS has used two different techniques to implement BIST. They differ on how the reconfigurable capabilities of the device are used in order to inject the faults. These two types are discussed in the next sections.

#### 2.3.4.1 Compile Time Reconfiguration

Any BIST implementation requires an infrastructure since the invasion of a CUT cannot be performed unless some kind of tool is used to inject a signal into the circuit. For this reason, different logic elements are added to a design at *compile time* in order to support BIST. The designer will add the different logic elements required for the designed CUT using the appropriate CAD program. Once the design is loaded into a device, the CUT is ready for testing. Figure 2.10 shows an example of an implementation of the logic required for implementing BIST in a circuit.

#### 2.3.4.2 Run-Time Reconfiguration

Partially reconfigurable devices provide the benefit of allowing access to their inner workings. With this advantage in hand, the injection of faults into a CUT is only a matter of modifying the complete or a part of a particular LUT with the test pattern. See figure 2.11 for an example.

## 2.4 Co-Design Environments

A co-design environment is a system that permits or enables the development of hardware and software based applications. These applications can be targeted to reconfigurable devices or ASICs. Co-design environments do not share a particular architecture, but their intent is the same: to produce hardware and software components that work together to form a single application. The typical flow for a co-design environment is shown in figure 2.12. A non-exhaustive list of co-design environments is shown below.

**Handel-C** A subset of the C programming language designed for compiling programs into hardware specifications for FPGAs which include software components.

**Single-assigment C (SA-C)** Another variant of C that can be directly mapped onto circuits, including FPGAs.

**SystemC** A C++ class library that provides constructs of objects that model system architectures, concurrency, and reactive behaviours.

**ImpulseC (Streams-C)** Permits the description of computational processes, their connections, and their parallel implementation on FPGAs and $\mu$Ps/DSPs.

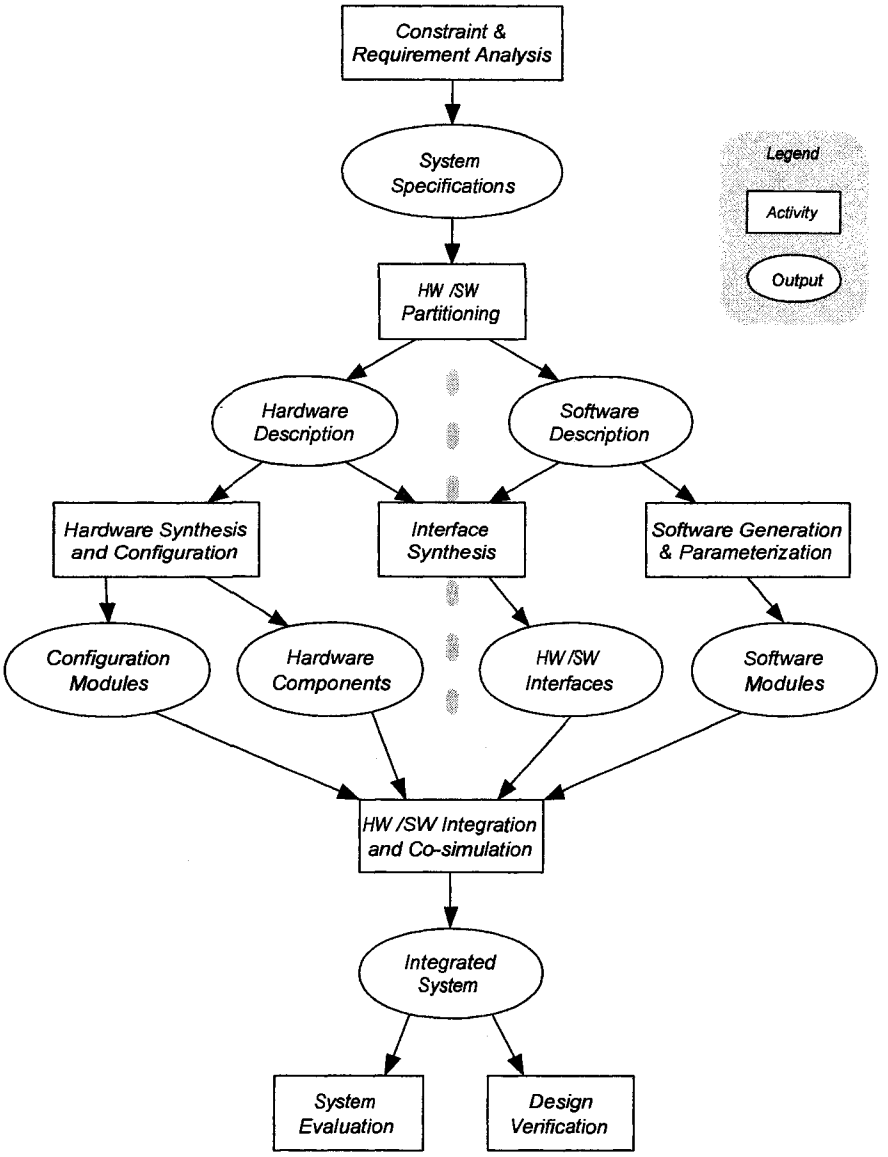



**Figure 2.12:** *Depiction of the generic flow for a co-design environment [23].*

## 2.5 JBits

JBits [24] is a collection of Java classes that provide an *application program interface* (API) which allows the directly interfacing with the Xilinx Virtex FPGA family and handling their bitstreams. This API provides low-level access to the FPGAs which allows

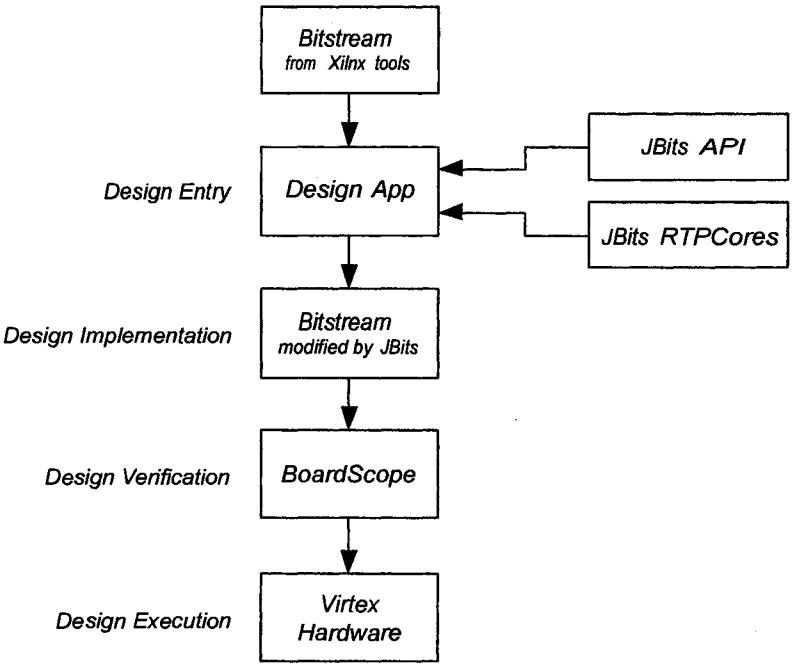



**Figure 2.13:** *JBits Static Design Flow ([25]).*

the manual placement of circuits, routing, and other functions related to the FPGA's fabric. JBits is one of the cornerstones for this project. JBits allows the inclusion of Xilinx technology through the published API into our own applications.

JBits is able to operate with bitstreams created with Xilinx tools as well as bitstreams obtained by reading back the FPGA configuration. This allows any application that is using JBits to have complete control of the device. JBits provides access to all resources such as look-up tables (LUT), configurable logic blocks (CLB), and the routing resources of the device. The normal use of JBits is seen in figure 2.13 which shows the design flow for a normal static application that does not support run-time reconfigurability. With this flow, the designer will create bitstreams using Xilinx tools. These bitstreams are then manipulated by the designer's own application thanks to the JBits API.

The power of JBits can be seen better when used in a RTR setting. JBits allows

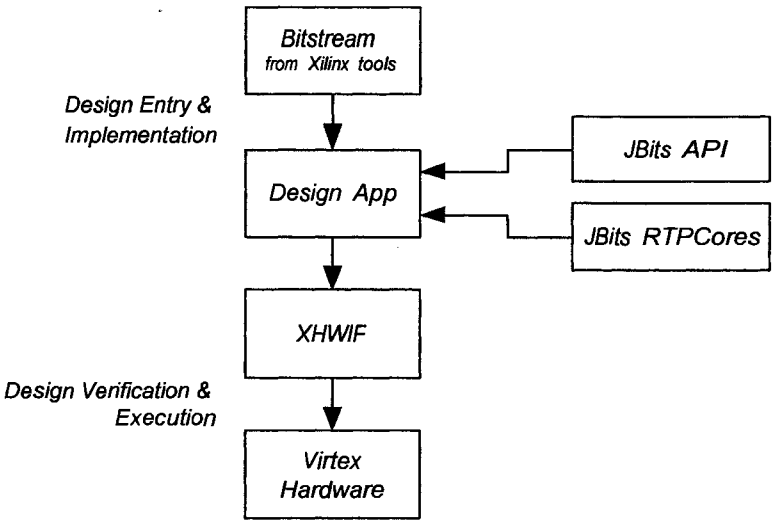



**Figure 2.14:** *JBits RTR Design Flow ([25]).*

designers to manipulate and change bitstreams before and during the their use in a Xilinx' FPGA. Thanks to this capability, JBits makes possible RTR research. This is shown in figure 2.14 which depicts the design flow for an RTR design using JBits. The diagram is similar to the one shown in figure 2.13 with the main difference of the new block entitled XHWIF which stands for "*Xilinx Hardware InterFace.*" This is as a programing interface used for programming Xilinx FPGAs from a Java application.

While both flows show the different steps, both require a hardware platform (*i.e.*, FPGAs). The hardware can be replaced with the use of BoardScope, an application that is shipped along side JBits and JBits' documentation in a complete package named "JBits SDK." This setup allows the development of applications without an actual hardware platform.

# Chapter 3

# Problem

Modern computer systems are more complex than ever and the effort required to make them a reality does therefore increase. For the past few decades computer systems have continuously crammed more circuits in the same space. This has led to the implantation as a "law" in the IC development community of an empirical observation made by Gordon E. Moore in 1965 which appeared in [26]. This was the birth of "Moore's law" which is interpreted by some to indicate a doubling in capacity every 18 to 20 months. Schaller [27] provides a well done historical overview, which span over several decades, of Moore's law.

The exponential increase in IC complexity has, as one may expect, been the result of more aggressive design methods which are driven by the consumer demand. As we can imagine, this requires more designers which are versed in IC design. And, as we have seen before in section 2.1.3, reconfigurable systems are systems which provide the almost the same functions as ICs but with the added benefit of being reconfigurable. This has the effect of sending many designers to the reconfigurable arena with the know how of ASIC design. This implies that the tools and methods used to develop reconfigurable systems are in the most part similar to those used in the development of ASICs.

Furthermore, the demand for more all-in one type of devices is increasing. This forces manufactures into searching for a way on integrating different functions into one single device. This factor and those pointed out in the previous paragraphs have led to a unique situation where the current design infrastructure can not scale appropriately given the demand for electronic goods.

## 3.1 The SPLASH Effect

The factors described above have been identified as the *SPLASH* effect by Groza *et al.* [28]. The SPLASH effect is a combination of the following factors:

- a **S**hism between the traditional ASIC tools and their required capacities,
- "a **P**lethora of gates and not enough designers to use them,"
- "a **L**ack of a techonology to which Moore's Law can be extrapolated to,"
- "an **A**ugmentation of the cost of fabrication plants,"
- "a **S**eparation between the digital producer and the digital consumer, and"
- "a **H**indrance of the optical lithographical process caused by physical limitations": "Physics could signal the end of Moore's Law."

Also a diagram is shown in figure 3.1 which depicts the challenge of the electronic and semiconductor industries in order to transform the ideas into final products. This figure is taken from [28] and it is entitled: The "loudspeaker bottleneck."

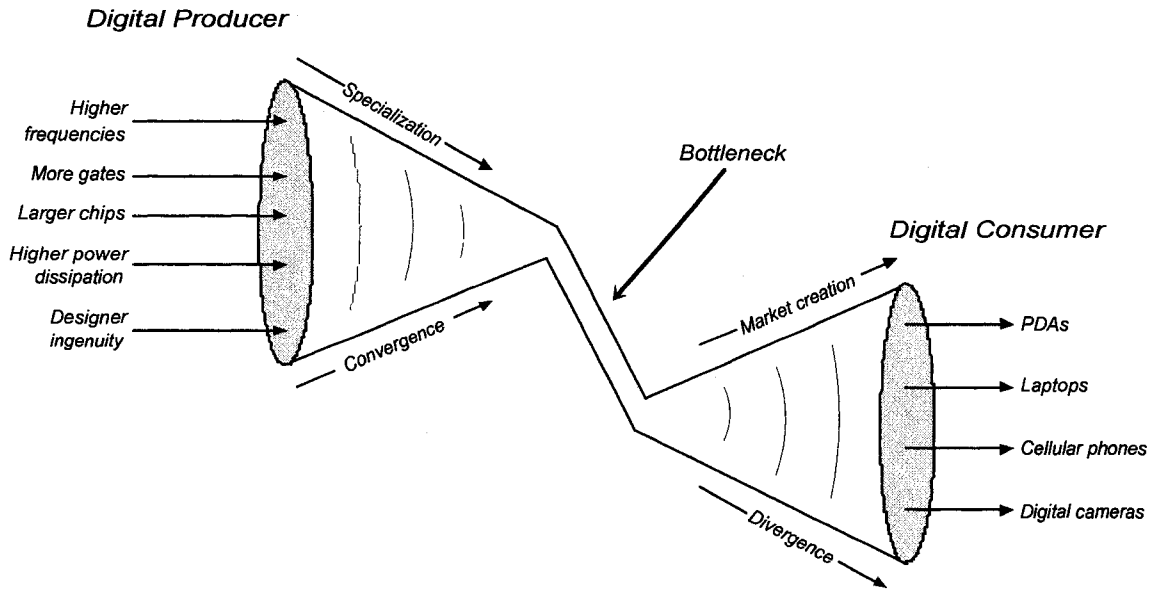



**Figure 3.1:** *This diagram depicts the digital producer output and digital consumer input. The bottleneck shown represents the different challenges that the industry must overcome in order to form a design into a finished product. This figure is extracted from [28] and entitled "The loudspeaker bottleneck".*

### 3.1.1 Separation of Fundamentals

The first item in the list factors as found by Groza et al. in [28] is "schism" and this indicates a separation between what the tools and process can deliver and the actual demands of the market. Current tools, methodologies, and process were perfected by the ASIC industry. With devices capabilities and their features growing at a fast pace, these tools and the corresponding design flows may not be up to the challenge given by the demand.

### 3.1.2 Abundance of Resources

This is referred as a "plethora" of logical gates in the new devices by the SPLASH factor. There are more and more gates for the same number of designers. This combined with

the "splash" issue seen above leads to an inefficient use of the different resources including logical gates. Hence, an inefficient use of these devices may be expected.

### 3.1.3 Fabrication Cost

The growth of computing in general is affected by the cost incurred in the creation and maintenance of the infrastructure needed to produce all the semiconductors. With the growth in demand of more powerful but also more diverse types of devices, the fabrication plants are called to work non-stop. While this may seem as a good thing, we must keep in mind of the massive investment necessary to put this type of facility in function.

### 3.1.4 Separation Between Producer and Consumer

An example of the separation between the digital producer and the digital consumer is the increase in power by the newer devices asked to manage and process the growing amount and complexity of data such as video, and the demand of mobile devices which while providing multimedia services these have very limited power supplies.

### 3.1.5 An end to Moore's Law?

Current semiconductor production process are hitting a limit on how small they can be produced. This is due to the physical limit of the lithographical process.

## 3.2 Escalating Chip Complexity

As the complexity in the submicron integrated circuit increases, it is very likely that the number of faults in them will also increase. Since most semiconductors are designed to

be used for extended periods of time in their intended environment, it is important to devise a method on how to test their reliability.

The design flow for all integrated circuits encompasses verification methods in every stage of the design. But even after the ICs have been synthesized, placed, and routed; it is important to test that the finished product is fault-free. BIST is a field that is heavily active in looking for new methods or algorithms that will speed the fault testing of ICs. But more importantly, it is looking for algorithms that allow to find the highest number of faults.

As we can expect, in order to test BIST methods we can use software-based circuit simulation that provide the required flexibility of not creating a physical circuit every time a new test is run. But this has the drawback of being relatively a slow process when compared to a comparable hardware-based test. BIST research will, therefore, gain from a method that could speed up the process without the drawbacks of synthesizing an IC every time the test changes. This can be appreciated better by understanding some of the BIST methods used. There are three test strategies that are of particular interest to us. These are:

**Sequential CTR** Sequential CTR (*i.e.*, Compile Time Reconfiguration) methods is a strategy where the CUT (*i.e.*, circuit under test) and compressors are synthesized and mapped to the target device, with fault-injection multiplexers (FIMs) included in the CUT.

**Partially-parallel CTR** Partially-parallel CTR method applies a particular fault-injection test to a different number of CUTs simultaneously. To become a fully-parallel strategy, the complete number of fault-injection test has to be applied.

**Sequential RTR** A sequential RTR strategy involves the use of the logic elements of a configurable device such as LUTs in an FPGA. FIMs are not required when

using this strategy since the fault-injections are synthesized directly into the logic elements.

These strategies are interesting since they can be compared against each other which allow us to evaluate the efficacy of the platform used to host the BIST's fault-injection tests.

## 3.3 A Designer Perspective

The problems highlighted in the SPLASH Effect section apply to the entire semiconductors industry and its derivatives. Some of those points were linked to the design process of systems and applications. This is of particular importance since the applications capabilities and features are hindered if designers are not working at an optimum speed and can not concentrate on the project. As we can imagine, if the designer must constantly address tool related issues, and consequently this one has less time available for actual design work.

It appears that the need for new set of tools that are not ASIC-centric is increasing. The use of such tools will allow designers to concentrate on the problem at hand and letting the tool handle the rest. This is highlighted in the article by Santarini [7] which discusses how ASIC designers have switched their designs into using FPGAs and how this affects their choice of tools.

## 3.4 Summary

The different elements identified by the SPLASH effect (*i.e.*, schism, plethora, lack, augmentation, separation, and hindrance) will produce a ever increasing bottleneck that limits the semiconductor industry on how well to respond to the demands of the digital

consumer. But more specifically, the increasing IC complexity requires faster and better test methods which are partially tackled by BIST methods.

There will always be challenges when developing technologies and products, but it seems that we are at a point where a new technology and methodology will be required to keep the momentum required to create devices that are more powerful than the preceding ones. All the different points highlighted in this chapter are presented in order to provide the background and rationale behind the development of a projects such as HOSes. While there is a myriad of reason why technologies, such as RTR systems, are necessary; HOSes tackles the need for finding better BIST test methodologies that permit to find IC's faults that provide higher degree of fault-coverage and ever increasing test speed as well.

# Chapter 4

# Hardware Operating Systems

The GEMS Research Lab has been working on RTR research and has produced several publications in this field [29]. One of the projects is the ***Hardware Operating Systems*** (HOSes) which is an RTE that is being developed in order to provide a new paradigm in hardware development (*i.e.*, co-design) targeting configurable devices such as FPGAs. The beginning of this project can be traced back to the paper [30] which presents HOSes and its architecture. It mentions the origins and ideas behind HOSes. There are no other system available that is similar to HOSes. But there are system that use RC technology for different applications. These are listed in section 2.1.3.3.

This project unifies the theories and presents the design and structure of HOSes which this author has contributed to. Before presenting the architecture and design features of HOSes, this chapter will address the different issues raised by the SPLASH factor presented in section 3.1 by providing possible solutions.

## 4.1 Reconfigurable Computing Response

The SPLASH effect, as shown in section 3.1, identifies several factors which are hindering the efficient development of computing technology that is demanded by modern digital consumers. Reconfigurable computing (RC) tries to overcome these challenges in different forms. Below is a list of how RC can affect the design and production methods.

**Getting Closer to the Market Demand** Current design methodologies and tools were devised to address the needs of ASIC development. Given the growth in power and complexity of such devices the designers are faced with what appears to be an antiquated form of design which can benefit from the advantages provided by reconfigurable architectures and environments. RTEs such as HOSes aim to bring tools and design methodologies that will allow the industry to design faster and more efficiently in order to address the market demands. But research is still being performed and it is premature to predict the effects that this research will provide.

**Efficient Technology Use** Given the increasing number of gates found in the latest reconfigurable chips and the development process that is ASIC centric we can expect inefficiencies to appear in how the resources are utilized. An RTE such as HOSes provides a different design paradigm to developers which could result in better use of those resources.

**Decreasing Cost by Standardizing** One of the benefits that can be predicted by the adoption of a RTE is that the designer will work with an abstract model; hence the hardware providers can supply a smaller number of customizations, reusing and standardizing on a smaller number of device models. By standardizing, fabrication plants should be able to save in production cost by concentrating in a particular technology.

**A New Road** Given the limits that are appearing in the fabrication process of semiconductors underlined in section 3.1, it is important to find new technologies that will allow the industry to overcome any challenges that may appear because of the limitations that are appearing in the production process. Research is not leaving any stone unturned in order to find new ways in producing more powerful devices. As an example, Todi [31] indicates that the industry is moving back to using *germanium* (Ge) in the production of semiconductors. Ge was replaced in the late 1960s and early 1970s by *silicon* (Si) mainly because of Ge's unfavourable surface properties and also because it has no stable natural oxide, unlike Si. Ge return is primarily due to the continued advances in production of semiconductors which are starting to use 22-nm technology. At this scale and below the use of high-mobility semiconductors is important; Ge has ≈2x the carrier mobility for electrons and 4x for holes when compared to Si.

## 4.2 HOSes Architecture

HOSes is a (RTE) (see section 2.1.3.3 for more information on RTEs) used to design, develop, and test hardware applications (HwApps) which are used in (RC) systems. RTEs such as HOSes aim to provide another option to the computer field which, as stated in chapter 3, may face serious challenges in the near future.

HOSes' modular model is shown in figure 4.1 which shows four layers, each representing a module which is independent of the others. The layers are: *application*, *architectural*, *bridging*, and *physical*. The following sections will discuss the different layers in depth. A small description of each is provide below.

**Application Layer (AppL)** Provides the user interface (*e.g.*, GUI) which the designer uses to create and execute the HwApps.

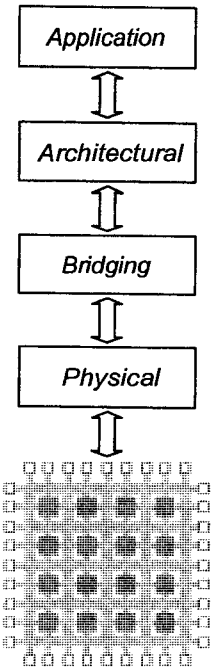



**Figure 4.1:** *HOSes is composed of four layers: application, architectural, bridging, and physical. There is an option to control an actual FPGA using the vendors supplied software.*

**Architectural Layer (ArcL)** Provides the engine for the development environment which uses the abstract model of the reconfigurable device (*e.g.*, FPGA).

**Bridging Layer (BL)** Provides the link between the abstract model of the RA and the actual hardware device being used. This implies the following:

- Translates of high-level commands sent by the ArcL into low-level instructions for the PL.
- Provides hardware-specific access.
- Manages the HwApp context information required for execution of more HwApp than the hardware permits.
- Provides an abstraction of the hardware to the above layers.

**Physical Layer (PL)** As it name implies, the PL manages all the hardware resources.

It allows the BL access to the different hardware platforms through a set of standard *application programming interfaces* (API).

This modular design allows the collaboration of several people which spanned over several years. It also allows the use of several programming languages such as Java or C++ which has the advantage of opening the door to programs and libraries such as JBits. Thanks to this architecture, HOSes development is flexible which allow us to tackle the different constraints listed below.

**Modifiability** Creation and changes done to HwApps should be done quickly and cost effectively.

**Portability** HOSes should be able to use different hardware platforms which implies that a HwApp is highly portable.

## 4.3 Application Layer

The *application layer* (AppL) is the interface available to the hardware application (HwApp) designer. While this can take the form of a *graphical user interface* (GUI), it can also take the form of simple files that contain all the instructions necessary to create a HwApp.

The AppL deals mostly, if not completely, with user (*i.e.*, designer) related tasks. The purpose for this layer, is to separate any user interface idiosyncrasies from affecting the internal structure of the system. Also, by concentrating all user interfaces in this layer, it is theoretically easier to port the system to different OS platforms such as Microsoft's Windows, Sun's Solaris, and/or GNU/Linux.

The AppL, as stated before, is in charge of providing an interface which allows the designers to use HOSes features. A prototype of a GUI that presents the development

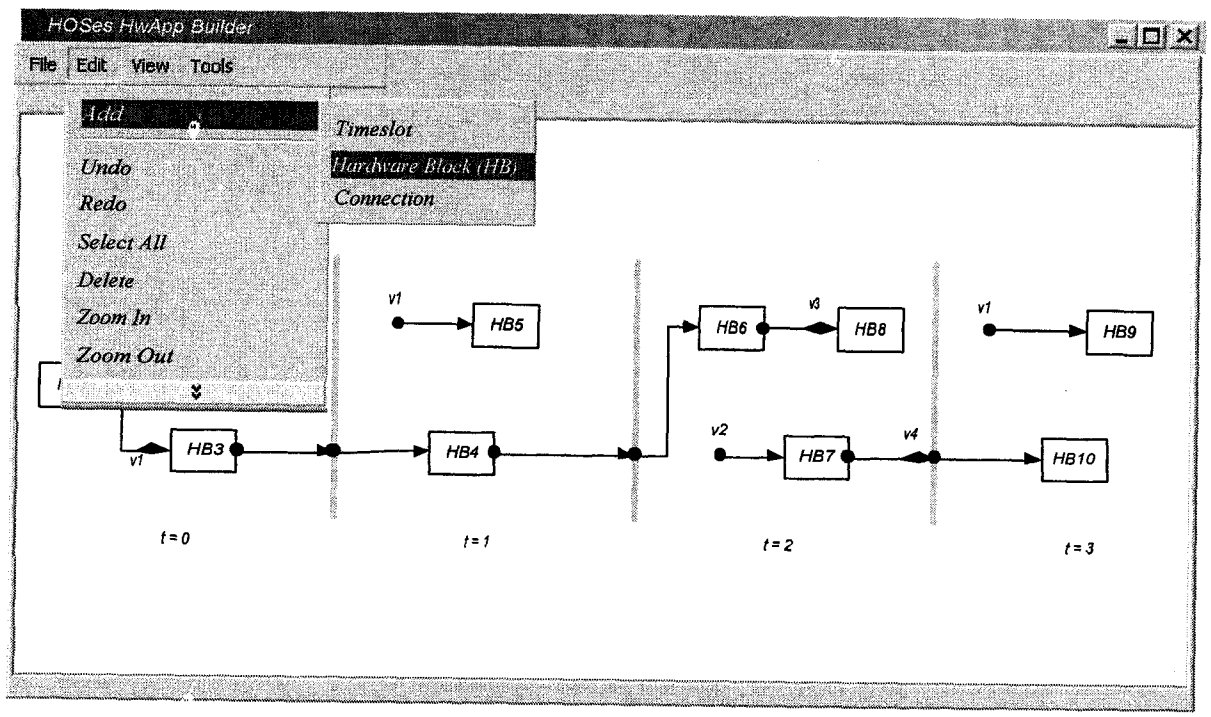



**Figure 4.2:** *Mockup of HOSes application layer (AppL) GUI used by designers to create hardware applications (HwApps).*

environment is shown in figure 4.2. This figure is meant to show the intent or goal that the GEMS Lab has regarding how HOSes will appear to the designer, since this is the main interface to the system.

The figure 4.2 shows a window which a designer will use in order to create a HwApp. This GUI will provide, to the designer, all the necessary functions required to create, test, and debug said HwApps. Hence, this GUI must access the services given by the other layers. The method by which the AppL will communicate with the architectural layer (ArcL) has not yet been implemented.

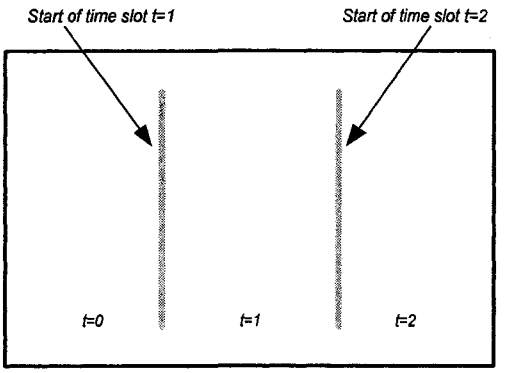



**Figure 4.3:** *This figure illustrates an empty environment that hosts the development of HwApps using AML.*

## 4.4 Architectural Layer

The *architectural layer* (ArcL) is in charge of managing and handling the high level (or more abstract) model of a given hardware application (HwApp). The goal of the ArcL is to provide an abstract representation of a HwApp which allows designers to create applications without knowing the internals of the platform.

Because the goal of HOSes is to provide an abstract model of an RC system, a method to model a HwApp must be designed. But this model should avoid as much as possible any platform-specific details for it to be friendly to most designers. One of the tools contributed in part by this author is called *Application Modelling Language* (AML). An earlier design can be traced back to [30].

When compared to other methods such as HDL or schematic capture, we can see that HOSes provides another means of hardware representation which does not require a high level of understanding of the hardware platform unlike other options.

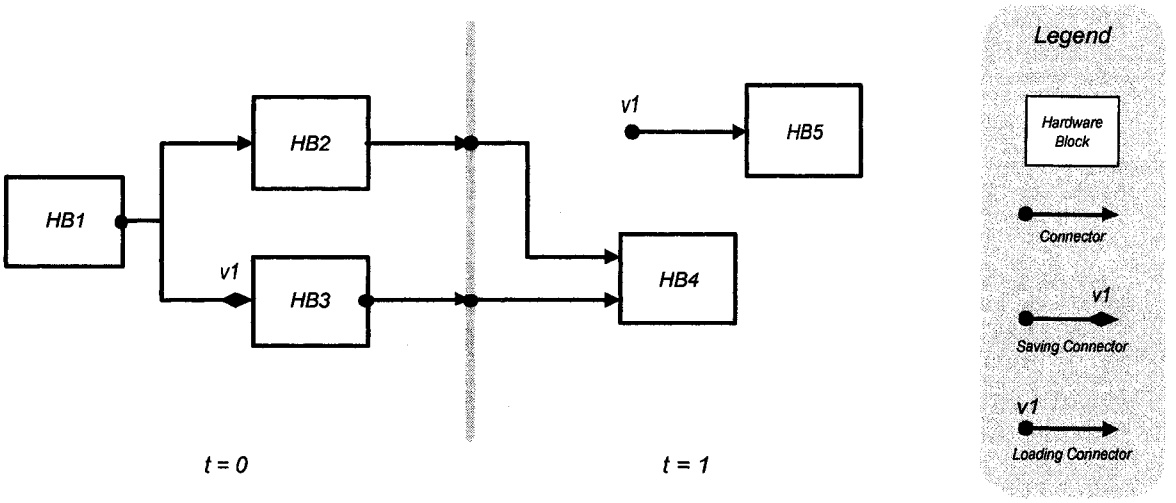



**Figure 4.4:** *Example of a simple AML hardware application (HwApp) composed of an environment, hardware blocks (HBs) and connectors.*

### 4.4.1 Application Modelling Language

HOSes' ArcL provides what is called the *Application Modelling Language* (AML). It consists of models of basic hardware blocks such as adders, multipliers, demultiplexers, etc. It also provides mechanisms to link these basic block between them. It also imposes rules on how the components are linked and operated in order to limit design errors but to also maintain compatibility with the actual hardware platform.

AML has three basic components: the environment, hardware blocks (HBs), and connectors which are used to connect HBs between them. The combination of these elements permits to represent complex systems. By connecting HBs together and processing each other inputs and output, more complex applications can be created. The environment is in charge of executing the formed HwApp accordingly to the rules set by the AML.

**Environment** The environment is in charge of managing the proper execution of the designed HwApp which uses HBs and connectors. The main characteristic of the environment is that it is divided in *timeslots* which are needed in order to synchronize the

execution of HBs. Because of the diversity found in the number of HBs, these behave differently from each other and their execution is not homogeneous. A given HB will execute its duties faster than another one which will result in a difference in time of the flow of information through the HwApp. Another reason for the delimitation of timeslots is to be able to stop execution of a given HwApp and save its context. This is done by waiting for the HwApp to execute all the HBs in a given timeslot and then save all the relevant information regarding the application. The system (*i.e.*, HOSes) can then replace the HwApp with another one, allowing several HwApps to run concurrently in a limited space, much like process in a system with virtual memory like most modern OS. A depiction of an environment without any HBs is shown in figure 4.3.

**Hardware Block** 

AML's hardware block (HB) is a basic element of the language that represents basic hardware components such as adders and multipliers. Although there is no limit as to what a HB can be or contain, they should be composed of the least elements possible. The HwApp designer should be able to create complex systems using these building blocks. Data is sent to a HB by making a connection to one of its inputs. The output data is then extracted by examining its output port and it to another HB's input. As we can see, a designer will treat a HB as a black box and will only manage its inputs and outputs. A more detailed description of the HBs architecture is presented in section 4.6.4.

**Connectors** 

Data flow is managed by connectors which link different HBs together. Connectors have other functions as well such as storing the output of a HB into a HwApp global variable. This global variable will store the supplied data during the entire life of the HwApp or until is overwritten with different data. This allows to retrieve the saved data by other HBs located in a later timeslot

without explicitly making a connection. The retrieval of the saved data is also done by a connector which supplies it to a given HB.

**Normal connector** •⟶ This connector is used to link the output of one HB with the input of another one. This is the simplest way of transferring data between HBs.

**Saving connector** •—v1→ The save connector is used to store the HB generated data in a global variable which in this example is named *v1*. This connector does not need to connect to the input of a HB since the data are stored elsewhere. Although the connector must be connected to the output of a HB.

**Loading connector** —v1→• This connectors is used to retrieve data from a global variable (*e.g.*, *v1*) and to feed it to a given HB. It is not possible to connect two HBs with this connector since the stored data will be overwritten with the HB's output.

### 4.4.2 Hardware Application

Before tackling the design and structure of HOSes, an overview of a *hardware application* (HwApp) is required since this is the main goal of the designers which we are trying to address. As mentioned before, a HwApp is a collection of hardware and software elements that, when executed, provides an output or service when given an input. Ideally, an RC system would load HwApp as needed the same way programs in a PC are loaded.

Figure 4.4 depicts a HwApp example which shows all the different elements of the AML since all HwApps in HOSes are designed with AML. The environment is composed of two timeslots which throttle the execution of the HwApp. This particular example is composed of 5 HBs which are connected using a mix of connectors. Allow us to study

this figure so that we can see the intended behaviour of the HwApp. The first HB is named HB1 and its input is managed by the interface which is not shown here. Just as the input of the first HB is not shown, the final outputs are also not shown (*i.e.*, HB4 and HB5 outputs) but nonetheless present.

When the execution of the HwApp starts, HOSes allows the hardware blocks execute at their own speed and waits until the last HB in the timeslots returns its output. Continuing with the example in figure 4.4, the output of HB1 is sent to two HBs (*i.e.*, HB2 and HB3). This output is saved to the global variable *v1* by the connector that links HB1 and HB3. Both HB2 and HB3 have a single output and this one is sent to the timeslot boundary. Internally, HOSes will grab a snapshot of the data and process it by sending it between its layers. This is done, in part, so that all the layers are synchronized including the user interface which will provide feedback to the user. The other reason is to provide a mechanism by which HOSes can grab a snapshot of the HwApp context which will allow it to replace a given HwApp by another HwApp that is waiting to execute. This provides the benefit of "swapping in and out" HwApps which in turns permits the development of HwApps that are virtually bigger than the available hardware.

Timing is an important factor when designing hardware systems and designing for HOSes is not different, hence the need for timeslots. The reason why timing is important is because different HBs do not function equally and some will take more time to execute. This may lead to imbalances regarding when the data are fed to the next HB in the HwApp. This may not be a problem in a serial HwApp where there are no multiple HB instances executing simultaneously. But more complex applications need mechanisms to ensure a correct execution. Timeslots aim to provide a secure mechanism to synchronize all HBs and a way to stabilize the HwApp.

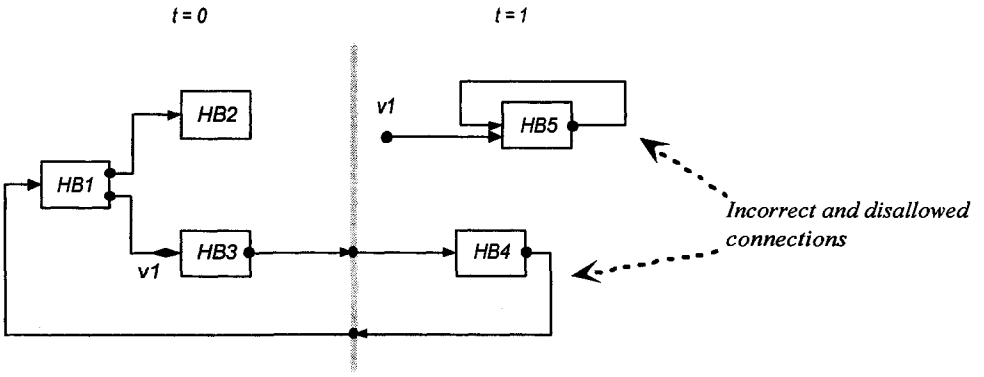



**Figure 4.5:** *Invalid connections which are not allowed in the design of a hardware application (HwApp). The inputs of some hardware blocks depend on the future output of the same or other blocks.*

Going back to the example in figure 4.4, the hardware blocks HB2 and HB3 send their output to the timeslot boundary. This permits HOSes to grab a snapshot of the system and update all internal variables. Once the internal execution is performed, the HwApp resumes execution by sending the data to any connectors attached to the boundary of *loading connectors* found in the executing timeslot. In this example, HB4 receives two inputs, one from HB2 and the other from HB3. HB5 receives its inputs from the global variable *v1* thanks to the loading connector. HB4 and HB5 then perform their computations, and their output is stored by HOSes and sent to the appropriate layers. This ends the overview of the example shown in figure 4.4. A more in-depth view of the internal working of HOSes while executing a HwApp is presented in section 4.5.4.

There are some instances where some connections are not allowed. These instances involve loops. An example of these connections is shown in figure 4.5. The first invalid connection links the output of HB5 with its own input. This is clearly invalid since this would require to know the output before having an input, which is not possible. The same applies to the other connection that links HB4 back to HB1. While the same applies to this connection where the future output of HB4 cannot be known before providing an

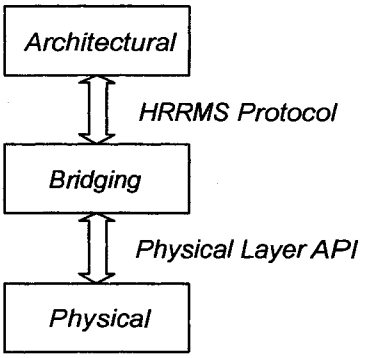



**Figure 4.6:** *The architectural layer (ArcL) communicates with the bridging layer (BL) using HRRMS protocol. The BL then controls the physical layer (PL) using the available API.*

input to HB1; the transition from timeslot 1 to timeslot 0 is also not possible.

## 4.5 Bridging Layer

The *bridging layer* (BL) provides the necessary bridge mechanisms between the physical (PL) and architectural (ArcL) layers while hiding the specific details of either one. The BL is in charge of calling (*i.e.*, executing) the high level commands in the PL by using the provided PL's API. In other words, the BL instructs the PL what to do depending on what the ArcL is requesting, hence, the BL plays also the role of a manager for the nuts and bolts of the systems. This is depicted in figure 4.6. The following sections will present the architecture and design created by this author.

### 4.5.1 Inter-layer Communication

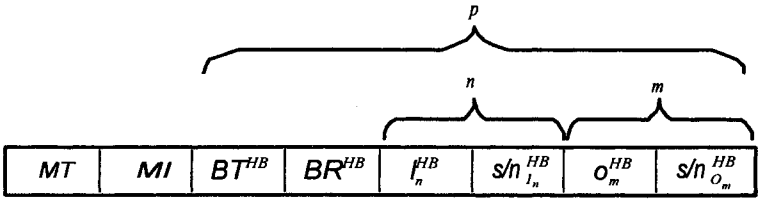
The bridging layer is in charge of correctly passing information from the architectural layer to the physical layer and vice versa. This implies different things such as translating information from one layer format to another which includes gathering the status of the physical layer and sending it in a friendly format to the application layer or GUI for

example. The bridging layer has several tasks to perform which are necessary for the correct functioning of HOSes. These are seen below.

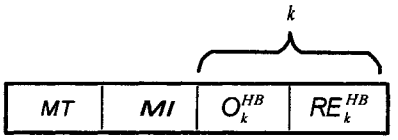
**Provides a Bridge Between Layers** As it name indicates, the bridging layer is in charge of providing a bridge between the architectural and physical layers. This is done through a well defined protocol in the case of the architectural layer. This protocol is named HRRMS and is seen in more detail in section 4.5.2. For the physical layer, the bridging layer uses its knowledge of the hardware platform in order to control effectively the physical layer by using the actual components of the FPGA such as CLBs.

**Translates Hardware Specifications** One of the main goals of HOSes is to provide an abstraction of the hardware devices in order to give freedom to the designers. As seen before, the architectural layer deals with agnostic device HBs which means that this layer ignores the device that HOSes is dealing with if any (since one of the features of JBits is to emulate a FPGA, which is limited to a few models). On the other hand the physical layer does need to know which device HOSes is working with in order to correctly use the JBits API. This is where the bridging layer comes to help by correctly translating the agnostic data produced by the architectural layer into a format that the physical layer understands. As it can be seen, this is one of the most important aspects of HOSes or for that matter, of any RTE.

**Synchronizes Execution of Hardware Blocks** HOSes has the ability to work with live FPGA devices that are supported by JBits. This imposes the task of controlling and synchronizing the execution of HwApps and its corresponding HBs with the development environment. This task is performed by the bridging layer since it is




(a) Structure of a request message.




(b) Structure of a response message.

**Figure 4.7:** *Structures for the HRRMS request and response messages.*

the only layer that can translate device specific data into architectural data which requires non-specific device data.

### 4.5.2 Communication Protocol

The bridging layer communicates with the architectural layer through a well defined protocol named *"hardware request and response message structure"* (HRRMS). The purpose of providing this protocol is to define a method that allows the abstraction of the hardware platform in a way that can be manipulated by the architectural layer. The structure of a HRRMS differs depending on the type of message being transmitted. There are 8 different fields possible. These are:

**Message Type (MT)** Indicates the type of the message. The types are: request, response, garbage collection, cannot deploy, and error.

**Message Index (MI)** Indicates the message position in a sequence. Used when assembling out-of-order messages.

**Block Type ( BT $^{HB}$ )** Denotes the type of the HB (*e.g.*, 8-bit adder, 4-bit multiplier, demultiplexer, etc).

**Block Reference ( BR $^{HB}$ )** Indicates a given HB in order to distinguish it from other HBs of the same BT within the same timeslot.

**Input ( I $^{HB}_{n}$ )** Input reference of a given HB.

**Output ( O $^{HB}_{m}$ )** Output reference for a given HB.

**Save/No-save ( s/n $^{HB}_{n}$ or s/n $^{HB}_{m}$ )** Used to indicate whether the input was saved or if the output should be saved.

**Output ( O $^{HB}_{k}$ )** Field with the output connection label in a response message.

**Reference ( RE $^{HB}_{I/O}$ )** Returned reference in an response message in response to the request message which indicated a "save" action in the s/n $^{HB}_{n}$ or s/n $^{HB}_{m}$ fields.

The structure of a HRRMS is shown in figure 4.7(a). HRRMS messages can carry information about multiple HB definitions. Also, each HB definition carries a dynamic number of input and outputs, and hence connectors. This is shown in the figure by the variables $p$, $n$, and $m$. $p$ indicates the number of HB definitions that are present in the message. I $^{HB}_{n}$ and O $^{HB}_{m}$ indicate the number of inputs and outputs respectively. A *save* ( s/n $^{HB}_{l}$ ) request message will instruct the bridging layer to answer with a reference to the saved data in the response message. The response message structure is shown in figure 4.7(b). This type of message is also dynamic since it cosnsist of a $m$ number of I/O $^{HB}_{k}$ & RE $^{HB}_{k}$ field pairs, one for each save request. Note that there will always be one response message for each request message in each timeslot.

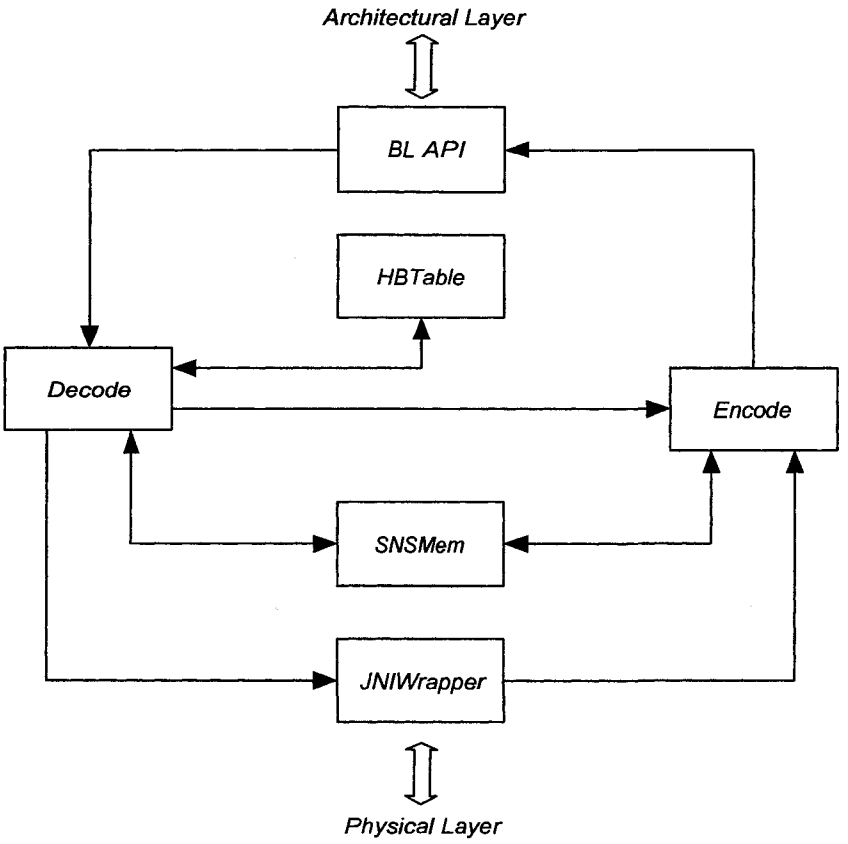



**Figure 4.8:** *HOSes: Bridging layer block diagram*

### 4.5.3 Internal Structure

The BL role is to correctly translate the high-level commands emanating from the ArcL into precise low level commands given to the PL. Therefore the BL is an essential part of HOSes since it is the part that manages low level behaviour given the high-level commands received in the form of HRRMS messages.

The block diagram for the bridging layer is shown in figure 4.8. This diagram shows the internal structure of the bridging layer. At the top we find the block entitled BL API. This block publishes different functions and classes to the ArcL so that it can communicate with the BL. The HRRMS messages are sent through this block. The

Decode block is then commanded by the BL API block to decode the received messages from the ArcL. Decode will then perform the necessary actions to accomplish the request made by the ArcL. As seen in the figure, the Decode block communicates with four other blocks since Decode must manage all possible actions that may appear in the incoming HRRMS messages. The block entitled HBTable keeps a table of all the HBs and their actual bitstreams which are loaded into the FPGA. Decode selects the appropriate bitstream from the information found in the received message. The SNSMem block keeps track of the global and temporary variables created when executing a given HwApp. JNIWrapper block is in charge of accessing the PL's API from the commands received from the Decode block. This results in low-level commands such as placing the HB bitstream in a particular location in the FPGA. The Encode block processes the information in the BL and translates it into a format to be used with the HRRMS protocol which is then sent to the BL API block.

### 4.5.4 A Hardware Application and the HRRMS Protocol

In order to comprehend the architecture of the BL we must study how a HwApp is decomposed using the HRRMS protocol. This will allow us to appreciate the architecture of the BL. It is important to understand this breakdown because the actual behaviour of the BL is dependent on the message packet contents and the relative position of this message in a given communication stream between the ArcL and PL. We will deconstruct the HwApp shown in figure 4.9 in its corresponding messages and how these are handled by the architectural and bridging layers.

The following lists the different elements, shown and hidden, in the HwApp depicted in figure 4.9. This is presented before studying the HRRMS messages that the ArcL would send to the BL regarding this HwApp which will be shown later. In this figure

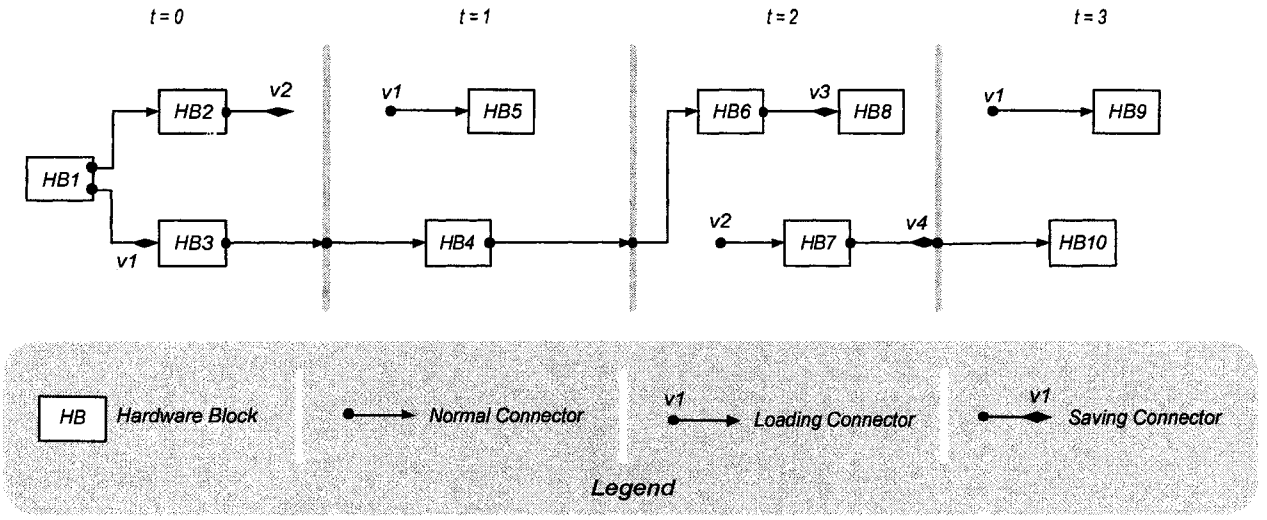



**Figure 4.9:** *Example of a hardware application developed with HOSes.*

(4.9) we can identify the following elements:

- *t = 0* 4 timeslots.
- HB 10 hardware blocks.
- v1 4 global variables.
- 3 temporary (timeslot) variables.
- 1 normal connector.

The above list shows the different elements that will allow us to construct the different HRRMS messages that the architectural layer will send to the bridging layer.

**Number of messages** HRRMS defines that each timeslot will require its own request (type MT=0) and response (type MT=1) message. The request message is sent by the ArcL to the BL. Once the system has performed the actions requested by said message, the BL will send a response message back to the ArcL with the results of the requested

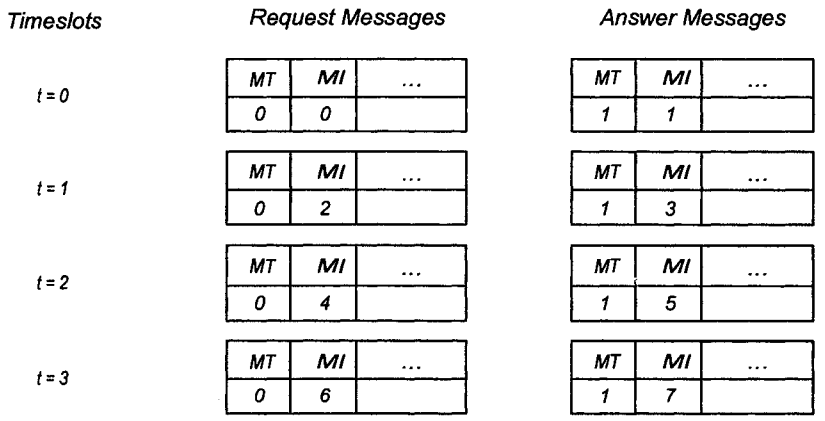



**Figure 4.10:** *HRRMS messages for a normal transaction between the architectural and bridging layers for the hypothetical hardware application shown in figure 4.9.*

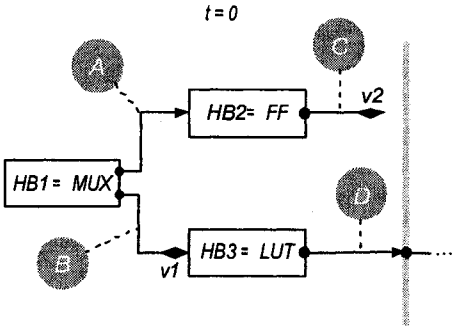
operations. We then obtain 4 messages that the ArcL will send to the BL and each will require a response message back from BL to the ArcL. This is shown in figure 4.10 where each timeslot has a pair of messages (*i.e.*, request and response).

**First Act** The first request message can now be built since we know that it requires 4 of these messages. Following the structure for the request message shown in figure 4.7(a) we can see that we need to define the different HBs which are represented by the collection of fields enclosed by the variable $p$. By studying the first timeslot in the example shown in figure 4.9 we can identify three HBs: HB1, HB2 and HB3. This translates into $p = 3$. Furthermore, let us assume that there are three different types of HBs available: demultiplexer (MUX), flip-flop (FF), and look-up tables (LUT). We will set HB1 = MUX, HB2 = FF, and HB3 = LUT. The resulting request message structure is shown in figure 4.11. Note that all the BR fields have a reference 0 since there is only one instance of each type of HB in the timeslot.

The request message for timeslot 1 is taking form as seen in figure 4.11. The three different HBs now have been defined with their respective types. The inputs and outputs

| MT | MI | $BT^{HB1}$ | $BR^{HB1}$ | ... | $BT^{HB2}$ | $BR^{HB2}$ | ... | $BT^{HB3}$ | $BR^{HB3}$ | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | MUX | 0 | ... | FF | 0 | ... | LUT | 0 | ... |

**Figure 4.11:** *Building a request message: Partial construct of a request message with three different HBs (MUX, FF, and LUT). This message represents the request made for timeslot t = 0 in figure 4.9.*




(a) Timeslot t = 0 breakdown

| | | HB1 | | | | | | HB2 | | | | | | HB3 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MT | MI | $BT^{HB1}$ | $BR^{HB1}$ | $O_0^{HB1}$ | $s/n_{O_0}^{HB1}$ | $O_1^{HB1}$ | $s/n_{O_1}^{HB1}$ | $BT^{HB2}$ | $BR^{HB2}$ | $I_0^{HB2}$ | $s/n_{I_0}^{HB2}$ | $O_0^{HB2}$ | $s/n_{O_0}^{HB2}$ | $BT^{HB3}$ | $BR^{HB3}$ | $I_0^{HB3}$ | $s/n_{I_0}^{HB3}$ | $O_0^{HB3}$ | $s/n_{O_0}^{HB3}$ |
| 0 | 0 | MUX | 0 | A | - | B | v1 | FF | 0 | A | - | C | v2 | LUT | 0 | B | v1 | D | - |

(b) Request message for timeslot t = 0

**Figure 4.12:** *Diagram 4.12(a) presents the different HBs and connector labels used for timeslot t = 0 for the HwApp in 4.9. Figure 4.12(b) shows a depiction of the request message that would be generated by the ArcL after processing the same timeslot.*

must now be defined. This is performed by labelling the corresponding connectors. For our example, we will label the connector from HB1 to HB2 "A." The connector from HB1 to HB3 "B." And the connector from HB3 to the timeslot boundary "C." The diagram in figure 4.12(a) depicts the chosen labelling.

The resulting request message for timeslot t = 0 is shown in figure 4.12(b). This message contains all the information necessary for the BL to execute this timeslot. Below is an in depth description of each field in the message depicted in figure 4.12(b).

**MT** = **0** Message type ("request") for timeslot t = 0 in HwApp in figure 4.9.

**MI** = **0** Message index in the data stream transmitted between ArcL and BL.

**BT** $^{\mathbf{HB1}}$ = **MUX** HB1's type (demultiplexer).

**BR** $^{\mathbf{HB1}}$ = **0** Unique reference in the message to the instance of type MUX which represents HB1. In this case it is 0 since it is the first and only block of type MUX.

**O** $_{0}^{\mathbf{HB1}}$ = **A** First output for HB1. This field uses the label chosen to represent the connector that links this output to the rest of the system.

**s/n** $_{\mathbf{O}\ 0}^{\mathbf{HB1}}$ = **-** Saving flag for HB1's first output. A dash ("-") indicates that this variable is not to be saved.

**O** $_{1}^{\mathbf{HB1}}$ = **B** Connector label linked to second HB2's output.

**s/n** $_{\mathbf{O}\ 1}^{\mathbf{HB1}}$ = *v1* Second HB1 output is to be saved in global variable *v1*.

**BT** $^{\mathbf{HB2}}$ = **FF** HB2's type (flip-flop).

**BR** $^{\mathbf{HB2}}$ = **0** HB2 unique instance reference. Set to 0 since there is only one HB of type FF.

**I** $_{0}^{\mathbf{HB2}}$ = **A** Connector label linked to HB2's input.

**s/n** $_{\mathbf{I}\ 0}^{\mathbf{HB2}}$ = **-** The connector linked to the **I** $_{0}^{\mathbf{HB2}}$ input connects this one directly to HB1's output without accessing global or temporary variables, hence the empty field.

**O** $_{0}^{\mathbf{HB2}}$ = **C** Connector label linked to HB2's output.

**s/n** $_{\mathbf{O}\ 0}^{\mathbf{HB2}}$ = *v2* HB2's output is to be saved in global variable *v2*.

**BT** $^{\mathbf{HB3}}$ = **LUT** HB3's type (look-up table).

**BR** $^{\mathbf{HB3}}$ = **0** HB3 unique instance reference which is set to 0 since HB3 is the only instance of type LUT.

**I** $_{0}^{\mathbf{HB3}}$ = **B** Connector label linked to HB3's input.

**s/n** $_{\mathbf{I}\ 0}^{\mathbf{HB3}}$ = ***v1*** HB3's input is being feed data that has been saved in the global variable *v1*.

**O** $_{0}^{\mathbf{HB3}}$ = **D** Connector label linked to HB3's output.

**s/n** $_{\mathbf{O}\ 0}^{\mathbf{HB3}}$ = **-** HB3's output is not to be saved, hence the empty field.

The above is the exhaustive description of every field that is sent by the ArcL to the BL in one single request message. The BL will then decode this message and act upon it. After processing and executing all the HBs, the execution will be paused at the end of the timeslot. At this point the BL will respond back to the ArcL with a response message that will contain some data that are relevant for the context. The generated response message is depicted in figure 4.13.

| MT | MI | $O_1^{HB1}$ | $RE_1^{HB1}$ | $O_0^{HB2}$ | $RE_0^{HB2}$ | $O_0^{HB3}$ | $RE_0^{HB3}$ |
|---|---|---|---|---|---|---|---|
| 1 | 1 | B | gv1 | C | gv2 | D | tv1 |

**Figure 4.13:** *HRRMS response message generated by BL after receiving the request message depicted in figure 4.12(b).*

The response shown in figure 4.13 shows how the BL answers a request message. The type is set to "response" (MT = 1) and the index to 1 because it is transmitted after the request which had the index of 0. Note the three pairs: O $_{1}^{\mathrm{HB1}}$ & RE $_{1}^{\mathrm{HB1}}$ , O $_{0}^{\mathrm{HB2}}$ & RE $_{0}^{\mathrm{HB2}}$ , and O $_{0}^{\mathrm{HB3}}$ & RE $_{0}^{\mathrm{HB3}}$ . The field are presented in depth below:

**MT = 1** Message type ("response") for timeslot t = 0 in HwApp in figure 4.9.

| Global Variables | | Temporary Variables | | |
|---|---|---|---|---|
| Temp. Ref | Real Ref. | Temp. Ref. | Real Ref. | Del. Timeslot |
| *v1* | *gv1* | D | *tv1* | 2 |
| *v2* | *gv2* | | | |

**Table 4.1:** *Contents of the SNSMem variable table after receiving the first request message from ArcL (timeslot t = 0).*

**MI = 1** Message index in the data stream transmitted between ArcL and BL.

**O** $_{1}^{\text{HB1}}$ **= B** Label of the connector which data were saved in a global variable. See next field.

**RE** $_{1}^{\text{HB1}}$ **= *gv1*** Reference to the global variable with data obtained by connector B.

**O** $_{0}^{\text{HB2}}$ **= C** Label of the connector which data were saved in a global variable. See next field.

**RE** $_{0}^{\text{HB2}}$ **= *gv2*** Reference to the global variable with data obtained by connector C.

**O** $_{0}^{\text{HB3}}$ **= D** Label of the connector which data were saved in a temporary variable because the connector was linked to the timeslot boundary. See next field.

**RE** $_{0}^{\text{HB3}}$ **= *tv1*** Reference to the variable that stores the HB3's output linked to the timeslot boundary. This is necessary since HOSes keeps a snapshot of the context every time that there is a timeslot change. The variable is named *tv1* since this variable is temporary and will be ready for deletion as soon as the timeslot t = 1 has finished execution. It will not be deleted automatically. This decision is left to the ArcL which has access to *garbage collection* messages for this purpose.

The table 4.1 shows the internal representation of the variables that the BL must manage for the HwApp shown in figure 4.9 after receiving the first request message. The

table shows two global variables and one temporary variable. The two global variables were created after the explicit request of the ArcL which provided its own reference. The BL creates its own reference in order to differentiate them with other HwApp variables. The temporary variable *tv1* is created since the timeslot t = 0 has finished executing and it is possible that HOSes could remove the HwApp in order to run another one. By creating *tv1* HOSes has in fact stored the context of the HwApp which can now be safely removed and reintroduced later. HOSes will then load the second timeslot HBs and their connections and resume execution thanks to the saved context. The variable *tv1* will be then used and available for deletion only until timeslot t = 1 has finished execution (*i.e.*, t = 2). This mechanism allows several HwApps to be executed concurrently in a limited space, much like applications in a virtual memory based system.

**Second Act** After the ArcL receives the response message from the BL with the result data, this one will update itself and transmit the new information upwards to the AppL in order to update the user of the execution status. After performing all necessary computations, ArcL will send the second request message stating the HBs to be used, the connections between them, and any variables from the SNSMem to be used.

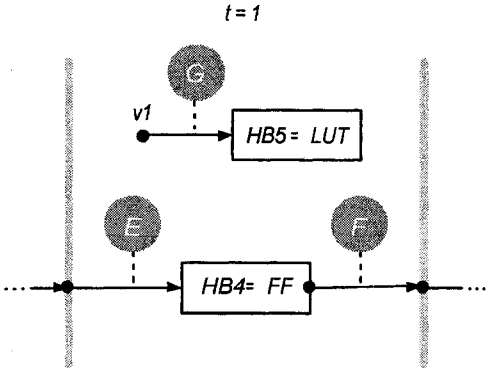
The figure 4.14(a) shows the HBs types and the connection labels for timeslot t = 1. The other figure, 4.14(b), shows the request message sent by the ArcL to the BL with the instruction regarding this timeslot.

**MT = 0** Second message type ("request") for timeslot t = 1 in HwApp in figure 4.9.

**MI = 2** Message index in the data stream transmitted between ArcL and BL.

**BT $^{HB4}$ = FF** HB4's type (flip-flow).

**BR $^{HB4}$ = 0** HB4's unique instance reference in the message.0 since it is the first instance of type "FF."




(a) Timeslot t = 1 breakdown

| | | HB4 | | | | | | HB5 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MT | MI | $BT^{HB4}$ | $BR^{HB4}$ | $I_0^{HB4}$ | $s/n\ I_{I_0}^{HB4}$ | $O_0^{HB4}$ | $s/n\ I_{O_0}^{HB4}$ | $BT^{HB5}$ | $BR^{HB5}$ | $I_0^{HB5}$ | $s/n\ I_{I_0}^{HB5}$ |
| 0 | 2 | FF | 0 | E | tv1 | F | - | LUT | 0 | G | gv1 |

(b) Request message for timeslot t = 1

**Figure 4.14:** *Figure 4.14(a) shows the breakdown for timeslot t = 1 for the HwApp in figure 4.9. The figure below, 4.14(b), shows the request message generated by the ArcL and sent to the BL.*

$\mathbf{I}_{0}^{HB4}$ = **E** Connection linked to HB4's only input.

$\mathbf{s/n}\ I_{0}^{HB4}$ = ***tv1*** Data flowing through the connection has been previously saved in temporary variable *tv1*. The BL did send this reference to the ArcL in the response message following the first request message.

$\mathbf{O}_{0}^{HB4}$ = **F** Connection linked to HB4's only output.

$\mathbf{s/n}\ O_{0}^{HB4}$ = **-** The output data from HB4 is not saved, hence the empty field.

$\mathbf{BT}^{HB5}$ = **LUT** HB5's type (look-up table).

$\mathbf{BR}^{HB5}$ = **0** HB5's unique instance reference in the message.

$\mathbf{I}_{0}^{HB5}$ = **G** Connector linked to HB5's input.

| Global Variables | | Temporary Variables | | |
|---|---|---|---|---|
| ArcL Ref | BL Ref. | Connector | BL Ref. | Del. Timeslot |
| *v1* | *gv1* | D | *tv1* | 2 |
| *v2* | *gv2* | F | *tv2* | 3 |

**Table 4.2:** *Contents of the SNSMem variable table after receiving the second request message from ArcL (timeslot t = 1).*

**s/n** $^{HB5}_{I\ o}$ = ***gv1*** Name of the variable linked by connector "G."

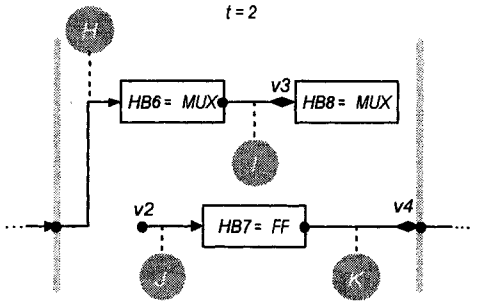
This request message (*i.e.*, 4.14(b)) has three new characteristics that were not present in the previous request message. The first particularity is the fact that the HB5 does not have a connector linked to its output. Hence, there is no reference to the output in the request message. The second particularity is that the message refers to the temporary variable *tv1*. The reason is that the HB4 gets its input from the connector that links it to the timeslot boundary, which translates into a temporary variable. This allows HOSes to load the circuits in timeslot t = 1 and restore the context of the HwApp which permits using several HwApps in the same hardware space. The third and last particularity is the reference to the global variable *gv1* because HB5's input is the ArcL variable *v1* which has the BL reference of *gv1*. The reference to the global variable was sent to ArcL in the first response message.

The request message for timeslot t = 1 had the effect on the SNSMem table shown on table 4.2. The only change after the first request message is the creation of the temporary variable *tv2* which corresponds to the connection F that links HB4 and the timeslot boundary. The answer to this request message, which is sent by the BL to the ArcL, is shown in figure 4.14(b).

The response to the second request message is short because there is only one variable reference returned. The returned variable is *tv2* (field RE $^{HB4}_{0}$ ) and is the result of the connection between HB4 and the timeslot boundary.

| MT | MI | $O_0^{HB4}$ | $RE_0^{HB4}$ |
|---|---|---|---|
| 1 | 3 | F | tv2 |

**Figure 4.15:** *HRRMS response message generated by BL after receiving the second request message depicted in figure 4.14(b).*




(a) Timeslot t = 2 breakdown

| | | HB6 | | | | | | HB7 | | | | | | HB8 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MT | MI | $BT^{HB6}$ | $BR^{HB6}$ | $I_0^{HB6}$ | $s/n\ I_0^{HB6}$ | $O_0^{HB6}$ | $s/n\ O_0^{HB6}$ | $BT^{HB7}$ | $BR^{HB7}$ | $I_0^{HB7}$ | $s/n\ I_0^{HB7}$ | $O_0^{HB7}$ | $s/n\ O_0^{HB7}$ | $BT^{HB8}$ | $BR^{HB8}$ | $I_0^{HB8}$ | $s/n\ I_0^{HB8}$ |
| 0 | 4 | MUX | 0 | H | - | I | v3 | FF | 0 | J | gv2 | K | v4 | MUX | 1 | I | v3 |

(b) Request message for timeslot t = 2

**Figure 4.16:** *Figure 4.16(a) shows the breakdown for timeslot t=2 for the HwApp in figure 4.9. The figure below, 4.16(b), shows the request message generated by the ArcL and sent to the BL.*

**Third Act** After that the ArcL receives and process the response message for timeslot t = 1, it will send the request message for timeslot t = 2. As with the previous request message, this one will use the BL variable references (*e.g.*, *gv1*, *tv2*) if applicable. The breakdown, for illustration purposes, of the third timeslot (t = 2) is shown in figure 4.16(a) along with the request message 4.16(b) generated by the ArcL which is sent to the BL.

The state of the SNSMem variable table is shown in table 4.3. It can be seen that two new global variables have been created following the reception of the request message for

| Global Variables | | Temporary Variables | | |
|---|---|---|---|---|
| ArcL Ref | BL Ref. | Connector | BL Ref. | Del. Timeslot |
| *v1* | *gv1* | D | *tv1* | 2 |
| *v2* | *gv2* | F | *tv2* | 3 |
| *v3* | *gv3* | | | |
| *v4* | *gv4* | | | |

**Table 4.3:** *Contents of the SNSMem variable table after receiving the second request message from ArcL (timeslot t = 2).*

timeslot t=2. The variables are *gv3* which holds the output from HB6 and the variable *gv4* which holds the output from HB7. Note that there is no temporary variable for the connection K which is linked to the timeslot boundary. The reason being that the data are already stored in global variable *gv4*.

| MT | MI | $O_I^{HBI}$ | $RE_I^{HBI}$ | $O_0^{HB2}$ | $RE_0^{HB2}$ |
|---|---|---|---|---|---|
| 1 | 5 | I | gv3 | K | gv4 |

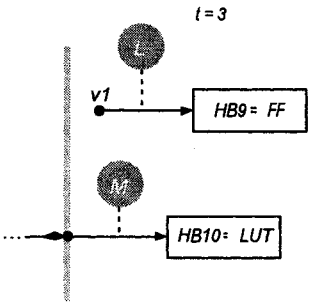
**Figure 4.17:** *HRRMS response message generated by BL after receiving the third request message depicted in figure 4.14(b).*

The response message in figure 4.17 reflects the contents of the SNSMem with only two new global variables: *gv3* and *gv4*.

**Fourth and Final Act** The last timeslot is resumed after the ArcL receives the response message depicted in figure 4.17. The ArcL resumes by sending a final request message to the BL with the data of the fourth and last timeslot (t=3).

The SNSMem table remains unchanged after receiving the fourth request (figure 4.18(b)) by the BL. The reason is that the request message does not contain any instructions regarding the creation of new variables. And since this is the last timeslot, no new temporary variables will be created either.

The response message generated by the BL for the ArcL is shown in the figure 4.19.




(a) Timeslot t = 3 breakdown

| | | HB9 | | | | HB10 | | | |
|---|---|---|---|---|---|---|---|---|---|
| MT | MI | $BT^{HB9}$ | $BR^{HB9}$ | $I_0^{HB9}$ | $s/n\ I_0^{HB9}$ | $BT^{HB10}$ | $BR^{HB10}$ | $I_0^{HB10}$ | $s/n\ I_0^{HB10}$ |
| 0 | 6 | FF | 0 | L | gv1 | LUT | 0 | M | gv4 |

(b) Request message for timeslot t = 3

**Figure 4.18:** *Figure 4.18(a) shows the breakdown for timeslot t=3 for the HwApp in figure 4.9. The figure below, 4.18(b), shows the request message generated by the ArcL and sent to the BL.*

| MT | MI |
|---|---|
| 1 | 7 |

**Figure 4.19:** *HRRMS response message generated by BL after receiving the fourth request message depicted in figure 4.14(b).*

This message is empty since there is no new changes in the SNSMem variable table. This concludes the step by step decomposition of the HRRMS messages necessary for the execution of a HwApp.

## 4.6 Physical Layer

The *physical layer* (PL) controls and manages the FPGA resources which includes the placement and execution of all HBs. Its main functions are modelling the hardware elements, placement of HBs and managing all the physical resources. Since this layer deals

directly with the hardware resources, its architecture is closely related to the architecture of the platform, which in this case is the Virtex FPGA from Xilinx. This author has contributed to its design and to the efforts of updating its architecture. The updating process is presented in more detail in section 4.7.1.

### 4.6.1 Operational Requirements

The PL's role is to accurately manipulate all the HBs that the layers above instruct it to manage. Because our application is an RTE, the physical layer must provide a solution to the following operational requirements:

**Reconfiguration** PL must be able to configure any HB on the FPGA during runtime and on demand, hence the term *reconfigure* since it has to be able to modify the configuration repeatedly while running.

**Unconfiguration** The PL must also be able to unconfigure (*i.e.*, delete) any HB from the FPGA without affecting the overall stability of the system during runtime and on demand.

**Swapping** The PL must be able to remove an HB and place another one which will reuse part or all of the resource of the previous HB during runtime and on demand.

**Linking** The PL must be able to link HBs to other HBs on the FPGA during runtime and on demand.

**Relocation** The PL must be able to move an HB from one location on the FPGA to another one during runtime and on demand.

**Allocation** The PL must allocate all necessary resources for an HB during its configuration.

**Deallocation** The PL must deallocate all used resources for the deleted HB upon un-configuration.

### 4.6.2 Functional Requirements

In order to perform the operation requirements, the PL must be able to implement solutions for the following functional requirements:

**Partial Reconfiguration** The PL must be able to partial reconfigure the FPGA since in order to implement an RTR system, it is imperative to allow the non-changing parts of the system to execute while the others are moved in or out. This implies the following:

- Partial reconfiguration does not affect the parts of the system that are not being reconfigure (*e.g.*, other HBs from the ones being manipulated).
- Partial reconfiguration does not interfere with the operation of the other HBs that are allocated elsewhere on the system. In the Virtex chips, this is limited to the circuits that are not located in the same column as the target HBs. This is a limitation of the Virtex FPGAs.
- Partial reconfiguration of the HBs does not affect their original functionality and operation at any time during their reconfiguration.

**Column allocation** Correct column allocation and manipulation is imperative since the Virtex architecture is based on columns of configurable resources. For this reason, the PL must respond successfully to the following requirements:

- The PL must be able to manage the allocation of HBs into separate columns. This is necessary when allocating an HB into a location with limited number of

reconfigurable logic. The system must then be able to use the space available in another adjacent column for the successful allocation of the target HB.

- The PL must allow the manual allocation of HBs onto selected columns if enough resources are available.

**Re-route** The PL must be able to place any HB anywhere in the space reserved for their allocation. This implies that the PL must be able to route all the logic resources used by the HBs.

**Defragmentation** The PL must be able to re-route HBs in order to decrease the use of resources. The use of these resources could be inefficient after moving HBs in and out of the system. Since HBs are most likely different in shape, the space used will be filled with unused "holes" in the logic fabric. To reduce the unused space, the PL must be able to defragment the system. The defragmentation most take into account the two following points because of the Virtex architecture:

- Take into account the column based architecture of the Virtex FPGAs. For example, shape HBs in forms that use columns in an efficient manner since modifying a column affects execution of all elements within said column.
- Minimize fragmentation in the FPGA when allocation different HBs that differ in size. This will minimize the unused space created by the "holes" of logic fabric created by the HBs manipulation.

**Testing** The PL must allow the testing of HBs by allowing the stimulation of their inputs pins, probing their output pins, and verification of their logic. The PL must also be able to verify that re-routing, partial reconfiguration, column allocation, and defragmentation work as intended.

The PL uses JBits to control and manage the hardware resources. JBits is a API created in the Java language by Xilinx. This API permits to model some FPGAs of the Virtex family. It also provides an interface to control an actual FPGA. JBits provide very low level access and control of the reconfigurable device. For these reasons the PL uses extensively this tool. See section 2.5 for more information on JBits.

### 4.6.3 PL's API

The PL is composed of three distinct Java classes. These classes are [30]:

**HOSes_HwBlk** This class implements the HB and keeps all the information related to it. For each HB there is one instance of this class. The information that it holds consist of hardware resources such as CLBs and routing.

**HOSes_FPGAResources** Management of the FPGA resources is done by this class. While HOSes_HwBlk holds the information about the hardware resources, HOSes_FPGAResources manages them.

**HOSes_PartialRTR** This class implements the API that the bridging class utilizes to communicate with the PL. As it name indicates, this module is responsible for the RTR functionality. Below is the list of available functions provided by the API.

`testHwBlk()` Function used to test a HB at run-time.

`insertHwBlk()` This function is used to insert a HB in the available space on the FPGA at run-time.

`removeHwBlk()` A call to this function removes a HB from the FPGA at run-time.

`defragFPGA()` This functions instructs the RTR system in the PL to perform defragmentation on FPGA.

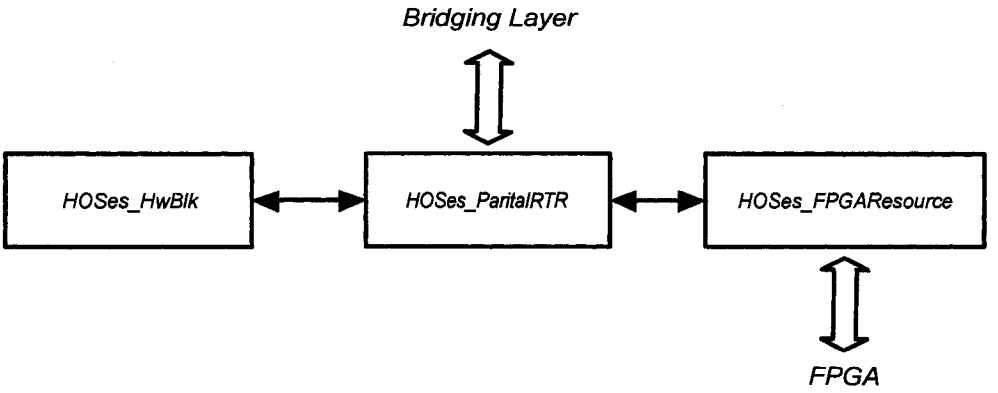



**Figure 4.20:** *Physical layer block diagram that presents its architecture and its relationship with the other layers..*

`clearFPGA()` Clears the entire FPGA.

`connectToIOB()` Used to connect a HB I/O with a given IOB.

`disconnectFromIOB()` This function disconnects a HB I/O from a IOB.

`connectClb()` This function connects to output of one HB to the input of another HB.

`disconnectClb()` This function is used to remove a connection between two HBs.

`testHwBlkRAM()` This function test the contents of a BRAM with another.

**Hardware Blocks RTR** The PL has the task of loading the different HBs into the configurable device. This implies that the PL must be able to manipulate partial or complete bitstreams that define the different circuits to be used. The PL must be able to load a given HB into a reconfigurable device regardless of the HB shape and composition. This allows the use of several HBs in a given space inside the device.

By having low level control by using JBits, the PL can support placement algorithms which goal are to find the most efficient use of the FPGA space. By stacking HB con-

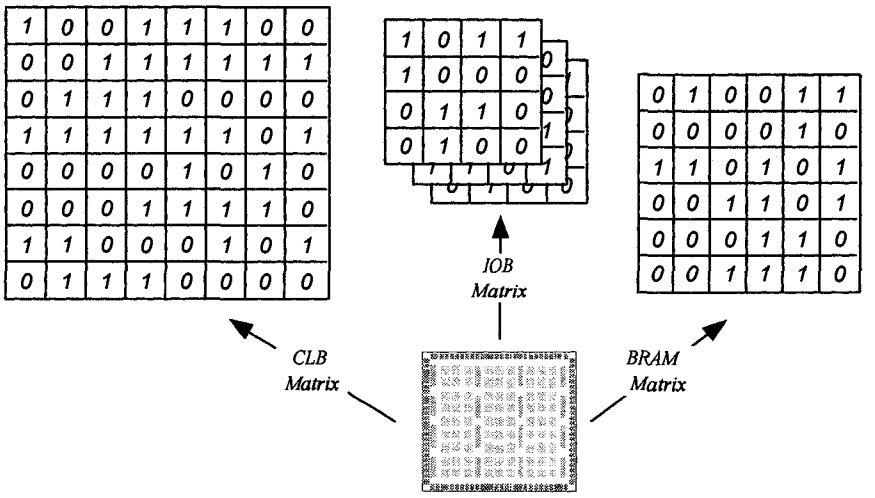



**Figure 4.21:** *The physical layer manages the FPGA resources using three different matrices; one for each resource (i.e., CLB, IOB, and BRAM).*

tiguously, HOSes provides an development environment able to support complex designs since the available space often dictates the amount of resources available.

**FPGA Resource Management** The management of the FPGA resources is done by the HOSes_FPGAResources block which is shown in figure 4.20. HOSes_FPGAResources is responsible for correctly administering all available resources. In the Virtex FPGA these resources are: CLBs, IOBs, and BRAMs. The method chosen to control these resources is one that maps all of them into a FPGA resource matrix where the different elements are represented by 0 and 1 which indicate a *free* or *user* resource respectively. This allows for a design that uses simple datatypes and classes which was primordial in order to implement a simple, but powerful, API which is presented to the BL.

For each Virtex FPGA element (*i.e.*, CLB, IOB, and BRAM) there is a matrix which keeps track of their usage. The CLB matrix is 2-dimensional (2D) where one dimension is used for the rows and the other is for the columns in the FPGA. A 4-dimensional (4D) matrix is used to keep information about IOBs. It requires 4 dimensions since we find

4 sites for each IOB index. The first dimension represents the FPGA side (*i.e.*, bottom, left, top, or right). The second dimension is the IOB index. The third dimension is the IOB site, as we find 4 sites at each IOB index (note that a bug in JBits 2.8 prevents the use of site 0). Finally, the fourth dimension is used to specify whether each site is being used as an input or/and output IOB. The BRAM matrix is 2-dimensional where the first dimension is the BRAM row, while the other is the BRAM column. This management system is depicted in figure 4.21.

**FPGA Defragmentation** The PL provides run-time reconfiguration (RTR) capabilities that can be used in an actual physical device thanks to the use of JBits. By having RTR access to the device, it is possible to perform defragmentation of the HBs. This is necessary because even if the placing algorithm that is in charge of selecting the location in an area big enough to fit the HwApp, the continual addition and deletion of HwApps will create holes (*i.e.*, unused portions of logic fabric). This is required because of the partition scheme used by HOSes. This scheme is fairly fine grained which is classified as variable partitioning (see "Task Sizing" in section 2.1.3.4). To perform defragmentation with HOSes, the function `defragFPGA()` should be called. The PL will then perform the defragmentation.

**PL in Action** The PL is one of the most advanced system in HOSes since it interacts with the hardware through JBits. The figure 4.22 shows a screenshot of an early implementation of the PL. It shows a GUI where the system is manipulating a simulated Virtex device.

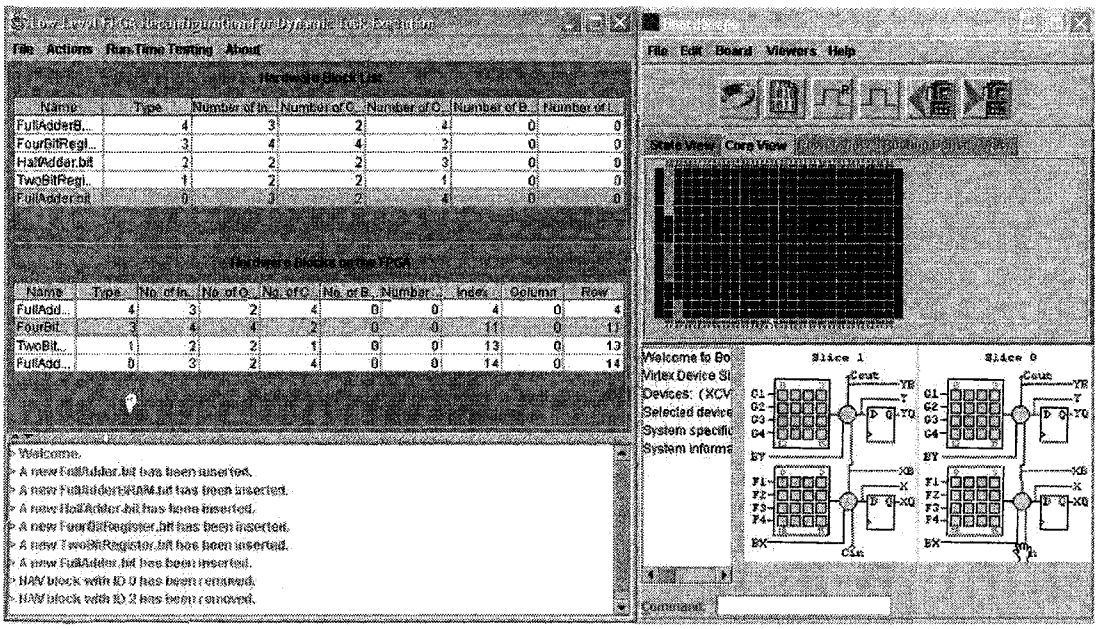


**Figure 4.22:** *Screen-shot of the legacy GUI developed which interacts with the physical layer ([30]).*

### 4.6.4 Hardware Block Architecture

The architecture of HBs are highly coupled with the platform (*e.g.*, FPGA) for which it is designed for. One of the intents of an HB is to exploit the inherent parallelism found in the reconfigurable devices. In a system such as HOSes, HBs need to provide mechanism for which they can control their internal elements with minimal supervision. This is achieved by providing internal registers that contain bit flags that are used to hold status information and control data. A diagram that shows this architecture is shown in figure 4.23.

The HB architecture diagram (figure 4.23) shows the elements inside the HB. These elements are separated into two sections , these are the *HB dependent unit* (HBDU) and the *HB independent unit* (HBIU). The HBDU are all the elements which are platform dependent. On the other hand, the HBIU does not depend on the platform since it uses logic elements such as registers that are standard elements in reconfigurable devices.

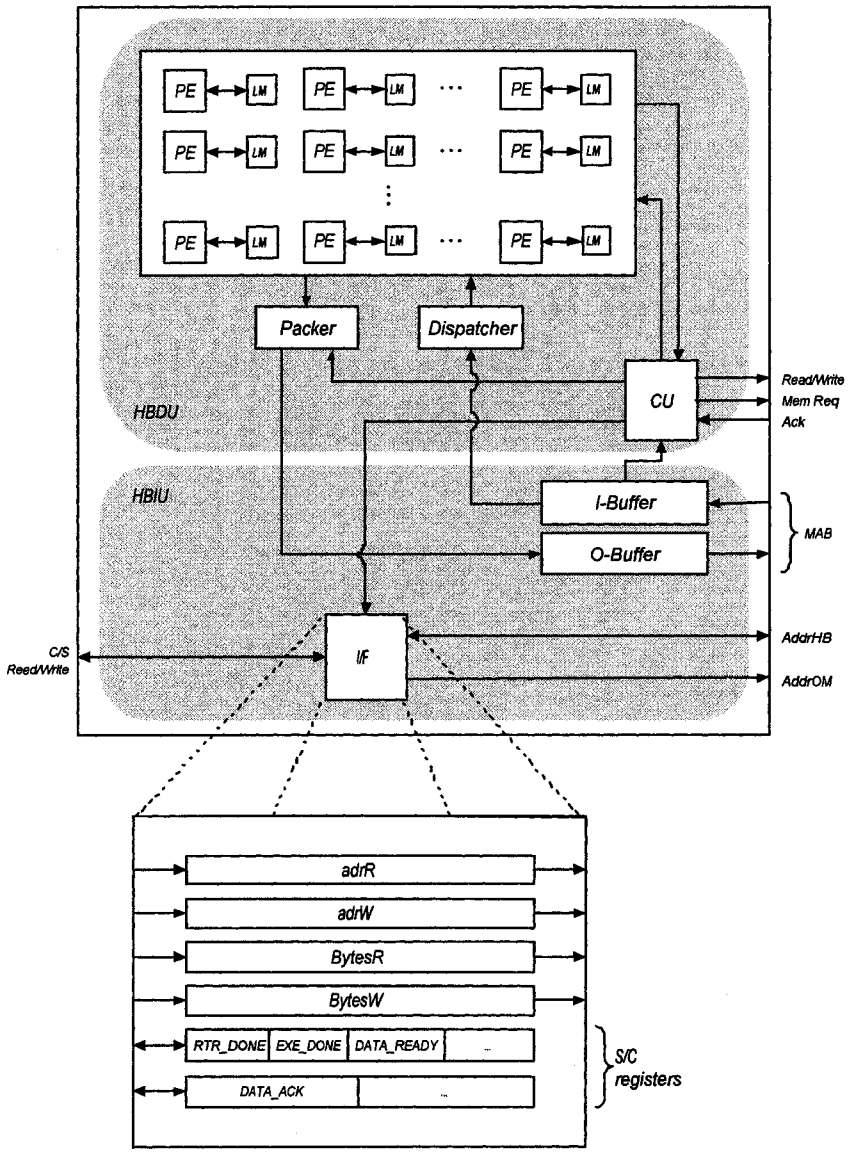



**Figure 4.23:** *Hardware Block (HB) architecture.*

The HBIU is composed of three important modules: input buffer (I-Buffer), output buffer (O-Buffer), and the interface (I/F) module. The I/F module consists of several addressing registers. It also contains a status/control register (S/C) which holds status and control flags. The different flags are used to synchronize the HB execution with its environment. The addressing registers, adrR and adrW, are used to transfer the input

and output data used by the HB. They indicate the starting address of the on-board memory (AddrOM) blocks that are read and written by the HB. The registers BytesR and BytesW indicate the number of bytes to be read and written. The other two modules are the I-Buffer and O-Buffer which interface the memory access bus (MAB) in order to read and write data to and from the on-board memory (OM).

The HBDU is the part of the HB that is dependant on the architecture of the platform. The HBDU is composed of four modules: the processing element (PE) alongside its local memory (LM), the control unit (CU), the packer, and the dispatcher. The set of PEs & LMs are specific to each HB. The process the incoming data so as to fulfil their task. The different PEs will process the data and when the last PEs have finished their processing, these ones will tell the CU that the execution has finished. The CU instructs the Packer to grab and format the data and to transfer it to the O-Buffer where it will be stored before being transfered to the OM. At the other end of the PE, we find the Dispatcher which receives the data from the I-Buffer and then it dispatches it to the PE for processing. The CU is the main module in a HB since it is its obligation to synchronize execution and memory access with the OM. It sets and examines the flags in the I/F module.

## 4.7 Implementation

The current implementation of HOSes has been limited by the options available and the limitations of the tools. Since the PL design was started using JBits 2.8, any other development was subject to the same constraint. To recall, JBits is a set of Java classes, therefore a Java virtual machine (JVM) must also be used. The use of JBits 2.8 limits the JVM choice to only one that is Java 1.2.2. Any other, more recent, version of Java is not supported. Consequently, the most recent language features cannot be used; also any new additions must be done with all the bugs that have been identified for said system,

if any. The current development setup is highlighted below.

**Hardware Environment** The current HOSes implementation has been done in a x86 based system. The microprocessor used is a Pentium III that runs at 662 MHz. The system has 385 Mb of RAM and a hard-drive capacity of 88 Gb.

**Software Environment** HOSes runs in a GNU/Linux based system. The GNU/Linux distribution used is RedHat 7.3. Several applications for development software have been installed. The applications required by the current implementation are GCC 2.96, GNU Make 3.79.1, Java 1.2.2, and JBits 2.8.

### 4.7.1 Upgrade Pitfalls

The current implementation of HOSes reposes on an outdated software environment by current standards. For example, the most recent version of Java is 1.5 (Java EE SDK 5) which is the most update development kit. While Java with version 1.5 is available it cannot be used without ensuring that the other software packages (*i.e.*, JBits and PL) can be linked against it. Sadly, the newest version of JBits is 3.0 which requires Java 1.4.1_02 as stated in its documentation (Available at [24] after registration).

Migration of the current implementation to using the updated Xilinx' JBits is not a simple task. Xilinx changed the internal architecture and APIs of JBits. Consequently, the migration of the PL to this new JBits version will require a complete rewrite because there has been substantial changes from one version to the next. This severely impedes the updating of HOSes and somewhat locks it to the "outdated" environment presently used. This also has the effect of locking the OS which is RedHat 7.3. The only GNU/Linux distribution that Java 1.2.2 supports. Both of these software packages are not supported anymore and both vendors (*i.e.*, RedHat and Sun) recommend to upgrade

to newer versions.

To recapitulate, in order to upgrade the tools, the PL must be modified for it to use the JBits 3.0 which requires Java 1.5. This will permit the use of a more up to date OS and other tools such as other programming languages (*e.g.*, Python) and newer versions of Make. It is theoretically possible to bring the new tools to the older version of RedHat if the source code is available, but this seems at first to be a misguided option because of the amount of work involved. The best option is to update the physical layer since the layer architecture of HOSes limits the modules to be modified.

## 4.8 Results

The implemented system was tested by loading four different circuits into a simulated FPGA. The four circuits are:

**Full adder** Circuit that performs an addition of two bits and a carry bit. The result is a two-bit value.

**Full adder BRAM** Same as a full adder with the additional function of saving the result in BRAM.

**Half adder** Circuit that addition two bits. The result is a two-bit value.

**Two bit register** This circuit only store the input (two bits). It does not perform any computation.

These circuits are stored in partial-bitstream files which are loaded into the simulated FPGA using HOSes. Time is used to measure the performance of the system. Time is used extensively in testing BIST methods. The results obtained can be seen in the table 4.4.

| Exhaustive BIST Test Results | | |
|---|---|---|
| CUT | Loading Time (sec) | Test Time (sec) |
| Full adder | 3.1 | 3.484 |
| Full adder BRAM | 3.28 | 3.494 |
| Half adder | 3.1 | 1.355 |
| Two bit register | 2.9 | 1.374 |

**Table 4.4:** *Exhaustive BIST Test Results*

**Loading time** Time used to load the circuit description stored in a partial-bitstream into the FPGA.

**Test time** The time taken to perform *exhaustive* BIST test on the circuit. This test does apply all the possible combinations of input values for the given circuit. The correct function of the circuit is then evaluated by comparing its output with the expected value for the given input.

## 4.9 Summary

This chapter presents HOSes architecture and its inner-workings. It explained the interaction between its components and followed the execution of a HwApp. A discussion of the current implementation is presented as well as the necessary steps for upgrading the system in order to use newer tools.

# Chapter 5

# Conclusion

The computing field and related industries are facing challenges continuously as stated in chapter 3. The relentless research for faster and better IC performance is putting a pressure to the design and testing of these ICs. HOSes provides a new paradigm in co-design of hardware/software applications by utilizing RTR in an innovative way.

I contributed to the design of key elements in the architectural layer that is the engine in charge of the hardware platform abstraction. I also contributed to the design of the physical layer which controls the hardware platform and that also presents a stable API to the upper layers in HOSes. This makes possible the abstraction which is a required characteristic for HOSes. Finally, I have designed the bridging layer which unites the architectural and physical layer into a working system. As explained in the bridging section 4.5, the bridging layer is more than a simple link between the other two layers. It is also in charge of managing the lower layer given the input from the layer above it.

The aim of tools, such as HOSes, is to remove or at least diminish the bottleneck highlighted in the SPLASH section (3.1). My contributions help to get closer to this goal since it pushes forward this project which is HOSes. The availability of HOSes, or other co-design RTE, offers possibilities for developing HwApps that allow the use of

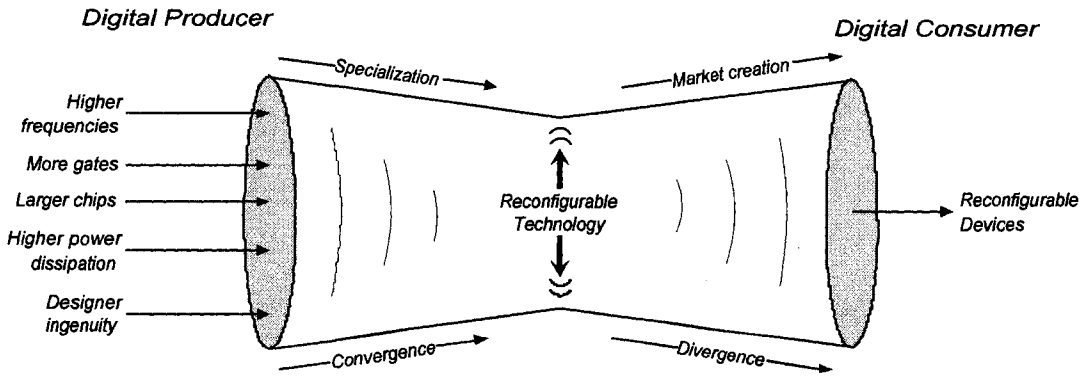



**Figure 5.1:** *Expanded SPLASH bottleneck. See the original SPLASH bottleneck in figure 3.1.*

reconfigurable devices which can replace several ASIC. This has the effect of decreasing the number of chips, hence decreasing the cost and physical space of PCBs while providing the same or more functionality.

The benefits brought by HOSes are depicted in the figure 5.1 where the "loudspeaker" bottleneck is expanded by the use of RTR based technologies such as HOSes.

## 5.1 Alternatives

The alternatives that are or can be used against HOSes, or any other co-design RTE, are listed below. A brief description is presented for each one thereafter.

- General-purpose microprocessors ($\mu$P)
- Digital signal processors (DSP)
- Application-specific integrated circuits (ASIC)
- System-on-chip (SoC)

### 5.1.1 $\mu$Ps and DSPs

Before the advent of microprocessors ($\mu$Ps), designers could only use discrete logic elements (*e.g.*, TTL) in order to devise a solution to a given problem. The solution is fixed and can not be reused for different types of problems. $\mu$Ps helped change that by bringing a general-purpose platform that designer could use to tackle an infinite number of problems. DSP can be seen as an specialized form of $\mu$Ps, and as such they also will be affected by the problems seen by $\mu$Ps.

The popularity of this technology is due in big part to the fact that the industry has been able to provide systems that are twice as powerful as the precedents every couple of years (*i.e.*, Moore's Law). This was been achieved by using several methods such as changing the chemical composition of the materials used in the production of the semiconductors. Another method and by far the most easily applied is the geometrically scaling down of a given circuit. But this method raises new challenges in production methods such as lithography which is reaching its limits because of the limited range of the light's amplitude used by it.

While research has been able to keep the continuous advance in production technology which permits faster and smaller devices; it seems that we are reaching the limits. Santo [32] indicates that the next generation in production technology is having problems and may be an indication of things to come.

### 5.1.2 Application-specific integrated circuits (ASIC)

Another type of system is ASIC which are systems that are designed with a very defined problem in mind. It targets very specific applications and as such it suffers from very high cost of production since creating a chip is a highly complex process. It also requires very specialized designers that are most likely in high demand. Because its logic fabric

is fixed, it is not reconfigurable and condemned to very specialized applications. But ASIC systems do have their place because they are designed to perform certain task very efficiently.

### 5.1.3 System-on-chip (SoC)

SoC are systems composed of various logic cores which may be designed in complete separation and later joined depending on the needs. One of the requirements is that all cores are capable to communicate with each other in one way or another. While it provides more flexibility than ASIC system it stills suffers from the production process but also from IP related issues such as the lack of easily available cores because of licensing issues. SoCs are gaining momentum but there are still many problems that limit the wide adoption of such systems.

## 5.2 Future Work

The work that I would suggest for the future is to bring to maturity the architectural and bridging layers. To develop the application layer (*i.e.*, GUI) and to adapt the physical layer to the latest technology from Xilinx. Regarding this last point, it is important to note that Xilinx has released, in a limited form to only some researchers, the Early Access (EA) partial reconfiguration [33]. Partial reconfiguration is an essential characteristic required for RTR system such as HOSes. The progress to be made in this field of research depend highly on the tools made available by the FPGA manufacturers. Therefore, any work to be performed in HOSes has to take into consideration the limitations and availability of such tools.

Apart from the technical issues already presented and which HOSes address by using

RTR technology, there is also other issues that have a bigger impact and which may not be avoided. In chapter 3 we identified several issues that will affect the computing industry. Many are technical such as the increase in the number of gates which can not be fully exploited because of deficient tools (*i.e.*, SPLASH's plethora). Others factors are more profound such as the increasing cost of energy which is greatly needed in the production of semiconductors. Many of these factors will affect all industries and whole economies may have to change in order to adapt and survive. But how is this going to affect the way we use semiconductors? This is obviously a difficult question that will take years to answer.

Because of the efficient use of resources that can be achieved with RTR technology, HOSes brings innovative solutions forward. By using HOSes to develop new and better BIST methods, the industry is able to increase the quality and production of hardware products. Also, by making available RTR based systems, the hardware is used more efficiently when compared to ASIC which, by specializing in a particular application, are not always utilized. There are lots of possibilities and HOSes is a promising one. The need for better utilized resources will provide a market for such a technology and HOSes has all the ingredients to step up to the challenge.

RC systems are in their infancy and there is lots of work to be done before there is a wide and popular adoption of this technology. This project presents a new tool which tackles the always present challenges in the computing field. This new tool, HOSes, does this from a new angle which is RTR. This allows to marry the hardware development process with software design practices such as OO design when manipulating hardware components. HOSes provides another approach to design which opens the door to a larger pool of designers that are versed in a given technology, be it hardware or software, and that allows them to contribute while knowing little about the hardware platform.

# Appendix A

# Glossary of Terms

**ALU** Arithmetic Logic Unit

**ASIC** Application Specific Integrated Circuit.

**CLB** Configurable Logic Block.

**CMC** Canadian Microelectronics Corporation.

**CPU** Central Processing Unit. A CPU is the central unit in a computer in charge of interpreting and executing machine-language programs.

**DCE** Data Communication Equipment. One of the two devices at the ends of communication channel defined by the RS-232C standard. An example of a DCE is a modem connected to a computer.

**DRAM** Dynamic Random Access Memory.

**DCR** Device Control Register. DCR is a bus used to transfer data between the CPU's general purpose register and the different DCR slave control registers.

**DSP** Digital Signal Processing.

**DTE** Data Terminal Equipment. One of the two devices at the ends of communication channel defined by the RS-232C standard. This device is usually a computer.

**EEPROM** Electrically Erasable Programmable Read-Only Memory.

**FLASH** Non-volatile memory. It can be erased and written electrically. Unlike EEPROM, the programming must be done in blocks of multiple consecutive addresses.

**FPGA** Field Programmable Gate Array.

**GEMS** Group Embedded Micro-Systems. Research group leaded by Dr. Groza. This group is in charge of developing the RTR platform ERACE and ERACE-II.

**GNU** GNU is a recursive acronym for "GNU is not Unix." The GNU project is know mostly because of the suite of program created by the GNU project. See also "GPL".

**GPL** GNU General Public License. License used to distribute software.

**HB** Hardware Block.

**HwApp** Hardware application formed by hardware and software components.

**NFS** Network File System. A network protocol designed by Sun Microsystems that that allows accessing shared files in a network.

**IEEE** Institute of Electrical and Electronics Engineers.

**I/O** Input/Output.

**IP** Intellectual Property.

**IP core** Circuit description that can be added to an FPGA configuration.

**LC** Logic Cell.

**LUT** Look-Up Table.

**OPB** On-chip Peripheral Bus.

**PCI** Peripheral Component Interconnect. PCI is a high speed 32 bit bus found in modern computers.

**PLB** Processor Local Bus.

**RC** Reconfigurable Computing.

**RPU** Reconfigurable Processor Unit.

**RTR** Run-Time Reconfigurable.

**SoC** System on Chip.

**SRAM** Static Random Access Memory. Faster and more reliable than its counterpart DRAM.

**UART** Universal Asynchronous Receiver/Transmitter.

# Bibliography


[1] D. Buell, T. El-Ghazawi, K. Gaj, and V. Kindratenko, "Guest editors' introduction: High-performance reconfigurable computing," *Computer*, vol. 40, pp. 23–27, March 2007. Available at `http://csdl.computer.org/comp/mags/co/2007/03/r3023.pdf`.

[2] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *International Conference Design Automation and Test in Europe*, (Piscataway, NJ, USA), pp. 642–649, IEEE Press, March 2001.

[3] J. Tripp, M. Gokhale, and K. Peterson, "Trident: From high-level language to hardware circuitry," *Computer*, vol. 40, pp. 28–37, March 2007.

[4] N. Moore, A. Conti, M. Leeser, and L. S. King, "Vforce: An extensible framework for reconfigurable supercomputing," *Computer*, vol. 40, pp. 39–49, March 2007.

[5] G. Morris and V. Prasanna, "Sparse matrix computations on reconfigurable hardware," *Computer*, vol. 40, pp. 58–64, March 2007.

[6] S. Alam, P. Agarwal, M. Smith, J. Vetter, and D. Caliga, "Using FPGA devices to accelerate biomolecular simulations," *Computer*, vol. 40, pp. 66–73, March 2007.

[7] M. Santarini, "Is FPGA a simpler puzzle for ASIC designers?," *EDN Magazine*, vol. 15, pp. 58–67, July 2007. Available at `http://www.edn.com/article/CA6459058.html`.

[8] J. M. P. Cardoso and M. P. Véstias, "Architectures and compilers to support reconfigurable computing," *Crossroads [Online document]*, 1999. [March 2007], Available at `http://www.acm.org/crossroads/xrds5-3/rcconcept.html`.

[9] V. Groza, M. El-Kadri, and P. Fu, "Software development environment for run-time reconfigurable system-on-chip," in *Instrumentation and Measurement Technology Conference*, May 2005.

[10] P. Merino, J. C. Lopez, and M. F. Jacome, "A hardware operating system for dynamic reconfiguration of FPGAs," in *FPL '98: Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm*, (London, UK), pp. 431–435, Springer-Verlag, 1998.

[11] N. Shirazi, W. Luk, and P. Cheung, "Run-time management of dynamically reconfigurable designs," in *Field-Programmable Logic: From FPGAs to Computing Paradigm* (R. W. Hartenstein and A. Keevallik, eds.), pp. 59–68, Springer-Verlag, Berlin, 1998.

[12] G. Haug and W. Rosenstiel, "Reconfigurable hardware as shared resource in multipurpose computers," in *Field-Programmable Logic: From FPGAs to Computing Paradigm* (R. W. Hartenstein and A. Keevallik, eds.), Springer-Verlag, Berlin, 1998.

[13] "www.quickflex.com." [Online document], 2007 April. Availalbe at `http://www.quickflex.com/`.

[14] O. Diessel and G. Wigley, "Opportunities for operating systems research in reconfigurable computing," tech. rep., Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, August 1999.

[15] C. Maxfield, *The Design Warrior's Guide to FPGAs.* Orlando, FL, USA: Newnes, Burlington, Massachusetts, USA, 2004.

[16] Xilinx Inc., *Virtex 2.5 V Field Programmable Gate Arrays*, April 2001. `http://www.xilinx.com/bvdocs/publications/ds003.pdf`.

[17] Xilinx Inc., *Virtex-II Platform FPGAs: Complete Data Sheet*, March 2005. Available at `http://direct.xilinx.com/bvdocs/publications/ds031.pdf`.

[18] Xilinx Inc., *Virtex-5 Family Overview LX, LXT, and SXT Platforms*, February 2007. Available at `http://direct.xilinx.com/bvdocs/publications/ds100.pdf`.

[19] A. Khalaf, V. Groza, and R. Abielmona, "Run-time reconfigurable built-in-self-test," in *IEEE Instrumentation and Measurement Technology Conference*, 2006.

[20] M. H. Assaf, R. S. Abielmona, P. Abolghasem, S. R. Das, E. M. Petriu, V. Groza, and M. Sahinoglu, "Implementation of embedded cores-based digital devices in jbits java simulation environment," in *CIT 2004 : International Conference on Information Technology*, pp. 315–325, 2004.

[21] M. H. Assaf, R. R. Abielmona, P. Abolghasem, S. R. Das, E. M. Petriu, and V. Groza, "Jbits implementation and design verification in space compressor design of digital circuits," in *IASTED International Conference on Modeling, Identification and Control*, pp. 415–420, February 2003.

[22] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, pp. 75–82, April 1997.

[23] R. Abielmona, "Lecture 3 - hardware/Software Co-Design." University of Ottawa CEG4910 course presentation, September 2006.

[24] Xilinx Inc., "Jbits SDK home page." Online document, April 2007. Available at `http://www.xilinx.com/products/jbits/index.htm`.

[25] Xilinx Inc., "JBits 2.8 documentation." Online package document, December 1999.

[26] G. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, 1965.

[27] B. Schaller, "The origin, nature, and implications of "Moore's Law"." [Online document], 1996. Available at `http://research.microsoft.com/~gray/Moore_Law.html`.

[28] V. Groza, R. Abielmona, and E. Petriu, "Reconfigurable computing: Exploring emerging technologies," in *IEEE International Conference on Intelligent Engineering Systems*, May 2002.

[29] V. Groza, "Group for Embedded MicroSystems Research Lab (GEMS) home." [Online document], 2007. Available at `http://gems.site.uottawa.ca/`.

[30] V. Groza and R. Abielmona, "What next? A hardware operating system?," in *Instrumentation and Measurement Technology Conference, 2004. IMTC 04. Proceedings of the 21st IEEE*, vol. 2, pp. 1496–1501 Vol.2, 2004.

[31] R. M. Todi and M. M. Heyns, "Germanium: The semiconductor comeback material," *IEEE Potentials*, vol. 26, pp. 34–38, March-April 2007.

[32] B. Santo, "Plans for next-gen chips imperiled," *IEEE Spectrum*, vol. 44, pp. 12–14, August 2007. Available at `http://www.spectrum.ieee.org/aug07/5394`.

[33] Xilinx, "Paritial reconfiguration FAQ." Online article. Viewed June 2007, available at `http://www.xilinx.com/products/design_tools/logic_design/advanced/partial_reconf_faq.htm`.