HI PROF : Hardware Interface for Pipelined Reconfiguration of FPGAs

BY

ALESSANDRO VALLERO
Laurea, Politecnico di Torino, Turin, Italy, 2013

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2014

Chicago, Illinois

Defense Committee:

Jon Solworth, Chair and Advisor
Igor Paprotny
Paolo Prinetto, Politecnico di Torino

ACKNOWLEDGMENTS

I want to thank all LAB 6 guys from Politecnico di Torino who created a stimulating environment that has accompanied me throughout the whole duration of the thesis. Special thanks go to Dr. Pascal Trotta who has transfered to me all its knowledge about reconfiguration, Dr. Giulio Gambardella who was always available to help me and Dr. Marco Indaco who inspired different ideas. I also want to thank all my family who supported and suffered with me during these years.

AV

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

LIST OF ABBREVIATIONS

AGF	Adaptive Gaussian Filter
AIDI	Adaptive Image Denoising IP-core
ASIC	Application Specific Integrated Circuit
ASSP	Application Specific Standard Product
BRAM	Block RAM
CPU	Central Processing Unit
CLB	Configurable Logic Blocks
DCM	Digital Clock Manager
DMA	Direct Memory Access
DPR	Dynamic Partial Reconfiguration
DSP	Digital Signal Processing
FEM	Feature Extraction and Matching
FEMIP	Feature Extraction and Matching IP-Core
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
F+V	Fixed-Plus-Variable

LIST OF ABBREVIATIONS (continued)

HE	Histogram Equalization
HI PROF	Hardware Interface for Pipelined Reconfiguration of FPGAs
HS	Histogram Stretching
HW	Hardware
IET	Image Enhancement Technique
IO	Input Output
IP	Intellectual Property
LUT	LookUp Table
LVE	Local Variance Estimator
NMS	Non-maxima suppressor
NVE	Noise Variance Estimator
OS	Operating System
PC	Personal Computer
PCB	Printed Circuit Board
PLD	Programmable Logic Device
PROM	Programmable Read-Only Memory
RAM	Random-Access Memory
RH	Reconfigurable Hardware

LIST OF ABBREVIATIONS (continued)

RHOS	Reconfigurable Hardware Operating System
RTR	Run-Time Reconfiguration
R&D	Research and Development
SAFE	Self Adaptive Frame Enhancer
SoC	System on Chip
SRAM	Static Random-Access Memory
SW	Software

SUMMARY

In recent years FPGAs (Field Programmable Gate Arrays) market has grown dramatically. Increasing of performances and available resources in FPGA devices, due to technological scaling, have led many designers to adopt FPGA-based solution instead of ASIC. Thanks to the fact that FPGAs offer the possibility of reconfiguring hardware, a new concept has born: *reconfigurable computing*. Reconfigurable computing exploits FPGA to perform tasks by mean of *dynamic partial reconfiguration* (DPR). DPR allows the task of reconfiguring a particular section of an FPGA design while the remaining part is still running.

Reconfigurable computing is becoming increasingly attractive for many applications thanks to its impressive performance and flexibility. However, since development of reconfigurable systems is still a maturing field, there are a number of challenges in developing a reconfigurable system.

The goal of this work is to develop a novel hardware infrastructure to implement a high-performance flexible reconfigurable system able to leverage reconfigurable hardware in an efficient way.

CHAPTER 1

INTRODUCTION

In recent years FPGAs (Field Programmable Gate Arrays) market has grown dramatically. Increasing of performances and available resources in FPGA devices, due to technological scaling, have led many designers to adopt FPGA-based solution instead of ASIC (Application Specic Integrated Circuit). Lately, some FPGAs vendors, such as Xilinx and Altera, introduced in their FPGAs Partial Reconfiguration functionalities. Partial Reconfiguration allow to reconfigure only specific parts of design, instead of reconfigure the entire device. Dynamic Partial Reconfiguration (DPR) allows the task of reconfiguring a particular section of an

FPGA design while the remaining part is still running.

As a consequence, a new idea in digital systems field has raised rapidly: reconfigurable computing. Reconfigurable computing is establishing itself as a major discipline that covers various subjects of learning, including both computing science and electronic engineering. In fact, reconfigurable systems accelerate computation leveraging architectures based on a mixture of general purpose processors and reconfigurable components. Reconfigurable computing is becoming increasingly attractive for many applications thanks to its impressive performance. In fact, recent research suggests that it is a trend rather than a one-off for massive data computing applications such as image processing and floating-point operations. Moreover, thanks to the possibility of reducing power

consumption, component size and count, the possibility of embedding a soft processor and the possibility to be upgraded after market, reconfigurable systems have captured the interest of embedded systems designer too.

Thanks to possibility to integrate hardware and software, hybrid systems, composed of general purpose processors and reconfigurable devices, have rapidly become of interest. In fact, research efforts were addressed to create a hardware and software layers, that is an efficient infrastructure allowing communication between CPU and reconfigurable hardware. Different system-level architecture were proposed for this pupopse (1) associated with the birth of first operating systems for reconfigurable systems(2) (3) (4). RHOSs (Reconfigurable Hardware Operating Systems) were introduced aiming to deal with hardware processes as well as mapping and scheduling functionalities. On the other hand, architectures aspire to provide a simple and efficient interface through which operating system can monitor reconfigurable hardware area. Thanks to the introduction of hardware interfaces and RHOSs, hybrid systems are improving day-by-day, promising to outperform state-of-the-art real-time and desktop computer systems.

However, since development of reconfigurable systems is still a maturing field, there are a number of challenges in developing a reconfigurable system. First challenge is to design an efficient system architecture capable of exploiting the benefits introduced by reconfigurable hardware. In particular, reconfiguration introduces a time overhead so that hardware tasks performance are influenced negatively. Moreover, because of the lack of an efficient hardware

layer interface, RHOS are not being completely exploited. In fact, reconfigurable devices do not offer effective interfaces to support services such as the scheduling and mapping of tasks in the proper reconfigurable area. Secondly, a technique that maps application to a reconfigurable computing system

is still an open issue. This involves determining which parts of the application should be mapped to the reconfigurable fabric and which should be mapped to the

processor. Finally, a scheduling policy for hardware task is another question.

1.1 The goals of the thesis

The goal of this work is to develop a novel hardware infrastructure to implement a high-performance flexible reconfigurable system able to leverage reconfigurable hardware in an efficient way. In particular, the thesis aims at:

- the design of a reconfigurable area architecture able to reduce reconfiguration time overhead
- giving guidelines for designing hardware applications to save the largest amount of reconfiguration time overhead
- the implementation of a system-level architecture providing flexibility of interaction between RHOS and reconfigurable modules
- the support of two new services for RHOS to improve the reconfigurable system flexibility: hardware task preemption and virtual reconfiguration
- the ease of integration of a reconfigurable unit into a bus-based SoC

1.2 Thesis organization

These are the arguments treated in each chapter.

Chapter 2 : Reconfigurable computing

The concept of reconfigurable computing is introduced , focusing on the benefits of reconfigurable systems. Then, hybrid systems, made of general purpose processors and reconfigurable devices, are deepen. Finally, FPGAs, most popular reconfigurable devices are analyzed.

Chapter 3 : It is not just a matter of reconfiguration: the state-of-the-art

In this chapter, the attention is focused on reconfiguration. Reconfiguration is on open issue for reconfigurable systems, in particular for hybrid systems. The importance of reconfiguration comes from the fact it improves systems performance through flexibility, but, on the other hand, a time overhead is introduced due to the reconfiguration process. Since hybrid systems are emerging, it is necessary to create an efficient means to support reconfiguration. Different reconfigurable hardware operating systems have been proposed as a solution. After a detailed analysis on what is reconfiguration and how it works, different reconfigurable architectures are presented. It follows a treatment concerning hybrid system architectures and RHOS.

Chapter 4 : The proposed hardware infrastructure : HI PROF

In this chapter a novel methodology for FPGA-based reconfigurable system is discussed. The goal is to create a flexible hardware architecture for reconfigurable systems. At first, the proposed architecture is analyzed into details, focusing on benefits and drawbacks. Guidelines for implementing tasks suitable for the proposed reconfigurable architecture are then provided. In particular, the process of designing hardware modules is treated deeply. Finally, interaction

between reconfigurable hardware and the OS is discussed, introducing the concept of virtual reconfiguration.

Chapter 5 : Experimental results

In this chapter experimental results are provided for HI PROF. In particular a reconfigurable system is set up in a SoC by means of a Virtex 4 FPGA embedding LEON3 processor. The goal of this case study is to run three image processing reconfigurable IP-core on HI PROF focusing on the design process, virtual reconfiguration space and performance. Special attention will be given to bitstream size, reconfiguration time and area overhead.

CHAPTER 2

RECONFIGURABLE SYSTEMS

In this chapter the concept of reconfigurable computing is given, focusing on the benefits introduced by reconfigurable systems. In particular, hybrid systems, made of general purpose processors and reconfigurable devices, are deepen. Finally, FPGAs, most popular reconfigurable devices are analyzed.

2.1 Introduction to the concept of reconfigurable computing

The idea of reconfigurable hardware belongs to Gerald Estrin and his colleagues at the University of California at Los Angeles. In the mid-1960s, they developed one of the earliest acknowledged reconfigurable computing machines, the Fixed-Plus-Variable (F+V) computer. Estrin thought reconfigurable computing as the technique to accelerate computation by means of variable configurations of specialized hardware modules in addition to a sequential processing unit. In fact, the F+V consisted of a standard processor unit that controlled many other variable units (5), (6), (7), (8).

Reconfigurable systems are a computer architecture supporting reconfigurable computing. They aim at the leveraging reconfiguration so that the designed systems offer high performance through flexibility. Reconfiguration is based on two main entities: a system and a functionality. To produce output data, the system has to elaborate input according to the functionality

required by the application the system runs. In fact, in a scenario where different applications have to be handled, a reconfigurable system can be thought as a system that changes at run-time according to the applications requirements.

2.2 Why reconfigurable computing is promising

In the computer and electronics world, there are two different ways of performing computation: hardware and software. Computer hardware, such as application-specific integrated circuits (ASICs) provide a means of addressing the processing requirements providing highly optimized resources for quickly performing critical tasks. The force point of an ASIC implementation is to provide a natural mechanism for implementing the large amount of parallelism found in applications. Their strength is the efficiency, in fact, ASICs, tailored according to the application's requirements, contain just the right mix of functional units for the target application. In other words, ASICs have the best possible performance. Despite the advantages, ASICs suffer from a long time to market and high costs (9), moreover they are permanently configured to an only-one application. As a consequence, in most of the cases, ASICs are suitable only for very high-volume applications.

On the other hand, computer software provides the flexibility to change applications and perform a huge number of different tasks. High-performance microprocessors provide an off-the-shelf solution to the processing constraint, but, usually, a single processor is not fast enough, so a system composed of many processors has to be designed, introducing complexity to the design. The problem of processors is that they are not optimized for the specific application

which is typical of an embedded system. As a result, their inefficiency is translated to a lack of computing performance, area efficiency and a high power consumption. Software implementation is orders of magnitude worse than an ASIC one. Moreover, the high cost of state-of-the-art processors represents an obstacle to their use in embedded systems. However, using microprocessors means a short time-to-market.

Compared to the past, today, satisfying application's requirements is becoming a challenge for designers. In fact, applications require more processing power than ever before. Applications such as streaming video, image recognition and processing, and highly interactive services are placing new demands on the computation units that implement these applications. At the same time, power consumption, manufacturing costs and time to market requirements add complexity to the system design.

To face up today-application requirements, reconfigurable hardware is coming into the computing field. In particular, the most innovative reconfigurable hardware solution is represented by Field-Programmable Gate Arrays (FPGAs). This concepts are well explained in (10):

> FPGAs are truly revolutionary devices that blend the benefits of both hardware and software. They implement circuits just like hardware, providing huge power, area, and performance benefits over software, yet can be reprogrammed cheaply and easily to implement a wide range of tasks. Just like computer hardware, FPGAs implement computations spatially, simultaneously computing millions of operations in resources distributed across a silicon chip. Such systems can be hundreds of times faster than microprocessor-based designs. However, unlike in ASICs, these computa-

tions are programmed into the chip, not permanently frozen by the manufacturing process. This means that an FPGA-based system can be programmed and reprogrammed many times. However, merging the benefits of both hardware and software does come at a price. FPGAs provide nearly all of the benefits of software flexibility and development models, and nearly all of the benefits of hardware efficiency but not quite. A recent study (11) reports that moving critical software loops to reconfigurable hardware results in average energy savings of 35%to 70% with an average speedup of 3 to 7 times, depending on the particular device used. However, creating efficient programs for FPGAs is more complex than CPUs. Typically, FPGAs are useful only for operations that process large streams of data, such as signal processing, networking, and the like. An old study shows FPGAs need on average 40 times as much area, draw 12 times as much dynamic power, and run at one third the speed of corresponding ASIC implementations (12).

However, recent FPGAs seem to hopefully confirm reconfigurable hardware potentiality so thar perfomance gap between ASICs and FPGAs is always decreasing. In fact, FPGAs such as the Xilinx Virtex-7 or the Altera Stratix 5 have come to rival corresponding ASIC and Application Specific Standard Product (ASSP) solutions by providing significantly reduced power, increased speed, lower materials cost, minimal implementation real-estate, and increased possibilities for re-configuration 'on-the-fly'. Despite the lack of performance, FPGAs offer several advantages compared to ASICs. While an ASIC design may take months to years to develop and have a multimillion-dollar price tag, an FPGA design might only take days to create and cost tens to hundreds of dollars. In fact, a reconfigurable system can be built out of off-the-shelf components, significantly reducing the long design-time inherent in an ASIC implementation. Also unlike an ASIC, the functional units implemented in the reconfigurable fabric can change over time. This means that as the environment or usage of the embedded system changes, the mix of functional units can adapt to better match the new environment. The reconfigurable fabric in a handheld device, for instance, might implement large matrix multiply operations when the device is used in one mode, and large signal processing functions when the device is used in another mode. Typically, not all of the embedded system functionality needs to be implemented by the reconfigurable fabric. Only those parts of the computation that are time-critical and contain a high degree of parallelism need to be mapped to the reconfigurable fabric, while the remainder of the computation can be implemented by a standard instruction processor. As a result, hybrid systems, composed of processors and reconfigurable hardware, seem to be promising in real time embedded systems.
For systems that do not require the absolute highest achievable performance

> or power efficiency, but that require flexibility, a short time-to-market and the ability to easily fix bugs and upgrade functionality, FPGAs are a compelling design alternative.

2.3 FPGA

A FPGA is an integrated circuit that can be programmed in the field one or multiple times after manufacturing. In order to implement a circuit in the FPGA a hardware descriptive language file is needed. This file is the input of a synthesis tool which generates a bitstream according to the constraints specified during the design phase. The bitstream is a binary file containing the configuration of the FPGA implementing the described circuit. In order to make the circuit run, the bitstream file is loaded into the FPGA through a configuration port.

2.3.1 History and performance

An introduction to the history of FPGAs can be found in (13), it is reported below:

> The FPGA industry sprouted from programmable read-only memory (PROM) and programmable logic devices (PLDs). PROMs and PLDs both had the option of being programmed in batches in a factory or in the field (field programmable). However programmable logic was hard-wired between logic gates. In the late 1980s the Naval Surface Warfare Department funded an experiment proposed by Steve Casselman to develop a computer that would implement 600,000 reprogrammable gates. Casselman was successful and a patent related to the system was issued in 1992.
>
> Some of the industry's foundational concepts and technologies for programmable logic arrays, gates, and logic blocks are founded in patents awarded to David W. Page and LuVerne R. Peterson in 1985.
>
> Xilinx co-founders Ross Freeman and Bernard Vonderschmitt invented the first commercially viable field programmable gate array in 1985, the XC2064. The XC2064 had programmable gates and programmable interconnects between gates, the beginnings of a new technology and market. The XC2064

boasted a mere 64 configurable logic blocks (CLBs), with two 3-input lookup tables (LUTs).

The 1990s were an explosive period of time for FPGAs, both in sophistication and the volume of production. In the early 1990s, FPGAs were primarily used in telecommunications and networking. By the end of the decade, FPGAs found their way into consumer, automotive, and industrial applications.

A recent trend has been to take the coarse-grained architectural approach a step further by combining the logic blocks and interconnects of traditional FPGAs with embedded microprocessors and related peripherals to form a complete system on a programmable chip. In 2010, Xilinx Inc introduced the first All Programmable System on a Chip branded Zynq-7000 that fused features of an ARM high-end microcontroller (hard-core implementations of a 32-bit processor, memory, and I/O) with an FPGA fabric to make FPGAs easier for embedded designers to use. By incorporating the ARM processor-based platform into a 28 nm FPGA family, the extensible processing platform enables system architects and embedded software developers to apply a combination of serial and parallel processing to their embedded system designs, for which the general trend has been to progressively increasing complexity. The high level of integration helps to reduce power consumption and dissipation, and the reduced parts count vs. using an FPGA with a separate CPU chip leads to a lower parts cost, a smaller system, and higher reliability since most failures in modern electronics occur on PCBs, in the connections between chips instead of within the chips themselves.

Historically, FPGAs have been slower, less energy efficient and generally achieved less functionality than their fixed ASIC counterparts. An old study had shown that designs implemented on FPGAs need on average 40 times as much area, draw 12 times as much dynamic power, and run at one third the speed of corresponding ASIC implementations. Most recent FPGAs provide significantly reduced power, increased speed, lower materials cost, minimal implementation real-estate, and increased possibilities for re-configuration 'on-the-fly'.

Several market and technology dynamics are changing the ASIC/FPGA paradigm (14):

- Integrated circuit costs are rising aggressively
- ASIC complexity has lengthened development time
- R&D resources and headcount are decreasing

- Revenue losses for slow time-to-market are increasing
- Financial constraints in a poor economy are driving low-cost technologies

Market size of FPGA devices is increasing year by year because the performance gap with ASICs is decreasing. ASIC design is still better than FPGA-based design for area, speed and power consumption, but for many designers are switching to FPGA-based designs when it is possible. Xilinx and Altera are the current FPGA market leaders and long-time industry rivals. Together, they control over 80 percent of the market, with Xilinx alone representing over 50 percent. Other competitors include Lattice Semiconductor and Actel.

2.3.2 FPGA architecture

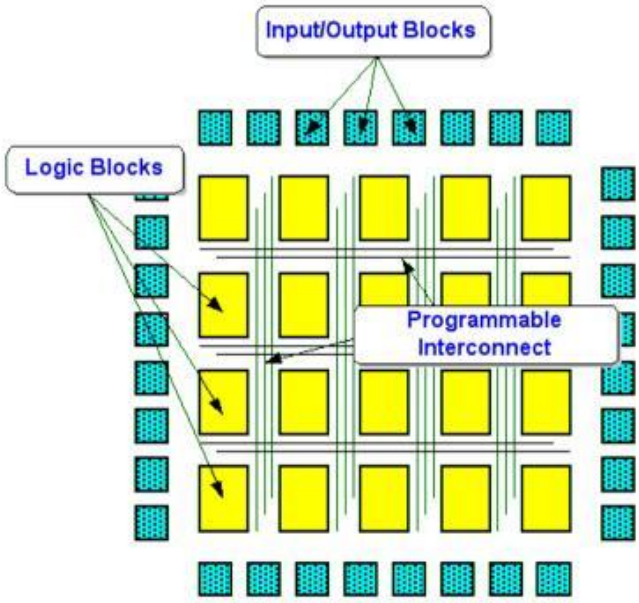


Figure 1. FPGA main structure

FPGAs are independently packaged parts marketed both as prototyping platforms and as reconfigurable alternatives to ASICs. The structure of an FPGA looks like Figure 1. About FPGA architectures is deeply analyzed in (15) and it is reported here:

> The canonical logic block is often considered to be a look-up table that takes four bits of input and generates one bit of output. By filling the table with the right bits, any four-input logic function can be realized. Various studies have suggested that four inputs is a good size for these lookup tables, trading utility (how powerful the blocks are) against utilization (what fraction of their power ends up idle) (16).
>
> Logic blocks in actual FPGAs tend to be more complex than a single lookup table; Figure 3 has a similar diagram for a Xilinx 4000-series logic block, which has two four-input look-up tables and an extra three input table, for a total of eleven bits of input and four bits of output (17), (18).
> A dedicated carry chain circuit at the top of the Figure 2 makes it easy to gang together a line of logic blocks to form a relatively fast multi-bit adder. This diagram in fact ignores many additional details, such as the way Xilinx's two 16-bit lookup tables can be used together as 32 bits of random access memory, or the options available for controlling the clocking of the two single-bit registers. Even a complex Xilinx logic block is quite small compared to the usual functional units of a computer. But in large numbers, small logic blocks can add up to considerable computing power. The die sizes of the largest parts are generally at the boundary of what can be manufactured, but this of course is not true of the smaller parts, and the future is expected to bring only greater densities (15). FPGA is an integrated circuit that contains many, in many case more than 10,000, identical logic cells (LC) that can be viewed as standard components. Each logic cell can independently take on any one of a limited set of personalities. The individual cells are interconnected by a matrix of wires and programmable switches. A user's design is implemented by specifying the simple logic function for each cell and selectively closing the switches in the interconnect matrix. Complex designs are created by combining these basic blocks to create the desired circuit. Field Programmable means that the FPGA's function is defined by a user's program rather than by the manufacturer of the device. Depending on the particular device, the program is either burned in permanently or semi-permanently as part of a board assembly process, or is loaded from an external memory each time the device is powered up.

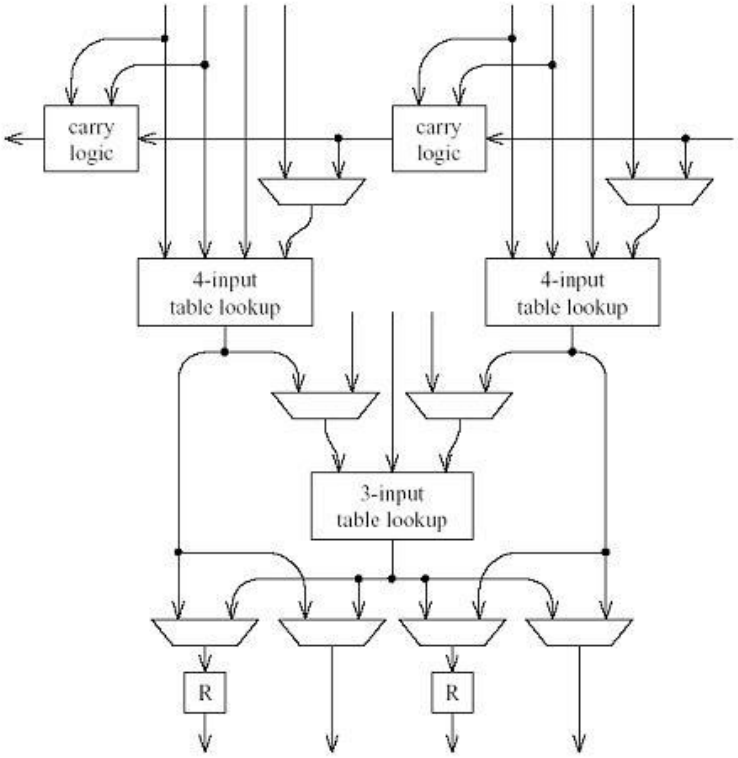


Figure 2. FPGA carry chain circuit

Configurable elements of an FPGA are described by (19) and are reported below:

> FPGA has three major configurable elements: CLBs, input/output blocks, and interconnects. The CLBs provide the functional elements for constructing user's logic. The IOBs provide the interface between the package pins and internal signal lines. The programmable interconnect resources provide routing paths to connect the inputs and outputs of the CLBs and IOBs onto the appropriate networks (20). Each CLB contains four LC's, organized in two similar slices. Figure 3 shows a detailed view of a single slice. Virtex function generators are implemented as 4-input look-up tables, LUT's.

FPGAs can provide perfomance that can be compared to CMOS VLSI. Moreover, costs delays and drawbacks related to masked gate array are reduced in a non negligible way.

2.3.3 FPGA types

Since there are different types of FPGAs, choosing the proper one is a crucial point. In fact they differ from architectures and processes. Four main categories can be distinduished among CPLD: symmetrical array, row-based, hierarchical PLD, and sea-of-gates. Main differences are interconnections and the way they can be programmed. Current technologies in use are: SRAM-based, flash-based and anti-fuse-based. Depending upon the application, one FPGA technology may have features desirable for that application:

- SRAM-based: configuration data is stored in Static RAM memory cells. Since SRAM is volatile and can't keep data without power source, such FPGAs must be configured upon startup. The SRAM cells maintaining configuration require about 6 to 7 MOS per connection; these extra transistors take up extra silicon and increase area. Moreover the external memory needed to load configuration data on the internal SRAM requires extra

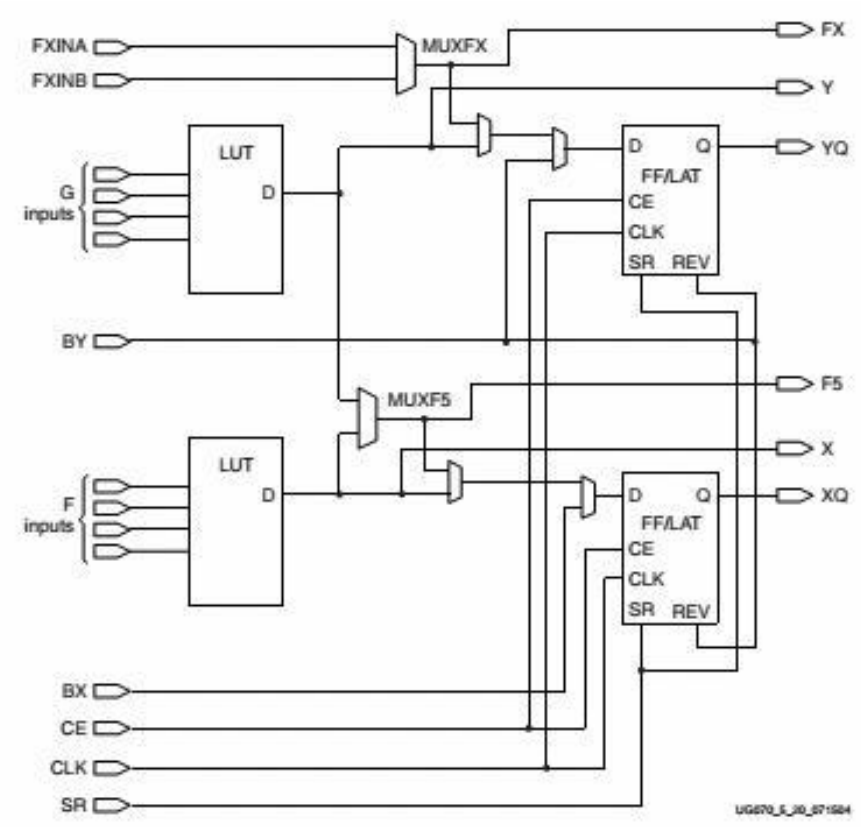


Figure 3. FPGA slice of a XILINX Virtex 4

board space which increases board and component cost to the overall system. The only advantage is the possibility to re-program FPGA at any time and configuration time is smaller than other solutions such Flash-based. SRAM-based FPGAs include most chips of Xilinx Virtex and Spartan families and Altera Stratix and Cyclone families. Some FPGA families, such as Xilinx Spartan-3AN family and Lattice XP2 family, have an internal flash memory that stores configuration data while FPGA is not powered and, on startup, automatically loads configuration data in SRAM cells, eliminating the need for external memory.

- FLASH-based: flash-based FPGAs use flash memory as a primary resource for configuration storage, and doesn't require SRAM; this technology has an advantage of being less power consumptive and is also more tolerant to radiation effects. Examples of Flash-based FPGA families are Igloo and ProASIC3 manufactured by Actel.
- Anti-Fuse based: an anti-fuse is put into high impedance state when it is produced. In a secondary moment, this can be programmed into both low impedance and fused state by means of high currents. Anti-fuse based FPGA can be programmed just once because a burned antifuse cannot come back into the high impedance state. Axcelerator produced by Actel is an example of anti-fuse based FPGAs.

CHAPTER 3

IT IS NOT JUST A MATTER OF RECONFIGURATION : STATE OF THE ART

In this chapter, the attention is focused on reconfiguration. Reconfiguration is on open issue for reconfigurable systems, in particular for hybrid systems. The importance of reconfiguration comes from the fact it improves systems performance through flexibility, but, on the other hand, a time overhead is introduced due to the reconfiguration process. Since hybrid systems are emerging, it is necessary to create an efficient means to support reconfiguration. Different reconfigurable hardware operating systems have been proposed as a solution. After a detailed analysis on what is reconfiguration and how it works, different reconfigurable architectures are presented. It follows a treatment concerning hybrid system architectures and RHOS.

3.1 Reconfiguration

An introduction to reconfiguration have been explained in (10) and it is reported below:

> The flexibility of reconfigurable devices allows them to be customized to a wide variety of applications. Even individual applications can benefit from reconfiguration by using the hardware to perform different tasks at different times.
> If not all of an application's configurations fit on the hardware simultaneously, they can be swapped in and out as needed. In some cases, the circuitry implemented on reconfigurable hardware can also be optimized based on specific run-time conditions, further improving system efficiency. The process of reconfiguring the hardware at run-time, whether to accelerate different applications or different parts of an individual application, is called run-time reconfiguration(RTR).

> Unfortunately, although RTR can increase hardware utilization, it can also introduce significant reconfiguration overhead. Reconfiguring the hardware, depending on its capacity and design, can be very time consuming. Modern high-end FPGAs can have tens of millions of configuration points, and writing this information can require on the order of hundreds of milliseconds (21), (22). n a reconfigurable computing system, where the compute-intensive portions of applications are implemented on reconfigurable hardware, computation and reconfiguration are mutually exclusive operations. Thus, time spent reconfiguring is time lost in terms of application acceleration. Studies estimate that, in some cases, reconfiguration time alone occupies approximately 25 to 98 percent of the total execution time of a reconfigurable computing application((23), (24), (25)).Therefore, management and minimization of reconfiguration overhead to maximize the performance of reconfigurable computing systems is essential.

At first, the process of reconfiguration discuss. Then, different configuration architectures are presented, including those designed specifically to help reduce reconfiguration overhead.

3.1.1 What is reconfiguration?

What reconfiguration is have been explained in (10) and it is reported below:

> In reconfigurable devices, such as field-programmable gate arrays (FPGAs), logic and routing resources are controlled by reprogrammable memory locations, such as SRAM or Flash RAM. Boolean values held in these memory bits control whether certain wires are connected and what functionality is implemented by a particular piece of logic. The process of loading the Boolean values into these memory locations is called reconfiguration. A specific implementation for particular memory locations in hardware defines a specific circuit and is called a configuration for a given hardware task. Runtime reconfiguration involves reconfiguring the device (loading a new set of 1s and 0s) with a different configuration (a specific sequence of 1s and 0s) from the one previously loaded in the reconfigurable hardware (RH). The specific sequence of 1s and 0s for a configuration is called bitstream. The bitstreams themselves are created by CAD software based on both the circuit design to be implemented and the architecture of the implementing RH. The architectural information is required for the design tools to know which

> configuration bits control which resources and what effect a 1 has versus a 0 in each of the configuration bit locations.

There are different models of reconfiguration, that can be classified according to the following scheme (26):

- who controls the reconfiguration;
- where the reconfigurator is located;
- when the configurations are generated;
- which is the granularity of the reconfiguration;
- in what dimension the reconfiguration operates.

The first subdivision (who and where) is between external and internal reconfiguration. In the first scenario, the reconfiguration is managed by an external entity, usually a PC or a dedicated processor. Internal reconfiguration, instead, is performed completely within the FPGA boundaries; to implement internal reconfiguration, the device must have a physical dedicated component, such as the ICAP component in Xilinx FPGAs.

It follows a description taken by (27) about reconfiguration:

> The generation of the configurations (when) can be done in a completely static way (at design time) by determining all the possible configurations of the system. Each module must be synthesized and all possible connections between modules and the rest of the system must be considered. Other possibilities are run-time placement of pre-synthesized modules, which requires

dynamic routing of interconnection signal, or completely dynamic modules generation. This last option is currently impracticable, since it would require run-time synthesis of modules from VHDL(or other hardware description language) code, that is a process requiring prohibitive times in an online environment.

Reconfiguration can take place at very different granularity levels (which), depending on the size of the reconfigured area. Two typical approaches are small bits and module based: the first one consists in modifying a single portion of the design, such as single Configurable Logic Blocks (clb) or IO blocks parameters (28), while the second one involves the modification of a larger FPGA area by creating hardware components (modules) that can be added and removed from the system: each time a reconfiguration is applied, one or more modules are linked or unlinked from the system.

The last property is the dimension. It can be distinguished between two different possibilities: mono-dimensional (1d) and bi-dimensional (2d) reconfiguration. In a truly 2d reconfiguration it is possible to reconfigure an arbitrary portion of the FPGA without affecting the execution of the rest of the implementation. Older FPGA, instead, require that in order to reconfigure a portion of a column of reconfigurable cells the whole column must stop its operations.

3.1.2 Types of reconfigurations

(27) gives an overview about types of reconfigurations, it is reported below:

The easiest way in which an FPGA can be reconfigured is called complete. In this case the configuration bitstream, containing the FPGA configuration data, provides information regarding the complete chip and it configures the entire FPGA, that is why this technique is called complete. With this approach there are no particular constraints that have to be taken into account during the reconfiguration action, obviously that does not mean that the designer is allowed whatever he/she wants only because she/he is using a configuration technique based on a complete-reconfiguration, in fact, if two different bitstreams implement two functionalities that have to work one after the other, see Figure 4 for an example of such a scenario, the designer has to take into account where to store the data between these two configurations. The main disadvantage of an approach based on the complete

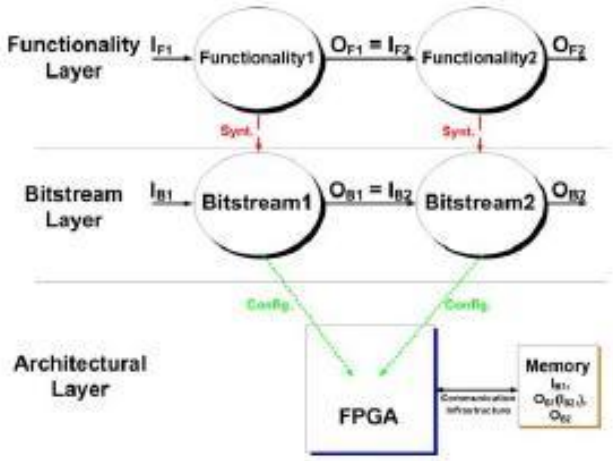


Figure 4. Communication problem between two different configurations

reconfiguration technique is the overhead introduced into the computation by the reconfiguration. In order to cope with this situation a partial reconfiguration approach has been proposed. Partial reconfiguration is useful for applications that require the load of different designs into the same area of the device or the flexibility to change portions of a design without having to either reset or completely reconfigure the entire device. For current FPGA devices, data is loaded on a column-basis, with the smallest load unit being a configuration bitstream frame, which varies in size based on the target device. Active partial reconfiguration of Virtex devices, or simply partial reconfiguration, is accomplished in either slave SelectMAP mode or Boundary Scan, JTAG mode. Instead of resetting the device and performing a complete reconfiguration, new data is loaded to reconfigure a specific area of the device, while the rest of the device is still in operation. The scenario shown in Figure 4 turns into the scenario proposed in Figure 5. Using an approach basedon partial reconfiguration, as the one proposed in Figure 5, the basic idea is to partition the system in a set of functionalities $f_1, f_2, ..., f_n$ able to produce a set of bitstreams $b_1, b_2, ..., b_n$ that are not used to reconfigure the entire system but just a known portion of it. The first bitstream is obviously a complete bitstream but the other functionalities are downloaded to reconfigure just portions of the architecture, as proposed in Figure 6. With such a scenario the reconfiguration time of a portion of the FPGA is hidden

by the computation of the remaining part. According to this last statement it is easy to see that an important component is still missing in the model proposed in Figure 5 and Figure 6. In order to be able to hide the reconfiguration time it is not only necessary to partition the FPGA to obtain the ability to compute partial reconfiguration bitstream, but it is also necessary to guarantee that a reconfiguration is not going to imply a standby in the computation of the not-involved logic of the FPGA. Such a scenario brings to the definition of Dynamic Partial Reconfiguration.

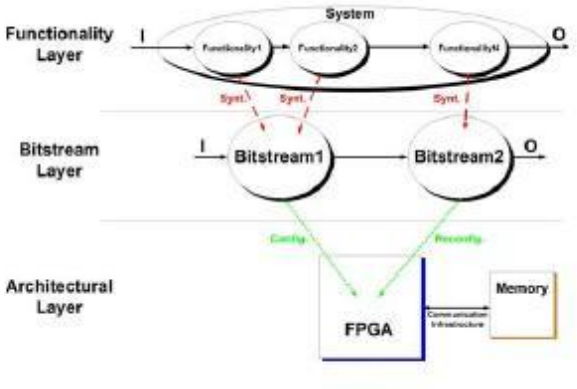


Figure 5. Partial reconfiguration scenario

Dynamic partial reconfiguration is performed when the device is active. Except during some inter-design communication, certain areas of the device can be reconfigured while other areas remain operational and unaffected by the reprogramming. Up to now reconfiguration has been defined from the area and the time prospective, but there is still an important factor that can be used to classify a reconfigurable approach: the location of the controller of the reconfiguration. External reconfiguration implies that an active array may be partially reconfigured by an external device such as a Personal Computer, while ensuring the correct operation of those active circuits that are not being changed. Self or Embedded Reconfiguration extends the concept of dynamic partial reconfigurability. It assumes that specific circuits

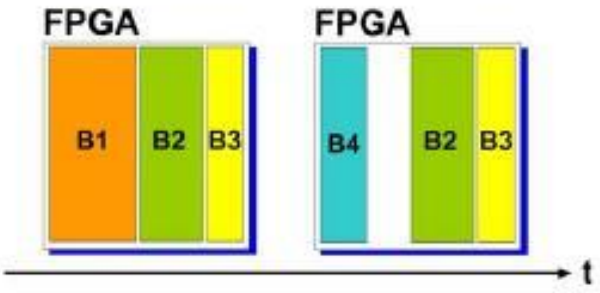


Figure 6. Partial reconfiguration example

on the array are used to control the reconfiguration of other parts of the FPGA. Clearly the integrity of the control circuits must be guaranteed during reconfiguration, so by definition self-reconfiguration is a specialized form of dynamic reconfiguration (28). An important feature in FPGA architectures is the ability to reconfigure not only all the device but also a portion of it while the remainder of the design is still operational. Once initially configured, self-reconfiguration requires an internal reconfiguration interface that can be driven by the logic configured on the array. Starting with Xilinx Virtex II parts, this interface is called the internal configuration access port, ICAP (29). These devices can be configured by loading application specific data into the configuration memory which is segmented into frames, the smallest unit of reconfiguration. The number of frames and the bits per frame are different for the different devices of the Virtex II family. The number of frames is proportional to the CLB width of the device.

3.2 Reconfigurable architectures

In a dynamic partial reconfiguration scenario, reconfigurable architectures aims to organize the reconfigurable space and provide an external interface employed for the communication with general purpose processors and more in general RHOS. In this section the most popular reconfigurable architectures are presented :

- block reconfigurable architectures
- pipeline reconfigurable architecture

3.2.1 Block reconfigurable architecture

It follows a summary about block reconfigurable architecture from (10):

> Block reconfigurable architectures rather than providing one large reconfigurable fabric, they are made up of multiple discrete blocks that can be used independently. Each block is named *reconfigurable slot*. In this case, each slot can contain many logic resources. An individual configuration may occupy one or more slots, but slots may not be subdivided between configurations. Blocks are connected either through a crossbar structure (30) or a bus/network (31), as shown in Figure 7. Although this would seem to describe any architecture formed from multiple connected FPGAs or FPGA cores, block reconfigurable devices have the ability to relocate configurations to different blocks at run time. Relocation is a technique that manipulate the bitstream for a block so that it is suitable for every other block. It is useful because when a module is designed, it can be implemented only for a specific block so that design phase is shorter and the number of bitstreams is smaller. For this reason, the slots of reconfigurable logic, in this style of architecture, have also been referred to as *swappable logic units* (SLU) (32). In the SLU architecture, a block reconfigurable design is implemented as an abstraction layer on top of a partially reconfigurable architecture to facilitate run-time relocation.
> A heterogeneous multiprocessor may fit the block reconfigurable model, provided multiple blocks of reconfigurable hardware are present and configurations can be relocated between the blocks for computational flexibility. These architectures may contain a single communication network used by the configurable blocks and other resources such as microprocessors and custom circuitry. Although the Pleiades reconfigurable architecture (33) has some of these features (a heterogeneous multiprocessor with multiple reconfigurable blocks), computations are preassigned to specific resources, violating one of the requirements of the block reconfigurable category. Finally, despite the creation of a static hardware interface between software and reconfigurable hardware, this architecture does not introduce any benefit to deal with reconfiguration time overhead.

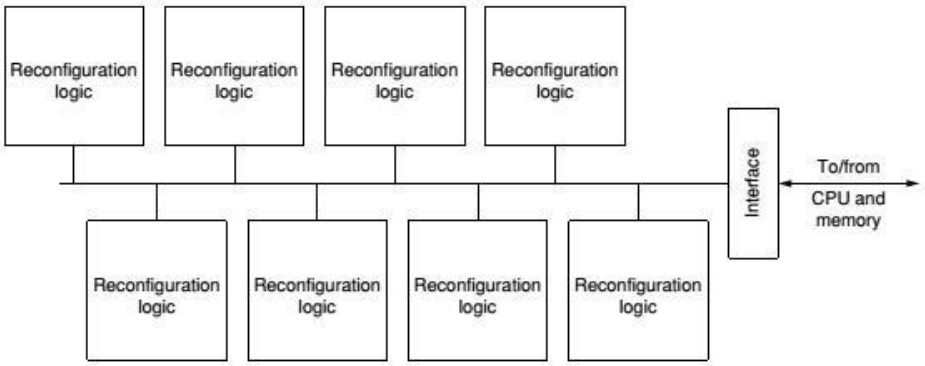


Figure 7. A typical implementation of a block reconfigurable architecture

3.2.2 Pipelined reconfigurable architecture

It follows a summary about pipelined reconfigurable architecture from (10):

> Pipeline reconfigurable arrays use a series of physical pipeline stages to implement the virtual pipeline stages of configurations. A virtual pipeline stage can be relocated to any physical pipeline stage, and the number of virtual stages is generally not constrained by the number of physical stages. The most well-known pipeline reconfigurable architecture is PipeRench (34), which is designed to implement deeply pipelined configurations, subdivided into a set of virtual pipeline stages. At run-time, the virtual pipeline stages are assigned to physical pipeline stage computation units. These units are arranged in a unidirectional ring, as shown in Figure 8(a). Although pipeline stages may be implemented in different physical locations over time, the virtual pipeline appears fixed to its own pipeline stages, with each stage receiving input from its predecessor and generating output to its successor. As execution proceeds, configuration proceeds too so that proper computations are always guaranteed. Pipeline reconfiguration eliminates many of the difficulties of using reconfigurable hardware as virtual hardware, moreover it copes with reconfiguration time because it allows to run execution and reconfiguration concurrently. However, it places restrictions on the circuits that can be implemented as information can only propagate forward through the pipeline stages, and any feedback connections must be completely contained

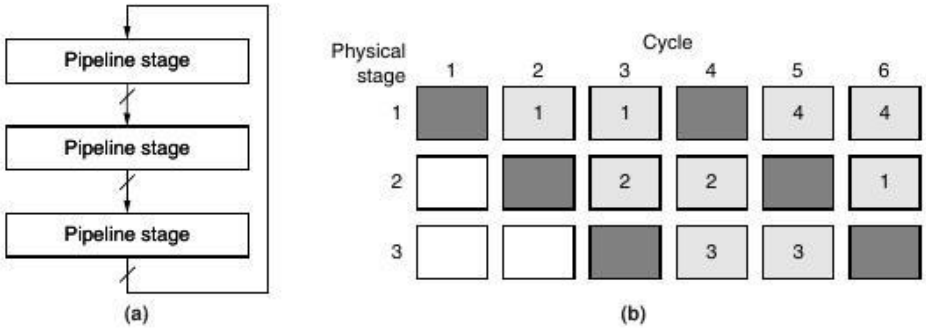


Figure 8. A pipeline reconfigurable architecture with three physical stages (a). A 3-stage physical pipeline implementing a 4-stage virtual pipeline (b). Numbers within physical pipeline stages indicate the implemented virtual pipeline stage. Shaded stages are reconfiguring for the given cycle.

> within a single stage. Finally, PipeRench is implemented in a custom reconfigurable fabric, so that FPGA implementation is not possible. Moreover, this architecture does not provide any interface for reconfigurable hardware and software. Figure 8(b) shows a 4-stage virtual pipeline implemented on a 3-stage physical architecture.

3.3 Hybrid reconfigurable systems

Recently, hybrid reconfigurable systems have captured the attention of embedded system designers. A hybrid reconfigurable system is a reconfigurable system composed of reconfigurable hardware and processors, so that it is to be preferred for several reasons. First of all, reconfigurable hardware offers flexibility and better performance than software, in terms of computational time, area and power consumption. Secondly, as stated in (35):

> FPGAs can reduce the chip count by serving as the glue logic as well as incorporating other pieces of the system. There is a wide range of available

> soft and hard IP cores, including microprocessors, that allows you to pull all these functions into a single chip.

Microcontroller core can be integrated on the silicon or the designer can pour soft IP into free gates and tailor a microcontroller's size and functions to the application at hand (36), (37), (38). Finally, FPGAs vendors are supporting designers needs introducing into the market hybrid architectures composed of high performance processors and reconfigurable hardware (39).

Typically, not all of system functionality needs to be implemented by the reconfigurable fabric. Only those parts of the computation that are time-critical and contain a high degree of parallelism need to be mapped to the reconfigurable fabric, while the remainder of the computation can be implemented by a standard instruction processor. However, in a hybrid reconfigurable system, in order to meet time and power constraint and to improve the quality of service, the designer can decide to implement a task in both software and hardware In this scenario, hardware and software are merged together. As the system is running, an intelligent unit, according to run-time conditions and resources employment, can decide whether application are executed by a processor or by means of a reconfigurable unit.

3.3.1 Hybrid system-level architectures

Hybrid system-level architectures have been deeply analyzed by Todman et al. (1):

> A reconfigurable system typically consists of one or more processors, one or more reconfigurable fabrics, and one or more memories. Reconfigurable systems are often classified according to the degree of coupling between the reconfigurable fabric and the CPU. Compton and Hauck (40) present the

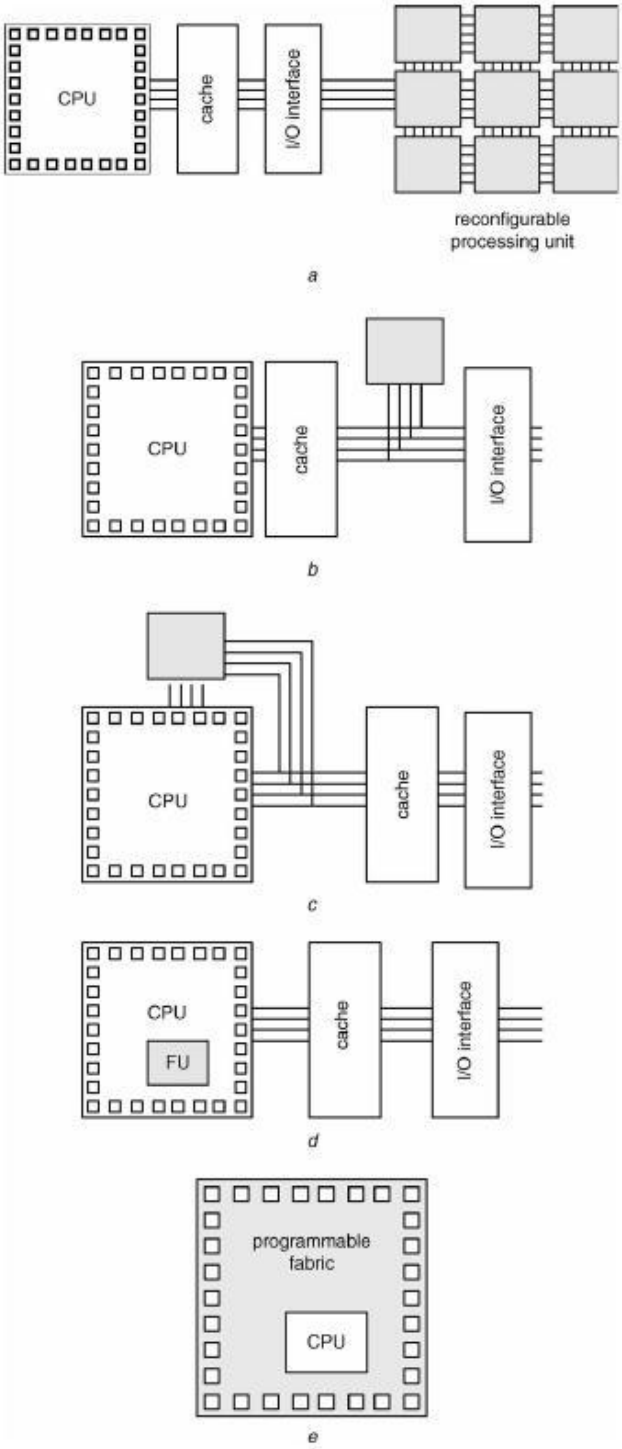


Figure 9. Five classes of reconfigurable systems: *a)* External stand-alone processing unit; *b)* Attached processing unit; *c)* Co-processor; *d)* Reconfigurable functional unit; *e)* Processor embedded in reconfigurable fabric.

four classifications shown in Figure 9a-d. In Figure 9.a, the reconfigurable fabric is in the form of one or more standalone devices. The existing input and output mechanisms of the processor are used to communicate with the reconfigurable fabric. In this configuration, the data transfer between the fabric and the processor is relatively slow, so this architecture only makes sense for applications in which a significant amount of processing can be done by the fabric without processor intervention. Emulation systems often take on this sort of architecture (41), (42). Figure 9.b and Figure 9.c show two intermediate structures. In both cases, the cost of communication is lower than that of the architecture in Figure 9.a. Architectures of these types are described in (34), (43), (44), (45), (46). Next, Figure 9d shows an architecture in which the processor and the fabric are very tightly coupled; in this case, the reconfigurable fabric is part of the processor itself; perhaps forming a reconfigurable sub-unit that allows for the creation of custom instructions. Examples of this sort of architecture have been described in (47), (48), (49). Figure 9.e shows a fifth organization. In this case, the processor is embedded in the programmable fabric. The processor can either be a *hard*core, or can be a *soft*core which is implemented using the resources of the programmable fabric itself. A summary of the above organizations can be found in Figure 10. Note that the bandwidth is the theoretical maximum available to the CPU Organization (a) is by far the most common, and accounts for all commercial reconfigurable platforms.

Class	CPU to memory bandwidth, MB/s	Shared memory size	Fine grained or coarse grained	Example application
(*a*) External stand-alone processing unit				
RC2000	528	152 MB	Fine grained	Video processing
(*b*)/(*c*) Attached processing unit/co-processor				
Pilchard	1064	20 kbytes	Fine grained	DES encryption
Morphosys	800	2048 bytes	Coarse grained	Video compression
(*d*) Reconfigurable functional unit				
Chess	6400	12288 bytes	Coarse grained	Video processing
(*e*) Processor embedded in a reconfigurable fabric				
Xilinx Virtex II Pro	1600	1172 kB	Fine grained	Video compression

Figure 10. Summary of system architectures

3.3.2 An OS supporting hybrid architectures

Recently, reconfigurable system captured the attention in order to provide an efficient interaction between reconfigurable hardware and software. In particular, many efforts were addressed towards the management of the system. Researchers came up with a new concept, reconfigurable hardware operating system (RHOS) (4), (2), (50),(51).

What is stated in (52) is reported below:

> The use of a fully-featured operating system introduces some fundamental advantages and enhancements, but also increases the SW complexity of the system, presenting new issues in resource management. One of the most important features an OS should provide is the exploitation of reconfigurable resources from different processes through multitasking and multiuser capabilities. Modern FPGAs have a reconfigurable area vast enough to allow mapping of a considerable number of IP-Cores, which might be made available to different processes at the same time, exploiting the intrinsic HW parallelism. Additionally, the OS must provide a completely free task-to-resource mapping, similarly to what happens with normal HW resources (memory, IO interfaces) of the system. In (53) Wigley et al. have presented a discussion on the components of an operating system for a reconfigurable computer. These components are equivalent to the ones of a standard operating system. Instead of managing processes, they handle HW tasks, which are mapped on the reconfigurable hardware architecture.

BORPH is the first OS supporting FPGAs, here it is reported what it is explained in (2):

> BORPH (2) is the first OS providing kernel support for FPGA applications by extending a standard Linux operating system. It establishes the notion of hardware process for executing user FPGA applications. Users therefore compile and execute hardware designs on FPGA resources the same

> way they run software programs on conventional processor-based systems.
>
> In conventional OS terminologies, a process is usually defined as an executing instance of a program. It means that the software program represented by an executable file becomes a running process when it is executed. Each process is allocated its own unique process ID together with its executing environment. A process forms a parent-child relationship with its spawning process. BORPH extends this idea to reconfigurable computers, defining a hardware process as an executing instance of a gateware program. In other word, a hardware process is similar to a conventional software process except it may be executing on reconfigurable fabrics of the system instead of the main processor. The notion of execution domain of a process is therefore extended to include spatial information, such as the reconfigurable fabrics that this process is executing.

Scheduling reconfigurable applications is different from the traditional scheduling as reported in (53):

> There are not obvious ways to preempt a hardware application due to the typical absence of the instruction fetch, decode, and, execute cycle. Thus there is no predefined point of completion in a reconfigurable application unless the designer specifically provides this.

For this reason the possibility to preempt a hardware process still needs an efficient solution and it remains an open question. Moreover, as mentioned previously, a scheduler of a reconfigurable system performs mapping and placement so that the operating system has to keep track of all the processes running in the system and which resources are associated to.

In fact, the operating system has to know which and where hardware modules are configured in the reconfigurable area and the time slice assigned to each software process. As a consequence, hardware modules can assume different states which have a physical association

due to the possibility of configuring the same module in different places. The states hardware modules can assume are configuration, execution and cached state. In the configuration status, bitstream of hardware modules and input data are loaded. After the execution of a hardware process is completed, the respective hardware module is cached. This means that hardware module remains configured but there is no process associated to it, in other words, the module remains idle. It is obvious that the scheduler can decide to reconfigure a cached area, but it cannot reconfigure an area where computation is running. When the system is reset, a default configuration is loaded. At system reset the configured modules are assumed as cached.

Here the life-cycle of a hardware process is explained, focusing on what append at operating system level. As the scheduler has mapped a process to hardware, it must decide where to place the associate hardware module. At this point the monitor is updated associating both the hardware process and the hardware module to the chosen reconfigurable portion. If the module is not configured yet, that is the module is not cached, the hardware process is put into suspended processes list and configuration starts. On the contrary, if the module is cached, configuration step is not performed. As configuration finishes, input data are loaded, only if necessary. Then, the process is put into running processes list and computation begins. Next step is when execution is completed, the process is moved to finished processes list and the hardware module is cached.

CHAPTER 4

THE PROPOSED HARDWARE INFRASTRUCTURE : HI PROF

In this chapter a sophisticated hardware infrastructure for FPGA-based reconfigurable System on Chip is discussed. The goal is to create a flexible hardware architecture for reconfigurable systems. At first, the proposed architecture is analyzed into details, focusing on benefits and drawbacks. Guidelines for implementing tasks suitable for the proposed reconfigurable architecture are then provided. In particular, the process of designing hardware modules is treated deeply. Finally, interaction between reconfigurable hardware and the OS is discussed, introducing the concept of virtual reconfiguration.

4.1 The prposed architecture

Typically, a reconfigurable system is composed of four main components: the processor, the main memory, a reconfigurable hardware area and an infrastructure acting as interface between software and hardware. Software processes are executed on the processor, while IP-cores in reconfigurable area. Communication between hardware and software is usually achieved by means of a shared memory. A RHOS has the task of managing all tasks running dispatching them when needed on the reconfigurable device to off-load the main CPU.

The goal of this thesis is to improve the usage of reconfigurable hardware into hybrid systems by resorting to a flexible hardware infrastructure, HI PROF, aiming at:

- The reduction of FPGA reconfiguration time overhead exploiting a pipelined hardware infrastructure .
- The introduction for the first time in literature of the concept of Virtual Reconfiguration Space.
- Provide a set of guidelines for designers in order to develop IP-cores suitable for the proposed solution.

HI PROF is a Hardware Interface for Pipelined Reconfguration of FPGAs. HI PROF, as shown in Figure 11, consists of different blocks which have different functions. The main components are: the configuration manager, the communication manager and the bus-module interfaces. Configuration manager is responsible for reconfiguration, communication manager is responsible for communication among the main memory and the bus-module interfaces and bus-module interfaces are responsible for the data flow through the pipeline. Hereafter each module is presented in detail.

4.1.1 The bus-module interfaces

The heart of the proposed solution is the pipelined reconfiguration. In a pipeline reconfigurable scenario, the main task is divided into many sequential sub-tasks. Each sub-task is implemented as an IP-core (a sub-module). Sub-modules can be thought as independent modules that can be accessed every time they are needed or as a pipeline stage that provides partial results. In order to support a pipelined reconfiguration we have designed a sophisticated interface which has the task to manage data flow among reconfigurable modules instantiated

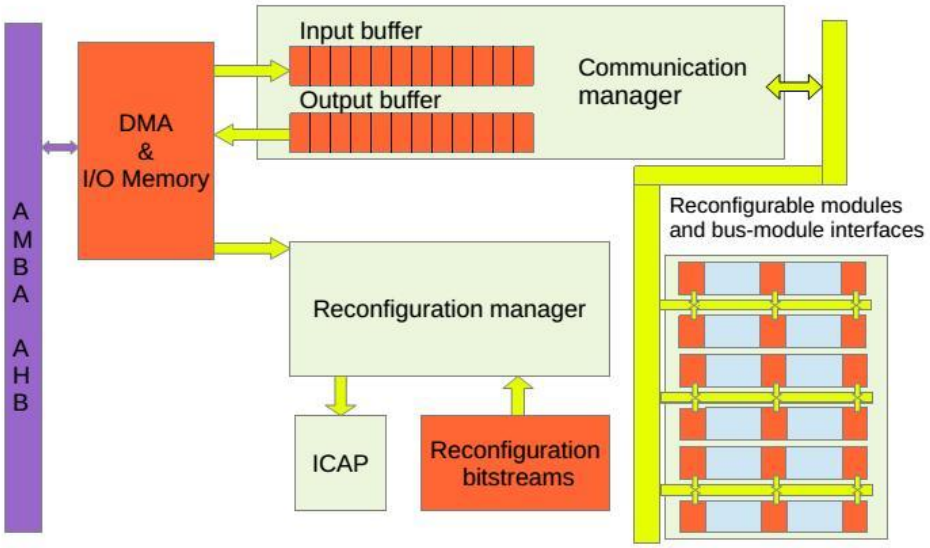


Figure 11. The structure of HI PROF

in the reconfigurable area. In fact, every stage of the pipeline has a non-reconfigurable interface. Bus-module interfaces (Figure 12) are placed between adjacent configurable modules and they consist of registers storing input-output data or. more in general, of memories. Interfaces make data flow correctly through the pipeline stages. In fact a synchronization mechanism is required. Interfaces recognize when the previous stage has finished computing and when next stage is configured. Computation of n^{th} stage of the pipeline can start if and only if n^{th} stage is configured and execution of $(n-1)^{th}$ stage has finished, otherwise it is not possible to obtain correct results. Moreover, to achieve flexibility, interfaces allow to load input data from main memory and to store output data into main memory through a bus.

Interfaces are programmed by the configuration manager and can assume four different state:

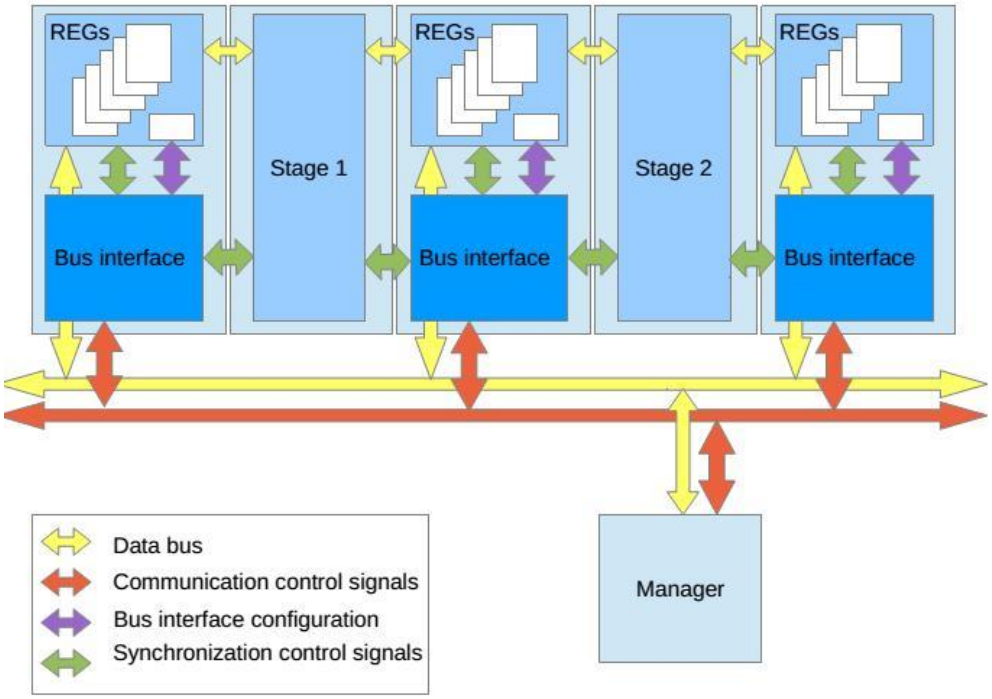


Figure 12. The structure of bus-module interfaces

idle, start, intermediate and stop interfaces. Idle interfaces do not take part in the task execution. Start interface is the first interface of the pipeline. As input data are loaded from the main memory into the registers of the start interface the execution of the first stage starts. Intermediate interfaces, instead, are the interfaces of middle stages. As input data are ready

and computation can start, data pass through intermediate interface registers. Finally stop interface is the interface of the last stage of the pipeline. When computation is over, output data are loaded from registers of the stop interface into main memory.
It must be noticed that according to available reconfigurable slots, the operating system has the possibility to choose which will be the start interface, so that every interface can be programmed to exert all kinds of interfaces.

In order to obtain a correct data flow it must be guaranteed that a module starts its execution after input data are available and after its configuration is completed. As a result, synchronization adds a time overhead to the time a hardware task needs to complete. In fact, in the worst case, three extra clock cycles are required by interfaces: one clock cycle to signal that data are available from the input module, one clock cycle to reset the output module and one clock cycle to move data from the input module to the output module.

To sum up, bus-module interfaces are responsible for storing input and output data of every stage of the pipeline and they guarantees synchronization between configuration and execution. In addition, they provide a mechanism to load/write data from/into main memory. In order to satisfy these requirements, interfaces are made of registers, synchronization signals, and communication signals.

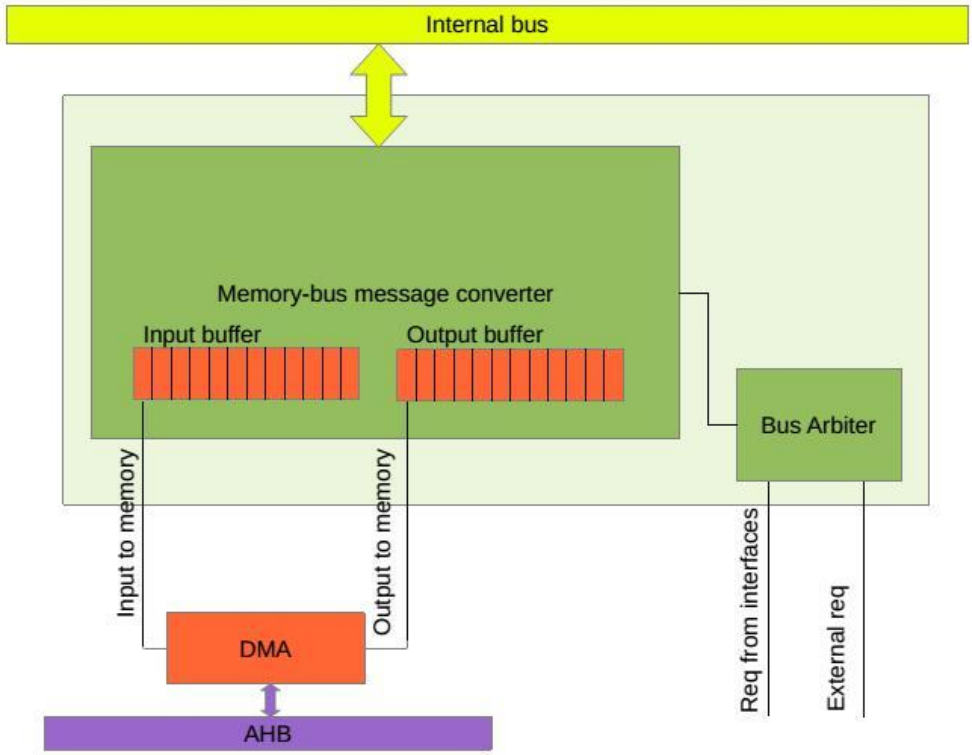


Figure 13. The communication manager

4.1.2 The communication manager

The communication manager (Figure 13) is the master of the communication inside HI PROF, in fact it coordinates data transfers involving main memory and reconfigurable slots. It is an intelligent and autonomous unit. It integrates a DMA, which is a master device of the system bus, so that it can access the memory without the intervention of the processor. In this way, CPU can carry out other task instead of wasting time moving data to reconfigurable devices. The communication manager satisfy requests from both the OS and reconfigurable slots. An arbiter schedules the order the requests are processed. When the OS submits a request the communication manager identify the destination reconfigurable slot, the starting memory address where input data are stored and the size of input data. On the basis of these three variables the communication manager read from the main memory input data and moves them into memory of the desired bus-module interface. As this operation ends, the communication manager writes into a reserved register that input data have been loaded. When a bus-module interface is served, on the contrary, the inverse process is executed.

In order to allow HI PROF run with a clock frequency higher than the one of the system bus, FIFOs have been interposed between the DMA and the internal bus.

4.1.2.1 The internal bus

Despite the aim of this thesis is not providing an efficient infra-module communication, communication inside HI PROF is achieved by means of a single-master multiple-slaves internal busproviding a high bandwidth through a 32-bits data parallelism(Figure 14). The communi-

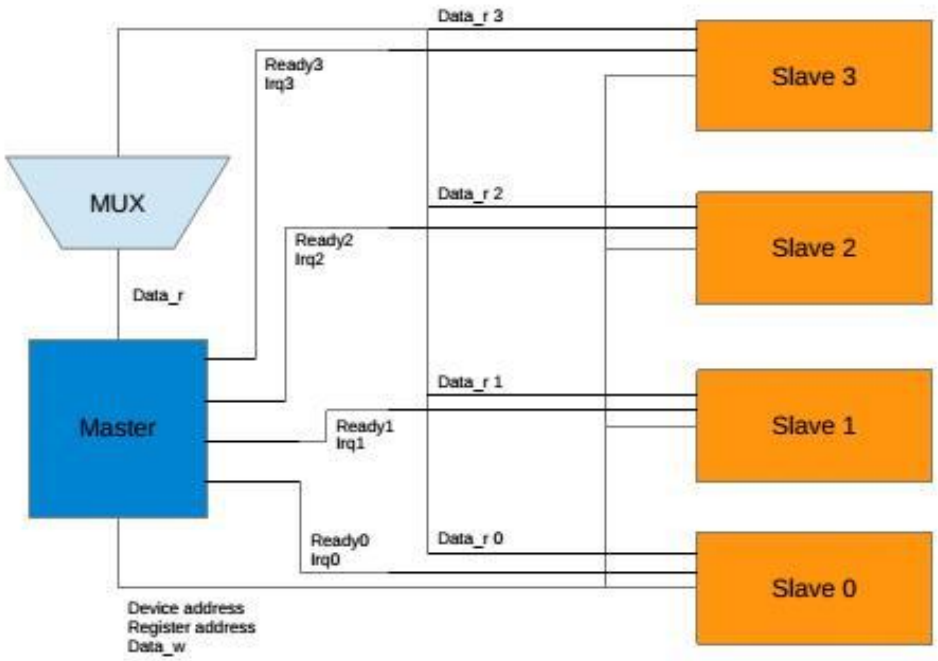


Figure 14. The internal bus

cation manager is the master, while bus-module interfaces are slave devices.

The bus is composed by the following signals:

- reset : this signal is used to reset all the registers of the interfaces
- clock : the bus is synchronous, synchronization is guaranteed by clock signal.
- device address [ ]: each bus-module interface of the pipeline is assigned a unique identification number. To address a device a proper device address must be set.
- register address [ ]: registers inside the bus-module interfaces are addressable.
- write enable : this signal establish if a read or write operation is performed.
- ready [ ]: one ready signal for each slave device. Ready is employed to synchronize master and slaves.
- data_w [ ]: these signals contain the input data
- data_r[ ]: these signals contain the output data. Each interface has its output data. Output data are multiplexed according to the device the communication manager is serving.
- irq [ ]: there is one irq signal for each slave device. Irq is raised by a device when output data have to be written to main memory. In particular, only stop bus-module interfaces can raise irq signal. When an irq signal is raised the arbiter of internal communication schedules the request.

When the communication manager wants to write a register(Figure 15) it set the address and write_enable signals properly, moreover the input data is output on data_w. As the addressed

device set its ready signal it is assumed that the input data is written correctly so that a new bus operation can start.

On the contrary, when the communication manager wants to read a register (Figure 16) it set

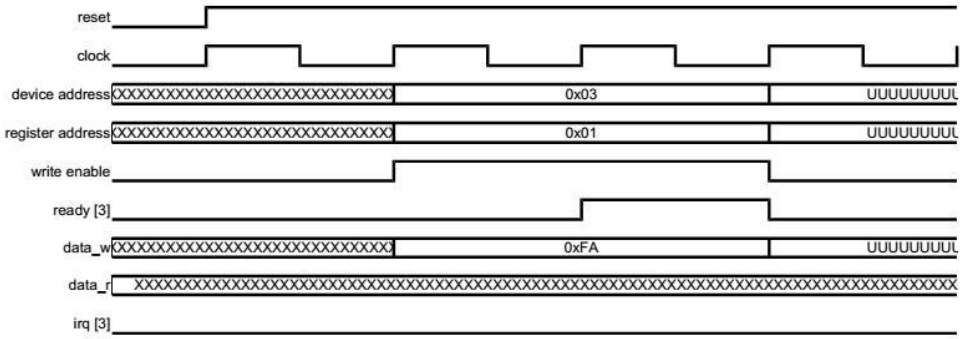


Figure 15. Write operation

the address and write_enable signals properly. As the addressed device set its ready signal it is assumed that the output data is on data_r signal. Then a new bus operation can start.

4.1.3 The configuration manager

The configuration manager is responsible for module configuration, but it is also responsible for bus-module interface programming. Actually, the configuration manager is the only manager which has the knowledge of a configuration completion. For this reason, it can configure interfaces so that synchronization of pipeline stages is achieved. In fact, when a module is con-

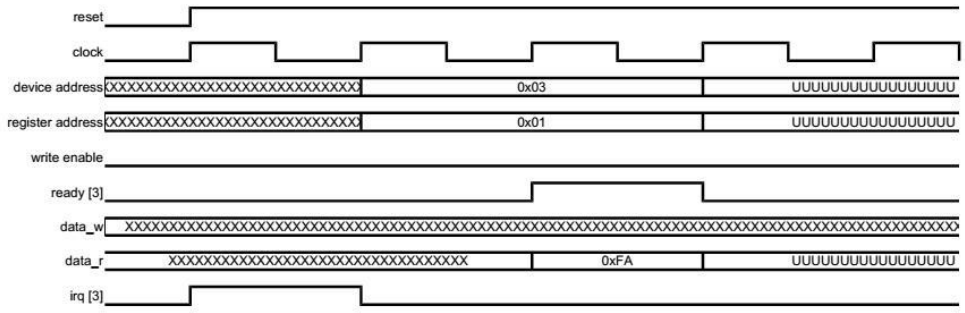


Figure 16. Read operation

figured, its input interface must be programmed to move from idle to a proper state, therefore module execution starts as fast as possible. It is important to notice that when a module is being configured, its adjacent interfaces must be idle since when configuration is performed it is impossible to predict the output signal of the under reconfiguration module. As a consequence, interfaces have to wait for the end of reconfiguration.

The configuration manager (Figure 17) is composed of a DMA to retrieve the bitstream of reconfigurations, an intelligent unit that controls the reconfiguration port, and a unit able to program all the bus-module interfaces.

4.1.4 The interface with the system

To interface with the processor, HI PROF provides two FIFOs, one for input and one for output. It is controlled by instruction written into an input FIFO that is accessible from the system bus. In particular, in the input FIFO contains instructions for both the communication manager and the reconfiguration manager. Moreover, information about the status of the

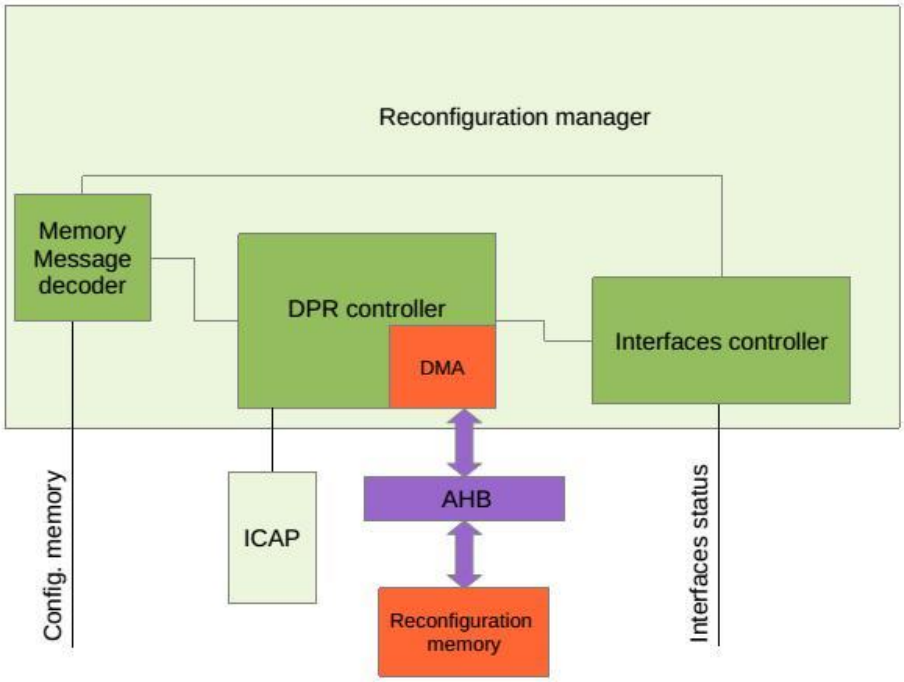


Figure 17. The configuration manager

accelerator and of hardware tasks are provided to the system by means of an output FIFO. Output FIFO can contain information about task execution, bus-module interfaces status and reconfiguration results. Finally, a register readable from the system bus contains information about FIFOs. This solution guarantees the ease of integrating an FPGA accelerator in a reconfigurable system. In fact, due to the advent of bus-based SoC, FPGA accelerator is designed as a bus peripheral. FIFOs and registers of the accelerator are memory mapped so that the processor and other devices can interact with it in an immediate way.

4.2 Performance of pipelined reconfiguration

In this section performance of pipelined reconfiguration is treated. At first, some definitions are given. Then, performance for the general case are illustrated. Finally, conclusions are provided focusing on benefits and drawbacks of pipelined reconfiguration.

Definition:*execution time, E_i. The execution time of i^{th} hardware module of the pipeline is the time needed by i^{th} module to complete, assuming its i^{th} pipeline stage is already configured.*

Definition:*reconfiguration time, R_i. The reconfiguration time of i^{th} hardware module of the pipeline is the time i^{th} pipeline stage needs to be configure. In particular, R is linear with respect to the module area.*

Definition:*i^{th} pipeline stage end time, T_i. i^{th} pipeline stage end time is the amount of time between the end of i^{th} pipeline stage execution and the time a the hardware task is started,*

assuming that all pipeline stages need to be configured.

Definition:*starting time offset of the i^{th} pipeline stage module, S_i. S_i is the amount of time between the configuration of first pipeline stage module and the start of i^{th} pipeline stage module execution, assuming that all pipeline stages need to be configured. As a consequence, $S_1 = 0$.*

Definition:*pipeline stage synchronization time, O. Pipeline stage synchronization time is the time required by a pipeline stage to synchronize with the following one. As described in previous section, for HI PROF, O is equal to 3 clock cycles in the worst case. O is assumed equal for each stage of the pipeline.*

In the general case, for pipelined reconfiguration:

$$S_n = \max(S_{n-1} + E_{n-1}, \sum_{i=1}^{n} R_i)$$

where

$$S_1 = R_1$$

and

$$T_n = S_n + E_n$$

Instead, in a traditional reconfigurable architecture, without pipelined reconfiguration, at first the whole module is configured then, after configuration is over, it is executed. As a consequence:

$$T_n = \sum_{i}^{n} R_i + \sum_{i}^{n} E_i$$

As a consequence it is possible to compute G, saved reconfiguration time:

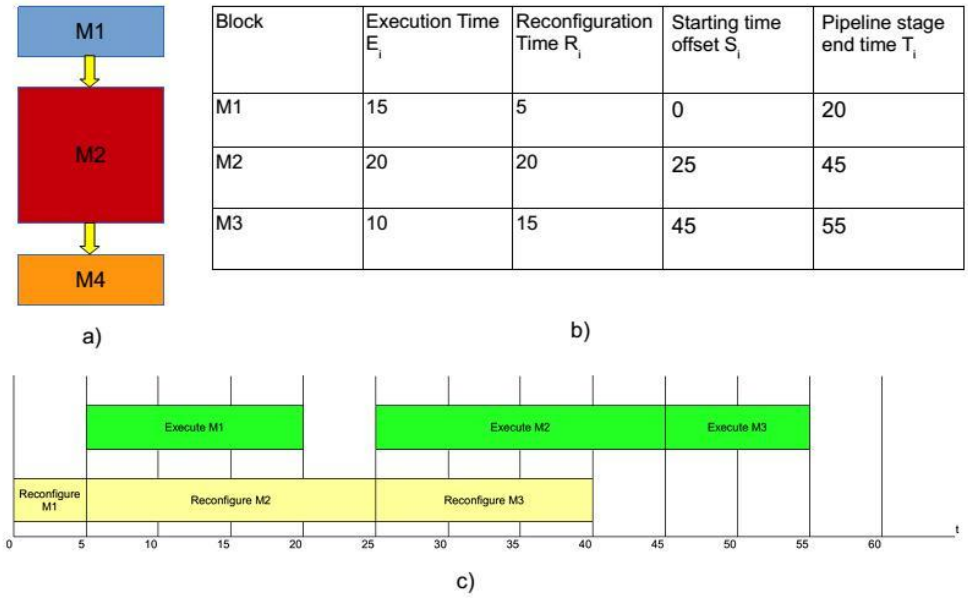
$$G = \sum_{i=1}^{n} E_i + R_i - T_n$$

A example is presented in Figure 18. In the example $G = 45 + 40 - 55 = 30$, so that we reduce configuration time overhead by 75%.

To reduce reconfiguration time overhead, T_n must be minimized, as a consequence S_n must be minimized too. This means that $\sum_{i=1}^{n} R_i$ should assume a lower value with respect to $S_{n-1} + E_{n-1}$ for every n. It can be noticed that, usually as reported in (23), (24) and (25), reconfiguration time alone occupies approximately 25 to 98 percent of the total execution time of a reconfigurable computing application. If this consideration is respected, the total reconfiguration time overhead is equal to R_1, as a consequence:

$$G = \sum_{i=2}^{n} R_i$$

Saved reconfiguration time overhead formula for the optimal case is a good results since it shows that, thanks to this technique increasing the number of pipeline stages produces an increase



Block	Execution Time E_i	Reconfiguration Time R_i	Starting time offset S_i	Pipeline stage end time T_i
M1	15	5	0	20
M2	20	20	25	45
M3	10	15	45	55

Figure 18. An example is reported. a) shows the sequential modules that must be implemented. b) reports execution and reconfiguration time for each pipeline stage. c) wants to illustrate how reconfiguration is masked

of saved time overhead. A further investigation occurs to establish a limit to the finesse of partitioning granularity, that is the number of stages of the pipeline for a module. In fact, in general case formula, computed above, it is assumed that O is negligible, while if O is considered:

$$T_n = S_n + E_n + O \times n$$

so that, for the optimal case:

$$G = \sum_{i=2}^{n} R_i - O \times n$$

This result shows the limit to the finesse of the main task partitioning, in fact, theoretically, pipelined reconfiguration does not provide any benefits if average reconfiguration time is lesser or equeal to pipeline stage synchronization time.

These results are promising, but some effort must be done to partition the main task in a optimal way. This topic is discussed later in next section.

Apart from advantages for performance, pipelined reconfiguration introduces area overhead. The sum of the area required by every sub-module of a task is larger than the area needed by a single module implementing the task. This is due to the fact that with pipelined reconfiguratoin some resources are replicated. Anyway, saved reconfiguration time remains unchanged if, for each sub-module, configuration time is smaller that execution time.

Another disadvantage of pipelined reconfiguration is the empoyment of static logic to imple-

ment architecture components since static logic occupies area that can be used by reconfigurable modules.

4.3 Guidelines for designing a hardware module systems

A designer who wants to design a hardware module suitable for pipelined reconfiguration, has to follow 2 steps:

1. Divide the main task into several sequential sub-tasks
2. Make the designed sub-modules suitable for the bus-module interfaces

Dividing a task into several sequential sub-tasks is carried out on the basis of pipelined reconfiguration performance. If the number of pipeline stages increases then resources replication has a growth too so that, in total, a larger reconfigurable area is required. As a consequence, augmenting the number of pipeline stages means increasing the total reconfiguration time. On the other hand, augmenting the number of pipeline stages means increasing the masked reconfiguration time. For example, main task, M, can be divided at most in four sequential sub-tasks: M1, M2, M3 and M4 (Figure 19.a). There are eight different possible implementations(Figure 19.b).
M1 and M4 do not require any resource that is common to other sub-task. On the contrary, M2 and M3 share some common resources. As a result, R2+R3 is greater than R23(Figure 20.a). In this example, the designer can choose among several possible implementation for M, however, they offer different performance (Figure 20.b).
Implementation F offers best performance, it outperforms A, the traditional implementation composed of a single huge block, 55 time units against 80. In fact, with F implementation, total

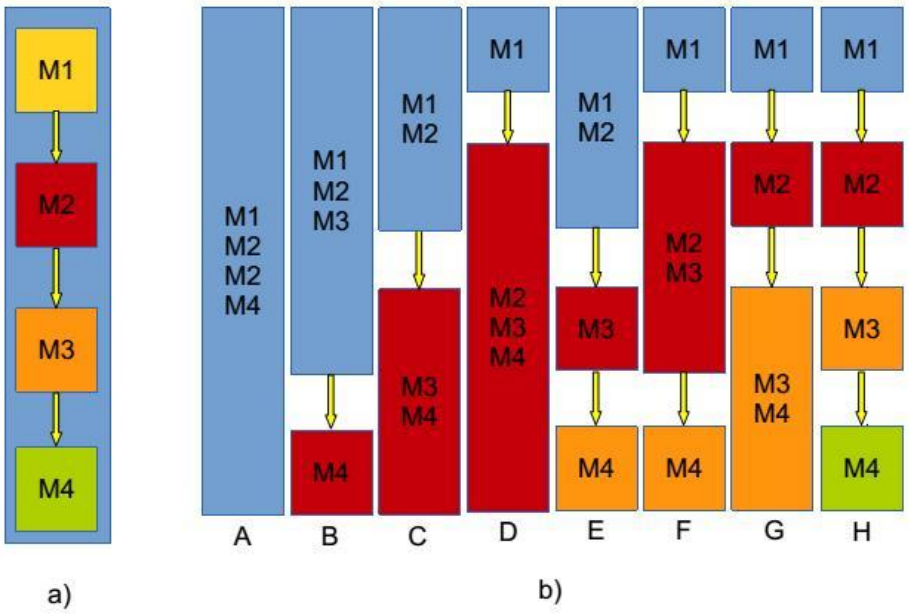
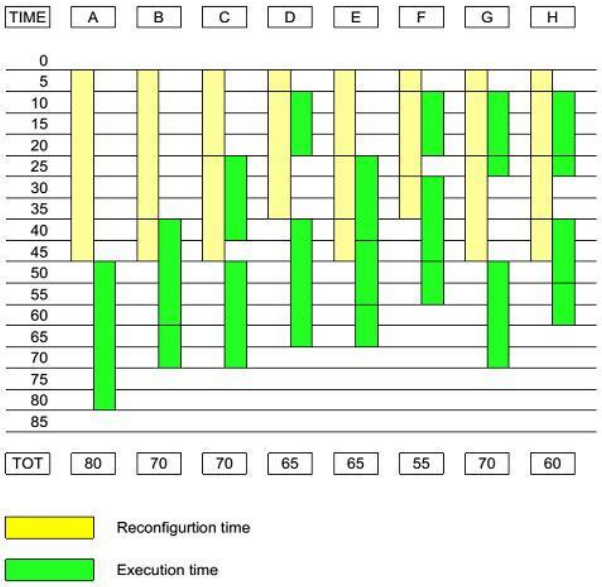


Figure 19. a) The sequential sub-task of the main module M b)All the possible partitionings for main task M

Block	Execution Time	Reconfiguration Time
M1,M2,M3,M4	45	35
M1,M2,M3	35	25
M2,M3,M4	30	30
M1,M2	20	20
M2,M3	20	20
M3,M4	25	25
M1	15	5
M2	5	15
M3	15	15
M4	10	10

a)



b)

Figure 20. a) The execution and reconfiguration time for every partitioned block b)time needed to complete is compared among all the possible implementation

time is reduced by almost 70%, underlining the effectiveness of the introduced idea of pipelined reconfiguration.

In conclusion, as can be deducted from Figure 20, in order to partition a task, the designer should:

- analyze which sub-tasks are worth to be partitioned in case they require common resources
- choose first sub-task with a small reconfiguration time, because it is always included into reconfiguration time overhead

In order to make pipeline stages working properly, the designer has to include in sub-modules all the signals that are necessary to communicate with bus-module interfaces. In particular, sub-modules must be synchronized according the data-flow. As shown in Figure 21.a, sub-modules, to communicate with input interface, include:

- an input reset signal
- an output reading data signal
- input data signals
- input data control signals

On the other end, to communicate with output interfaces, sub-modules have:

- an input signal indicating output interface is ready

- an output signal indicating output data are available
- output data signals
- output data control signals
- an output signal indicating computation of sub-module is over

In order to to implement a sub-module, the designer can implement a FSM to handle interface signals. In particular, the proposed FSM is composed by four states:

1. reset
2. reading
3. executing
4. writing

The state diagram is simple and it is illustrated in Figure 21b. In addition, in order to support services offered by the RHOS, the designer has to implement in its sub module extra registers(Figure 21a). These registers flow through the pipeline without any modifications and represent the hardware process information. For example there are registers indicating the address and size of output data, the stack memory address where to store data when a task is preempted. According to requirements, the designer has to decide which registers have to be included in the reconfigurable system.

In conclusion, design phase is extremely simple so that a designer has not to worry about

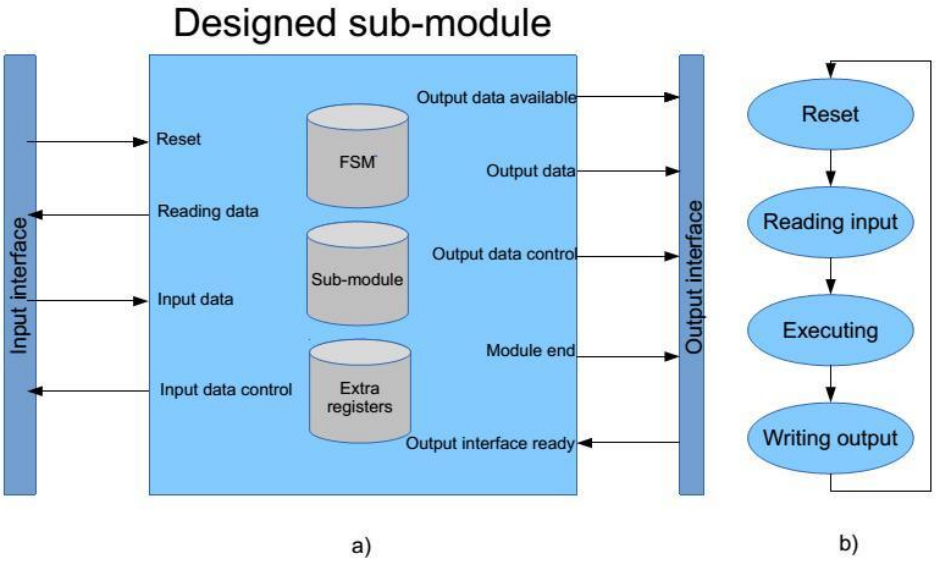


Figure 21. a) The signals required by sub-modules to be interfaced with bus-module interfaces, the FSM and extra registers b)The simple FSM embedded in sub-modules implementation

the integration of its sub-modules into the reconfigurable pipeline. Moreover, it can be stated that customization of the proposed architecture is extremely flexible.

4.4 Interface with the OS

In order to make a hardware process execute in HI PROF three main operations are needed. They are:

- Loading input data into the start bus-module interface
- Configuration of pipeline stages, that is loading bitstream into FPGA configuration memory

- Configuration of bus-module interfaces, that is setting properly the start, the intermediate and the stop interfaces

In particular, loading input data can be done in parallel with configuration of the first pipeline stage as input data and bitstreams are placed in different memories. Of course the stages of the pipeline have to be configured with an order that follows the data-flow: the start module is the first one, then intermediate modules and, at last, stop module bitstream is loaded. Moreover, bus-module interfaces have to be set synchronously with respect to the modules bitstream. In particular, bus-module interfaces can change state from idle to a proper one only after their output module is configured (with the exception of the stop interfaces that has no output module because output data have to be written into main memory).

When issuing one of the three operations a particular protocol must be adopted. Figure 22 shows the order that must be respected when writing commands into the input FIFO. The chosen protocol guarantees an easy interaction between the operating system and HI PROF. In fact, the operating system has only to write few words to make a task run on hardware.

For example, task M has to be executed (Figure 19) and the configuration the designer wants to employ is F(Figure 20). Input data memory address is 0x400FFA8B and the size of the input data is 10KB, while the output data memory address is 0x400FFBCB and size of output data is 3KB. The chosen start bus-module interface is interface 4. Bitstream of M1 is located at 0x0000BB00, of M2,M3 at 0x0000BF0A and of M4 at 0x0000C000. In this case, these are the requested steps (Figure 23) to follow in order to guarantee a correct data-flow.

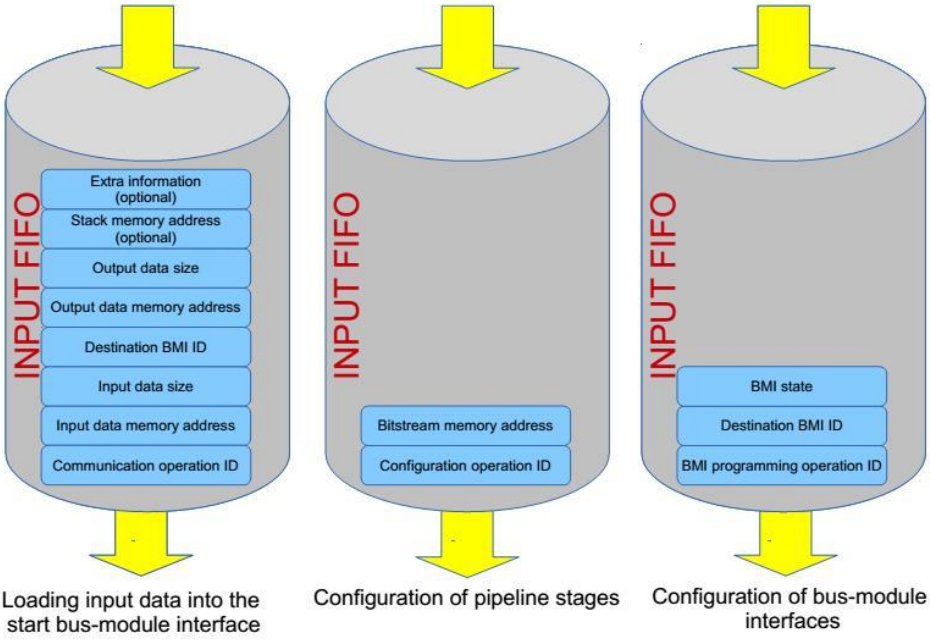


Figure 22. The different protocols must be respected according to the issued operation when writing commands into the input FIFO

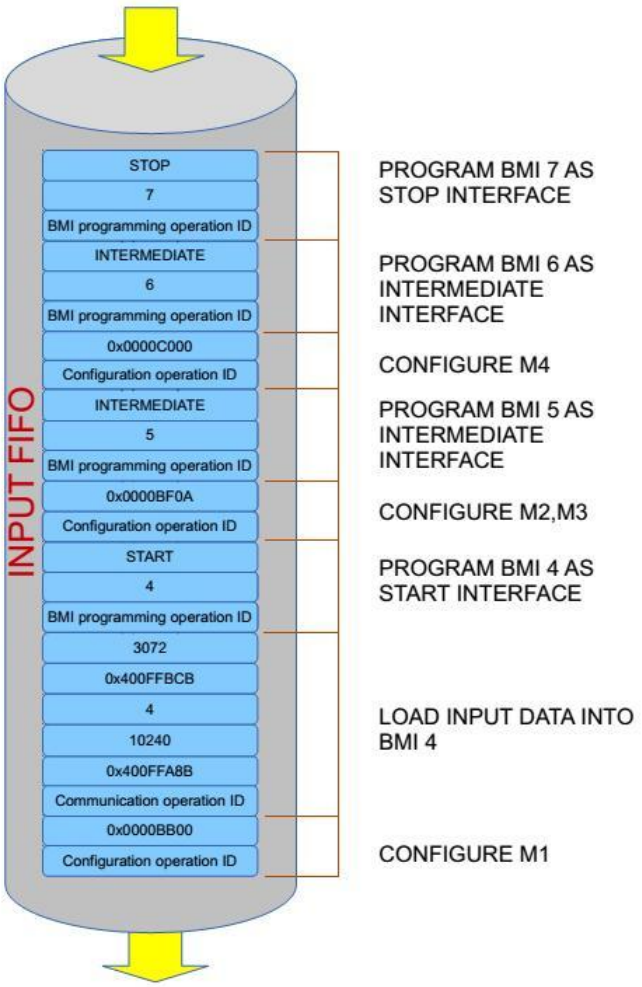


Figure 23. The commands to push into the FIFO when, in the example, task M must be executed

4.4.1 The scheduler

In such a scenario, the operating system carries out mapping and placement of a task. In this case, placement of hardware tasks consists of the choice of the starting stage of the pipeline(Section4.1). In order to perform mapping and placement, a ROS has to include a reconfigurable area monitor table and a bitstream memory position table. Reconfigurable area monitor table stores the current configurations in the reconfigurable area. In fact, the table contains the module configured every reconfigurable slot and the status of each stage of the pipeline. In particular, pipeline stages can be under configuration, running and cached. Under configuration stages are stages where configuration process is being carried out, as a consequence execution is has not started yet. Then, running stages are stages where configuration has already completed while execution is being performed. Finally, cached stages are stages where both configuration and execution are over and so that they can be rescheduled without configuration or configured again for other purposes. On the contrary, in the bitstream memory position table the memory addresses of the bitstreams associated to module implementations are stored. Actually, as a hardware module is scheduled, the scheduler must send the associated bitstream memory address to the reconfiguration manager.
Scheduler tables can be periodically updated by an OS daemon reading HI PROF output FIFO.

In conclusion, the propose reconfigurable accelerator offer an easy interaction between OS and FPGA accelerator.

4.4.2 The possibility to preempt a hardware task

Most of the architectures designed for reconfigurable computing do not offer the possibility to preempt a hardware task. In a scenario where every task has a priority, this feature constitute a force point. In fact, when a new hardware task arrives and no reconfigurable region is available, it would be useful to stop a task with a lower priority and execute the new one. Fixed-priority preemptive scheduler is widely employed in real-time systems, but it only applies to software. Thanks to pipelined reconfiguration, hardware tasks can have a finer granularity without loss of performance so that they are suitable for preemption. To preempt a task, it is only needed to program as stop interface the interface of the last configured pipeline stage. Moreover, a memory address where to save temporary result should be indicated as a hardware process parameter.

4.4.3 Virtual reconfiguration space

Pipelined reconfiguration introduces a new concept in the field of reconfigurable systems. Virtual reconfiguration is a technique to manage reconfiguration that maps virtual hardware modules to physical hardware modules. In particular, physical hardware modules are represented by sub-modules while virtual modules are represented by hardware task module which can be composed by several physical modules. Hardware processes have the perception of just the virtual module, while the operating system has the perception of both virtual and physical modules. For example, if a system is designed to execute two tasks, A and B. Task A consists of the division between two numbers, while task B consists of the average of ten numbers, at most. The designer can choose to implement two hardware modules: an accumulator and a hardware

division unit. When A and B are executed the operating system configures the two physical modules: the accumulator and the hardware divider. Process A has the perception that a divider is configured, while process B has the perception that a hardware average computer is configured. For process A physical module and virtual module are equal, on the contrary, for process B they are different. To achieve virtual reconfiguration management, the operating system has to know which physical modules are required by virtual modules. In conclusion, virtual reconfiguration can be thought an a technique to exploit module reusability in a pipelined reconfiguration architecture.

CHAPTER 5

EXPERIMENTAL RESULTS

In this chapter experimental results are provided for HI PROF. In particular a reconfigurable system is set up in a SoC by means of a Virtex 4 FPGA embedding LEON3 processor. The goal of this case study is to run three image processing reconfigurable IP-core on HI PROF focusing on the design process, virtual reconfiguration space and performance. Special attention will be given to bitstream size, reconfiguration time and area overhead.

5.1 Building the reconfigurable system

5.1.1 Target FPGA

FPGA device utilized in this work is VIRTEX 4 XC4VLX100-10FF1513C that provides about 110,600 logic cells organized in 50,000 slices; it provides many hard macro blocks such as Digital Signal Processing (DSP) blocks, Random Access Memory (RAM) blocks, Digital Clock Managers (DCMs) to allow designers to realize a complete systemon-chip (SoC) in a quick and easy way. Figure 24 summarizes available resources in XC4VFX12 FPGA device and next sub-sections show descriptions of some important blocks of this FPGA.

5.1.2 Target board

Virtex 4 FPGA device utilized in this work is mounted on an evaluation board that provides some facilities to realize and test a complete design, configuring FPGA through JTAG interface.

Logic cells	Slices	Distributed RAM	DSP slices	Block RAM	DCMs	I/Os
110592	49152	768	96	240	12	960

Figure 24. Available resources in XC4VFX12 FPGA

(Figure) shows front-side of GR-CPCI-XC4V board. information about this board from (54) are reported below:

> GR-CPCI-XC4V board (Figure 25) is a Compact PCI format development board which has been developed in cooperation with Gaisler Research especially to support the early development and fast prototyping of LEON systems. Although suitable for general purpose Virtex 4 designs, the incorporation of on-board volatile and non-volatile memory interfaces, together with serial and ethernet interfaces makes this board ideal for implementing SoC designs.

5.1.3 LEON3 and GRLIB

As stated in (55):

> The LEON3 processor is a synthesizable VHDL model of a 32-bit processor compliant to the SPARC V8 architecture. It is provided in full source code under the GNU LGPL license, allowing free and unlimited use in both research and commercial applications.

The designed reconfigurable system is a bus-based system and it is composed of the LEON3 processor, a 256Mbyte SDRAM and the hardware accelerator. The cores are interfaced using the AMBA 2.0 AHB bus protocol supporting the IP core plug&play method provided in the Gaisler

Figure 25. GR-CPCI-XC4V board

Research IP library (GRLIB). As can be seen in Figure 26 HI PROF has been plugged&played into the AMBA bus.

5.2 Cases of study

5.2.1 FEMIP

FEMIP (56) is a high performance FPGA-based IP-core to hardware accelerate the Features Extraction and Matching (FEM) tasks. It is employed for Video Based Navigation in space-applications. In particular, it is composed of five main sequential blocks(Figure 27):

- 7x7 gaussian filter
- a derivative filter

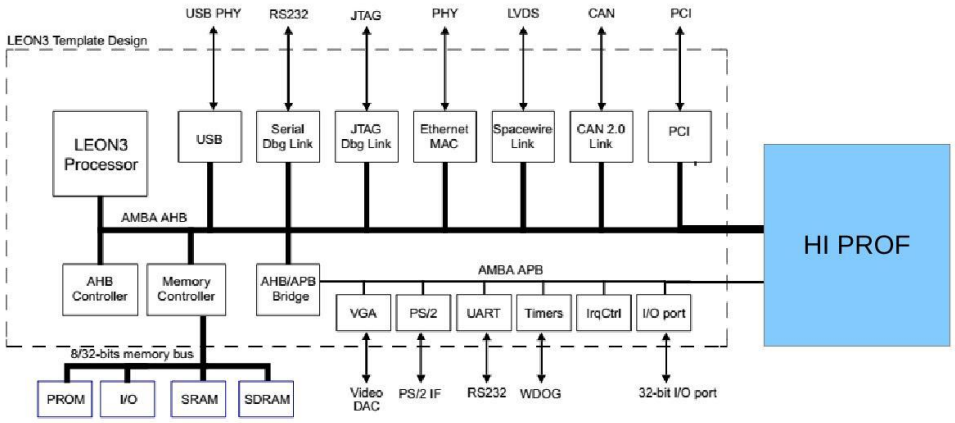


Figure 26. The bus-based SoC embedding HI PROF

- a harris feature extractor filter
- a non-maxima suppressor (NMS) filter
- a feature matcher filter

Task partitioning was performed by dividing main task into five sequential sub-tasks, corresponding to the filter that must be applied (Figure 27). To facilitate the design of a sub-modules suitable for the proposed hardware infrastructure, a VHDL template file has been created. In particular, this file contains: extra registers, the FSM that allows communication and synchronization with the interface. Moreover, the FSM is also responsible for starting, suspending and stopping the filter entities by means of reset and enable signals. In order to adapt FEMIP cores to HI PROF it was just necessary to move registers employed for handling input and output data from filters to bus-module interfaces. This operation took very little time and did not require

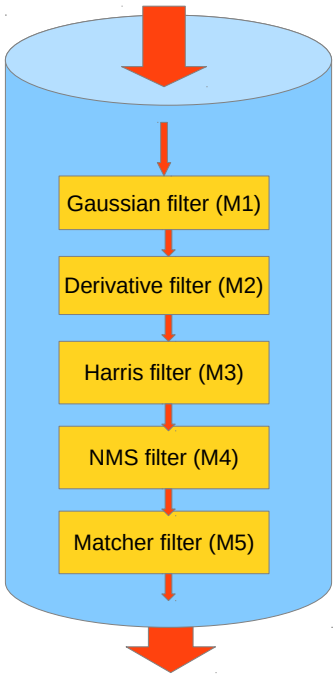


Figure 27. FEMIP sequential task subdivision

a lot of efforts, so that the ease of adapting an IP-core to HI PROF architecture has been proved.

We provide two solution to host FEMIP into HI PROF. The first implementation consists of the employment of 5 reconfigurable stages, that is each filter is associated to a unique pipeline stage. As a result the area required for FEMIP implementation is the sum of the area occupied by each filter. With this solution we have five distinct modules in the physical reconfiguration space, instead, in the virtual reconfiguration space many more IP-cores are supported. For example, in the virtual reconfiguration space there can be FEMIP, a Gaussian filter, a Derivative filter, a Gaussian+Derivative filter and so on. In practice, each of the virtual IP-cores can be accessed independently.

The second implementation, on the contrary is composed of pipeline 4 stages. This solution provide a wrap-around connection of the pipeline, so that there is not a starting pipeline stage. To make FEMIP run on the 4-stages pipeline it is necessary that one of the stages is shared by two modules because there are 4 stages while FEMIP requires 5 of them. In particular, as computation of first stage is over, the bitstream of fifth module is configured in it. This is achievable because of the sequentiality of operations. This implementation introduces a substantial advantage in terms of required area. Since one stage is shared, the total area is the sum of the four stages. On the contrary, this provokes drawbacks in the virtual reconfiguration space: there are only four physical modules, this means that the system support less virtual IP-cores than in the 5-stage pipeline.

Before giving experimental results, it is necessary to focus on the execution of FEMIP sub-modules. Harris filter, NMS filter and matcher filter are fully sequential task, while Gaussian filter, derivative filter and Harris filter execution is different from a proper sequential execution. In fact, they are a mix of sequentiality and parallelism. First module, after a brief period starts writing output data into a buffer. However, when the buffer is full execution of first module is suspended until the second one starts reading it. In an already configured environment, second module executes immediately while reading operation is started as the buffer is written. On the contrary, in a reconfigurable pipeline scenario, execution and reading operation of second module start only after configuration(Figure 28). The same happen when data are pushed into third module, this time second module is suspended to wait for M3, moreover, as a consequence, M1 waits for M2. As a result all already reconfigured modules have to wait for reconfiguration.

In this scenario, we can complete the definition of end of a hardware task given in Section 4.2 by introducing a new variable: V_i. V_i : time to first data available for i_{th} pipleline stage. In this sense, V_i determines the amount of time needed for a pipeline stage to address the following stage in the pipeline.

As a consequence:

$$S_n = max(S_{n-1} + V_{n-1}, \sum_{i}^{n} R_i)$$

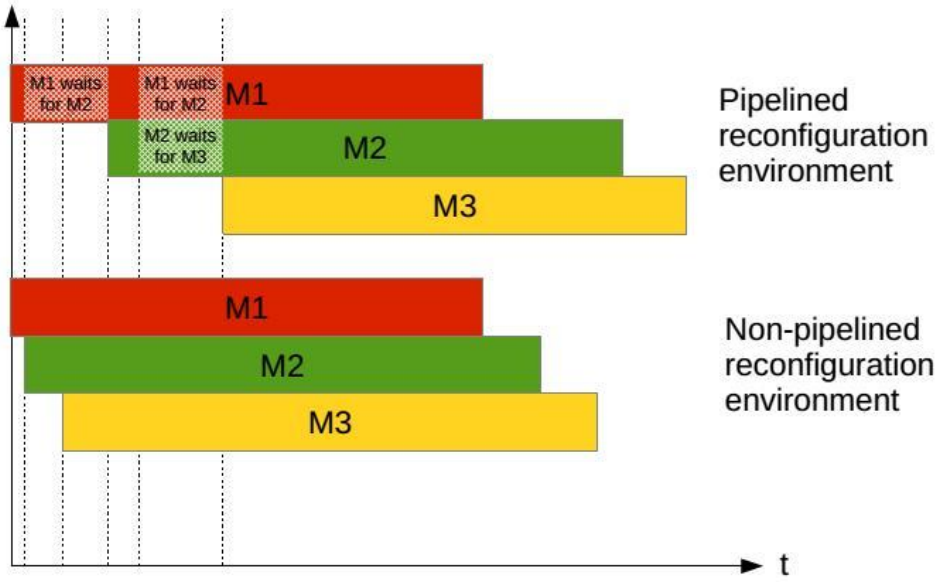


Figure 28. Pipleined reconfiguration limitations

and

$$T_n = S_n + E_n$$

Of course, for a sequentail task:

$$V_i = E_i$$

Figure 29 shows what happens for a 4-stages HI PROF imlpementation when FEMIP is run.

5.2.1.1 Obtained results

In this section reconfiguration time overhead and area employment are analyzed and compared with a traditional block architecture implementation where FEMIP is designed as a

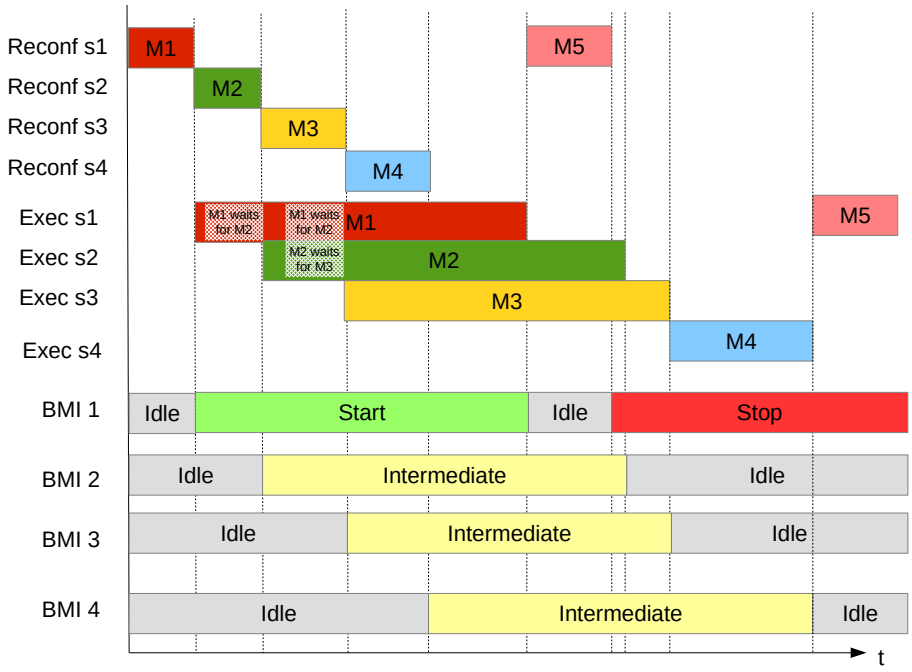


Figure 29. What happens inside HI PROF for a 4-stages pipeline

unique module. Obtained results are reported in the following tables(Figure 30-Figure 32). Figure 30 shows that adopting this technique it is possible to save 20% of reconfiguration

	Bitstream size [Bytes]	Bitstream size [32-bit words]	ReconfigurationTime [ms]	Execution Time [ms]	First data available after [ms]	Masked reconfiguration time [ms]	Masked reconfiguration time ratio [%]	Reconfiguration time overhead [ms]
gauss	171872	42968	0,651030303	17,13	0,03		0	0,651030303
deriv	103320	25830	0,3913636364	17,521	0,0268	0,03	7,6655052265	0,3613636364
harris	194832	48708	0,738	17,5212	17,5212	0,0268	3,6314363144	0,7112
nms	49200	12300	0,1863636364	5,383	5,383	0,1863636364	100	0
matcher	67520	16880	0,2557575758	5,5195	5,5195	0,2557575758	100	0
5-stages HI PROF	586744	146686	2,2225151515	28,34	28,34	0,4989212121	22,4484954256	1,7235939394

	Bitstream size [Bytes]	Bitstream size [32-bit words]	ReconfigurationTime [ms]	Execution Time [ms]	First data available after [ms]	Masked reconfiguration time [ms]	Masked reconfiguration time ratio [%]	Reconfiguration time overhead [ms]
gauss	171872	42968	0,651030303	17,13	0,03		0	0,651030303
deriv	103320	25830	0,3913636364	17,521	0,0268	0,03	7,6655052265	0,3613636364
harris	194832	48708	0,738	17,5212	17,5212	0,0268	3,6314363144	0,7112
nms	49200	12300	0,1863636364	5,383	5,383	0,1863636364	100	0
matcher	171872	42968	0,651030303	5,5195	5,5195	0,651030303	100	0
4-stages HI PROF	691096	172774	2,6177878788	28,34	28,34	0,8941939394	34,1583803119	1,7235939394

	Bitstream size [Bytes]	Bitstream size [32-bit words]	ReconfigurationTime [ms]	Execution Time [ms]	First data available after [ms]	Masked reconfiguration time [ms]	Masked reconfiguration time ratio [%]	Reconfiguration time overhead [ms]
Femip	572032	143008	2,1667878788	28,34	28,34	0	0	2,1667878788

Figure 30. Reconfiguration time overhead and bitstream size are analyzed for the traditional, the 4-stages and the 5-stages implementations

overhead despite the number of pipeline stages. This is a positive results because it underlines that reconfiguration time overhead has been reduced, although, as mentioned before, first three modules are introducing some problems. In fact, reconfiguration time overhead is mainly accumulated during the reconfiguration of Gaussian, derivative and Harris filters. However, reconfiguration overhead is completely masked for NMS and matcher filter.

Reducing configuration time overhead comes at a price(Figure 31). In fact, as analyzed in the

	Associated stage	Required SLICE	Required BRAM
gauss	1,00	2560,00	8,00
deriv	2,00	1344,00	8,00
harris	3,00	3456,00	0,00
nms	4,00	360,00	4,00
matcher	5,00	650,00	6,00
5-stages HI PROF	1,2,3,4,5	8370,00	26,00

	Associated stage	Required SLICE	Required BRAM
Gauss & matcher	1,00	2560,00	8,00
deriv	2,00	1344,00	8,00
harris	3,00	3456,00	0,00
nms	4,00	360,00	4,00
4-stages HI PROF	1,2,3,4	7720,00	20,00

	Required SLICE	Required BRAM
Femip	8051,00	20,00

Figure 31. Resources employment is analyzed for the traditional, the 4-stages and the 5-stages implementations

previous section, for the 4-stages implementation, as a module is reconfigured we have restrictions concerning the number of virtualized IP-cores. Moreover, despite a slight decrease of required SLICE, there is a remarkable growth of the bitsream size, in fact it is 20% larger. On the other hand, for the 5-stages, there is a slight increase of the required SLICE and BRAM so that the bitstream size augments too.

	Saved reconfiguration time overhead [ms]	Saved reconfiguration time overhead ratio [%]	Saved reconfiguration SLICES	Saved reconfiguration SLICES ratio [%]	Saved reconfiguration BRAM	Saved reconfiguration BRAM ratio [%]	Bitstream size increase [Bytes]	Bitstream size increase ratio [%]
5-stages HI PROF	0,44	20,45	-319,00	-3,96	-6,00	-0,07	14712,00	2,57
4-stages HI PROF	0,44	20,45	331,00	4,11	0,00	0,00	119064,00	20,81

Figure 32. These are FEMIP benefits and drawbacks intoduced by HI PROF

5.2.2 SAFE

SAFE: a Self Adaptive Frame Enhancer FPGA-based IP-core for real-time applications. It is employed for image processing and it carries out images enhancement (57) by providing more defined and contrasted frames so that high precision feature extraction is assured.

Image enhancement can be performed in intensity, spatial or frequency domains. Among the available techniques, the ones that better improve FEM algorithms are those working in the intensity domain. Histogram Equalization and Histogram Stretching (58) proved to be two of the most effective Image Enhancement Techniques (IET).
An image histogram is a graphical representation of the tonal distribution in a digital image. It plots the number of pixels in the image (vertical axis) that present a particular intensity value (horizontal axis).

Histogram Equalization (HE), in particular Linear HE, changes the intensity value of each pixel to produce a new image with a more uniform image histogram (i.e. the image covers most

of the brightness dynamic range). A better distributed histogram increases the image contrast, especially if the original image has close intensity values. The method is useful in images with backgrounds and foregrounds that are both bright or both dark, since these images are characterized by narrow and smoothed histograms.

The Histogram Stretching (HS) technique is based on redistribution of the pixel intensities to spread their values on the entire spectrum of colors. It increases the contrast among pixels, but it becomes ineffective when the input image features a wide histogram.

According to input thresholds and image statistics, SAFE is able to select automatically the best frame enhancement technique (i.e., HS or HE).

SAFE is composed of three sequential main blocks (Figure 33): the Histogram Calculator, the Histogram Analyzer and the Equalizer/Stretcher. The Histogram Calculator computes the histogram of the input image, providing the value of each bar. It simply counts the occurrence of each pixel intensity, in order to compute the bar values. The Histogram Analyzer analyzes the image histogram in order to select the best IET to be applied. It scans the histogram to find the minimum and maximum intensities and the maximum difference between two consecutive bar values. By comparing this two quantities with input thresholds, it selects the best IET. The Equalizer / Stretcher performs both HE and HS on the input image, but it provides in output the image enhanced by the algorithm selected by the Histogram Analyzer.

To adapt SAFE to HI PROF the partitioning consists of the simple division into sequential blocks: the first is composed of both the Histogram Calculator and the Histogram Analyzer,

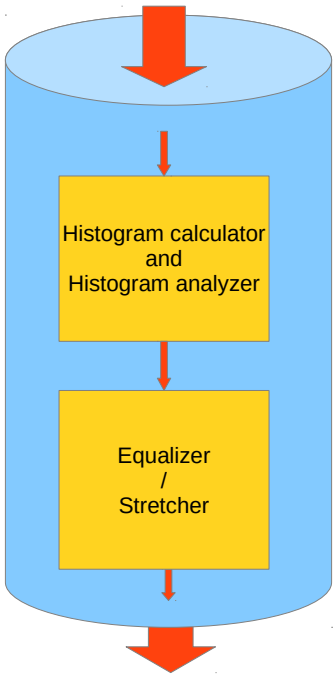


Figure 33. SAFE partitioning

while the second is the Equalizer/Stretcher. The execution of these blocks is completely sequential as the second can start as the first is over.

For this case study two solutions are provided. The first solution consists of the creation of two FPGA partitions which are tailored to the resource requirements of each block, while for the second one the resources of FPGA partitions are identical for every block. These two implementation are then compared with respect to the traditional implementation of SAFE.

5.2.2.1 Obtained results

Obtained results are reported in the following tables (Figure 34, Figure 35, Figure 36).

	Bitstream size [Bytes]	Bitstream size [32-bit words]	Reconfiguration-Time [ms]	Execution Time [ms]	First data available after [ms]	Masked reconfiguration time [ms]	Masked reconfiguration time ratio [%]	Reconfiguration time overhead [ms]
HC+HA	56416	14104	0.20	3.61	3.61	0.00	0.00	0.20
E/S	82880	20720	0.29	3.61	0.00	0.29	100.00	0.00
Tailored 2-stages HI PROF	139296	34824	0.48	7.22	7.49	0.29	59.50	0.20

	Bitstream size [Bytes]	Bitstream size [32-bit words]	Reconfiguration-Time [ms]	Execution Time [ms]	First data available after [ms]	Masked reconfiguration time [ms]	Masked reconfiguration time ratio [%]	Reconfiguration time overhead [ms]
HC+HA	70356	17589	0.24	3.61	3.61	0.00	0.00	0.24
E/S	70356	17589	0.24	3.61	0.00	0.24	100.00	0.00
Non-tailored 2-stages HI PROF	140712	35178	0.49	7.22	7.49	0.24	50.00	0.24

	Bitstream size [Bytes]	Bitstream size [32-bit words]	Reconfiguration-Time [ms]	Execution Time [ms]	First data available after [ms]	Masked reconfiguration time [ms]	Masked reconfiguration time ratio [%]	Reconfiguration time overhead [ms]
SAFE	105780	26445	0.37	14.98	14.98	0.00	0.00	0.37

Figure 34. Reconfiguration time overhead and bitstream size are analyzed for traditional, tailored and non-tailored implementations

	Associated stage	Required SLICE	Required BRAM	Required DSP
HC+HA	1	512	8	0
E/S	2	576	4	20
Tailored 2-stages HI PROF	1,2,3	1088	12	20

	Associated stage	Required SLICE	Required BRAM	Required DSP
HC+HA	1	575	8	20
E/S	2	576	8	20
Non-tailored 2-stages HI PROF	1,2,3	1151	16	40

	Associated stage	Required SLICE	Required BRAM	Required DSP
SAFE	1	960	12	20

Figure 35. Resources employment is analyzed for traditional, tailored and non-tailored implementations

In both the implementations it is possible to save reconfiguration time despite an increase of required resources and bitstream size (Figure 36). In particular, for the tailored implementation it is possible to save almost 50% of reconfiguration time overhead, however there is a slight growth of required SLICES. For the non-tailored implementation, on the opposite, it is possible to save more than 30% of reconfiguration time overhead, while there is a small rise of BRAM employment and a double request of DSP as drawbacks.

5.2.3 AIDI

AIDI, as FEMIP, is an IP core employed for image processing. It carries out the task of removing noise from images (59). In fact, it is an adaptive image denoising IP-core.

The core first estimates the level of noise in the input image. It then applies an adaptive Gaussian smoothing filter to remove the estimated Gaussian noise. The filtering parameters

	Saved reconfiguration time overhead [ms]	Saved reconfiguration time overhead ratio [%]	Saved reconfiguration SLICES	Saved reconfiguration SLICES ratio [%]	Saved reconfiguration BRAM	Saved reconfiguration BRAM ratio [%]
Tailored 2-stages HI PROF	0.17	46.67	-128	-13.33	0	0.00
Non-tailored 2-stages HI PROF	0.12	33.49	-191	-19.90	-4	-33.33

	Saved reconfiguration DSP	Saved reconfiguration DSP ratio [%]	Bitstream size increase [Bytes]	Bitstream size increase ratio [%]
Tailored 2-stages HI PROF	0	0.00	33516	31.68
Non-tailored 2-stages HI PROF	-20	-100.00	34932	33.02

Figure 36. These are SAFE benefits and drawbacks intoduced by HI PROF

are computed on- the-fly, adapting them to the level of noise of the current image. Furthermore, the filter uses local image information to discriminate whether a pixel belongs to an edge in the image or not, preserving it for subsequent edge detection or image registration algorithms.

AIDI is composed of three blocks: the Noise Variance Estimator (NVE), the Local Variance Estimator (LVE) and the Adaptive Gaussian Filter (AGF). NVE estimates Gaussian noise variance, while LVE computes the local variance of each pixel of the input image, then AGF outputs pixel-by-pixel the denoised image depending on the results of NVE and LVE (Figure 37).
To adapt AIDI to HI PROF the partitioning consists of the simple division into the three main block: NVE, LVE and AGF (Figure 37). The first module that is executed is the NVE, as its computation is over, then LVE and AGF are then executed in a pipeline fashion, so that at

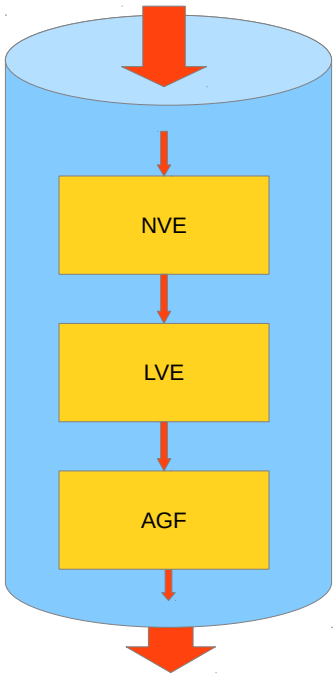


Figure 37. AIDI partitioning

each clock cycle a denoised pixel is provided. For this reason the execution of AGF starts when the first output data of the LVE are available.

In a virtual reconfiguration scenario it is important that each reconfigurable slot can be configured with several partitions, so that these partitions must have the same resources. In this way, the scheduler can choose among all reconfigurable slots available for the task that is taken into account at that moment. If there is an implementation for many FPGA block, it is more likely that there is an available slot for a task that has to be scheduled and also a larger number of tasks can be mapped to hardware.

For these reasons, I provide two different implementation for AIDI. The first one is a 3-stages implementation: every module is assigned to a specific stage whose available resources depend on the requests of the core. The second one, on the contrary, consists of a 2-stages implementation. In this case the the resources of each FPGA slot are identical and every module can be configured in every slot.

5.2.3.1 Obtained results

Obtained results are reported in the following tables (Figure 38, Figure 39, Figure 40).
In both the implementations it is possible to save reconfiguration time despite an increase of required resources and bitstream size (Figure 40). In particular, for the 3-stages implementation it is possible to save more than 90% of reconfiguration time overhead, however there is a slight growth of required BRAM. For the 2-stages implementation, on the opposite, it is possible to

	Bitstream size [Bytes]	Bitstream size [32-bit words]	Reconfiguration Time [ms]	Execution Time [ms]	First data available after [ms]	Masked reconfiguration time [ms]	Masked reconfiguration time ratio [%]	Reconfiguration time overhead [ms]
NVE	82328	20582	0.21	7.49	7.49	0.00	0.00	0.21
LVE	414262	103566	1.04	7.49	0.00	1.04	100.00	0.00
AGF	400488	100122	1.00	7.49	0.00	1.00	100.00	0.00
3-stages HI PROF	897078	224270	2.24	14.98	7.49	2.04	90.82	0.21

	Bitstream size [Bytes]	Bitstream size [32-bit words]	ReconfigurationTime [ms]	Execution Time [ms]	First data available after [ms]	Masked reconfiguration time [ms]	Masked reconfiguration time ratio [%]	Reconfiguration time overhead [ms]
NVE	441816	110454	1.10	7.49	7.49	0.00	0.00	1.10
LVE	441816	110454	1.10	7.49	0.00	1.10	100.00	0.00
AGF	441816	110454	1.10	7.49	0.00	1.10	100.00	0.00
2-stages HI PROF	1325448	331362	3.31	14.98	7.49	2.21	66.67	1.10

	Bitstream size [Bytes]	Bitstream size [32-bit words]	ReconfigurationTime [ms]	Execution Time [ms]	First data available after [ms]	Masked reconfiguration time [ms]	Masked reconfiguration time ratio [%]	Reconfiguration time overhead [ms]
AIDI	875328	218832	2.19	14.98	14.98	0.00	0.00	2.19

Figure 38. Reconfiguration time overhead and bitstream size are analyzed for traditional, 3-stages and 2-stages implementations

	Associated stage	Required SLICE	Required BRAM
NVE	1	1216	4
LVE	2	6528	12
AGF	3	7104	0
3-stages HI PROF	1,2,3	14848	16

	Associated stage	Required SLICE	Required BRAM
NVE	1	7104	12
LVE	2	7104	12
AGF	3	7104	12
2-stages HI PROF	1,2,3	14208	24

	Associated stage	Required SLICE	Required BRAM
AIDI	1	14656	14

Figure 39. Resources employment is analyzed for traditional, 3-stages and 2-stages implementations

save almost 50% of reconfiguration time overhead, while there is a remarkable rise of BRAM employment as a drawback.

	Saved reconfiguration time overhead [ms]	Saved reconfiguration time overhead ratio [%]	Saved reconfiguration SLICES	Saved reconfiguration SLICES ratio [%]	Saved reconfiguration BRAM	Saved reconfiguration BRAM ratio [%]	Bitstream size increase [Bytes]	Bitstream size increase ratio [%]
3-stages HI PROF	1.98	90.59	-192	-1.31	-2	-14.29	21750	2.48
2-stages HI PROF	1.08	49.53	448	3.06	-10	-71.43	450120	51.42

Figure 40. These are AIDI benefits and drawbacks intoduced by HI PROF

CONCLUSIONS

In this thesis work new hardware interface for an hybrid system have been presented. Experimental results show the effectiveness of HI PROF under many points of views. First of all, it is powerful for the reduction of reconfiguration time overhead by means of pipelined reconfiguration. It must be noticed that one of the reasons that make designer avoid reconfigurable systems is the loss of performance due to reconfiguration time overhead. Moreover, this innovative idea do not influence any task scheduling policies. So that it is compatible with scheduling techniques that aims at the reduction of reconfiguration overhead, such as prefetching (23).
The possibility of having a reconfigurable system with such performance in a single FPGA should attract real-time embedded systems designer. In fact drawbacks in terms of replicated resources seem to be negligible.
Finally, nevertheless RHOS-level benefits have not received a deep analysis int this work, we can glimpse the potentiality of the two services exported by HI PROF. Hardware task preemption, in a real-time scenario, aims to introduce software flexibility in hardware so that hybrid systems are promising to outperform CPUs based solution in real-time applications. Virtual reconfiguration space laeds to another advantage. It offers the possibility of having more IP-cores than the number of configured modules.

Since the results obtained with HI PROF are promising, a further investigation concerning efficient techniques for partitioning sequential tasks can highlight the real potential of pipelined reconfiguration, aside from contributing to improve the presented work. In particular, it would be nice to give estimates of required area and reconfiguration overhead so that an optimal number of pipeline stages can be identified according to designed IP-sores. Future works may also be devoted to overcome pipelined reconfiguration deficiencies, that is the case of firsts blocks of FEMIP where concurrent tasks cannot fully benefit of reconfiguration time overhead reduction.

On the other side, it would be interesting to address my efforts towards hardware task scheduling, as hardware task preemption and virtual reconfiguration have created new degrees of flexibility and no research has been carried out in these fields.

CITED LITERATURE

1. Todman, T., Constantinides, G., Wilton, S. J. E., Mencer, O., Luk, W., and Cheung, P. Y. K.: Reconfigurable computing: architectures and design methods. Computers and Digital Techniques, IEE Proceedings -, 152(2):193–207, 2005.

2. So, H. K.-H. and Brodersen, R. W.: BORPH: An Operating System for FPGA-Based Reconfigurable Computers. Doctoral dissertation, EECS Department, University of California, Berkeley, Jul 2007.

3. Steiger, C., Walder, H., and Platzner, M.: Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks. IEEE Trans. Computers, 53(11):1393–1407, 2004.

4. Iturbe, X., Benkrid, K., Hong, C., Ebrahim, A., Torrego, R., Martinez, I., Arslan, T., and Perez, J.: R3tos: A novel reliable reconfigurable real-time operating system for highly adaptive, efficient, and dependable computing on fpgas. IEEE Transactions on Computers, 62(8):1542–1556, 2013.

5. Estrin, G.: The fixed plus variable structure computer. Proceedings of the Western Joint Computer Conference, May 1960.

6. Estrin, G. et al.: Parallel processing in a restructurable computer system. IEEE Transactions on Electronic Computers, 1963.

7. Estrin, G. et al.: Automatic assignment of computations in a variable structure computer system. IEEE Transactions on Electronic Computers, 1963.

8. Estrin, G. et al.: Organization of a "fixed-plus-variable" structure computer for eigenvalues and eigenvectors of real symmetric matrices. ournal of the ACM9, 1962.

9. Saxe, T. et al.: Less is more with fpgas. EE Times, 2004.

10. Hauck, S. and DeHon, A.: Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2007.

CITED LITERATURE (continued)

11. Stitt, G., Vahid, F., and Nematbakhsh, S.: Energy savings and speedups from partitioning critical software loops to hardware in embedded systems. ACM Trans. Embedded Comput. Syst., 3(1):218–232, 2004.

12. Kuon, I. and Rose, J.: Measuring the gap between fpgas and asics. In Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays, FPGA '06, pages 21–30, New York, NY, USA, 2006. ACM.

13. Field-programmable gate array. In http://en.wikipedia.org/wiki/Field-programmable_gate_array.

14. Erjavec, T.: Introducing the Xilinx Targeted Design Platform: Fulfilling the Programmable Imperative, 2009.

15. Hauser, J.: Augmenting a Microprocessor with Reconfigurable Hardware. University of California, Berkeley, 2000.

16. Hadley, , Hadley, J. D., and Hutchings, B. L.: Design methodologies for partially reconfigured systems. In in Proc. FCCM95, P. Athanas and K.L. Pocek (eds.), IEEE Computer, pages 78–84. Society Press, 1995.

17. Trimberger, S., Duong, K., and Conn, B.: Architecture Issues and Solutions for a High-Capacity FPGA. In FPGA, pages 3–9, 1997.

18. Inc., X.: Virtex-II Pro X FPGA User Guide, 2007.

19. About fpgas. In http://home.mit.bme.hu/∼szedo/FPGA/fpgahw.htm.

20. QuisquateR, J. J. et al.: Design strategies and modified descriptions to optimize cipher fpga implementations: Fast and compact results for des and triple des.

21. Altera, I.: Stratix-II Device Handbook, 2005.

22. Inc., X.: Virtex-II Platform FPGAs: Complete Data Sheet, 2004.

23. Resano, J., Mozos, D., and Catthoor, F.: A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware. In Proceedings of the conference on Design, Automation and Test in Europe - Volume 1, DATE '05, pages 106–111, Washington, DC, USA, 2005. IEEE Computer Society.

CITED LITERATURE (continued)

24. Wirthlin, M. J.: A dynamic instruction set computer. In Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines, FCCM '95, pages 99–107, Washington, DC, USA, 1995. IEEE Computer Society.

25. Wirthlin, M. J. and Hutchings, B. L.: Sequencing run-time reconfigured hardware with software. In FPGA, pages 122–128, 1996.

26. Mishra, V., Raju, K. S., and Tanwar, P. K.: Article: Implementation of dynamically reconfigurable systems on chip with os support. International Journal of Computer Applications, 49(6):33–35, July 2012. Published by Foundation of Computer Science, New York, USA.

27. FPGA technology and dynamic reconfiguration. In Reconfigurable System Design and Verification, pages 15–44. CRC Press, February 2009.

28. Vince Ech, P.: In circuit partial reconfiguration of rocket io attributes. Technical report, Xilinx Inc., 2003.

29. Tapp., S.: Configuration quick start guidelines, 2003.

30. J. Noguera, R. M. B.: Multitasking on reconfigurable architectures: Microarchitecture support and dynamic scheduling. ACM Transactions on Embedded Computing Systems, 2004.

31. Caspi, E., DeHon, A., and Wawrzynek, J.: A streaming multi-threaded model. In In Proceedings of the Third Workshop on Media and Stream Processors, pages 21–28, 2001.

32. Inc., X.: Virtex-5 fpga configuration user guide, 2005.

33. A. Abnous, H. Z. M. W. G. V. V. P. J. R.: The pleiades architecture. Application of Programmable DSPs in Mobile Communications, 2002.

34. Goldstein, S. C., Schmit, H., Budiu, M., Cadambi, S., Moe, M., and Taylor, R. R.: Piperench: A reconfigurable architecture and compiler. IEEE Computer, 33(4):70–77, 2000.

35. Soft fpga cores attract embedded developers. In http://www.embedded.com/electronics-blogs/-include/4024958/Soft-FPGA-cores-attract-embedded-developers.

CITED LITERATURE (continued)

36. Vereen, L.: Soft fpga cores attract embedded developers, 2004.

37. Corp., A.: Nios II Processor Reference Handbook, 2004.

38. Inc., X.: Microblaze Processor Reference Guide, 2004.

39. Inc., X.: Zynq-7000 All Programmable SoC Overview, 2013.

40. Compton, K. and Hauck, S.: Reconfigurable computing: a survey of systems and software. ACM Comput. Surv., 34(2):171–210, 2002.

41. Inc, C. D. S.: Palladium Datasheet, 2004.

42. Graphics, M.: Vstation pro: High performance system verification. 2004.

43. Hauser, J. R. and Wawrzynek, J.: Garp: a mips processor with a reconfigurable coprocessor. In Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines, FCCM '97, pages 12–, Washington, DC, USA, 1997. IEEE Computer Society.

44. Rupp, C. R., Landguth, M., Garverick, T., Gomersall, E., Holt, H., Arnold, J. M., and Gokhale, M.: The napa adaptive processing architecture. In Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, FCCM '98, pages 28–, Washington, DC, USA, 1998. IEEE Computer Society.

45. Singh, H., Lee, M.-H., Lu, G., Kurdahi, F. J., Bagherzadeh, N., and Filho, E. M. C.: Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. IEEE Trans. Computers, 49(5):465–481, 2000.

46. Wittig, R. and Chow, P.: Onechip: an fpga processor with reconfigurable logic. In FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on, pages 126–135, 1996.

47. Marshall, A., Stansfield, T., Kostarnov, I., Vuillemin, J., and Hutchings, B. L.: A reconfigurable arithmetic array for multimedia application. In FPGA, pages 135–143, 1999.

48. Mirsky, E. and DeHon, A.: MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In IEEE Symposium on

CITED LITERATURE (continued)

FPGAs for Custom Computing Machines, eds. K. L. Pocek and J. Arnold, pages 157–166, Los Alamitos, CA, 1996. IEEE Computer Society Press.

49. Taylor, M. B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffman, H., Johnson, P., Lee, J.-W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., and Agarwal, A.: The raw microprocessor: A computational fabric for software circuits and general-purpose programs. IEEE Micro, 22(2):25–35, 2002.

50. Lbbers, E. and Platzner, M.: Reconos: An operating system for dynamically reconfigurable hardware. In Dynamically Reconfigurable Systems, eds. M. Platzner, J. Teich, and N. Wehn, pages 269–290. Springer, 2010.

51. Santambrogio, M.: Hardware-Software codesign methodologies for dynamically reconfigurable systems. Doctoral dissertation, Ph. D. thesis, Politecnico Di Milano, Italy, 2008.

52. Santambrogio, M. D., Rana, V., and Sciuto, D.: Operating system support for online partial dynamic reconfiguration management. In Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on, pages 455–458. IEEE, 2008.

53. Wigley, G. and Kearney, D.: The first real operating system for reconfigurable computers. In Proceedings of the 6th Australasian conference on Computer systems architecture, ACSAC '01, pages 130–137, Washington, DC, USA, 2001. IEEE Computer Society.

54. Gr-cpci-xc4v product sheet. In http://aeroflex.bentech-taiwan.com/aeroflex/PDF/GR-CPCI-XC4V_product_sheet.pdf.

55. Leon3 processor. In http://www.gaisler.com/index.php/products/processors/leon3.

56. Di Carlo, S., Gambardella, G., Prinetto, P., Rolfo, D., Trotta, P., and Lanza, P.: Femip: A high performance fpga-based features extractor amp; matcher for space applications. In Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on, pages 1–4, 2013.

57. Di Carlo, S., Gambardella, G., Lanza, P., Prinetto, P. E., Rolfo, D., and Trotta, P.: Safe: a self adaptive frame enhancer fpga-based ip-core for real-time space applications.

CITED LITERATURE (continued)

In IEEE 7th International Design and Test Symposium (IDT), Los Alamitos (CA), 2012. IEEE Computer Society.

58. Lakshmanan, R., Nair, M., Wilscy, M., and Tatavarti, R.: Automatic contrast enhancement for low contrast images: A comparison of recent histogram based techniques. In Computer Science and Information Technology, 2008. ICCSIT '08. International Conference on, pages 269–276, 2008.

59. Di Carlo, S., Prinetto, P., Rolfo, D., and Trotta, P.: Aidi: An adaptive image denoising fpga-based ip-core for real-time applications. In Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on, pages 99–106, 2013.

VITA

Alessandro Vallero has received:

- Bachelor of Science (BS) degree in Electrical and Computer Engineering, Politecnico di Torino, Torino (Italy), in 2011.
- Master of Science (MS) degree in Electrical and Computer Engineering, Politecnico di Torino, Torino (Italy), in 2013.

Alessandro Vallero at the moment is a PhD Student in Computer Engineering at Politecnico di Torino since 2014. His research efforts are addressed to reconfigurable computing and digital system reliability.