Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής

Πρόγραμμα Μεταπτυχιακών Σπουδών

«Προηγμένα Συστήματα Πληροφορικής»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**Αξιολόγηση της απόδοσης ετερογενών επαναδιαμορφώσιμων πολυπύρηνων επεξεργαστών** **Performance evaluation of heterogeneous reconfigurable multicore processors**
Ονοματεπώνυμο Φοιτητή	**Παπανικολάου Ιωάννης**
Όνομα Πατρός	**Κοσμάς**
Αριθμός Μητρώου	**ΜΠΣΠ 15065**
Κατεύθυνση	**Προηγμένες τεχνολογίες ανάπτυξης λογισμικού**
Επιβλέπων	**Ψαράκης Μιχαήλ, Επίκουρος Καθηγητής**

Πανεπιστήμιο Πειραιώς-Τμήμα Πληροφορικής
Πρόγραμμα Μεταπτυχιακών Σπουδών στα
Προηγμένα Συστήματα Πληροφορικής

Ημερομηνία Παράδοσης 25-10-2018

Περίληψη

Τα τελευταία χρόνια τα FPGA έχουν διεισδύσει στις αγορές και επιλέγονται όλο και περισσότερο από σχεδιαστές για την υλοποίηση πολυάριθμων εφαρμογών. Τα FPGA εξελίχθηκαν από εξειδικευμένα λογικά κυκλώματα που χρησιμοποιούνταν για την διασύνδεση πολλαπλών εμπορικών κυκλωμάτων, σε συσκευές με υψηλή πυκνότητα επαναπρογραμματιζόμενων λογικών πόρων πού είναι σε θέση να υλοποιήσουν σύνθετα συστήματα σε ένα ολοκληρωμένο κύκλωμα. Ένα από τα μεγαλύτερα πλεονεκτήματα των FPGA είναι η ευελιξία που προσφέρουν καθώς η λειτουργικότητα τους μπορεί να αλλάξει φορτώνοντας ένα νέο configuration file στην μνήμη τους. Επεκτείνοντας την ευελιξία των FPGA με την χρήση δυναμικής μερικής επαναδιαμόρφωσης (Dynamic Partial Reconfiguration) ο εκάστοτε σχεδιαστής έχει την δυνατότητα να επαναδιαμορφώσει περιοχές του FPGA χωρίς να επηρεάζει ολόκληρο το σύστημα. Η δυνατότητα αυτή επιτρέπει στα FPGA να προσαρμόζονται δυναμικά σε συγκεκριμένες προδιαγραφές.

Η παρούσα μεταπτυχιακή διατριβή παρουσιάζει και αξιολογεί μια αποτελεσματική αρχιτεκτονική ετερογενών επαναδιαμορφώσιμων πολυπύρηνων επεξεργαστών. Η αρχιτεκτονική χρησιμοποιεί δυναμική μερική επαναδιαμόρφωση για να προσαρμόσει τους διαθέσιμους πόρους σύμφωνα με τον φόρτο εργασίας. Η αποδοτικότητα επιτυγχάνεται βελτιστοποιώντας την χρήση των πόρων καθώς και την χρήση της ενέργειας παρέχοντας αρκετή υπολογιστική ισχύ έτσι ώστε το σύστημα να τηρεί τις προδιαγραφές. Η αρχιτεκτονική αυτή υλοποιήθηκε και αξιολογήθηκε στην αναπτυξιακή πλατφόρμα Zybo που διαθέτει το Xilinx Zynq-7000 FPGA. Τα αποτελέσματα αυτού του πειράματος καταδεικνύουν ότι με την χρήση της προτεινόμενης αρχιτεκτονικής είναι εφικτό να διατηρηθεί η επιθυμητή απόδοση μειώνοντας την χρήση της ενέργειας.

Abstract

Over the last few years, Field Programmable Gate Arrays have penetrated the markets and are increasingly embraced by designers for numerous applications. FPGAs evolved from small glue logic circuits to devices with high-density of reconfigurable logic resources that are capable of implementing large systems in a single chip. One great asset that FPGAs offer is high flexibility since their functionality can be altered by simply loading a new binary file in their configuration memory. Extending the flexibility, a unique capability of FPGAs called Dynamic Partial Reconfiguration (DPR) allows regions to be programmed with new functionality while applications are still running in the remainder of the device. Hence, making FPGAs able to adapt to specific constraints by modifying its hardware in real time.

This thesis demonstrates and evaluates an efficient architecture of a heterogeneous reconfigurable multicore system on a Xilinx MPSoC. The architecture utilizes DPR to adjust hardware resources according to the workload while the device is operating. Efficiency is achieved by optimizing both area and power usage while delivering high performance in order to meet strict time deadlines. The architecture is being implemented, evaluated and tested on a Zybo development board featuring a Xilinx Zynq-7000 FPGA. This system can load custom reconfigurable modules containing multiple instances of Xilinx's Intellectual Property softcore Microblaze to supplement the preexisting hardwired processor. The results of this experiment demonstrate that it is possible to maintain a desirable performance while decreasing power usage.

Contents

Περίληψη **3**

Abstract **3**

1 Introduction **7**

1.1 Motivations and Objectives **7**

1.2 Thesis Outline **7**

2 FPGA Overview **8**

2.1 Introduction **8**

2.1.1 FPGA Evolution through the Years 8

2.1.2 Basic Components 10

2.1.3 Uses and Application Fields 14

2.2 FPGA Development Flow. **15**

2.3 Partial Reconfiguration. **19**

2.3.1 Partial Reconfiguration Fundamentals 21

2.3.2 Configuration Mechanisms 22

2.3.3 Floorplaning for Reconfigurable Designs 22

2.3.4 Reconfigurable Families 22

2.3.5 Benefits of Partial Reconfiguration 23

2.4 Processors on FPGAs **26**

2.4.1 Soft Processors 26

2.5 Zynq-7000 Architecture **28**

2.5.1 Processing System 28

2.5.2 Programmable Logic 30

2.5.3 Processing System — Programmable Logic Interfaces.. 32

3 Heterogeneous Multicore Systems.. 33

3.1 Multi-Processors System-on-Chip 33

3.1.1 Heterogeneous MPSoC Architecture 34

3.1.2 Homogeneous MPSoC Architecture 35

3.1.3 Hardware Accelerators 36

3.2 Hardware Reconfiguration in MPSoCs 37

3.2.1 Runtime Adaptive MPSoC 37

3.2.2 Benefits of the RAMPSoC Approach............................ 38

3.3 MPSoCs Design Challenges... 39

3.3.1 Cache Coherence.. 39

3.3.2 Power and Temperature 39

3.3.3 Reliability Issues 40

3.4 MPSoC Applications 40

3.4.1 Applications Mapping.. 40

4 Implementation of a Heterogeneous Reconfigurable Multicore System 42

4.1 General System Architecture 43

4.1.1 Building Blocks 43

4.1.2 Interfacing the Softcores to the Processing System 43

4.2 Hardware Development.. 44

4.2.1 Design Entry 45

4.2.2 Hardware Synthesis and Implementation...................... 46

4.3 Software Development... 48

4.3.1 Hardware Handling and Memory Management. 48

5 Experimental results.. 50

5.1 Testing 50

5.1.1 Benchmarks.. 50

5.2 Results 50

6 Conclusion and future work 54

Bibliography and references 55

Appendix A. Template applications source code 57

A1. ARM Processor 57

A2. Microblaze0 softcore 64

A3. Microblaze1 softcore 65

1 Introduction

1.1 Motivations and Objectives

The main motivation behind this thesis is to study the advanced FPGA design technique called Partial Reconfiguration and to highlight some of the strengths and the weaknesses of this technique. Starting from a concept we progress through all the design stages required reaching an intricate system which we test extensively. The objective is to utilize a low budget FPGA and with the help of Partial Reconfiguration to present a novel architecture that improves the performance of the system.

1.2 Thesis Outline

This thesis is composed of six chapters. The remainder of this thesis is summarized as follows:

Chapter 2 provides the needed background theory of the presented thesis work. The Xilinx FPGA architecture is introduced, by providing terminology and details of the different components. Partial Reconfiguration is also introduced in this chapter with an overview of its development flow, key concepts, and benefits. Finally, a description of the processors used in FPGAs and the Zynq-7000 architecture are presented, since both of them are concepts strongly used in many aspects of this thesis.

Chapter 3 is a literature review of the main branches of multicore systems. Starting with an introduction on MPSoCs, the heterogeneous and homogeneous architectures are presented. This chapter also presents the concept of hardware reconfiguration in MPSoCs along with a novel architecture proposed in a white paper. The chapter concludes with a discussion on MPSoC design challenges and a brief reference on MPSoC applications.

Chapter 4 describes the implementation of a heterogeneous reconfigurable multicore system, with a focus on its architecture as well as on its hardware and software development methodology.

Chapter 5 presents the experimental results from the deployment of the heterogeneous reconfigurable system on a Zybo Zynq MPSoC. The system was tested with multiple applications coming from various commercial backgrounds. Furthermore, the power consumption of the system is presented and compared against a static standalone system.

Chapter 6 evaluates the results and justifies the cases that the proposed architecture fulfills its purpose. Drawbacks and potential improvements are also discussed in this chapter as well as potential future work.

2 FPGA Overview

Field programmable gate arrays or FPGAs are a special type of integrated circuits that can be programmed at the field after manufacture. This technology is used by system developers to design, debug, and implement unique hardware solutions without having to develop custom silicon devices. Vendors such as Xilinx and Altera are selling FPGA chips blank or unprogrammed to customers with no intended function. The customers then configure these devices to implement their unique system design and evaluate the results. If a feature changes or a bug is discovered, the user can simply load a new configuration to the FPGA to create a new product. This process can be repeated until an ideal system has been achieved, giving the designer the flexibility to make changes very late in the design cycle and even after product shipment in the form of firmware upgrades.

2.1 Introduction

FPGA configuration creates digital circuits described in Hardware Description Languages (HDL) and should be distinguished from loading any associated software programs. These digital circuits consist of basic components such as logic gates, flip-flops and memories, and wires. There are three types of configuration technologies with each having unique features. Most commonly used the static random access memory (SRAM) is fast and small, and it offers unlimited reprogrammability. One of the drawbacks though is that the FPGA needs time to reload the entire design into SRAM every time the FPGA powers up. This approach also takes more power. Flash and antifuse technologies are non-volatile (meaning that they can retain data even if power is turned off) so that they provide the benefit of “instant on” without needing to reload the FPGA bit file every time we power up the FPGA or system. They also draw less power than the SRAM approach. Antifuse technology can only be programmed once and can’t match the performance of SRAM technology. The only reasons to use an antifuse technology FPGA today is due to its super high reliability and security.

FPGA architectures have been intensely investigated over the past two decades. A significant aspect of FPGA architecture research is the development of Computer Aided Design (CAD) tools for mapping applications to FPGAs. It is well established that the quality of an FPGA-based implementation is largely determined by the effectiveness of accompanying suite of CAD tools. Benefits of an otherwise well design, feature rich FPGA architecture might be impaired if the CAD tools cannot take advantage of the feature the FPGA provides. Thus, CAD algorithm research is essential to the necessary architectural advancement to narrow the performance gaps between FPGAs and other computational devices.

The primary alternative to an FPGA is an ASIC (Application Specific Integrated Circuit) that has speed, power and area advantages over an FPGA. Compared to FPGAs, ASICs have certain disadvantages in the form of higher non-recurring engineering (NRE) cost, longer manufacturing time and increasingly complicated design process. In high volumes, ASIC implementations have resulted in the most cost-effective and lower energy solutions. However, increasing costs and the fact that there is no room for error or change once an ASIC is released have made the FPGA a much more attractive solution in specific applications.

2.1.1 FPGA Evolution through the Years

In the 1970s, logic systems were created by building PCB boards consisting of transistor-transistor logic (TTL) logic chips. However, as functions got larger, the logic size and levels increased and thus compromised the speed of the design. Typically, designers used logic minimization techniques, such as those based on Karnaugh maps to create a sum of products expression by generating the product terms using AND gates and summing them using an OR gate.

Eight years later a company named Monolithic Memories introduced the concept of creating a structure to achieve implementation of this functionality with a new programmable array logic (PAL). The PAL comprised a sum of products structures to be implemented directly from minimized expression. In Figure 1 the programmable elements (shown as a fuse) connect both the true and complemented inputs to the AND gates. These AND gates, also known as product terms, are ORed together to form a sum-of-products logic array. A matrix with fixed AND and OR gates became the key feature of a class of devices known as programmable logic devices (PLDs).

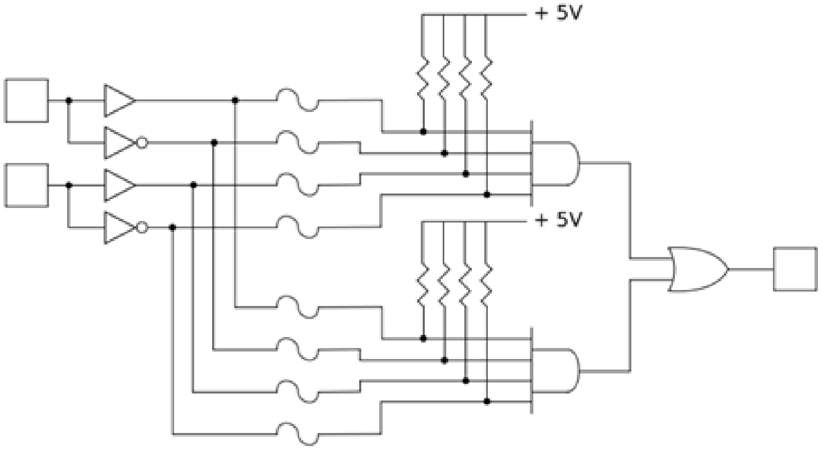


Figure 1 A simplified programmable logic device. [1]

The initial PLDs could replace ten or so TTL gates and were one time programmable. This led to the reprogrammable PLDs based on EEPROM or EPROM technologies. The PLD structure had a number of advantages. It clearly matched the process of how the sum of products was created by the logic minimization techniques. The function could then be fitted into one PLD device, or, if not enough product terms were available, then it could be fed back into a second PLD stage. Another significant advantage was that the circuit delay is deterministic, either comprising one logic level or two.

As lithography advanced through the years, it enabled a new class of device, the FPGA. FPGAs introduced two important new architecture features: programmable routing to interconnect the increasing number of gates on a device and a programmable gate called a LUT or lookup table with an associated register. A LUT is a small amount of read-only memory. A four-input, one output LUT can generate any four input Boolean function. It is also small enough to achieve efficient utilization of the chip area, but large enough to implement a reasonable range of functions. If a higher number of inputs is required, then LUTs are cascaded together to provide implementation, but at a lower speed. This was judged to provide an acceptable trade-off. The initial devices from Xilinx contained up to a hundred LUT and flip-flop pairs in a basic logic element called CLB or configurable logic block. Rather than using a permanently programmed EPROM or EEPROM memory, Xilinx relied on CMOS memories to hold programming information. The first commercial concept emerged in 1985 with the XC2064 FPGA family from Xilinx which had 64 programmable logic cells. At the same time, Altera was also developing a programmable device later to become the EP1200, which was the first high-density programmable logic device (PLD).

The early FPGA offerings were based on the Manhattan-style architecture (Figure 2) where each individual cell comprised simple logic structures and cells were linked by programmable connections. Thus FPGA could be viewed as comprising the following:

- programmable logic units that can be programmed to realize different digital functions
- programmable interconnect to allow different blocks to be connected together
- programmable I/O pins.

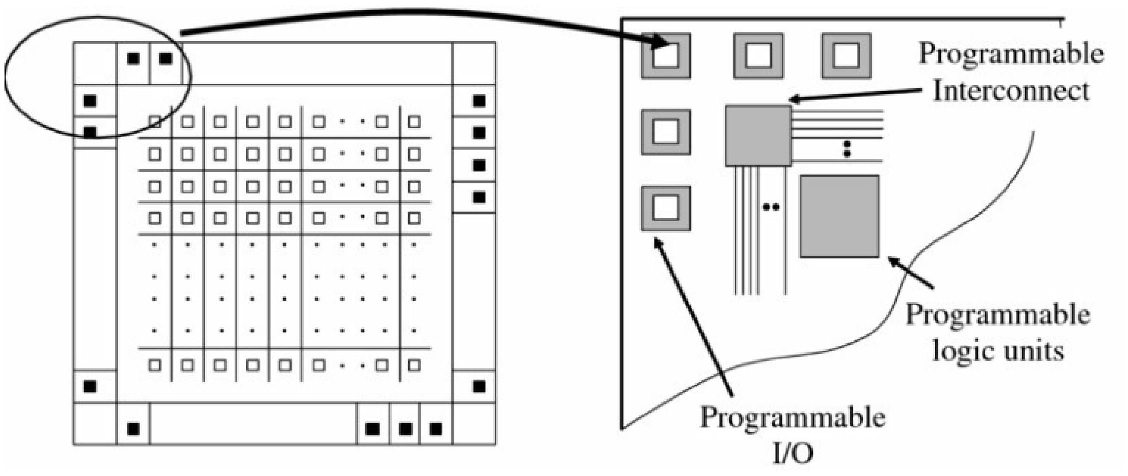


Figure 2 Early Manhattan architecture. [2]

This was ideal for situations where FPGAs were viewed as glue logic as programmability was then the key to provide redundancy and protection against PCB board manufacture errors, it might even provide a mechanism to correct design faults. However, technology evolution, outlined by Moore’s law, now provided scalability for FPGA vendors. During the 1980s, this was exploited by FPGA vendors in scaling their technology regarding numbers of interconnectivity and number of I/Os. Nonetheless, it was recognized that this approach had limited scope, as scaling meant that interconnect was becoming a major issue and technology evolution now raised the interesting possibility that dedicated hardware could be included, such as dedicated multipliers and, more recently ARM processors. In addition, the system interconnectivity issue would be alleviated by including dedicated interconnectivity in the form of Serializer/Deserializer and RapidIO.

At an early stage, the FPGA market was populated by a number of vendors, including Xilinx, Altera, Actel, Lattice, Crosspoint, Prizm, Plessey, Toshiba, Motorola, Algotronix and IBM. However, the costs of developing technologies not based on conventional integrated circuit design processes and the need for programming tools saw the demise of many of these vendors and a reduction in the number of FPGA families.

The FPGA took its place as a central component in digital systems, replacing PLDs and TTL for implementing glue logic. In the 1990s new uses began to emerge for FPGAs, which were becoming more capable than just gluing I/O to processors. The emerging Internet became a growth driver for FPGAs with FPGAs being used for prototyping, initial deployment, and full-scale production of Internet switches and routers. By 2000 communications systems were the primary market for FPGAs. Other new markets for FPGAs also emerged for ASIC prototyping and high-performance DSP (digital signal processing) systems. FPGAs also began to be used for implementing soft control processors such as the Xilinx Microblaze and PicoBlaze architectures. As of today, the market is dominated by Xilinx and Altera and more importantly, the FPGA has grown from a simple glue logic component to a complete system on chip (SoC) comprising onboard physical processors, soft processors, dedicated DSP hardware, memory and high speed I/O.

2.1.2 Basic Components

The computing functionality in an FPGA is provided by its programmable logic blocks and these blocks connect to each other through programmable routing network. Modern FPGAs include unique components that cater specific needs to cover a broad spectrum of applications.

Programmable Interconnect:

FPGAs are built around an array of programmable logic blocks embedded in a sea of programmable interconnect. This array is often referred to as the programmable logic fabric (PL) or just the fabric. At the edges are programmable I/O blocks design to interface the fabric signals to the external world. Interestingly, nearly all the other special FPGA features such as carry chains, block RAM, or DSP blocks can also be implemented in programmable logic. This is in fact the approach the initial FPGAs took and users did implement these functions in LUTs. However, as the FPGA markets developed, it became clear that these special functions would be more cost effective as dedicated functions build from hard gates and later FPGA families such as the Xilinx 4K series and Virtex began to harden these special functions. This hardening improved not only the cost but also improved frequency substantially.

Within any one FPGA family, all devices will share a typical fabric architecture, but each device will contain a different amount of programmable logic. This enables the user to match their logic requirements to the right-sized FPGA device. FPGAs are also available in two or more package sizes which allow the user to match the application – I/O requirements to the device package. FPGA devices are also available in multiple speed grades and multiple temperature grades as well as multiple voltage levels. The highest speed devices are typically 25% faster than the lower speed devices. By designing to the lowest speed, users can save on cost, but the higher performance of faster devices may minimize the system level cost. Modern FPGAs commonly operate at 100-500 MHz. In general, most logic designs which are not targeted at FPGA architectures will run at lower frequency range, and designs targeted at FPGAs will run in the mid-frequency range. The highest frequency designs are typically DSP designs explicitly constructed to take advantage of FPGA DSP and BRAM blocks.

A high-level overview of FPGA architectures can be split into different pieces. Starting at the foundation, the Programmable Interconnect is a set of wires woven through the FPGA logic fabric which can be wired together to connect any two blocks in an FPGA. This enables arbitrary logic networks to be constructed by the user. The architecture of the interconnect wires varies from generation to generation and is hidden from the user tools.

Programmable Logic Blocks:

An array of programmable logic blocks are embedded into the Programmable Interconnect and form the main logic resources for implementing sequential as well as combinatorial circuits. These are called CLBs (configurable logic blocks) in Xilinx devices. Today, each logic block consists of one or more programmable logic functions implemented as a 4-6 bit configurable lookup table (LUT), a configurable carry chain, and configurable registers. We use the word configurable to indicate a hard block which can be configured through the FPGA's configuration memory to be used as part of the user's logic. The combination of a LUT, carry chain, and register is called a logic cell or LC. The capacity of FPGAs is commonly measured in logic cells. For instance, Xilinx Virtex UltraScale FPGA supports up to 4 million LCs, while the smallest Spartan device contains as few as 2000 logic cells. Depending on usage, each logic cell can map between 5 and 25 ASIC gates. The lower number is commonly used for ASIC netlist emulation, while the higher number is achievable under expert mapping. For Xilinx UltraScale devices, the CLB supports up to 8 x 6-input LUTs, 16 registers, and 8 carry chain blocks. Each 8-LUT can be configured as 2 x 5-LUTs if the 5-LUTs share common signals. Figure 3 illustrates the UltraScale CLB architecture.

Embedded in the CLB is a high-performance look-ahead carry chain which enables the FPGA to implement very high-performance adders. Current FPGAs have carry chains which can implement a 64-bit adder at 500MHz. Associated with each LUT is an embedded register. The rich register resources of the FPGA programmable logic enable highly pipelined designs, which are a key to maintaining higher speeds. Each register can be configured to support a clock enable and reset with configurable polarity.

An important additional feature of the Xilinx CLB's 6-LUT is that it can configure to implement a small 64-bit deep by 1-bit wide memory called a distributed RAM. An alternate configuration allows the 6-LUT to implement a configurable depth shift register with a delay of 1-32 clocks.

Access to memory is critical in modern logic designs. Programmable logic designs commonly use a combination of memories embedded in the FPGA logic fabric and external DDR memories. Within the logic fabric, memory can be implemented as discrete registers, shift registers, distributed RAM, or block Ram. Xilinx UltraScale devices support two sizes of block RAM, 36-kbit RAMs, and 288-kbit RAMs. In most cases, the Xilinx tools will select the best memory type to map each memory in the user design. In some cases, netlists optimized for FPGAs will hand instantiate memory types to achieve higher density and performance.

Special forms of memory called dual-port memories and FIFOs are supported as special modes of the block RAMs or can be implemented using distributed RAM. System memory access to external DDR memory is via a bus interface which is commonly an AXI (advanced Xilinx interface) protocol to the FPGA. UltraScale FPGAs support 72-bit wide DDR4 at up to 3200 MB/s.

In general, registers or flip-flops are used for status and control registers, pipelining, and shallow (1-2 deep) FIFOs. Shift registers are commonly used for signal delay elements and pipeline balancing in DSP designs. Distributed RAMs are used for shallow memories up to 64 bits wide and can be as wide as necessary. Block RAMs are used for buffers and deeper memories. They can also be aggregated together to support arbitrary widths and depths. For instance, a 64-bit wide by 32 K-bit deep memory would require 64 block RAMs. Generally, FPGAs contain around 1 36 K block RAMs for every 500-1000 logic cells.

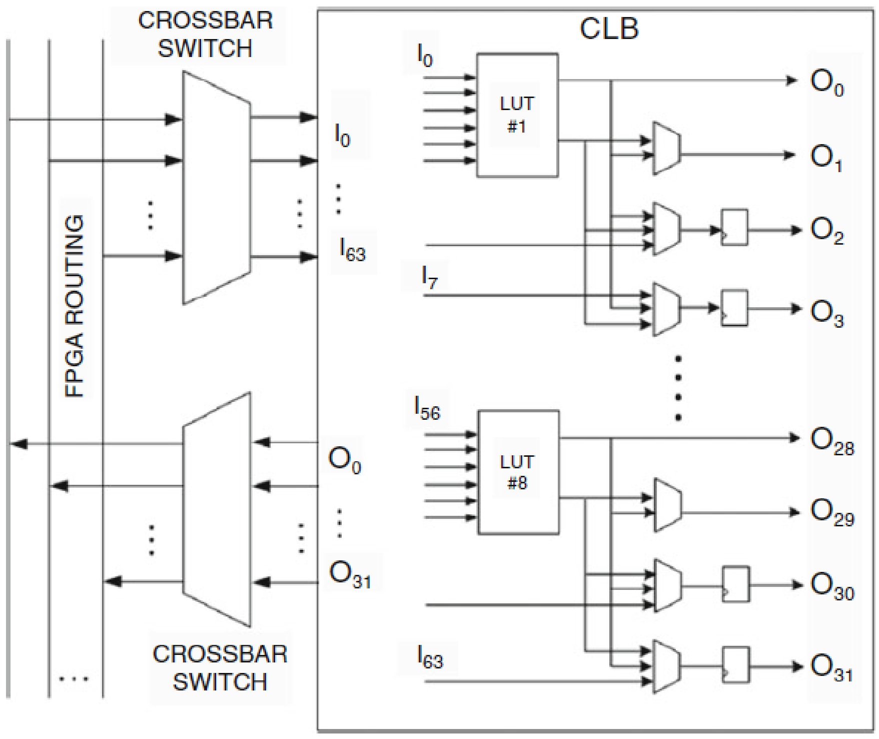


Figure 3 UltraScale CLB architecture. [3]

Digital Signal Processing:

Modern FPGAs contain discrete multipliers to enable efficient DSP processing. Commonly DSP applications build pipelines or flow graphs of DSP operations and data streams through this flow graph. A typical DSP filter called FIR (finite impulse response) filter, it consists of sample delay blocks, multipliers, adders, and memories for coefficients. Interestingly it can be implemented almost directly as an FPGA circuit. For filtering and many other DSP applications, multiplies and adders are used to implement the flow graph. Xilinx FPGAs contain a DSP block known as a DSP48 which supports an 18-bit× 25-bit multiplier, a 48-bit accumulator, and a 25-bit pre-adder. In addition up to four levels of pipelining can be supported for operation up to 500 MHz. The DSP48 supports integer math directly; however, 32-bit and 64-bit floating point operations are supported as library elements. A 32-bit floating point multiplier will require two DSP48s and several hundred LCs.

Xilinx tools will generally map multipliers and associated adders in RTL or HDL languages to DSP48 blocks. For highest performance however, designs optimized for DSP in FPGAs may use DSP48 aware libraries for optimal performance, power, and density.

Clock Management:

Logic netlists almost universally require one or more system clocks to implement synchronous netlists for I/O and for internal operation. The synchronous operation uses a clock edge to register the results of upstream logic and hold it steady for use by downstream logic until the next clock edge. The use of synchronous operation allows for pipelined flow graphs which process multiple samples in parallel. External digital communications interfaces use I/O clocks to transfer data to and from the FPGA. Commonly, interface logic will run the I/O clock rate (or a multiple of the I/O clock rate).

I/O blocks:

One of the critical capabilities of FPGA is that they interface directly to external input and output (I/O) signals of all types and formats. To support these diverse requirements, modern FPGAs contain a special block called the I/O block or IOB. This block contains powerful buffers to drive external signals out of the FPGA and input receivers, along with registers for I/O signals and output enables (OE). IOBs typically support 1.2- 3.3 V CMOS as well as LVDS and multiple industry I/O memory standards such as SSTL3. I/Os are abstracted from the user RTL and HDL design and are typically configured using a text file to specify each I/O’s signaling standard.

UltraScale devices also include multiplexing and demultiplexing features in the I/O block. This feature supports dual data rate (DDR) operation and operation for 4:1 or 8:1 multiplexing and demultiplexing. This allows the device to operate at a lower clock rate than the I/O clock. For example, Gigabit Ethernet (SGMII) operates at 1.25 GHz over a single LVDS link, which is too fast for the FPGA fabric to support directly. The serial signal is expanded to 8/10 bits in the IOB interface to the fabric allowing the fabric to operate at 125 MHz.

I/Os are commonly a limited resource and FPGAs are available in multiple package sizes to allow the user to use smaller lower-cost FPGAs with lower signal count applications and larger package sizes for higher signal count applications. This helps to minimize system cost and board space. A primary application of FPGA I/Os is for interfacing to memory systems. UltraScale devices support high-bandwidth memory systems such as DDR4.

High-Speed Serial I/Os (HSSIO):

CMOS and LVDS signaling are limited in performance and can be costly in terms of power and signal count. For this reason, high-speed serial I/Os have been developed to enable low-cost, high-bandwidth interfaces. This evolution can be seen in the evolving PCI standard which has moved from low-speed 32-bit CMOS interfaces at 33 MHz to PCIe Gen3 with 1-8 lanes at Gb/s lane. An eight-lane PCIe Gen3 interface can transfer 64 Gb/s of data in each direction. Xilinx UltraScale devices support up to 128 MGT (Multi-Gigabit Transceivers) at up to 32.75 Gb/s.

Within the FPGA, the HSSIO are interfaced directly to a custom logic block which multiplexes and demultiplexes the signals to wide interfaces at lower clock rates.

2.1.3 Uses and Application Fields

FPGAs are used extensively for computing problems that benefit from parallel computer architectures due to their parallel nature and optimality in terms of the number of gates used for a certain process. An FPGA can even be proven faster than a computer if the tasks are separated into discrete pieces. Another aspect of FPGAs that is now on the rise is hardware acceleration, where one can use the FPGA to accelerate certain parts of an algorithm and share part of the computation between the FPGA and a generic processor. Furthermore, reusability of the FPGA is a crucial advantage. Prototypes of a design can be implemented on FPGAs, which then can be verified to closely replicate how an ASIC would behave. In case of design faults, new versions can be created by changing the HDL code, generating a new bitstream, and configuring the FPGA in order to test it again since modern FPGAs are reconfigurable both partially and on the fly. A new design can be uploaded even remotely, instantly due to FPGAs field reprogram ability. This is very useful in cases that the designer cannot physically access the FPGA for example in a satellite that orbits earth or if a product has been released and needs an update.

FPGAs have justly gained their own position as suitable platforms to deal with increasingly complex control tasks and are also getting at very fast pace, into the world of High-Performance Computing (HPC). This technological trend has also extended the applicability of FPGAs in their original application domains. For instance, emulation techniques are evolving into mixed solutions, where the behavior or parts of a system can be evaluated by combining simulation models with hardware emulation, in what is nowadays referred to as hardware-in-the-loop (HIL). Tools exist, including some of general use in engineering, such as MATLAB, which allow this combined simulation/emulation approach to be used to accelerate system validation.

FPGAs are also increasingly penetrating the area of embedded control systems because in many cases, they are the most suitable solution to deal with the growing complexity problems to be addressed in that area. Some important fields of application (not only regarding technical challenges but also in terms of digital systems market share) are in automated manufacturing, robotics, control of power converters, motion and machinery control, and embedded units in automotive. It is worth noting that a modern car has some 70-100 embedded control units onboard. As the complexity of the systems to be controlled grows, microcontroller and DSPs are becoming less and less suitable, and FPGAs are taking the floor.

A clear proof of the excellent capabilities of current FPGAs is their recent penetration in the area of HPC, where a few years ago, no one would have thought they could compete with software approaches implemented in large processor clusters. However, computing-intensive areas such as big data applications, astronomical computations, weather forecast, financial risk management, complex 3D imaging (e.g., in architectures, movies, virtual reality, or video games), traffic prediction, earthquake detection, and automated manufacturing may currently benefit from the acceleration and energy efficient characteristics of FPGAs.

Traditionally, FPGAs have been reserved for specific industries and markets where the volume of production is small. For these low-volume applications, the premium that companies pay in hardware costs per unit for a programmable chip is more affordable than the development resources spent on creating an ASIC for the low-volume application. Today, new cost and performance dynamics have broadened the range of viable applications such as:

- ASIC prototyping.
- Wired communications: for system development, while the standards themselves are evolving.
- Wireless communications: DSP in FPGAs is a major attraction for algorithmic computations.

- Video systems and machine vision: Implement software algorithms at higher speed and lower power.
- Industrial systems: Communication link between sensor nodes and robotic systems.
- Medical systems: I/O interfaces including A-to-D and D-to-A conversion.
- Automotive systems: video processing for driver assistance.
- Military and aerospace: Radio waveform processing and processing of huge amounts of sensor data.
- Data center: Interfaces to SSD, machine learning related algorithms.

2.2 FPGA Development Flow.

Tools and methodologies for FPGA-based design have been continuously improving over the years for them to accommodate the new and extended functionality requirements imposed by increasingly demanding applications. Today's designs would take unacceptable extremely long times to be completed if tools coming from more than 20 years ago were used. The first important incremental step in accelerating design processes was the replacement of schematic-based design specifications by HDL descriptions. On the one hand, this allows complex circuits described at different levels of abstraction to be more efficiently simulated, and on the other hand, designs to be quite efficiently translated by means of synthesis, mapping, placement, and routing tools from HDL into netlists, as a step previous to its translation into the bitstream with which the FPGA is configured.

Designers quite rapidly adopted conventional synthesis tools due to the productivity jump they enabled. At that point, it soon became apparent that FPGAs were very well suited to rapid prototyping and emulation flows because very little HDL code rework (or even none at all) was required in order to migrate designs initially implemented in FPGAs to other technologies. Either for prototyping or final deployment, FPGAs rapidly increased their manufacturing share. As a consequence, and thanks to the improvement of manufacturing technologies, their complexity was continuously increased to cope with the ever-growing demand for more and more complex integrated systems. This in turn, contributed to higher market penetration, which pushed for additional complexity and expanded functionality.

The fast adoption of conventional synthesis tools as part of the natural design process for all types of digital hardware devices was not as fast in the case of High-Level-Synthesis (HLS) tools. The difference between both types of tools resides in clock cycle explicitness. A conventional synthesizable HDL file mostly consists of descriptions where the transfers between memory elements can be directly and explicitly inferred from the code, clock cycle by clock cycle. In contrast, HLS tools start from descriptions that do not explicitly specify clock activity but work at algorithmic level instead. The contribution or refinement HLS tools provide their ability to allocate logic resources or operators and assign functions to such operators within the required time slots so that the algorithm may be mapped to a circuit with efficient resource sharing. Additionally, logic functions can be extended into optimized pipelined structures (so that the translation of such slots into clock cycles makes timing explicit), and clock speed can be optimized by adequately balancing critical paths within the pipelined structures. Regarding memories, different accessing schemes enable variable bandwidth adjustment so that it may fit appropriately to the functions being carried out by the logic reading/writing data from/to such memories. Finally, HLS tools also support two I/O types: memory mapped and stream based.

Traditional or HLS tools alone cannot support the design of many of today's complex FPGA embedded systems. They need to be combined with platform-based tools that, in essence, automate different processes within the FPGA design flow. These tools combine standard components from integrated IP libraries with custom-made block designed using either conventional or HLS flows. Most current embedded systems are not fully customized designs, but rely on the combination of some standardized functions and interfaces with custom-made IP blocks. Therefore, module reuse and automated tools are mandatory in order to speed up the design process. Complex systems may be built with relatively little designer intervention if the design is based on library modules connected with standardized on-chip interfaces. These tools provide, among many other features, module customization, automatic connection, automated memory map generation, as well as easy access to software code programmers using hardware abstraction layers for easy hardware/software interfacing. This design methodology allows highly complex designs to be readily obtained, for instance, a dual-core processor system with complex DMA schemes providing efficient access to a gigabit Ethernet media access control layer, plus some other I/O interfaces (such as SPI, I^2C, USARTS, or GPIO), can be built in only a few hours.

Other tools are currently available whose design languages allow explicit parallelism to be described, aimed at achieving the maximum algorithm acceleration in HPC applications. They are based on OpenCL, which allows multithread parallelism to be mapped to heterogeneous computing platforms, such as FPGAs. In the last years, the leading FPGA vendors are continuously releasing new specialized tools to ease the translation from OpenCL code into FPGA designs. These tools provide ways for designs running in a host usually a computer, to be accelerated by attaching one or more FPGA boards to it, often using PCIe connections.

Increasingly, complex tools and design flows must necessarily be complemented with proper validation and debugging methods. Verification can (and should) be done at early design stages, prior to circuit configuration, by means of simulation techniques. These techniques may be performed at functional level, to validate logic functionality, or after placement and routing, where accurate timing data are available to be annotated into the simulation. Very interestingly, it is also possible to use integrated logic analyzers (embedded into the FPGA) for debugging purposes. These elements allow for combined hardware/software evaluation, which is very useful, especially for field-programmable SoC designs.

One of the major research aspects of FPGAs is the development of software flow required to map hardware applications on an FPGA. The effectiveness and quality of an FPGA are mainly dependent on the software flow provided with an FPGA. The software flow takes an application design description in a Hardware Description Language (HDL) and converts it to a stream of bits that is eventually programmed on the FPGA. Figure 4 shows a complete software flow for programming an application circuit on an FPGA. A brief description of various modules of software flow is described below.

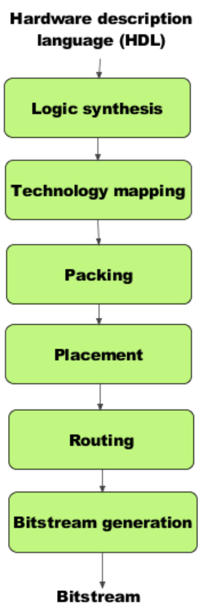


Figure 4 FPGA software configuration flow.

Logic synthesis:

Logic synthesis transforms an HDL description (VHDL or Verilog) into a set of Boolean gates and flip-flops. The synthesis tools transforms the register-transfer-level (RTL) description of a design into a hierarchical Boolean network. Various technology-independent techniques are applied to optimize the Boolean network. The typical cost function of technology-independent optimizations is the total literal count of the factored representation of the logic function. The literal count correlates very well with the circuit area.

Technology mapping:

After logic synthesis, technology-dependent optimizations are performed. These optimizations transform the technology-independent Boolean network into a network of gates in the given technology library. The technology mapping for FPGAs transforms the given Boolean network to the available set of blocks on an FPGA. For a traditional FPGA architecture, the Boolean network is transformed into Look-Up tables and flip-flops. Given a library of cells, the technology mapping problem can be expressed as finding a network of cells that implement the Boolean network. In the FPGA technology mapping problem, the library of cells is composed of k-input LUTs and flip-flops. Therefore, FPGA technology involves transforming the Boolean network into k-bounded cells. Each cell can then be implemented as an independent k-LUT. Technology mapping algorithms optimize a given Boolean network for a set of different objective functions including depth, area and power. The FlowMap algorithm is the most widely used academic tool for FPGA technology mapping. FlowMap is a breakthrough in FPGA technology mapping because it is able to find a depth-optional solution in polynomial time. The final output of FPGA technology mapping is a network of I/Os, LUTs, and flip-flops.

Clustering/Packing:

The logic elements in a mesh-based FPGA are typically arranged in two levels of hierarchy. The first level consists of logic cells which are k-input LUT and flip-flop pairs. The second level hierarchy groups *k* logic cells together to form logic cell clusters. These clusters can then be directly mapped on the CLBs of an FPGA. The main optimization goal is to cluster the LUTs, flip-flops and logic cells in such a way that inter-cluster communication is minimized. Less inter-cluster communication ensures less routing resource utilization in an FPGA. There are also different approaches to clustering algorithms that achieve specific features based on the needs of the application. These algorithms can be broadly categorized into three general approaches, namely top-down, depth-optimal and bottom-up. Top-down approaches partition the logic cells into clusters by successively subdividing the network or by iteratively moving logic cells between parts. Depth-optimal solutions attempt to minimize delay at the expense of logic duplication. Bottom-up approaches are generally preferred for FPGA CAD tools due to their fast run times and reasonable timing delays. They only consider local connectivity information and can easily satisfy clusters pin constraints. Top-down approaches offer the best solutions. However, their computational complexity can be prohibitive. The final output of packing is a network of I/Os and CLBs.

Placement:

The placement algorithm determines the position of CLB and I/O instances in a packed netlist on the respective CLB and I/O blocks on the FPGA architecture. The main goal of placement algorithm is to place connected blocks near each other so that minimum routing resources are required to route their connections. The placement algorithm can also serve to fulfill other architectural or optimization requirements, such as balancing the wire density across FPGA.

Three major types of commonly used placement algorithms are (I) min-cut (partitioning) based placement algorithm which is an approach suitable for hierarchical FPGA architectures. The partitioner is recursively applied to distribute netlist instances between clusters. The aim is to reduce external communication and merge highly connected instances in the same cluster. (II) Analytical placement algorithms commonly utilize a quadratic wire length objective function. Although, a quadratic objective is only an indirect measure of the wire length; its main advantage is that it can be minimized very efficiently and is this suitable for handling massive problems. A quadratic function does not give the best possible wire length, and it is often followed by some local iterative improvement techniques. (III) The simulated annealing placement algorithm uses the annealing concept for molten metal which is cooled down gradually to produce high-quality metal objects. The simulated annealing algorithm is very effective at finding an acceptably good solution in a limited amount of time. The algorithm is also good at approximating an acceptable placement solution for a netlist to be placed on an FPGA. A wire length cost function is used to measure the quality of the placement. Netlist instances are initially placed randomly on the FPGA. Different instance moves are made to gradually improve the quality of the placement.

Routing:

Once the instances of a netlist are placed on FPGA, connections between different instances are routed using the available routing resources. The FPGA routing problem consists of routing signals (or nets) in such a way that no more than one signals use the same routing resource. PathFinder routing algorithm is commonly used for FPGAs. In order to perform routing on FPGA architecture, the routing architecture is initially modeled as a directed graph where different nodes are connected through edges. Each routing wire of the architecture is represented by a node, and the connection between two wires is represented by an edge. When a netlist is routed on the FPGA routing graph, each net (i.e., connection of a driver instance with its receiver instances) is routed using congestion driven Dijkstra's "Shortest Path" algorithm. Once all nets in a netlist are routed, one routing iteration is said to be completed. At the end of an iteration, there can be conflicts between different nets sharing the same nodes. The congestion parameters of the nodes are updated, and routing iterations are repeated until routing converges to a feasible solution (i.e., all conflicts are resolved) or routing fails (i.e., maximum iteration count has reached, and few routing conflicts remain unresolved).

Timing analysis:

Timing analysis is not a part of the actual software flow since it does not alter the design in any way, but instead, it provides important reports that are crucial to the success of the final result. Timing analysis is used for two basic purposes:

- To determine the speed of circuits which have been completely placed and routed.
- To estimate the slack of each source-sink connection during routing (placement and other parts of the CAD flow) in order to decide which connections must be made via fast paths to avoid slowing down the circuit.

First, the circuit under consideration is presented as a directed graph. Nodes in the graph represent input and output pins of circuit elements such as LUTs, registers and I/O pads. Connections between these nodes are modeled with edges in the graph. Edges are added between inputs of combinational logic blocks (LUTs) and their outputs. These edges are annotated with a delay corresponding to the physical delay between the nodes. Register input pins are not joined to register output pins. To determine the delay of the circuit, a breadth-first traversal is performed on the graph starting at sources (input pads, and register outputs). Then the arrival time which is the time elapsed for a signal to arrive at a certain point is computed with the following equation:

$$T_{arrival}(i) = max_{j\in fanin(i)}\{T_{arrival}(j) + delay(j,i)\}$$

where node i is the node currently being computed, and $delay(j,i)$ is the delay value of the edge joining node j to node i. The delay of the circuit is then the maximum arrival time, $D_{max,}$ of all nodes in the circuit.

To guide a placement or routing algorithm, it is useful to know how much delay may be added to a connection before the path that the connection is on becomes critical. The amount of delay that may be added to a connection before it becomes critical is called the slack of that connection. To compute the slack of a connection, one must compute the required arrival time, $T_{required}$, at every node in the circuit. We first set the $T_{required}$at all sinks (output pads and register inputs) to be D_{max}. Required arrival time is then propagated backwards starting from the sinks with the following equation:

$$T_{required}(i) = min_{j\in fanout(i)}\{T_{required}(j) - delay(j,i)\}$$

Finally, the slack of a connection (i,j) driving node, j is defined as:

$$Slack(i,j) = T_{required}(j) - T_{arrival}(i) - delay(i,j)$$

Bitstream Generation:

Once a netlist is placed and routed on an FPGA, bitstream information is generated for the netlist. This bitstream is programmed on the FPGA using a bitstream loader. The bitstream of a netlist contains information to program the SRAM bits of Look-Up Tables. The routing information of a netlist is used to correctly program the SRAM bits of connection boxes and switch boxes.

2.3 Partial Reconfiguration.

The configuration possibilities offered by FPGAs created a new paradigm in digital circuit design since the same device (i.e., the same hardware) can be adapted to provide different functions by just reconfiguring it. In other words, a device may implement different functions in the course of its operation, allowing it to be adapted to different operating conditions in response to modifications in the required functionality, changes in the environment, or even faults that might take place, therefore allowing its usability to be extended.

When hardware reconfiguration capabilities are required for a given application, the use of FPGAs offers many advantages and opportunities. Although reconfigurable systems are not limited to just FPGA-based ones, these are the most significant at a commercial level. Other possibilities exist, based on custom devices with specific reconfiguration features, mainly oriented toward reconfigurable computing systems. However, these devices are intended to overcome some limitations of FPGAs in very specific areas, for instance ultrafast reconfiguration time (i.e., reconfiguring a complete device in just one clock cycle). Coarse-grained reconfigurable architectures are an example of this, where different functions are implemented in the same silicon die so that the resulting system may be adapted to changing conditions. These solutions have limited flexibility because the functions are decided at design time and can be neither be changed nor modified once the device is manufactured. On the other hand, the configurability of most FPGAs of truly reconfigurable devices in general allows the functions to be performed by the system to be adapted at any moment during its lifetime, even during infield operation.

Based on the ability of reconfiguration a new concept emerged. Partial reconfiguration takes it one step further and allows the dynamic modification of part of an operating FPGA design without impacting the rest of the design. Summing all the architectures so far we can classify them with respect to their configuration capabilities, as illustrated in Figure 5. At the highest level, FPGAs can be separated into one-time configurable devices that can only be used as an ASIC substitute and configurable FPGAs. Configurable FPGAs can in turn be distinguished in globally and partially reconfigurable devices. When globally reconfiguring an FPGA, the complete device configuration is exchanged. As a consequence, all the internal states get lost and the FPGA will have to restart its operation. This is appropriate for an in-field update of the FPGA but is unlikely for self-adapting reconfigurable systems. For the purposes of this thesis, we will focus mostly on partial reconfiguration.

From the perspective of the design functionality, partial reconfiguration can be split into parts:

- Static Partial reconfiguration (passive), the FPGA is stopped during the reconfiguration process. While the partial portion of data is sent into the device, the rest of the FPGA is in the shutdown mode. The device is brought up after the partial configuration is completed.
- Dynamic partial reconfiguration (active), the FPGA is active during the reconfiguration process. While the part of the device is being reconfigured, the rest of the FPGA is still running. In other words, dynamic partial reconfiguration permits configuration of the FPGA without stopping of the device.

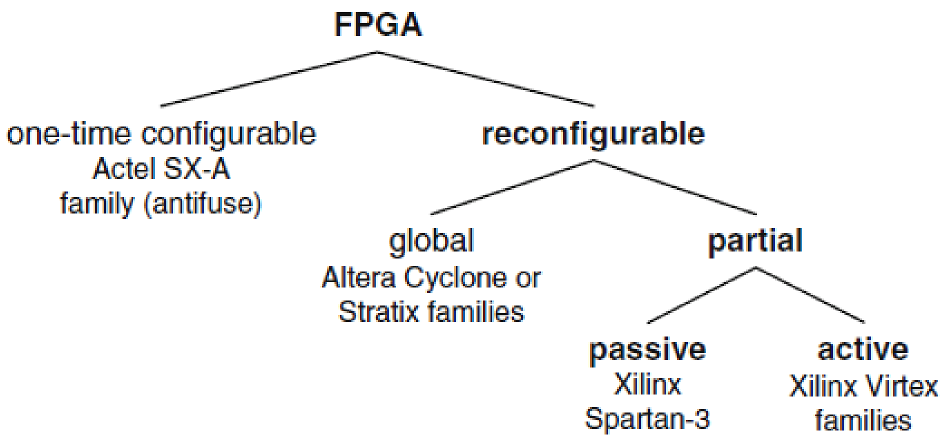


Figure 5 Classification of FPGAs by their configuration capabilities. [3]

2.3.1 Partial Reconfiguration Fundamentals

All partial reconfiguration designs consist of three basic parts. The Static is portion of the design that does not change and is expected to continue to function at all times. The Reconfigurable Partition is the instance or level of hierarchy within which multiple Reconfigurable Modules are defined and implemented. Each Reconfigurable Module represents one of the time-multiplexed functions that will be switched in and out of the FPGA (Figure 6).

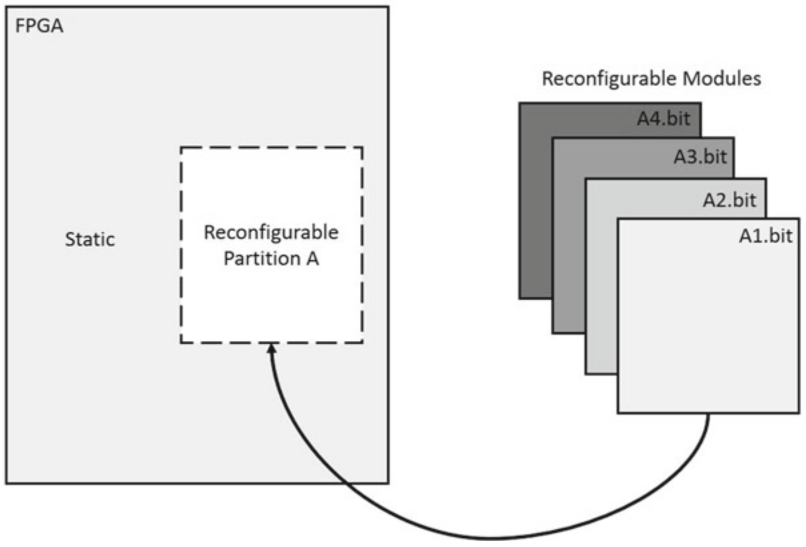


Figure 6 Basic partial reconfiguration concept and terminology. [5]

Partial reconfiguration designs can contain one or more reconfigurable partitions, each of which must occupy a mutually exclusive physical area of the FPGA. The physical area for a given reconfigurable partition must contain the aggregated resources required to individually implement each of the reconfigurable modules associated with it. The resource types and granularity of the physical area within the FPGA that can be reconfigured at any given time vary by device family.

Both Static and the interface points between the Static and the reconfigurable partition need to be identical for all the reconfigurable modules in the design. Vivado achieves this by preserving the static implementation and reusing it to implement subsequent reconfigurable modules. An additional innovation in Vivado is the creation of virtual I/O for each of the interface port called a Partition Pin. Partition Pins can be locked to specific anchor points within the routing tiles and maintained across reconfigurable modules. This consumes no LUTs or flip-flops, thus reducing resource overhead and timing delays at the interface.

Vivado generates a partial bitstream file for each reconfigurable module in each reconfigurable partition as well as a full bitstream which contains the data for both the static and the reconfigurable module(s) being implemented. The full bitstream is used for initial configuration of the FPGA, while the partial bitstreams are used for switching in and out of the various reconfigurable modules. Loading of partial bitstreams into the FPGA is generally performed via the FPGA’s standard external configuration ports or via internal configuration ports which can be incorporated into the static portion of the design.

Partial reconfiguration takes advantage of the FPGA’s addressable configuration infrastructure which allows specific areas of the FPGA to be configured. The smallest addressable segment of the FPGA is known as a configuration frame. Each frame typically corresponds to a single column of resource type, for example, DSP, block RAM, CLB, or routing interconnect; the actual number of resources in each frame depends on the resource type and varies by device family.

2.3.2 Configuration Mechanisms

Storing and managing partial bitstreams is key to the success of partial reconfiguration in a design. Storage of partial bitstreams is typically outside the FPGA, either on a nonvolatile flash memory on the board or on another remote medium, and accessible to the FPGA via PCIe, Ethernet, SD card, or other data transfer protocol. Managing these partial bitstreams can be done using an external processor or an internal state machine or processor within the static region of the FPGA. The processor or state machine determines which reconfigurable module should be loaded, where the partial bitstream for that reconfigurable module resides as well as when and how it will be downloaded into the FPGA's configuration memory. IP blocks provided by manufacturers such as the partial reconfiguration controller IP from Xilinx can also be used to help manage the partial bitstream configuration.

Depending on the location of the partial bitstreams and the management engine used, various configuration ports can be used to configure the FPGA. The following are the available configuration ports:

- ICAP (internal configuration access port): The primary choice where configuration management is being done internally to the FPGA. This requires a controller as well as logic to drive the ICAP interface.
- MCAP (media configuration access port): Provides access to configuration memory from one specific PCIe block only in UltraScale devices.
- PCAP (processor configuration access port): The primary configuration mechanism for Zynq-7000 SoC designs.
- JTAG: Test and debug port. Mainly driven by the Vivado Hardware Manager.
- Slave SelectMAP or slave serial: A good choice to perform full and partial reconfiguration, especially when using an external processor.

2.3.3 Floorplaning for Reconfigurable Designs

While static-only FPGA-based systems may be implemented without additional area constraints in a completely automatic fashion with the help of a placer tool, floorplanning is essential when designing runtime reconfigurable systems. In this case, all partial modules have to be constrained into bounding boxes and during the implementation of the static part of the system, the assigned reconfigurable area must be prohibited for implementing any static logic or routing.

Floorplanning typically starts with a preliminary synthesis run for budgeting the resource requirements of the static system and of all reconfigurable modules individually. Based on this initial resource budgeting, the target FPGA can be selected. On this target, one or more reconfigurable areas have to be defined that will be tiled into a grid of resource slots. In this step, the system is split into a static part and a dynamically reconfigurable part containing the FPGA resources that are shared by multiple modules over time. These slots will accommodate the partial modules and the grid must provide sufficient resources to fulfill all logic and memory requirements.

2.3.4 Reconfigurable Families

From the commercial point of view, major manufacturers embraced this ability and offered reconfiguration support on their devices. Altera's Excalibur was the company's first devices that allowed the whole FPGA fabric to be dynamically configured from the on-chip hard processor at any moment, by retrieving the corresponding bitstream from an external nonvolatile memory. Later, some Altera devices started to provide limited partial reconfiguration capabilities by allowing specific elements, such as serializers/deserializers or PLLs, to be reconfigured. More recently, Altera V devices (Stratix V, Arria V, and Cyclone V families) extended the support for partial reconfiguration.

A different approach is used by Atmel's FPGAs, which implement partial run-time-reconfiguration through cache logic designs, where part of the FPGA fabric can be reconfigured without loss of register data, while the remainder of the fabric continues to operate without disruption. The main drawback of these in addition, small FPGAs in this context is that the reconfiguration access method is bit based, which requires very low-level reconfiguration control, although it has the advantage of providing very high flexibility.

Most Xilinx SRAM-based FPGAs can be partially reconfigured. This is the reason why they are used in the majority of applications where this feature is required. Their configuration bitstream format allows a designer to modify one or more configuration packets and perform partial reconfiguration by accessing specific portions of the FPGA configuration memory. Each Xilinx device family has different reconfiguration features:

- The low-cost Spartan 3 series supports the reconfiguration of entire columns, including top and bottom I/O blocks. The first Spartan 3 family does not include an ICAP, and thus it is not well suited to designing self-reconfigurable systems.
- All Xilinx high-performance FPGA families provide glitch-less reconfiguration and include an ICAP. Virtex-II and Virtex-II Pro families implement column-based reconfiguration, whereas in the more recent families (Virtex-4, Virtex-5, and all Series 7 families: Artix, Kintex, Virtex, Zynq, and UltraScale), reconfiguration frames do not span entire columns, but several rows are associated with clock domains that are horizontally laid across the FPGA layout. As for clock domains, frames for different families are of different sizes (16 rows for Spartan-6, 20 for Virtex-5, 40 for Virtex-6, 50 for Zynq and former series 7 devices, and up to 60 for UltraScale).
- Some devices have double ICAP support, which may be useful for increased fault tolerance. Zynq devices also have a PCAP, controlled from the processing system, in addition to the conventional ICAP.

Device improvements in this area are slow, mostly pushed by the research community's efforts regarding architectures, tools, and applications.

2.3.5 Benefits of Partial Reconfiguration

In the early days of field programmable gate arrays, the available logic capacity was very limited and using runtime reconfiguration had been suggested to raise resource utilization or to squeeze larger circuits into the available logic. For example, the dynamic instruction set computer (DISC) is capable of changing its instruction set at runtime according to a running program. Consequently, only the currently used instructions are loaded on the FPGA which takes fewer resources than having a system providing all instructions at the same time. However, for decades implementing such systems required deep knowledge about the used FPGA architecture and had to follow an error prune difficult design flow. Furthermore, long configuration time was another crucial issue that detained the use of runtime reconfiguration in industrial applications.

With the progress in silicon process technology, logic capacity raised steadily while getting cheaper (and often more power efficient per logic cell) at the same time. The capacity for the FPGAs of the Virtex family from Xilinx increased from 93k LUTs [6] on Virtex II XC2V8000 device to 1.2 million LUTs [7] on Virtex 7 XC7V2000T device. Respectively the Stratix family from Altera increased from 79k LUTs [8] on Stratix EP1S80 device to 0.7 million LUTs [9] on Stratix V 5SGXA device. The explosion in capacity removed the pressure on the FPGA vendors to add better support for runtime reconfiguration in their tools and devices. However, by heading beyond 1M LUT devices (million look-up table FPGAs), things are changing dramatically at the moment.

For present high capacity FPGAs, the configuration time required to write tens of megabytes of initial configuration data is too long for many applications and partial reconfiguration can be used to speed up the system start.

A further consequence of having large high-density FPGAs is their higher risk of failure due to single event upsets (SEU). However, SEUs can be detected and compensated with the help of partial runtime reconfiguration (e.g., using configuration scrubbing). Another factor arising for current high capacity FPGAs is a strong relative increase in the static power consumption. The static power consumption is related directly to the device capacity. With the help of partial runtime reconfiguration, a system might be implemented on a smaller and consequently less power hungry device. This will further result in a cheaper system.

The factors fast system start, SEU recovery, and static power consumption force FPGA vendors to enhance the support for runtime reconfiguration. As a consequence, partial reconfiguration will be available in the majority of future FPGA devices and it will be better supported by the corresponding tools.

Besides the mentioned functional aspects, the design productivity of complex multi-million system gates designs can be substantially improved by closely integrating partial design methodologies into the standard FPGA design flow. While present tools support to render the logic synthesis process to a compute farm for fast parallel processing, the place and route phase is mostly implemented as a global optimization process. As a consequence, place and route can take many hours or even days to finish, hence making design respins very costly. However, it is possible for the physical implementation of a design to be precisely constrained for individual modules. In other words we can specify all the resources (including wires of the fabric) that are allowed for the place and route step of each module. This would permit to carry out the place and route process for multiple modules in parallel.

Additionally to application benefits there are some generic advantages of partial reconfiguration to expand such as area and power reduction and performance improvement. Due the increased leakage current in deep sub-micrometer CMOS process technology, static power consumption will dominate the total power consumption. By enhancing device utilization with the help of partial runtime reconfiguration, power and monetary cost could be saved. This is essential when following the trend towards more and more mobile devices. Furthermore, low power consumption results in also lower cooling effort.

An example of a self-adaptive system featuring a mobile internet device is illustrated in Figure 7. The system provides a software-defined radio part (SDR), different decryption modules, and protocol processing accelerators for various protocols.

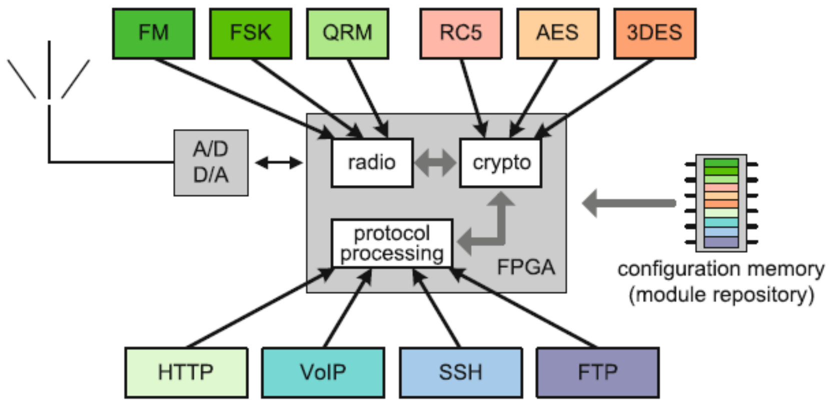


Figure 7 Area saving by reconfiguring only the currently required accelerator module to the FPGA. Configurations are fetched from the module repository at runtime. [5]

Assuming that the radio part will be adjusted according to the available bandwidth, and that the crypto and protocol processing accelerators are changed on-demand. We can then save substantial FPGA resources by not providing all variants in parallel, but by only loading the currently required modules to the device. This requires that the accelerator modules are either needed exclusively or that the system can time-multiplex the reconfigurable resources by sufficient fast reconfiguration. However, for low power operation, it should be mentioned that the configuration itself requires some power. This includes the power to fetch a configuration from the configuration memory and the power required by the FPGA itself for the configuration. Furthermore, it should be noted that the reconfigurable part will consume static power without providing useful work during the whole configuration process. If we assume that the system changes its operation modes on human interaction, the update rate will be sufficiently low such that it easily amortizes the configuration power.

There are more potential benefits than only power and cost savings. If we can implement a system with a smaller FPGA, we might be able to use a smaller package which then permits to build the mobile device smaller. Or in the case of more complex systems that demand multiple FPGAs, runtime configuration might be used to achieve a higher level of integration with the goal to save complete FPGAs. This might further simplify the PCB design and also help to save further power. The latter is possible because due to higher integration, we will typically have more on-chip and less off-chip communication thus saving power.

Beyond resource and power savings, partial reconfiguration can help to increase system performance. In most cases, this is achieved by spending more resources for specific accelerator modules. With more area, we can exploit a higher level of parallelism, which in turn permits a faster processing of the accelerator modules. If different accelerators are used sequentially (i.e., different accelerators are executed mutually exclusive in the time domain), we can exploit runtime reconfiguration in order to provide more resources per accelerator.

Therefore resulting in faster processing of the individual algorithms as compared to a static only system. This method is visualized in Figure 8 where on the left the resources are split into three tasks resulting in lower performance and area inefficiency since we are using only a third of the resources at any given time. On the other hand on the right by exploiting runtime reconfiguration we dedicate more area in each task, therefore, improving performance and area utilization.

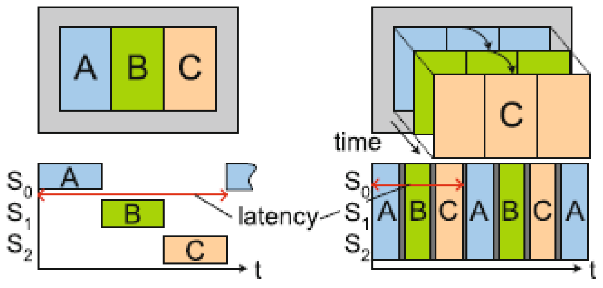


Figure 8 System acceleration due to partial reconfiguration. [5]

Even considering the configuration overhead before a new module can be started, this can result in a substantially lower overall latency. If we assume that the modules are executed periodically, we see that the reduced latency allows higher throughput of the system. A practical example for this method would be a hardware accelerated secure SSL connection. In this protocol, we firstly have to exchange a session key which is computed using asymmetric key exchange. After this, the session key is used by a symmetric (de-)cipher for the actual data transfer. Consequently, both steps are executed strictly sequential and we can apply the method that is shown in Figure 8 for reducing latency.

Partial reconfiguration allows to trade-off computation arbitrary in time and space. However, there are algorithms that scale differently with respect to these domains. For example, matrix multiplication of very large matrices scales superlinear in the space domain. This means that by spending more area, the speed-up will be typically proportionally higher than the relative area increase. For instance, if we spend twice the resources, we can typically compute a matrix multiplication in less than half the time. The reason for this is that very large matrices will not fit entirely into the on-FPGA memory. Consequently, the computing space has to be tiled into smaller portions and switching from one tile to another will introduce some overhead. By spending more area, we need less tiles, and this will reduce some of that overhead. For huge matrices, we can use runtime reconfiguration to compute larger tiles at lower latency. This can speed-up the overall system, as illustrated in Figure 8. The trade-off between processing in time and space works nicely (within the throughput limits) for many compute-intensive algorithms, including motion estimation, crypto accelerators, FFT, or FIR filters.

Partial reconfiguration can be exploited multiple times and a performance improvement is not limited to the pure data processing part of a system. Recalling the SDR example shown in Figure 7 and assuming that we have to update or add some accelerator modules from time to time during the lifetime of the system. In this case, we only have to upload a smaller partial configuration bitfile rather than a full FPGA configuration file. Moreover, this permits the system to stay active during the whole process without a time-consuming system restart after the update. Speeding-up system upgrades is in particular of interest for distributed embedded systems (e.g., automotive control units). In these applications, slow field buses are often used to transfer the update data. In some cases, the time to upgrade can be essential. For example, if a low orbit satellite has only a low-speed upstream link and if the radio link can only be established during limited time intervals passing a ground control station.

2.4 Processors on FPGAs

Considerable amount of FPGA area can be reduced by incorporating a microprocessor in an FPGA. A microprocessor can execute any less compute-intensive task, whereas compute-intensive tasks can be executed on an FPGA. Similarly, a microprocessor based application can have huge speed-up gains if an FPGA is attached with it. An FPGA attached with a microprocessor can execute any compute-intensive functionality as a customized hardware instruction. These advantages have compelled commercial FPGA vendors to provide microprocessors in their FPGAs so that a complete system can be programmed on a single chip. Few vendors have integrated fixed hard processor on their FPGA like AVR Processor integrated in Atmel FPSLIC or PowerPC and ARM processors embedded in Xilinx devices.

Others provide soft processor cores which are highly optimized to be mapped on the programmable resources of FPGA. Altera’s Nios and Xilinx’s Microblaze are soft processors meant for FPGA designs which allow custom hardware instruction. For the purposes of this thesis we will expand on soft processors and more specific on Xilinx’s Microblaze.

2.4.1 Soft Processors

Soft processors make use of the FPGA fabric for implementing the processors. For low speed (200 MHz and below), soft processors are a good option. These come in multiple flavors and are user configurable. Depending on the nature of the application, you can choose to trim down the functionality from the processors.

At the lowest end, you can implement an 8-bit processor with bare minimal instruction set. One example of such a processor is PicoBlaze from Xilinx. This processor is a good replacement for state machines. PicoBlaze does not have a compiler toolchain and hence requires the program to be written in assembly language. This program is stored in the local memory available on the FPGA as a memory store. The simplicity of the architecture enables a processor which can be implemented in just about 26 slices.

As a step up, Xilinx introduced a 32-bit highly configurable processor named Microblaze in early 2000. This RISC-based soft processor is capable of achieving clock speed of around 400MHz on the UltraScale FPGA architectures. It supports an option of a three-stage or a five-stage pipeline, configurable cache, optional multiply and divide units, optional barrel shifter, single and double precision floating-point unit and more. Every additional feature selected in hardware will lead to usage of FPGA resources and can have impact on the max frequency (F_{max}). The choice can be made based on the needs of the user application. For example, if there are many multiply operations to be done, it is better to enable a hard multiplier. It can save over 1000 clock cycles for every multiply operation done over a software library-based solution.

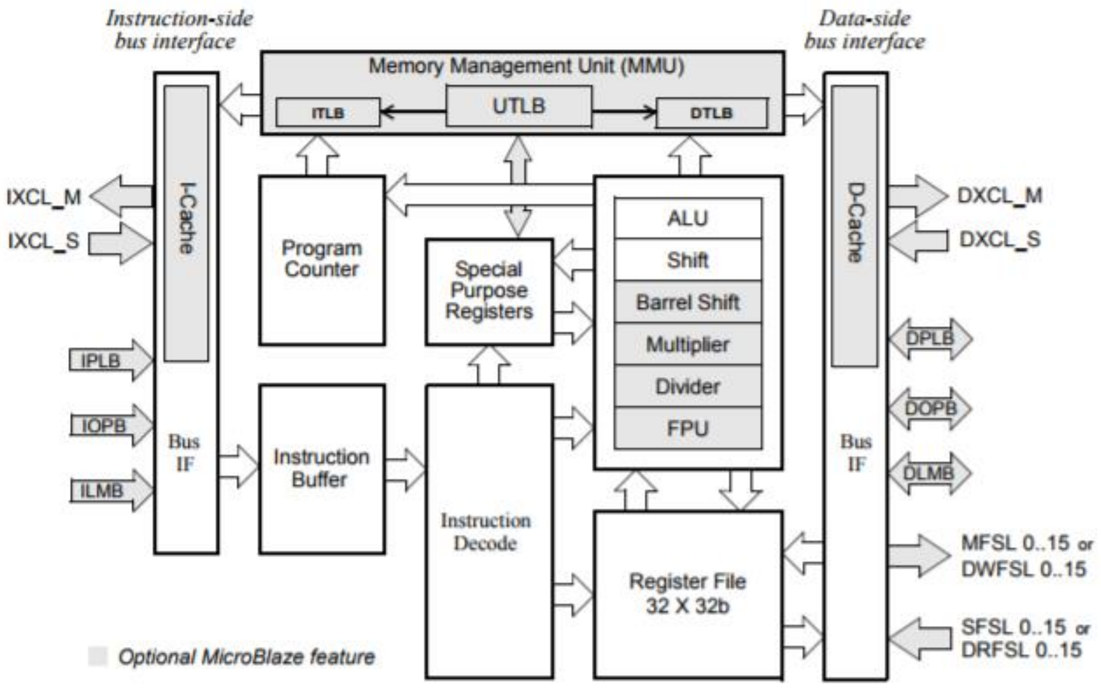


Figure 9 Microblaze core block diagram [10]

To derive the best performance, a capable soft processor core needs to be coupled with the right silicon. Xilinx offers some of the broadest and most cost-effective solutions for diverse requirements. Each device family delivers the best value for its target applications:

- Spartan-6 FPGA: With best-in-class I/O ratios and form factor, the Spartan-6 family is ideal for simple to moderately complex bridging applications or companion co-processing chips commonly found in infotainment, consumer, and industrial automation. The Microblaze processor can be optimized to complement existing or preferred host processors for rapid extension of system interfaces, peripherals, or processing capabilities with minimal development effort.
- Artix-7 FPGA: For FPGA applications that demand more advanced functionality, the 28nm-based Artix-7 family offers exceptional performance per watt. The Artix-7 family leads the industry in nearly every aspect of performance in low-end device, including logic fabric performance, memory line rates, signal processing bandwidth, and particularly transceiver line rates. With 30% greater performance than Spartan-6 FPGA due to fabric, and support for more advanced serial protocols, Artix-7 devices can be leveraged for more advanced embedded tasks.
- Zynq-7000 AP SoC: For applications that demand the robust processing power of a dual-core ARM Cortex-A9 processor, the low-end Zynq-7000 AP SoC devices provide the highest level of system integration of the three families by fusing an ARM APU subsystem with the Artix-7 device fabric. Multiple Microblaze processor cores can be implemented in the fabric and co-exist with the Cortex-A9 as a form of offloading less demanding tasks, such as system management and input monitoring.

In addition to the soft processors from Xilinx, one can also build his own processors or produce one from IP vendors and open source. There have been implementations of ARM done on Xilinx FPGAs in academia as well as industry. There have also been implementations of processors with reduced instruction set targeted toward specific applications. Microblaze has been optimized for FPGA implementation and is usually better suited for both resource count as well as F_{max}.

2.5 Zynq-7000 Architecture

The Zynq-7000 family is based on the Xilinx All Programmable System On a Chip (AP SoC) architecture. These devices integrate a feature-rich dual or single-core ARM Cortex A9 MPCore based processing system (PS) and Xilinx programmable logic (PL) in a single device, built on a state-of-the-art, high performance, low-power, 28nm process technology. The ARM Cortex-A9 MPCore CPUs are the heart of the PS which also includes on-chip memory, external memory interfaces, and a rich set of I/O peripherals.

All Zynq devices have the same basic architecture, and all of them contain, as basis of the processing system, a dual-core ARM Cortex-A9 processor. This is a hard processor meaning it exists as a dedicated and optimized silicon element on the device, opposite to the soft processors discussed earlier. In general, the advantage of soft processors is that the number and precise implementation of processor instances is flexible. On the other hand, hard processors can achieve considerably higher performance, as is true for Zynq's ARM processor.

It is worth noting that one or more Microblaze soft processors can be used within the PL portion of the Zynq, to operate in conjunction with the ARM processor. The Microblaze instances may have, for example, the role of coordinating specific low-level functions within the system; less demanding tasks which can be delegated away from the main ARM Cortex-A9 processors to enhance overall performance. In other words, the presence of the ARM processor in the system does not preclude the use of soft processors, and indeed many applications may benefit from employing a processing model as this, which uses both types. Figure 10 illustrates the positions of the ARM and Microblaze processors on the Zynq device, the ARM as a dedicated resource, and the Microblaze located in the logic fabric. It also illustrates the three key parts that define the Zynq-7000 architecture: PS, PL and the connections between them.

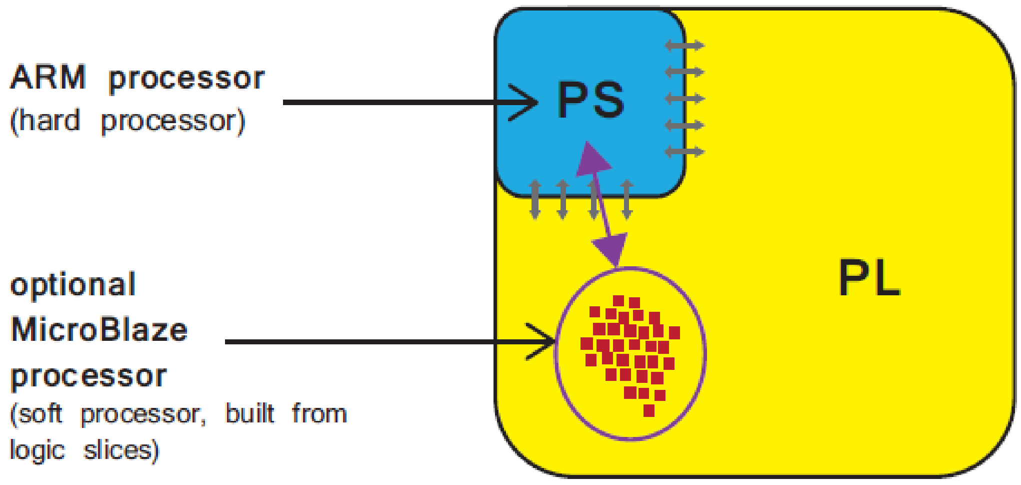


Figure 10 Locations of hard (ARM Cortex-A9) and soft (Microblaze) processors on a Zynq device [11]

2.5.1 Processing System

The Zynq processing system includes not just the ARM processor, but a set of associated processing resources forming an Application Processing Unit (APU), and further peripheral interfaces, cache memory, memory interfaces, interconnect, and clock generation circuitry. A block diagram showing the architecture of the PS is shown in Figure 11, where the APU is on the top right corner.

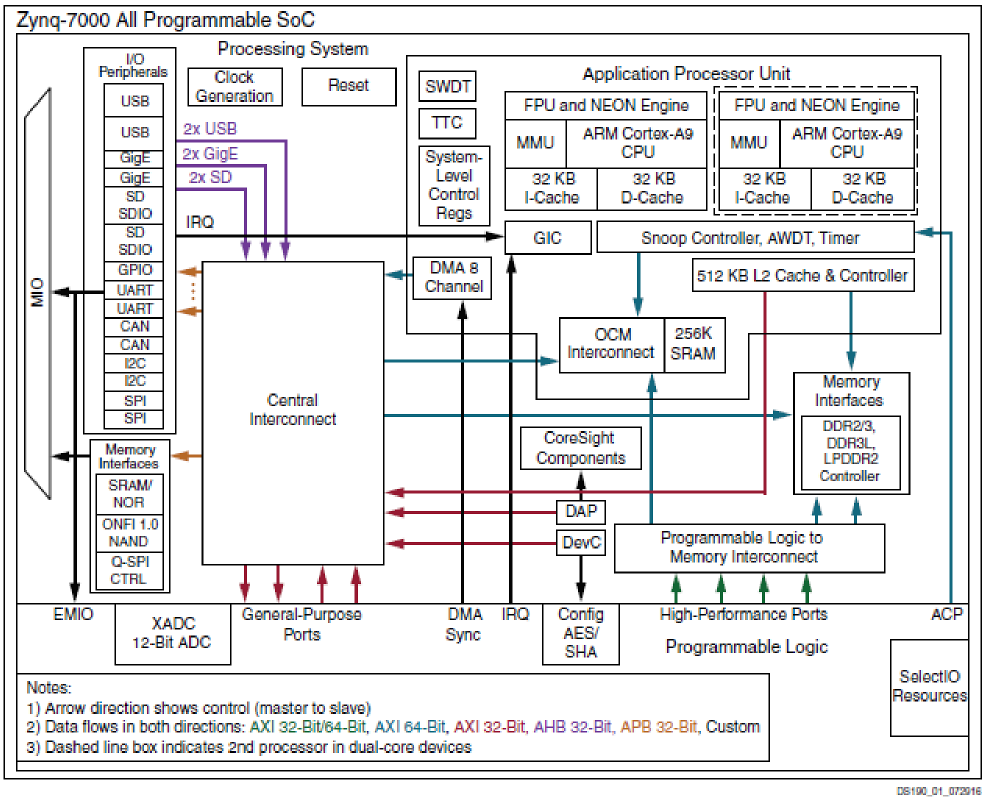


Figure 11 Zynq-7000 PS Overview [12]

The APU is primarily comprised of two ARM processing cores, each with associated computational units: a NEON Media Processing Engine (MPE) and Floating Point Unit (FPU), a Memory Management Unit (MMU), and a Level 1 cache memory (in two sections for instructions and data). The APU also contains a Level 2 cache memory, and a further On-Chip Memory (OCM). Finally, a Snoop Control Unit (SCU) forms a bridge between the ARM cores and the Level 2 cache and OCM memories; this unit also has some responsibility for interfacing with the PL.

The ARM Cortex-A9 can operate at up to 1GHz, depending on the particular Zynq device. Each of the two cores has separate Level 1 caches for data and instructions, both of which are 32KB. As in the general case, this permits local storage of frequently required data and instructions for fast access times and optimal processor performance. The two cores additionally share a larger Level 2 cache of 512KB for instructions and data, and there is a further 256KB of on-chip memory within the APU. The primary role of the MMU is to translate between virtual and physical addresses.

The Snoop Control Unit undertakes several tasks relating to interfacing between the processors and Level 1 and 2 cache memories (‘snooping’ is one of several mechanisms for ensuring cache coherency, i.e. managing the consistency of data across shared cache resources). The SCU is responsible for maintaining memory coherency between the processor data cache memories, which are marked as D-Cache and L2 Cache in Figure 11. It also initiates and controls access to the Level 2 cache, arbitrating between requests from the two cores where necessary. The SCU additionally manages transactions that take place between the PS and PL via the Accelerator Coherency Port (ACP).

From a programming perspective, support for ARM instructions is provided via the Xilinx Software Development Kit (SDK) which includes all necessary components to develop software for deployment on the ARM processor. The compiler supports the ARM and Thumb instruction sets (16-bit or 32-bit), along with 8-bit Java bytecodes.

As additional functionality to the main ARM processor, the NEON engine provides Single Instruction Multiple Data (SIMD) facilities to enable strategic acceleration of media and DSP type algorithms. NEON instructions are an extension to the standard ARM instruction set and can either be used explicitly or by ensuring that the C code follows an expected from and thus allows NEON operations to be inferred by the compiler. As the SIMD term suggests, the NEON engine can accept multiple sets of input vectors, upon which the same operation is performed simultaneously to provide a corresponding set of output vectors. This style of computation caters well to applications like image and video processing, which operate on a large number of data samples (pixels) simultaneously, and inherently parallel, generic signal processing functions such as Finite Impulse Response (FIR) filters and Fast Fourier Transforms (FFTs).

Communication between the PS and external interfaces is achieved primarily via the Multiplexed Input/Output (MIO), which provides 54 pins of flexible connectivity, meaning that the mapping between peripherals and pins can be defined as required. Certain connections can also be made via the Extended MIO (EMIO), which is not a direct path from the PS to external connections, but instead passes through and shares the I/O resources of the PL. These are both shown on the left-hand side of Figure 11. The EMIO can be used when extension beyond 54 pins is required, or as a method of interfacing the PS with an IP block implemented in the PL.

The available I/O includes standard communication interfaces, and General Purpose Input/Output (GPIO) which can be used for a variety of purposes including simple buttons, switches, and LEDs.

2.5.2 Programmable Logic

The second part of the Zynq architecture is the programmable logic. It is composed mostly of general purpose FPGA fabric such as CLBs, LUTs, Programmable Interconnects and Input/Output blocks alongside with special resources like DSPs and Block RAMs (Figure 12).

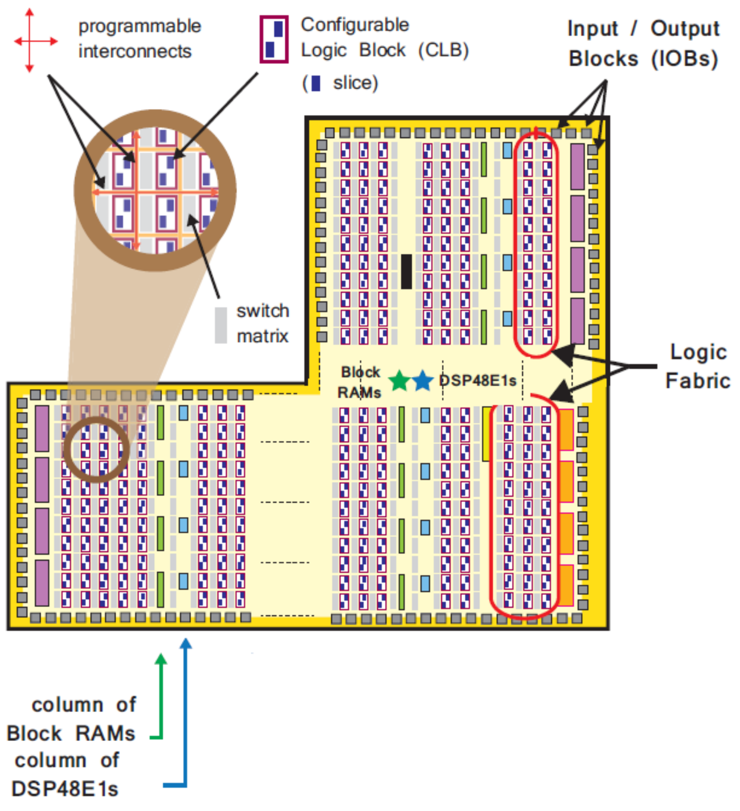


Figure 12 The programmable logic fabric and its composing elements. [11]

The block RAMs in the Zynq-7000 are equivalent to those on Xilinx 7 series FPGAs, and they can implement Random Access Memory (RAM), Read-only Memory (ROM), and First In First Out (FIFO) buffers, while supporting Error Correction Coding (ECC).

Each Block Ram can store up to 36Kb of information, and may be configured either as one 36Kb RAM, or two independent 18Kb RAMs. The default word size is 18 bits, and in this configuration each RAM comprises 2048 memory elements. The RAM can also be 'reshaped' such that it contains more, smaller elements (for example 4096 elements x9 bits, or 8192 x 4 bits), or alternatively, fewer, longer elements (e.g. 1024 elements x 36 bits, 512 x72 bits). Larger capacity memories can be formed by combining two or more Block RAMs together.

Using a Block RAM means that a large amount of data can be stored in a small physical space on the device, within a dedicated and optimized memory element; the alternative is Distributed RAM, which is constructed from the LUTs within the logic fabric. A significant number of LUTs (spanned over a larger area) are required to form a memory of comparable size to a Block RAM, and the resulting implementation suffers from restricted timing performance due to the increased small memories using distributed RAM, both for resource efficiency, and because their placement is more flexible (distributed memories can be located close to the components that interact with them, which can result in fast timing performance too). Block RAMs can normally be clocked at the highest clock frequency supported by the device.

The LUTs in the logic fabric can be used to implement arithmetic operators of any arbitrary length, but are most suitable for arithmetic operators with short wordlengths (arithmetic circuits for long wordlengths can have a large footprint in slice logic, with placement and routing factors resulting in sub-optimal clock frequencies).

When designing with Zynq, it makes sense to identify deterministic, computationally parallel functions and implement them in the PL sections of the device, specifically targeting DSP and Block RAM resource where possible. In this way, the PL can be used to accelerate algorithms residing in the PS. There are many conceivable examples where the availability of the PL directly adjacent to the processor, and the opportunity to allocate certain system functions to the PL, can bring significant benefits to the overall system implementation.

Communication within the logic fabric is achieved with the general purpose input/output facilities (IOBs). On Zynq IOBs are collectively referred to as SelectIO Resources, and are organized into banks of 50 IOBs each. Each IOB contains one pad, which provides physical connection to the outside world for a single input or output signal.

The I/O banks are categorized as High Performance (HP) or High Range (HR), and these support a variety of I/O standards and voltages; the HP interfaces are limited to voltages of 1.8V and are typically used for high-speed interfaces to memory and other chips, while the HR interfaces permit voltages of up to 3.3V and cater for a wider variety of IO standards. Both single-ended and differential signaling are supported, requiring 1 IOB and 2 IOBs per connection, respectively. Each IOB also includes an IOSERDES resource for programmable conversion between parallel and serial data formats (serialization and deserialization), of between 2 and 8 bits.

Summarizing the remaining external interfaces to the PL:

- Analog to Digital Conversion – The PL includes another hard IP component: the XADC block. This is a dedicated set of Analogue to Digital Converter (ADC) mixed-signal hardware, which features two separate 12-bit ADCs both capable of sampling external analog input signals at 1Msps.
- Clocks – The PL receives four separate clock inputs from the PS, and additionally has the facilities to generate and distribute its own clock signals independently of the PS.
- Programming and Debug – A set of JTAG ports are provided in the PL section to facilitate configuration and debugging of the PL. Although more secure methods are normally preferred in deployment, JTAG configuration is often used during the development phase. The facilities offered via JTAG support debugging with both ARM and Xilinx tools.

2.5.3 Processing System — Programmable Logic Interfaces

The appeal of Zynq lies not just in the properties of its constituent parts, the PS and the PL, but in the ability to use them in tandem to form complete, integrated systems. The key enabler in this regard is the set of highly specified AXI interconnects and interfaces forming the bridge between the two parts. There are also some other types of connections between the PS and PL, in particular EMIO.

Most of these connections are based on the AXI standard. AXI stands for Advanced eXtensible Interface, and the current version is AXI4, which is part of the ARM AMBA 3.0 open standard. The AMBA standard was originally developed by ARM for use in microcontrollers, with the first version being released in 1996. Since then, the standard has been revised and extended, and it is now described by ARM as the de facto standard for on-chip communication. The focus is now on System-on-Chip, including SoCs based on FPGAs or, in the case of Zynq, a device which includes FPGA fabric.

Support for AXI was first introduced into the Xilinx tool flow in release 12.3 of the ISE Design Suite, and extensive support is now available in the Vivado Design Suite. AXI buses can be used flexibly, and in the general sense are used to connect the processor(s) and other IP blocks in an embedded system. In fact there are three flavors of AXI4, each of which represents a different bus protocol, as summarized below. The choice of AXI bus protocol for a particular connection depends on the desired properties of that connection.

- AXI4 is memory-mapped interfaces and allows high throughput bursts of up to 256 data transfer cycles with just a single address phase.[13]
- AXI4-Lite is a light-weight, single transaction memory-mapped interface. It has a small logic footprint and is a simple interface to work with both in design and usage.[13]
- AXI4-Stream removes the requirement for an address phase altogether and allows unlimited data burst size. AXI4-Stream interfaces and transfers do not have address phases and are therefore not considered to be memory-mapped.[13]

The term 'memory mapped' is used the above descriptions, and it is useful to briefly confirm its meaning. If a protocol is memory mapped, an address is specified within the transaction issued by the master (read or write), which corresponds to an address in system memory space.

The primary interface between the PS and the PL is via a set of nine AXI interfaces, each of which composed of multiple channels. These make dedicated connections between the PL, and interconnects within the PS. An interconnect is effectively a switch which manages and directs traffic between attached AXI interfaces. There are several interconnects within the PS, some which are directly interfaced to the PL, and others are for internal use only. The connections between these interconnects are also formed using AXI interfaces. An interface is a point-to-point connection for passing data, addresses and hand-shaking signals between master and slave clients within the system.

3 Heterogeneous Multicore Systems

In order to match the ever-increasing computing needs of each processor type, the traditional response was to increase the clock frequency of the processor core. However, as processor frequencies increase, other issues such as power consumption, thermal power and the inability to find sufficient parallelism in the program became real obstacles to further advancements. Since the single-core approach reached the point that performance improvement brought diminishing returns multicore technology was the next natural step. Market leaders made a shift and started the production of embedded devices with an increasing amount of cores (processors) in which multiple cores communicate directly through shared hardware caches, providing high concurrency instead of high clock speed. As a result instead of having a super processor that operates in high frequency and drains a lot of power, it became possible to achieve comparable performance through multiple smaller processors that operate at lower frequency thus consuming less power.

In the same manner, the System-on-Chip (SoC) technology evolved over the years to fabricate Multiprocessor System-on-Chip (MPSoC) by putting together multiple processing elements, memory hierarchy, I/O components and an on-chip interconnect. This advancement was important for performance since most of the applications for which MPSoCs are used have precise requirements. In traditional computing we care about speed but not about deadlines. Control systems, protocols, and most real-world systems are not just about average performance, but also tasks are done by a given deadline.

Embedded systems for applications such as video streaming require very high MIPS performance, of the order of Giga operations per second, which cannot be obtained through a single on-chip processor. For example, a broadcast quality video with a specification of 30 frames/second, 720 x 480 pixels per frame requires about 400,000 blocks to be processed per second. In a broadcast with a higher quality video where the requirement is for 60 frames/second and 1920 x 1152 pixels per frame, about 5 million blocks must be processed per second. For applications such as these, system architects resort to the use of multiprocessor architectures to get the required performance.

Most algorithms used in high-performance computing have high inherent parallelism, which can be exploited by such multiprocessor systems. The disadvantage is that the hardware architecture of multiprocessor systems is fixed at design and runtime. This means that the processor architecture and the communication infrastructure, as well as the memory architecture, are mostly optimized for one application scenario. Therefore, the user has to choose an appropriate multiprocessor for its application carefully. Moreover, then the user needs to partition the application for the chosen multiprocessor system. This application integration has to follow the given hardware architecture of the multiprocessor system, which often leads to inefficient task allocation and therefore unequal workload. Especially, if versatile algorithms are used as in image processing, some will map better and others will map worse on the chosen architecture.

3.1 Multi-Processors System-on-Chip

A typical MPSoC includes several optimized components integrated together to execute a specific application. The architecture varies with each processing element serving a specific role and is influenced by the application. As a result, there are multiple different architectures available to solve problems by taking advantage either hardware, software or both. MPSoCs are broadly categorized as homogeneous or heterogeneous:

- Heterogeneous MPSoC, also referred to Chip Multi-Processing or Multi (Many) Core Systems: these systems are composed of processing elements (PEs) of different types, such as one or several general-purpose processors, DSPs, hardware accelerators, peripherals and interconnection infrastructure like a NoC. Heterogeneous MPSoCs provide high performance under tight area and power budgets.

- Homogeneous MPSoC, in this approach, the basic PE embeds all the elements required for a SoC: one or several processors (general purpose or dedicated), memory and peripherals. This tile is then instantiated several times, and all these instances are interconnected through a dedicated communication infrastructure. Homogeneous MPSoCs are scalable, have a larger footprint and higher power consumption; hence, are more suitable for general purpose systems rather than embedded systems.

The first approach offers the best performance on power consumption trade-off and the second one is obviously more flexible and scalable but less power efficient. Due to their good power efficiency, heterogeneous MPSoC approaches are used for portable systems, and more generally for embedded systems, while homogeneous approaches are commonly used for video game consoles, desktop computers, servers, and supercomputing. In general, the limitation of all multiprocessor systems is the lack of adaptivity at design and at runtime.

For the comparison of different MPSoC, there are some fundamental objectives and metrics that can be defined. Probably the most important design objective, the performance is typically measured in terms of latency and throughput. Especially, meeting the performance constraints included by applications is highly challenging but necessary for a successfully operating device. Energy efficiency is one of the most severe design issues and platform differentiators. Especially for mobile and battery powered devices energy efficiency is essential. Unfortunately, over the last years, battery capacity has not been able to cope with the increasing performance demands, leading to a growing performance-energy gap. Cost is another metric that affects the viability of an MPSoC since many times these devices are targeted for mass production where the price must be within reasonable ranges. In contrast to the previously discussed objectives and metrics, flexibility cannot be given merely as a single value but nonetheless is important. Flexibility defines the capability to execute a specific functionality on a particular processing element.

3.1.1 Heterogeneous MPSoC Architecture

Heterogeneous MPSoCs are different from traditional embedded systems. Figure 13 provides an overview of a generic heterogeneous MPSoC, composed of a set of a general-purpose processor (CPU), several accelerators (video, audio, etc.), memory elements, peripherals and interconnection infrastructure.

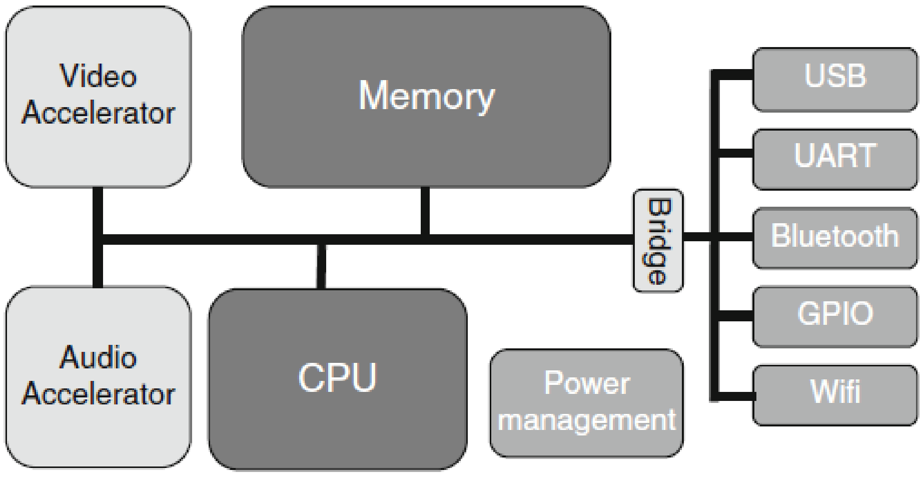


Figure 13 Simplified overview of a heterogeneous MPSoC [14]

The main motivation of these systems is that many applications, such as MPEG-2 encoder, have more than one algorithm during their execution life. This means a given application has different operations, different memory access patterns, and different communication bandwidth at different execution periods. Another example is in the advanced safety automobile devices, where multiple applications, consisting of several tasks, are executed simultaneously. Each task, invoked by applications, such as image processing, recognition, control or measurement, is assigned to a single processor core.

Heterogeneous MPSoCs provide the best performance/power efficiency tradeoffs and are a natural choice for embedded systems. The heterogeneous cores increase performance by dividing the work among well-matched cores. This requires many CPU cores for general purpose processing as well as several Single Instruction Multiple Data (SIMD) processor cores to accelerate specific performance-critical processing. The heterogeneous SOC also can save energy almost at all levels (device, circuit, and logic) of abstraction. In addition, these systems generally use irregular memory and irregular interconnection networks that also save power by reducing the loads in the whole network

The component that mostly defines the hardware architecture each time is the Processing Element. Different types range from highly flexible general purpose processors (GPPs) to dedicated hardwired accelerators optimized for a particular function. Lately, the demand for post-fabrication flexibility has led system architects to increasingly use flexible and programmable components like FPGAs, DSPs and application-specific instruction-set processors. The class of the Processing Elements can roughly be classified into the following groups:

- General Purpose Processor (GPP)
- Digital Signal Processor (DSP)
- Application Specific Instruction Set Processor (ASIP)
- Reconfigurable Application Specific Instruction Set Processor (rASIP)
- Field Programmable Gate Array (FPGA)
- Application Specific Integrated Circuit (ASIC)

Dedicated hardwired accelerators (ASICs) are specially tailored for a particular algorithm, whereas DSPs are optimized to the common characteristics of such algorithms, e.g. multiplications, multiply-accumulate and add-compare-select. ASIPs are specialized processor cores which have been specially developed for a particular algorithm or multiple ones. The key principle of ASIPs is to minimize the provided flexibility to increase performance and to minimize overheads regarding area, power, and energy consumption. To incorporate such specialized architectures software development cannot follow the general-purpose approach, as current high-level language compilers can hardly exploit such features optimally due to their irregular structure. In contrast to specialized processing elements, general purpose applications require a higher degree of flexibility. Hence, GPPs are typically, utilized for their execution and the latest techniques and architectures.

Beyond its hardware architecture, an MPSoC system is generally running a set of software applications divided into tasks and optionally an operating system devoted to managing both hardware and software through a middleware layer (e.g., drivers). An operating system can be ignored in the cases of bare metal applications where software runs directly on the processors. However due to the complexity and heterogeneity of the system the cooperation of hardware and software increase the challenges significantly to be faced by software designers. Since the processing elements are unique, the software designer needs to have expertise on all of them as entities but as a whole system too.

3.1.2 Homogeneous MPSoC Architecture

An alternative lies in building a homogeneous system based on the same programmable block instantiated several times. This architectural model is often referred to as parallel architecture model. Parallel architectures were particularly studied in Computer Science and Computer Engineering during the past 40 years. There is nowadays a growing interest for such approaches in embedded systems. The basic principle of an architecture that exhibits parallel processing capabilities relies on increasing the number of physical resources in order to divide the execution time of each resource. Theoretically, an architecture made of N processing resources may provide a speedup of at most N; however, this speedup is difficult (or impossible) to obtain in practice. Another benefit of using multiple processing elements versus a single one is that it allows decreasing the frequency correspondingly and therefore the power supply voltage.

A homogeneous MPSoC based on programmable parallel processors could provide performance thanks to the “speed-up” and a reduced power-consumption by decreasing the operating frequency and the power supply and could be considered as a real alternative to heterogeneous MPSoC. Moreover, their inherent structure if more flexible and more scalable than heterogeneous systems. Practically, exploiting the parallelism efficiently is not straightforward; flexibility and scalability could also be limited due to several factors such as the organization of the memory, the interconnection infrastructure, etc. Figure 14 illustrates a typical homogeneous MPSoC organization example.

The first famous architecture classification was proposed by Flynn [15]. He classifies according to the relationship between processing units and control units. He defines four execution models: SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), MISD (Multiple Instruction Single Data) and MIMD (Multiple Instruction Multiple Data). The SIMD model is the classical Von Neumann model, where a single processing resource executing a single instruction per unit time processes a single data flow. In SIMD architecture, a single control unit shares the data flows and distributes data to each processing resource. The MISD architectures execute several instructions simultaneously on a single data flow. Finally, several control units manage several processing units in MIMD architectures.

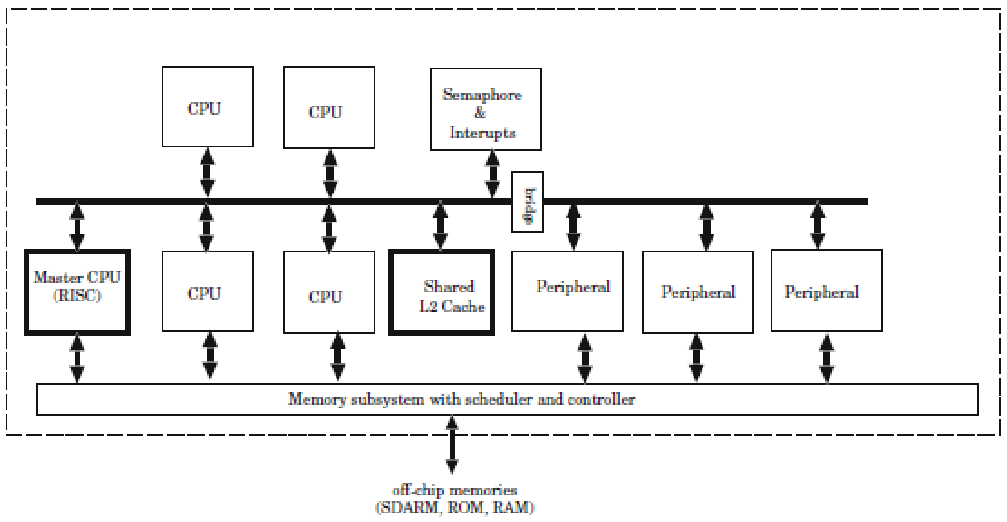


Figure 14 Homogeneous MPSoC architecture example [16]

3.1.3 Hardware Accelerators

The hardware accelerator is used on MPSoCs as a way to efficiently execute some classes of algorithms. Many applications have algorithmic functions that do not map very well to a given architecture. Hardware accelerators can be used to solve this problem. Also, a conventional storage model may not be appropriate to execute these algorithms effectively. A specialized hardware accelerator can be built to perform bit manipulation efficiently which sits next to the CPU for bit manipulation operations.

Fast I/O operations are another area where a dedicated accelerator with an attached I/O peripheral will perform better. Finally, applications that are required to process I/O peripheral will perform better. Finally, applications that are required to process streams of data do not map well to the traditional CPU architecture, especially those that implement caching systems. Specialized hardware with special fetch logic can be implemented to provide dedicated support to these data streams.

3.2 Hardware Reconfiguration in MPSoCs

With FPGAs being an option for a processing element in MPSoCs hardware reconfiguration can be achieved after fabrication by reconfiguring the FPGA or by means of partial reconfiguration. This is useful if, for example, an image processing filter needs to be exchanged, but the connection with the camera and a monitor must be preserved. If the whole FPGA were reconfigured, frames from the camera would be lost. Using partial dynamic reconfiguration instead assures that the frames of the camera will not be lost, because the camera interface module on the FPGA stays operative, while only the image processing module is reconfigured. Therefore, FPGAs with dynamic and partial reconfiguration feature are extremely flexible.

Another benefit is that only the currently needed functionality has to be implemented at a given point in time. This way a smaller FPGA can be used and the overall power consumption can be decreased. Their disadvantage is the programmability. While it is already difficult to program multiprocessor systems, the programming of an FPGA is even more difficult, because to achieve maximum performance hardware description languages, such as VHDL or Verilog, have to be used. There exist some Electronic System Level (ESL) design tools that ease the programmability by offering special C-to-Gates or Matlab-to-VHDL tool flows.

3.2.1 Runtime Adaptive MPSoC

An example of hardware reconfiguration in MPSoCs was proposed by D. Göhringer, M. Hübner, V. Schatz and J. Becker in 2008 [17] where they described an MPSoC architecture based on reconfigurable processors. Runtime Adaptive Multiprocessor on Chip (RAMPSoC) is an MPSoC that can be adapted on demand during run-time by exploiting dynamically and partially reconfigurable hardware. This presents a new degree of freedom, as now not only the software, like in traditional MPSoCs, but the hardware can be adapted during run-time. Therefore a new design flow is used for RAMPSoC that is illustrated in Figure 15. It combines the top-down approach used in traditional MPSoCs with a new bottom-up approach. This bottom-up approach is used during design-time to generate an initial RAMPSoC version optimized for the requirements of the application.

The design flow of a RAMPSoC however, uses the above mentioned bottom-up approach not only during the design-time but also during run-time to adapt the hardware architecture on-

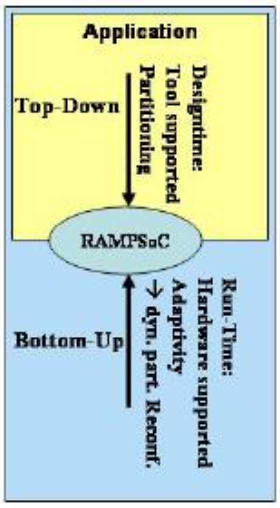

Application
Top-Down
Designtime: Tool supported Partitioning
RAMPSoC
Bottom-Up
Run-Time: Hardware supported Adaptivity → dym. part. Reconf.

Figure 15 Design flow approach for RAMPSoC [17]

demand. This unique on-demand run-time adaptation of the hardware architecture gives the designer a new degree of freedom to ensure an optimized distribution of computing tasks on the adapted processing cells and to fulfill constraints such as on performance, power, and area consumption.

The hardware-adaptation during run-time is done by using the dynamic partial reconfiguration. With this feature, RAMPSoCs support the adaptation of processor cores, for example by modifying their architecture (e.g., RISC, CISC, VLIW, Superscalar), their bitwidth or their instruction sets. Also, the communication infrastructure of a RAMPSoC can be adapted using dynamic and partial reconfiguration. This is necessary as a RAMPSoC represents a heterogeneous MPSoC, whose processors can be exchanged during run-time, so a change in the interface must be accompanied by an adaptation of the communication structure. Such an adaptive communication structure could for example be a Network-on-Chip. This makes a RAMPSoC to a very flexible MPSoC which can be used efficiently and optimally for a broad spectrum of different applications domains.

3.2.2 Benefits of the RAMPSoC Approach

Compared to pure hardware solutions of functionality or algorithm, RAMPSoC is much more flexible by using Hardware-Software co-design. Due to this, software adaptation and the usage of existing processor cores the design flow is more time-efficient. In addition a software design being less time consuming than a hardware design allows adapting the system more easily to new tasks. Another advantage is the fact that its processing elements come with a well-defined programming model and tools such as a compiler.

In comparison to a single RISP core, it is faster due to extended parallelism given by the usage of multiple processor cores. Hence it has higher computing power and the possibility to execute several applications in parallel.

In contrast to a traditional MPSoC, where the hardware architecture is fixed and the application is implemented purely in software, a RAMPSoC is much more flexible because its hardware architecture can not only be optimized during the design-time but also during run-time. This on-demand functionality makes RAMPSoC very versatile. Due to this hardware optimization capability during run-time the computing power is adapted on demand to the given requirements and therefore the power consumption is smaller than in traditional MPSoCs. An additional benefit is a better performance as the processor cores and reconfigurable hardware accelerators are optimized for the application. Other advantages are the reduced costs, because of the missing mask costs necessary with ASIC design, and a faster time to market for a broad spectrum of applications.

3.3 MPSoCs Design Challenges

The introduction of multicore processors signals a major shift in the structure and design ways of all computing platforms. Before this shift, almost all embedded software could be written with the assumption that there is only a single processor core and where multiple processors were involved, they were either relatively loosely coupled or were used in easily parallelized applications.

While multicore systems will change this model somewhat, there is a real expectation that the number of cores will grow rapidly, roughly doubling with each processor generation. This growth will create unique challenges for run-time systems and compilers. If multiple cores on a processor share a cache, contention for the shared cache memory and cache coherence are major issues.

Power and temperature management are also two concerns that can increase exponentially with the addition of multiple cores. The other issue is the problem of using a multicore processor to its full potential. Applications should be written properly so that different parts of the program run concurrently. Finally the necessity to move beyond parallel computing paradigm and towards heterogeneous embedded multicore distributed systems will likely drive changes in how embedded software will be created.

3.3.1 Cache Coherence

Allowing multiple processors to share memory complicates the design of the memory hierarchy in a multicore system. Cache coherency, or cache consistency, is a big concern in this multicore environment. Since each core has its own cache, the copy of the data in the cache may not always be the most up to date version. For example, consider a scenario of a processor with two cores where each core brought a block of memory into its private cache. One core writes a value to a specific location. When the second core attempts to read that value from its cache, it will not have the updated copy unless its cache entry is invalidated and a cache miss occurs. This cache miss forces the second core’s cache entry to be updated. This is real trouble for the correctness of the application being executed.

A system is said to be coherent if all copies of the main memory location in multiple caches remain consistent when the contents of that memory location are modified. A cache coherency protocol is a mechanism by which the coherency of the caches is maintained. Maintaining coherency means taking special actions when one core writes to a block of data that exists in other caches.

3.3.2 Power and Temperature

While multicore systems may limit power consumption in some areas, they present real challenges to energy management paradigms optimized for single-chip systems. In particular, multicore limits the scope of the capability of DVFS (dynamic voltage and frequency scaling) because most SoC subsystems share power supplies and clocks. As a result, scaling the operating voltage of one of several SoC subsystems may limit its ability to use local buses to communicate with other subsystems, and access shared memory. Clock frequency scaling of a single SoC subsystem also presents a big challenge, especially for synchronous buses.

To lessen the heat generated by multiple cores on a single chip, the chip is architected so that the number of hot spots does not grow too large and the heat is spread out across the chip. For example, the majority of the heat in the CELL processor is dissipated in the Power Processing Element, and the rest is spread across the Synergistic Processing Elements.

3.3.3 Reliability Issues

Emerging embedded applications running on MPSoCs are getting more and more complex, demanding good architectures to ensure sufficient bandwidth for any transaction between memories and cores as well as communication between different cores on the same chip. The significant heterogeneity in MPSoCs which are likely to mix logic layers with memory layers and even more complex technologies increases the fault's probability in a system. As a result, multicore systems are becoming susceptible to a variety of faults caused by crosstalk, the impact of radiations, oxide breakdown, and so on. A simple failure in a single transistor caused by one of these factors may compromise the entire system reliability where the failure can be illustrated in corrupted message delivery, time requirements unsatisfactory, or even sometimes the entire system collapse.

To ensure their correct functionality and reliability, MPSoCs must be fault-tolerant to any short-term malfunction or permanent physical damage to ensure correct functionality while minimizing the performance degradation as much as possible.

3.4 MPSoC Applications

As with general architectures, MPSoCs are mainly driven by performance requirements of applications. Therefore, knowing the target application(s) of the system before starting the design is important not only for the selection of appropriate PEs but also for reducing the overall cost of the system.

There are four well-known applications for MPSoC systems: (1) wireless, (2) network, (3) multimedia, and (4) mobile applications.

- Wireless Applications: In this class of applications, MPSoCs are mainly used as wireless base stations (i.e., Lucent Daytona [19]) in which identical signal processing is performed on a number of data channels. Daytona is a homogeneous system with four SPARC V8 CPU cores attached to a high-speed split-transaction. Each CPU has an 8-KB 16-bank cache, and each bank can be configured as instruction cache, data cache or scratchpad. The cores share a common address space.
- Network Applications: In this second class, MPSoCs can be used as a network processor for packet processing in off-chip networks. The C-5 processor is an example of network processor [20]. In this system, packets are handled by channel cores that are grouped into four clusters of four units each. The traffic of all cores is handled by three buses. In addition to the channel cores, there are also several specialized cores. The executive processor core is a RISC architecture.
- Multimedia Applications: Multimedia applications implemented on consumer electronics devices span a vast range of functionality, from audio decoder such as MP3 to video decoder such as H.264 up to advanced picture quality processing such as frame rate up-conversion and motion accurate picture processing (MAPP). Hybrid TV solutions are an excellent example because they are virtually capable of executing any of these multimedia applications
- Mobile Applications: The fourth class of MPSoCs application is on the mobile cell phone. Earlier cell phone processors performed baseband operations, including both communication and multimedia operations. As an example, the Texas Instruments' OMAP architecture has several implementations. The OMAP 5912 has two CPU cores: an ARM9 and a TMS320C55x digital signal processor (DSP). The ARM core acts a master and the DSP core acts as a slave that performs signal processing operations.

3.4.1 Applications Mapping

Today's MPSoC architectures are composed of commercially off-the-shelf available Intellectual Property (IP) blocks. Ultimately, engineers would like to design a generic heterogeneous

MPSoC architecture that is flexible enough to run different applications. However, mapping an application to such heterogeneous SoC is more difficult compared to mapping to a homogeneous one. Today, general practice is to map applications to the architecture at design-time or run-time. Run-time mapping offers a number of advantages over design-time mapping. It mainly offers the following possibilities:

- To avoid defective parts of a SoC. Larger chip area means lower yield. The yield can be improved when the mapper is able to avoid faulty parts of the chip. Also, aging can lead to faulty parts that are unforeseeable at design-time.
- To adapt the available resources. Only at runtime the available resources are known to the mapping algorithm. In addition, the available resources may vary over time for example due to applications running simultaneously or adaptation of algorithm to the environment.

The objective of the runtime mapping is to determine at runtime a near-optimal mapping of the application to the architecture using the library of process implementations and the current status of the system.

The mapping of the functional subsystems onto SoC hardware resources may be based on a number of considerations:

- Support: support for industry standards. This is very important for processor cores that are programmed by the designers. Generally, industry standard CPU cores have extensive toolchain and library support that eases the application design and debug.
- Performance: computationally intensive algorithms such as HD H.264 decoder cannot be efficiently implemented on a general-purpose processor because of the computational complexity. Instead, a function-specific HW core is needed.
- Flexibility: evolving standards require flexibility in implementations so that new codecs can be added without the need for a new SoC. This reduces cost and time.
- Re-usability: implementation, integration, and verification are time-consuming tasks, and sometimes it is appropriate no to implement a function on the most optimum SoC HW resource in order to make it reusable in future.

4 Implementation of a Heterogeneous Reconfigurable Multicore System

Performance and power consumption on MPSoCs are two of the most important concerns that designers must assess. Due to their contrary nature performance and power consumption cannot be simultaneously at the ideal level of maximum performance and minimum power consumption. The most common approach is to balance those two based on the application among other metrics like cost and reliability.

In this chapter, we present an adaptive architecture for Zynq MPSoCs that allows the system to self-adjust on real-time the performance and power consumption based on the designer's guidelines. The heterogeneous reconfigurable multicore architecture as depicted on Figure 16Figure 1 contains one physical processor along with one or more softcores of a different instruction set. By being reconfigurable, these softcores have the ability to interchange in such way that either performance is boosted or power consumption decreases. A portion of the FPGA fabric is dedicated to the static part of the design, leaving the rest available for reconfigurable partitions. The module repository is maintained on the external storage of the device and consists of multiple different configurations regarding the number or the Instruction Set of the softcores.

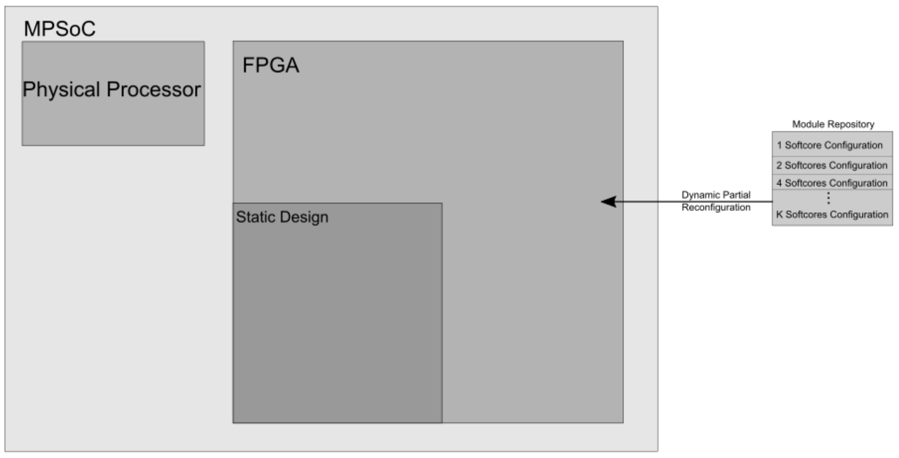

MPSoC
Physical Processor
FPGA
Static Design
Dynamic Partial Reconfiguration
Module Repository
1 Softcore Configuration
2 Softcores Configuration
4 Softcores Configuration
⋮
K Softcores Configuration

Figure 16 The heterogeneous reconfigurable multicore architecture.

As most of the FPGA designs that use partial reconfiguration, hardware development is separated into two major parts: the static and the reconfigurable design. The static design acts as the base of the system and contains the initial configuration of the FPGA. At this stage, the main foundation of the system is realized including individual subsystems and functional tasks. Subsystem functionality and required performances, as well the various interactions between them are then defined. Due to the complexity of the design, some subsystems are further decomposed into lower levels of hierarchy. The reconfigurable partition(s) is being instantiated in the static design as a black box without any logic components and preoccupies an area in the PL fabric. Furthermore, the reconfiguration mechanism needs to be added in order to manage and control the reconfiguration process.

The reconfigurable design contains all the reconfigurable modules that are being called on demand while the FPGA is already online either by software or hardware trigger. Each module is being designed separately and the main concerns are that the module must fit in the area and match the connection of the already declared reconfigurable partition.

Next, the software development follows and includes programming each processor separately. The primary purpose of each processor is to offer either utility or processing power or both. Utility contains actions like the control of the reconfiguring mechanism; internal hardware resets, shared peripheral handling and more.

4.1 General System Architecture

The architecture of the heterogeneous reconfigurable multicore system is based on establishing communication between the physical processor and the reconfigurable softcores through shared memory. It is tailored to provide high-performance connections and access to the system's peripherals among all the processors. By sharing the memory, each softcore is given a specific amount of memory for instruction and data cache. Since the physical processor has direct access to the whole range of the memory it can access and utilize the data that are being processed.

Therefore this architecture offers the option to save power while providing a decent performance output when only the physical processor is in use and if and only if it is needed to load the reconfigurable softcores in order to increase performance and inevitably power consumption. This concept can be best applied to applications where long-term high performance is not essential instead in special cases of high workload where not enough performance is detrimental to the outcome (i.e., strict time deadlines).

4.1.1 Building Blocks

The architecture consists of three main components: the processing system, the HWICAP controller, and the PR decoupler. The processing system acts as a logic connection between the physical processor and the rest of the embedded system in the PL fabric. Most importantly, in this case, it allows direct access from the softcores to the memory with the help of interfaces. The interfaces between the PS and the PL mainly consist of three main groups: the extended multiplexed I/O (EMIO), the programmable logic I/O, and the AXI I/O groups. The HWICAP controller enables the physical processor to read and write the FPGA configuration memory through the Internal Configuration Access Port with the purpose to modify the circuit structure and functionality during the operation of the device. The PR decoupler is an assisting block that provides a safe and managed boundary between the static logic and the reconfigurable partition during partial reconfiguration. Especially in cases like this that reconfigurable partitions have direct access to the memory, it is essential to assure the data integrity.

4.1.2 Interfacing the Softcores to the Processing System

Each softcore is connected to the processing system using three interfaces: a data cache (DC), an instruction cache (IC) and a peripheral data (PD). Since multiple softcores must connect to the physical processor a block called Interconnect is used to allow one or more master interfaces to be connected with one or more slave interfaces. The whole idea can be seen in Figure 17; two interconnects are used to separate peripheral data from memory due to different destinations on the processing system. The same pattern can be repeated several times, but it is limited to the available PL resources and the number of interfaces on the processing system.

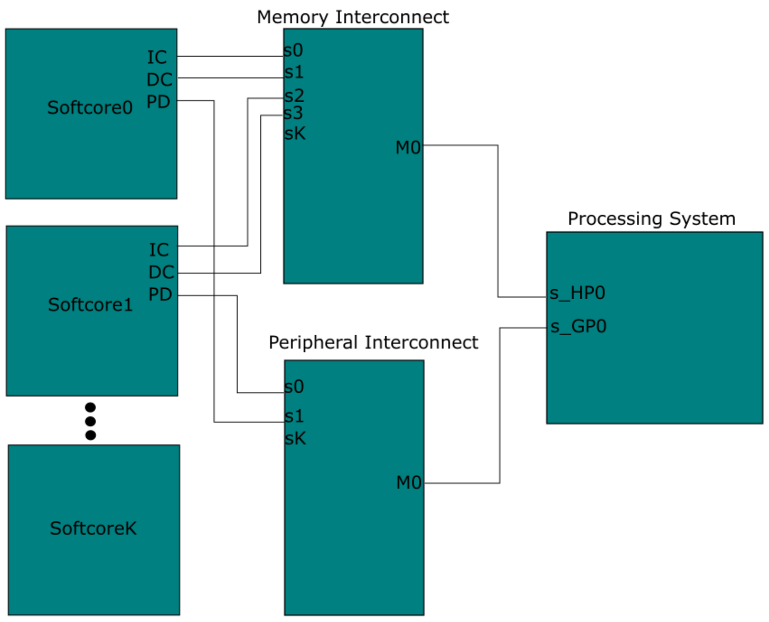


Figure 17 Softcore interfacing

4.2 Hardware Development

In order to implement and evaluate this architecture, hardware was developed using Xilinx's Vivado. For this specific experiment, the processor setup includes one ARM Cortex A9 as the physical processor and two Microblaze v10.0 as the reconfigurable softcores. The hardware platform that was used is Digilent's Zybo development board (Figure 18).The Zybo (diminutive of Zynq Board) is an ultra-low-cost development board featuring the smallest Zynq family device from Xilinx, the Z-7010, which is based on the Artix-7 PL fabric. By being a member of the Zynq architecture (described in chapter 2.5), the Z-7010 integrates a dual-core ARM Cortex-A9 processor with Xilinx 7-series FPGA logic. Some of the board's specifications include:

- 4.400 logic slices
- 240 KB Block RAM
- 80 DSP Slices
- 512 MB DDR3 Memory
- 450 MHz Internal clock

Also connectivity and onboard I/O such as:

- 6 PMod ports
- Microphone and line in jacks
- Gigabit Ethernet
- MicroSD card slot
- HDMI and VGA port
- Switches, Buttons and LEDs

- JTAG/UART

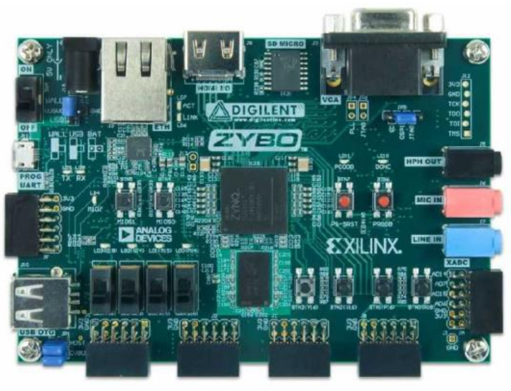


Figure 18 The Zybo Development Board [24]

4.2.1 Design Entry

Starting the static hardware development with the block design on Vivado IPI canvas the first piece is the processing system. The processing system is configured to enable access to I/Os that cross the boundary to the PL. Thus three AXI interfaces are activated, a Master AXI GP, a Slave AXI GP and a Slave AXI HP (High Performance), and since all of them connect with multiple blocks each interface is paired with one interconnect. The next block is the HWICAP which provides the necessary interface to access the board's ICAP and connects to the PS allowing the ARM processor to trigger the partial reconfiguration process. For the reason that this design contains only a single reconfigurable partition, one blackbox is placed on the canvas to establish the connections that the reconfigurable module will be using. The outputs of the blackbox are connected to a PR decoupler with the intention to protect the memory during the reconfiguration process. Once the reconfiguration process is completed, the ARM processor via the AXI GPIO controller disables the PR decoupler and allows the softcores to access the rest of the system. Finally, necessary clocks and resets are added and connected either manually or automatically with the help of Vivado. Concluding the block design addresses and ranges are assigned to each AXI interface. Figure 19 depicts the validated block design of the static part of the experiment.

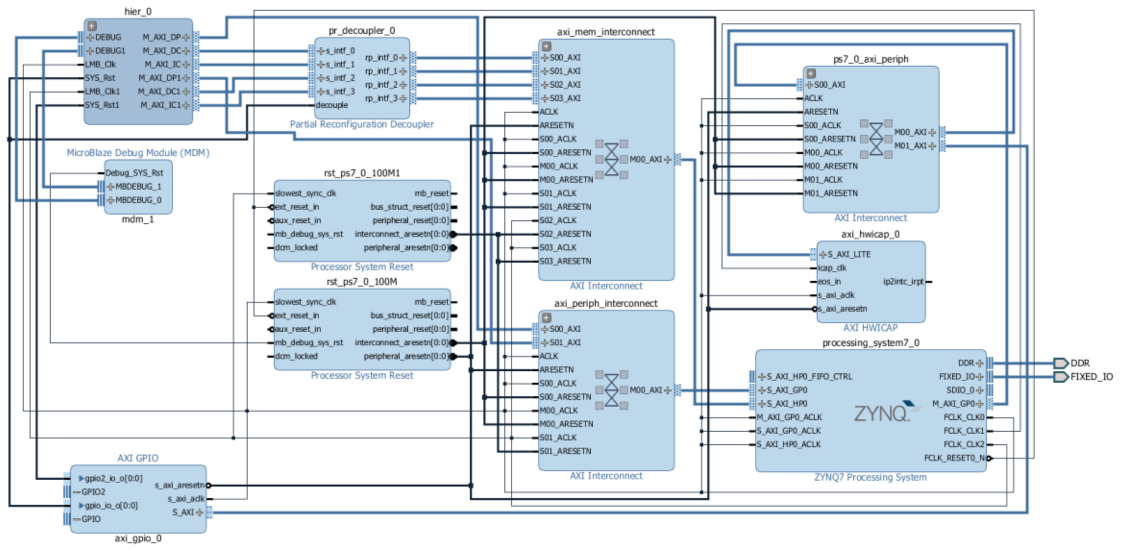


Figure 19 Static hardware block design

In the same manner, the block design of the reconfigurable module includes two Microblaze softcores optimized for performance with an operating frequency of 240MHz. As seen in Figure 20 the module has as inputs a debug probe, a clock, and a reset interface and as outputs an instruction cache, a data cache, and a peripheral data interface for each softcore. To increase the performance of the softcores optional instruction settings are enabled such as the extended floating point unit, the 64-bit integer multiplier and the integer divider. Each softcore claims 8kB for instruction cache and 8kB for data cache from the system's memory. To avoid memory conflicts and overlaps a unique base vector address is assigned per softcore.

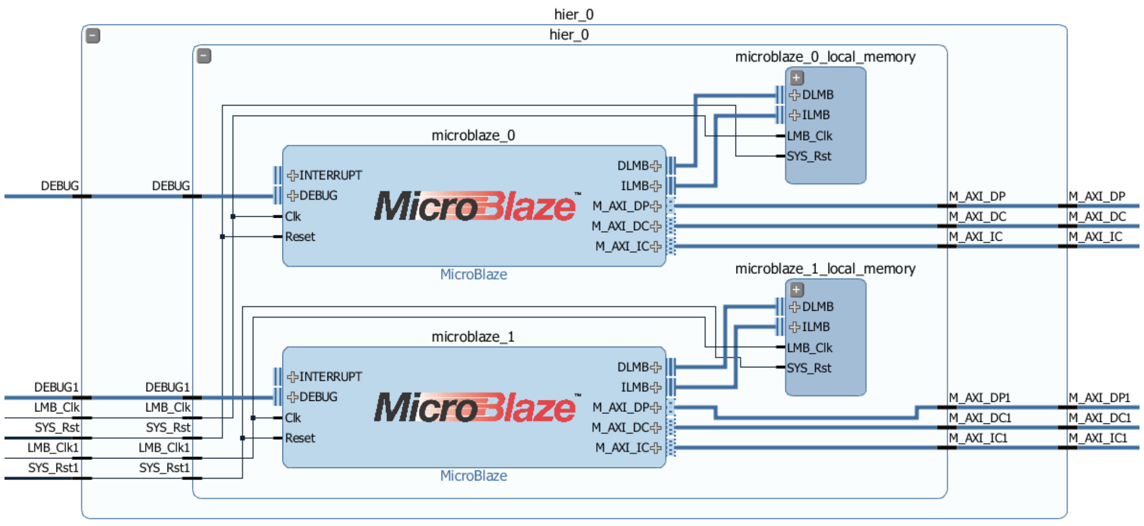


Figure 20 Reconfigurable module block design

After the static and the reconfigurable block design are completed an HDL wrapper is created that contains the system's hierarchy. The system wrapper is a Verilog constraint file that consists of different hierarchical levels with the top level describing the correlation between the inputs and outputs of the block design to the physical pins and resources of the board. Since Vivado is board-aware and this project is targeted for a specific board, the whole process is fully automated. Below the top level, the wrapper describes the connections between all the blocks in the design with occasional blocks dividing into sub-blocks. The blocks themselves are presented as black boxes without any logic circuit containing only the block's name and the port names.

4.2.2 Hardware Synthesis and Implementation

The block design is synthesized in out-of-context mode in Vivado creating individual design checkpoint (dcp) files for each block and its sub-hierarchy. Every dcp contains a netlist that describes the block's I/Os (type, size), parameters and the translated functionality assigned in resources (LUTs, flip-flops, BRAMs, etc.). At this point, all the necessary files from the block design are generated and for the rest of the workflow Vivado is being used in non-project mode by taking advantage of the Tcl console. In order to reach the final hardware that will be loaded on the FPGA, multiple design checkpoints are created containing specific states of the design. Starting with the linked synthesized design, the system wrapper is used as the backbone, and all the block dcps are loaded. Furthermore, the block that represents the reconfigurable partition is defined as partially reconfigurable by setting the property HD.RECONFIGURABLE to 1. The assembled design state is saved in a checkpoint as "top_linked" for later use.

The next step of the workflow is to floorplan the reconfigurable partition region on the PL fabric. Based on the type and amount of resources used by the reconfigurable module, the reconfigurable partition region must be appropriately defined so it can accommodate the reconfigurable module. Depicted in Figure 21 is the PL fabric after the floorplan where the area (p_block) highlighted in the purple rectangle is assigned to the reconfigurable partition (hier_0).

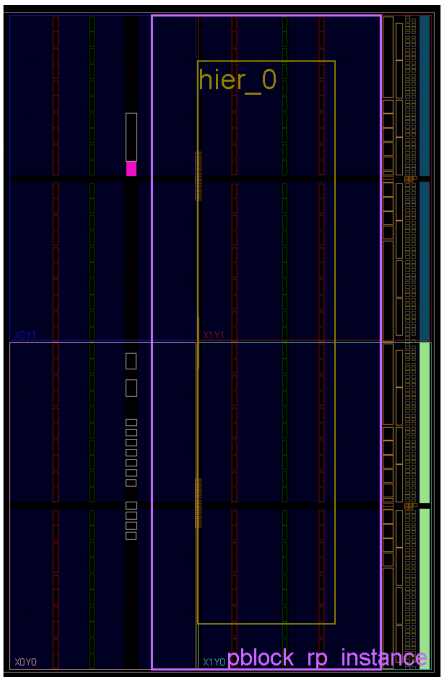
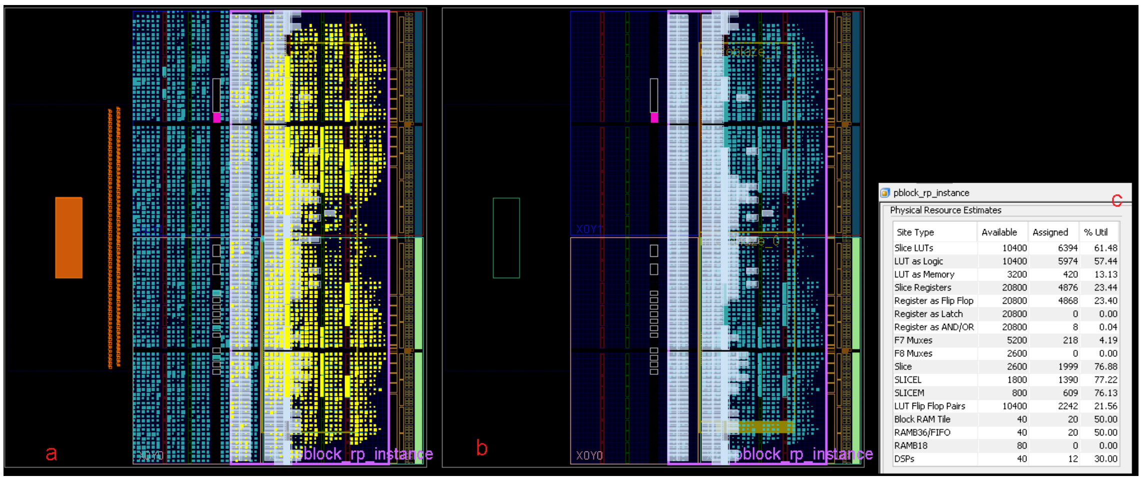


Figure 21 The floorplan Zynq-7000 PL fabric

After the floorplan the design is implemented manually through the Tcl console with three commands, first it is optimized (opt_design) then it is placed (place_design), and finally, it is routed (route_design). From the results of the implementation, we can evaluate the resource usage of the reconfigurable module on the pblock (Figure 21c) and observe the system placed on the PL fabric (Figure 21a), where cyan illustrates the static part and yellow the reconfigurable module of the design. Lastly, from this state of the design, we extract two design checkpoints, one with the full design, and one with the reconfigurable module configuration Figure 21b.



pblock_rp_instance

Physical Resource Estimates

Site Type	Available	Assigned	% Util
Slice LUTs	10400	6394	61.48
LUT as Logic	10400	5974	57.44
LUT as Memory	3200	420	13.13
Slice Registers	20800	4876	23.44
Register as Flip Flop	20800	4868	23.40
Register as Latch	20800	0	0.00
Register as AND/OR	20800	8	0.04
F7 Muxes	5200	218	4.19
F8 Muxes	2600	0	0.00
Slice	2600	1999	76.88
SLICEL	1800	1390	77.22
SLICEM	800	609	76.13
LUT Flip Flop Pairs	10400	2242	21.56
Block RAM Tile	40	20	50.00
RAMB36/FIFO	40	20	50.00
RAMB18	80	0	0.00
DSPs	40	12	30.00

Figure 22 a) Full design b) Isolated reconfigurable module c) Physical resources

In order to create the static configuration, the reconfigurable module is removed and updated as a blackbox. To preserve the routing, the design is locked at the routing level and the state of the design is saved. Finally, the last checkpoint is the blanking configuration which is used to erase all the reconfigurable logic and routing while the static logic and routes in that region continue to operate. More specifically the blanking configuration comes handy in scenarios that during runtime the reconfigurable module is loaded on the FPGA and after it completes its purpose we want to remove it without affecting the static hardware. Figure 23 summarizes all the design states, checkpoints and configurations that were created.

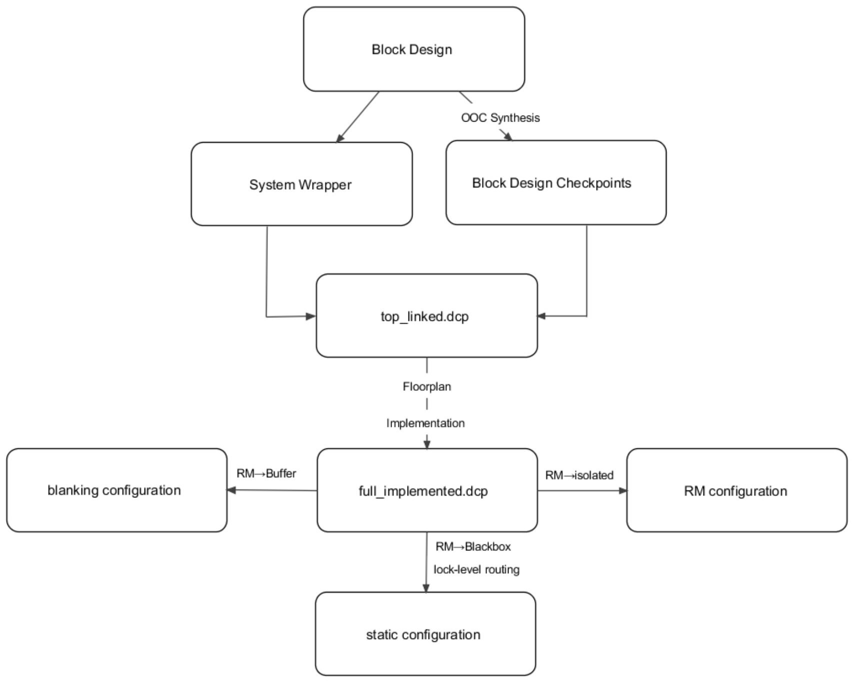


Figure 23 Hardware implementation block diagram

The hardware development concludes with the generation of full and partial bitstreams from the three configurations that were created.

4.3 Software Development

Even though the proposed architecture of this Thesis is focused on hardware, implementing the system on a board does not offer enough metrics in order to evaluate and compare its results. For that reason, software development for the physical processor and the varying number of softcores is a major aspect of the overall system development. Those applications were developed so as to fully utilize the hardware testbed as well as to provide tangible functionality similar to these that realistic systems have. Therefore, the applications perform two separate tasks: the first task includes hardware handling and memory management and the second task includes the unique software implementation of each designer that uses this system architecture.

The applications were developed on the Xilinx Software Development Kit (SDK) and the flow starts by exporting the hardware design file of the static design from Vivado to the SDK. The hardware design file contains important data such as the memory map information of the processors that are crucial in setting up the SDK environment.

4.3.1 Hardware Handling and Memory Management.

The physical processor is responsible for the partial reconfiguration process along with the control of the hardware that assists the partial reconfiguration. More specifically it is in charge of the AXI GPIO controller that manages the PR Decoupler and the softcore resets. The partial reconfiguration process starts by declaring an address for each reconfigurable module as well as its size. Next, all the required functions are being implemented in order to read and transfer the configuration file of each reconfigurable module from the external storage (in this case the SD card) to the memory. Lastly, the function that transfers and writes the configuration from the memory to the HWICAP is implemented and consequently enables the physical processor to trigger and perform the partial reconfiguration process whenever this function is called.

On the subject of memory management, the physical processor declares a communication area in the form of an array of structs with a size proportionate to the number of the available softcores. Each struct contains three fields, a data array, an owner flag, and a progress flag. In the same manner, each softcore declares an identical communication struct in its own scope. By using pointers, the physical processor can access the communication area of each softcore. Overall the memory scope of all the processors is illustrated in Figure 24.

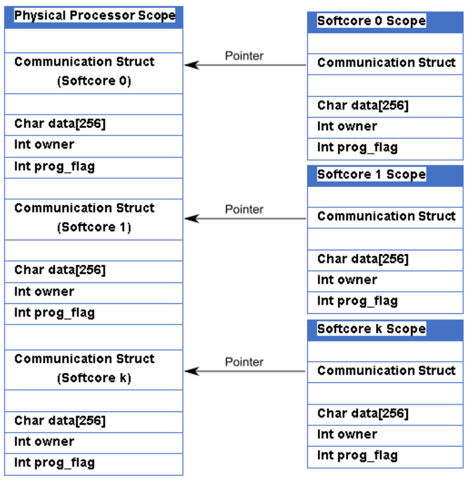


Figure 24 The memory scope of all the processors.

During runtime the physical processor controls the communication flow using the following algorithm:

1. Set the owner of all communication areas to 0, which means that only the physical processor can write to them
2. Clear the data variable of the first softcore
3. Allow the first softcore to access the communication area (Com_struct[0]->Owner = 1;)
4. Wait until the softcore gives up ownership of the communication area
5. Read the data and/or the progress flag of the softcore
6. Repeat 2-5 for the rest of the softcores

Similarly, each softcore uses the following algorithm:

1. Wait until we own the communication area
2. Write data and/or update progress flag
3. Release ownership of the communication area

5 Experimental results

In this chapter, we present the results that were obtained through multiple experiments based on the heterogeneous reconfigurable multicore architecture implemented on the Zybo board. The experiments are targeting the processing power of the proposed architecture measured in execution time compared to the standalone physical processor.

Since the purpose of this architecture is to provide a generic performance improvement solution, multiple applications were needed to justify its potential success and its limits. For that reason benchmarks that simulate real-world applications were used in order to get concrete results. The benchmarks were carefully selected to create different processing workloads that can be characterized as integer-intensive, memory-intensive and control-intensive as well as to tackle commercial categories including Telecommunications, Network Security and so forth.

5.1 Testing

Using the same hardware as a reference point multiple applications were developed in Xilinx's SDK. In contemplation of result integrity, a template application was created for each processor, containing the utility functions such as the partial reconfiguration and the communication between the processors leaving blank space to be filled by the different benchmarks.

5.1.1 Benchmarks

The four benchmark test suites that were selected are:

1) MiBench [25,26]
2) WCET Benchmarks [27,28]
3) DSPStone [29,30]
4) CHStone [31,32]

Out of those test suites seven benchmarks were used to extract results:

1) Fast Fourier Transform (FFT)
2) String Search algorithm
3) AES encryption-decryption
4) Fast Discrete Cosine Transform (FDCT)
5) Microprocessor without Interlocked Pipelined Stages (MIPS)
6) Blowfish (cipher) algorithm
7) Finite Impulse Response (FIR)

5.2 Results

The main ambition of the proposed architecture is to prove that by taking advantage of the strengths of partial reconfiguration that it is possible to increase performance while maintaining or even reducing power consumption. The following results come from two different hardware setups.

- Setup 1. ARM Cortex 9 Operating at 600MHz combined with 2 Microblaze softcores operating at 240MHz.
- Setup 2. Standalone ARM Cortex 9 Operating at 600MHz

It must be noted that on both setups the workload is identical for every scenario. Additionally, the compiler optimization is also identical on both setups at max level (-O3).

Scenario 1				Workload: FFT, String Search	
Setup 1		Setup 2		Total Execution Time	
Processor	Benchmark	Processor	Benchmark	Setup 1	Setup 2
ARM	FFT	ARM	FFT	61.92s	76.2s
Mb0	sSearch	ARM	sSearch	Performance Improvement	
Mb1	sSearch	ARM	sSearch	18.74%	

Table 1 Results from scenario 1

Scenario 2				Workload: AES	
Setup 1		Setup 2		Total Execution Time	
Processor	Benchmark	Processor	Benchmark	Setup 1	Setup 2
ARM	AES	ARM	AES	29.86s	41.69s
Mb0	AES	ARM	AES	Performance Improvement	
Mb1	AES	ARM	AES	28.37%	

Table 2 Results from scenario 2

Scenario 3				Workload: FDCT	
Setup 1		Setup 2		Total Execution Time	
Processor	Benchmark	Processor	Benchmark	Setup 1	Setup 2
ARM	FDCT	ARM	FDCT	35.55s	57.66s
Mb0	FDCT	ARM	FDCT	Performance Improvement	
Mb1	FDCT	ARM	FDCT	38.34%	

Table 3 Results from scenario 3

Scenario 4				Workload: MIPS	
Setup 1		Setup 2		Total Execution Time	
Processor	Benchmark	Processor	Benchmark	Setup 1	Setup 2
ARM	MIPS	ARM	MIPS	30.18s	46.98s
Mb0	MIPS	ARM	MIPS	Performance Improvement	
Mb1	MIPS	ARM	MIPS	35.75%	

Table 4 Results from scenario 4

Scenario 5				Workload: Blowfish	
Setup 1		Setup 2		Total Execution Time	
Processor	Benchmark	Processor	Benchmark	Setup 1	Setup 2
ARM	Blowfish	ARM	Blowfish	30.1s	38.5s
Mb0	Blowfish	ARM	Blowfish	Performance Improvement	
Mb1	Blowfish	ARM	Blowfish	21.28%	

Table 5 Results from scenario 5

Scenario 6				Workload: FIR	
Setup 1		Setup 2		Total Execution Time	
Processor	Benchmark	Processor	Benchmark	Setup 1	Setup 2
ARM	FIR	ARM	FIR	30.69s	39.19s
Mb0	FIR	ARM	FIR	Performance Improvement	
Mb1	FIR	ARM	FIR	23.22%	

Table 6 Results from scenario 6

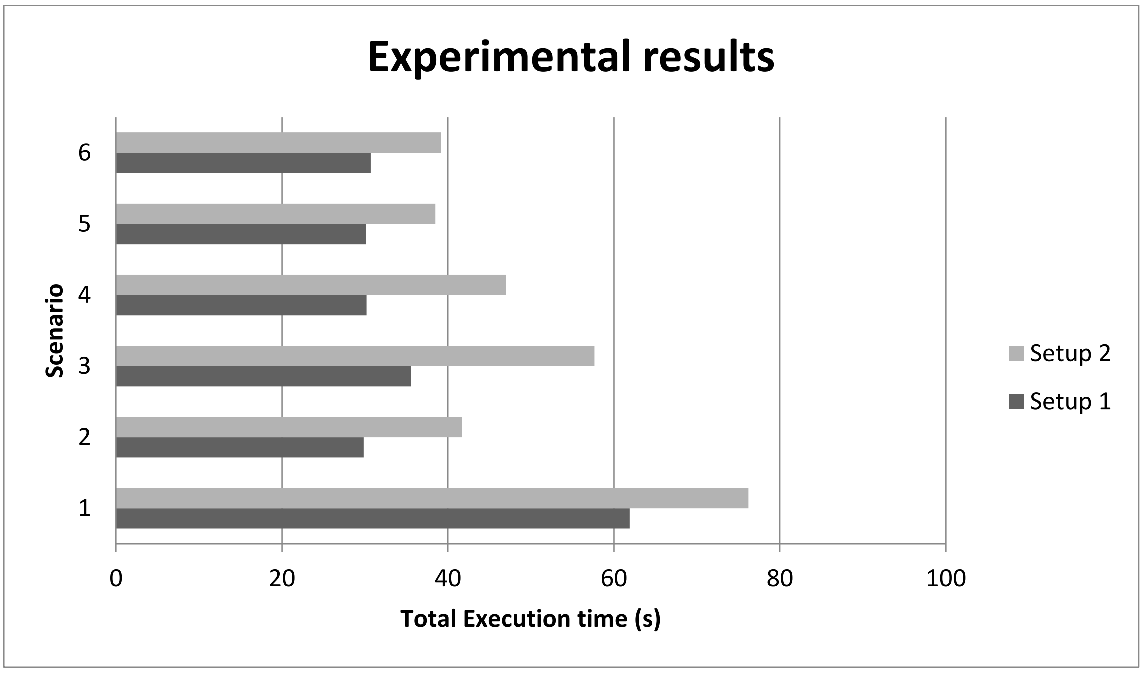
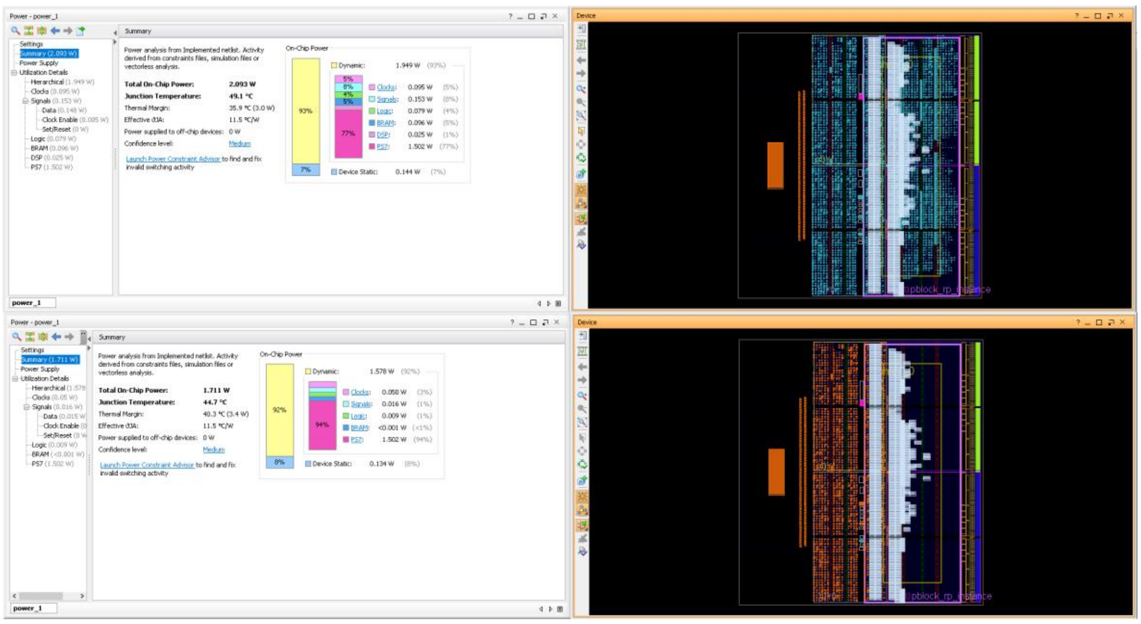


Figure 25 Experimental results

Figure 25 encapsulates a complete overview from all the six experiments and Figure 26 presents the power profiles from the two hardware setups at the top part of the figure Setup 1 with total on-chip power 2.093 W and on the bottom Setup 2 with total on-chip power 1.711 W.



Power - power_1

Summary

- Settings
- Summary (2.093 W)
- Power Supply
- Utilization Details
  - Hierarchical (1.949 W)
  - Clocks (0.095 W)
  - Signals (0.153 W)
    - Data (0.148 W)
    - Clock Enable (0.005 W)
    - Set/Reset (0 W)
  - Logic (0.079 W)
  - BRAM (0.096 W)
  - DSP (0.025 W)
  - PS7 (1.502 W)

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**2.093 W**
Junction Temperature:	**49.1 °C**
Thermal Margin:	35.9 °C (3.0 W)
Effective ϑJA:	11.5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	1.949 W	(93%)
Clocks:	0.095 W	(5%)
Signals:	0.153 W	(8%)
Logic:	0.079 W	(4%)
BRAM:	0.096 W	(5%)
DSP:	0.025 W	(1%)
PS7:	1.502 W	(77%)
Device Static:	0.144 W	(7%)

power_1

Power - power_1

Summary

- Settings
- Summary (1.711 W)
- Power Supply
- Utilization Details
  - Hierarchical (1.578
  - Clocks (0.05 W)
  - Signals (0.016 W)
    - Data (0.015 W
    - Clock Enable (0
    - Set/Reset (0 W
  - Logic (0.009 W)
  - BRAM (<0.001 W)
  - PS7 (1.502 W)

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**1.711 W**
Junction Temperature:	**44.7 °C**
Thermal Margin:	40.3 °C (3.4 W)
Effective ϑJA:	11.5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	1.578 W	(92%)
Clocks:	0.050 W	(3%)
Signals:	0.016 W	(1%)
Logic:	0.009 W	(1%)
BRAM:	<0.001 W	(<1%)
PS7:	1.502 W	(94%)
Device Static:	0.134 W	(8%)

power_1

Figure 26 The power consumption of the 2 setups

6 Conclusion and future work

This thesis has proposed an architecture that enables the high-level design of adaptive systems using FPGA partial reconfiguration. It has shown that FPGAs are suitable hardware platforms for high-performance adaptive system implementations and that PR offers great advantages for such systems. The suitability of hybrid FPGA platforms, which integrate physical processors with reconfigurable softcores on the same physical chip, has also been demonstrated and tested with commercial applications.

This chapter draws the conclusions from this thesis, evaluates the experimental results, and outlines areas for further research.

Looking at the results, one can notice that the performance improvement when using two partially reconfigurable softcores ranges from 18-38% on applications that require more than 30 seconds of processing. The variance of the performance improvement is tied to the relationship between the softcore architecture and the type of the application. For example, softcores that are designed for DSP applications will offer better improvement than a general purpose processor on applications that involve DSP processing. On the other hand, softcores that are not architecturally designed for a specific field like for example floating point calculations will offer slight or no improvement with applications that involve floating point processing.

Two major paths derive for further research of this project, a hardware-oriented approach and a software-oriented approach. The hardware approach includes more potential softcores on MPSoCs with bigger FPGA fabrics as well as a bigger variation on the architectures of the softcores. The software-oriented approach includes a scheduler that controls the workload of the system and adapts the processing power by triggering the partial reconfiguration process for an ideal amount of softcores.

Bibliography and references

[1] Wikipedia URL: https://en.wikipedia.org/wiki/Programmable_Array_Logic

[2] Roger Woods, John McAllister, Gaye Lightbody, Ying Yi (2017) FPGA-based implementation of Signal Processing Systems (2nd ed.). Wiley.

[3] Xilinx Inc. UltraScale Architecture Configurable Logic Block. (2017)

URL: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf

[4] Sanjay Churiwala (2017) Designing with Xilinx® FPGAs: Using Vivado. Springer.

[5] Dirk Koch Partial (2017) Reconfiguration on FPGAs Architectures, Tools and Applications. Springer.

[6] Xilinx Inc. Virtex- II Platform FPGAs: Complete data sheet. (2014)

URL: https://www.xilinx.com/support/documentation/data_sheets/ds031.pdf

[7] Xilinx Inc. 7 Series FPGAs Data Sheet: Overview. (2017)

URL: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[8] Altera Corporation Stratix Device Handbook, Volume 1 (2006)

URL: https://www.altera.com/content/dam/stratix_vol_1.pdf

[9] Altera Corporation Stratix V Device Overview (2015)

URL: https://www.altera.com/stratix-v/stx5_51001.pdf

[10] Xilinx Inc. Microblaze Processor Reference Guide (2013)

URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/mb_ref_guide.pdf

[11] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, Robert W. Stefwart (2014) The Zynq Book Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc. Strathclyde Academic Media.

[12] Xilinx Inc. Zynq-7000 All Programmable SoC Data Sheet (2017)

URL: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[13] Xilinx Inc. Vivado Design Suite AXI Reference Guide (2017)

URL: http://www.xilinx.com/support/documentation/ug1037-vivado-axi-reference-guide.pdf

[14] Michael Hubner, Jurgen Becker (2011) Multiprocessor System-on-Chip Hardware Design and Tool Integration. Springer.

[15] M. Flynn. Some Computer Organizations and Their Effectiveness, IEEE Trans. Computer, vol. 21, pp. 948, 1972

[16] Abderazek Ben Abdallah (2017) Advanced multicore Systems-On-Chip. Springer.

[17] D. Göhringer, M. Hübner, V. Schatz, J. Becker: "Runtime Adaptive Multi-Processor System-on-Chip: RAMPSoC"; In Proc. of IPDPS 2008, April 2008.

[18] D. Gohringer, J. Becker, High Performance Reconfigurable Multi-Processor-Based Computing on FPGAs; In Proc. of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), Atlanta, USA, April, 2010

[19] J. Knobloch, E. Micca, M. Moturi, C.J. Nicol, J.H. O'Neill, J. Othmer, E. Sackinger, K.J. Singh, J. Sweet, C.J. Terman, J.Williams,Asingle-chip, 1.6-billion, 16-bMAC/s multiprocessorDSP. IEEE J. Solid-State Circuits 35(3), 412–424 (2000)

[20] C-5 Network Processor Architecture Guide, C-Port Corp., North Andover, MA, 31 May 2001

[21] Wikipedia URL: https://en.wikipedia.org/wiki/VHSIC

[22] Valery Skylarov, Iouliia Skliarova, Alexander Barkalov,Larysa Titarenko (2014) Synthesis and optimization of FPGA-based systems. Springer

[23] Xilinx Inc. Partial Reconfiguration User Guide (2013)

URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug702.pdf

[24] Digilent URL: https://reference.digilentinc.com/reference/programmable-logic/zybo/start

[25] MiBench Source Files https://github.com/embecosm/mibench

[26] MiBench: A free, commercially representative embedded benchmark suite URL:https://ieeexplore.ieee.org/document/990739/

[27] WCET Source Files: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

[28] The Mälardalen WCET Benchmarks: Past, Present and Future. Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper URL: http://drops.dagstuhl.de/opus/volltexte/2010/2833/pdf/15.pdf

[29] DSPStone Source Files: https://github.com/etherzhhb/Shang/tree/master/testsuite/benchmark/DSPStone

[30] DSPSTONE: A DSP-Oriented Benchmarking methodology. URL: https://www.ice.rwth-aachen.de/fileadmin/publications/Zivojnovic94icspat.pdf

[31] ChStone Source Files: https://github.com/etherzhhb/Shang/tree/master/testsuite/benchmark/ChStone

[32] CHStone: A benchmark program suite for practical C-based high-level synthesis URL: https://ieeexplore.ieee.org/document/4541637/

Appendix A. Template applications source code

A1. ARM Processor

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "xparameters.h"
#include "xil_printf.h"
#include "xil_cache.h"
#include "ff.h"
#include "xdevcfg.h"
#include "xhwicap.h"
#include "xil_io.h"
#include "xil_types.h"
#include "xtime_l.h"
#include <xil_mmu.h>
#include <math.h>
#include <stddef.h>
#include <string.h>
#include <limits.h>
#include "xgpio.h"
#include "sleep.h"


//-----PARTIAL RECONFIGURATION DECLARATION AREA-----
#define PROCESSORS (2)

#define tStart (*(volatile XTime *)0x00200000)
#define tEnd (*(volatile XTime *)0x00200008)


/* The following remap function is based on the example of XAPP1093. */
// Remap all 4 64KB OCM to top of memory starting at 0xFFFC0000
// Open address filtering to include DDR at 0x00000000
#define REMAP()   __asm__ __volatile__(\
"mov  r5, #0x03                                       \n"\
"mov  r6, #0                                          \n"\
"LDR  r7, =0xF8000000  /* SLCR base address    */         \n"\
"LDR  r8, =0xF8F00000  /* MPCORE base address  */         \n"\
"LDR  r9, =0x0000767B  /* SLCR lock key        */         \n"\
"mov  r10,#0x1F                                       \n"\
"LDR  r11,=0x0000DF0D  /* SLCR unlock key                 \n"\
"dsb                                                  \n"\
"isb                   /* make sure it completes */       \n"\
"pli  do_remap     /* preload the instruction cache */     \n"\
"pli  do_remap+32                                          \n"\
"pli  do_remap+64                                          \n"\
"pli  do_remap+96                                          \n"\
"pli  do_remap+128                                         \n"\
"pli  do_remap+160                                         \n"\
"pli  do_remap+192                                         \n"\
"isb                   /* make sure it completes */        \n"\
"b    do_remap                                             \n"\
".align 5, 0xFF         /* forces the next block to a cache line
alignment */ \n"\
"do_remap:              /* Unlock SLCR                          */ \n"\
"str  r11, [r7, #0x8]   /* Configuring OCM remap value          */ \n"\
"str  r10, [r7, #0x910] /* Lock SLCR                            */ \n"\
"str  r9,  [r7, #0x4]   /* Disable SCU & address filtering      */ \n"\
"str  r6,  [r8, #0x0]   /* Set filter start addr to 0x00000000 */ \n"\
"str  r6,  [r8, #0x40]  /* Enable SCU & address filtering       */ \n"\
"str  r5,  [r8, #0x0]                                              \n"\
"dmb                                                               \n"\
);

/* The format of a communication area.  This area will be shared between
the ARM and the MicroBlazes. */
struct com_area
{
    char Data[256];
    int Owner;
    int fin_flag;
};

volatile struct com_area * Com_area[PROCESSORS];

// Partial Reconfiguration functions
#define PARTIAL_ADDER_ADDR   0x200000
#define PARTIAL_ADDER_BITFILE_LEN  0x52FE5 // in number of words
// Turn on/off Debug messages
#ifdef DEBUG_PRINT
#define  debug_printf  xil_printf
#else
#define  debug_printf(msg, args...) do {  } while (0)
#endif

// Read function for STDIN
extern char inbyte(void);
static FATFS fatfs;
```

```
// Driver Instantiations
static XDcfg_Config *XDcfg_0;
XDcfg DcfgInstance;
XDcfg *DcfgInstPtr;
static XHwIcap HwIcap;  // The instance of the HWICAP device
XHwIcap *HwIcapInstPtr;

int SD_Init()
{
    FRESULT rc;

    rc = f_mount(&fatfs, "", 0);
    if (rc) {
        xil_printf(" ERROR : f_mount returned %d\r\n", rc);
        return XST_FAILURE;
    }

    return XST_SUCCESS;
}

int SD_TransferPartial(char *FileName, u32 DestinationAddress, u32
ByteLength)
{
    FIL fil;
    FRESULT rc;
    UINT br;

    rc = f_open(&fil, FileName, FA_READ);
    if (rc) {
        xil_printf(" ERROR : f_open returned %d\r\n", rc);
        return XST_FAILURE;
    }

    rc = f_lseek(&fil, 0);
    if (rc) {
        xil_printf(" ERROR : f_lseek returned %d\r\n", rc);
        return XST_FAILURE;
    }

    rc = f_read(&fil, (void*) DestinationAddress, ByteLength, &br);
    if (rc) {
        xil_printf(" ERROR : f_read returned %d\r\n", rc);
        return XST_FAILURE;
    }
    rc = f_close(&fil);
    if (rc) {
        xil_printf(" ERROR : f_close returned %d\r\n", rc);
        return XST_FAILURE;
    }

    return XST_SUCCESS;
}

int XDcfg_TransferBitfile(XHwIcap *HwIcapInstPtr, u32 *PartialAddress,
u32 bitfile_length_words)
{
    u32 Status = 0;

    Status = XHwIcap_DeviceWrite(HwIcapInstPtr, PartialAddress,
bitfile_length_words);
    if (Status != XST_SUCCESS)
    {
/* Error writing to ICAP */
        xil_printf("error writing to ICAP (%d)\r\n", Status);
        return -1;
    }
    while(XHwIcap_IsDeviceBusy(HwIcapInstPtr));
    return XST_SUCCESS;
}

//Benchmark Declarations



//End of Benchmark

int main()
{

    u32 PartialAddress;
    int Status;
    int arm_dbg = 0;
    for(int arm_k=0; arm_k<400000000; arm_k++){
        arm_dbg +=1;
        arm_dbg -=1;
    }

    Xil_Out32(XPAR_AXI_GPIO_1_BASEADDR,0x1);
    printf("Decouple is Active\r\n");
    int  Proc, Addr;

/* Test ARM output. */
    printf("ARM ready!\n");

    XHwIcap_Config *ConfigPtr;

// Flush and disable Data Cache
    Xil_DCacheDisable();

// Initialize SD controller and transfer partials to DDR
    SD_Init();

    SD_TransferPartial("add.bin", PARTIAL_ADDER_ADDR,
(PARTIAL_ADDER_BITFILE_LEN << 2));

// Invalidate and enable Data Cache
    Xil_DCacheEnable();

// Initialize Device Configuration Interface
    DcfgInstPtr = &DcfgInstance;
    XDcfg_0 = XDcfg_LookupConfig(XPAR_XDCFG_0_DEVICE_ID) ;
    Status =  XDcfg_CfgInitialize(DcfgInstPtr, XDcfg_0, XDcfg_0-
>BaseAddr);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

// Deselect PCAP as the configuration device as we are going to use the
ICAP
    XDcfg_ClearControlRegister(DcfgInstPtr, XDCFG_CTRL_PCAP_PR_MASK);

    ConfigPtr = XHwIcap_LookupConfig(XPAR_AXI_HWICAP_0_DEVICE_ID);
    if (ConfigPtr == NULL) {
        return XST_FAILURE;
    }

    HwIcapInstPtr = &HwIcap;
    Status = XHwIcap_CfgInitialize(HwIcapInstPtr, ConfigPtr,
        ConfigPtr->BaseAddress);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
    XHwIcap_Reset(HwIcapInstPtr);
```

```
//  while(!XHwIcap_IsDeviceBusy(HwIcapInstPtr));
    print("HWICAP Initialized\r\n");

XTime_GetTime(&tStart); //Starting the timer
PartialAddress = PARTIAL_ADDER_ADDR;
xil_printf("Starting Addition Reconfiguration\n\r");
Status = XDcfg_TransferBitfile(HwIcapInstPtr, (u32 *)PartialAddress,
PARTIAL_ADDER_BITFILE_LEN);
if (Status != XST_SUCCESS) {
    xil_printf("Error : FPGA configuration failed!\n\r");
    exit(EXIT_FAILURE);
}

xil_printf("Addition Reconfiguration Completed!\n\r");

for(int arm_k=0; arm_k<1000; arm_k++){}
/* Remap all 4 64KB blocks of OCM to top of memory. */
    REMAP();

/* Disable L1 cache for OCM. */
Xil_SetTlbAttributes(0xFFFC0000, 0x04de2);

/* Set the owner of all communication areas to 0, which means that only
the
ARM core can write to them. */
Addr = 0xFFFC0000;
for (Proc = 0; Proc < PROCESSORS; Proc++)
{
    Com_area[Proc] = (volatile struct com_area *) Addr;
    Com_area[Proc]->Owner = 0;
    Com_area[Proc]->fin_flag = 0;
    Addr += 0x4000;
}


//unlock SLCR
Xil_Out32(0xF8000008,0xDF0D);
//enable level shifters
Xil_Out32(0xF8000900,0xF);
// clear resets on AXI fabric ports
Xil_Out32(0xF8000240,0x01F33F0F);
Xil_Out32(0xF8000240,0x0);
//lock SLCR
Xil_Out32(0xF8000004,0x767B);
```

```
for(int arm_k=0; arm_k<1000; arm_k++){}

    for(int arm_k=0; arm_k<9000; arm_k++){
        //Benchmark call x times
    }

    XTime_GetTime(&tEnd);
    printf("ARM took %15.5lf seconds.\r\n",(1.0 * (tEnd - tStart)
/COUNTS_PER_SECOND));
    xil_printf("Benchmarks completed \r\n");

    do{
        Com_area[0]->Owner = 1;
        for(int arm_k=0; arm_k<100000; arm_k++){
            arm_dbg +=1;
            arm_dbg -=1;
        }
    } while (Com_area[0]->fin_flag != 1);


    xil_printf("EXITED WITH mb0 flag %d  \n\r",Com_area[0]->fin_flag);
    while (Com_area[0]->Owner != 0);

    do{
        Com_area[1]->Owner = 1;
        for(int arm_k=0; arm_k<100000; arm_k++){
            arm_dbg +=1;
            arm_dbg -=1;
        }
    } while (Com_area[1]->fin_flag != 1);

    xil_printf("EXITED WITH mb1 flag %d  \n\r",Com_area[1]->fin_flag);
    while (Com_area[1]->Owner != 0);

    XTime_GetTime(&tEnd);
    printf("Output took %15.5lf seconds.\r\n",(1.0 * (tEnd - tStart)
/COUNTS_PER_SECOND));

    while(1){

    }

    return 0;
}
```

A2. Microblaze0 softcore

```
#include <stdio.h>
#include <xil_cache.h>
#include <stddef.h>
#include <string.h>
#include <limits.h>
#include <stdlib.h>


//Benchmark Declarations



//End of Benchmark

struct com_area
{
    char Data[256];
    int Owner;
    int fin_flag;

};
/* A pointer to the communication area */
volatile struct com_area * Com_area = (volatile struct com_area *)
0xFFFC0000;

int mb0_dbg = 0;
int main()
{

//    /* Enable the caches. */
    Xil_ICacheEnable();
    Xil_DCacheEnable();


    for(int mb0_j=0; mb0_j<120000; mb0_j++){
        //Benchmark call x times
    }

    while(1){

        while (Com_area->Owner != 1);

           //xil_printf("mb0 completed \r\n");
        for(int mb0_k=0; mb0_k<100; mb0_k++){
            mb0_dbg +=1;
            mb0_dbg -=1;
        }
        Com_area->fin_flag = 1;

        for(int mb0_k=0; mb0_k<100; mb0_k++){
            mb0_dbg +=1;
            mb0_dbg -=1;
        }

        Com_area->Owner = 0;


    };
    return 0;
}
```

A3. Microblaze1 softcore

```
#include <stdio.h>
#include <xil_cache.h>
#include <stddef.h>
#include <string.h>
#include <limits.h>
#include <stdlib.h>


//Benchmark Declarations



//End of Benchmark



struct com_area
{
    char Data[256];
    int Owner;
    int fin_flag;
};

/* A pointer to the communication area */
volatile struct com_area * Com_area = (volatile struct com_area *)
0xFFFC4000;
int mb1_dbg = 0;
int main()
{


      /* Enable the caches. */
    Xil_ICacheEnable();
    Xil_DCacheEnable();


    for(int mb1_l=0; mb1_l<120000; mb1_l++){
        //Benchmark call x times
    }


    while(1){

        while (Com_area->Owner != 1);

       //xil_printf("mb1 completed \r\n");
        Com_area->fin_flag = 1;
        for(int mb1_k=0; mb1_k<100; mb1_k++){
            mb1_dbg +=1;
            mb1_dbg -=1;
        }

        Com_area->Owner = 0;
        for(int mb1_k=0; mb1_k<100; mb1_k++){
            mb1_dbg +=1;
            mb1_dbg -=1;
        }


    };
    return 0;
}
```