Master Thesis

Design and implementation of an ARMv4 tightly coupled multicore in VHDL and validation on a FPGA

Carlos Ariño Alegre

Matriculation Number: 341982
A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in Electronic Engineering

according to the examination regulations at the Technische Universität Berlin for the Diploma in Electronic Engineering.

Department of Computer Engineering and Microelectronics
Embedded Systems Architectures (AES)
Technische Universität Berlin

Author Carlos Ariño Alegre

Thesis period 1st October 2011 to 9th July 2012

Referees Prof. Dr. B. Juurlink, Embedded Systems Architectures, Technische Universität Berlin
Prof. Jaume Comellas Colome, Teoria Senyal i Comunicacions, Universitat Politècnica de Catalunya (UPC)

Supervisor Dipl.-Ing. N. Moser, Embedded Systems Architectures, Technische Universität Berlin
Declaration

According to §13(10) of the Examination Regulations

I hereby confirm to have written the following thesis on my own, not having used any other sources or resources than those listed.

Berlin, 1st October 2011

Carlos Ariño Alegre
Abstract

On one hand, few years ago, increasing the clock speed was the preferred tactic by manufacturers to gradually increase the performance of computers. However, from certain speeds there are some limitations. Some effects of these physical limitations can be the high heat dissipation, problems with the synchronization of the signals, or the high energy consumption. Therefore now, more multicore microprocessors are being developed to increase the performance of the devices without increasing the clock frequency of the microprocessors.

On the other hand, the ARM instruction set architecture is the most widely used 32-bit instruction set architecture in numbers produced and clearly dominates the field of mobile embedded systems.

Therefore, in this document, we achieve to join these two points designing a multicore processor from an existing microprocessor based on soft IPs of general purpose CPUs. The idea is to implement a generic multicore processor with n cores, however during this thesis, a multicore microprocessor with only three ARM architecture cores is used to test the implementation on a FPGA.
Contents

List of Tables ix
List of Figures xi
Listings xiii

1 Introduction 1

2 Background 3
  2.1 The Arm Architecture 3
  2.2 A multicore processor 4
  2.3 On-chip-communication 5
    2.3.1 On-chip-communication topologies 5
    2.3.2 System-on-chip Buses Overview 7
  2.4 FPGA 10
    2.4.1 FPGA architecture 11
  2.5 Pipeline Parallel Programming 12
    2.5.1 Different uses of the pipeline model 13
    2.5.2 Different implementations of the pipeline model 13
  2.6 I/O integration 14
    2.6.1 Port Mapped I/O 14
    2.6.2 Memory-mapped I/O 14

3 Design 17
  3.1 On-chip-communication design choices 17
  3.2 Implementing multicore functionality 19
  3.3 Implementing debug capabilities 20
  3.4 Hardware test / simulation 21
  3.5 Software applications 21

4 Implementation 23
  4.1 Implementation tools 23
  4.2 HDL Hardware description language 23
    4.2.1 VHDL 24
  4.3 Xilinx Virtex 5 Lx110T 24
  4.4 Tools 26
    4.4.1 Design Flow 26
## Contents

4.5 Adapting the processor .............................................. 28
4.5.1 The Digital Clock Manager and the Phase-Locked Loop .... 28
4.5.2 UCF file ......................................................... 29
4.6 Modifying into a multicore processor ............................ 30
4.6.1 A common DCM ............................................... 30
4.6.2 Communication between cores ................................ 30
4.6.3 ArmCORE3, the top-level entity .............................. 30
4.6.4 The control panel .............................................. 32
4.7 The FIFO .......................................................... 33
4.7.1 Creating the FIFO ............................................ 34
4.7.2 Characteristics of the FIFO .................................. 34
4.7.3 The FIFO STATUS register ................................. 37
4.7.4 Operation of the FIFO ........................................ 38
4.8 The communication module ...................................... 39
4.8.1 The operation of the communication module ............... 40
4.8.2 The Chip Select Generator module .......................... 42

5 Results 45
5.1 Showing the operation of the communication module ......... 45
5.2 Testing the multicore processor on the FPGA .................. 47
5.3 Benchmarking ....................................................... 48
5.3.1 Basic Statistics of a Survey ................................ 49
5.3.2 Sobel Filter ................................................... 51
5.4 Implementing N cores ............................................. 56

6 Conclusions 59
6.1 Contributions ....................................................... 59
6.2 Future direction ................................................... 59

7 Appendix 61
7.1 ArmCoreCom code ............................................... 61
7.2 Sobel filter code .................................................. 62
7.3 Hallo Welt assembly code ....................................... 66

Bibliography 67
List of Tables

2.1 List of ARM microprocessor cores ........................................... 4
3.1 Feature comparison between different on-chip-communication topologies. 18
4.1 Addresses of the ArmCoreCom-registers in hexadecimal. ................... 42
5.1 Number of cycles per instruction [7] ....................................... 49
5.2 Basic Statistics of a Survey: number of cycles per task .................... 50
5.3 Results of the Basic Statistics of a Survey application ...................... 51
5.4 Sobel filter: number of cycles per task .................................... 54
5.5 Comparison between original processor and the adaptation to the Virtex-5 56
5.6 Comparison between the implementation of different number of cores ... 57
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Network on chip. [15]</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Principle of a FIFO.</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>AMBA based system architecture [4].</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>Avalon based system architecture. [22]</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>Coreconnect based system architecture. [20]</td>
<td>9</td>
</tr>
<tr>
<td>2.6</td>
<td>Arrangement of Slices within the CLB. [38]</td>
<td>11</td>
</tr>
<tr>
<td>2.7</td>
<td>Memory-mapped I/O bus structure</td>
<td>15</td>
</tr>
<tr>
<td>2.8</td>
<td>Memory-mapped I/O address space</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>Xilinx Virtex 5 Lx110T</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Design Flow.</td>
<td>27</td>
</tr>
<tr>
<td>4.3</td>
<td>DCM and PLL.</td>
<td>29</td>
</tr>
<tr>
<td>4.4</td>
<td>ArmCORE3-block.</td>
<td>30</td>
</tr>
<tr>
<td>4.5</td>
<td>Schematic of the ArmCORE3 high-level entity.</td>
<td>31</td>
</tr>
<tr>
<td>4.6</td>
<td>Schematic of the ArmTop entity.</td>
<td>32</td>
</tr>
<tr>
<td>4.7</td>
<td>Control panel.</td>
<td>33</td>
</tr>
<tr>
<td>4.8</td>
<td>FIFOs represented in the system.</td>
<td>34</td>
</tr>
<tr>
<td>4.9</td>
<td>FIFO-block.</td>
<td>35</td>
</tr>
<tr>
<td>4.10</td>
<td>FIFO status-register.</td>
<td>38</td>
</tr>
<tr>
<td>4.11</td>
<td>Write interface [38].</td>
<td>38</td>
</tr>
<tr>
<td>4.12</td>
<td>Read interface [38].</td>
<td>39</td>
</tr>
<tr>
<td>4.13</td>
<td>The ArmCoreCom-block.</td>
<td>40</td>
</tr>
<tr>
<td>4.14</td>
<td>Connection of the ArmCoreCom module.</td>
<td>41</td>
</tr>
<tr>
<td>4.15</td>
<td>The formation of the Status-FIFO register.</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>ArmCoreCom write and read operations. ArmCoreCom of the Core 1 over the red line, ArmCoreCom of the Core 2 under the red line.</td>
<td>46</td>
</tr>
<tr>
<td>5.2</td>
<td>ArmCoreCom checking status-fifo register. ArmCoreCom of the Core 1 over the red line, ArmCoreCom of the Core 2 under the red line.</td>
<td>47</td>
</tr>
<tr>
<td>5.3</td>
<td>Checking a write operation with Chipscope</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>Checking a write operation with Chipscope</td>
<td>48</td>
</tr>
<tr>
<td>5.5</td>
<td>Diagram of Basic Statistics of a Survey application with a 3-core processor.</td>
<td>50</td>
</tr>
<tr>
<td>5.6</td>
<td>Calculating the total duration of the Basic Statistics of a Survey application in a 3-core processor.</td>
<td>51</td>
</tr>
<tr>
<td>5.7</td>
<td>Operations for the horizontal and vertical differentiation respectively.</td>
<td>52</td>
</tr>
<tr>
<td>5.8</td>
<td>Calculating the gradient magnitude.</td>
<td>52</td>
</tr>
<tr>
<td>5.9</td>
<td>Original image before the Sobel filter.</td>
<td>52</td>
</tr>
</tbody>
</table>
List of Figures

5.10 Horizontal differentiation and vertical differentiation. . . . . . . . . . . . . . . . . 53
5.11 Result of the Sobel filter. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
5.12 Diagram of the Sobel filter with 4-core processor. . . . . . . . . . . . . . . . . . . . . 54
5.13 Calculating the total duration of the Sobel filter application for a image of 100x100 pixels in a 4-core processor. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
5.14 Number of the FIFOs depending the number of cores implemented (n= number of cores implemented). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
Listings

4.1 Variables of ArmConfiguration.vhd ..................................... 29

7.1 Communication module (ArmCoreCom.vhd) ............................. 61
7.2 Main core task - Sobel filter .............................................. 62
7.3 Horizontal differentiation - Sobel filter ............................... 63
7.4 Calculation of the gradient magnitude ................................. 64
7.5 Sending data operation - Hallo Welt assembly code frame .......... 66
7.6 Receiving data operation - Hallo Welt assembly code frame ......... 66
1 Introduction

One way to make use of the vast logic resources found in modern FPGAs is to implement multicore architectures based on soft IPs of general purpose CPUs. The biggest advantage of this solution is the availability of massively parallel computing power, which can be programmed in a straightforward way by using existing software source code. Current FPGAs possess enough logic resources for several instances of suitable embedded CPUs. Beside the possibility to use them independently, there are different approaches to use them together (e.g. processor array, multicore systems). Of course these approaches differ in many ways (e.g. hardware complexity, programming model). When using a multitude of communicating CPU cores in a single system, the problem of interconnecting the cores in a feasible manner needs to be solved. Common architectures for this problem include bus topologies with a single bus instance, partitioned busses and network on chip topologies. Almost all architectures share a common drawback: Core-to-Core communication usually features latency to a degree, which makes it inappropriate for tight CPU coupling and distributed computing problems with data and control dependencies only a couple of clock cycles apart.

We are looking for a lightweight implementation that enables explicit and direct use of multiple core instances. Data should be movable from datapath of one instance straight to another, therefore the different cores has to be coupled very tight. The possibility of using standard compiler to compile C programs for this architecture is essential to use this multicore infrastructure.

The general points to follow during this thesis are:

- Analyse the given ARMv4 micro architecture implementation, modify it to achieve a multicore processor and adapt it on a certain FPGA. In this case, a Virtex 5 board is used.

- Design a concept of a generic infrastructure to couple these cores with minimum impact to existing control and datapath. In order to do this, the principle of a queue processing technique called FIFO (First-In First-Out) is used.

- Implement a synthesizable solution without changing the instruction set architecture. New addresses for sending data to the FIFOs are created, in this way we can use the original instructions of moving data that already has the processor.

- Test and demonstrate the solution on a FPGA developer board with n instances of the core.

The remainder of this thesis is organized as follows: In Chapter 2 we introduce a multicore processor, its architecture and we give an overview of the on chip communication,
1 Introduction

the resources of the used FPGA, the Pipeline Parallel Programming and the methods of mapping I/O. Then in Chapter 3, we focus on the points decided during the design and afterwards we present how these points are implemented in Chapter 4. In Chapter 5 we detail and analyze the results obtained from the experiments. And finally in Chapter 6 we present a summary of the thesis, our conclusions and we discuss the possible improvement implementations that could have our design in a future.
2 Background

This chapter is structured as follows: First in Section 2.1 we present the ARM architecture, the architecture used in our multicore processor, afterwards in Section 2.2, we describe a multicore processor. Then in Section 2.3, we discuss about the different communication-on-chip architectures. In Section 2.4, the device FPGA (field-programmable gate array) used is introduced. In Section 2.5, an overview about the Pipeline Parallel Programming is given. Finally, in Section 2.6, we see different methods of performing input/output between the CPU and peripheral modules in a computer.

2.1 The Arm Architecture

The processor designed for this project is a processor with n cores based in the ARM architecture. ARM is a reduced instruction set computer (RISC) instruction set architecture [9] of 32 bits developed by ARM Holdings. It was named Acorn RISC Machine and later Advanced RISC Machine. The ARM architecture is the most widely used 32 bits architecture. Originally created by Acorn Computers for personal computers purpose, the first ARM product was the Acorn Archimedes introduced in 1987 [27].

ARM is the industry's leading provider of 32-bit embedded microprocessors, offering a wide range of processors based on a common architecture. This architecture deliver high performance, industry leading power efficiency and reduced system cost. The ARM architecture has the broadest ecosystem in the industry with over 900 Partners delivering silicon, tools and software [3]. In 2005, around the 98% of the thousand of millions of mobile phones sold each year used at least one ARM processor [17]. Since 2009, ARM processors are approximately 90% of all 32-bit RISC processors [18] and they are widely used for consumer electronics, including mobile phones, tablets, game consoles, calculators, music players, PDAs, etc.

The ARM architecture is licensable. Companies that are current or former ARM licensees include for example: Apple Inc., Freescale, Intel (through DEC), LG, Microsoft, Nintendo, Nvidia, Sony, Qualcomm, Samsung, Texas Instruments, Yamaha, etc.

In addition to the abstract architecture, ARM offers several microprocessor core designs (Table 2.1). Companies often license these designs from ARM to manufacture and integrate into their own system on a chip (SoC) with other components like RAM, GPUs, or radio basebands (for mobile phones).
2 Background

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Family</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv1</td>
<td>ARM1</td>
</tr>
<tr>
<td>ARMv2</td>
<td>ARM2, ARM3</td>
</tr>
<tr>
<td>ARMv3</td>
<td>ARM6, ARM7</td>
</tr>
<tr>
<td>ARMv4</td>
<td>StrongARM, ARM7TDMI, ARM9TDMI</td>
</tr>
<tr>
<td>ARMv5</td>
<td>ARM7EJ, ARM9E, ARM10E, XScale</td>
</tr>
<tr>
<td>ARMv6</td>
<td>ARM11, ARM Cortex-M</td>
</tr>
<tr>
<td>ARMv7</td>
<td>ARM Cortex-A, ARM Cortex-M, ARM Cortex-R</td>
</tr>
<tr>
<td>ARMv8</td>
<td>No cores available yet. They will support 64-bit data and addressing.</td>
</tr>
</tbody>
</table>

Table 2.1: List of ARM microprocessor cores

The ARM instruction set architecture [13] (ISA ARM) has many versions ARMv4, ARMv5, etc. and all of them are registered in the ARM Architecture Reference Manual [5]. The present thesis is based on a ARMv4 processor designed by Carsten Böhme during his Master thesis "Entwurf und Implementierung einer ARMv4 konformen Mikroarchitektur in VHDL und Validierung auf einem FPGA" [7] (Design and implementation of an ARMv4 microarchitecture in VHDL validation on a FPGA) at the Technische Universität of Berlin.

2.2 A multicore processor

A multicore processor is a single computing component to which two or more processors (called "cores") have been attached for enhanced performance, reduced power consumption, and more efficient simultaneous processing of multiple tasks.

These cores are the units that read and execute program instructions and these instructions are ordinary CPU instructions such as add, move data, and branch, but the multiple cores can run multiple instructions at the same time, increasing overall speed for programs amenable to parallel computing. In this case concerning the Flynn’s taxonomy [9], the classification of computer architectures proposed by Michael J. Flynn in 1966, this type of architecture belongs to the MIMD classification. "Multiple Instruction, Multiple Data" consists in each core fetches its own instruction and performs operation on its own data.

Manufacturers typically integrate the cores onto a single integrated circuit (IC) die or onto multiple dies in a single chip package and these multi-core processors are widely used across many application domains including general-purpose, embedded, network, digital signal processing (DSP), and graphics.

In principle, for example a dual core processor has to be nearly twice as powerful as a single core processor but in practice, performance gains are said to be about fifty per cent: a dual core processor is likely to be about one and a half times as powerful as a single core processor. The Amdahl’s law [2], a law named by the computer architect Gene Amdahl, is used to find the maximum expected improvement to an overall system when only part
2.3 On-chip-communication

Many cores in a single chip can improve performance, but an efficient network intra-chip communication, with low latency and high bandwidth is fundamental for high communication throughput among the cores.

In the next subsections, a classification concerning topology on-chip communication architectures is introduced, and later, in respect to this topology classification, some different types of buses available will be presented.

2.3.1 On-chip-communication topologies

In this subsection different communication-on-chip topologies are presented.

Bus topologies

Concerning topology on-chip communication architectures, a possible general classification for buses topologies [16] is introduced:

**Shared bus:** The system bus is the simplest example of a shared communication architecture topology and is commonly found in many commercial SoCs. Several masters and slaves can be connected to a shared bus. A block, bus arbiter, periodically examines accumulated requests from the multiple master interfaces and grants access to a master using arbitration mechanisms specified by the bus protocol. Increased load on a global bus lines limits the bus bandwidth. The advantages of shared-bus architectures include fairly simple and familiar topology, extensibility, low area cost, easy to build, efficient to implement. The disadvantages of shared bus architecture are larger load per data bus line, longer delay for data transfer, larger energy consumption, and lower bandwidth.

**Hierarchical bus:** this architecture consists of several shared busses interconnected by bridges to form a hierarchy. SoC components are placed at the appropriate level...
2 Background

in the hierarchy according to the performance level they require. Low-performance SoC components are placed on lower performance buses. These buses are bridged to the higher performance buses, in this way the higher performance SoC components do not overwork. Commercial examples of such architectures include the AMBA bus [4], CoreConnect [12], etc. Transactions across the bridge involve additional overhead, and during the transfer both buses remain inaccessible to other SoC components. Hierarchical buses offer large throughput improvements over the shared busses due to: firstly, decreased load per bus; secondly, the potential for transactions to proceed in parallel on different buses; and multiple ward communications can be preceded across the bridge in a pipelined manner.

Ring: in numerous applications, ring based applications are widely used, such as network processors, ATM switches [23]. In a ring, each node component (master/slave) communicates using a ring interface, are usually implemented by a tokenpass protocol [25]. A example of ring topology is the device control register bus (DCR) [12] that will be introduced later.

Network on Chip

The Network-on-chip concept [19] is an approach for the communication sub-system between IP cores on a System-on-a-Chip (SoC) [10].

In a NoC system, modules such as processor cores, memories and specialized IP blocks exchange data using a network. A NoC is constructed from multiple databases interconnected by switches (Figure 2.1). Therefore, making routing decisions at the switches, the data can be sent over the databuses. The big drawback of this communication-on-chip topology is the high latency produced by the movement of data between switches because the routing at the switches is done dynamically and needs resources

![Network on chip](image)

Figure 2.1: Network on chip. [15]

Queue interface using FIFOs

Another possibility of designing an interface between cores is connecting directly a FIFO to the core. This possibility is like the QIF32 [24] (Queue InterFace of 32 bits) developed
2.3 On-chip-communication

by Tensilica. The QIF32 is an addition of two 32-bit queue interfaces for FIFO-like data streaming into and out of the processor.

A FIFO is a principle of a queue processing (First In, First Out). And it primarily is a set of read and write pointers, storage and control logic. Therefore the data are sent one by one from one core to the FIFO. The core assigned to receive the data will take these data according to the order of arrival. The performance of the principle of the FIFO is shown in the Figure 2.2.

![Figure 2.2: Principle of a FIFO.](image)

2.3.2 System-on-chip Buses Overview

At this subsection, some examples of SoC buses are introduced as different possibilities of NoC, just explaining a general description without going into detail.

**AMBA BUS**

AMBA (Advanced Microcontroller Bus Architecture) [4] [16], is a particular bus standard devised by ARM to support on-chip communications among the processor cores manufactured by this particular company. AMBA is hierarchically organized into two bus segments, system- and peripheral-bus, mutually connected via a bridge that buffers data and operations between them (Figure 2.3) [4]. Standard bus protocols for connecting on-chip components, independent of the processor type, are defined by AMBA specifications. However, the method of arbitration is not defined in AMBA specifications. Instead, the arbiter is allowed to be designed as per the suitability of the application requirements.

The three distinct buses specified within the AMBA bus are:

- Advanced High-performance Bus (AHB): The AMBA AHB is for high-performance, high-clock-frequency system modules. It is a system bus used for simple cost-effective designs that support burst transfer, pipelined transfer operation, and multiple bus masters.
2 Background

- Advanced System Bus (ASB): The AMBA ASB is used when the high-performance features of AHB are not required.

- Advanced Peripheral Bus (APB): AMBA APB is optimized for minimal power consumption and reduced interface complexity to support peripheral functions. APB can be used in conjunction with either version of the system bus.

Figure 2.3: AMBA based system architecture [4].

Avalon bus

The bus architecture called Avalon bus [1] is designed for connecting on-chip processors and peripherals together into a system-on-a-programmable chip (SOPC) [34]. As an Altera’s parameterized bus Avalon is mainly used for FPGA SoC design based on Nios processor [26].

The Avalon implement simultaneous multi-master bus architecture 2.4. Masters and slaves interact with each other based on a technique called slave-side (distributed) arbitration. Slave-side arbitration determines which master gains access to a slave, in the event that multiple masters attempt to access the same slave at the same time. The bus masters contend for individual slaves, not for the bus itself. Therefore, multiple masters can be active at the same time and can simultaneously transfer data to their slaves. As long as another master does not access the same slave at the same time, masters can access a target slave without any delay or waiting. The main advantage of this architecture lies in its eliminating the bandwidth-bottleneck as it offers increased bandwidth between peripherals regardless of the bus standard that connects them [22].

The wizard-based Altera’s SOPC Builder system development tool automatically generates the Avalon switch fabric logic. The generated switch fabric logic includes chip-select signals for data-path multiplexing, address decoding, wait-state generation, interrupt-priority assignment, dynamic bus sizing, multi-master arbitration logic and advanced switch fabric transfers. The advanced transfers include streaming transfers, read transfers with latency and bus control signals.
CoreConnect Bus

CoreConnect 2.5[12] [16] is an IBM-developed on-chip bus communications link that enables chip cores from multiple sources to be interconnected to create entire new chips. It was designed to ease the integration and reuse of processor, system, and peripheral cores within standard and custom SoC designs to achieve overall greater system performance. As a standard SoC design point, it serves as the foundation of IBM or non-IBM devices. Elements of this architecture include the processor local bus (PLB), the on-chip peripheral bus (OPB), a bus bridge, and a device control register (DCR) bus. High-performance peripherals connect to the high-bandwidth, low-latency PLB. Slower peripheral cores connect to the OPB, which reduces traffic on the PLB. It is usually used in Microblaze processors.

Figure 2.5: Coreconnect based system architecture. [20]
On the one hand, the PLB bus addresses the high performance, low latency system modules and provides the design flexibility needed in a highly integrated SOC. On the other hand, the OPB bus is optimized to connect to lower speed peripherals and low power consumption. The device control register (DCR) bus is designed to transfer data between the CPU’s general purpose registers (GPRs) and the DCR slave logic’s device control registers (DCRs). Finally, the DCR bus removes configuration registers from the memory address map, which reduces loading and improves bandwidth of the PLB. DCR is synchronous bus based on a ring topology implemented as distributed multiplexer across the chip. It consists of a 10-bit address bus and a 32-bit data bus. The DCR bus is typically implemented as a distributed mux across the chip.

**Fast Simplex Link**

The Fast Simplex Link (FSL) Bus is a uni-directional point-to-point communication channel bus, designed by Xilinx. The FSL interface is available for the Xilinx MicroBlaze processor.

FSL provides a mechanism for unshared and non-arbitrated communication. This can be used for fast transfer of data words between master and slave implementing the FSL interface. Synchronous and asynchronous FIFO modes are supported, this allows the master and slave side of the FSL to clock at different rates.

**Buses conclusion**

Different companies and standards committees have attempted to standardize buses and interfaces with varied results. In this thesis, some of the most implemented buses (AMBA, CoreConnect, Avalon) have been described to show a little overview of the state of art of the actual buses. The FSL communication bus is less famous than the buses mentioned before, but it has been also described to show the vast possibilities of creating new topologies for bus architectures.

However, at last, choosing between all the many different possibilities of buses in SoC designs is a mere decision of the designer and requirements of the system.

**2.4 FPGA**

We used a field-programmable gate array (FPGA) [37] as the target device to test the implementation. A FPGA is an integrated circuit designed to be configured by the customer or designer after manufacturing. The FPGA configuration is generally specified using a hardware description language [32] (HDL) that in our case is the VHDL [11] (this HDL will be introduced later in Subsection 4.2.1). FPGAs can be used to implement any logical function that an application-specific integrated circuit [31] (ASIC) could perform. The ability to update the functionality after shipping, partial re-configuration of a portion of the design and the low non-recurring engineering costs relative to an ASIC design (notwithstanding the generally higher unit cost), offer advantages for many applications.
2.4 FPGA

2.4.1 FPGA architecture

FPGAs contain programmable logic components called 'logic blocks', and a hierarchy of reconfigurable interconnects that allow the blocks to be wired together. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

Some of the used resources of the Virtex 5 FPGA [38] used in this thesis are: CLBs, I/O blocks, BRAM (Block RAM) and DCM. These resources are introduced in this section.

CLB

The Configurable Logic Blocks (CLBs) are the main logic resources for implementing sequential as well as combinatorial circuits. Each CLB element is connected to a switch matrix for access to the general routing matrix 2.6. A CLB element contains a pair of slices. These two slices do not have direct connections to each other, and each slice is organized as a column. Each slice in a column has an independent carry chain. For each CLB, slices in the bottom of the CLB are labeled as SLICE(0), and slices in the top of the CLB are labeled as SLICE(1).

![Figure 2.6: Arrangement of Slices within the CLB. [38]](image)

Every slice contains four logic-function generators (or look-up tables, LUT), four storage elements, wide-function multiplexers, and carry logic. These elements are used by all slices to provide logic, arithmetic, and ROM functions. In addition to this, some slices support two additional functions: storing data using distributed RAM and shifting data with 32-bit registers.
2 Background

I/O block

I/O blocks provide the interface between package pins and the internal configurable logic. Most popular and leading-edge I/O standards are supported by programmable I/O blocks (IOBs). The IOBs can be connected to very flexible ChipSync logic for enhanced source-synchronous interfacing. Source-synchronous optimizations include per-bit deskew (on both input and output signals), data serializers/deserializers, clock dividers, and dedicated I/O and local clocking resources.

BRAM

Block RAM modules provide flexible 36 Kbit true dualport RAM that are cascadable to form larger memory blocks. In addition, Virtex-5 FPGA block RAMs contain optional programmable FIFO logic for increased device utilization. Each block RAM can also be configured as two independent 18 Kbit true dual-port RAM blocks, providing memory granularity for designs needing smaller RAM blocks.

DCM

A digital clock manager (DCM) is a function for manipulating clock signals by: multiply and divide an incoming clock, recondition a clock to, for example, ensure 50% duty cycle, phase shift and eliminate clock skew.

A clock management tile (CMT) blocks provide a flexible, highest-performance clocking for the FPGA. Each CMT contains two digital clock managers (DCM) blocks (self-calibrating, fully digital), and one PLL block (self calibrating, analog) for clock distribution delay compensation, clock multiplication/division, coarse-/fine-grained clock phase shifting, and input clock jitter filtering.

2.5 Pipeline Parallel Programming

Traditionally, software is written for serial computation, in other words, the software runs on a single computer with a single CPU. Therefore the instructions are executed one after another and only one instruction can be executed at any moment in time. But nowadays multicore processors became increasingly widespread and the parallel computing [14] is more used. Parallel computing consist in the simultaneous use of multiple resources to solve a computational problem. In order to do so, the software is divided into discrete parts that can be solved concurrently using multiple CPUs. Consequently, multiple instructions can be executed at any moment in time. In this way, some limits of serial computation mentioned in Section 2.2 are overcome, like for example, the limitations of increasing the clock speed and its cost, because it is more expensive to make a single processor faster than using a larger number of moderately fast commodity processors to achieve the same or better performance.

In this subsection, an overview of the pipeline parallel programming model [6] is given and its typical implementations for multiprocessors. Different ways of implementing
2.5 Pipeline Parallel Programming

Pipeline parallel programming and their impact on the program are described to show the benefits and drawbacks.

Mainly, three reasons motivate us to use pipeline parallel programming. First of all, dividing the software in discrete parts benefits the structure of the development of a large-scale software project. In this way, the project can be divided into smaller, well-defined stages so that different design teams can develop different pipeline stages efficiently. Consequently an improved software quality and lowered development cost are obtained due to simplification of the project and specialization of the developers. Secondly, pipelined programs have clearly defined characteristics between stages, so they can be mapped on different hardware to achieve better hardware utilization. And finally, this model of programming increase program throughput because the different pipeline stages of a workload can operate concurrently from each other, as long as enough input data is available. Using pipeline parallel programing, the load of the program is distributed and the chance for bottlenecks are reduced by keeping data in memory and transferring it directly between the relevant processing elements.

2.5.1 Different uses of the pipeline model

In practice, there are three uses for implementing the pipeline model. The first use is a hybrid model with data-parallel pipeline stages to increase concurrency. The top-level structure of the program is a pipeline, but each pipeline stage is further parallelized so in this way it can process multiple work units concurrently. Thanks to this program structure, the overall concurrency is increased and higher speedups result are obtained.

The second use is a pipelining that allow asynchronous I/O. This model exploits parallelism between the CPUs and the I/O subsystem by increasing concurrency. In order to do so, special non-blocking subsystem are used for the I/Os which manage and move the data obtained into the operating system.

The last use is pipelining to model algorithmic dependencies. This method decomposes a complex program into simpler execution steps with clearly defined interfaces. These steps or dependencies are difficult to analyze and can even change dynamically at runtime, however the developer only needs to keep track of them and expose to the operating system scheduler, which will pick and execute a job as soon as all its prerequisites are satisfied.

2.5.2 Different implementations of the pipeline model

We can implement the pipeline model in two ways. The first way is the "fixed data", this approach uses a static assignment of data to threads. All the pipeline stages to the work unit in the predefined sequence are applied by each thread until the tasks have been completed. This ways is useful for a full replication of the original program, in this way, several instances of the program are executed concurrently.

The advantage of this approach is that it exploits data locality well. Because there is no data movement between threads. However, the drawback is that the working set of the entire execution is proportional to the number of concurrent threads, because if
the working set exceeds the size of the low-level cache such as the level-two cache, this approach may cause many DRAM accesses due to cache misses. So this approach may not be scalable to a large number of CPU cores.

The second way to implement pipeline model is the "fixed code". This approach assigns a pipeline stage to each thread. Only one stage is executed by each thread throughout the program execution and the pipeline structure determines the data movement between threads. Pipeline stages do not have to be parallelized if no more than one thread is active per pipeline stage at any time, which makes this a straightforward approach to parallelize serial code. This fixed code approach is very common because it allows the mapping of threads to different types of computational resources and even different systems.

The advantage of this approach is the flexibility, which overcomes the disadvantages of the fixed data approach. With this approach, the software projects can be divided into well-defined an well-interfaced modules and it allows to the engineers to consider fine-grained processing steps to fully take advantage of hardware. The main drawback of this approach is that each pipeline stage must use the right number of threads to create a load-balanced pipeline. This takes full advantage of the target hardware because the throughput of the whole pipeline is determined by the rate of its slowest pipeline stage. Usually fixed code pipelines implement mechanisms to tolerate fluctuations of the progress rates of the pipeline stages, typically by adding a small amount of buffer space between stages that can hold a limited number of work units if the next stage is currently busy.

2.6 I/O integration

Part of this thesis was to add a new peripheral. In order to do so, the input-output (I/O) system in a computer system can be mapped (addressed) to the CPU mainly in two ways: the port Mapped I/O and the Memory mapped I/O [33].

2.6.1 Port Mapped I/O

In port mapped I/O, the peripheral devices are addressed directly by the CPU using the port addresses. The length of port address is generally less than the length of a memory address. There are specific instructions for carrying out input and output tasks at the ports. This method of mapping does not facilitate the use of memory oriented complex instruction set directly on port addresses.

2.6.2 Memory-mapped I/O

Memory-mapped I/O uses the same address bus to address both memory and I/O devices (Figure 2.7), the memory and registers of the I/O devices are mapped to (associated with) address values (Figure 2.8). So when an address is accessed by the CPU, it may refer to a portion of physical RAM, but it can also refer to memory of the I/O device. Thus, the CPU instructions used to access the memory can also be used for accessing devices. Each I/O device monitors the CPU’s address bus and responds to any CPU access of an
address assigned to that device, connecting the data bus to the desired device’s hardware register. To accommodate the I/O devices, areas of the addresses used by the CPU must be reserved for I/O and must not be available for normal physical memory.

Figure 2.7: Memory-mapped I/O bus structure

Figure 2.8: Memory-mapped I/O address space

One merit of memory-mapped I/O is that, by discarding the extra complexity that port I/O brings, a CPU requires less internal logic and is thus cheaper, faster, easier to build, consumes less power and can be physically smaller; this follows the basic principles of reduced instruction set computing (RISC, the type of architecture used in this multi-core processor), and is also advantageous in embedded systems. The other advantage is that, because regular memory instructions are used to address devices, all of the CPU’s addressing modes are available for the I/O as well as the memory, and instructions that perform an ALU operation directly on a memory operand loading an operand from a memory location, storing the result to a memory location, or both, can be used with I/O
device registers as well. In contrast, port-mapped I/O instructions are often very limited and provide only simple load and store operations between CPU registers and I/O ports. Therefore, to add a constant to a port-mapped device register would require three instructions: read the port to a CPU register, add the constant to the CPU register, and write the result back to the port.

To sum up, after discussing the background, we have to decide which alternatives mentioned in this Chapter 2 we are going to implement for our multicore processor. We have to take into account the scalability of the processor and two of the most important objectives is to achieve our purpose with minimum impact to existing control and datapath and without changing the instruction set architecture.
3 Design

After giving a foundation overview, we discuss several design decisions in this chapter.

Taking the ARMv4 designed by Carsten Böhme [7] as a starting point, we are looking for tight coupled multicore processor design implemented with a minimum possible impact in the core architecture. That is to say, without changing the instruction set architecture (ISA) and with a minimum possible modification of the control and datapath.

The expected multicore processor implementation has a high and easy scalability. In other words, we can implement easily as many cores as the available resources of the FPGA allows us (resources explained in Section 2.4). However, the example presented during the Implementation chapter (chapter 4) is a three-core processor because we think that is easier to explain the performance of the multicore processor with an example with a low number of cores than another one with many cores. In spite of that, in Chapter 5 there are some examples and comparatives of multicore processors implemented with more than three cores.

Following Section 3.1 we discuss the different on-chip-communication possibilities. In Section 3.2 we explain the method used to adapt the original ARMv4 processor to the FPGA board that we use, the implementation of the entity with N cores and the changes to establish multicore functionality. Then in Section 3.3 we introduce the implementations used in order to debug our design. In section 3.4 we introduce the methods to test and simulate the design. And finally in Section 3.5 we explain the decisions made to write some applications in order to obtain results for the comparison between a single core processor and our multicore processor design.

3.1 On-chip-communication design choices

In this present section we discuss the chosen communication between the cores implemented in our design. In order to do so, we take into account the three different on-chip-communications topologies introduced in Subsection 2.3.1.

The first one of these different on-chip-communication topologies consist of using a bus. The second one is using the concept of Network-on-Chip. And the last one is implementing a queue interface between cores.

In order to decide which on-chip-communication we use, we take into account these 5 features: Hardware cost, latency, impact to the control and datapath, the changes on the ISA and the scalability. In Table 3.1 we can see the comparison between the different on-chip-communication possibilities.
3 Design

<table>
<thead>
<tr>
<th>Features</th>
<th>Shared Bus</th>
<th>Hierarchical Bus</th>
<th>Ring Bus</th>
<th>NoC</th>
<th>Queue interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware cost</td>
<td>Low</td>
<td>Medium</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Latency</td>
<td>High</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Impact to the control &amp; datapath</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>Changes on the ISA</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Scalability</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 3.1: Feature comparison between different on-chip-communication topologies.

The advantages of the shared bus topology are the high scalability and the low hardware cost because of its simple topology.

The big drawback of implementing a larger number of cores is that they can not move data between of them at the same time because all them use the same bus. The cores have to wait until the rest of cores stop sending data and, in this way, the bus is available, therefore we have an essential problem with high latency.

Otherwise, the cores can be connected easily to the shared bus, and all of them will be controlled by a common arbiter. However, because of the shared bus protocol, new instructions have to be created, we do not want to cause an impact to the ISA. Moreover, the cores are enabled with a CS signal of the arbiter, and because of the CS signal, we cause impact to the control and datapath.

To sum up, we have the big disadvantages of the high latency, the changes on the ISA and the minimum impact to the control and datapath, so this option is rejected.

The architecture of the hierarchical bus consists of several shared busses interconnected by bridges to form a hierarchy. In our case, we want that all the cores have the same relative amount of authority, so we do not want a hierarchy and we reject this possibility directly. Moreover, this type of bus topology has more or less the same advantages and drawbacks than the shared bus. We consider a "medium" hardware cost because with this topology we have more buses than with the previous one and moreover bridges between the buses.

The latency is different from the preceding bus topology. If we take into account a hierarchy, there will be more buses and therefore more data can be transferred at the same time, but this only decreases a little bit the latency.

The Ring Bus topology has low hardware cost because the cores are connected just with buses forming a ring. However, a token passing protocol cause a high latency because the token has to be passed through all the cores continuously in order to allow the cores to transfer data. Because of this protocol, a larger number of cores are impossible to implement and moreover we need new instructions and control and datapath changes. Therefore, this topology is rejected because we want that all the cores can send and receive data at the same time.
3.2 Implementing multicore functionality

An advantage of the NoC topology allows us to implement a larger number of cores. Looking at Table 3.1, we can consider the hardware cost of the this topology as "medium" because the NoC topology is formed with a lot of buses and switches. Moreover, sending data through all this switches cause a high latency, and besides, we have to implement a way to define the destination of the sent data Therefore we have to change the ISA and finally this possibility is rejected too.

At last, we discuss the queue interface using FIFOs. This option allows us to have a low impact to the control and datapath, because these FIFOs will be connected to the existing bus of the cores. Moreover we do not change the ISA because we use the original instructions of moving data that already has the core are used to send data to the FIFOs. With this topology we can obtain low latency because all the cores can send data at the same time to the FIFOs, and this is one of the important features that we want for our tightly coupled multicore. Finally, with this topology we can also achieve a high scalability. However we consider the hardware cost area "medium" because the number of FIFOs increase considerably with a larger number of implemented cores.

To sum up, considering the low latency, the minimum impact to the control and datapath, not changing the ISA and the high scalability, the implementation of a queue interface topology using FIFOs is chosen.

3.2 Implementing multicore functionality

First of all, the initial ARMv4 processor was designed for a Spartan-3E FPGA board [30], in consequence, the first objective is to adapt the processor to our Virtex 5 FPGA board [38].

To adapt the core we decide to synthetise the VHDL code of the original processor with the ISE program (with the configuration of the Virtex-5), then we correct the different errors that we obtain. These errors are originated because the VHDL code is not designed for our FPGA board.

When we already correct all these errors, we have to modify the UCF file. The UCF file is an ASCII text file that verify constraints in the logical design for programmable chips. We modify this file because we are using a different FPGA with a different package. When we finally adapt the initial ARMv4 to our FPGA, we can start to implement our multicore processor design.

Then we implement N of the adapted cores in only one entity. We present the implementation of three cores in Chapter 4. We do that because we think that in this way, it is easier to understand, the implementation of the multicore processor for the reader. Using certain number of cores makes easier the understanding of the implementation than using all the time the reference of 'N cores'. Moreover, during the debug process of the multicore processor, we use an implementation of three cores processor. In spite of we are explaining a three core implementation, in Chapter 5 we will see implementations with
more than three cores.

Another important point of creating this top-level entity is that we implement a common digital clock manager for all the cores. Because having a digital clock manager for each core will cause problems of information synchronization.

As we mentioned before, we are using FIFOs in order to do our intercommunication between cores. These FIFOs are created with an "IP core generator" of the 'ISE software' (these two programs are introduced in Chapter 4). We chose all the features that the "IP core generator" allows us in order to have a versatile FIFO and, in this way, we can have a flexible communication between cores. In order to implement a communication between two cores, we need two FIFOs, because each FIFO is unidirectional.

Then, when we already have the FIFO’s entity, a new module to manage the FIFO modules has to be created. With this new module we can manage the bus protocol used in the processor. This protocol consist in using an arbiter that enables the corresponding module concerning the instruction executed. Concerning the Section 2.6, the communication module has to be mapped creating new addresses, so we use method of memory-mapped I/O in order to do it. In this way, instructions of moving data that already exist in the processor are used to move data to/from the FIFOs and, therefore, we do not change the ISA, that is one of the important objectives of the thesis.

### 3.3 Implementing debug capabilities

We decide to implement debug capabilities to the core in order to use the debugging process. We debug the three core processor that we mentioned before.

Given the three implementation on the FPGA we decide to use a panel control on the FPGA to select which core we want to debug. When a button of the control panel is pushed, the RS232 [35] communication port of the FPGA will be linked to the corresponding core. Moreover we use three LEDs close of each button. These three LEDs indicate us with core is selected. So when we push one button, its respective core is linked to the RS232 port and its corresponding LED is turned on to indicate us that we select the wished core successfully. In this way, we can had a connexion between the selected core and a PC via RS232. Consequently, we can have an comfortable interaction with the selected core using a PC. The idea is loading applications into the instruction memory on to the processor with the PC and checking the operation with the same PC. We will explain with more detail the hardware test in the following Section 3.4.

On the same mentioned panel, is implemented a reset button. The reset button obviously allows us to reset all the system, and it is a useful feature during the debugging procedure. Besides, eight LEDs are implemented too on the panel control. These eight LEDs indicate us if there is data in the databus. All this features of the control panel will be explained in details in Subsection 4.6.4.
3.4 Hardware test / simulation

In this section we introduce the different decisions for simulating and testing the hardware. Moreover, we discuss the software tests done in order to check the improvement of our multicore processor regarding a single core processor.

Due to the complexity of the design, it is very complicated to simulate it. We decide to simulate parts of the core with the Modelsim simulator (Section 4.4). For testing the whole multicore processor we use a system analyzer (called Chipscope, Section 4.4), that allows us to check the operation of the processor meanwhile it is actually running on the FPGA.

In the previous section we talked about selecting the cores using a button on the FPGA board. Then the RS232 communication port of the FPGA will be linked to the corresponding selected core. Therefore, in this way, we can manage data between the PC and the selected core, and consequently, we can load applications in the processor memory. The possibility of using standard compiler to compile C programs for this architecture is essential, because it is easier to generate these mentioned applications. Therefore, a GCC [21] (GNU Compiler Collection) is used to compile the programs written in C that will be loaded later into the instruction memory on the FPGA. The application written for testing the processor consist in the movement of some data between different cores in order to check the correct operation of the intercommunication implementation (this application is shown in Section 5.2).

3.5 Software applications

In section 2.5, we talk about the methods to program a multicore processor and the motivations to do it. With the multicore processor implemented in this thesis, the three mentioned uses for implementing the pipeline model can be accomplish. For example, establishing an input or output of the FPGA to each core, which manage its own I/O separately, achieving in this way a concurrent program. Or we can implement a pipeline to model algorithmic dependencies, so a complex program is decomposed into simpler execution steps with clearly defined interfaces that run concurrently too.

Moreover, implementing FIFOs we can tolerate fluctuations of the progress rates (Subsection 2.5.2), because the FIFOs work like a buffer space between the cores.

Therefore we decide to implement two applications to demonstrate the hybrid model (Subsection 2.5.1) of pipeline parallel programming. We use this model because we think that is the most promising implementation model for checking the improvement achieved of our multicore processor regarding the initial single core processor.
The first application design, called "Basic Statistics of a Survey", is a simple application that calculates statistics of given data. This application is implemented for a three core processor. The second application, called "Sobel filter", is an edge detection algorithm for images. This application is implemented for a four core processor. In Chapter 5 the mentioned comparison concerning these two applications are shown.

All implementation details of these design choices are explained in the next chapter 4.
4 Implementation

The previous chapters set out the idea of what is a multi-core processor and which type of architecture will be used (ARM architecture), and as it was mentioned, this thesis is about to make the connection between these cores. In this section is about what methods and tools are available for implementing the mentioned task.

In this chapter, firstly we present the tools that we used: the HDL (Section 4.2), the FPGA (Section 4.3) and the software (Section 4.4). Then in Section 4.5 we explain the adaptation of the original given ARMv4 processor to our FPGA. The following Section 4.6 we focus on the points of the implementations to design a multicore processor. In Section 4.7 we introduce the FIFO designed and finally in Section 4.8 we explain the communication module that is in charge of the management of the FIFOs.

4.1 Implementation tools

The processor description is mapped using the development tool Xilinx ISE (integrated software environment) on an Virtex 5 [38] series FPGA. Xilinx, the first and biggest provider of FPGAs today distributes a free version of ISE for different operating systems. Apart from Xilinx there are other manufacturers like Altera with Quartus software or Lattice with IspLEVER software. The description of the FPGA behavior can be developed either within the development environment in a graphical circuit editor (schematic), or by different hardware description languages (HDL) like Verilog or VHDL. In this thesis the ISE 13.3 version we can use Verilog and VHDL, however the HDL chosen is the VHDL.

4.2 HDL Hardware description language

The basic level for FPGA design entry is Register Transfer Level (RTL) which represents a digital circuit as a set of connected primitives (adders, counter, multiplexers, registers etc.). There are two basic ways to create an RTL design: schematic entry and HDL [32] (Hardware description language) entry. Schematic entry is somewhat close to netlist: it is not very convenient to use it for large projects. HDL entry is more convenient, but needs an additional program (synthesizer) in order to translate HDL description to netlist.

Hardware description languages were designed merely to provide means of digital circuits simulation. Synthesizers were created much later. Therefore, each major HDL language has two subsets of language constructs: synthesizable (suitable for synthesis) and non-synthesizable (suitable only for simulation).

The two major HDL languages are VHDL and Verilog. Both of these languages are widespread.
4.2.1 VHDL

VHDL is HDL used in this thesis. VHDL is the acronym that represents the combination of VHSIC and HDL (‘Very High Speed Integrated Circuit’ and ‘Hardware Description Language’ respectively). This hardware description language is used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits.

VHDL was originally developed at the behest of the U.S Department of Defense in order to document the behaviour of the ASICs that supplier companies were including in equipment. That is to say, VHDL was developed as an alternative to huge, complex manuals, which were subject to implementation-specific details.

Now it is a language defined by the IEEE (Institute of Electrical and Electronics Engineers) (ANSI / IEEE 1076-1993) used by engineers to describe digital circuits [11].

Sequential and Concurrent Domains

A VHDL description has two domains: a sequential domain and a concurrent domain. The sequential domain is represented by a process or subprogram that contains sequential statements. These statements are executed in the order in which they appear within the process or subprogram, as in programming languages. The concurrent domain is represented by an architecture that contains processes, concurrent procedure calls, concurrent signal assignments, and component instantiations.

The Register-transfer level

The RTL (register-transfer level), in digital circuit design, is a design abstraction which models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals. RTL abstraction is used in hardware description languages (HDLs) like Verilog and VHDL to create top-level representations of a circuit, from which lower-level representations and ultimately actual wiring can be derived.

4.3 Xilinx Virtex 5 Lx110T

The FPGA used during this project is the Xilinx Virtex 5 Lx110T (Figure 4.1), this FPGA is a feature-rich general-purpose evaluation and development platform with on-board memory and industry standard connectivity interfaces and is a unified platform for teaching and research in disciplines such as for example: digital design, embedded systems, digital signal processing and communications or computer architecture.

Some of the Virtex 5 Lx110T development system features are:

- Two Xilinx XCF32P Platform Flash PROMs (32 Mbyte each) for storing large device configurations.
- Xilinx SystemACE Compact Flash configuration controller.
• 64-bit wide 256Mbyte DDR2 small outline DIMM (SODIMM) module compatible with EDK supported IP and software drivers.

• On-board 32-bit ZBT synchronous SRAM and Intel P30 StrataFlash.

• 10/100/1000 tri-speed Ethernet PHY supporting MII, GMII, RGMII, and SGMII interfaces.

• USB host and peripheral controllers.

• Programmable system clock generator.

• Stereo AC97 codec with line in, line out, headphone, microphone, and SPDIF digital audio jacks.

• RS-232 port, 16x2 character LCD, and many other I/O devices and ports [7].

Figure 4.1: Xilinx Virtex 5 LX110T

The different parts used of the Virtex 5 FPGA in this article are marked in red:

1. Virtex-5 FPGA where is placed our multicore processor.

2. GPIO LEDs, this LEDs represent if there is data in the bus.

3. NESW (Nord, East, South, West) and central buttons, each with a corresponding LED: used to select which core to program

4. GPIO DIP switches: used to load the program.
5. Board power switch.
6. Serial port: used to send the programs to the board.
7. JTAG header: used to program the board and test it.
8. Power connector.

4.4 Tools

Xilinx ISE is a software tool for synthesis and analysis HDL designs, which enable the developer to synthesize their designs, perform timing analysis, examine RTL diagrams, simulate a design’s reaction to different stimulus, and configure the target device with the programmer. A GCC toolchain is used to load the different programs to the FPGA, and for testing/debugging, we used the ChipScope Pro tool. The purposes of each program of the ISE tool, the GCC toolchain and the ChipScope Pro are explained in the next subsection taking as point of reference the design flow that is used.

4.4.1 Design Flow

In Figure 4.2 is shown the process used to achieve the purpose of this thesis. The design flow is divided in three parts, one concerning to the ISE tool, another one to the GCC toolchain and the last one to the ChipScope Pro.

ISE Flow

First of all, the specifications analysis of the block-diagram has to be conceived, when this step is done, the code in VHDL is written with the integrated text editor and afterwards, the simulations are done with the Modelsim 10.1 simulator. This flow repeats until a acceptable result of the desired design-behaviour is obtained.

The next step is synthesizing and implementing on a structural level with timing information using the ISE program and the UCF file. The user constraints file (UCF) is an ASCII file specifying constraints on the logical design. These constraints affect how the logical design is implemented in the target device.

Then the BIT file is ready to be generated. The BIT file contains all of the information necessary to properly configure the FPGA device. It was created by the implementation tools for the design. The program used for this is the Bitgen, integrated already in the ISE interface. When the BIT file is created, the iMPACT program is used to load this file on the FPGA.

GCC toolchain Flow

Finally different programs written in C are compiled with the GCC (GNU Compiler Collection) [21] toolchain and when the assembly compiled file is generated, then the generated binary is loaded into the instruction memory on the FPGA via RS232.
4.4 Tools

However, although a good simulation was obtained with the Modelsim simulator, it is possible that the design on the FPGA does not work as it was expected. For obtaining a completely wished behaviour, the ChipScope program is used. ChipScope tool inserts logic analyzer, system analyzer, and virtual I/O low-profile software cores directly into the design, allowing us to view any internal signal or node, including embedded hard or soft processors. Signals are captured in the system at the speed of operation and

Figure 4.2: Design Flow.

**ChipScope Flow**

However, although a good simulation was obtained with the Modelsim simulator, it is possible that the design on the FPGA does not work as it was expected. For obtaining a completely wished behaviour, the ChipScope program is used. ChipScope tool inserts logic analyzer, system analyzer, and virtual I/O low-profile software cores directly into the design, allowing us to view any internal signal or node, including embedded hard or soft processors. Signals are captured in the system at the speed of operation and
brought out through the programming interface. In this way, the design behaviour is checked easily. Due to the complexity of a three-core processor, it is difficult to do a post-synthesis simulation with the Modelsim tool, consequently it has been decided the option of using the Chipscope tool.

4.5 Adapting the processor

As it was mentioned before, the first step is the adaptation of the original given ARMv4 core to the FPGA board that is used in this thesis. This original ARMv4 core was designed for a Xilinx Spartan 3E FPGA board [30], so the adaptation to the Virtex 5 board [38] has to be made. When the adaptation to the new board was successful, the next step is designing the multicore processor with three of these cores. In this chapter we are going to focus in the adaptation of the ARMv4 core to the Virtex 5 board.

4.5.1 The Digital Clock Manager and the Phase-Locked Loop

The first problem discovered during the adaptation of the core was the Digital Clock Manager (DCM). Primarily, DCMs eliminate clock skew, thereby improving system performance. Similarly, a DCM optionally phase shifts the clock output to delay the incoming clock by a fraction of the clock period. DCMs optionally multiply or divide the incoming clock frequency to synthesize a new clock frequency.

The DCM in the original ARMv4 core was made by an IP CORE block of the Xilinx ISE program. A IP CORE is a reusable unit of logic, cell, or chip layout design that is the intellectual property of one party. Therefore there is the possibility of using IP CORE blocks in the ISE software. These IP CORE blocks are blocks of HDL code that other engineers have already written to perform a specific function, and they are normally used in a complex design where an engineer wants to save time.

In Chapter 5 we can see that the maximum frequency of our multicore processor is 48 MHz. But we choose to debug the processor using the RS232 port, that is to say, we want to see if the processor can send data us via RS232 to the PC to check the correct operation of the processor. In order to do so, the RS232 module of the processor is used, and it works with a 10 MHz frequency, so our three core processor that we are explaining during the thesis work at 10 MHz. However, the minimum frequency that the Virtex 5-DCM can obtain is 19 MHz. Therefore to achieve this frequency, a PLL is needed to support the DCM 4.3. A Phase-Locked Loop (PLL) is a control system that generates an output signal and its phase is related to the phase of an input "reference" signal. The frequency is the derivative of phase. Consequently, a phase-locked loop can track an input frequency, and generate a frequency that is a multiple of the input frequency (in this case, it is wanted to obtain a 10 MHz signal of the 33 MHz clock frequency of the board).

Accordingly, as it was mentioned before, the frequency obtained by the DCM is 60 MHz from the 33 MHz, and now with this 60 MHz it is easier to achieve the 10 MHz with the PLL block. The factors used to obtain all these frequencies are:

- In DCM block: $33 \text{ MHz} \times \left(\frac{20}{11}\right) = 60 \text{ MHz}$
4.5 Adapting the processor

- In PLL block: 60 MHz x (8/48) = 10 MHz

![DCM and PLL](image)

Figure 4.3: DCM and PLL.

After achieving the frequencies with the DCM+PLL IP CORE block, the frequencies have to be declared in the ArmConfiguration.vhd library (Listing 4.1), because there are different modules of the processor that have to know at which frequency the system is working, if not, a bad behaviour of the system may have resulted.

Variables that have to be changed in ArmConfiguration.vhd:

```vhdl
constant ARM_EXT_CLK_FREQ : unsigned(31 downto 0) := to_unsigned(33000000,32);
constant ARM_EXT_CLK_PERIOD : real := 30.030;  -- ns
constant ARM_SYS_CLK_FREQ : unsigned(31 downto 0) := to_unsigned(100000000,32);
constant ARM_SYS_CLK_PERIOD_INT : integer := 100;  -- ns
constant ARM_SYS_CLK_PERIOD : TIME := ARM_SYS_CLK_PERIOD_INT * 1 ns;
constant ARM_DCM_CLKFX_MULTIPLY : integer range 2 to 32 := 20;
constant ARM_DCM_CLKFX_DIVIDE : integer range 1 to 32 := 11;
```

Listing 4.1: Variables of ArmConfiguration.vhd

4.5.2 UCF file

The FPGA used for this project is the Virtex 5 instead the Spartan 3E and we have now a different package for the FPGA, then the pins of the board obviously have changed too. That is why the UCF file has to be modified.

The UCF file is an ASCII file specifying constraints on the logical design. These constraints affect how the logical design is implemented in the target device.

The new variables that have to be modified are: the 8 LEDs that indicate if there are data in the bus, the reset button, the LDP (Load Program) dipswitch, the 33 MHz FPGA clock pin, the RS232 sending data pin and the RS232 receiving data pin. We will talk about this topic in more detail later on because when the three cores are implemented to form the multicore processor, more pins will be created for the UCF file.
4.6 Modifying into a multicore processor

The top level entity that we mentioned in Chapter 3 is called ArmCORE3.vhd. The three cores are inside this entity (we will talk about this entity in more detail later). Three buttons placed on the FPGA board will select each core. When one of this buttons is pressed, the RS232 communication port of the FPGA will be linked to the selected core and its respective LED will turn on to indicate us which of the cores are we using. In this way, it is easy to send data from a PC via RS232 to each core. In Subsection 4.6.4 the control panel with all these buttons is shown.

4.6.1 A common DCM

One of the problems for the architecture of the multicore processor is the DCM, because it is not possible that each core has its own DCM, therefore it is need to use a single DCM for all the cores and it will be situated in the top entity of the system (ArmCORE3.vhd). Owning different DCM produce serious problems of synchrony. In Figure 4.4 we can see the top level implementation view with DCM and PLL.

4.6.2 Communication between cores

As we mentioned before, the FIFOs have been chosen for the communication between the cores, these FIFOs are placed in the top-level entity, like the DCM. But it is needed a new module that controls these FIFOs, this module is called 'ArmCoreCom' (Arm Core Communication), and each core has its own ArmCoreCom module. The FIFOs and the module ArmCoreCom have their respective section later (Section 4.7 and Section 4.8), but they are just mentioned here to show where they are situated in the system.

4.6.3 ArmCORE3, the top-level entity

A representative block of the top-level entity ArmCORE3 is shown in Figure 4.4. As we can see, there are five inputs (left side of the block) and three outputs (right side):

![Figure 4.4: ArmCORE3-block.](image-url)
4.6 Modifying into a multicore processor

The CLK input is the 33 MHz-clock used, then three of the four buttons available on the FPGA are used, one of them is the RST (resets all the system) and the other three are the CORE_BUTTONS (3), these three buttons select which core will be used. Each of these three buttons have its respective CORE_LEDS (3) (three LEDs that represent which core is selected). The LDP_DIPSWITCH loads the program, with the LDP_DIPSWITCH = 1 starts the machine to initialize the memory as soon as the clock signals are stable. Eight LEDs are used to show if there are data in the bus (BUS_LEDS (8)), these LEDs are a way to check if the FPGA is working or not. Finally, as we mentioned, the RS232 port is used to debug the system, so we have the RX_RS232 input, where the FPGA can receive data via RS232, and the TX_RS232 output, where the FPGA can send data via RS232. All these inputs-outputs have to be declared in the UCF file shown at the end of this thesis.

In Figure 4.5, the schematic of the high-level entity or top module is shown. Inside the top module ArmCORE3.vhd, there are the three cores (ArmTop.vhd). These cores are the cores that in Section 4.5 have been adapted to the Virtex 5 board. However, these cores do not have anymore the DCM+PLL inside them, because the DCM+PLL is now in the top module, as it was explained in Section 4.6.1. In this entity there are also all the FIFOs. And moreover, the different libraries are showed in this schematic in gray because they are not really inside the system; they are a support of all the design. In Figure 4.6, we can see the schematic of each core (ArmTop.vhd).

![Figure 4.5: Schematic of the ArmCORE3 high-level entity.](image)
4 Implementation

In Figure 4.6, we can see the schematic of each core (ArmTop.vhd). The modules modified are marked with red: the ArmSystemController.vhd (where before the DCM was there), Armtop.vhd and ArmCoreCom.vhd. This two last modules are explained later.

![Schematic of the ArmTop entity](image)

Figure 4.6: Schematic of the ArmTop entity.

4.6.4 The control panel

Three buttons of the board are used to choose which core to send a program via RS232. When one of this buttons is pushed, the RS232 pins of sending and receiving data are connected to the chosen core, moreover a LED is turned on to indicate which core is selected.
A dipswitch is used to load the program sent via RS232, when the dipswitch is turned off the program cannot be executed.

In Figure 4.7, we can see 5 buttons, NESW (North, East, South and West) and central, and 8 dipswitches. The West, central and East buttons are used to change between the different cores, close to each button there is a respective LED that shows which core is selected. The South button corresponds to the RESET of all the system.

Elements used on the control panel:

- 1 red: button for choosing the core number one
- 2 red: button for choosing the core number two
- 3 red: button for choosing the core number three
- 4 red: button to reset the whole system
- 1 green: LED indicating the core number one is selected
- 2 green: LED indicating the core number two is selected
- 3 green: LED indicating the core number three is selected
- 1 yellow: dipswitch to load the program

Figure 4.7: Control panel.

4.7 The FIFO

As it was mentioned, it has been chosen the FIFOs as method to communication between cores. In this section we can see all the points concerning the creation of the FIFO: characteristics, the operation of the FIFO and the status register used.
4 Implementation

4.7.1 Creating the FIFO

The FIFO is created with an IP CORE of the Xilinx ISE program. There are six FIFOs as we can see in Figure 4.8.

![Figure 4.8: FIFOs represented in the system.](image)

Each core has its own communication module (ArmCoreCom.vhd) that manages the sending and receiving data of the FIFOs connected to this core. The communication module will be explained later in Section 4.8

Different registers are used to send the data from each core to another one, so for example, when we want to send data from core 1 to core 2 we will use the write-register that corresponds to the FIFO1_2 and then, with the core 2 we will use the read-register of the FIFO1_2. If we want to send data from core 2 to core 1 we will use the write-register that corresponds to the FIFO2_1 and then, with the core 2, we will use the read-register of the FIFO2_1. For the rest of options is the same procedure.

Another register is created to know the status of the FIFO. This status-register let us to know if the FIFO is empty, almost empty, full, almost full, if there is overflow, etc. In this way we know if we cannot send more data to the FIFO because is already full or if we cannot receive data from it because is empty. This status-register is explained in Subsection 4.7.3.

4.7.2 Characteristics of the FIFO

The different characteristics chosen for the FIFO and all the FLAGs of the FIFO-block shown in Figure 4.9 are explained in this subsection.

Clock Implementation Operation

The FIFO IP CORE Generator enables FIFOs to be configured with either independent or common clock domains for write and read operations. The independent clock configuration of the FIFO IP CORE Generator enables the user to implement unique clock domains on the write and read ports.
4.7 The FIFO

This implementation category of the FIFO IP CORE generator allows us to select block RAM, distributed RAM, or shift register and supports a common clock for write and read data accesses. The block RAM has been chosen due to the complete feature set supported for this configuration includes status flags (full, almost full, empty, and almost empty). In addition, handshaking and error flags are supported (write acknowledge, overflow, valid, and underflow), and a data count provides the number of words in the FIFO. And we have the option to select a synchronous or asynchronous reset for the core.

Status FLAGS used:

- **Empty Flag**: When asserted, this signal indicates that the FIFO is empty. Read requests are ignored when the FIFO is empty, initiating a read while empty is not destructive to the FIFO.

- **Full Flag**: When asserted, this signal indicates that the FIFO is full. Write requests are ignored when the FIFO is full; initiating a write when the FIFO is full is not destructive to the contents of the FIFO.

- **Almost Empty Flag**: When asserted, this signal indicates that the FIFO is almost empty and one word remains in the FIFO.

- **Almost Full Flag**: When asserted, this signal indicates that only one more write can be performed before the FIFO is full.

Operation FLAGS:

- **Write Enable**: If the FIFO is not full, asserting this signal causes data (on DIN) to be written to the FIFO.
4 Implementation

• Read Enable: If the FIFO is not empty, asserting this signal causes data to be read from the FIFO (output on DOUT).

Handshaking FLAGs:

• Write Acknowledge: This signal indicates that a write request (WR_EN) during the prior clock cycle succeeded.

• Overflow: This signal indicates that a write request (WR_EN) during the prior clock cycle was rejected, because the FIFO is full. Overflowing the FIFO is not destructive to the FIFO.

• Valid: This signal indicates that valid data is available on the output bus (DOUT).

• Underflow: Indicates that read request (RD_EN) during the previous clock cycle was rejected because the FIFO is empty. Underflowing the FIFO is not destructive to the FIFO.

Read mode

There are two read mode options: First-Word Fall-Through (FWFT) and Standard FIFO, the chosen option is the first one. The first-word fall-through (FWFT) feature provides the ability to look-ahead to the next word available from the FIFO without using a read operation, if there is no word in the FIFO, the output is on high impedance. When data is available in the FIFO, the first word falls through the FIFO and appears automatically on the output bus (DOUT). FWFT is useful in applications that require low-latency access to data and the purpose of this project is to achieve the tightest as possible multicore. With a STANDARD FIFO, we have to send a read signal and then the FIFO send the data to the output, in this way the read operation needs one cycle more, so with the FWFT, reading is faster.

Data Port Parameters

• Write/Read Width: 32, due to the 32-bit ARM architecture.

• Write/Read Depth: 128, we can choose a range between 16 and 4194304, but 128 has been chosen because it has been considered enough and if we choose a high value, more FPGA area is used, so more costs we have.

Implementation options

Error Injection and Correction (ECC) Support: the block RAM and FIFO macros are equipped with built-in Error Correction Checking (ECC) in the Virtex-5 FPGA architecture. The FLAGs used for this option are SBITERR (Single Bit Error) and DBITERR (Double Bit Error).

• Single Bit Error: Indicates that the ECC decoder detected and fixed a single-bit error.
4.7 The FIFO

- Double Bit Error: Indicates that the ECC decoder detected a double bit error.

Asynchronous Reset

The asynchronous reset (RST) input asynchronously resets all counters, output registers, and memories when asserted. When reset is implemented, it is synchronized internally to the core with each respective clock domain for setting the internal logic of the FIFO to a known state. This synchronization logic allows for proper timing of the reset logic within the core to avoid glitches and metastable behaviour.

Data count FLAG

Data Count: This bus indicates the number of words stored in the FIFO.

Options rejected

Full/Empty Threshold Value: for example, we can define a value for a Full threshold assert value and this programmable full will assert when number of words in the FIFO is greater than or equal to this value, or we can define a value for a Empty Threshold negate value and this programmable empty will only deassert if numbers of words in the FIFO is greater than this value. All these options were rejected because the processor is a general-purpose device, therefore programming these FLAGs were considered not useful.

4.7.3 The FIFO STATUS register

A status-register is created to know in which status is the FIFO at any time. In this way, the data can be sent carefully looking this register before. The status-register is shown in Figure 4.10. This register consists of a word of 32-bits, but only the range from bit 0 to bit 9 and 12 to 19 are actually used, the rest of the bits are not needed. The reason of having the bits 10 and 11 not used is just a matter of comfort, because in this way, DATA_COUNT is placed in the fourth group of 4-bits and then it is easier to represent the value of DATA_COUNT in hexadecimal in the ChipScope Pro program. In this case is the Chipscope Pro program, however with another program will be also easier to represent and manage the value of DATA_COUNT, then it will be more comfortable for the user of this processor.

In Figure 4.10, there are all the status FLAGs, the handshaking FLAGs and the DATA_COUNT. All these FLAGs were explained in Subsection 4.7.2. This register is created in the top-level entity ArmCORE3.vhd. Comparing this Figure 4.10 with the FIFO-block Figure 4.9, we can see the green outputs for a writing operation (Figure 4.9) correspond with the 4 green FLAGs of writing operation in the the status register (Figure 4.10), and the same for the 7 red outputs (6 FLAGs of one bit and the data count of 8 bits) for a reading operation.
4 Implementation

4.7.4 Operation of the FIFO

Figure 4.11 illustrates the Handshaking flags. On the write interface, FULL is deasserted and therefore writes to the FIFO are successful (indicated by the assertion of WR_ACK). When a write occurs after FULL is asserted, WR_ACK is deasserted and OVERFLOW is asserted, indicating an overflow condition.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>19..12</td>
<td>DATA_COUNT [7..0]</td>
</tr>
<tr>
<td>11..10</td>
<td>NOT USED</td>
</tr>
<tr>
<td>9</td>
<td>DBITERR (ECC)</td>
</tr>
<tr>
<td>8</td>
<td>SBITERR (ECC)</td>
</tr>
<tr>
<td>7</td>
<td>UNDERFLOW</td>
</tr>
<tr>
<td>6</td>
<td>VALID</td>
</tr>
<tr>
<td>5</td>
<td>ALMOST_EMPTY</td>
</tr>
<tr>
<td>4</td>
<td>EMPTY</td>
</tr>
<tr>
<td>3</td>
<td>OVERFLOW</td>
</tr>
<tr>
<td>2</td>
<td>WR_ACK</td>
</tr>
<tr>
<td>1</td>
<td>ALMOST_FULL</td>
</tr>
<tr>
<td>0</td>
<td>FULL</td>
</tr>
</tbody>
</table>

Figure 4.10: FIFO status-register.

Figure 4.11: Write interface [38].

On the read interface (Figure 4.12), once the FIFO is not EMPTY, the FIFO accepts read requests. In standard FIFO operation, VALID is asserted and DOUT is updated on
the clock cycle following the read request. In FWFT operation, VALID is asserted and DOUT is updated prior to a read request being issued. When a read request is issued while EMPTY is asserted, VALID is deasserted and UNDERFLOW is asserted, indicating an underflow condition.

Figure 4.12: Read interface [38].

The underflow flag (UNDERFLOW) is used to indicate that a read operation is unsuccessful. This occurs when a read is initiated and the FIFO is empty. This flag is synchronous with the read clock (RD_CLK). Underflowing the FIFO does not change the state of the FIFO (it is non-destructive).

### 4.8 The communication module

The communication module or ArmCoreCom is the entity in charge of managing the FIFOs. Each core has its own ArmCoreCom that controls four FIFOs (4 in this case, because this multicore processor has three cores, if it has a higher number of cores, more FIFOs are implemented). For example, looking at Figure 4.8, the communication module of the core 1 is responsible for managing the FIFO_1_2, FIFO_2_1, FIFO_1_3, and FIFO_3_1. Managing the FIFOs means sending properly the corresponding signals to achieve a successful write and read operation. In Figure 4.13 the ArmCoreCom module is shown, at the right side there are the signals for the data-bus of the core and at the left side there are the signals the FIFOs. The VHDL code of the ArmCoreCom is shown in Section 7.1.

In Figure 4.15 the communication between core 1 and core 2 is shown. There are two FIFOs to send and receive data, when some data is wanted to write, we use the write register, the same for a reading-operation with the read register. To check the actual status of the FIFO we used the STATUS-register that was mentioned before. In Figure 4.15 we can see how is implemented the data into this register. The information of this register consist in the outputs of the FIFO-blocks 4.9, the 4 green outputs are to see the status of the FIFO when we want to do a write operation and the 7 red outputs (6 FLAGS of one bit and the data count of 8 bits) for a reading operation.

A little example of rendering and receiving data between core 1 and 2. DIN_1_2 4.13 represent the data that will be sent through the FIFO_1_2 to the core 2. The WR_EN_1_2
is the enable signal in charge of the write-operation for the FIFO_1_2 and the same for the RD_EN_2_1 for the read-operation of the FIFO_2_1. The STATUS_FIFO_1_2 signal contain all the information that needs core 1 to sending to core 2 and receiving from core 2 through the corresponding FIFOs. That means that STATUS_FIFO_1_2 has the writing-status of the FIFO_1_2 to send data to core 2 and the reading-status of the FIFO_2_1 to receive data from core 2. In Figure 4.15 we can see the formation of STATUS_FIFO_1_2.

### 4.8.1 The operation of the communication module

First of all, it is important to explain where is placed and connected the communication module. The ArmCoreCom is part of the ArmTop entity, we can see it better in Figure 4.5. The ArmCoreCom is connected between the data-bus of the core and the corresponding FIFOs 4.14.

Seven signals are used to manage the communication module (see Figure 4.15):

- **CS (Chip Select):** control signal that enables the module.
- **DnRW:** control signal that specifies if the operation that will be executed by the communication module is a write-operation or a read-operation, when this signal has a value of '1' means that is a write operation and when it has a value of '0' means that is a read operation.
- **DMAS (Data Memory Access Size):** control signal of 2 bits that specifies the size of the data that will be written or read. This signal is useful for another modules of the core but for writing-reading FIFOs not, because the data used in these operation
4.8 The communication module

Figure 4.14: Connection of the ArmCoreCom module.

will have always a size of 32 bits, so this signal actually is superfluous for the ArmCoreCom module.

- DA (Data Address): signal of 5-bits that indicates the address where the data will be sent. The DA of the data-bus has 32-bits, however in this module only use the 5 less significant bits because with only this 5-bits is enough to know which kind of operation has to execute the communication module. The different addresses of the registers of the communication module will be explained later.

- DIN (Data Input): the data used to send the FIFOs. It is only used when a writing operation is executed.

- DOUT (Data Output): the data read from the FIFOs is sent through this output to the data-bus of the core.

- DABORT (Data Abort): others modules of the core, that manage data of different sizes, use this control signal to indicated that the size of the data for the chosen operation is not correct. This signal depends on the DIN and the DMAS, if they are not correct (they do not correspond between them), then the DABORT indicates to the core that the sent data is not correct for the chosen operation. Consequently, in this module that only manages data of 32 bits, the DABORT will be always disabled.

In Chapter 5 there are simulation examples to understand better the operation of this concrete signals and the general operation of the ArmCoreCom module.
4.8.2 The Chip Select Generator module

The module called ArmChipSelectGenerator (CSG) is the module in charge of the management of the CS signals (see Figure 4.14), it is the arbitrator of the bus. These CS signals enable the corresponding module that has to work according to the requested operation. The CSG will select one module or another depending on the address of the register that we are writing or reading. In Table 4.1 it is shown the different addresses used for the ArmCoreCom module.

<table>
<thead>
<tr>
<th>Register Address Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORECOM_STATUS1_ADDR</td>
<td>0x080004000</td>
</tr>
<tr>
<td>CORECOM_RD1_ADDR</td>
<td>0x080004004</td>
</tr>
<tr>
<td>CORECOM_WR1_ADDR</td>
<td>0x080004008</td>
</tr>
<tr>
<td>CORECOM_STATUS2_ADDR</td>
<td>0x08000400C</td>
</tr>
<tr>
<td>CORECOM_RD2_ADDR</td>
<td>0x080004010</td>
</tr>
<tr>
<td>CORECOM_WR2_ADDR</td>
<td>0x080004014</td>
</tr>
</tbody>
</table>

Table 4.1: Addresses of the ArmCoreCom-registers in hexadecimal.

When one register corresponding to these addresses is used, the CSG use the CS signal to enable the ArmCoreCom, this correspond to the memory-mapped I/O method explained in Section 2.6.2.
In order to achieve the recognition of this address by the CSG, several modifications and address-declarations have been realized in the ArmConfiguration.vhd library.

After explaining the implementation of the multicore processor design, in the next Chapter 5, the results obtained are shown.
5 Results

In this chapter, the operation of the communication module (ArmCoreCom, Section 4.8) is shown. To show it, some simulations made with Modelsim simulator are used, Section 5.1). Moreover, the results of testing a "Hallo Welt" program (it consist of sending the words 'Hallo Welt' from one core to another) are shown. This 'Hallo Welt' program is checked with the ChipScope program (Section 5.2).

Then, in Section 5.3, two benchmarks [8] are shown. Both of these two benchmarks compares the duration in cycles of a determinate application running in a single core processor and in a multicore processor. In this way we can see the improvement of executing an application in a multicore processor.

Finally, a comparative between the original processor design by Carsten Böhme [7] and the same processor adapted to the new FPGA board is shown (Section 5.4). In this way, we see the maximum frequency and resources used of the FPGA obtained with the adaptation. Additionally, to demonstrate the high scalability of this multicore processor, some implementations with different number of cores are shown too.

5.1 Showing the operation of the communication module

In this section we see the correct operations of sending/receiving data done by two communication modules (ArmCoreCom 4.8). Each ArmCoreCom is moving the data to its respective FIFO to send and receive data successfully between the two communication module, in this way, we have the system shown in Figure 4.15.

The Modelsim program is used to do these simulations which are done only with the communication module and FIFOs. As it was mentioned before, due to the complexity of the design of the multicore processor, it is difficult to do a "post place & route" simulation of the whole processor with Modelsim. Therefore, in the debugging process of the project, firstly only the communication module is simulated with Modelsim, and then, the whole multicore processor is tested with the Chipscope program.

In Figure 5.1, the operations of sending/receiving data from/to the FIFO are shown. First of all, we can see the different inputs and outputs of the ArmCoreCom:

- **CS**: Chip Select, if CS = 1, the communication module is enable. Every time that the communication module executes an operation, the CS is asserted.

- **DA**: Address to know where the communication module has to send or receive the data. DA correspond to the last least significant bits of the addresses shown in Table 4.1.
5 Results

- **DnRW:** indicates if the operation is for writing or for reading.
- **DDout:** is the result obtained by the operation.

![Figure 5.1: ArmCoreCom write and read operations. ArmCoreCom of the Core 1 over the red line, ArmCoreCom of the Core 2 under the red line.](image)

We have two of the I/O mentioned before. The second ones (the I/O followed by a number '2') correspond to the second communication module.

The first two operations that we see in Figure 5.1, the ArmCoreCom1 (ArmCoreCom corresponding to the Core 1) is sending the values '1296' and '3840' to the FIFO_1_2 (this FIFO corresponds to the FIFO for sending data from Core 1 to Core 2). Then the ArmCoreCom2 (ArmCoreCom corresponding to the Core 2) is also sending data (the values '1' and '3') to the FIFO_2_1 (this FIFO corresponds to the FIFO for sending data from Core 2 to Core 1). Later these two values are read by the ArmCoreCom1 taking them from the FIFO_2_1, and afterwards, the ArmCoreCom2 takes the values '1296' and '3840' from the FIFO_1_2.

In Figure 5.2 we see that the two first operations correspond to a checking status-FIFO operation. The values in hexadecimal obtained are '00000030', that means that the bits 4 and 5 of the status-FIFO register (Figure 4.10) are asserted. These bits correspond to the Almost Empty and Empty flags, that indicates us that there is nothing stored in the FIFO.

The third operation in Figure 5.2 is a writing operation executed by the ArmCoreCom1, therefore, we obtain a value of '00001060' with the next operation of checking the status of the FIFO done by the ArmCoreCom2. This hexadecimal value means that FIFO is almost full (bit 5 asserted). There is a valid data in the FIFO (bit 6) and there is one word stored in the FIFO (bits 12 to 19 corresponding to the "Data_Count" = 1). Take into account that the Empty flag is not asserted anymore.

Then, the following operation is another writing operation done by the ArmCoreCom1, and the next result of checking the status-FIFO register done by the ArmCoreCom2 is '00002040', this values means that there is valid data in the FIFO (bit 6 asserted) and there are two words stored in the FIFO (bits 12 to 19 corresponding to the "Data_Count" =2). Take into account that the Almost Empty flag is not asserted anymore.
5.2 Testing the multicore processor on the FPGA

In this section the behavior of a multicore processor of three cores is shown. The set up of the whole system consist in: the Virtex-5 FPGA with the 3-core processor, a PC connected to the FPGA board via RS232 to load the 'Hallo Welt' ("Hello World") program and a laptop connected to the FPGA board via a JTAG cable running the Chipscope program to analyze the operation of the processor.

With this set up, we check the correct operation of the communication between the cores. And in next figures 5.3 and 5.4 we can see the correct operation of sending data between the cores with the Chipscope program.

In Figure 5.3, we can see the following signals:

- **out 1**: bus corresponding to the data sent to the FIFO input.
- **top 1 in fifo 2 in**: input of the FIFO.
- **DA 1**: the 5 least significant bits corresponding to the address of the register CORECOM_WR2_ADDR (Table 4.1).
- **DnRW top1**: signal that indicate to the ArmCoreCom that the operation requested is for writing (DnRW=1 in Figure 4.13).

---

Figure 5.3: Checking a write operation with Chipscope

In Figure 5.3, we can see the following signals:

- **out 1**: bus corresponding to the data sent to the FIFO input.
- **top 1 in fifo 2 in**: input of the FIFO.
- **DA 1**: the 5 least significant bits corresponding to the address of the register CORECOM_WR2_ADDR (Table 4.1).
- **DnRW top1**: signal that indicate to the ArmCoreCom that the operation requested is for writing (DnRW=1 in Figure 4.13).
5 Results

- **top 1 wr 2**: signal to indicate to the FIFO that we are writing to the FIFO ("WR_EN=1" in figure 4.9).

- **top 1 rd 2**: signal to indicate to the FIFO that we are reading from the FIFO ("RD_EN=0" in Figure 4.9).

And in Figure 5.4, we can see how the other core reads the 'H' letter from the FIFO:

![Figure 5.4: Checking a write operation with Chipscope](image)

The "top 3 rd2" correspond to the "RD_EN" (Figure 4.9), that means that the Arm-CoreCom is making a reading operation with the FIFO. And the address used is the CORECOM_RD1_ADDR (Table 4.1).

Therefore, to explain the addresses used, in this system of three cores (Figure 4.8), we are seeing in Figure 5.3 that the Core 1 is sending the letters of "Hallo Welt" to the address CORECOM_WR2_ADDR, this address correspond to the Core 3. In Figure 5.4, we can see the Core3 reading from the address CORECOM_RD1_ADDR that correspond to the Core 1.

5.3 Benchmarking

In this section two benchmarks with two different applications are presented. The first application, called "Basic Statistics of a Survey", consist in taking data of a survey from a register and calculate the mode, the least repeated value and the average, when the results are calculated, they are sent back to the register. The second application consist in a "Sobel Filter", an edge detection algorithm (for images). Both of the two applications are an example of the **Hybrid model** (uses of pipeline parallel programing, Subsection 2.5.1).

The benchmarks written in C language and obtaining the assembler code with the GCC toolchain. Then, we use Table 5.1 to calculate the total number of cycles of the application. Therefore, first we write the application for a single core processor and we calculate the total duration of the application in number of cycles. Then we adapt the same application for a multicore processor, and finally we compare the results obtained to demonstrate how much faster the application is executed on the multicore processor.

The code of the different applications used in the benchmarks are shown in the appendix.
### 5.3 Benchmarking

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Number of Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL, MLA</td>
<td>1</td>
</tr>
<tr>
<td>arithmetic-logic, Rd != PC</td>
<td>1</td>
</tr>
<tr>
<td>arithmetic-logic, Rd = PC</td>
<td>5</td>
</tr>
<tr>
<td>LDRH, LDRSH, LDRSB, STRH</td>
<td>1</td>
</tr>
<tr>
<td>STR, STRT, STRB, STRBT, LDRB, LDRBT</td>
<td>1</td>
</tr>
<tr>
<td>LDR, LDRT, Rd != PC</td>
<td>1</td>
</tr>
<tr>
<td>LDR, LDRT, Rd = PC</td>
<td>5</td>
</tr>
<tr>
<td>MRS</td>
<td>1</td>
</tr>
<tr>
<td>MSR, without access of CPSR(7:0)</td>
<td>3</td>
</tr>
<tr>
<td>MSR, with access of CPSR(7:0)</td>
<td>5</td>
</tr>
<tr>
<td>SWAP, SWPB</td>
<td>2</td>
</tr>
<tr>
<td>CDP, MCR</td>
<td>1</td>
</tr>
<tr>
<td>MRC, Rd != PC</td>
<td>1</td>
</tr>
<tr>
<td>MRC, Rd = PC</td>
<td>2</td>
</tr>
<tr>
<td>SWI, Undefined</td>
<td>5</td>
</tr>
<tr>
<td>B, BL</td>
<td>5</td>
</tr>
<tr>
<td>LDC, STC for N Words</td>
<td>N</td>
</tr>
<tr>
<td>STM for N Words</td>
<td>N+1</td>
</tr>
<tr>
<td>LDM for N Words, PC is not part of the registration list</td>
<td>N+1</td>
</tr>
<tr>
<td>LDM for N Words, PC part of the registration list</td>
<td>N+4</td>
</tr>
</tbody>
</table>

Table 5.1: Number of cycles per instruction [7]

#### 5.3.1 Basic Statistics of a Survey

The "Basic Statistics of a Survey" application is a simple program that allows us to understand, in an easy way, the improvement of a program running in a multicore processor environment. This program takes ten values from a register in order to calculate the mode, the least repeated value and the average of these ten values.

To execute this application with a multicore processor, the program is divided in three tasks. Therefore there are three cores, and each core is executing one of these tasks. The first core, called "Core Main", is in charge of taking 10 values from a register and send the values to the other two cores. The second core, called "Core Mode-Least", is in charge of calculating the mode and the least repeated value of the given 10 values, finally the last core called "Core Average", calculates the average of the values. When the mode, the least repeated value and the average are already calculated, this results are sent to the Core Main and it sends this results back to the register. We can see the operation of the application with three cores in Figure 5.5.
5 Results

In order to calculate the mode, the least repeated value and the average, both cores need all the values. Therefore, they do not start to calculate its own task until the Core Main sends all the values. When the Core Main takes one value from the register, this same value is sent immediately to the other two cores. In Table 5.2 we can see the number of cycles per each task.

<table>
<thead>
<tr>
<th>Task</th>
<th>Num. Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>taking one value from the register</td>
<td>8</td>
</tr>
<tr>
<td>sending one value</td>
<td>8</td>
</tr>
<tr>
<td>receiving one value</td>
<td>8</td>
</tr>
<tr>
<td>taking the 10 values from the register and sending them to the other cores</td>
<td>240</td>
</tr>
<tr>
<td>calculating mode-least</td>
<td>527</td>
</tr>
<tr>
<td>calculating average</td>
<td>605</td>
</tr>
</tbody>
</table>

Table 5.2: Basic Statistics of a Survey: number of cycles per task

We clearly can see in Table 5.2 that the bottleneck is the task of calculating the average. The reason for the long duration of this task is the division of 32 bits, because it takes a lot of cycles. In Figure 5.6 we see how the total duration in cycles is calculated.

Therefore, doing the sum of the following number of cycles we obtain the total duration in cycles of the application:

- Initialization of the Core Main = 20 cycles
- Taking the 10 values from the register and sending them to the other cores = 240 cycles
- Receiving the last value (of the Core Average) = 8 cycles
5.3 Benchmarking

- Calculating average = 605 cycles
- Receiving the average result, sending it back to the register and finishing the program = 55 cycles

![Figure 5.6: Calculating the total duration of the Basic Statistics of a Survey application in a 3-core processor.](image)

We have a total duration of the application of 928 cycles. The application running in a single core takes 1,182 cycles, that means that we save 254 cycles (a 21.5%). Moreover, meanwhile the Core Mode-Least and the Core Average are working, the Core Main is idling. Therefore, there are a lot of idle cycles (527+8 cycles). However, we can see this drawback like an interesting advantage if we implement another statistic calculation for the Core Main during these idle cycles.

To sum up, as it is shown in Table 5.3, we achieve a 21.5% speed up of our benchmark application on a three core multiprocessor instance. However, another drawback if we take account the energy consumption, is that we have a large amount of overhead: 2,784 cycles of three cores (928 * 3) Vs. the 1,182 cycles of 1 core.

<table>
<thead>
<tr>
<th>Results</th>
<th>Duration in cycles</th>
<th>% respect of the duration (1 Core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>application 1 core</td>
<td>1182</td>
<td>-</td>
</tr>
<tr>
<td>application 3 cores</td>
<td>928</td>
<td>21.5</td>
</tr>
</tbody>
</table>

Table 5.3: Results of the Basic Statistics of a Survey application

5.3.2 Sobel Filter

A Sobel Filter (or Sobel operator) [36] is an edge detection algorithm. The operator calculates the gradient of the image intensity at each point, giving the direction of the largest possible increase from light to dark and the rate of change in that direction. The codes of this application are shown in Section 7.3.

Mathematically, the operator uses two 3x3 kernels which are convolved with the original image to calculate approximations of the derivatives (one for horizontal changes, and
one for vertical). If we define $A$ as the source image, and $G_x$ and $G_y$ are two images which at each point contain the horizontal and vertical derivative approximations, the computations are shown in Figure 5.7. At each point in the image, the resulting gradient approximations can be combined to give the gradient magnitude, using the operation of Figure 5.8.

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \ast A; \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \ast A$$

Figure 5.7: Operations for the horizontal and vertical differentiation respectively.

$$G = \sqrt{G_x^2 + G_y^2}$$

Figure 5.8: Calculating the gradient magnitude.

To understanding better the concept of the Sobel filter, an example is represented. In Figure 5.9, an original image before the Sobel filter is shown, Figure 5.10 shows the horizontal and the vertical differentiation of the image, finally, Figure 5.11 shows the final result of calculating the gradient magnitude of the two differentiations.

Figure 5.9: Original image before the Sobel filter.
For this benchmark, we decided to divide the application in three different tasks. One core does the horizontal differentiation, another one the vertical differentiation and a third one the calculation of the gradient magnitude. In order to do this, another core is sending continuously the values of the pixels to the two cores in charge of the differentiation, and when they do their respectively differentiation, the results are sent to the last core that is in charge of doing the calculation of the gradient magnitude, finally, these results are sent back to the main core. In Figure 5.12, we can see the operation of the application implemented on a four core multiprocessor instance.

The main difference between this benchmark and the previous one (Basic Statistics of a Survey application, Subsection 5.3.1) is that in this case the cores calculate the differentiation meanwhile they are receiving pixel values. That is to say, these cores do not need to receive all the pixel values in order to do the differentiation. And the 4th core does the same, it calculates the gradient magnitude meanwhile it is receiving values from the differentiation cores.
In Table 5.4 we can see the number of cycles per each task. Clearly the bottleneck is in the operation of the gradient magnitude because the square root needs a lot of cycles to be executed.

<table>
<thead>
<tr>
<th>Task</th>
<th>Number of Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>sending data</td>
<td>8</td>
</tr>
<tr>
<td>receiving data</td>
<td>8</td>
</tr>
<tr>
<td>horizontal differentiation</td>
<td>123</td>
</tr>
<tr>
<td>vertical differentiation</td>
<td>123</td>
</tr>
<tr>
<td>calculating the gradient magnitude</td>
<td>278</td>
</tr>
</tbody>
</table>

Table 5.4: Sobel filter: number of cycles per task

In Figure 5.13 we see how the total duration in cycles is calculated for a example of an image of 100x100 pixels. One important point is that, in order to calculate the first value of the differentiation, we need a matrix of the first 3x3 values of the image to do the convolution. Afterwards, the differentiation cores are receiving columns of three values until finishing the three first rows of the image, then finally, the differentiation cores are able to do the differentiation every time that they receive just one pixel value. To sum up, there are three different times for the differentiation, the first one is getting a first matrix of 3x3 pixel values, the second one is getting columns of three pixel values and the last one is doing the differentiation just receiving one pixel value. For this last one time, we have to take into account that the other pixel values received have been stored. Storing all the pixel values occupy a lot of unnecessary memory. A possible improvement for this application would be erasing the pixel values that have been already convolutionated, in this way we save memory. However, occupying a lot of memory does not matter for the objective of our benchmark. We just want to know the percentage of speed up that we
can achieve.

We have to take into account that there is no differentiation for the first and last row of the image and for the first and last column of the image due to the 3x3 kernel. Therefore we obtain a result image from the Sobel filter of 98x98 pixels. However all the 100x100 pixel are needed in order to do the differentiation, so we send all the 100x100 pixel values to the differentiation cores but the differentiation cores send a result of 98x98 pixels to the 4th core.

If we want to improve the application, there is a possibility of achieving a result image of 100x100 pixels. In order to achieve this result image, the differentiation cores have to modify the image that has to be convolutioned. The rows and columns of pixels of the image edges has to be duplicated in virtual rows and columns. In this way the 3x3 kernel is able to do the convolution. However, having a result image size smaller does not matter for the objective of our benchmark. As we mentioned before, we just want to know the percentage of speed up that we can achieve.

The task of calculating the gradient magnitude takes more time than the calculation of the differentiation. Therefore, the 4th core never wait for the results of the differentiation except for the first one.

Figure 5.13: Calculating the total duration of the Sobel filter application for a image of 100x100 pixels in a 4-core processor.

In Figure 5.13, the parts of calculating the first value of the differentiation includes the number of cycles of receiving the data and sending the results to the 4th core. In the part of calculating the gradient magnitude is also included the number of cycles of receiving data from the differentiation cores and the number of cycles of sending the results to the main core.

Another point to consider is that the calculation of the gradient magnitude is not always the same for all the pixel values. If the pixel value is big, the square root takes more time to be executed. Thus, we use an average value to show the average duration. The input images for the Sobel filter are in black and white, and generally the pixel values are data of 8 bits. So, in the grey scale, there is a range of values between 0 and 256. Our estimation is the average of this value, 128.

In order to calculate the total duration in cycles of the Sobel filter application with a 4-core processor, we do the sum of the following number of cycles:
\section*{5 Results}

- Initialization of the differentiation cores = 38 cycles
- Obtaining the first value of the differentiation = 203 cycles
- Calculating the gradient magnitude of the 98x98 pixel values = 300,793,208 cycles
- Last result received and finishing the program = 19 cycles

We have a total duration in cycles of the application of 300,793,468 cycles. The application running on a single core takes 547,857,118 cycles, that means that we save 247,063,650 cycles (a 45\%). We achieve an improvement of almost the half of the duration compared with a single core. Another interesting point is that this percentage also scales for bigger images. Moreover, if a faster square root is achieved, the bottleneck will be reduced significantly and we will obtain percentages higher than 45\%, and therefore, very interesting faster applications.

Without doubt it is demonstrated that the application executed in this multicore processor run faster than in a single core processor. An interesting point is that writing and reading a FIFO takes only 8 cycles, so the overhead of one transmitted data is only of 16 cycles, that is not a important high number of cycles.

\subsection*{5.4 Implementing N cores}

In this section, results of implementing different number of cores are shown. Firstly, in figure 5.5 we see a comparative of the original processor designed by Carsten Böhme and the new adaptation to our FPGA.

<table>
<thead>
<tr>
<th>Versions</th>
<th>Configuration</th>
<th>Slices</th>
<th>%</th>
<th>LUTs</th>
<th>%</th>
<th>Fmax (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 CORE Spartan-3E</td>
<td>AREA</td>
<td>1677</td>
<td>36</td>
<td>3556</td>
<td>38.2</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>SPEED</td>
<td>2181</td>
<td>46.8</td>
<td>4526</td>
<td>48.6</td>
<td>48</td>
</tr>
<tr>
<td>1 CORE Virtex-5</td>
<td>AREA</td>
<td>1518</td>
<td>8</td>
<td>4139</td>
<td>5</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>SPEED</td>
<td>1126</td>
<td>6</td>
<td>4339</td>
<td>6</td>
<td>56.3</td>
</tr>
</tbody>
</table>

Table 5.5: Comparison between original processor and the adaptation to the Virtex-5

We see in Table 5.5 an improvement of the maximum frequency from 48 MHz to 56.3 MHz for a configuration of speed optimization and from 32 MHz to 47 MHz for an area optimization. And we can see too that the percentage of resources used of the FPGA (Slices and LUTs, Section 2.4) are less. This two results are obvious because the Virtex-5 is more powerful than the Spartan-3E. Moreover, there is no normal optimization when using timing constraints.

In Table 5.6, we see the comparison of the single core with multicore versions on Virtex-5 FPGA.

Sometimes an area optimization achieves more frequency than a speed optimization and a speed optimization achieves less percentage of resources used than an area optimization.
5.4 Implementing N cores

<table>
<thead>
<tr>
<th>Num. Cores</th>
<th>Config.</th>
<th>Slices</th>
<th>%</th>
<th>LUTs</th>
<th>%</th>
<th>Fmax (MHz)</th>
<th>Num. FIFOs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AREA</td>
<td>1518</td>
<td>8</td>
<td>4139</td>
<td>5</td>
<td>47</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>SPEED</td>
<td>1126</td>
<td>6</td>
<td>4339</td>
<td>6</td>
<td>56.3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>AREA</td>
<td>5796</td>
<td>33</td>
<td>13995</td>
<td>20</td>
<td>42.8</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>SPEED</td>
<td>5484</td>
<td>31</td>
<td>14542</td>
<td>21</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>AREA</td>
<td>7046</td>
<td>40</td>
<td>18838</td>
<td>27</td>
<td>44.8</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>SPEED</td>
<td>7308</td>
<td>42</td>
<td>19493</td>
<td>28</td>
<td>48.9</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>AREA</td>
<td>10505</td>
<td>60</td>
<td>29333</td>
<td>42</td>
<td>48.9</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>SPEED</td>
<td>10617</td>
<td>61</td>
<td>30184</td>
<td>43</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>AREA</td>
<td>13677</td>
<td>79</td>
<td>40896</td>
<td>59</td>
<td>43.8</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>SPEED</td>
<td>13879</td>
<td>80</td>
<td>41974</td>
<td>60</td>
<td>44.4</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.6: Comparison between the implementation of different number of cores

\[
\sum_{k=1}^{n} (n - k) \cdot 2
\]

Figure 5.14: Number of the FIFOs depending the number of cores implemented (n= number of cores implemented).

The reason of this is the high complexity of the design. Moreover, the optimization made was the default optimization of the program without adding more constraints.

The maximum frequency achieved with a multicore processor is less than with a single core processor, this is due to the DCM+PLL is placed in the top-level entity and now the clock signal is distributed to all the cores, increasing the timing critical path. So the maximum frequency is stable until an eight core processor is implemented (44.4 MHz), therefore, with more cores is possible that the maximum frequency decrease a little bit more. However, 44.4 MHz is a good result taking into account that a significant number of cores is available.

We see that the number of FIFOs increases considerably when we implement more cores. Each FIFO only uses around 60 slices and 57 LUTs comparing with the 2000 slices and 4490 LUTs that is used by each core (ArmTop, figure 4.6). We have to consider that the FIFOs have a Write/Read Depth of 128, if we want FIFOs with more capacity, they will obviously use more resources. The formula to know the number of FIFOs concerning to the number of cores implemented is shown in figure 5.14.

Looking at the resources used, we can see that with 8 cores we have a percentage of 80% of slices used, so maybe is possible to achieve a 10-core processor with this same Virtex-5 board with and an area optimization (because the speed optimized is probably very close to the 100%, considering the extrapolation made). It is a very interesting result achieving a 10-core processor, although the maximum frequency could be around 40 MHz. Definitely a great scalability is demonstrated.
6 Conclusions

6.1 Contributions

Multicore processing is a growing industry trend as single core processors rapidly reach the physical limits of possible complexity and speed, consequently, we arrive to the conclusion that it is more expensive to make a single processor faster than using a larger number of moderately fast commodity processors to achieve the same or better performance.

This thesis makes contributions in the area of multicore intercommunication with a lightweight implementation that enables explicit and direct use of multiple core instances. The different cores are coupled very tightly with a fast communication between them.

With our design we achieve the two important objectives. The first one is that we designed a concept of a generic infrastructure to couple these cores with minimum impact to existing control and datapath. And the second one is that we implemented a synthesizable solution without changing the instruction set architecture. In this way we accomplish the mentioned lightweight implementation.

With the design done in this thesis, implementing a larger number of cores can be done in a easy way. Therefore, we can achieve a multicore processor with as many cores implemented as resources have our FPGA. We can conclude that our design accomplishes a high scalability. In this way, having a high scalability let us to run applications faster than with a single core processor.

Concluding, this design allows us to make less expensive processors with a larger number of cores to achieve the same or even a better performance than making single core processors.

6.2 Future direction

At this point we demonstrate that the multicore processor designed in this thesis has a great scalability and the applications can be executed faster. However, a core is consuming energy meanwhile is waiting for available data in the FIFO. For example, with the Sobel filter application, the 4th core is waiting during 242 cycles for the first value of the differentiation, and during all this time is consuming energy without doing anything except waiting for data, therefore this core is wasting energy.

The energy consumption is a critical point during a design of a portable device. If the whole battery is consumed in few minutes, does not matter if the processor is very fast. If we are thinking about the possibility of using this design with an ASIC, definitely it has to be improved for a low energy consumption.
6 Conclusions

One of the possibilities is to implement a signal between cores that enables the operation of them. Therefore, with the example mentioned before of the Sobel filter application, the 4th core will be in a sleeping-mode (a low consumption energy mode) and when the differentiation cores have already calculated the first value, they send an enable signal to the 4th core therefore, in this way, it can start working properly. Concerning to the management of a possible sleep-mode: if one core is sending data to another one but the FIFO is already full, the core could stop sending data and pass to the sleep-mode.

Respecting the Sobel filter application, in the case of having full the FIFOs between the differentiation cores and the 4th core, we have another implementation for not having indling cores waiting to available FIFOs. This implementation would be a load balancing. Therefore, one of the differentiation cores could do also the task of the 4th core (calculating the gradient magnitude), and in this way, we reduce the workload of the 4th core. However, now we have results that come from different cores (the 4th core and one of the differentiation cores). So, for the main core is difficult to know the order of the results. Therefore, we need to implement now an ID for the results, in this way, the results of the calculation of the gradient magnitude (that come from the 4th core and the differentiation core) can be reorganized in the main core.

To sum up, the two mentioned possible future directions for the multicore processor designed are: the implementation of a sleep-mode and the implementation of IDs in the data that we send.

Finally, considering the ideas of an ARM processor (that is already thought for portable devices), the speed of a multicore processor and the improvement of a low energy consumption, we could have a good candidate for a powerful processor in the portable devices area.
7 Appendix

7.1 ArmCoreCom code

In Listing 7.1, the code of the communication module of a 3 core processor implementation is shown.

```
---Communication module in charge of the FIFOs management
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
library work;
entity ArmCoreCom is
Port (  
    CCom_CLK : in std_logic;
    CCom_RST : in std_logic;
    CCom_CS : in std_logic; -- signal from chipselector
    CCom_DnRW : in std_logic; -- 1: write or 0: read
    CCom_DMA : in std_logic_vector(1 downto 0); -- type of data 00: bytes, 01: half word, 10: word, 11: Trit formal nicht auf
    CCom_DA : in std_logic_vector(4 downto 0); -- data address
    CCom_DIN : in std_logic_vector(31 downto 0); -- data in (write)
    CCom_DDOUT : out std_logic_vector(31 downto 0); -- data out (read)
    CCom_DABORT : out std_logic; -- abort, always '0'
    CCom_din_A_B : out STD_LOGIC_VECTOR(31 DOWNTO 0);
    CCom_wr_en_A_B : out STD_LOGIC;
    CCom_din_A_C : out STD_LOGIC_VECTOR(31 DOWNTO 0);
    CCom_wr_en_A_C : out STD_LOGIC;
    CCom_dout_B_A : in STD_LOGIC_VECTOR(31 DOWNTO 0);
    CCom_rd_en_B_A : out STD_LOGIC;
    CCom_dout_C_A : in STD_LOGIC_VECTOR(31 DOWNTO 0);
    CCom_rd_en_C_A : out STD_LOGIC;
    CCom_STATUS_FIFO1_A_B : in STD_LOGIC_VECTOR(19 DOWNTO 0); -- STATUS_FIFO register
    CCom_STATUS_FIFO2_A_C : in STD_LOGIC_VECTOR(19 DOWNTO 0)
);  
end ArmCoreCom;
architecture Behavioral of ArmCoreCom is
begin
    --OUTPUT
    CCom_DOUT <= CCom_dout_B_A when CCom_DnRW = '0' and CCom_CS = '1' and CCom_DA(4 downto 2) = '"001"'  
      else CCom_dout_C_A when CCom_DnRW = '0' and CCom_CS = '1' and CCom_DA(4 downto 2) = '"100"'  
      else X'000' & CCom_STATUS_FIFO1_A_B when CCom_DnRW = '0' and CCom_CS = '1' and CCom_DA(4 downto 2) = '"000"'  
      else X'000' & CCom_STATUS_FIFO2_A_C when CCom_DnRW = '0' and CCom_CS = '1' and CCom_DA(4 downto 2) = '"011"'  
      else (others => 'Z');  
    --reading operation
    CCom_rd_en_B_A <= '1' when CCom_CS = '1' and CCom_DnRW = '0' and CCom_DA(4 downto 2) = '"001"' else '0';  
    CCom_rd_en_C_A <= '1' when CCom_CS = '1' and CCom_DnRW = '0' and CCom_DA(4 downto 2) = '"100"' else '0';  
    --writing operation
```
7 Appendix

Listing 7.1: Communication module (ArmCoreCom.vhd)

7.2 Sobel filter code

In this section, the codes of the Sobel filter application are shown. The first one (Listing 7.2) corresponds to the main core. In Listing 7.3, the horizontal differentiation is shown. The vertical differentiation is the same as Listing 7.3 but changing the corresponding value parameters (we can see these value parameters in Figure 5.7). The last code (Listing 7.4) corresponds to the task of the 4th core (calculation of the gradient magnitude).

```
/* sobel.c main */
int main(
    
    volatile unsigned int *CORECOM_STATUS1_ADDR = (unsigned int*) 0X80004000;
    unsigned int *CORECOM_WR1_ADDR = (unsigned int*) 0X80004002;

    volatile unsigned int *CORECOM_STATUS2_ADDR = (unsigned int*) 0X80004003;
    unsigned int *CORECOM_WR2_ADDR = (unsigned int*) 0X80004005;

    volatile unsigned int *CORECOM_STATUS3_ADDR = (unsigned int*) 0X80004006;
    unsigned int *CORECOM_RD3_ADDR = (unsigned int*) 0X80004007;

    const int FIFO_FULL_MASK = 0x00000001; // check if the fifo is full
    const int FIFO_EMPTY_MASK = 0x00000010; // check if the fifo is empty

    int x, y;
    unsigned char original_image[100][100]; // the final result of the sobel filter
    unsigned char filtered_image[98][98];

    // first we send the three complete first rows to start the convolution, because of our 3 x3 kernel
    for (x=0; x<100; x++){
        for (y=0; y<3; y++){
            while (*CORECOM_STATUS1_ADDR & FIFO_FULL_MASK){
                ; // waiting
            }
            *CORECOM_WR1_ADDR = original_image[x][y];
            while (*CORECOM_STATUS2_ADDR & FIFO_FULL_MASK){
                ; // waiting
            }
            *CORECOM_WR2_ADDR = original_image[x][y];
        }
    }

    // then, we can send row by row to continue with the convolution
    for (y=3; y<100; y++){
        for (x=0; x<100; x++){
```

62
```c
while (*CORECOM_STATUS1_ADDR & FIFO_FULL_MASK) {
    // waiting
}

CORECOM_WR1_ADDR = original_image[x][y];
while (*CORECOM_STATUS2_ADDR & FIFO_FULL_MASK) {
    // waiting
}

CORECOM_WR2_ADDR = original_image[x][y];
```

```c
for (y=0; y<99; y++) {
    // now our picture only has 98x98 pixels, because the first and last row and the first and last column can not be convolutioned
    for (x=0; x<99; x++) {
        while (*CORECOM_STATUS3_ADDR & FIFO_EMPTY_MASK) {
            // waiting
        }
        filtered_image[x][y] = *CORECOM_RD3_ADDR;
    }
}
```

### Listing 7.2: Main core task - Sobel filter

```c
int main() {

    volatile unsigned int *CORECOM_STATUS1_ADDR = (unsigned int *) 0X80004000;
    unsigned int *CORECOM_RD1_ADDR = (unsigned int *) 0X80004001;
    volatile unsigned int *CORECOM_STATUS3_ADDR = (unsigned int *) 0X80004006;
    unsigned int *CORECOM_WR3_ADDR = (unsigned int *) 0X80004008;
    const int FIFO_FULL_MASK = 0x00000001; // check if the fifo is full
    const int FIFO_EMPTY_MASK = 0x00000010; // check if the fifo is empty

    int weight[3][3] = {{-1, 0, 1},
                        {-2, 0, 2},
                        {-1, 0, 1}};
    int x, y, i, j;
    int p = 0;
    int pm = 0;
    unsigned char imagec[100][100];
    unsigned char pixel_memo[9604]; // number of the pixels convolutioned: 9604 (98x98), because the first and last row and the first and last column of the image can not be convolutioned

    // at the beginning we need to take three complete rows to do the convolution with our 3x3 kernel
    for (x=0; x<100; x++) {
        for (y=0; y<3; y++) {
            while (*CORECOM_STATUS1_ADDR & FIFO_EMPTY_MASK) {
                // waiting
            }
            imagec[x][y] = *CORECOM_RD1_ADDR;
        }
        if (x>1) { // convolution (for the three complete rows):
            // (x>1) because the first column can not be convolutioned
            pixel_memo[p] = 0; // initialization of the pixel memory
            for (i = -1; i <= 1; i++) {
                for (j = -1; j <= 1; j++) {
                    pixel_memo[p] += weight[i + 1][j + 1] * imagec[x + i - 1][1 + j]; // [1+j] because the first raw can not be convolutioned; [x+i-1] because we start the convolution when we have already taken the third element of the row "(x>1)"
                }
            }
        }
    }

```
```c
for (y=3; y<100; y++) {
    for (x=0; x<100; x++) {
        while ((*(CORECOM_STATUS1_ADDR & FIFOEMPTY_MASK)) != 0x00000001) {
            // waiting
        }
        imagec[x][y] = *(CORECOM_RD1_ADDR);
        if (x>1) {
            pixel_memo[p] = 0;  // initialization of the corresponding pixel memory
            for (i = -1; i <= 1; i++) {
                for (j = -1; j <= 1; j++) {
                    pixel_memo[p] += weight[i + 1][j + 1] * imagec[x + i - 1][y - 1 + j];
                    // "[y - 1 + j]" because when we are taking pixels for a new row from
                    // the first core, we are convolutioning the previous row
                }
            }
            p = p + 1;
            // try to send the data, if the fifo is full, the next position is memorized
            if ((*(CORECOM_STATUS3_ADDR & FIFOFULL_MASK)) != 0x00000001) {
                *(CORECOM_WR3_ADDR = pixel_memo[pm]);  // we send the result to the core 4
                pm = pm + 1;
            }
        }
    }
}
```

Listing 7.3: Horizontal differentiation - Sobel filter

```c
unsigned int unsigned_divide(unsigned int dividend, unsigned int divisor) //DIVISION
{
    unsigned int t, num_bits;
    unsigned int q, bit;
    int i;
    unsigned int remainder = 0;
    unsigned int quotient = 0;
    num_bits = 8;
    for (i = 0; i < num_bits; i++) {
        bit = (dividend & 0x80000000) >> 31;
        remainder = (remainder << 1) | bit;
        t = remainder - divisor;
        q = !(t & 0x80000000) >> 31;
        dividend = dividend << 1;
        quotient = (quotient << 1) | q;
        if (q) {
            remainder = t;
        }
    }
    return quotient;
} /* unsigned_divide */
```
# 7.2 Sobel filter code

```c
int main() { //now with the results of the horizontal (Gx) and vertical differentiation (Gy):
    // sqrt((Gx^2)+(Gy^2))
     volatile unsigned int *CORECOM_STATUS_ADDR = (unsigned int *) 0X80004000;
     unsigned int *CORECOM_WRI_ADDR = (unsigned int *) 0X80004002;
     volatile unsigned int *CORECOM_STATUS2_ADDR = (unsigned int *) 0X80004003;
     unsigned int *CORECOM_RD2_ADDR = (unsigned int *) 0X80004004;
     volatile unsigned int *CORECOM_STATUS3_ADDR = (unsigned int *) 0X80004006;
     unsigned int *CORECOM_RD3_ADDR = (unsigned int *) 0X80004007;
     const int FIFO_FULL_MASK = 0x00000001; // check if the fifo is full
     const int FIFO_EMPTY_MASK = 0x00000010; // check if the fifo is empty

     int i=0;
     int j = 0;
     int x1=0;
     int x2=0;
     int Gx, Gxp, Gy, Gyp;
     int sum=0;
     
     while (1){
         while (*CORECOM_STATUS2_ADDR & FIFO_EMPTY_MASK){  // waiting
            Gx = *CORECOM_RD2_ADDR; // taking results from the horizontal differentiation
         }

         while (*CORECOM_STATUS3_ADDR & FIFO_EMPTY_MASK){  // waiting
            Gy = *CORECOM_RD3_ADDR; // taking results from the vertical differentiation
            Gxp=Gx*Gx;
            Gyp=Gy*Gy;
            sum=Gxp + Gyp;
            
            // ///// square root /////////// of sum
            while( (i*i) <= sum )
                i++;  
            x1=i;
            for (j=0; j<10; j++){
                x2=sum;
                x2 = unsigned_divide(x2, x1);
                x2+=x1;
                x2 = unsigned_divide(x2, 2);
                x1=x2;
               
            }
            // ///// square root ///////////
             
            while (*CORECOM_STATUS1_ADDR & FIFO_FULL MASK){
                // waiting
            }
            *CORECOM_WRI_ADDR = x2;
        }
    }
```

Listing 7.4: Calculation of the gradient magnitude
7 Appendix

7.3 Hallo Welt assembly code

In this section, the assembly code of the Hallo Welt program is shown. In Listing 7.5, the assembly code of sending data operation is shown, and in Listing 7.6, we can see the assembly code of receiving data operation.

```
Listing 7.5: Sending data operation - Hallo Welt assembly code frame

Listing 7.6: Receiving data operation - Hallo Welt assembly code frame
```

66
Bibliography

Bibliography