FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Educational package based on the MIPS architecture for FPGA platforms

João Luís Silva Campos Pereira

Thesis submitted under the course of:

Integrated Master In Electrical and Computer Engineering

Major in Telecomunications

Tutor: Professor José Carlos dos Santos Alves

June 2009

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES 2008/2009

A Dissertação intitulada

"PACOTE EDUCACIONAL BASEADO NA ARQUITECTURA MIPS PARA PLATAFORMA FPGA"

foi aprovada em provas realizadas em 23/Julho/2009

o júri

[illegible]

Presidente Professor Doutor José Alfredo Ribeiro da Silva Matos
Professor Catedrático do Departamento de Engenharia Electrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

[illegible]

Professor Doutor Mário Pereira Véstias
Professor Coordenador do Departamento de Engenharia Electrónica e Telecomunicações e de Computadores do Instituto Superior de Engenharia de Lisboa

[illegible]

Professor Doutor José Carlos dos Santos Alves
Professor Associado do Departamento de Engenharia Electrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projecto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extractos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são correctamente citados.

[illegible]

Autor - JOÃO LUIS CAMPOS PEREIRA

Faculdade de Engenharia da Universidade do Porto

Resumo

O paradigma do ensino tem vindo a evoluir como resultado da constante evolução tecnológica. As recentes evoluções, tanto na implementação de tecnologias digitais como na tecnologia de informação tornam possível um processo de aprendizagem mais activo por parte dos estudantes. Além disso, o acesso ao conhecimento é na sua maioria suportado por sistemas computacionais capazes de processar grandes quantidades de informação. Esta competência requer, entre outros componentes, processadores de elevado desempenho.

A arquitectura MIPS 32bit teve origem no trabalho realizado por uma equipa liderada por John Hennessy, tornando-se em 1984 numa arquitectura pioneira de CPUS de tipo RISC. Juntamente com o seu sucesso no mercado embutido, a arquitectura MIPS 32bit tem vindo a ser largamente utilizada como caso de estudo em Arquitecturas de Computadores por todo o mundo. Nesta dissertação a arquitectura MIPS 32bit foi estudada de modo a que os processares pudessem ser correctamente implementados.

Alguns trabalhos anteriormente existentes já incluem simuladores de processadores MIPS e até implementações de processadores MIPS baseadas em hardware. No entanto, este trabalho refere-se ao desenvolvimento de uma implementação fiel das três versões de processadores referenciadas em "Computer Organization and Design - the hardware/software interface", de John Hennessy e David Patterson, com o intuito de implementar exactamente as mesmas funcionalidades referenciadas no livro. O projecto visou a implementação na Spartan3 XC3S200 presente num kit de desenvolvimento de baixo custo frequentemente usado em fins didácticos em graduações em Engenharia Electrotécnica e de Computadores.

O trabalho desenvolvido compreende ambos os processadores de um ciclo por instrução e de mais de um ciclo por instrução, assim como, uma versão *pipeline* com mecanismos que evitem irregularidades na execução. Estes processadores foram integrados com módulos de controlo adicionais, incluindo contadores de eventos realizados, e com a interface desenvolvida em software para interagir com os referidos processadores.

Abstract

The teaching paradigm has been evolving as a result of the constant technological developments. Recent developments, either in the digital implementation technologies and in information technology make possible a more active learning process by the students. Furthermore, the access to knowledge is mainly supported by computer systems, capable of processing large amounts of information. This competence requires, among other components, high performance processors.

The MIPS 32bit architecture started with the work performed by a team led by John Hennessy and thus became a pioneer in RISC CPUs in 1984. Along with its success in the embedded market, MIPS 32bit is being widely used as a case of study in computer architectures around the world. In this dissertation, the MIPS 32bit architecture was studied so the processors could be properly implemented.

Some previously existent works already comprise MIPS processors simulators and even MIPS based processors implementations in hardware. However, this work addresses the development of a truthful implementation of the three processors versions referenced in “Computer Organization and Design - the hardware/software interface”, of John Hennessy and David Patterson, to implement exactly the same functionalities addressed in the reference text book. The project targeted the Spartan3 XC3S200 which is a low-cost development kit commonly used for teaching purposes in Electrical and Computer Engineering graduations.

The developed work comprehends both single clock and multi clock cycle per instruction processor, as well as, a pipeline version with hazard evading mechanisms. These processors were integrated with additional control modules, including performed event counters, and the software interface developed to interact with the referred processors.

Acknowledgments

This thesis was the consummation of a rich learning period. There were many people who positively contributed to this thesis achievement. To these people I want to thank you for your contribution either in a more technical level inherent to the project development, or in a second stage level providing the proper support.

Therefore, I start by thanking to my family, to my Father, Mother and Brother for encouraging me and for all the economic effort and affective support provided through my entire life.

A very special thank you goes out to my dear Vanessa for the given motivation and support. She patiently tolerated the absence of my person during the project development as well as during the thesis writing when I retreated to long days with my computer.

I would like to express my gratitude to my tutor, Professor José Carlos Alves for the devoted time to my project along with all the recommendations and provided directions towards the project development.

A word of appreciation to Professor João Canas Ferreira is legitimately expressed to acknowledge the enduring suggestions towards the inclusion of features to the developed system.

The burden of writing this thesis was substantially mitigated by the support and humor of my lab mates; Alfredo Moreira, Carlos Resende, João Rodrigues, João Santos, Nuno Pinto and Pedro Santos. Thank you for your recommendations and good company.

I would also like to thank to my "brother in arms" Renato Caldas for the suggestions, and for the enthusiasm shown during the project realization.

Finally, I would like to thank to Tim Parys from the University of Rhode Island and to Sebastian Kuligowski respectively for the Configurable Assembler and the Java serial Port drivers which undoubtedly spared considerable time.

To each of the above mentioned, I extend my deepest appreciation.

João Luís Silva Campos Pereira

“The major difference between a thing that might go wrong
and a thing that cannot possibly go wrong
is that when a thing that cannot possibly go wrong goes wrong,
it usually turns out to be impossible to get at and repair.”

Douglas Adams

Contents

List of Figures

List of Tables

Abbreviations

ARM	Advanced RISC Machine
CISC	Complex Instruction Set Computer
COD	Computer Organization Design
CPI	Clocks per Instruction
CPU	Central Processing Unit
DVD	Digital Video Disc
FEUP	Faculdade de Engenharia da Universidade do Porto
GUI	Graphical User Interface
HD DVD	High Density Digital Video Disc
I-Type	Immediate Instruction Format
IP	Intelectual Property
MIPS	Microprocessor without Interlocked Pipeline Stages
FPGA	Field Programmable Gate Array
RISC	Reduced Instruction Set Computer
R-Type	Register-Register Instruction Format
SoC	System on Chip
SPARC	Scalable Processor ARChitecture

Chapter 1

Introduction

The computer has become an essential tool in the support of the modern teaching methods, managing knowledge at a time when the information is expanding exponentially. The increasingly computation needs among Personal Computers, Servers and Embedded systems demand for more capacity in matter of program execution speed. The computer architecture is a key factor in the computer performance and is subject of study in many universities over the world.

Computer architecture is a major subject in current Electrical and Computer Engineering graduations in its importance in providing to the students the know-how of the processors operation and processor design. In such a technical field of study, the hands-on experience is considered to be a fundamental pedagogic technique for motivating students to understand new topics. Furthermore, learning the hardware organization and the physical limitations associated to the implementation of real processors can help students to understand the hardware foundations of modern computers and consequently foster the development of new processing systems.

Among the existent computer architectures, the 32bits MIPS is considered the architecture of choice for didactic purpose by the prestigious Universities around the world in this field of study. The MIPS 32bit was a pioneer in the CPUs RISC development, being nowadays present in many consumer electronics and embedded systems.

This thesis addresses the computer architecture book “Computer Organization and Design - the hardware/software interface”, of John Hennessy and David Patterson, which is a work of reference on Computer Architectures also used in many universities worldwide [2].

Aiming at educational purposes, the scope of this work is to develop hardware implemented processors targeting low complicity and low cost FPGA boards, which are of common use in Universities for practicing digital design. This is an open source design that the students may freely modify in order to add new functionalities or adapt to their requirements.

1.1 Motivation

The tendency nowadays in the training of the engineers is to make use of recent technologies for better understanding and consolidation of knowledge [5]. The previous observation combined with the importance of computer processor advances in modern society bears the aim of this work. It consists in creating an educational package to support the teaching of Computer Architectures introductory courses as a complement to the previously referenced book. This way, students can improve and extend their knowledge in computer architecture and, at the same time, develop skills and acquire experience on the draft of digital reconfigurable systems (FPGAs). The chosen architecture is MIPS 32bit whose CPUs formed the basis of embedded Processors field since the 1980s until now. To overcome this, the work proposes the development of a set of processors that follow the models addressed in the referenced text book.

1.2 Objectives

Taking out the motivation that led to this work, the general objective was defined as to develop a low-cost educational package targetable to low-cost FPGA boards. This package consisted of processors models studied in the book previously referred, as well as, software tools and their documentation. More detailed objectives are set as:

- develop a single clock cycle per instruction Processor, a multi clock cycle per instruction Processor and a Pipelined Processor version, all in hardware [1];
- develop a Computer-FPGA interface capable of control the processor's clock and provide random access to registers and memory;
- develop counters to monitor performed events, such as number of clocks, executed instructions types, data accesses and memory accesses;
- develop a JAVA Software Graphic User Interface to interact with the FPGA;
- develop a MIPS language assembler;
- develop a support for a Cache Memory;
- writing of the thesis and all the manuals in English;

1.3 Work summary

The developed work includes the integration of two different systems towards the conception of simple and easy to understand processor models. The three proposed processor versions were

[1] from now on, the single clock cycle per instruction Processor, as well as, the multi clock cycle per instruction Processor will be respectively called Unicycle and Multicycle

successfully implemented and comprehend a basic MIPS instruction set. The third processor version complies pipeline hazards resolution techniques, which can be optionally enabled from the interface. As a complement to the processors design, the events performed during the processor runtime can be identified and enumerated. An instruction set dedicated assembler was created in order to spare the tedious and manual assembly language translation, into processor recognizable operational codes. The user interaction with the processors is assured by a Java based graphic user interface, providing total access to both processors and counters inherent storage elements, as well as, to the processors operational modes.

1.4 Document Overview

This work describes the development of hardware Processors implemented in a FPGA board, as well as, the development of the Processors correspondent software interface. This Dissertation is composed of five chapters. In Chapter 2, the MIPS 32bit Architecture, as well as, the referenced Book are reviewed and discussed. In the same chapter are specified both hardware and software platforms, respectively the FPGA used for the project and the software development tool. In the end of the very same chapter, relevant related projects will be addressed. Chapter 3 shows the methodology employed to develop the modules, the used Instruction Language and the specification of the developed modules. Chapter 4 describes the performed implementation and presents the results obtained. In Chapter 5 a review is made about the accomplished objectives and suggestions for improvement are presented.

Chapter 2

Background

This chapter presents the literature considered relevant for this work, the development of the MIPS architecture and discusses the main tools used to carry out the work performed. At the end of the chapter, similar educational projects are addressed.

2.1 Relevant Literature

The most important Literature obviously is the referenced book "Computer Organization and Design - the hardware/software interface", of John Hennessy and David Patterson [2]. Some other books such as "Computer Architecture and Organization", of John P. Hayes [6] or "Computer Architecture - concepts and evolution", of Gerrit A. Blaauw and Frederick P. Brooks, Jr. [7] can be read for complementary perspectives.

Throughout the book, the author follows two approaches. A more theoretically oriented for the readers whose only purpose is to understand the Processor information flow and another one more technical and detailed approach for those with higher ambitions, like understanding the processors hardware details. As it will be seen, some programming methods can be improved by understanding all the processes taken place in the processor operation. Not all the MIPS operations are described in the reference text book, however, suggestions presented by the Author were used in order to implement some of them. The most relevant reading chapters are the second, the third and the fourth chapters which address the computer language, both Unicycle and Multicycle processors and the Pipelining implementation technique.

For a better historical evolution survey the reader can consult “See MIPS Run”, of Dominic Sweetman [1], and in order to take note of related projects in this field of study the reader can also consult “Arquitectura de computadores - dos Sistemas Digitais aos Microprocessadores”, of Guilherme Arroz, José Monteiro and Arlindo Oliveira [8], as well as “Arquitectura de computadores”, of José Delgado and Carlos Ribeiro [9].

2.2 Computer Architectures used for Teaching

Nowadays, there are a few Computer Architectures used in Didactic environments. If a web search is performed, the most common ones are 32bit MIPS, 16bit Motorola 68K and the mobile devices architecture leader, 32bit Arm. Well known reference teaching institutions, such as the Carnegie Mellon University, the University of California in Santa Barbara, the University of California in Berkeley and the Massachusetts Institute of Technology adopt the use of MIPS Architecture, with the referenced COD (Computer Organization and design)text book when concerning the lecture of computer architectures courses, it was considered the most useful and appropriate for this work. Therefore, in this section it will be approached the 32bit MIPS Architecture and a similar one (DLX) whose purpose is not commercial, like in MIPS case, but didactic.

2.2.1 MIPS 32bit Architecture

In 1984, MIPS Computer Systems was founded becoming a pioneer in the CPUs RISC development. It developed one of the first commercial RISC (*Reduced Instruction Set Computer*) microprocessors whose properties included small and highly-optimized instructions rather than more complex instructions as in CISC (*Complex Instruction Set Computer*). RISC processors have a CPI (clock per instruction) of one cycle and usually incorporate a larger number of registers to prevent large amounts of interactions with memory [1]. The company was purchased buy Silicon Graphics, Inc. in 1992, and was spun off as MIPS Technologies Inc. in 1998. MIPS Technologies, Inc. is the world's second largest semiconductor design IP worldwide company with more than 250 customers around the globe [10], with 100 M MIPS CPUs shipped in 2004 into embedded applications. Today, MIPS powers many consumer electronics and MIPS designs are used in SGI's (*Silicon Graphics Incorporated*) computer product line, founding broad application in embedded systems including digital television, broadband access, cable set-top boxes, DVD recorders, HD DVDs and VoIP. Windows CE devices, Cisco routers, the Nintendo 64 console, the Sony PlayStation 2 console, and the Sony PSP handheld system use MIPS processors. The MIPS cores scalability, performance and low power consumption meet the increasing demand for sophisticated digital devices. The architecture is similar to that of other recent CPU designs, including Sun's SPARC, IBM and Motorola's PowerPC, and ARM-based processors [11].

The MIPS Architecture has evolved through the years of existence as a compromise between program demanding and hardware resources. From the MIPS I, in 1985, to the modern MIPS 32/64 architecture, some remarkable improvements concerning the registers width and numbers representation took place. Since the instructions set extensions were inclusive, each new architecture version included the former levels, meaning that a processor can run binary programs from previously implemented instruction sets [1]. Illustrated in figure 2.1.

The MIPS I instruction set comprehended the simplest and basic instructions to what would become a well performed and important RISC architecture. Some of those instructions are present in the instruction set used in this work. The MIPS I processors shipped to market were the R2000 and the R3000. The R3000 took the advantage of a more advanced manufacturing process along

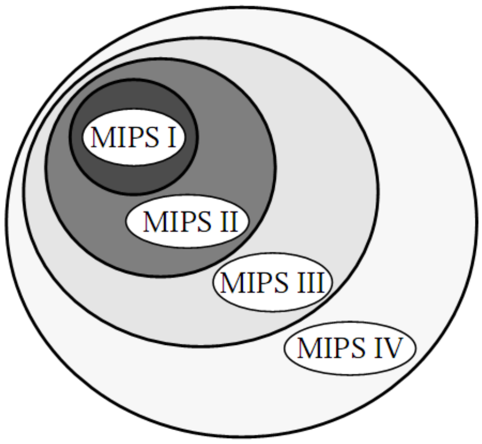

Figure 2.1: MIPS Set extensions, based in [1].

with some well-judged hardware enhancements, giving a substantial improvement in execution speed. Some pioneers embedded applications such as high-performance laser printers and typesetting equipment used R3000. The second set (MIPS II) added load linked, store conditional and branch likely instructions. Although it has not passed beyond the preproduction is the closest set to the modern MIPS32. A couple years later, in 1990, to fulfill the programs growth and consequently longer registers necessity, MIPS became the first 64 bit RISC architecture to reach production with the R4000 processor. That was the MIPS III era. The two following instruction sets (MIPS IV and MIPS V) respectively added the floating point units and the Single Instruction, Multiple Data parallelism technique (SIMD). While the MIPS IV was implemented in two new processors (the R10000 and the R50000), the MIPS V just represented improvements to the previous sets, being present nowadays in MIPS64. Presently, the MIPS32 and MIPS64 are respectively supersets of MIPS II and MIPS IV.

MIPS uses both 32bit and 64 bit architecture but our concern to this work will be only the 32bit, which comprehends 32 registers, each 32 bits wide (a bit pattern of this size is referred to as a word). This way the register addresses are 5 bits long and the data inputs and outputs are 32-bits wide. More than 32 registers might seem better, however, more registers mean more decoders and multiplexers to select individual registers making the design more expensive than using ram, which is cheaper and denser than registers. Nevertheless the registers should be used as much as possible for the simple reason they are faster than ram. Nowadays most compilers can wisely use registers and minimizing ram accesses but in the past using registers intelligently was a programmer concern [2].

It was previously stated that the registers should be used as much as possible, meaning that sometimes the use of ram is inevitably. Arithmetic operations occur only on registers, however, to have data on registers, data transfer instructions must occur. Data transfers occur very often when the computer has to represent complex data structures, such as arrays and structures. The data transfer instruction that copies from the ram to the registers is called Load (*LW* as Load Word), while the data transfer in the opposite way, from registers to ram, is called Store (*SW* as Store Word). Loads and stores use indexed addressing to access ram because, and here comes another important architecture property, the memories are byte-addressable. This means that each memory address is 8 bit referenced, which combined with the 32 address lines result in a 4 Gbyte address space. Therefore, given that a 32 bit word can be occupied by 4 bytes each word must start at address multiple of 4 (0, 4, 8, 12, ...) also called alignment restriction.

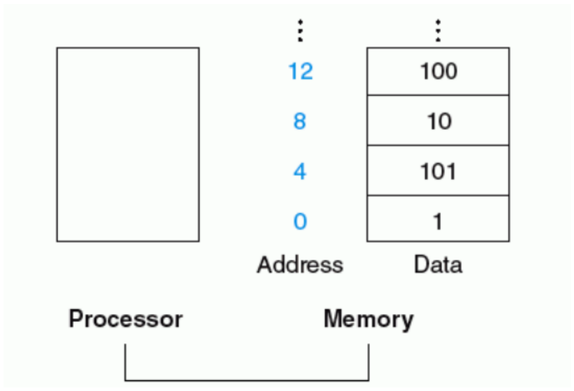

Figure 2.2: Byte addressable Memory, based in [2].

An example of a MIPS instruction can be:

add $s0, $t0, $t1 ,

where the instruction “tells” the processor to compute the sum of the values in registers $t0 and $t1 and store the result in register $s0. In section 3.3 the Instruction Language will be described in more detail [12].

As it was referenced in the previous chapter the basic concept in the development of the MIPS Architecture was to dramatically increase performance with less die area through the use of pipelining technique. Pipelining consists in partially overlap instructions in order to increase performance by "breaking" the instructions process in a series of stages linked between them. Each stage performs a small part of the overall instruction processing while the instruction flows through the datapath. The stages should be perfectly balanced so that each stage takes approximately the same time. The reason for this, concerns the fact that the machine clock cycle is defined by the longest stage execution time. The pipelining paradox is that, for each instruction, the time since the start of the first step until the last one is complete is not shorter than a non-pipelining approach, but when executing several instructions continuously the performance increases so that for the same amount of time the pipelined version surpasses the non-pipelined. The high performance is achieved by parallel processing of multiple instructions at the same time, each in different stages of the pipeline. It is therefore a key processing technique which was introduced in the RISC architectures, being later adopted by CISC architectures [2], [13], [1]. There are, however, some limitations concerning pipelining instruction execution called hazards, which can be grouped in three different types. The Structural Hazards concern the hardware flaw in supporting the combination of instructions, set to execute in same clock cycle. The Data Hazards, which occur when an instruction cannot be executed in the right clock cycle due to the needed data not being available at that time. Some forwarding techniques can be implemented, as we will see in sub-section 3.4.3, to avoid these kind of situations, still, some cases are inevitable. Finally, Control Hazards refer to the issue of the instruction not being executed in the appropriate clock cycle, due to the fact the fetched instruction is not the needed one [2].

2.2.2 DLX Architecture

DLX Architecture is a generic open source 32bit RISC architecture oriented for didactic purpose. In DLX, the complex instructions are simulated in software, by the use of the simple instructions performed by the architecture. It is very similar to MIPS architecture with some evident influences from competing architectures such as DEC VAX, IBM /370 and Intel 8086. Like MIPS, DLX bases its performance on the use of an instruction pipeline and supports the same 3 instructions types (R-Type, I-Type and Jump), yet with some differences in the instruction fields organization. While in the MIPS Instruction Set, the shift amount is set by a reserved 6 bit field called *shamt*, in the DLX Instruction Set there is no shift amount field, and the function field is 11bit wide. This difference allows DLX Set to perform dynamic shift operations, meaning that the amount of shifts can be provided not only by a constant but also by a register value. The difference present in the I-Type Instructions is defined by the feature of the DLX Set to allow the load destination register ([1]) type as unsigned byte, float or double. Both architectures provide byte, half word and word memory loads/stores. The Jump instruction specification is the same as in MIPS Instruction Set since, both architectures allow linked and not-linked jumps ([2]), as well as the two types of jump destination address. The program counter relative address and the register content address [14], [15].

2.3 Hardware Platform

The FPGAs are integrated circuits consisting of configurable logic blocks (CLBs), I/O pads, and routing channels. These routing channels consist of reconfigurable interconnections, which can be local connections, like links between CLBs neighbors, or global connections that can connect CLBs on opposite sides. These CLBs are logic structures which comprise the majority of the die area of a FPGA. Design specification can be done using schematic capture or RTL synthesis tools, using standard HDL languages such as Verilog and VHDL. In order to program the FPGAs, an application circuit must be mapped into an FPGA with adequate resources. FPGAs avoid the high initial cost, the lengthy development cycles, and the inherent inflexibility of conventional ASICs. Adding to that, the FPGA programmability permits design upgrades in the field with no hardware replacement necessary [16], [3], [17], [18].

The concept when the FPGA technology appeared was to build an array of general-purpose logic blocks which could be programmed to produce any possible logic configuration. Since this

[1]or origin register in case of a store.

[2]jump linkage is defined as the storing of the return address in case system call attendance

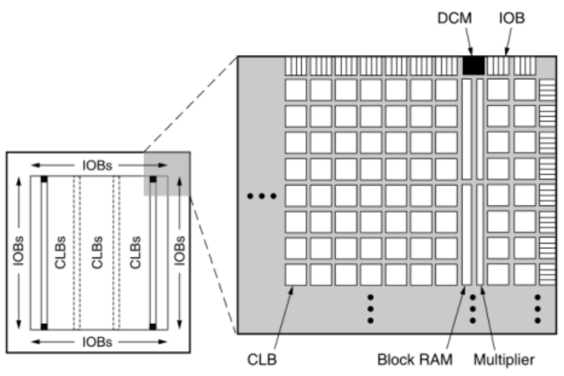

Figure 2.3: Spartan-3 Family Architecture. [3].

approach was limited in its ability to handle more complicated designs, big evolution has taken place on the last 15 years with improvements and market share dominated by Xilinx and Altera. These devices have been constantly evolving in capacity and logic density, being generally used in systems that take advantage of performing multiple tasks simultaneously, optimizing the consumption and space required, with lower operating frequencies than processors. Some of the major evolution steps concern the inclusion of special-purpose resources along with the configurable FPGA fabric. With this evolution arises the complexity of finding an optimal mapping of the design, since fabric logic is flexible but not efficient for all design structures. Fortunately, synthesis tools do a great job on managing the trade-offs inherent to resource allocation and timing constraints. These tools allow a more congruent design verification flow, due to the ease of modeling at the RTL level. Sometimes logic on the critical path of a design is best implemented within fabric logic, while in other occasions, only the use of special resources allow the design to meet performance goals. Other modern FPGAs common resource much more area efficient is built-in multipliers, which can be much faster than a corresponding circuit built with fabric logic. Some architectures, likewise endue dedicated DSP blocks containing a multiply-accumulate function along with pipeline registers, in order to increase performance [19], [20].

The resource management and special resource mapping have become a current issue, since the options available have increased, becoming more challenging for present designers [19].

2.3.1 Spartan 3

The Spartan 3 family (introduced in 2005) uses 90nm process technology and staggered I/O pads with the lowest cost per gate and lowest cost per I/O of any FPGA [16]. It is one popular low cost FPGA used in numerous educational boards and is ideal for applications that have restrictions on power consumption. It contains some platform features such as embedded 18x18 multipliers, up to 1.8Mb of block RAM, and embedded 32-bit and 8-bit soft processors [3].

Without being the most actual and equipped FPGA board, the Spartan3 XC3S200 comprises 4.320 Logic Cells and 200K Logic Gates, being the choice when considering the development of the processors on a low-cost platform. Among the four different types of connectors (40-pin expansion connectors; VGA; RS-232; PS/2) the RS-232 was the chosen to support the Computer-FPGA connection. The main objective is not to achieve the maximum frequency but to target the project to the FPGA available in the most used Xilinx educational boards, the Spartan3 Starter kit. As the project development is not very demanding concerning the clock frequency needed, the design bottleneck was the available resources.

For this work implementation and required designs, the use of multipliers was not necessary since there is no multiplication required. However, the Block RAMs represented a major advantage in resource allocation efficient, being the chosen method to implement the memories, which are integral part of the processors models. Both Unicycle and Pipelined version required an Instruction Memory and a RAM, while the Multicycle required only one Memory (more details about the Processors versions will be presented in section 3.4). The choice of mapping the memories on the Block RAMs was based on theirs size and consequently the fabric logic available on the FPGA. Another important block ram utility was the option to map some ROM logic into the Block RAM. This was a fundamental choice taken in the extra implementation of the Multicycle version (32bits counters) in section 4.1.11 so this processor version could be implemented on the Spartan 3 XC3S400All the design was performed at RTL level optimized for Xilin FPGAs avoiding the need for boards with external memory.

2.4 Software Platform

The software inherent to this work is supposed to support the development of the hardware description and design processes, as well as the development of the JAVA Graphic User Interface. Through this section it will be presented and discussed the programming language used to develop the User Interface capable to interact with the developed hardware Processors.

2.4.1 JAVA

The language Java is a platform-independent, high-level object-oriented programming language developed by Sun Microsystems and commonly used in standard enterprise programming. Some of its major advantages are the existence of free tools, a vast base of knowledge, and a large community of developers. As a result of being built to include a high level of security, the Java program execution succeeds the bytecode verification and the memory leaks are contained within the JVM.

Java Language was mostly influenced by C in its syntax, by C++ in its object orientation and by Smalltalk in its “almost accomplished” portability. Java was created, but with a simpler object model to eliminate language features that cause common programming errors. The platform-independent feature enables Compiled Java code to run on most computers and Operative Systems including UNIX, the Macintosh OS, and Windows. Although its major strength is portability, its weak point is the execution speed which is commonly similar or even slower than, for example, C++. In Java, the source code files need to be compiled into a format called bytecode and then be executed by a Java interpreter. There are two different methodological interpreters; the Java

Virtual Machine (JVM) and, the most recent, Just-in-time compiler. The JVM converts on-the-fly the bytecodes by calling on a set of standard libraries, such as those provided by Java Standard Edition (SE) or Enterprise Edition (EE), while the just-in-time (JIT) compilers convert all of a Java program from JVM bytecode to native code as the program starts up. The JVM is a simulation of a processor which looks to java bytecodes as native as a processor looks to compiled code [21], [22], [23]. Both interpreters operation is illustrated in figure 2.4.

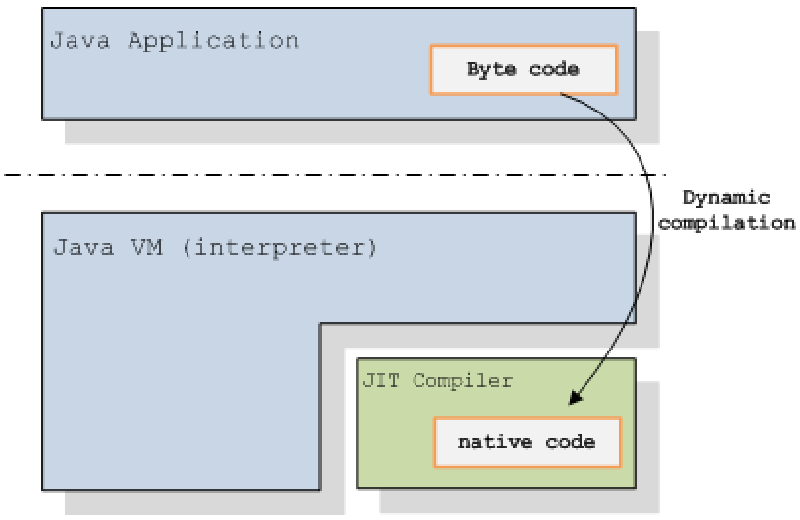

Figure 2.4: JVM and JIT Compiler [4]

2.4.2 Eclipse

The Eclipse Software Development Kit (SDK) is a free Java Development Tool which provides superior Java editing with on-the-fly validation and code assistance. It allows the use of VE (Visual Editor) open source editor through a plug-in system, being one of the most complete Integrated Development Environment for Java Developers. The Visual Editor was of extreme importance when concerning the Application Java Interface development. The choice of the Eclipse SDK version was made having in mind its stability with the most recent Serial Port Library (RXTX). The most used and reasoned version available at the time the work started was the Eclipse SDK 3.2, which with the Callisto optional features supported the Visual Editor [24].

2.5 Related Projects

This section will present some of the great variety of simulators and hardware implementations currently existent. Some of them have their own properties but the majority of them converge to the same effect, being complete and very useful.

A quite similar work is dated 2003, when Mark Holland, James Harris and Scott Hauck from the department of Electrical Engineering in University of Washington, Seattle, decided to implement a MIPS processor on an FPGA as a students learning tool. The processor implementation equally referenced the text book, *Computer Organization and Design* by David A. Patterson and John L. Hennessy. The implemented processor design executes each instruction in a single clock cycle and comprehends a restricted instruction set. As a complement, an assembler was developed allowing floating point operations to be mapped into the referred eight instructions set. The other major differences to this thesis work concern the targeted FPGA class, the design methodology, the not so portable debugging tool and the communication between the FPGA and the computer. The project was developed for the XESS-XSV board which comprehends a XILINX Virtex XCV300 FPGA and SRAMs used to map both processor memories. The debugging tool was developed using Visual Basic language and the mentioned communication is supported by a parallel port [25].

In Faculty of Engineering of University of Porto (FEUP), the teaching of computer architecture courses under the graduation of Informatics and Computing Engineering adopts a similar project (nanoMIPS). The project allows the students to study the MIPS 32bit architecture by the interaction with hardware implemented in FPGAs. It relates to a multi clock cycle per instruction processor which operates with a 50MHz frequency, executes a basic set of seven instructions (and, or, add, sub, slt, nor) and interfaces with the board peripherals. The data is written in the implemented Processor registers through the use of switches and can be consulted in the 7 segment displays.

One relevant project for this dissertation project was a Processor created for didactic purpose only, named *PEPE* (Processador Especial Para Ensino; *Special Processor For Teaching*, in English). PEPE's creators were professors from Instituto Superior Técnico, whose objectives were teaching the processors general characteristics through a didactic oriented, simpler and complete processor instead of using a commercial one whose finality was optimized performance only. Therefore, the developed processor is set up on a 16bit simple architecture with a JAVA based simulation software which allows not only programming and execution but also to simulate a computational system environment wrapping the processor. In the referred work, is also mentioned the possibility to install a software application in a commercial microcontroled based board to run the processor [9].

Another relevant and similar to the previous project is the P3 (Pequeno Processador Pedagógico - *Small Pedagogic Processor*, in English) and P4 (Pequeno Processador Pedagógico Pipelined). These processors were also developed by Professors from the Instituto Superior Técnico with the major differences of been implemented in hardware on a FPGA and having a second generation processor instruction language. Both the processors were developed with a 16 bit RISC architecture in order to explore all the pipeline potential in the P4 implementation. These processor projects also include their respective simulators and specific assembly language. The more relevant processor properties are the use of a stack and the interaction with external peripheries such as interrupt buttons, an LCD display, LEDs and 7 segment-displays [8].

A Flexible MIPS soft Processor was created in a Master Thesis by a Massachusetts Institute of Technology student whose purpose was to develop a flexible microprocessor architecture based in high-performance computing technologies. The flexibility of the processor architecture resides in the possibility to modify and extend the instruction set and the Processor´s functional units. As it is described in the student's thesis, the microprocessor was implemented on a FPGA board and the user can add and edit functional units by specifying the architecture parameters to suit the processor structure. The processor is compatible with the standard MIPS ISA, and supports pipelining configuration [26].

Especialização e Síntese de Processadores para Aplicação em Sistemas de Tempo-Real (Specialization and Synthesis of Processors for Real-Time Systems Applications, in English) is a Ph.D. thesis which addresses the development of architectures and synthesizable models of a deterministic, multitasking, pipelined processor and the respective coprocessor for real-time operating system support. Among other aspects, the author describes an implementation of a configurable 32bit MIPS based pipelined processor and provides a metric analysis support of FPGAs targetable implementations consisted of MIPS processors [27].

A paper, whose only access was to the abstract via internet, describes an educational system aimed at providing a course of computer architecture for graduate students called *ARCHO*. It is composed by both teacher and student oriented modules providing a system, named ARCAL, to support the study of theoretical concepts and a simulator, named APE, to support the laboratory activities [28].

One particular MIPS Processors Simulator is *WebMIPS*, which is accessible from the Web and has been used by the Faculty of Information Engineering in Siena, Italy in a computer architecture course. One advantage provided by this user friendly simulator, when comparing to the other existing ones, concerns the fact it is Web based not needing any installation process, being capable of uploading and assembling MIPS code provided by the user, as well as running a pipelined version in costume steps [29].

Most simulators as *spim* are very simple and do what is expected from a simulator; run MIPS32 assembly language programs, most of the time with the inclusion of a debugger. They do not handle exceptions and interrupts because the architecture has changed over time. spim is available for MAC OS, Linux and Windows but with specific execution files [30].

The last MIPS related project considered relevant is *MARS*. MIPS Assembler and Runtime Simulator was developed by Missouri State University and is a more advanced simulator that has an improved graphical interface and more features comparatively with most simulators. It allows registers and memory values access similar to a spreadsheet, floating point registers and cache performance analysis tool. This project has the particularity of been developed with the JAVA language [31].

DLX Gold project consisted of a 32bit pipelined DLX microprocessor capable of handling single precision FP operations. It makes use of the Class 1 architecture for FP operations, having an independent unit for addition, multiplication, and division. Additionally, the project considered dynamic scheduling to handle the parallel execution of instructions by the use of the speculative Tomasulo Algorithm. The algorithm provides the execution of sequential instructions that would normally be stalled due to certain execution dependencies. This project has the particularity of being designed for asic implementation [32].

FPGA Implementation of DLX Microprocessor with WISHBONE SoC Bus is a successful project case of the open source DLX architecture implementation. The objective of the work was to use the DLX microprocessor implemented with a Wishbone bus interface in a SoC (System-on-Chip) design, fostering design reuse by alleviating System-On-Chip integration problems. The main reason for using DLX concerns economical aspects. It allows saving money by avoiding the use of licensed IP core in the ASIC design. The target of the work is the use in a smart camera SoC [33].

Many 32bit MIPS simulators were not mentioned here, (for example, the *J-MIPS*, the *MIPSIM* and more recently *MIPSIM2* [34]) but the generality of present simulators features were mentioned in this section.

2.6 Concluding Remarks

Throughout this chapter some literature was suggested, it was provided a simple and useful description of the used architecture, as well as the system platforms involved during project development. At the end of the chapter, the projects which were considered relevant to the system development and its requirements identification, were presented. Next chapter will introduce the development methodology, as well as the system specifications.

Chapter 3

System Specifications

This chapter starts by presenting the procedure followed during system implementation, continues by describing the system abstraction model, details the Instruction Set Architecture used in this work and concludes by presenting the processor models, which are the focus of this work.

3.1 Development Methodology

The work was divided in several tasks, some of them executed simultaneously. Specifically, the Java interface followed a parallel implementation process with the processors and their wrapping modules, so program tests could be performed. The Graphic User Interface is one key component of the system by enabling the processor operation. The development of the GUI in java provides platform independence, since it runs on a JVM.

3.1.1 The Hardware

The hardware was developed, targeting a FPGA device using a top down design methodology. The modules were described using the hardware description language Verilog. The modules functional verification and simulations were performed using ModelSim whereas the synthesis, the mapping and the place and route were performed in Xilinx ISE 9.2i tool. A final verification phase was performed by executing a test program in the implemented processors.

It is relevant to mention that the functional verification achieved with ModelSim was done only for the processor core. The additional accessory models (event counters), whose implementation was relatively elementary, were tested during the final verification phase. Since the modules to implement were processor versions, it is reasonable to test all the executable instructions one by one. However, the tests were beyond a single test to each instruction and it was adopted the bubble sort algorithm to test a representative set of instructions. For this purpose, a set of random words was stored in the data memory in contiguous addresses. During the algorithm execution, the numbers were sequentially loaded to registers, compared two at a time and then orderly stored back to the

RAM by its content weight. The program created for this test comprehended ADDs, ADDIs, ORs, SLTs, LWs, SWs, BEQs, BNEQs, and JUMPs comprising a mix of 50% Arithmetic operations and 12,5% of loads, stores, Jumps and Branches. A complete instruction set is explained in section 3.3.

One possible implementation for the Bubble Sort algorithm can be described as:

Listing 3.1: Bubble sort algorithm

```
procedure bubbleSort( A : list of words present in RAM) :
 do
  swapped := false
   for( i = 0 to i = length(A) - 2)
    if A[i] > A[i+1] then
     swap( A[i], A[i+1] )
     swapped := true
    end if
   end for
 while swapped
end procedure
```

Algorithm implemented in MIPS 32bit ISA:

Listing 3.2: Bubble sort program

```
line0  ;  nop
line1  ;  and   s2, s1, s2        ; clear flag used to notify swaps
line2  ;  and   t2 ,s1, t2        ; clear RAM addr
line3  ;  or    t2, t3, t2        ; set RAM addr = 0x00000004
line4  ;  lw    t0, t2, 0x0000    ; load first word
line5  ;  lw    t1, t2, 0x0004    ; load second word
line6  ;  slt   t5, t0, t1        ; set t5 = 1 if t0 < t1
line7  ;  beq   t4, t5, 0x0007    ; branch to line15 if swap needed
line8  ;  add   t2, t3, t2        ; increment RAM addr by 0x00000004
line9  ;  add   s3, t4, s3        ; increment iteration index
line10 ;  bne   s4, s3, 0xfff9    ; branch to line4 if iteration not end
line11 ;  and   s3, s1, s3        ; clear iteration index
line12 ;  beq   s2, t4, 0xfff4    ; branch to line1 if swap performed
line13 ;  halt                    ; stop here
line14 ;  nop                     ; do nothing
line15 ;  sw    t1, t2, 0x0000    ; store second word into first word address
line16 ;  sw    t0, t2, 0x0004    ; store first word into second word address
line17 ;  or    s2, t4, s2        ; set swap flag = 1
line18 ;  jmp   8                 ; jump to line8
line19 ;  nop                     ; do nothing
line20 ;  nop                     ; do nothing
```

Listing 3.3: Registers initial content

```
t0  value is don't care              t5  value is don't care
t1  value is don't care              s1  value is 0 (used to clear)
t2  value is don't care              s2  value is 0 (swap flag)
t3  value is RAM first word addr     s3  value is 0 (counter)
t4  value is 1                       s4  value is 6 (iteration number - 1)
```

The communication with the Java Interface is assured by a low speed connection. A public IP Core was used for implementing the serial interface and a custom module was developed to handle the commands sent by the software interface.

3.1.2 Software

A graphical application was developed to provide interaction with the processor and the event counters modules. The purpose of the GUI is to allow the read/write access to the storage elements and control the program execution. This interface was developed in Java language through the use of Eclipse SDK, which required a Java Runtime Environment (JRE) to be executed. The Java source code was structured in different levels to make the visual classes independent from the lower level classes. Some extra functions were created to support data manipulation regarding the interface visual classes specifications, as can be seen in 4.2.1.

3.2 System Abstraction Model

In order to provide a better perspective of the global system, figure 3.1 illustrates an abstraction model of the system.

The system is constituted by two distinct platforms, the PC and the FPGA board, physically linked by a serial communication channel. The choice for the serial port as the communication link lies in the fact of being available in the targeted development board and being easy to use either in hardware and software. In the Software platform, both *Java GUI* and *serial port drivers* used by the Java Application run on a *Java Virtual Machine* allowing independence of the host computer operating system presented in section 2.4.1.

From the hardware point of view, the serial communication is handled by the *suart* block, which is the most adjacent hardware block to the serial port. The block *serial manager* is responsible for managing all the information flow between the inside blocks and the software application. A control block, *Control Manager*, was created to manage the processor execution over a flexible number of clocks. The necessary commands to enable the processor execution are transmitted from the *serial manager*. In section 4.1.5, it can be seen that the control block function is beyond steps/clocks management, as a result of the memories implementation into the block rams and some individual particularities imposed by each Processor version. The *Wrapper* includes the *processor* and the associated *event counters*, providing reading and writing of the processors

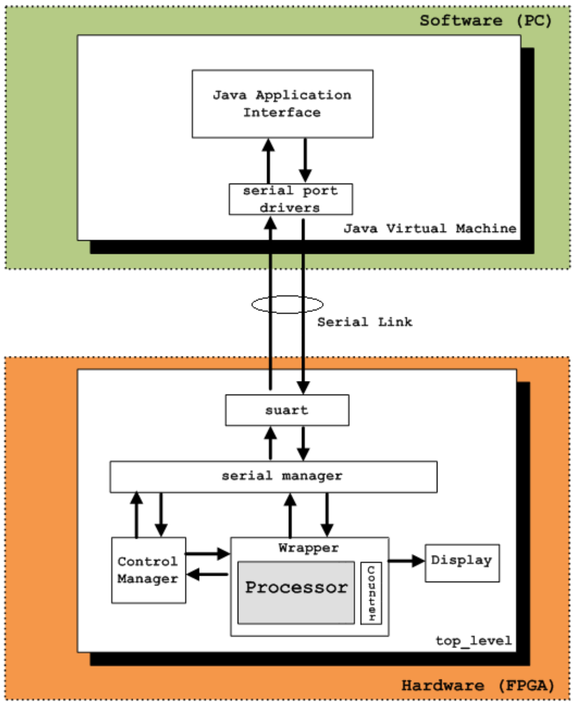

Figure 3.1: System Abstraction Model.

registers and memories, as well as access to the *event counters*. The errors in case of unrecognized opcode are handled in the error module by displaying error messages in the leds displays.

3.3 MIPS Instruction Set

MIPS 32bit is a RISC architecture with fixed length instructions. Although there are three different instruction types, some similarities can be found in the size of the instructions and in the number of the fields. Each type is assigned a distinct set of values for the opcode field so the instructions can be properly treated by the hardware. The instruction set selected for this work is the basic MIPS set present in the reference text book. It is composed by arithmetic and logic operations, shifts, compares, conditional and unconditional jumps and data transfers between memories and registers. The data transfers between memory and registers (loads and stores) are the particular instructions which allow the processor to access the external data memory.

The contemporary 32bit MIPS processors possess a vast instruction set with byte, half word, word and double word size memory access operations comprehending complex arithmetic operations and floating point format.

A detailed explanation about the three instruction types is discussed in tables 3.1, 3.2 and 3.3.

- the arithmetic-logical instructions (*R-Type*) The R-Type instruction consists of six instruction fields. The *opcode* field is common to all instruction types and defines the operation to be taken by the processor when executing the instruction. The *rs* and *rt* fields concern respectively to the first and second register source operand, while the *rd* field indicates the

opcode	rs	rt	rd	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

Table 3.1: R-Type Instruction, [2].

destination register to where the arithmetic-logical result will be stored. The *shamt* field holds the number of bits to perform in case of shift operation, which is defined in the *funct* field as in all arithmetic and logical operations.

- memory reference instructions such as load and store (*I-Type*)

opcode	rs	rt	constant or address
6 bit	5 bit	5 bit	16 bit

Table 3.2: I-Type Instruction, [2].

Since all instructions have the same length, three types of instructions had to be implemented in order to perform operations of different nature. The I-type instructions are frequently associated to data transfers between memory and registers. However, due to the necessity of having a field with the constant value, the arithmetic and logical operations with immediate values, such as *beq*, *bneq* and *addi*, also belong to this instruction type. Differently from the previous instruction type, in I-Type the field *rt* has two meanings. In the case in which the instruction concerns immediate instructions or a load instruction, the *rt* field specifies the destination register. In the case of a store, the *rt* field specifies the source register. The *rs* field is the complementary of the *rt* register, meaning that in case of a store it contains the destination memory base address and in case of immediate instructions or a load instruction it specifies the source memory base address. The I-Type targeted memory address is the sum of the address field plus the correspondent register content. The *branch if equal* and the *branch if not equal* are a special kind of I-Type since they perform flow control. These branch instructions are executed upon a met condition and the address to be branched to is the sum of the program counter with the constant field. This happens in order to provide a branch to any address within a 2^{32} range (memory size) since the 16 bit field *address* would only allow branchs within a 2^{16} range.

- jump instruction

opcode	address
6 bit	26 bit

Table 3.3: Jump Instruction, [2].

The jump instruction is the simplest one with no need for conditions verification. The processor when executing this instruction, simply updates the Program Counter with the 26 bit address specified in the respective field.

As can be seen in table 3.4, the instructions presented in this work are some of the basic initial MIPS instructions. This reduced instruction set is thought to be enough to provide the understanding of the mechanisms inherent to the MIPS architecture. The instruction set comprises arithmetic operations as *add*, *sub*, *addi* (add immediate) and *slt* (set less then), the logical operations *and*, *or*, *nor*, *sll* (shift left logical), *srl* (shif right logical), the conditional branch instructions *beq* (branch if equal), *bneq* (branch if not equal), the data transfers instructions *lw* (load word) and *sw* (store word) and finally the unconditional *jump* instruction. The *nop* instruction is not referenced in the instruction set although it is performed by the processor. The shift operations (sll and srl) were established as a one bit shift instead of the original variable bit shift. In order to distinguish these operations from the originals, the opcode is different, being respectively 1 and 62 (the bitwise not). The reasons that led to the implementation of the one bit shifting instructions, instead of the original variable bit shifts, refer to the absence of correspondent hardware description in the reference text book and to the fact that, from the ALU from of view, the one bit shifting instructions are faster to perform.

Name	**Format**	**opcode**	**rd**	**rt**	**rd**	**shamt**	**funct**	**address**
add	R-Type	0	reg	reg	reg	X	32	n.a.
sub	R-Type	0	reg	reg	reg	X	34	n.a.
and	R-Type	0	reg	reg	reg	X	36	n.a.
or	R-Type	0	reg	reg	reg	X	37	n.a.
nor	R-Type	0	reg	reg	reg	X	39	n.a.
slt	R-Type	0	reg	reg	reg	X	42	n.a.
sll	R-Type	0	reg	n.a.	reg	X	1	n.a.
srl	R-Type	0	reg	n.a.	reg	X	62	n.a.
addi	I-Type	8	reg	reg	n.a.	n.a.	n.a.	constant
lw	I-Type	35	reg	reg	n.a.	n.a.	n.a.	address
sw	I-Type	43	reg	reg	n.a.	n.a.	n.a.	address
beq	I-Type	4	reg	reg	n.a.	n.a.	n.a.	address
bneq	I-Type	5	reg	reg	n.a.	n.a.	n.a.	address
jump	J-Type	2	n.a.	n.a.	n.a.	n.a.	n.a.	address

Table 3.4: Adopted Instruction Set. *X* stands for "don't care", *n.a.* stands for not applicable. Based in MIPS Instruction Set [2]

3.4 Implemented Processor Models

In this section, the processor versions will be presented and discussed. There will be performed a simple and basic explanation since the objective of this work is not to rewrite the reference text

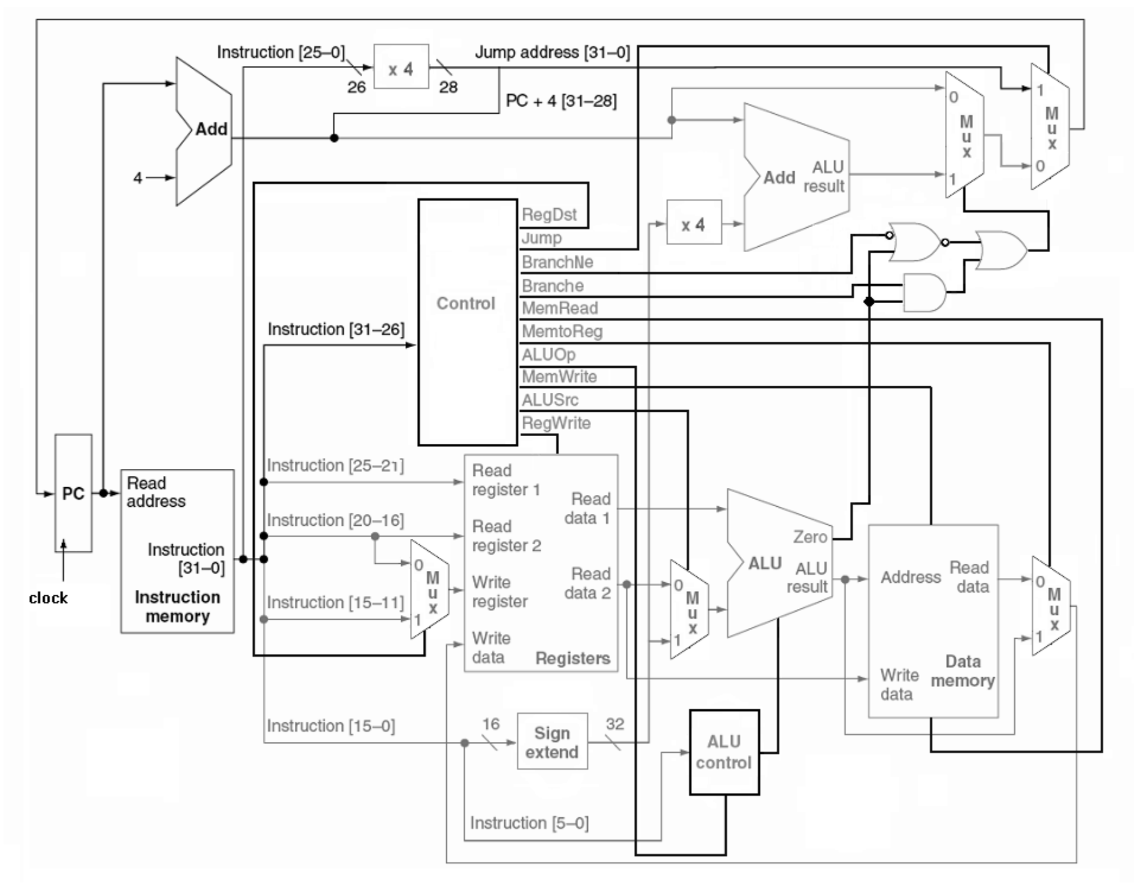

Figure 3.2: Unicycle, based in [2].

book. However, and in case of being the first contact of the reader with this architecture, a further reading of reference text book is advised. For the reader understanding interest, the Unicycle will be firstly approached expecting to be the basis for the following processors explanation.

3.4.1 Unicycle

The Unicycle stands for a single cycle per instruction and its design is basically composed of a Program Counter, a Register File, a Control Unit, a ALU and two memories. There are some supplementary arithmetic and logical blocks inherent to the processor operation and control flow. All the blocks are asynchronous except the PCounter. In each performed clock cycle, the PCounter is updated with the address of the instruction to be executed. In the same clock cycle, the instruction is executed and a new value is supplied to the PCounter input. This action allows the PCounter to be updated with the instruction expected to be executed in the following clock cycle. Figure 3.2 illustrates the Unicycle design.

From the left to the right, as the instruction execution takes place, it can be observed the PCounter, the Instruction Memory where the instructions are stored, the bank of the registers, the ALU whose role is to perform the arithmetic-logical calculations and, at the end of the datapath, the data Memory. In the previous chapter it was explained the three groups in which the instructions are classified. All these instructions have two steps in common, preceding the rest of the specific actions needed to complete each instruction execution. The first of the common steps consists of providing the PCounter content to the Instruction Memory address and consequently instruction fetch. The second step consists of reading the registers. All the instructions, besides *jump*, require a read from the register file and an ALU operation. For the present instruction set, the immediate type instruction *addi*, the load instruction *lw* and the arithmetic shifts *sll* and *srl* require only one register read whereas the rest of the instructions require reading two registers. The data transfer instructions require the ALU for address calculation (the ALU adds the register content with the address field content) whereas the arithmetic-logical instructions perform the correspondent function with the content present at the two ALU inputs.

Signal	**R-format**	**lw**	**sw**	**branche**	**branchNe**	**jump**
RegDest	1	0	x	x	x	x
ALUSrc	0	1	1	0	0	x
MemtoReg	0	1	x	x	x	x
RegWrite	1	1	0	0	0	x
MemRead	0	1	0	0	0	x
MemWrite	0	0	1	0	0	x
Branche	0	0	0	1	0	x
BranchNe	0	0	0	0	1	x
ALUOp1	1	0	0	0	0	x
ALUOp0	0	0	0	0	1	x
Jmp	0	0	0	0	0	1

Table 3.5: Control Unit table. *x* stands for “dont’t care”. Based in [2]

After the ALU stage, the load and store instructions need to access the memory to respectively read and write. The store instruction finishes the execution, whereas the load instruction needs additional write-back step, storing the loaded value into the register file. The arithmetic-logical instructions finish the same way as the load instruction, yet, without performing any memory reference. The branchs complete their execution after the equality test performed in the ALU, by selecting the next PCounter content. The *jump* instruction does not need any additional processing after the instruction decode, but a simple 2bit shift of the address field. This shift is commonly performed in all address calculation concerning the byte addressable memory present in the MIPS architecture. Differently from the branchs address calculation, where the address is obtained relatively to the present PCounter, the jumps perform a PCounter update with the value present in the instruction address field. The presented data flow is controlled by a module, which coordinates the processor modules. The Control Unit, a pure combinational block, decodes the instruction opcode and enables the necessary signals so the memories, the register file and necessary multiplexers can perform the requested operations. The control can be defined as:

- *RegDst*, which selects the register file address source. This control signal is necessary since the instruction set comprehends more than one instruction type. For example, in the R-Type instructions, the write register input most be supplied with the Instruction [15 - 11] bits, while in the I-Type load instruction, the correct bits to be handed are Instructions [20 - 16];

- *Jump*, which selects the destination address to be available at the PCounter input;
- *BranchNe* and *Branche*, along with the ALUZero signal, enable the address branch propagation;
- *MemRead*, enables data memory read;
- *MemtoReg*, selects the input data for the register file write data port;
- *ALUOp*, along with the instruction function field determines the operation to be performed by the ALU;
- *MemWrite*, enables data memory write;
- *ALUSrc*, defines the value to be provided to one of the ALU inputs. This ALU input requires two different sources due to the arithmetic operations performed with both R-Type and I-Type.
- *RegWrite*, enables the register file write;

The ALUOp and the instruction *funct* field are propagated into the ALU control block to be decoded, so the ALU control signals can be set. The encoding of the three control signals can be observed in table 3.5, and 3.6.

Instruction Operation	**ALUOp**	**function field**	**desired action**	**ALU control input**
lw	00	X	add	0010
sw	00	X	add	0010
addi	00	X	add	0010
branchE	01	X	subtract	0110
branchNe	01	X	subtract	0110
add	10	100000	add	0010
subtract	10	100010	subtract	0110
and	10	100100	and	0000
or	10	100101	or	0001
nor	10	100111	nor	1100
slt	10	101010	set on less than	0111
sll	10	000001	shift left	0011
srl	10	111110	shift right	0100

Table 3.6: ALU Control table. *X* stands for “don’t care”. Based in [2]

Some changes were performed to the reference text book design to support the execution of *BranchNe* instruction. These modifications can be observed at the top right area of the figure 3.2.

Instruction Class	**Instruction memory**	**Register read**	**ALU operation**	**Data memory**	**Register write**	**Total**
R-Type	200	50	100	0	50	400ps
Load word	200	50	100	200	50	600ps
Store word	200	50	100	0	0	550ps
Branch	200	50	100	0	0	350ps
Jump	200	0	0	0	0	200ps

Table 3.7: Instructions length, based in [2].

Two logic gates, a *nor* with one negative input and an *or*, were added to the original design in order to support the *BranchNe* operation. The output select signal of the multiplexer, which supplies the PCounter input, is set by the signal derived from the logic gates. When the ALU Zero value is "0" and the control *BranchNe* value is "1" the multiplexer outputs the branch calculated address.

3.4.2 Multicycle

The reason that led to evolve to a new architecture conception is based in the Unicycle timing inefficiency. Table 3.7 presents, hypothetical propagation delays required for each hardware block to perform the desired operation. In the Unicycle version, the clock period is determined by the time of the longest instruction, which is the load word and takes 600ps. The inefficiency problem can now be easily realized: the quicker instruction,(jump), should take only 200ps to be performed, but as a result of the hardware design, it must take the same time as the rest of the instructions, 600ps. The perfect approach would consider a variable clock so each instruction could be executed without overhead. If a mix of instructions (25% loads, 10% stores, 45% ALU instructions, 15% branches and 5% jumps) is considered, the average time per instruction would be [2]:

$$600 \times 25\% + 550 \times 10\% + 400 \times 45\% + 350 \times 15\% + 200 \times 5\% = 447.5ps$$

instead of the fixed 600ps provided by the Unicycle. However, such a design comprising a variable-speed clock would be extremely difficult to implement.

In the Multicycle processor version, the Unicycle inefficiency does not occur, since the instruction is broken into several shorter steps being executed one step per clock cycle. Each step is restricted to at most one ALU operation or one register access, or one memory access. The clock period of the Multicycle is therefore determined by the longest step, which in this instruction set takes 200ps. These steps will be addressed after the Multicycle architecture presentation so it can be easily understood by the reader.

The Multicycle Processor architecture is quite different from the Unicycle. As can be illustrated in figure 3.3 the control signals are not exactly the same, there is a single memory and, besides the ALU, there are no supplementary arithmetic blocks for address computing. Since there are no supplementary arithmetic blocks, the *ALU* must perform all the arithmetic operations as well as, the next address calculation. This, required the presence of a 4 way multiplexer at the *ALU* lower input. The use of less complex hardware to perform the same instruction set represents another advantage when comparing to the Unicycle. Due to the existence of only single memory it must be shared by instructions and data, requiring an multiplexer at the memory address input. Another difference is the existence of registers between combinational blocks. This register set comprehends the *Instruction register*, the *Memory data register*, both registers *A* and *B* and a register at the end of the ALU, namely *ALUOut*. The necessity of using these registers resides in the fact that each clock cycle can accommodate at most one functional operation, and therefore, the data produced by any of the functional units (memory, register file or ALU) must be temporary stored. In the upper part of the schematic it can be observed an *or* gate, an *and* gate and a multiplexer. The multiplexer does not appear in the reference text book design, since no *branchNe* instruction was described. As can be seen, when the *branchNe* signal is set, the multiplexer selects the *not ALUZero* signal, which will consequently enable the *PCWriteCond* control signal. Like in the Unicycle, the *PCounter* has three different sources. The output of the *ALU* is PC + 4 during fetch condition, the *ALUOut* which contains the address in a branch case and the lower 26 bits of the instruction register, shifted two bits to the left [1] and concatenated with the upper four bits of the incremented program counter (jump case).

The capability to execute instructions in a different number of clock cycles, as well as the ability to share functional units within the execution of a single instruction, requires more than temporary registers. It additionally requires a control unit performing as a finite state machine. The control unit implements eleven control signals which and can be set as [2]:

- *RegDst*, selects the register file address source;
- *RegWrite*, enables the register file write;
- *BranchNe*, and *Branche*, necessary for the branch execution;
- *MemRead*, enables data memory read;
- *MemtoReg*, selects the input data for the register file write data port;
- *ALUOp*, along with the instruction function field determine the operation to be executed by the ALU;
- *MemWrite*, enables data memory write;
- *ALUSrcA*, selects the source for the upper ALU input. If asserted selects the register A content, if deasserted selects PCounter content;
- *ALUSrcB*, selects the source for the lower ALU input. If 00_2 selects the register B value, if 01_2 selects the constant 4, if 10_2 selects the lower 16 bit of the instruction register (branch) and in the case of being 11_2, it selects the lower 16 bit shifted left by two bits (jump).

[1]in decimal system equals multiply by 4

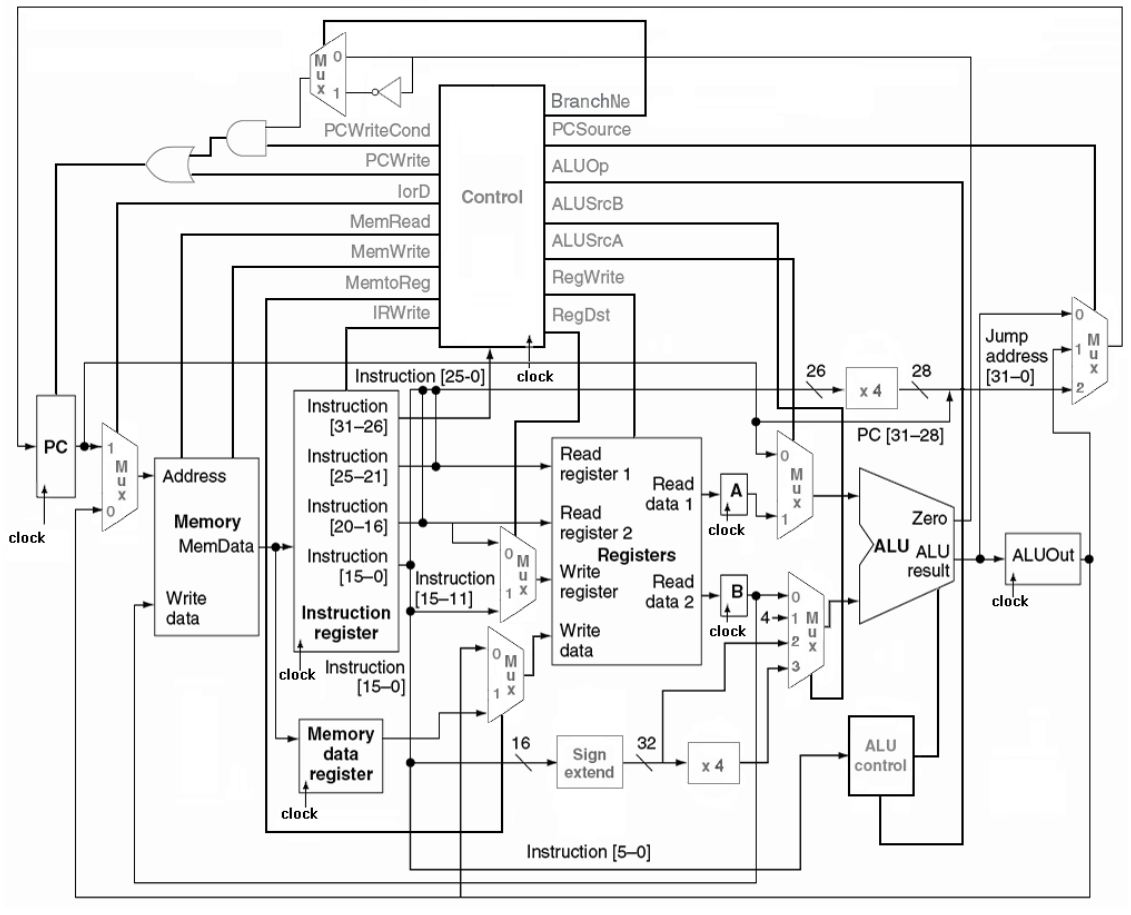

Figure 3.3: Multicycle, based in [2].

- *PCSource*, two bit signal which selects the PCounter source. If 00_2 the output of the ALU (PC+4) is selected, if 01_2 the source is the ALUOut register (branch target address) and if 10_2 the jump address is supplied to the PCounter.
- *IorD*, selects the source to the memory address input. When asserted, the ALUOut register is selected (in case of memory accesses), when deasserted the PCounter is selected (ordinary fetch).
- *IRWrite*, enables the instruction register write;
- *PCWrite*, enables PCounter write;
- *PCWriteCond*, along the result of the ALUZero, enables the PCounter write (branch);

The control signals value is determined by the current execution stage. The instructions execution requires up to five different steps, although the number of steps needed for each instruction is variable. The mentioned steps and the correspondent micro-instructions are detailed as [2]:

- Instruction fetch;

```
IR <= Memory[PC];
PC <= PC + 4;
```

- Instruction decode and register fetch;

```
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
ALUOut <= PC + (sign-extend (IR[15-0]) << 2);
```

- Execution, memory address computation, or branch completion;

```
ALUOut <= A + sign-extend (IR[15:0]);; memory reference

or

ALUOut <= A op B;; arithmetic-logical instruction

or

if(branche and (A == B))  PC <= ALUOut;
if(branchNe and (A != B))  PC <= ALUOut;; branchs

or

PC <= {PC [31:28], (IR[25:0],2'b00)};; jump
```

- Memory access or R-type instruction completion;

```
MDR <= Memory [ALUOut];; memory reference

or

Memory [ALUOut] <= B;; memory reference

or

Reg[IR[15:11]] <= ALUOut;; arithmetic-logical instruction
```

- Memory read completion;

```
Reg[IR[20:16]] <= MDR; load
```

The first two steps of the instructions execution are shared among all the instructions, while the rest of the steps are specific to each instruction. The finite state machine is illustrated in the figure 3.4.

The disadvantage of using a single-cycle design is significant. However, for the instruction set defined for this work, the Unicycle performs faster than the Multicycle.

3.4.3 Pipelined

After presenting both Unicycle and Multicycle the following processor version is a pipeline adaptation of the Unicycle version. The reason for adopting the Unicycle as the basis version to the pipelined version concerns the use of two memories (an instruction memory and a data memory) to avoid structural hazards (e.g. two unrelated write and read operations to be performed at the same memory address and at the same time). The hazard situations present in the pipelined version will be discussed later in this section. The principle of the pipeline design was already explained in section 2.2.1, however a brief presentation is included in this section.

Pipelining consists, like in Multicycle, in breaking the instructions into smaller stages, yet, with a major improvement: it allows the fetch of the following instruction before the current one is finished. This allows continuous feed of the stages, and therefore the execution of more than one instruction step at each clock cycle. This approach introduces some overhead to the program execution since the instructions are longer. Nevertheless, after some clock cycles the difference is notorious. Let the set of instructions be considered [2]:

```
add $t0, $t1, $s0;
lw $s5, $t4, 16;
sw $t5, $s4, 4;
```

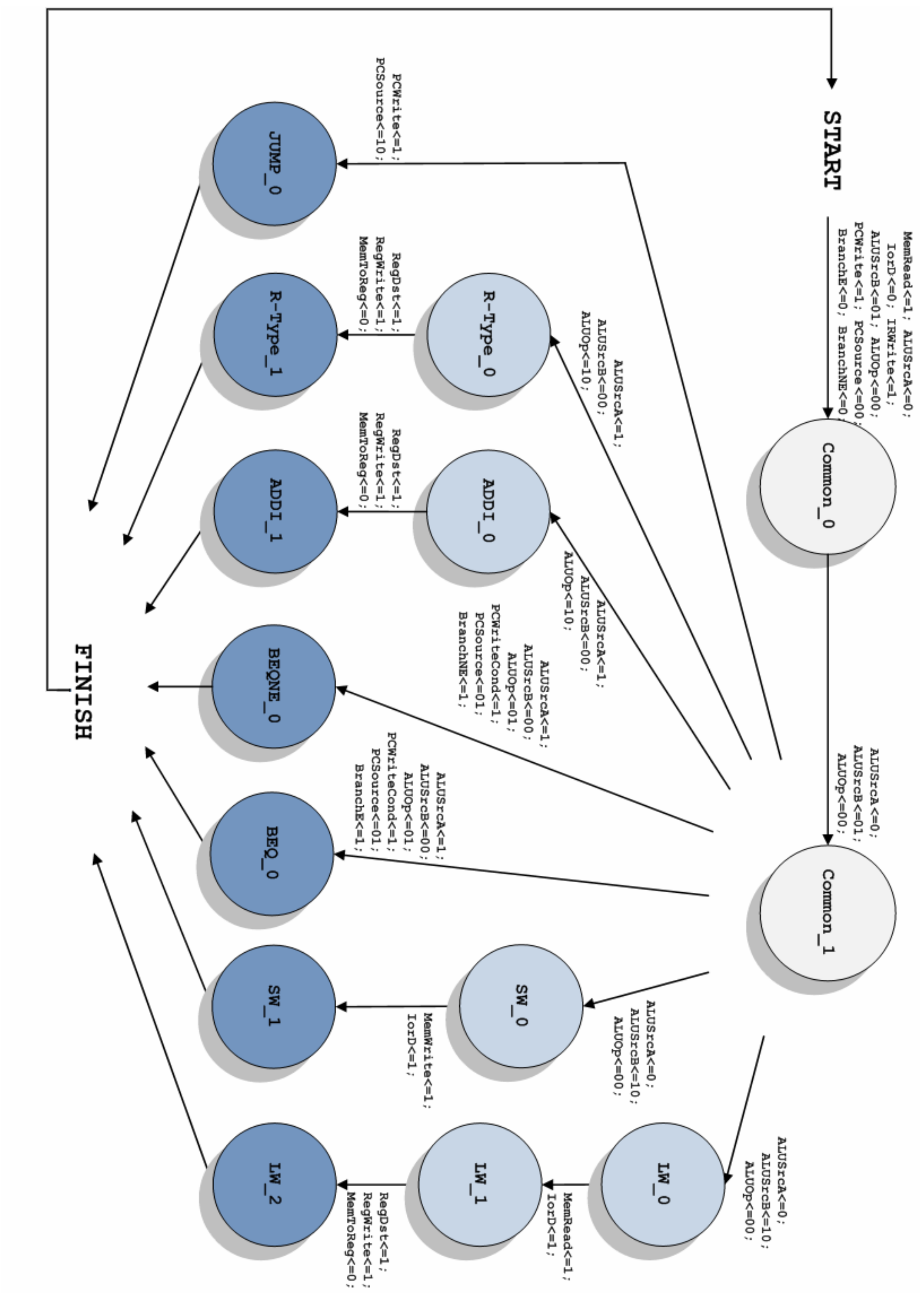

Figure 3.4: Multicycle FSM.

The figure 3.5 shows the time necessary for the instructions to be executed. After some clock cycles, the Pipelined design exhibits advantage to the Unicycle since the design stages are always active, improving the data throughput over time. This example is a simplistic approach, since there are some hazard situations that prevent the next instruction in the program flow from being sequentially executed.

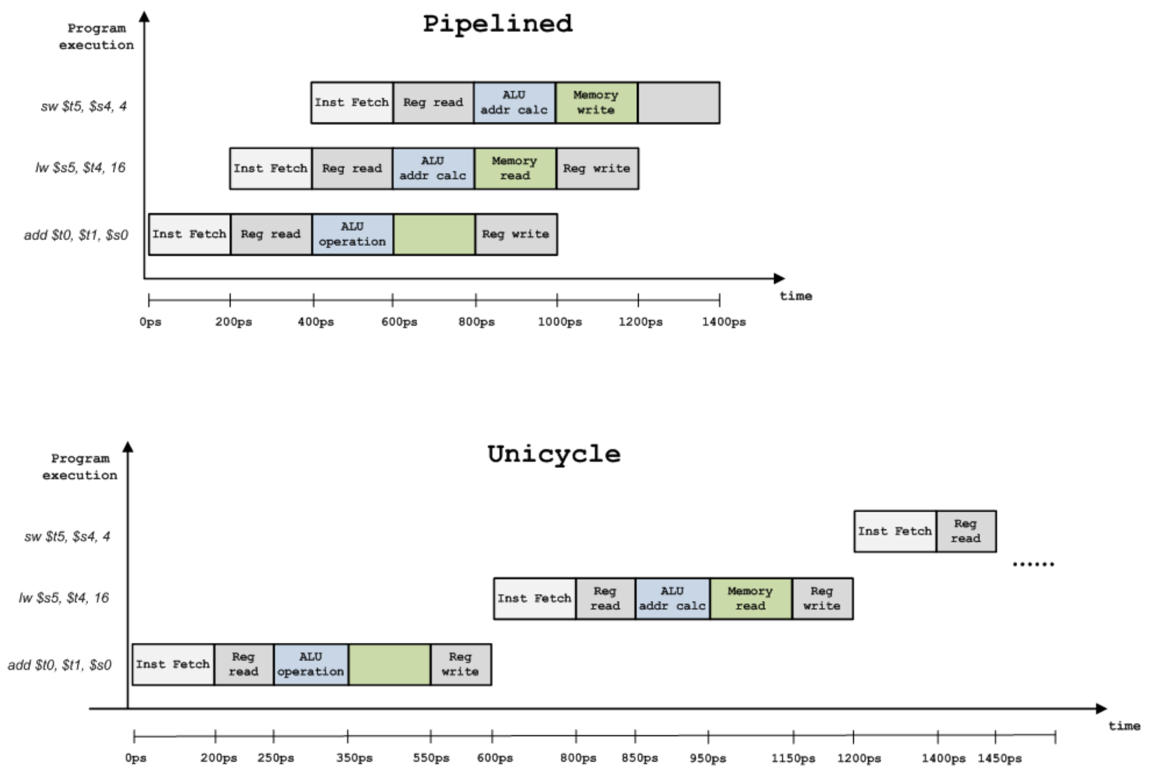

Figure 3.5: Pipeline versus Unicycle.

The pipeline design can be observed in figure 3.6. The datapath is broken into five different stages separated by registers. The function of the registers is to store the instruction data and control signals in each clock cycle, so the instruction stream can proceed through the pipeline. As is illustrated in figure 3.7, the control signals content is defined at the instruction decode stage and travels with the correspondent instruction data along the pipeline. The ALU control signals are the same as in the Unicycle version and therefore, mentioned in section 3.4.1. The processor control requires additional signals, such as the *PCSrc*, which selects the source of the PCounter, and the *ID/ID_flush* necessary for the stall insertions needed after a *lw*.

The registers name matches the bound between two adjacent stages (IF/ID, ID/EX, EX/MEM and MEM/WB). The stages are illustrated in figure 3.6 and can be defined as:

- Instruction fetch;
- Instruction decode and register read;
- Execute and address calculation;

- Memory access;
- Write back;

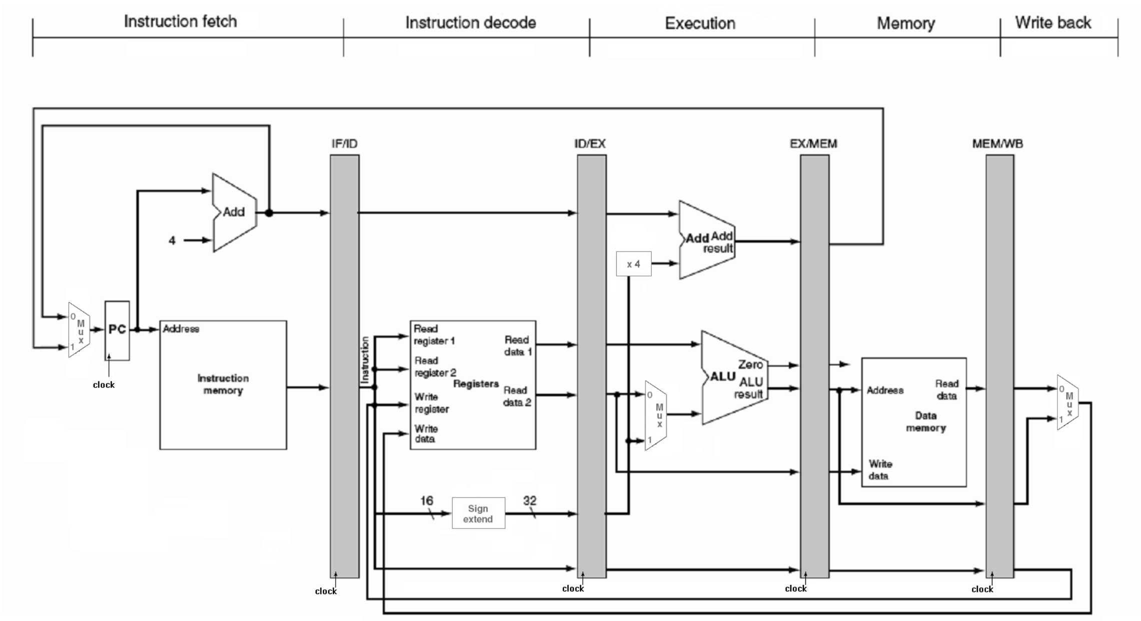

Figure 3.6: Pipeline basic design, based in [2].

The reference textbook addresses three different types of hazards, but only two types are referred in this text since the adopted design is not affected by structural hazards. Thus, the pipeline design resultant hazards are the data hazards and the control hazards. The data hazards arise when the result of an instruction is necessary as an operand in the following instruction. Consider the example of two instructions inserted in the processor datapath in figure 3.6. If the second instruction execution requires a read of a register which is updated by the first instruction, it can be seen that the second instruction is affected by a data hazard. More data hazards will be addressed as the processor is discussed. The second type of hazard that can happen in this version, occurs when a branch instruction is interpreted by the control. Referring to figure 3.6, suppose a branch instruction followed by any other type of instruction, consider as an example the instruction *add*. Since both instructions are consecutive, when the branch reaches the instruction decode stage, the *add* instruction is fetched. The branch operation depends on its on condition to be performed, which will only be verified in the next clock cycle when at the execution stage (remember the equality condition is verified at the ALU). Thus, when the *branch* instruction is at the execution stage, the *add* instruction is at instruction decode stage and, therefore already in the pipeline datapath. If the condition is positively verified the branch is taken only in the third clock cycle. This simple exercise serves the purpose of showing that two instructions are then incorrectly performed, if such circumstances are met.

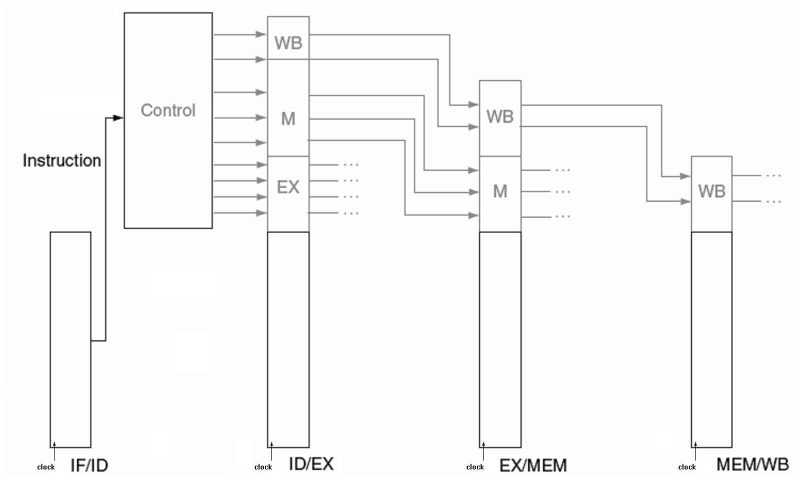

Figure 3.7: Control signals transition in pipeline, based in [2].

A possible solution to the hazard problems is the inclusion of forwarding mechanisms, as well as a hazard detection unit capable of managing the instruction stream flow. Still, some situations have no solution but to stall the processor for a clock cycle via the insertion of a “bubble” in the datapath. A bubble is one or more pipeline stages that do not contain useful instructions. The concept inherent to these mechanisms will be addressed in this section while the implementation details will be subject of section 4.1.8. The hazard situations can also be avoided if the program is set, so that no sequential dependencies are verified. However, this instructions rearrangement may increase the program execution time.

The forwarding mechanisms are managed by a unit, which detects data hazards and controls signals in order to override the situation. Consider the example of two adjacent *add* instructions with read after write data dependence. The first *add* instruction stores its result value in register $s0 and the second instruction adds the register $s0 with $s1. If no forwarding is performed, the second operation will not add the most recent value of $s0. The forwarding concept concerns the moment in which the registers are used to operate. If, in the moment when the second *add* instruction is at the execution stage, somehow the updated value of $s0 could be supplied to the ALU input instead of the read value from the register file, the hazard would not occur. In figure 3.8 such implementation can be observed when inspecting the ALU inputs. The ALU inputs are fed with the most recent value, which can be obtained from the instruction decode stage (normal case without hazards), from the memory stage or even from the WriteBack stage. In the considered example, the value to be supplied to the ALU input would come from the memory stage, since the dependable instruction is right before the current one. Yet, if the most updated value of $s0 was present at the write_back stage, that would be the value to be supplied to the ALU. All the forwarding mechanisms are based in this concept; to provide the dependable value at the time it is needed. Although not exhibit in figure 3.8, another case of forwarding implementation happens in the memory stage. Suppose two sequential instructions once more, but at this time memory referenced. A *lw* instruction that updates the register $t0 with a word from the data memory fol-

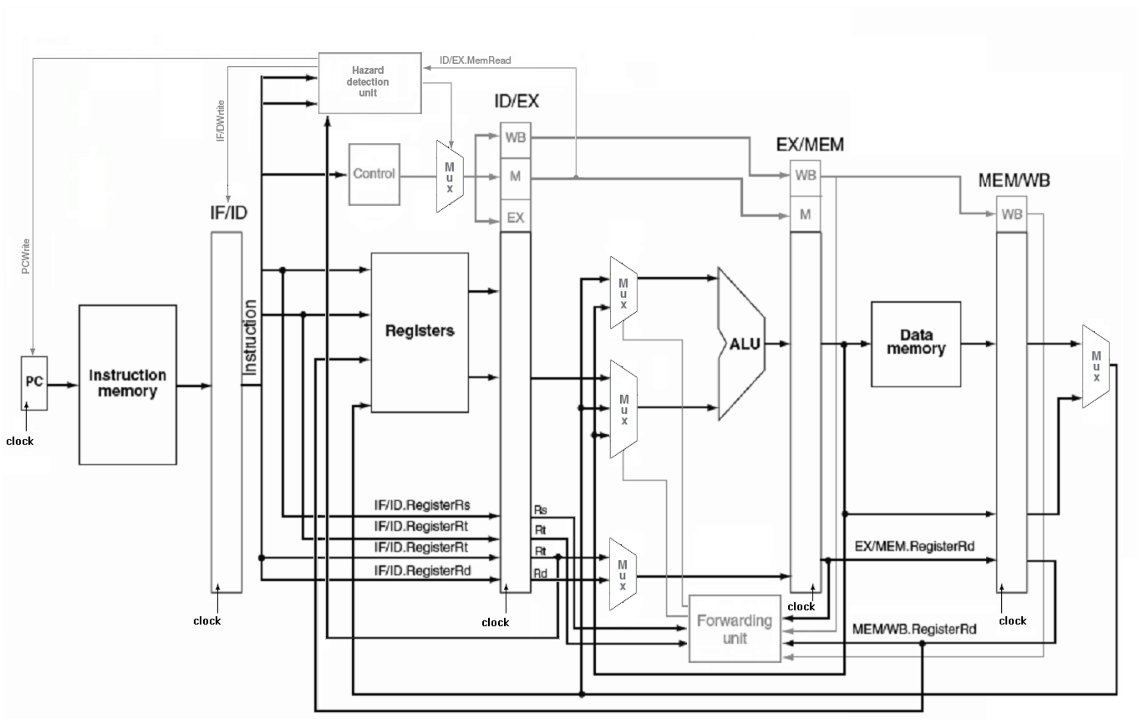

Figure 3.8: Pipeline version with forwarding mechanisms, based in [2]

lowed by a *sw* which stores the value of \$t0 in any address. As can be seen in the figure, at the time the register \$t0 is updated by the *lw* instruction, the *sw* instruction is present at the memory stage, writing in the data memory. With the same arrangement shown in the previously example, a multiplexer can be implemented in order to supply the memory input, and therefore provide the value present at the MEM/WB register. The *lw* instruction is a problematic instruction since it requires the processor to introduce a bubble in the instruction stream after its fetch (except if the following instruction is a *sw*). The reason for this can be explained if another example is considered. Suppose that after the *lw* an *add* instruction is fetched and one operand is the destination register of the *lw* instruction. The datapath comprehends one stage between the write_back stage and the execution stage. As a result of the design implementation, the mentioned *add* instruction would not be performed with the most recent value of the register, and must be stalled one clock cycle. The stall operation is triggered by a block called Hazard detection unit.

The Hazard detection unit is additionally used to prevent the branch hazards. In the beginning of this section, an example of the control hazards has been exposed. The idea used to overcome this type of hazard is to anticipate the *branch* [2] condition verification, as well as its computation. To enable that, an equality control test must be performed between the two register file outputs, and the blocks used to perform address calculation must be placed in the instruction decode stage rather than in the execution stage. The mentioned blocks are the 32bit adder, the sign-extend block and the $\times 4$ block. This procedure allows the *branch* consummation step to be executed early but, as a result of the hardware design, even though the *branch* test is anticipated, at the time

[2]same situation for *branch not equal*

the condition is verified, the consecutive instruction has already been fetched. Thus, a decision is taken and, if necessary, corrected after the condition verification. In the reference text book, some ideas are suggested by the author to deal with the decision of taking or not the *branch*. Among the presented suggestions, figure the branch prediction and the simple decision of always not taking the *branch*. The chosen method was to not take the branch and, in case the branch condition is confirmed as true, a stall is inserted by flushing the IF/ID register. As a result, a "bubble" will be pushed into the pipeline and delay the program execution by one clock cycle. This penalty may be considered acceptable when comparing with the absence of the hazard detection unit.

3.5 Event Counters

One the objectives of this work was to provide to the user the needed resources to detect and count some event occurrences. The objective underlying the event counters is to provide total acknowledge of the instructions execution process and data flow. There were considered three different versions of processors imposing different timings and conditions in the events evaluation. Still, the basic principle to each event counter remained unchanged. The event counters can be distinguished in five groups, accordingly to the action performed or to the storage element monitored:

- Clock counter
  This counter is incremented every time the main clock is triggered.

- Instruction Type counter
  Each instruction type (total of 8) has a correspondent counter which is incremented at the end of each instruction execution.

- Instruction Program accesses counter
  This counters group can be set to monitor up to four different addresses at the same time. They provide the sum of the reads performed in the instruction program.

- Memory accesses counter
  Similarly to the Instruction Program accesses counter, this counter provides the sum of the reads, but also the writes, performed to the memory address ranges. The addresses set is performed the same way as in the previous group.

- Registers accesses counter
  The registers accesses counter, counts the reads and writes performed in the most commonly used registers. The evaluated registers set is {t0, t1, t2, t3, t4, t5, t6, t7, s0, s1, s2, s3, s4, s5, s6, s7}.

All the counters are 8 bit wide [3] and are flushed in case of a reset command from the graphic user interface. The configurable counters are additionally flushed when the correspondent address field is set up.

[3] except clock counter which is 16 bit wide

3.6 Concluding Remarks

This chapter presented the system specification by defining the both used platforms; the board used for the hardware implementation and the programming language used for the Graphic User Interface development. The instruction set was also discussed, and a brief revision to the processor versions was performed in order to introduce some implementation aspects to be described in the next chapter.

Chapter 4

Implementation and Results

This chapter refers to the implementation performed, and discusses the choices made in the development of the hardware design and the software applications. Through the chapter, the operation mode of the processor versions and associated modules will be addressed. The FPGA resources utilization by the implementations is also matter of subject. At the end of the hardware section, a performance test will be presented regarding the implemented processors execution time. The reading of the previous chapter is considered important so the discussion and exposition of the implementation can be background supported.

4.1 Hardware

The hardware design comprises three main blocks. The function of the first block (*Serial Manager*), is to provide proper communication between the software application and the hardware. It receives data bytes from the *suart* module and assures the commands and data distribution between the software and hardware. The second block is a wrapper which includes the processor model, as well as the event counters. Each processor model requires a specific top-level module since they differ in design. The last block is *control manager*. It handles the processor operation modes by providing the option of running a flexible number of clocks, reset the program counter and enable the hazards resolution mode in the pipelined version. The *control manager* function also considers the enable signals necessary to the proper operation of the memories and the register file. Each top level design comprehends two clock signals. A master clock is used by the processor, whereas a two times faster auxiliary clock was created to trigger secondary blocks. The Processors were designed to achieve a maximum operation speed of 25 MHz (master clock). However, due to design limitations, the frequency of operation varies among the processor versions. The original design of the Processors comprise 8 bit width counters. Nevertheless, in order to allow the monitorization of longer execution programs, another implementation considering 32 bit width counters was also performed. The referred implementation can be consulted in section 4.1.11.

4.1.1 Serial Manager

The communication between the FPGA and the Software is assured by the *serial manager* (SM) block. The *serial manager* can be described as a Mealy finite state machine, responsible for updating the wrapper storage elements and transmitting the received commands to *control manager*. Figure A.1 presents the Serial Manager ports. Among other connections, it can be found the connection with the *suart* block. This connection comprehends two data transfer buses, namely *datain* and *dataout*, and three flagging bits, such as *load*, *enout* and *ready*. Both *load* and *ready* are signals from *suart* whereas *enout* is a SM output signal to *suart*. When data is provided at the SM bus input (the suart dataout), the *load* signal is asserted by *suart* and when *suart* is ready to receive data it asserts the *ready* signal. On the opposite way, the *enout* signal is asserted by the SM when the data is set at the output (the suart datain). Another group of ports concerns the transmission of commands to the *control manager* block. In figure 4.1, it can be seen the *CMInterrupt* signal and *CMRequest bus*. The *CMInterrup* flags a command to be transmitted through the *CMRequest* bus to the control manager. The rest of the SM ports concern the data transmission to, and from, the wrapper (e.g. writing in/reading from memories or registers).

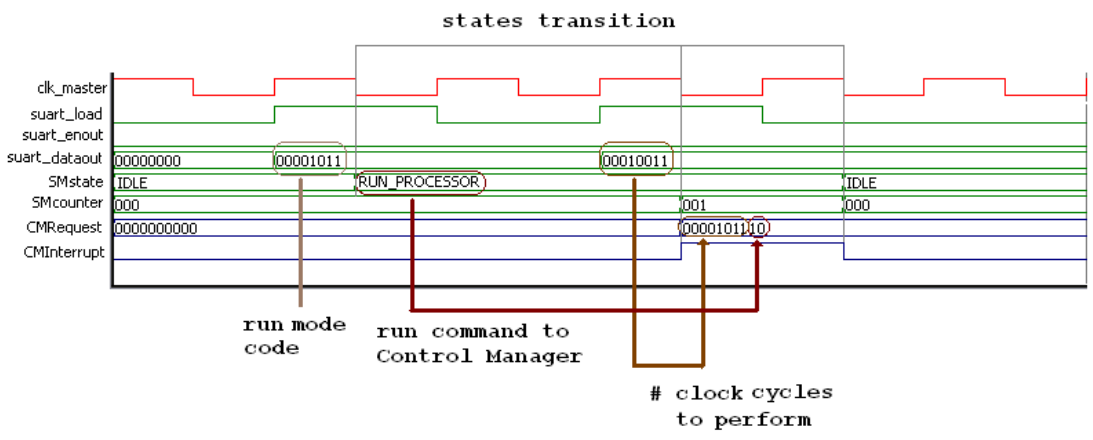

Figure 4.1: Serial Manager operation example.

The SM FSM is synchronously reseted and the state is defined at the negative transition of the master clock. When in the default state IDLE, if the *load* signal is asserted and an incoming command is identified, the machine changes its state to COMMAND EXECUTION, figure 4.2. Figure 4.1 illustrates an example of the data transfer protocol. In this shown example, the Serial Manager receives a command from the software application to run the processor. As can be seen, the data from the suart block is available at the positive transition of the clock. After receiving both command and the number of clock cycles to run, the Serial Manager interacts with the Control Manager in order to transmit the referred information.

Two FSM operation cases can be seen in figures 4.3 and 4.4. These examples respectively exemplify the instruction memory write word operation and the read of the instruction type counters. As can be seen in figure 4.3, once in the command execution state, the SM initiates another

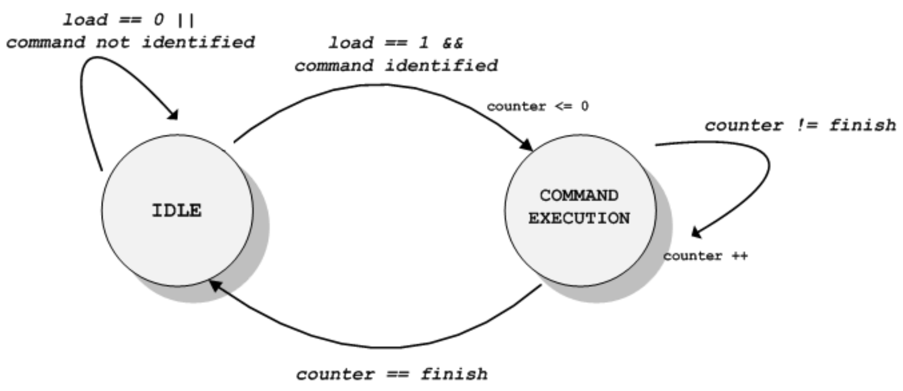

Figure 4.2: Finite State Machine of the Serial Manager block. The comments in italic refer to state transition conditions.

particular sub finite state machine. The sub state machine waits for the address of the instruction memory position to be written and for the respective content. Once both present, they are provided to the instruction memory inputs and the instruction memory enable signals are asserted. In the example of figure 4.4 the data flow process is similar. However, since the instruction types counter block comprises fourteen counters, (as many as the number of instructions present in the ISA) the data read from the instruction types counter is an iterative process. The figures 4.3 and 4.4 represent states transition examples, considering just a part of the necessary iterations to complete the command.

4.1.2 Memory

In section 2.3.1 it was stated that all the memories used for the processors implementation were mapped into the Spartan3 XC3S200 18Kbit block rams. This decision concerned the size chosen for the project implemented memories. The size chosen for the Instruction Memory and for data RAM (in both Unicycle and Pipeline) was respectively, 1024 per 32bit and 256 per 32bit. The chosen size for the Multicyle Memory was 1536 per 32bit. Both Unicycle and Pipelined total memory (Instruction Memory plus RAM) amounts 40 Kbit, being 133% of the device distributed RAM (30 Kbit), but only 18,5% of the block ram resources (which amounts 216 Kbit). The Instruction Memory implementation comprises two read ports and one write port. One of the read ports is part of the processor design, while both second read port and single write port refer to the communication with the serial manager, so the data can be read and write by the GUI. The data memory, in both Unicycle and Pipeline processors, and the Memory in the Multicycle processor include two read ports and two write ports. The difference to the Instruction Memory lies in the write function present on the processors data memory. The time instant in which the write ports are accessed differ accordingly to the block that writes to the memory. Specifically, if the serial manager writes to the memory at the positive transition of the clock, the processor writing occurs at the negative transition of the clock. Because block rams do not support asynchronous access,

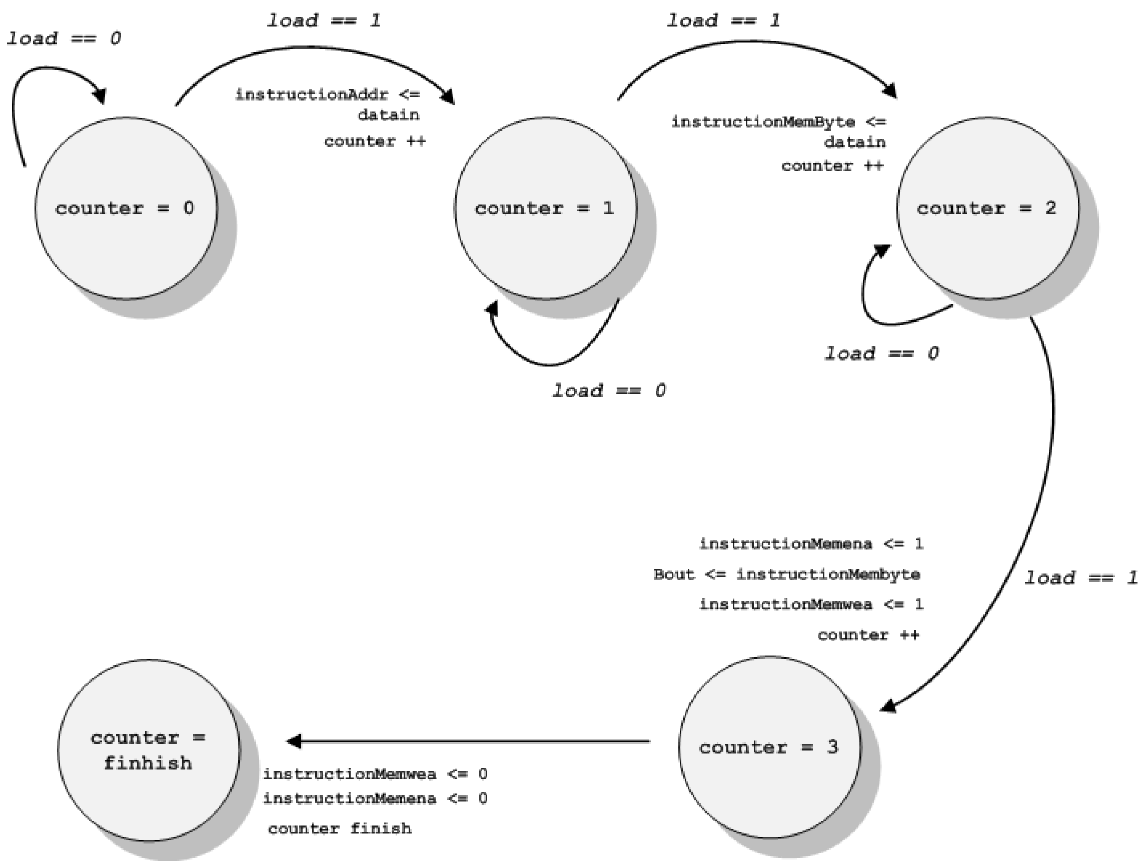

Figure 4.3: Command execution of the Instruction Memory write command. The values of "counter" refer to each command execution state and the comments in italic refer to state transition conditions.

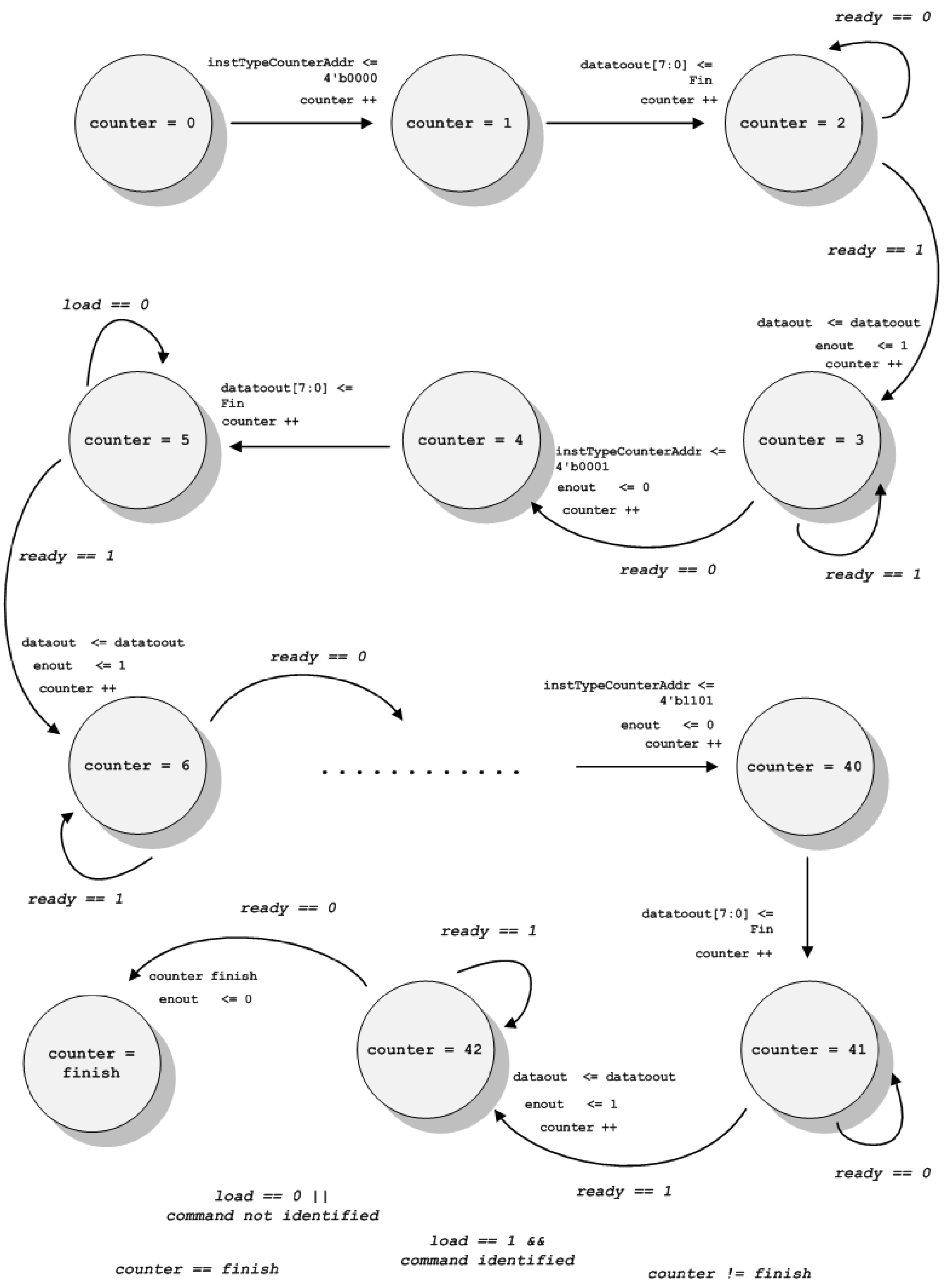

Figure 4.4: Command execution of the read of the Instructions Type counter. The values of "counter" refer to each command execution state and the comments in italic refer to state transition conditions.

the memories were implemented synchronously and requiring the use of the auxiliary clock, as well as extra control signals, all managed by the block *Control Manager*.

The implementation of the memories was performed using Xilinx Synthesis Technology (XST) templates in order to be easily adjusted if amendments were needed.

4.1.3 The Register File

The register file, as in the MIPS 32bit architecture, comprehends 32 registers, each 32 bit wide. Even though the registers considered in the referenced text book refer only to the set {t0, t1, t2, t3, t4, t5, t6, t7, s0, s1, s2, s3, s4, s5, s6, s7}, this work includes the whole 32 bit registers set. The first approach considered in the register file implementation included the three data ports, present in the reference text book design, plus two additional ports for the access by the software interface. However, since it was not necessary for the two different accesses to occur at the same time, the inputs of the register file were multiplexed, sharing a considerable amount of resources. The implemented register file module and the multiplexers necessary to the port sharing are illustrated in figure 4.5. As can be observed, a multiplexer controlled by the signal *ena* selects the inputs to the register file. The signals *wea*, *addra*, *dina* and *ena* are set by the module Serial Manager, whereas the rest of the signals are set by the processor blocks. A wrapper module 4.6 was created to include the register file and adjacent multiplexers to abstract this configuration from the processor.

4.1.4 Program Counter

The project implementation provides the possibility to update the Program Counter. This feature allows the continuous execution of non sequential instructions. A module placed between the processor design and the Program Counter inputs, is updated by the software application, allowing its content to be used in the following Program Counter fetch.

4.1.5 The Control Manager

The function of the control manager is to manage the signals required for enabling the program counter, the memories, the register file and the hazard resolution mechanisms (in the pipelined processor). Figure 4.7 illustrates the Control Manager ports.

The control signals are the same for all the processor versions and are set according to the processor version (the Multicycle implementation does not consider any instruction memory). The *Control Manager* receives commands from the Serial Manager, which are acknowledged at the negative transition of the master clock when the signal *CMInterrupt* is set. The control manager supports two different commands; the *reset* (01_2) and *run* (10_2). Both signals refer to the processor operation mode and are transmitted in the two least significant bits of *CMRequest*. The most significant bits of *CMRequest* correspond to the number of clocks to perform (*e.g.* 06_{16} equals 0110_2 corresponding to a RUN request of 2 clock cycles, because zero value is considered a countable cycle). The finite state machine of the control manager is illustrated in figure 4.8.

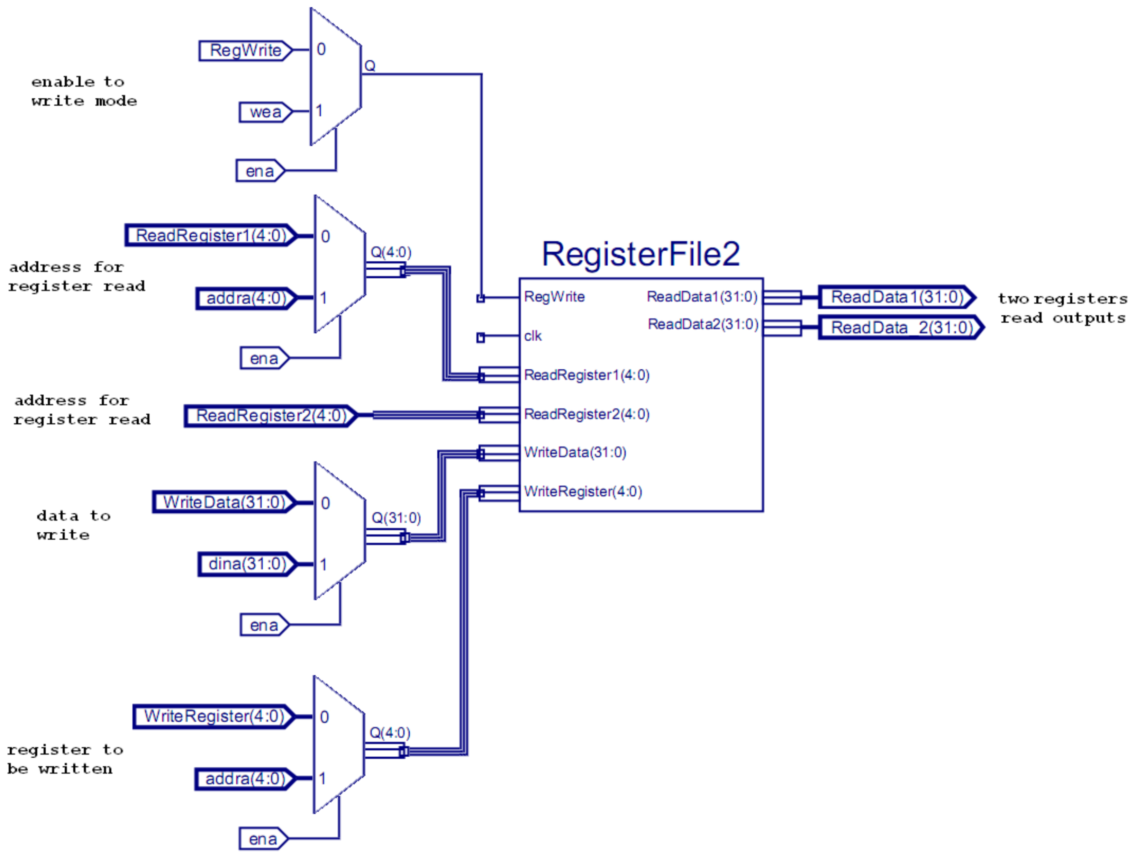

Figure 4.5: Register file and associated multiplexers.

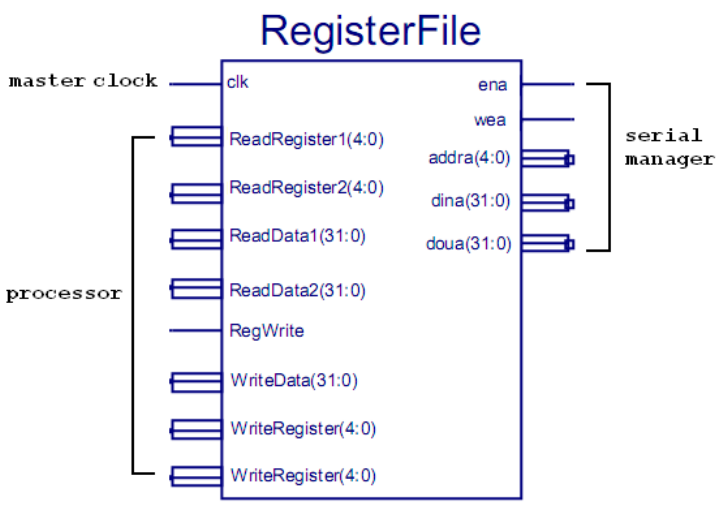

Figure 4.6: Register file wrapper which includes the contents of figure 4.5.

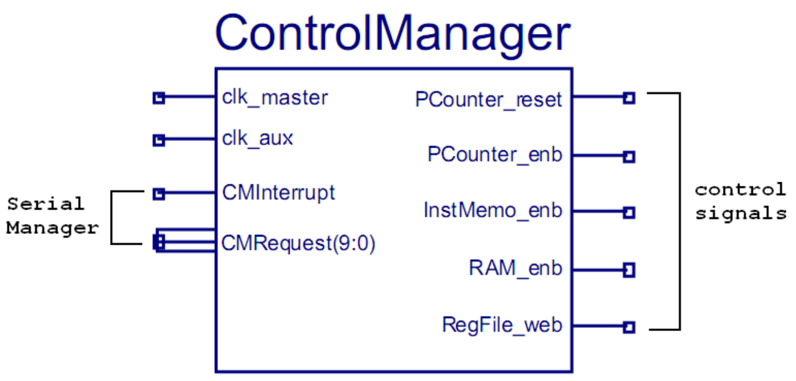

Figure 4.7: Control Manager ports.

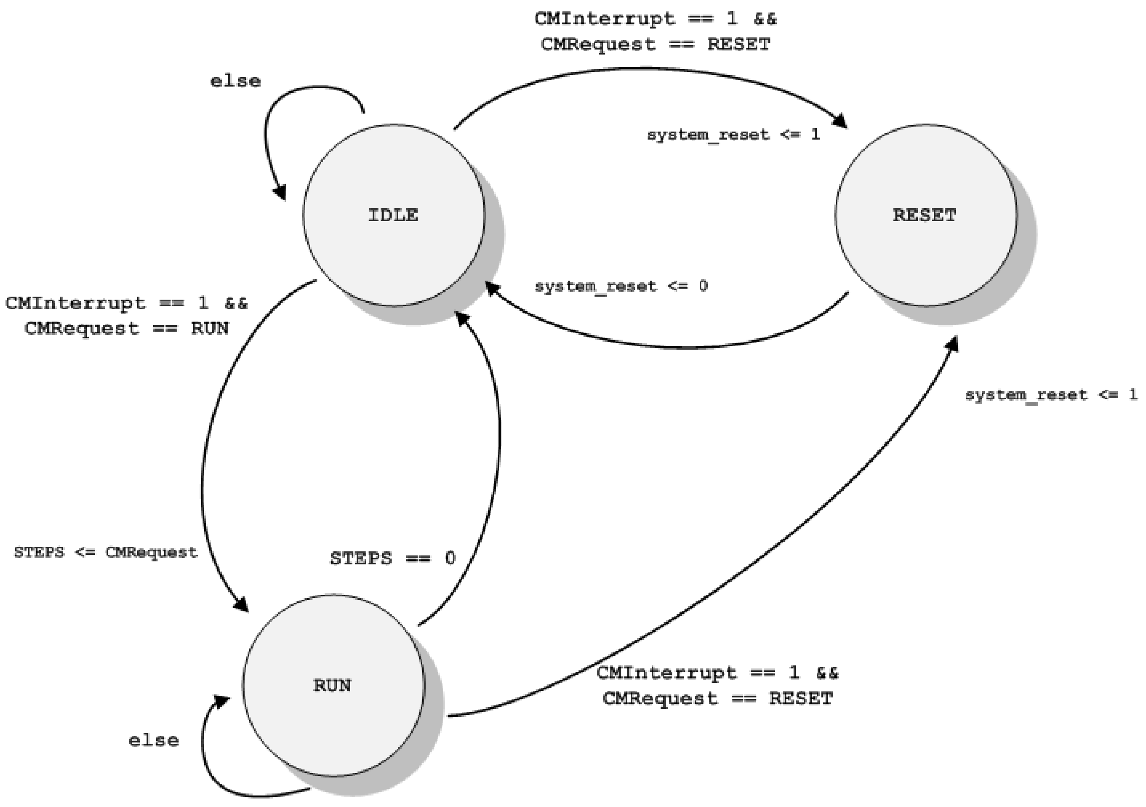
CMInterrupt == 1 && CMRequest == RESET
system_reset <= 1
else
IDLE
RESET
system_reset <= 0
CMInterrupt == 1 && CMRequest == RUN
system_reset <= 1
STEPS <= CMRequest
STEPS == 0
RUN
CMInterrupt == 1 && CMRequest == RESET
else

Figure 4.8: Control Manager finite state machine.

As can be observed, the state IDLE can be switched to state RUN and RESET. The machine is kept at the RUN state until the number of steps to perform is accomplished or if a RESET is requested. Once in the RESET state, the single possible following state is IDLE.

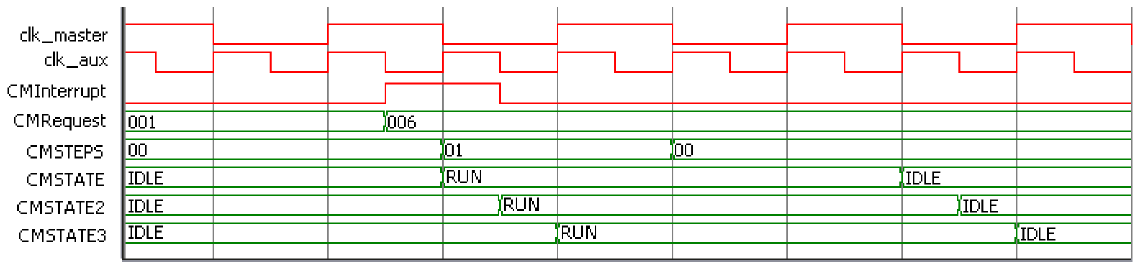

Figure 4.9: Control Manager auxiliary states.

The figure 4.9 illustrates a RUN command accomplishment by the control manager. The CMRequest is represented in hexadecimal codification as 006_{16}. The signal *state* is used to enable the set of the PCounter, whereas the auxiliary states *state2* and *state3* are necessary to manage the signals, which allow the memories and the register file to be accessed. The enable of the hazards resolution mechanisms is also performed in the *Control Manager* and consists in setting an enable signal present in both Hazard Detection Unit and Forwarding Unit. These signals will be properly approached in the respective processor implementation.

4.1.6 The Unicycle processor

The implemented processors are capable of running a flexible number of clock cycles for each received *run* command. In order to implement that feature, and since the memories are asynchronous, a set of signals were created to enable the memories, the register file and the program counter. These enable signals are controlled by the control manager and must be sequentially set, accordingly to the instruction flow through the processor. The ideal unicycle processor executes one instruction in a single clock cycle. However, this model considers that the memory access (program and data) is fully asynchronous, either for reading and writing. Because the used block RAMs only support synchronous access, the execution of an instruction can require up to 3 clock transitions: increment the program counter register, read the instruction from the program memory and read/write the data memory (for *lw* and *sw* instructions). To overcome this, an auxiliary clock was created that runs at twice the base processor clock frequency. Revisiting figure 3.2, if a *lw* instruction is executed, the necessary blocks to be triggered are, by this order: the program counter, the instruction memory, the ram and the register file. The mentioned blocks totalize four transitions against the two available in each clock cycle. The auxiliary clock is two times faster than the master clock which is considered the processor clock.

Figure 4.10 illustrates the time instants in which the PCounter, the memories and register file are triggered. The figure contemplates two clock cycles execution. The instant in which the first instruction is writing in the register file is the same instant in which the program counter of the second instruction is updated. The positive transition of the auxiliary clock between the instruction

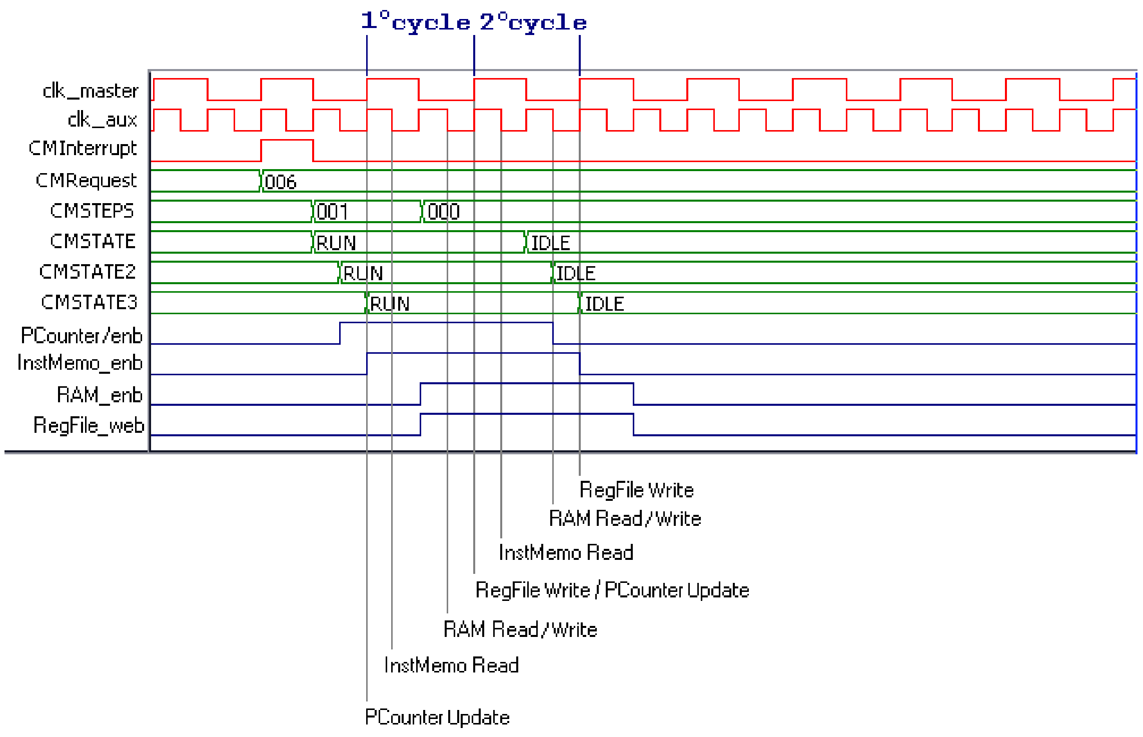

Figure 4.10: Unicycle control signals during the execution of two instructions.

memory read and the ram access is purposely not used to provide enough time to the instruction fetch, the register file read and the ALU operation.

Table 4.1, presents the resources used in the Unicycle processor implementation.

	Used	Available	Utilization
# Slices	1710	1920	89%
# Slices flip flop	1073	3840	27%
# Luts	2618	3840	68%
# BRams	3	12	25%
# DCM	2	4	50%
Max frequency	23 MHz		
Master clock frequency	10 MHz		
Aux clock frequency	20 MHz		

Table 4.1: Unicycle processor used resources and maximum frequency

The maximum frequency of the design is 23MHz and is defined by the critical path which relates to the ALU validation of the branch condition. The ALUZero signal, along with the control signals, selects the program counter source for the following instruction. As stated, the master clock period is twice the period of the auxiliary clock, therefore the frequency adopted for the processor execution was 10 MHz. The use of two DCMs concerns the impossibility to generate the two required clock frequencies in just one DCM.

4.1.7 The Multicycle processor

The Multicycle implementation is quite different from the Unicycle. Revisiting figure 3.3, the control block was implemented as a synchronous finite state machine to allow the execution of the instruction steps. The state machine follows the state machine transitions presented in 3.4. The states codification adopted for this finite state machine can be defined as:

```
Common0  =  0;                                     Addi0    =  7;
Common1  =  1;                                     Addi1    =  8;
RType0   =  2;                                     LW0      =  9;
RType1   =  3;                                     LW1      =  10;
JMP0     =  4;                                     LW2      =  11;
BEQ0     =  5;                                     SW0      =  12;
BNEQ0    =  6;                                     SW1      =  13;
```

The two states “Common0” and “Common1” are the two first steps performed by all the instructions whereas the rest of the steps are specific to each instruction. In the Multicycle design, the address calculation of the memory reference instruction is performed in the ALU. This characteristic is reflected in the behavior of the control signals managed by the control module.

Figure 4.11 illustrates the control signals during the execution of two instructions (*add* and *lw*). When the value of *Control_step* is zero and the processor is in run mode, the memory is read before the program counter is updated. The memory access is performed at the auxiliary clock negative transitions and relates to issues concerning the finite state machine implementation. This change to the original execution process sets no difference in the processor behavior since the micro-instructions are performed at the same step. The following step of the *add* execution decodes the instruction and fetches the source registers to register A and register B. In the following clock cycle, when *Control_step* equals 2, the *add* operation is performed in the ALU and the result is stored in the register ALUOut. The final step of this instruction concerns the write of the ALUOut content in the destination register, which is performed at the positive transition of the auxiliary clock. The registers A, B and ALUOut are updated at every positive transition of the master clock, if the processor is in *run* mode. However, by a matter of simplicity, the registers update shown in figure 4.11 relate only to the discussed example.

The two first steps of the *lw* instruction are common to the *add* instruction. The decode of the instruction and the registers fetch are performed when *Control_step* is 9, whereas the memory address to be fetched is calculated when *Control_step* equals 10. The instruction ends when the destination register is written at the positive transition of the auxiliary clock. Even though the program counter update, the accesses to the memory and the register file write are performed in more than one different *Control_step*, the trigger instant remains unchanged.

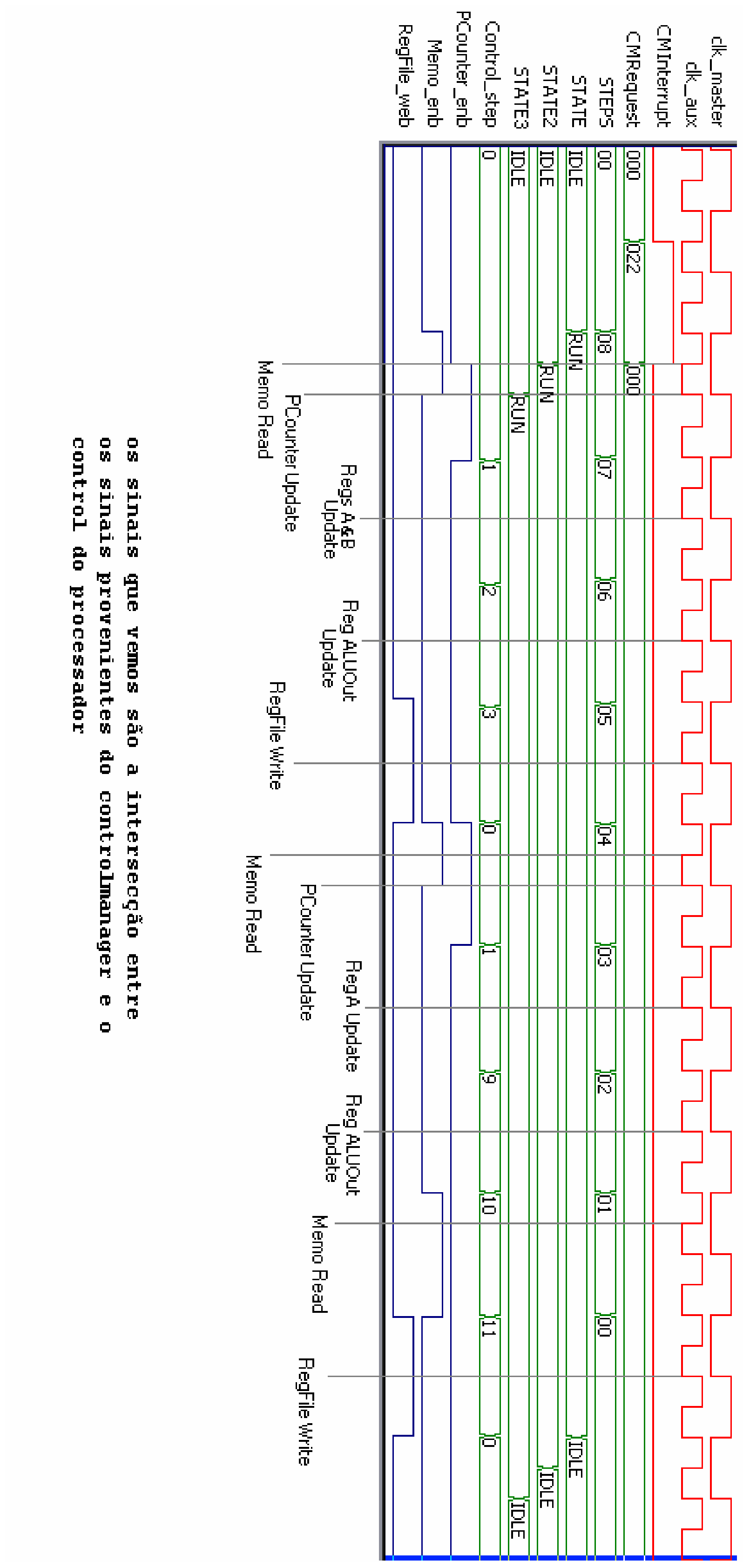

Figure 4.11: Muticycle control signals.

	Used	**Available**	**Utilization**
# Slices	1659	1920	86%
# Slices flip flop	1198	3840	31%
# Luts	3064	3840	79%
# BRams	5	12	41%
# DCM	1	4	25%
Max frequency	50 MHz		
Master clock frequency	25 MHz		
Aux clock frequency	50 MHz		

Table 4.2: Multicycle processor used resources and maximum frequency, post place and route.

Table 4.2 presents the FPGA resources used in the processor implementation. The main difference to the other implemented versions concerns the block ram utilization, relating to the memory size. In the Multicycle, the implemented memory is larger than the sum of the two memories (instruction memory and data memory) used in the Unicycle processor and in the Pipelined version processor. In the Multicycle implementation, all timing constraints were met and consequently the maximum frequency is 50 MHz. Since the auxiliary clock is two times faster than the master clock, the processor execution frequency is 25 MHz.

4.1.8 The Pipelined version processor

The pipeline processor was based on the pipeline design present in the reference text book. Although the reference text book does not address all the forwarding mechanisms necessary for a pipeline without hazard situations, this processor implementation evades all the possible hazards. This feature is available from the GUI where can be enabled by the user. Figure 4.12 shows the execution of 6 clock cycles, as many as, the necessary to perform two adjacent *add* instructions with read after write data dependence. When the processor is in RUN mode, the intermediate registers are enabled so the instruction can travel through the pipeline at the master clock cadence.

The first *add* instruction execution starts when the Program Counter is updated with the instruction address 004_{16}, whereas the second *add* starts at the following clock transition. The write operations performed in the register file, before the completion of the first *add* instruction, concerns the existence of initial values in the pipeline stages. Since the project is implemented in a FPGA, the default initial content of the registers was set as zero, which is recognized as a *nop*. The operation performed by the *nop* instruction is explained in section 3.3. As shown in the figure 4.12, the required register for the second *add* operation, is updated by the previous instruction. Due to the forwarding mechanisms, this value is propagated to the second *add* execution stage in order to avoid the hazard. If not, the resultant value of the second *add* would be 140_{16} instead of the correct 045_{16}.

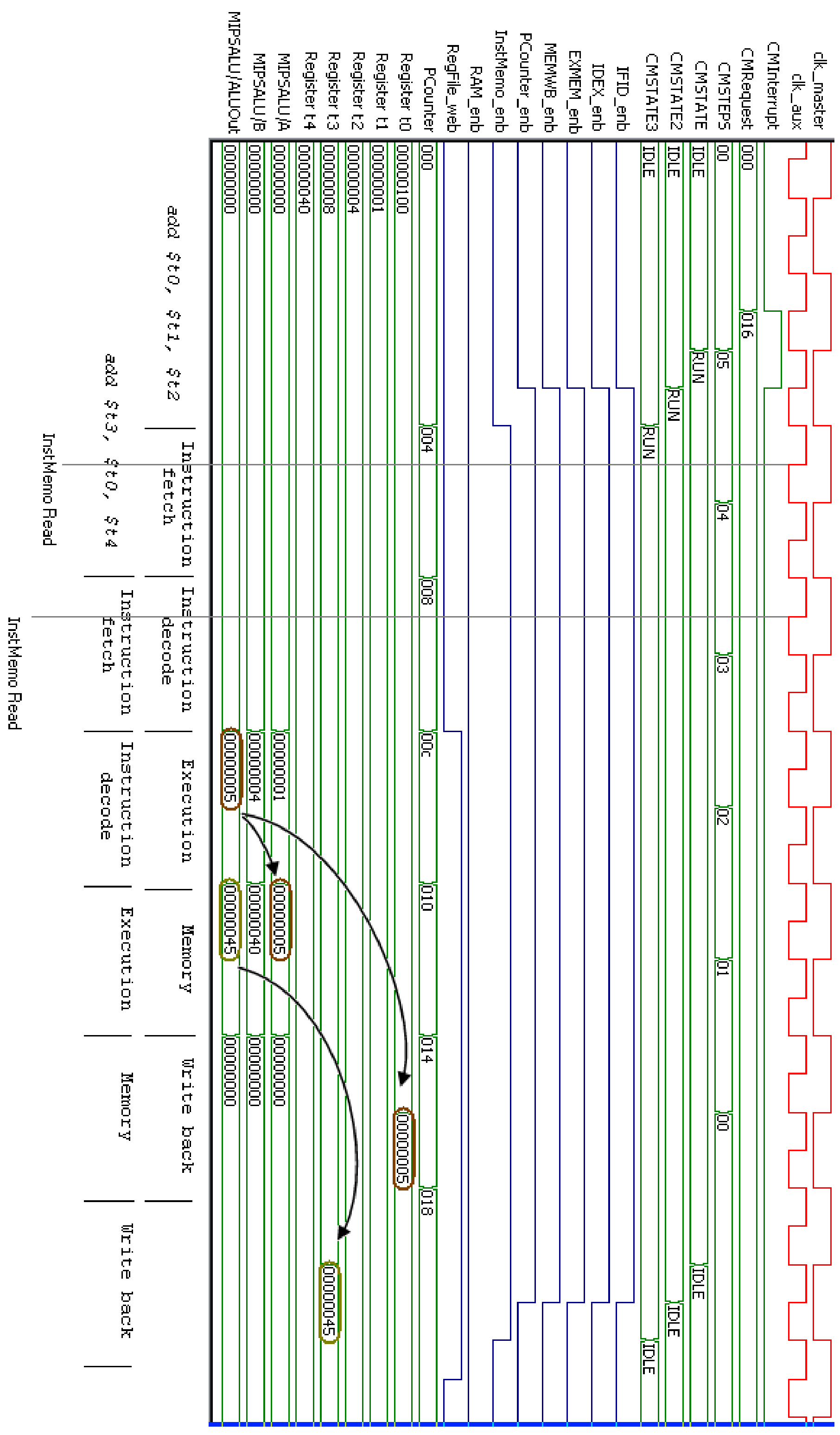

Figure 4.12: Pipeline control signals, A.

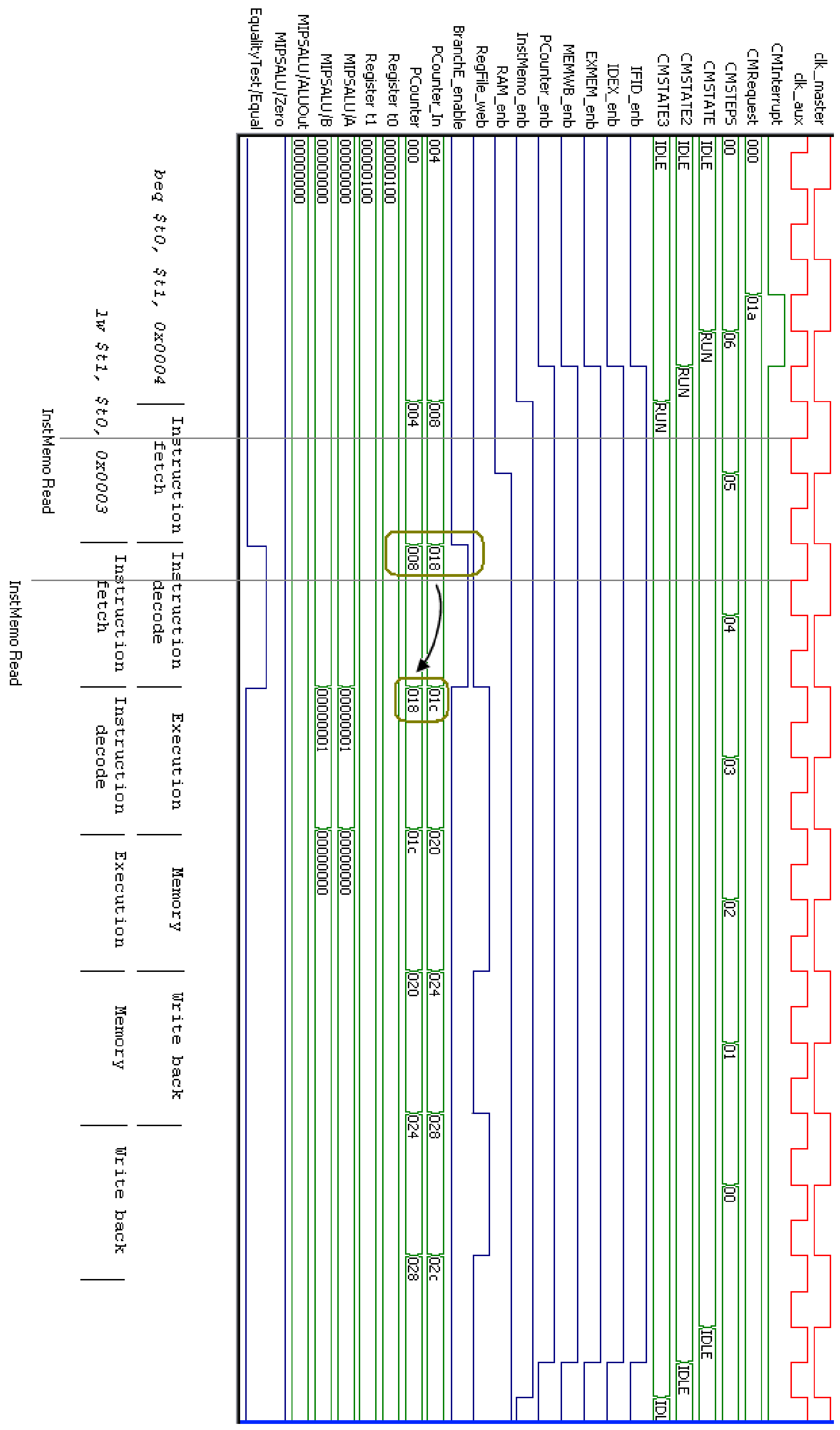

Figure 4.13: Pipeline control signals, B.

The option chosen to handle the branch decision assume that the branch instruction would not take place making the necessary corrections in case of wrong assumption. This decision was influenced by the lack of FPGA resources required to implement the hardware necessary for branch prediction. The control signals of the evading mechanisms concerning branch hazards are illustrated in figure 4.13. The *BranchE_enable* is set by the *logical and* between the branch signal from the control unit and the *EqualityTest_Equal* signal. The equality test block is done at the instruction decode stage and evaluates the equality of the two register file outputs. The signal *BranchE_enable* selects the branch address (018_{16}) to the Program Counter inputs, allowing the validated branch to be performed at the following master clock positive transition. If the branch condition was evaluated in the execution stage by the ALU, a two clock cycle delay would affect the normal program execution. Since the evaluation is performed at the instruction decode stage, the program execution is affected by only one delay (bubble insertion). Observe that at the execution stage of the *lw* instruction, the ALU is supplied with zeros (nop), and at the memory stage no data memory read is performed. As can be seen in the figure, the *lw* instruction is not executed since it is “replaced” by a *nop* instruction.

Figure 4.14 illustrates the implemented hazard evading mechanisms. The additional hazard avoidance situation implemented in this work, relates to the possibility of not stalling the processor when a *lw* instruction is followed by a *sw*. Concerning the reasons explained in section 3.4.3, the *lw* instruction demands for a “bubble” to be inserted right after its execution starts to avoid data hazards. However, since the critical stage of the *sw* instruction is the previously adjacent stage to the *lw* write_back stage, the content to be written can be propagated to the data memory inputs. This way, a cycle delay is avoided.

	Used	**Available**	**Utilization**
# Slices	1967	1920	102%
# Slices flip flop	1403	3840	36%
# Luts	3626	3840	94%
# BRams	3	12	25%
# DCM	1	4	25%
Max frequency	37 MHz		

Table 4.3: Pipeline processor used resources and maximum frequency, post synthesis.

Tables 4.3 and 4.4 present the FPGA resources used in the processor implementation. As can be seen in the post-synthesis results table, the design was too large for the targeted device. However, in the place and route process, the ISE tool managed to optimize the use of the resources so the design could be mapped into the FPGA. The performed area optimization effort is reflected in the design maximum frequency which decreased to 28 MHz. The critical path concerns the equality test performed in the instruction decode stage and its related signals. In that same stage, the instruction is fetched, the register file is read and the equality test is performed. A place and

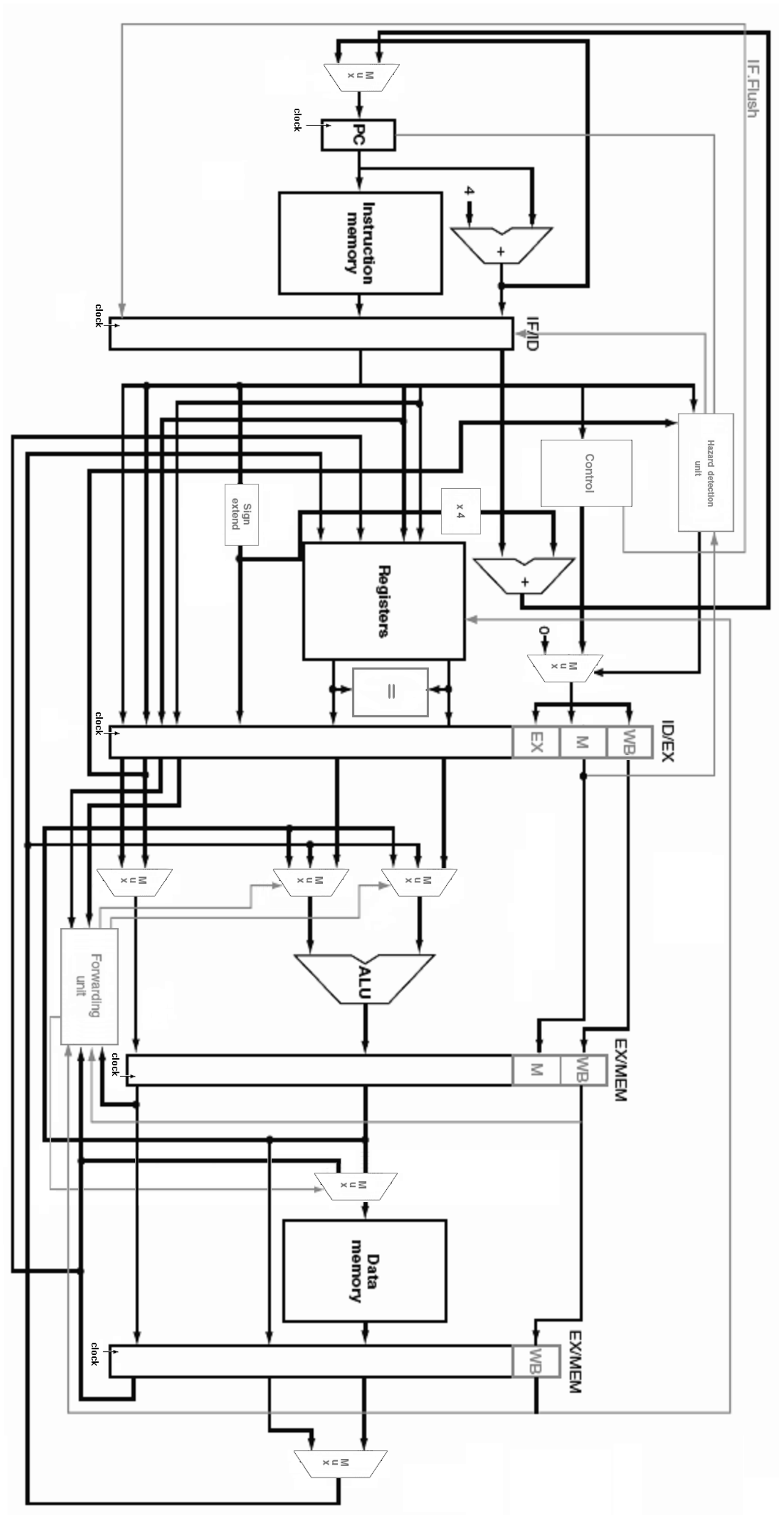

Figure 4.14: Pipeline hazard avoiding mechanisms, based in [2].

	Used	**Available**	**Utilization**
# Slices	1864	1920	97%
# Slices flip flop	1407	3840	36%
# Luts	2869	3840	74%
# BRams	3	12	25%
# DCM	1	4	25%
Max frequency	28 MHz		
Master clock frequency	12,5 MHz		
Aux clock frequency	25 MHz		

Table 4.4: Pipeline processor used resources and maximum frequency post place and route.

route was performed targeting a XC3S400 and the maximum frequency of the design kept the same estimated for the post synthesis model (37 MHz).

4.1.9 Event Counters

All the processor versions include event counters. The counters implementation was kept unchanged since the differences among the processor versions refer only to the evaluation instant of the referred action. The counters reset is set by the general reset signal present in the design. All the counters allow the transmission of data to the *Serial Manager* so the counters content can be downloaded by the GUI. The event counters can be thereby defined as:

- Clock cycles counter

- Instruction types counter
  The instruction types counter evaluates at the end of the instruction execution which instruction was performed and increments the respective counter. The duration time of the instructions varies accordingly to the processor design. Specifically, the evaluation time in the Unicycle is performed at the end of each clock cycle whereas the evaluation in both Multicycle and Pipeline alter accordingly to the executed instruction.

- Instructions accesses counter (reads)
  The referred counter is connected to both enable and address inputs of the instruction memory, so the reads performed to the Instruction Memory can be counted at the exact same moment they are accomplished. The counter allows up to four different addresses to be considered at the same time. The memory address to be monitored must be previously set so the counter can know which address interval to monitor.

- Ram accesses counter
  The RAM accesses counter is similar to the accesses counter used for the Instruction Memory, with the supplement of monitoring also the write actions performed to the data memory.

(Since the Ram implemented in the Multicycle processor is larger than the sum of the memories used in both Unicycle and Pipelined processors, the associated ram accesses counter allows up to eight different addresses to be monitored.)

- Register accesses counter
  The Registers accesses counter sums the reads and writes performed to a set of sixteen registers. The considered registers are included in the registers set used by the instructions. Similarly to the previous counters, the register file enable and write signals, as well as, the register address are evaluated by the access counter so the reads and writes can be properly identified and counted. The mentioned registers set is {t0, t1, t2, t3, t4, t5, t6, t7, s0, s1, s2, s3, s4, s5, s6, s7}.

4.1.10 Performance evaluation

In order to register the performance of the three processors, a test program implementing the bubble sort algorithm was executed. In listing 3.2, it can be seen the instructions resultant from the algorithm compilation into the existent instruction set. The program used for execution was the same for the three different processors, apart from a small difference in the Multicycle program. Since the Multicycle processor has only a shared memory, it required a different address memory approach. The register $t3$, containing the memory address of the first item to be sorted, had therefore different content. Instead of having the address 04_{16}, the register $t3$ was set with the address 54_{16}. Since on the original program the temporary register ($t2$), containing the memory address to be fetched, was incremented with the register $t3$, in the Multicycle program $t2$ had to be incremented by using the register, $t6$. However, these modifications do not change the program execution nor the number of instructions executed. Table 4.5 presents the adopted operation frequencies for the processor versions.

Processor	Operation frequency
Unicycle	10 MHz
Multicycle	25 MHz
Pipelined	12,5 MHz

Table 4.5: Processors frequency.

The processor performance values [1] are presented in table 4.6. For this instruction set, the Unicycle proved to be faster than the Multicycle processor, since the instruction set includes only simple instructions.

This result was expected, even though the Multicycle operation period is less than half of the Unicycle operation period. Furthermore, the Unicycle processor is 1,37 times faster than the

[1]the Pipeline values refer to a test using hazard evading mechanisms

Processor	**Number of clock cycles**	**Clock period**	**Program execution time**	**Clocks per instruction** (CPI)
Unicycle	348	100ns	35ms	1,00
Multicycle	1182	40ns	48ms	3,40
Pipelined	452	80ns	36ms	1,30

Table 4.6: Processors performance case of study.

Multicycle. The clock period of the Pipelined version was severely affected by the high utilization of the FPGA resources (97% of slices). As a result, the Pipelined version clock period is somewhat higher than the Unicycle version clock period. However, if no resource utilization constraint was applied to the Pipelined implementation (e.g. implementation in a Spartan3 XC3S400), even though with some delays caused by hazard evading techniques, it would operate with a frequency of 17,5 MHz [2] and would be 1,85 times faster than the Multicycle processor and also 1,35 times faster than the Unicycle Processor. Accordingly to the Pipeline concept, the processor execution efficiency is proportional to the increase of the number of instructions to execute.

4.1.11 Implementation with 32bit counters

Due to the need of more resources, this extra implementation was performed targeting the Spartan3 XC3S400. The changes performed to the Processors design, which considered the upgrade to 32 bit counters, resulted to additional modifications to the project. Since the serial port only transmits a byte at a time, the module *Serial Manager* was changed in order to assure the sending of the counters value, meaning that each counter requires three more transmission cycles. The resources utilization by the referred implementation is presented in tables 4.7, 4.8, 4.9.

	Used	**Available**	**Utilization**
# Slices	3280	3584	91%
# Slices flip flop	2660	7168	37%
# Luts	3761	7168	52%
# BRams	3	16	18%
# DCM	2	4	50%
Max frequency	23 MHz		
Master clock frequency	10 MHz		
Aux clock frequency	20 MHz		

Table 4.7: Unicycle processor used resources and maximum frequency with 32bit counters (XC3S400).

[2] the maximum frequency is 37 MHz, therefore the auxiliary clock would be 35 MHz and the master clock 17,5 MHz

	Used	**Available**	**Utilization**
# Slices	3554	3584	99%
# Slices flip flop	2859	7168	39%
# Luts	4429	7168	61%
# BRams	16	16	100%
# DCM	1	4	25%
Max frequency	50 MHz		
Master clock frequency	25 MHz		
Aux clock frequency	50 MHz		

Table 4.8: Multicycle processor used resources and maximum frequency with 32bit counters (XC3S400).

	Used	**Available**	**Utilization**
# Slices	3482	3584	97%
# Slices flip flop	2984	7168	41%
# Luts	4199	7168	58%
# BRams	3	16	18%
# DCM	2	4	25%
Max frequency	37 MHz		
Master clock frequency	17,5 MHz		
Aux clock frequency	35 MHz		

Table 4.9: Pipeline processor used resources and maximum frequency with 32bit counters (XC3S400).

As the previous tables present, the number of slice flip-flops and luts increased with the implementation of the 32 bit event counters. The block ram utilization in the Multicycle version totals 100% because some rom logic had to be mapped into block rams. This was performed so this processor version could be implemented in the targeted FPGA. When comparing to the other processors implementation, the Multicycle version represented a higher increase of the slice flip-flops. This happens due to the fact that the Multicycle memory access counter comprehends more registers than the sum of the counters responsible for monitoring the accesses to the two memories present in both Unicycle and Pipelined versions. Specifically, the Multicycle memory access counter comprehends 16 registers (one for the reads and another for the writes, both for each of the 8 possible addresses), while each of the other two processors versions (Unicycle and Pipelined) comprehends 12 registers, 4 registers related to the instruction memory counter and 8 registers related to the ram counter (one for the reads and another for the writes, per each register). The use of two DCMs in the Unicycle and in the Pipelined version relate to the impossibility to generate the two required clock frequencies in just one DCM.

4.2 Software

The developed software comprises two different programs. The most demanding in development effort was a Java application whose function is to provide user access to the processors storage elements, to the processor operation mode and to access the event counters. The Java program was developed under a hierarchical structure which can be segmented in three different layers according to its functional level. The second developed program was a dedicated assembler so the instructions present in the selected ISA could be translated into the processors machine code. The need to develop an assembler concerns the use of specific shift operations.

4.2.1 JAVA Graphic User Interface

The Java developed program includes several classes to support an intuitive graphic interface. All the developed classes can be grouped in three different layers:

- **User Interface** — which is the highest layer in the Java hierarchy and concerns only the visual classes, which are directly accessible to the user.
- **Middle-level** — the middle level layer comprising the functions needed to handle and manipulate data types. The data retrieved from the interface is available in string type requiring dedicated treatment in order to be sent to the FPGA.
- **Serial Port Drivers** — the lowest layer comprises all the classes that directly interact with the serial port created object. The serial port object requires setting operation parameters, connection establishment and stream management. These classes were based in [35].

The major aim of using Java language concerned the intention of providing portability among different operating systems. In order to provide such portability, the Java program holds a feature which supports the choice of three different operative systems (Linux, MacOS and Windows).

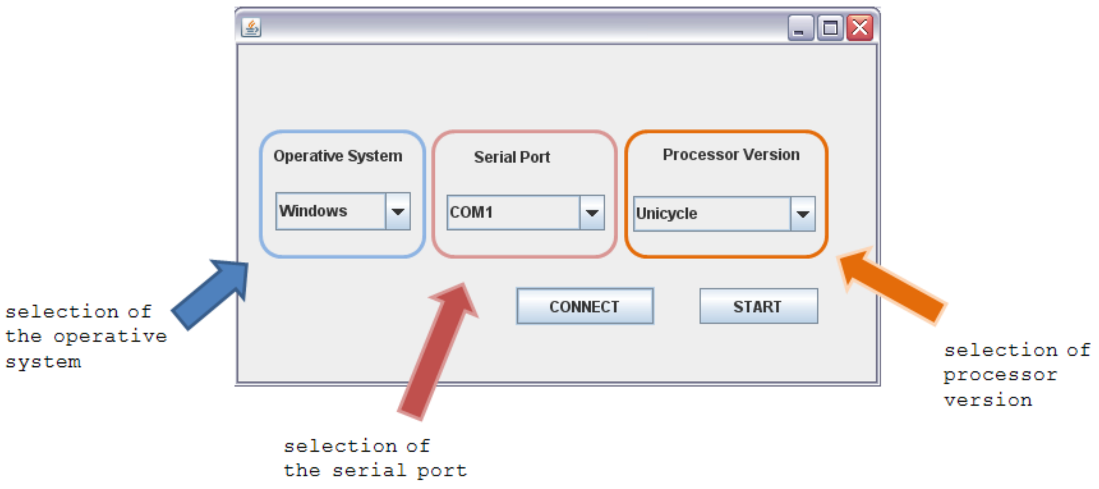
Operative System
Windows
Serial Port
COM1
Processor Version
Unicycle
CONNECT
START
selection of the operative system
selection of the serial port
selection of processor version

Figure 4.15: Setup Interface.

Figure 4.15 shows the setup window used to initiate the Processor insterface. The user, once in program execution, needs to select the appropriated operative system, the respective serial port and initiate the desired processor version interface. Only one processor version interface is allowed at a time. However, to change among the existent interfaces, there is no need to restart the software since it allows to choose the processor version in the setup window. The interface buttons operation is very simple. Each time a button is pressed, a middle-level class is called to process the required fields, collect the related data and send a command to the FPGA using low-level classes. If a reply is solicited, the middle-level class initiates a thread addressing the text box to be updated. This way the text box will be immediately updated after the reply data be available. Figure 4.16 exemplifies the Unicycle processor version.

Although they do not differ much from each other, three different interfaces were created. The differences among the developed interfaces respect to the different processor versions, e.g. the Unicycle and Pipeline interfaces have two memory fields (Instruction Memory and Data Memory) whereas the Multicycle interface has only one memory field (Memory). The Pipeline interface also includes a singular Special Mode field in which the hazard evading mechanisms can be activated. The visual interfaces can be seen in more detail in figures A.2(Unicycle), A.3(Multicycle) and A.4 (Pipeline). The functional fields which allow the write and read of data into the storage elements (memories and registers) are situated in the bottom part of the interface, under the processor schematic. The counters access fields are located in the top of the interface in both corners. The top left corner counters require address specification so the counters can be set with the address to be monitored. Each time an address is set, the correspondent counter is cleaned, so the count can start from the zero value. In the bottom left corner of the interface, the user can set the next program counter value and consult both next and current content of the program counter. The number of clocks to run, as well as the processor reset can be set at the bottom right corner.

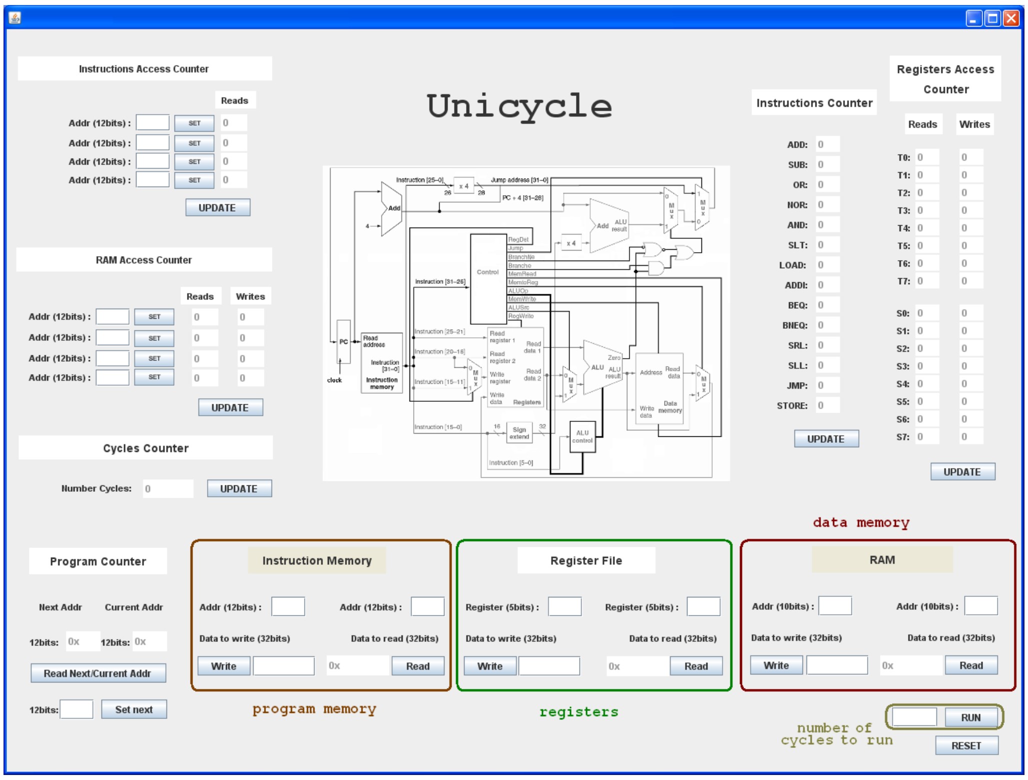

Figure 4.16: Unicycle interface.

4.2.2 MIPS Language Assembler

The implementation test and debug processes combined with the utility of a language assembler in the didactic package led to the inclusion of a MIPS 32bit assembler. For this purpose it was used the open source Configurable Assembler Program (CASPR) which was modified to suit the MIPS Language specifications [36]. The original program source files were developed in C language and were compiled under the cygwin environment. The executable file requires an input file with the **.asm* extension and provides a **.rom* output file with the instructions translated to machine binary code presented in hexadecimal codification. Some changes were introduced in the source code, so the program would output 32 bit instruction words instead of the original 64 bit words.

The assembler also required changes in tiny1.cfg, which is the file containing the correspondence between instructions and their opcodes. These changes were performed so the instructions present in the input file **.asm* could be interpreted.

4.3 Concluding Remarks

This chapter presented the implementation and results of all the implemented modules and the developed software. The implemented hardware comprises the processor models and associated blocks, as well as the event counters. The processors were implemented accordingly to the maximum frequency allowed by each processor design. Although the objective was to implement the processors in a XC3S200, an extra implementation with 32 bit width counters was performed targeting the XC3S400 and the results were presented. The developed software includes the GUI to support the interaction with the referred processors and an instruction set dedicated assembler.

Chapter 5

Conclusions and Future Work

5.1 Objectives Accomplishment

This work required an enlightening study of the MIPS 32bit computer architecture, in order to proceed with a proper design implementation. The adopted text book for such exercise was the "Computer Organization and Design - the hardware/software interface", of John Hennessy and David Patterson, which is a work of reference in this field of study. The result of this study and related projects is addressed in chapter 2.

The main objectives for this work have been accomplished. The three processor versions proposed in section 1.2 were implemented to fit a low cost FPGA board. Some instructions present in the MIPS architecture but not described in the text book were added to the instruction set. The implementation carried out allows the user to understand how the operations are performed by consulting the storage elements and the events counters. The hardware limitations related to the clock frequency are present, providing to the user the contact with real hardware implementation. A software interface was designed to be appealing to the user, providing full access and control of the processor operation. The developed assembler simplifies the manual assembly language translation for simple programs, suitable for academic purposes. The support for the cache memory was not accomplished due to the lack of time.

The major obstacles in the project development regard the memories implementation in block rams, which concerning their synchronous operation mode, required a different approach from the referenced in the text book that considers asynchronous memories. As expected, the project bottleneck relates to the FPGA available resources, consequently affecting the working frequency of the processors. All the design was performed at RTL level, optimized for the Spartan3 Starter Kit board, which is a low cost development board commonly used in major universities. The Java based interface provides portability among operative systems.

This work is expected to be an effective framework to learning computer architectures and a low-cost educational tool concerning a world-leading architecture in embedded applications.

5.2 Future Work

The implemented processors instruction set comprehends basic instructions present in the reference text book. The instruction set should be extended with more instructions (simple or even complex), in order to provide more cases of study of operations execution.

The processors do not comply overflow exceptions neither interruption attendance. The overflow signalization is not critical but allows acknowledgment of unexpected data errors. The support for interrupt processing would be a considerable improvement towards the processor integration with peripherals.

In the specific case of the Pipelined processor implementation, the reason for the absence of a more complex branch prediction mechanism concerns the lack of resources in the FPGA. It would be interesting from the user point of view to understand such prediction operations, and therefore the mechanism implementation is proposed as future work.

The GUI is quite appealing to use, yet, the addition of some features would represent less setup work to the user. If the storage elements content could be consulted from a text file, the user would have a global perception of the processor current state. The option to program the FPGA from the GUI, even with the possibility of doing it by using bitstreams stored in the board flash memory, should be considered. In the GUI, an easy to implement feature would be the option to update, at once, all the text fields.

An additional development could be the implementation of the support module for the cache memory. The cache memory is used in every MIPS CPUs in order to reduce the average time to access memory and, therefore, would provide to the user the possibility to acknowledge the process inherent to both instructions execution and data accesses at a higher cadence. However this will only be important when using external memories with long access time.

Another interesting feature would be the software upgrade in order to support remote access to the FPGA board. This way, the user could be in contact with a real implemented processor without needing to be at the same place. As an example, the Java application could be upgraded to be built-in a web page, and therefore, be accessed by a student from home.

Appendix A

Figures

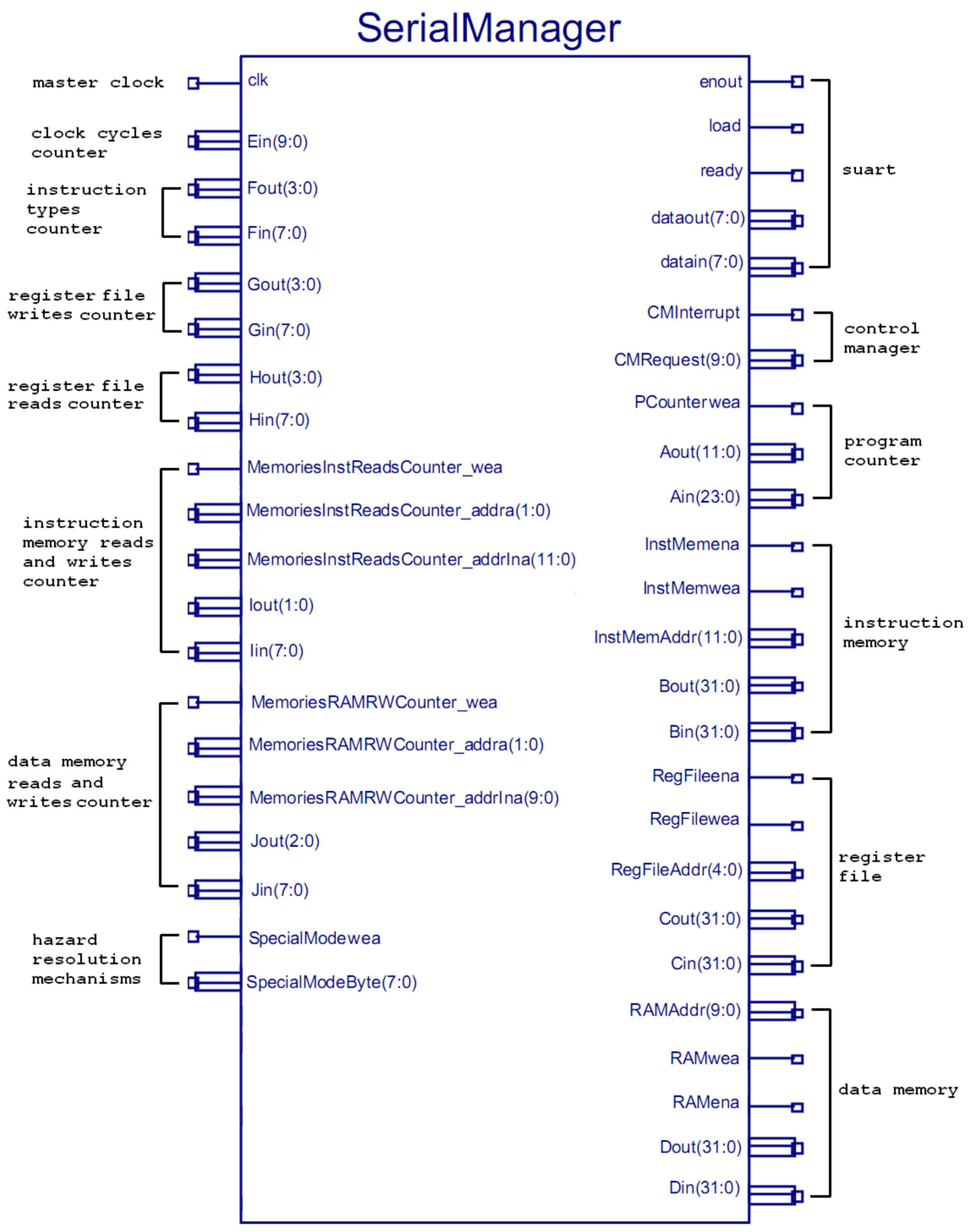

Figure A.1: Serial Manager.

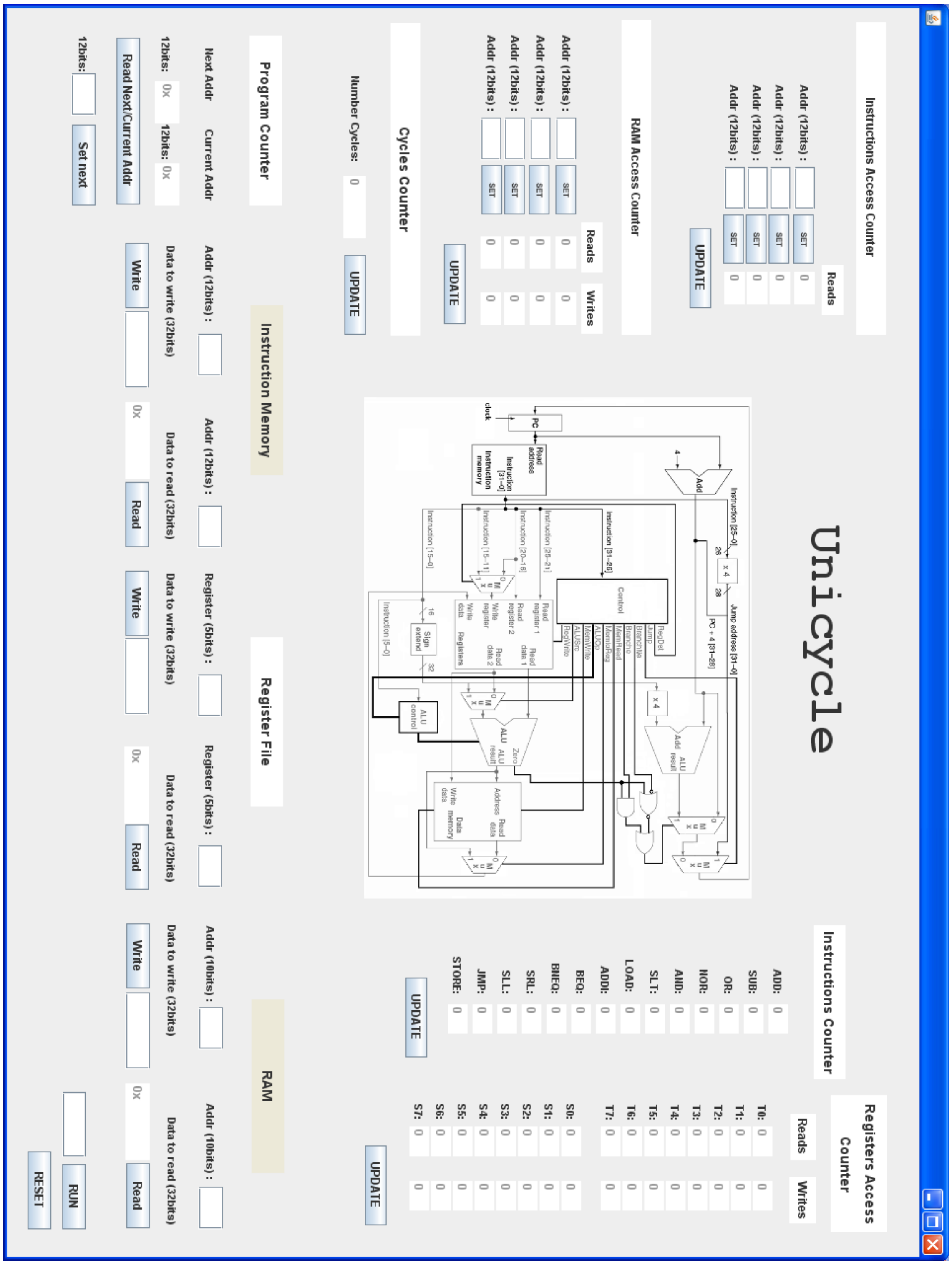

Figure A.2: Unicycle interface.

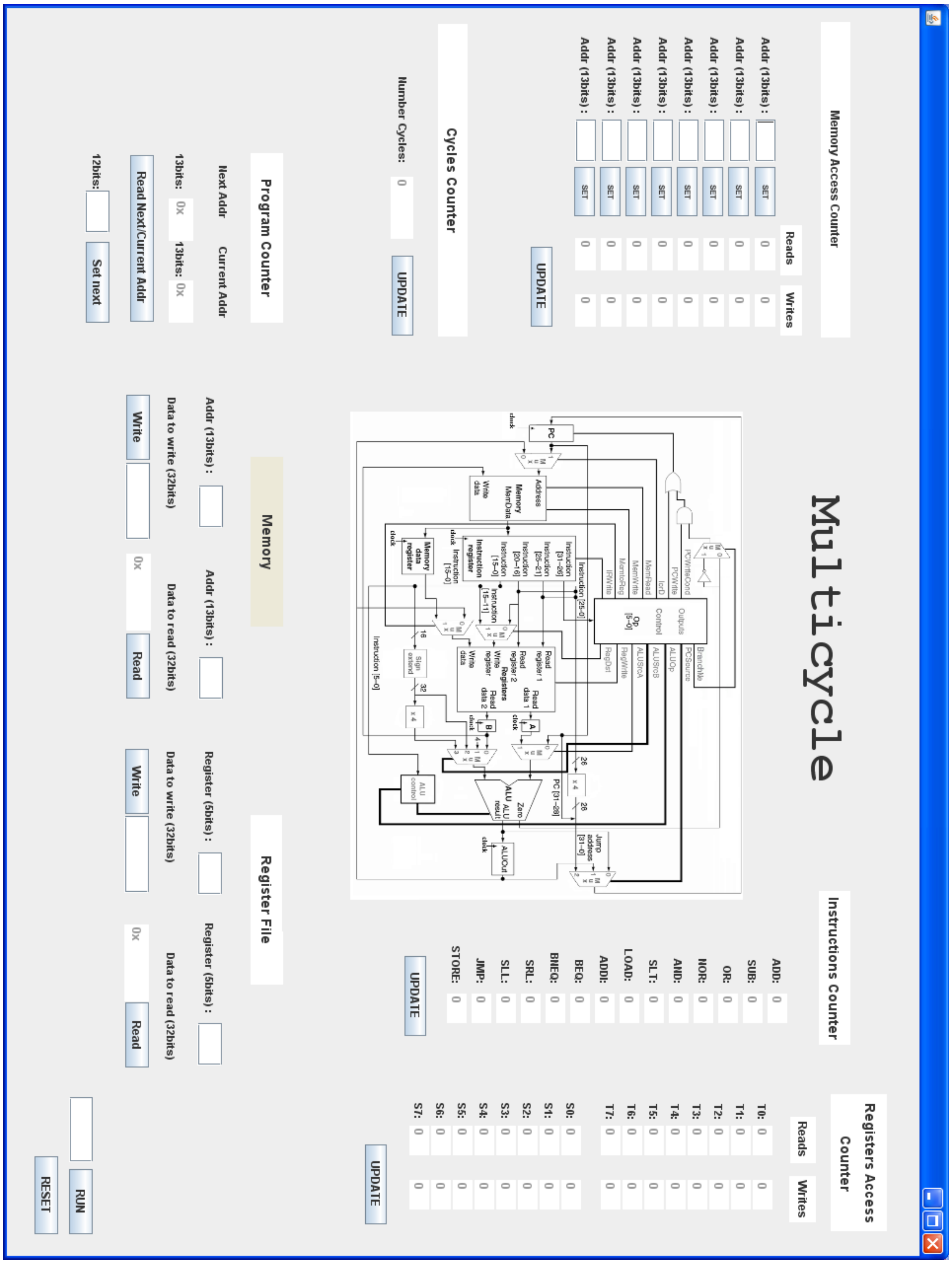

Figure A.3: Multicycle interface.

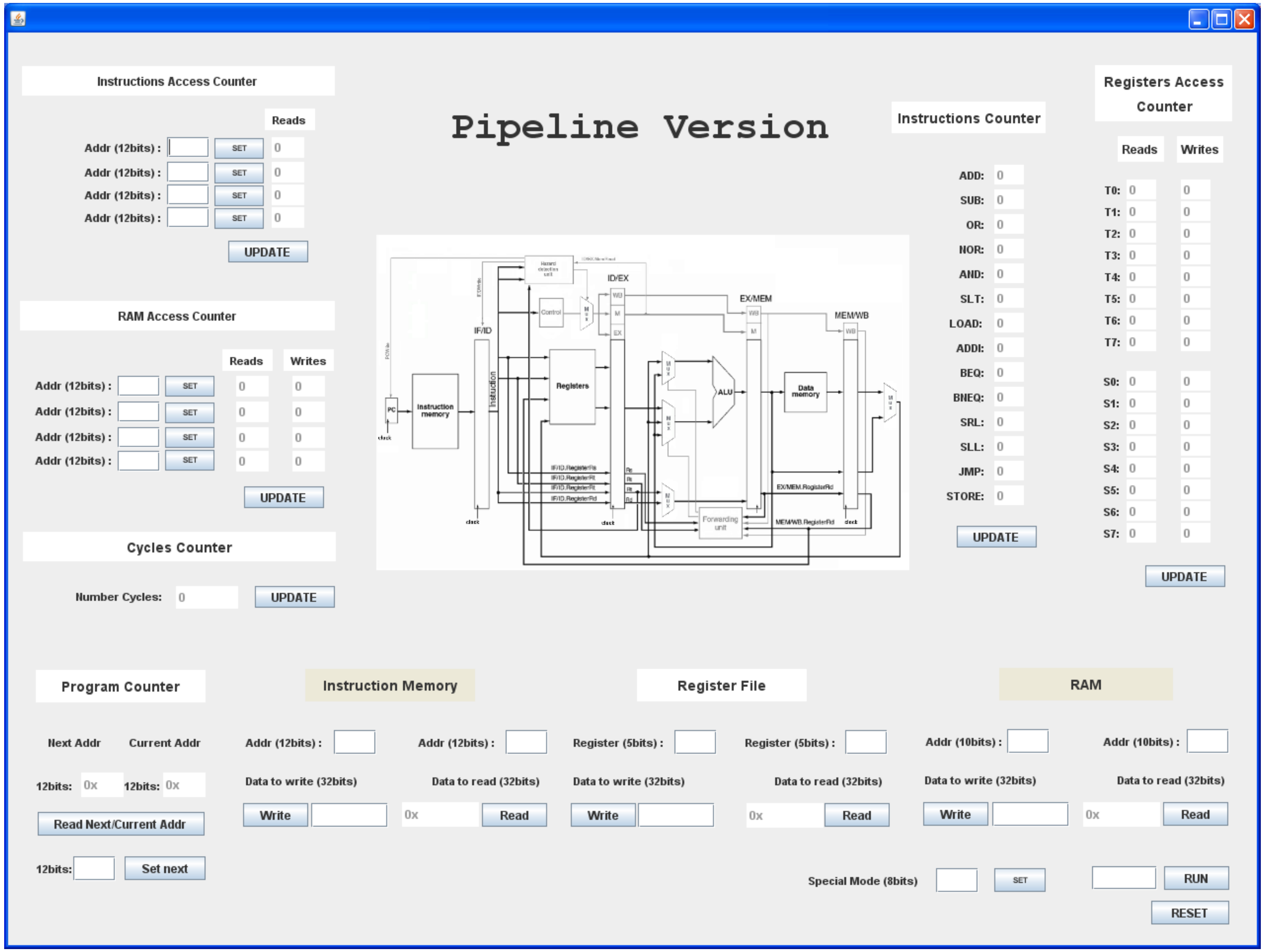

Figure A.4: Pipeline version interface.

References

[1] Dominic Sweetman. *See MIPS Run*. Morgan Kaufmann, 2007.

[2] John Hennessy and David Patterson. *Computer Organization and Design - the hardware/-software interface*. Morgan Kaufmann, 2005.

[3] Spartan-3 FPGA Family: Complete Data Sheet, January 2005.

[4] Java jit compiler, May 2009. http://www.trl.ibm.com/projects/jit/index_e.htm.

[5] Serbanescu Alexandru EIA Quinquis Andre, Cernaianu Leonardo. Didactic Software for Signal and Image Processing. 2009.

[6] John P. Hayes. *Computer Architecture and Organization*. McGraw-Hill International Editions, 1998.

[7] Gerrit A. Blaauw and Jr Frederick P. Brooks. *Computer Architecture - Concepts and Evolution*. Addison-Wesley, 1997.

[8] José Monteiro Guilherme Arroz and Arlindo Oliveira. *Arquitectura de Computadores: dos Sistemas Digitais aos Microprocessadores*. IST Press, 2007.

[9] José Delgado and Carlos Ribeiro. *Arquitectura de Computadores*. FCA - Editora de Informática, Lda, 2008.

[10] GCT Semiconductor Leverages MIPS Technologies' Analog IP for Mobile WiMAX(TM), 2009. http://www.design-reuse.com/news/20310/afe-mobile-wimax.html.

[11] Knowledgerush.com. MIPS Processor, 1999. http://knowledgerush.com/kr/encyclopedia/MIPS_processor/.

[12] RISC Architecture, 2009. http://cse.stanford.edu/class/sophomore-college/projects-00/risc/mips/index.html.

[13] Zé Paulo Leal. As optimizações com o pipelining, 2009. http://www.dcc.fc.up.pt/~zp/aulas/9899/me/trabalhos/alunos/Processadores/pipelining/main.htm.

[14] Sonza Reorda. The DLX Architecture, March 2003. Pdf.

[15] David R. Kaeli and Philip M. Sailer. *The DLX Instruction Set Architecture Handbook*. Morgan Kaufmann, 1996.

[16] The Spartan-3 Platform FPGA Family. The World´s Lowest-Cost FPGAs, 2004. Xilinx, Inc Manual.

[17] João Fabio Pegorin Di Lello. Tópicos em Arquitetura e Hardware, March 2006.

[18] Carl Wilhelmsson. FPGAs - How does they work?, February 2009.

[19] Darren Zacher. Using the Resource Manager in Precision® RTL Plus Synthesis. September 2007.

[20] Narinder Lall. FPGA Judgment Day: Rise of Second Generation Structured ASICs.

[21] John Withers. *Devel Java Entertainment Applets*. Wiley Computer Publishing, 1997.

[22] Michael Juntao Yuan. *Enterprise J2ME - Developing Mobile Java Applications*. Prentice Hall, 2004.

[23] Chris Adamson. What is java, 2009. http://www.onjava.com/pub/a/onjava/2006/03/08/what-is-java.html.

[24] 2009. Callisto Discovery Site. ftp://ftp.inescn.pt/pub/util/eclipse/technology/phoenix/demos/install-ve/install-ve.html.

[25] James Harris Mark Holland and Scott Hauck. Harnessing FPGAs for Computer Architecture Education. *IEEE International Conference on Microelectronic Systems Education*, 2003.

[26] Roberto Carli. Flexible mips soft processor architecture. Master's thesis, Massachusetts Institute of Technology, 2008.

[27] Arnaldo Silva Rodrigues de Oliveira. *Especialização e Síntese de Processadores para Aplicação em Sistemas de Tempo-Real*. PhD thesis, Universidade de Aveiro, 2007.

[28] S. Pizzutilo and F. Tangorra. Archo: A computer based learning system for teaching computer architecture. 2003. http://www.actapress.com/PaperInfo.aspx?PaperID=14891\&reason=500.

[29] Enrico Martinelli Irina Branovic, Roberto Giorgi. Webmips: A new web-based mips simulation environment for computer architecture education. 2009.

[30] James Laurus. spim, a MIPS32 Simulator, 2009. http://pages.cs.wisc.edu/~larus/spim.html.

[31] MARS - MIPS Assembler and Runtime Simulator, January 2009. http://courses.missouristate.edu/KenVollmar/MARS/index.htm.

[32] John Benedict B. Villangca Anastacia P. Ballesil John Edrian H. Aguilar, Rosario M.Reas and Joy Alinda P. Reyes. DLX Gold: Design and Implementation of a DLX Microprocessor with Single Precision Floating Point Operations. 2007.

[33] R.Selvakumar Rajagopal and Muhammad Mun'im Ahmad Zabidi. FPGA Implementation of DLX Microprocessor With WISHBONE SoC Bus. 2009. http://www.design-reuse.com/articles/18600/dlx-microprocessor-wishbone-bus.html.

[34] Çetin Koca. MIPSIM - MIPS Assembly Language Simulator, February 2008. http://www.mipsim.com/mipsim/.

[35] Sebastian Kuligowski. RS232 in Java for Windows, 2008. `http://www.kuligowski.pl/java/rs232-in-java-for-windows,1`.

[36] Tim Parys. Caspr - a configurable assembler program, July 2008. `http://www.ele.uri.edu/~tparys/caspr/`.