Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Development of a Highly Capable Processor for a Multicore Architecture on FPGAs #4

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Mark Antony Holland

Tutores:
José Flich Cardo
Xavier Molero Prieto
Pedro López Rodríguez

CURSO 2014-2015

Resumen

En este trabajo se diseña e implementa un procesador con ejecución fuera de orden siguiendo como modelo el algoritmo de Tomasulo. El procesador es reconfigurable y permite tanto la instanciación de un número variable de unidades funcionales y recursos como la obtención de diferentes configuraciones, cada una con una relación prestaciones/recursos distinta. El procesador se está integrando en la arquitectura PEAK desarrollada en el Grupo de Arquitecturas Paralelas (GAP) del Departamento de Informática de Sistemas y Computadores (DISCA) de la Universitat Politècnica de València (UPV).

El procesador incluye todos los componentes esenciales para su completa operatividad así como soporte para un conjunto amplio del juego de instrucciones de la arquitectura MIPS32. Cabe añadir que todos los componentes se han diseñado e implementado por completo en el marco del presente trabajo.

El trabajo incluye el diseño de tests de prueba y diferentes programas para verificar y validar cada componente y las diferentes configuraciones finales del procesador. Por otro lado, se ha sintetizado cada uno de los componentes con el fin de obtener los recursos que necesita para su implementación en un sistema FPGA. A lo largo del desarrollo del trabajo se han utilizado herramientas comerciales como Vivado de Xilinx, simuladores (QtSpim) y software de control de versiones (Git).

Palabras clave: procesador, PEAK, FPGA, multinúcleo, parametrizable.

Resum

En aquest treball es dissenya i implementa un processador amb execució fora d'ordre seguint com a model l'algorisme de Tomasulo. El processador és reconfigurable i permet tant la instanciació d'un nombre variable d'unitats funcionals i recursos com l'obtenció de diferents configuracions, cadascuna amb una relació prestacions/recursos diferent. El processador s'està integrant en l'arquitectura PEAK desenvolupada en el Grup d'Arquitectures Paral·leles (GAP) del Departament d'Informàtica de Sistemes i Computadors (DISCA) de la Universitat Politècnica de València (UPV).

El processador inclou tots els components essencials per a la seua completa operativitat així com suport per a un conjunt ampli del joc d'instruccions de l'arquitectura MIPS32. Cal afegir que tots els components s'han dissenyat i implementat per complet en el marc del present treball.

El treball inclou el diseny de tests de prova i diferents programes per a verificar i validar cada component i les diferents configuracions finals del processador. D'altra banda s'ha sintetitzat cadascun dels components amb la finalitat d'obtenir els recursos que necessita per a la seua implementació en un sistema FPGA. Al llarg del desenvolupament del treball s'han fet servir diverses ferramentes co-

mercials com ara Vivado de Xilinx, simuladors (QtSpim) i programari de control de versions (Git).

Paraules clau: processador, PEAK, FPGA, multinucli, parametritzable.

Abstract

This project involves the design and implementation of a processor with out-of-order execution using the Tomasulo algorithm. The processor is configurable, allowing a variable number of resources and functional units. Different configurations can be created, each with a different performance/resource ratio. The processor is being integrated into the PEAK architecture developed by the Grupo de Arquitecturas Paralelas (GAP) del Departamento de Informática de Sistemas y Computadores (DISCA) de la Universitat Politècnica de València (UPV). PEAK is a multi-core arquitecture for multi-FPGA development environments and prototyping.

The processor includes all the essential components and is fully operational along with support for a wide array of the MIPS32 architecture instruction set. All components have been designed and implemented as part of this project.

A multitude of tests and programs have been designed to verify and validate each component along with the different configurations of the processor. The synthesis of each of the components and the processor (in its different configurations) has also been performed with the goal of obtaining the resource usage on a FPGA system. During the development of this project different commercial tools have been used such as Xilinx Vivado, simulators (QtSpim) and version control software (Git).

Keywords: processor, PEAK, FPGA, multicore, configurable.

Abbreviations

ALU (*Arithmetic Logic Unit*)

BRANCH (*Branch*)

BTB (*Branch Target Buffer*)

BUS (*Common Data Bus*)

CAS (*Controlled Adder Subtractor*)

CLA (*Carry-Lookahead Adder*)

COMMIT (*Commit stage*)

CPA (*Carry Propagation Adder*)

CPI (*Cycles Per Instruction*)

CSA (*Carry Save Adder*)

DEC (*Decoder*)

DISCA (*Department of Computer Engineering*)

EX (*Execute stage*)

FPGA (*Field Programmable Gate Array*)

FPU (*Floating Point Unit*)

GPR (*General Purpose Register*)

HDL (*Hardware Description Lenguage*)

HI (*High register*)

ID (*Instruction Decode*)

IDE (*Integrated Development Environment*)

IF (*Instruction Fetch*)

IPC (*Instructions per cycle*)

L1 (*First Level Cache*)

LO (*Low register*)

MEM (*Memory*)

RAM (*Random Access Memory*)

PC (*Program Counter*)

RB INT (*Integer Register Bank*)

RB FP (*Floating Point Register Bank*)

RISC (*Reduced Instruction Set Computer*)

ROB (*Re-Order Buffer*)

RR (*Round Robin arbiter*)

RS (*Reservation Station*)

WB (*Writeback*)

Contents

1. Introduction **1**

1.1. Context . . . 1

1.2. Objectives . . . 3

1.3. Structure of the Document . . . 3

1.4. Use of References . . . 4

2. Prior Knowledge **7**

2.1. IEEE Standard for Floating-Point Arithmetic . . . 7

2.2. The MIPS Architecture . . . 8

2.2.1. Instruction Format . . . 10

2.3. Instruction Scheduling . . . 11

2.3.1. Tomasulo Algorithm . . . 13

2.3.2. Dynamic Branch Prediction . . . 14

2.3.3. Memory Disambiguation . . . 14

3. Tools **17**

3.1. Xilinx Vivado . . . 17

3.2. QtSpim . . . 19

3.3. Git . . . 20

3.4. log2timetable . . . 21

4. Overview of the Processor **23**

4.1. Datapath . . . 23

4.2. Dataflow . . . 27

4.3. Processor Configurations . . . 28

5. Specific Components **31**

5.1. 32-bit Register Bank 31
5.2. Branch Target Buffer 34
5.3. Memory Access Unit 36

6. Results **41**

6.1. Design Verification 41
6.1.1. Superscalar 42
6.1.2. Memory Disambiguation 44
6.1.3. Conditional Branches 46
6.1.4. Benchmarks 48
6.2. Design Synthesis 50

7. Conclusion **55**

7.1. Outcome 55
7.2. Extensions and future work 56

Appendix **57**

A. Datapath buses **59**

A.1. Register Banks 59
A.2. Branch Target Buffer 60
A.3. Memory Access Unit 60

B. Benchmark source code **61**

B.1. axpy 61
B.2. pi 62

Bibliography **64**

List of Figures

2.1. Simple precision floating point 8
2.2. MIPS 10
2.3. I-type 10
2.4. J-type 11
2.5. R-type 11
2.6. MIPS program 1 12
2.7. MIPS program 2 13
2.8. MIPS program 3 15

3.1. Vivado 18
3.2. QtSpim 19
3.3. Git 20
3.4. log2timetable 21

4.1. Datapath structure 24
4.2. Dataflow overview 28

5.1. Diagram of RB register fields 32
5.2. Diagram of the RB 32
5.3. Diagram of the BTB 34
5.4. Diagram of BTB register fields 35
5.5. Automaton used by the BTB 36
5.6. Diagram of the MEM 37
5.7. Diagram of byte enable bits 38

6.1. Arithmetic program 1 42
6.2. Timetable 1 43

6.3. Timetable 2 . . . 43
6.4. Arithmetic program 2 . . . 44
6.5. Timetable 3 . . . 44
6.6. Memory access program 1 . . . 45
6.7. Timetable 4 . . . 45
6.8. Branch program 1 . . . 46
6.9. Timetable 5 . . . 47
6.10. Branch program 2 . . . 47
6.11. Timetable 6 . . . 47
6.12. *axpy* benchmark . . . 49
6.13. *pi* benchmark . . . 50
6.14. Configuration G synthesis (LUTs) . . . 51
6.15. Configuration G synthesis (REGs) . . . 52
6.16. Resource usage of all configurations (LUTs) . . . 53
6.17. Resource usage of all configurations (REGs) . . . 53

A.1. fromDEC bus encoding . . . 59
A.2. fromBUSX bus encoding . . . 59
A.3. fromCOMMITX bus encoding . . . 59
A.4. fromCOMMITX bus encoding . . . 60
A.5. RSX bus encoding . . . 60

List of Tables

6.1. Processor configurations 42

Listings

4.1. Macro usage example 28
4.2. Processor configuration using macros 29

CHAPTER 1

Introduction

The work detailed in this document involves the design and implementation of a processor with *out-of-order* execution based on a MIPS architecture.

An important aspect about the implemented processor is that it is quickly and widely configurable providing the possibility to add and remove resources in order to create distinct configurations which can later be studied. The processor has the possibility of being in any one of seven different configurations.

The most complex configuration supported by the processor contains four instruction decoders with a configurable size re-order buffer, four buses, four commit units, two arithmetic-logic units, two floating point arithmetic units and multiple reservation stations for each operator.

Some other notable aspects are that the floating point unit is contained entirely inside the core of the processor. Jumps and conditional branches with prediction are also supported so that more complex algorithms may be executed on the processor. Finally, memory disambiguation is supported which enables safe *out-of-order* execution of read & write operations.

One last thing to consider is that this project was developed by a group of four students. We have all been involved since the first day and the work has been distributed such that each member could specialize in certain parts of the designed processor. Once it was time to validate and evaluate the resulting processor as a whole, all members worked together to solve bugs and draw conclusions to the capabilities of the different configurations of the processor.

1.1 Context

The project came about as a continuation of the work being done on the PEAK architecture created by the *Grupo de Arquitecturas Paralelas* (GAP) of the *Departamento de Informática de Sistemas y Computadores* (DISCA) located at the *Universitat*

Politècnica de València (UPV). PEAK is a shared memory multicore processor architecture. The objective of PEAK is the use of emulation techniques on FPGA systems for use in investigation and teaching about the design of new multicore processor architectures. PEAK allows multiple variations of an architecture design starting with the definition of the cores and then moving on to changes in the memory hierarchy and the management of resources then finishing by allowing the use of different memory coherence protocols and defining communication protocols for accessing the processor from an external system.

The main purpose of defining and using PEAK is to facilitate the investigation of advanced methods of core & cache memory management in multicore systems. This is under the concept of *capacity computing* which defines the use of partitioned resources (in this case cores and cache memories) of a multicore processor where each partition is assigned to a different application. This partitioning allows the processor to concurrently execute applications using disjoint sets of resources while providing security and privacy between applications through isolation. This means that applications never interact with each other but results in a more efficient use of the resources provided by the chip. In order to provide these characteristics PEAK has defined a set of coherence protocols and routing mechanisms along with the possibility of reconfiguration at the network level inside of the chip, all of which can be modified from the software control layer.

The latest work being done on the PEAK architecture has the objective of creating a processor with 256 cores that exhibits the previously mentioned improvements provided by resource partitioning. This work is part of an agreement between UPV and a Chinese multinational corporation. The project intends to show off the concept of capacity computing with a large number of cores. Additionally the PEAK architecture is being used as the starting point in a European project centered on High Performance Computing (HPC) which will begin in October 2015. In this project the concept of *tile* that is defined by the PEAK architecture will be used such that each *tile* will have specialized cores for different types of tasks while also having different levels of performance and consumption. As an example, the processors defined in the work presented in this document can be used in different tiles and in each of them be configured with a different number of resources (FPU units, ROB size, etc). The goal of the project is to ensure the use of the optimum number of resources in each moment so as to achieve the best performance/resource ratio.

Another important part of the work contained in this document is its relationship with Field Programmable Gate Arrays (FPGAs). By using a programmable board with its accompanying software we were able to use rapid prototyping and behavioural analysis of the digital designs we have implemented so as to quickly and cheaply verify our designs without the need of having to turn to a chip fabrication plant and then physically debug the resulting silicon. This also required us to learn how to use a hardware description language we had not worked with before, which is Verilog. This involved adjusting to thinking in a paradigm where

nothing is sequential, everything is a cable and electricity travels at the speed of light so great care must be taken when writing code because essentially it all happens instantly and at the same time.

Finally, the chance to work in a group was a big reason to go ahead with the project because we knew that by combining each of our skills we could finish a more expansive and complete body of work while supporting each other and allowing greater efficiency during it's development.

1.2 Objectives

- The main objective of the work presented in this document is to design and implement the necessary modules that will form an architecture based on that of the MIPS32. The development of the architecture will be with the use of Verilog code and the Xilinx Vivado IDE.
- The processor supports different types of instructions: read/write to memory and registers, arithmetic-logic operations and conditional branches along with jumps.
- Provide rapid reconfiguration of the processor allowing more or less resources to be present in the architecture which could imply for example, providing multiple arithmetic-logic units or by changing the number of Reservation Stations (RS) available to each operational unit.
- Use pipelined designs for the arithmetic-logic and memory units in order to support concurrent execution of instructions and memory disambiguation.
- Develop a range of test programs written in assembly in order to verify the correctness of the final implementation with all the modules interconnected that form the architecture.
- Synthesize the project using Vivado and then perform an analysis of the space that each component of the architecture would occupy on an FPGA. This part will be performed using several different configurations of the processor providing more or less resources and observing the variations in resource usage on the FPGA by the processor.
- Make the adjustments necessary to incorporate the project into the PEAK architecture with the goal to then program the processor onto an FPGA.

1.3 Structure of the Document

This document is made up of seven chapters each of which is briefly introduced below.

- **Chapter 1, Introduction:** To begin with we explain the motivation and context behind the work produced along with the objectives to complete.
- **Chapter 2, Prior knowledge:** Here we provide the necessary concepts to better understand the implications and mechanics of the work produced.
- **Chapter 3, Development tools:** In this section we detail the tools used during the project along with the advantages they bring and how they are used.
- **Chapter 4, Overview of the processor:** Next we describe the set of components required to create a functioning processor along with the tasks they perform and how they are all interconnected.
- **Chapter 5, In-depth view of components:** Each member of the group was in charge of the development of certain units. Here these specific modules of each member are introduced in greater detail.
- **Chapter 6, Verification and results:** A series of tests in order to verify the developed project are explained while also analysing the resulting efficiency and resource usage of the final product.
- **Chapter 7, Conclusion:** In this final chapter we detail how each initial objective was completed and also possible work that could be added in the future.

Because the work detailed in this document was performed as a team composed of Francisco Guaita, Mark Holland, Raúl Lozano and Tomás Picornell, each member dedicates sections in this document to a more in-depth look at the implications the units they spent most time working with had on the rest of the architecture.

At the end of the document there is an appendix with the interfaces of the components seen in chapter 5 along with the testbench source code used in chapter 6

1.4 Use of References

During the development of the project we have made use of multiple bibliographic materials which are detailed here along with their relation to the different parts of the project.

- To situate ourselves in the historical context of the type of architecture our design is based on, which is that of the MIPS32 microprocessors, we make use of a Wikipedia article [9] where MIPS processors in general are explained.

- Because one of the objectives of this project is to construct a processor with *out-of-order* execution we base ourselves off of the structures defined in the books by Hennessy and Patterson about computer architecture [6, 7]. We also use the official MIPS manual [10] that introduces the MIPS32 architecture.

- In order to correctly decode instructions and execute them on our architecture we follow the encoding defined in the official MIPS manuals that cover the set of instructions supported by the MIPS32 architecture [11, 12].

- Another important factor is correctly following the official standard for floating point numbers set by the IEEE organization [8].

- For some of the more detailed parts of the arithmetic-logic unit implementation the reference [16] is used.

- Finally, to solve questions related to the use of the chosen development tools during the creation of this work we consult the official book on Git [5] where they provide examples of common use cases, the official documentation for the program PCSPIM that contains all the needed information for its usage and to end with the user guide for Vivado [18] made available by Xilinx.

CHAPTER 2

Prior Knowledge

This chapter provides an overview of different technical aspects in order to better situate the reader in the context that this project is placed. This includes explanations of the mechanics of some of the components implemented during the development of this project.

2.1 IEEE Standard for Floating-Point Arithmetic

This standard came about because of the usage of different ways of representing floating-point numbers on large computers near the end of the 1970s. From the years 1977 to 1985 a series of meetings were held by the IEEE (Institute of Electrical and Electronics Engineers) resulting in the publishing of the IEEE 754 standard.

The format defined in IEEE 754 is used to perform operations involving real numbers in the `float` and `double` representations. The support of these representations depends on the implementation of the floating point unit (FPU). The standard also shows how simple precision (32 bits) and double precision (64 bits) floating point numbers should be represented along with how arithmetic operations should be performed when using these types of numbers.

The IEEE 754 format allows the representation of a wide range of very large numbers while using a limited number of bits. For example, a simple precision floating point number occupies a single word of 32 bits and is made up by the fields shown in **Figure 2.1**.

The first bit represents the sign of the number (S), the following 8 bits form the exponent (E) while the remaining 23 bits are the mantissa (M). A number X represented in exponential notation can be written in the form:

$$X = (-1)^S \times 1.M \times 2^{E-127}$$

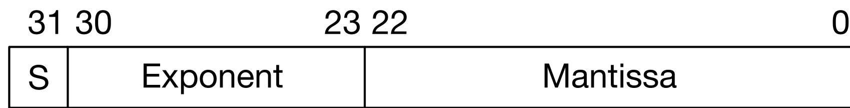

Figure 2.1: Fields of bits that make up a floating point number with simple precision

2.2 The MIPS Architecture

The MIPS (Microprocessor without Interlocked Pipeline Stages) family of microprocessors developed by MIPS Technologies use a RISC (Reduced Instruction Set Computing) architecture where instructions are expected to only make use of an operator once during the execution of an instruction unlike a CISC (Complex Instruction Set Computing) architecture where the instructions may use the same operating unit multiple times over the course of the execution of a single instruction.

MIPS processors were initially designed at Stanford University in 1981 by a research group lead by Dr. John Henessy with the goal of adopting the principles advocated by the RISC philosphy, pioneered by Henessy, to a working microprocessor. This group later founded the company MIPS Technologies Inc. in 1984 with Henessy as a co-founder. The company's first product was the R2000 model which was the first commercial RISC processor and the first to use the MIPS architecture, it was released in January 1986. The R2000 provided 32 general purpose registers but certain things taken for granted today, such as floating point operations, had to be performed by a separate chip, the R2010 floating-point accelerator. A similar thing ocurred with memory operations where four R2020 write buffer chips were also included on the R2000 chipset to allow the queuing of up to four pending memory operations and thus freeing up the microprocessor to continue executing other instructions.

The follow up to the R2000 was released in 1988 and was called the R3000. The main improvement was the addition of an on-chip cache controller along with support for cache coherence to ensure data consistency. So although this microprocessor didn't yet have any level 1 cache it did support external instruction and data cache with sizes up to 256 KB and allowed access to both caches in the same cycle. The R3000 chipset included 32 KB of cache which was later doubled to 64 KB with the introduction of the R3000A in 1989 that also supported higher clock frequencies.

The first 64-bit instruction set supporting RISC microprocessor came in 1991 with the MIPS R4000 which used a scalar superpipelined architecture and now integrated an on-die floating-point operator. Increases in clock frequency and

attempts for it to replace the CISC microprocessors of the day such as the Intel i486 weren't enough and in 1992 SGI acquired MIPS Technologies Inc.

Under SGI, MIPS Technologies Inc. began to license their designs and by the end of 1999 had consolidated in the MIPS licensing system with the architectures MIPS32 and MIPS64 (referring to 32 and 64 bit architectures). Licensing use was quickly adopted and today these architectures can still be found in all types of embedded systems ranging from portable computers to TV decoders.

The change of ownership allowed heavy investment in new designs and in 1994 the first MIPS superscalar microprocessor was released as the model R8000 which allowed the concurrent execution of two arithmetic-logic operations and two memory operations in a single clock cycle. This was achieved by introducing the concept of pipelined operators with the use of an external floating-point operator called the R8010 which also provided the possibility of *out-of-order* execution of instructions by decoupling the integer and floating-point pipeline of the processor. Unfortunately, the elevated costs limited its use outside of scientific fields and it only managed to stay on the market for little over a year.

The successor to the R8000, the R10000, was released in January 1996 and continued with the idea of *out-of-order* instruction execution by introducing register renaming and a four-way superscalar design that permitted the launching of up to four new instructions from the cache every cycle. Along with a higher clock frequency the R10000 incorporated all the necessary components on a single chip including the floating-point unit (FPU) that consisted of four functional units made up by an adder, a multiplier, divide unit and square root unit.

All of the designs that followed would be derivatives of the R10000 that included small changes to improve clock speeds and instruction throughput, until the R18000 that was cancelled during development and thus ending the R series of MIPS microprocessors.

In **Figure 2.2**, we can see the complete data path of a pipelined MIPS microprocessor along with the main operational units and how they are interconnected. **Figure 2.2** also shows the different stages that instructions pass through:

- **Instruction Fetch (IF)**: Fetch the instruction from memory.
- **Instruction Decode (ID)**: Decoding of the instruction and request for the operands from the register bank.
- **Execute (EX)**: In this step the instruction is executed. The actions performed depend on the instruction type. If it is arithmetic then the necessary operation is performed. If it is a memory access then the address is calculated from the base and shift values. If it is a branch instruction then the destination address is calculated relative to the program counter (PC) and if necessary the branch condition is calculated.

- **Memory Access (MEM)**: In this stage the load instructions read from memory while store instructions write their value. Branch instructions update the value in the PC. Arithmetic instructions continue their course towards the next stage.
- **Write Back (WB)**: In the case that the instruction produces a result then it is written to the register indicated by the instruction.

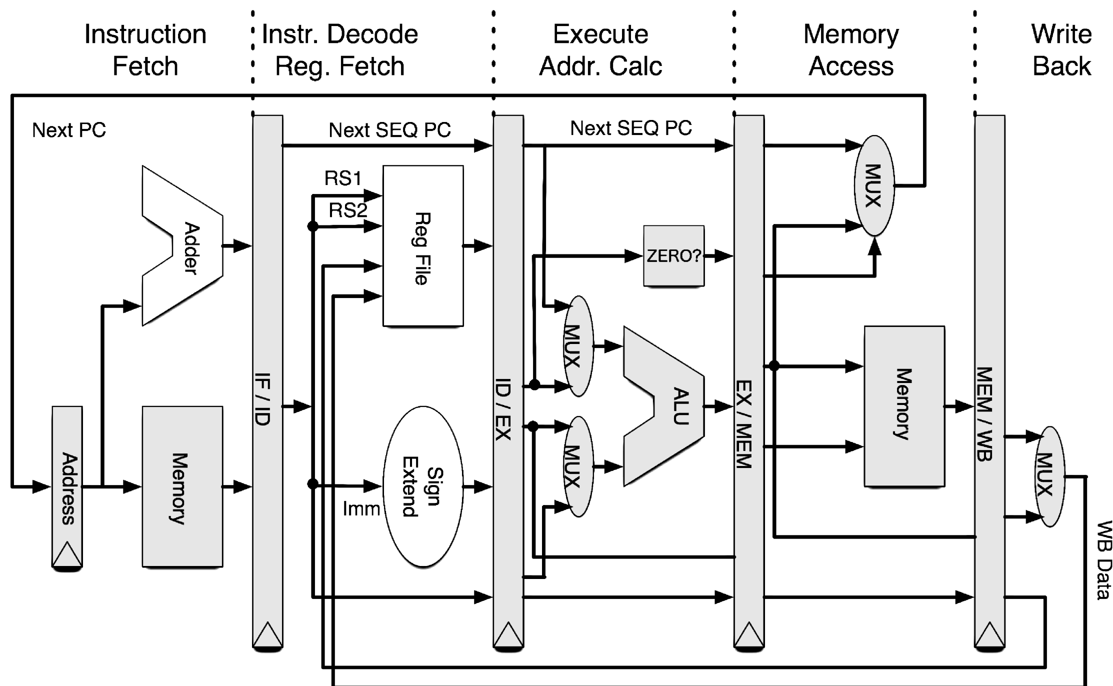

Figure 2.2: Datapath of the MIPS architecture

2.2.1. Instruction Format

The MIPS architecture has three different types of instruction formats:

1. **I-type**: These are immediate operand instructions, it always has a 16-bit integer immediate value encoded in the instruction. Two other operands may be used (rs & rt) stored in registers with index `rs` and `rt`. There is also an operation code field that has the binary encoding of the instruction to perform.

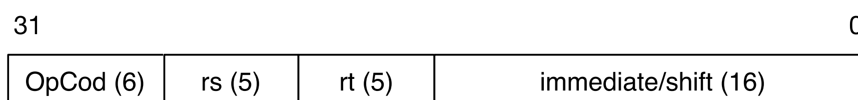

Figure 2.3: I-type instruction format

2. **J-type**: Used for encoding jump instructions, in order to change the program flow, the amount added (or subtracted in the case of a negative number) to the Program Counter (PC) is provided in the *address* field and uses 26 bits.

Figure 2.4: J-type instruction format

3. **R-type**: This format is similar to the I-type except that all operands come from registers. The three registers to use are encoded in `rs`,`rt` & `rd`. There is also a *function code* field that contains the encoded instruction that indicates the operation to perform.

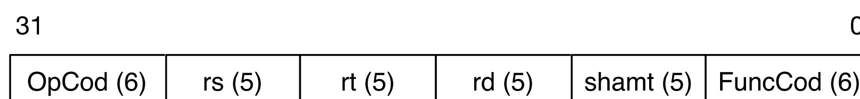

Figure 2.5: R-type instruction format

2.3 Instruction Scheduling

Historically, program instructions would be executed sequentially one after the other. If the resources for a following instruction were available the instruction would still have to wait until the previous one had finished. This way of designing a data path meant that while one instruction was being executed, the entire data path would be blocked until the instruction had completely finished. The downside to this is that in each cycle of the processor, an instruction is using only one stage of the data path when the next instruction could be using the stages the previous instruction had already passed through. Dynamic instruction scheduling solves this inefficiency.

With dynamic instruction scheduling the reordering of instructions is permitted along with multiple instructions being in the datapath at the same time. This means that subsequent instructions that don't suffer data dependencies with previous instructions can begin execution before the previous one has even began its operations. This allows a much greater throughput of instructions and vastly improves the performance of microprocessors that incorporate this type of scheduling. The introduction of non-blocking execution is of special interest when cache misses occur that can take many tens of cycles to resolve. With register renaming some data dependencies can be avoided completely allowing initially poorly

written data-dependent code to be solved at execution time and be run without any delays.

Now we will explain how the previously mentioned additions provided by dynamic instruction scheduling can be used to avoid blocking the processor datapath with the use of programs using real MIPS assembly instructions.

```
div.d $f1, $f2, $f4          # $f0 = $f2 / $f4
add.d $f10, $f1, $f8         # $f10 = $f0 + $f8
sub.d $f12, $f8, $f14        # $f12 = $f8 - $f4
```

Figure 2.6: Example of a program using MIPS32 insturctions

In the example of **Figure 2.6** we can observe how if we were using static instruction scheduling the `sub.d` instruction will be delayed because there exists a data dependency between the two previous instructions even though the `sub.d` instruction doesn't use any of the data involved in the two previous instructions which results in lost cycles while the `add.d` is waiting to begin. With *out-of-order* execution the `sub.d` could be placed before the `add.d` and begin executing. Then, when the `div.d` finishes, the pushed back `add.d` can now begin as the dependency is no longer a risk.

To support *out-of-order* instruction execution the instruction decoding stage must be split into two new stages which are:

1. **Issue** - Performs instruction decoding and detection of hazards that would create a dangerous program structure with unknown consequences.

2. **Read operands** - Waits for all the operands to be ready with their latest values in order to avoid conflicts leading to incorrect results.

Along with this division, for *out-of-order* execution to make sense, the processor must have available multiple operational units of each type so as to allow the concurrent execution of instructions of the same type that do not have dependencies between them.

With register renaming we can also solve an additional type of dependency called an anti-dependency. An anti-dependency, also known as write-after-read (WAR), happens when an instruction saves its result to a location that is used in a previous instruction so if the later instruction is re-ordered and finishes before the previous instruction then the programs structure has been changed and will provide an incorrect result.

```
div.d $f0, $f2, $f4          # $f0 = $f2 / $f4
add.d $f6, $f0, $f8          # $f6 = $f0 + $f8
s.d   $f6, 0($t1)            # MEM[0+$t1] = $f6
sub.d $f8, $f10, $f14        # $f8 = $f10 - $f14
mul.d $f6, $f10, $f8         # $f6 = $f10 * $f8
```

Figure 2.7: Example of a program using MIPS32 instructions that has an anti-dependency data hazard

In the program of **Figure 2.7** there is an anti-dependency between the `sub.d` and `add.d` instructions, the later instruction stores its result in `$f8` while the earlier `add.d` instruction uses that same register as an operand. So if the `sub.d` were to finish before the `add.d` through *out-of-order* execution then the `add.d` would be using incorrect data as an operand. By using register renaming we can remove this anti-dependency by renaming the `$f8` register used in the `sub.d` instruction to, for example, `$f9` then if the `sub.d` were to occur before the `add.d` they no longer share the same register and the dependency has been eliminated.

2.3.1. Tomasulo Algorithm

The most well known and widely used method of dynamic scheduling is the Tomasulo algorithm developed by Robert Tomasulo while working at IBM in 1967. This algorithm introduced the concepts of register renaming and reservation stations along with a common data bus (BUS) to allow all the operational units to register their results in the corresponding reservation station if necessary. To allow *out-of-order* instruction execution, the Tomasulo algorithm makes use of a re-order buffer (ROB) to ensure that even though the calculations of instructions may occur in a dynamic order, instructions must finish and write their result in a new final stage – the commit stage – where the instructions enter this stage in the original order of the source code. Reservation stations are used as an intermediate store for the resulting value of an instruction that is used in a following instruction as an input operand, thus eliminating the delay caused by waiting for the instruction producing the result to have committed its result to a register.

With these new structures, the Tomasulo algorithm completely eliminates write-after-read (WAR) and write-after-write (WAW) dependencies through register renaming and allows optimum use of the processors resources with the re-order buffer while also reducing delays for the latest operands with reservation stations. The Tomasulo algorithm saw greater use from the 1990s when cache memory was more feasible because of the cost penalty reduction of cache misses that the algorithm provides along with branch speculation that will now be introduced to the reader.

2.3.2. Dynamic Branch Prediction

With the Tomasulo Alogrithm we have seen how different types of data dependencies can be eliminated and thus avoid the hazard of incorrect results in programs run on a processor. But there is a different type of hazard, the so called *control hazard*. Control hazards occur when a program makes use of conditional branch instructions, these instructions allow the control of the program flow. There are two methods that can increase the IPC (instructions per cycle) of branch instructions: static branch prediction which occurs at program compile time and dynamic branch prediction which occurs in the processor while the program is executing. This project uses speculative execution of branches with dynamic branch prediction.

Dynamic branch prediction involves keeping a record of all the conditional branches executed in a program along with a prediction of whether the branch will be *taken* or *not taken* and whether this prediction was correct the last time the branch instruction was executed. The prediction is usually calculated by the use of a deterministic automaton that allows multiple incorrect predictions before it changes its prediction for the following branch instructions.

As soon as a conditional branch instruction is decoded the prediction is made and in the next cycle the instructions that are decoded will be the ones following this prediction. Once the branch instruction has the result of its condition calculated and is then later confirmed, if the prediction was correct then execution can continue as normal and multiple cycles have been saved. But if it turns out that the branch prediction was incorrect then all the instructions after the branch must be cancelled. Thanks to the commit stage of the Tomasulo algorithm, even though incorrect instructions were being executed, data will not be corrupted as no instruction after the branch can enter the commit stage, where results are written, until the branch has entered this stage. At which point the commit stage has discovered the incorrect prediction and launched a *flush* of the incorrect instructions already in execution including their operands and any temporary calculation.

2.3.3. Memory Disambiguation

We have contemplated how to enable *out-of-order* execution of instructions but in the case of memory operations such as reads and stores there is an extra level of complexity involved which is known as memory disambiguation. This more advanced technique is used to detect dependencies between memory operations, the most important of which is what's known as true dependencies. Resolving this type of dependency involves correctly detecting when a load operation has a conflict with a previous store operation.

```
addi $t0, $0, 0               # $t0 = $t0 + 0
add  $t0, $t0, $0             # $t0 = $t0 + $0
sw   $t1, 0($t0)              # MEM[0+$t0] = $t1
lw   $t2, 0($0)               # $t2 = MEM[0+$0]
```

Figure 2.8: Example of a program using MIPS32 memory access instructions that has a data hazard

In the example of **Figure 2.8** we can quickly observe that the `lw` operation has a conflict with the previous `sw` operation and must wait until the value in `$t1` has finished being stored in the first memory entry before loading the fresh data into register `$t2`. Another important part of memory disambiguation is storing write operations in a buffer so that they may be executed speculatively, this is needed should a conditional branch fail in its prediction and forces all following instructions to be cancelled. By using speculative stores, if a store is cancellled it only will have written to the store buffer and not to the external memory and consequently data correctness is assured and write-after-read and write-after-write dependencies are eliminated.

CHAPTER 3

Tools

In this chapter we describe the main tools (external to the project) that were used to help complete the body of work presented in this document, along with what function each of them serves.

3.1 Xilinx Vivado

The most important tool used during the project is a development environment called Vivado provided by the company Xilinx. We choose this particular product provided by Xilinx because the FPGA boards that the architecture will later run on are provided and supported by Xilinx. Vivado is an IDE that provides all the tooling needed to design, implement and integrate a design onto an FPGA.

This provides us with a single program where we can:

1. Design the modules that form the architecture by writing Verilog code in the integrated editor.

2. Use static analysis for finding syntax errors in the Verilog code and incoherencies in the design.

3. Write *testbench* code for each module that creates an instance of the module and then sends valid binary signals to the various inputs of each module in different instances of time. This is later used with the integrated simulator.

4. Run the simulation of the design using the previously created testbench modules and then observe the resulting timetable of output and register signals over a period of time in order to verify that each module's implementation is correct and eventually the architecture as a whole.

5. Synthesize the design observing statistics related to the cost of different configurations of the architecture as if it were to be implemented on an FPGA.

6. Implement the final design on the FPGA.

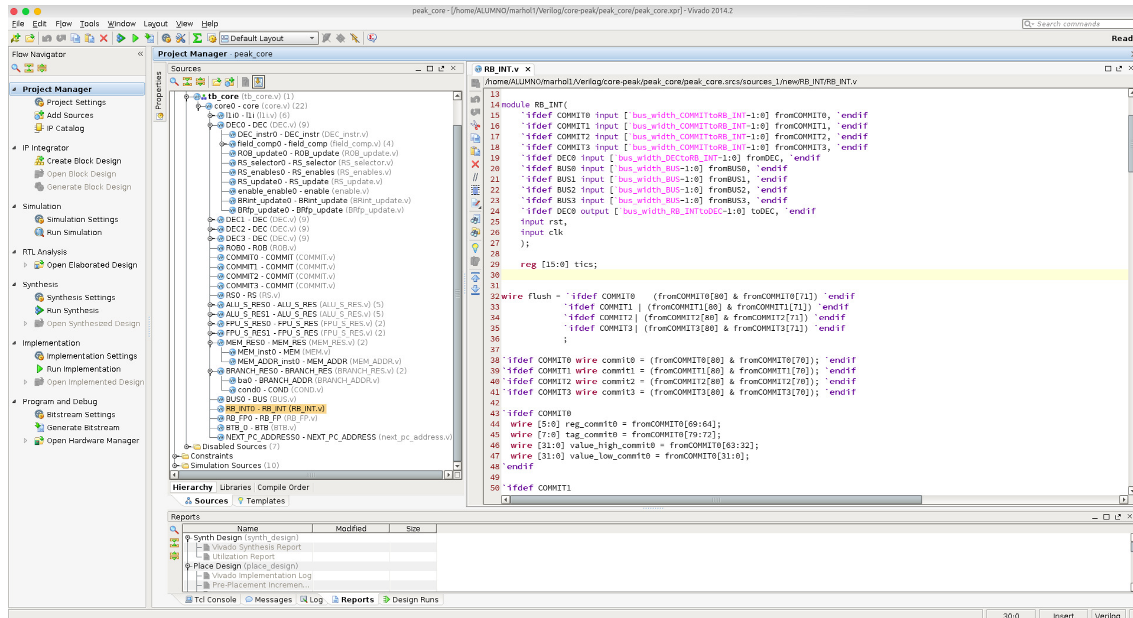

Figure 3.1: Screen capture of the Vivado IDE while editing a module

In **Figure 3.1** we can see how the Vivado IDE is split up into different sections providing an overview of all the modules in the project in a hierarchy that includes the submodules that some modules are made up of and also a text editor for the writing of Verilog code. On the far left we can see all the different steps from design to programming an FPGA, all of which can be carried out using Vivado.

3.2 QtSpim

Spim is a simulator for running MIPS32 programs. Because our architecture is implemented following this instruction set we can use the Spim simulator to test our programs written in assembly and view the correct result. Then when running the same assembly programs on our simulated architecture in Vivado we can compare the results and ensure the architecture is correct. QtSpim is the most recent version of Spim and utilizes the Qt UI framework allowing QtSpim to be cross-platform which means that everyone can use the same program for checking their assembly code.

QtSpim also provides a useful feature as can be seen in **Figure 3.2** in that it displays the encoding of the assembly instructions in hexadecimal. We take advantage of this by having the instruction cache of our architecture directly accept this encoding, which allows us to quickly pass from MIP32 assembly code to hexadecimal values that our architecture can decode and run. These encoded programs are used in *testbench* code in order to verify that the processor works correctly.

Figure 3.2: Example of execution of a program using QtSpim

3.3 Git

The work presented in this document, as has already been mentioned, has been developed in parallel as a team of four people and as such requires a great deal of synchronisation and care when designing and implementing the architecture. For this to be feasible it is necessary to use some kind of version control for the actual files with the code of the project. We choose Git because the team members were already familiar with it and it suits our needs. The University has provided us with a server so that we can have a remote repository accessible from wherever each team member may be working at any particular time.

The workflow used with git can be observed by a snippet of the project in **Figure 3.3** and involved creating new git branches for each new feature or bug fix, thus allowing time for proper code review and verification before incorporating new code into the main development trunk.

```
| | * | | | f8d1991 Finished adding ROB to TOP
| | * | | |   e524649 Merge branch 'feature/Add_DEC_to_TOP' into feature/Add_ROB_to_TOP
| | |\ \ \ \
| | | * | | | 98eb6b2 Add DEC wires to TOP
| | | * | | | d67874d Several wires added
| | * | | | | b59c1bb Started adding ROB module to TOP
| * | | | | | 9b4e865 Removed .cache from staging
| * | | | | | d1521db Updated .gitignore
| |/ / / / /
* | | | | | 16de20f Verified modules RB, RS, BUS and ROB
| |_|/ / /
```

Figure 3.3: A small list of commits in descending order showing how they diverge into a new branch and are later merged

3.4 log2timetable

To facilitate faster conclusions from the test programs run on the processor we implement a set of *python* and *bash* scripts that when combined allow a quick overview of the program execution in the form of a timetable. A bash script is used that takes as input the `vivado.log` file generated by Xilinx Vivado during a simulation run on the architecture. In this input we have specially crafted logged text from the Verilog code that contains the stage each instruction from the re-order buffer is in for each program cycle. The script cleans up this log file and then passes it to a python program that parses our special timetable logs and then prints out a timetable of the execution of each instruction.

This final output allows us to quickly check if the programs were functioning as expected and to detect improvements between configurations. An example of a resulting timetable is in **Figure 3.4** and these outputs will be used extensively in chapter 6. The x-axis shows the time measured in clock cycles. Each row of the table is a instruction and by following the x-axis it can be seen how each instruction passes through the different execution stages and in which cycles.

instr/cycles	2	3	4	5	6	7	8	9
addi $8, $0, 10	DEC	ADD	ADD	WBK	COM			
addi $9, $0, 20		DEC	ADD	ADD	WBK	COM		
addi $10, $0, 30			DEC	ADD	ADD	WBK	COM	
addi $11, $0, 40				DEC	ADD	ADD	WBK	COM

Figure 3.4: An example timetable output from log2timetable

CHAPTER 4

Overview of the Processor

In this chapter we provide a general overview of the processor created so that the reader may better understand how certain parts of the architecture work.

4.1 Datapath

In **Figure 4.1** there is a diagram with a general overview of the processor that has been developed. It is important to note that this diagram does not contain the entirety of the design due to its complexity, as such it only shows the most important modules in order to provide a brief overview while detailing how the units are interconnected. Even so, some less crucial connections are missing in order to provide a cleaner diagram.

In the previously detailed **Figure 4.1**, it can be seen how the datapath is made up of several different groups of modules (remarked in grey boxes). Each group will be detailed in-depth before passing on to the next one.

Starting from the left we will begin by looking at the group made up by:

- **Program Counter (PC):** This is nothing more than a 32-bit register that holds a pointer to the current program line being executed.
- **Branch Target Buffer (BTB):** The BTB provides the possibility of predicting the result of conditional branch instructions, that is, whether the branch is taken or not. Internally it contains a register table of configurable size where each entry represents a branch instruction of the program being executed, adding the entry as each branch instruction is first executed. Each branch instruction has an entry in the table that saves the destination address if the branch is taken, the PC of the branch instruction and a set of bits that represent the latest prediction and if it was correct. The prediction is updated by the use of an automaton with four states. If the BTB predicts a branch will

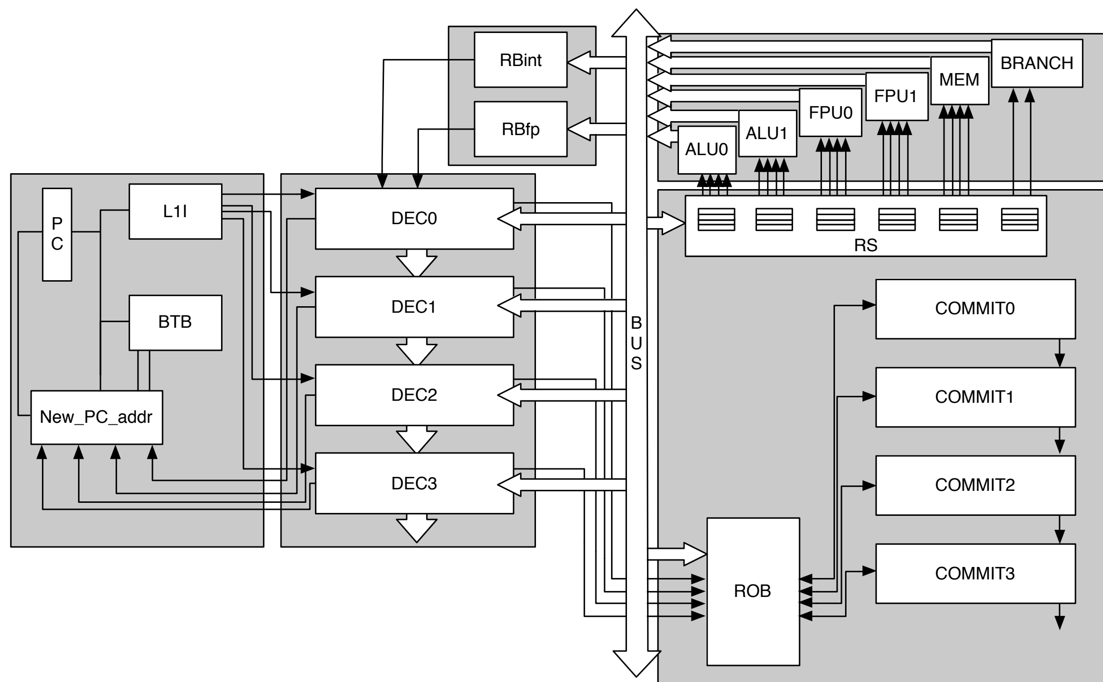

Figure 4.1: The general structure of the datapath that forms the processor along with the different operational units

be taken, it also provides the new destination address for the branch along with updating the corresponding entry in the internal table of the BTB.

- **New_PC_addr:** This unit takes the address calculated by the BTB and correctly updates the pointer in the PC so that the program flow will continue correctly after the execution of a branch.

- **Instruction cache (l1I):** This contains the set of MIPS32 instructions represented in hexadecimal that make up the program to be executed by the processor.

Next up in the pipeline we have the set of **instruction decoders (DEC)** the number of which can be configured allowing more or less instructions to enter the pipeline in a single cycle. The decoders are in charge of interpreting the binary code that represents operations on the processor and consequently gives orders to the rest of the modules in the architecture so that the expected operation is performed as indicated by the current instruction being decoded. In order to achieve this, the decoders must be made aware of the data flow in nearly all of the modules that make up the processor so that they can act as *control* units making decisions and giving orders to each module with how they should procede. All control in the processor is centralized in the decoders and as such any performance bottleneck in the processor is usually blamed on the decoders.

The **register banks (RB INT & RB FP)** take on the task of holding data during the execution time of programs on the processor. The register banks provide the input data to operations and a location to store the result of said operations. The architecture has two register banks: one for integer and one for floating point numbers. The integer register bank contains 34 registers ($0,...,$33) each of which holds a 32-bit length piece of data. The $0 register contains a hardwired value of 0 while registers $32 and $33 provide LO and HI registers to facilitate operations that have a 64 bit result, such as integer divisions and multiplications. The floating point register bank has 32 registers ($f0,...,$f31) and no hardwired 0. It also doesn't include the LO and HI registers that the integer register bank has. A choice was made to provide a simpler architecture design by having all of the registers wired to the decoder. This allows instructions to quickly move from the decoding stage to the execution stage without having to make requests to the register banks.

The next module enables the implementation of Tomasulo's algorithm, this means the pipeline requires a **re-order buffer (ROB)** so that we can make use of the *out-of-order* execution of instructions. The ROB is in charge of storing all the instructions as they are decoded and later executed as a *first in first out* (FIFO) queue so that instructions can't change the state of any data before any instruction that come before have done so first. With this we ensure the correct execution of programs. When an instruction is decoded it is added to the first available entry in the ROB. Once the instruction has finished its execution by receiving the result of the operation from the bus and setting its corresponding *write back* bit in the ROB, it then waits in the ROB until it is the oldest entry at which point it moves on to the commit stage where it saves its result to a register bank or memory. The ROB can support a configurable number of commit stages that allows multiple completed instructions to be committed and removed from the ROB in the same cycle.

Another important module to allow the correct *out-of-order* execution of instructions is the set of **Reservation Stations (RS)**. Each operator has a configurable number of reservation stations assigned to them. A reservation station contains the operands of instructions that are waiting to be executed. If an operand isn't available then the reservation station will store a mark indicating where the operand will come from in order to capture it from the bus when it is made available.

To end with the group of modules that bring support of the Tomasulo algorithm to the processor we have the **commit** module, which, as previously mentioned can be replicated to allow multiple concurrent instructions to pass to the commit stage. This unit has strong ties to the ROB and provides order with what to do with each instruction contained in the ROB and even gives the order to *flush* the ROB which involves removing all of the entries in its table of instructions. A flush occurs when a branch prediction failed (or an exception occurred) and instructions that shouldn't have been executed have started their way through the

pipeline and must be cancelled, freeing up the processor for the correct instructions that will come next.

Now that we've got all the modules needed to supply the instructions, decoded them and prepared their operands in the correct reservation stations, provided the mechanisms so that instructions can be executed and committed *out-of-order*, we next have the operators which provide the results from executing each instruction. To start with we have arguably the most important operators: the **Arithmetic-Logic Unit (ALU)** that operates with integers and the **Floating Point Unit (FPU)** for operations using real numbers represented using the IEEE 754 standard. These units follow a combinational logic design but are contained inside a wrapper with a gated output register so as to correctly regulate data flow inside the processor. Following with the high configurability of previously presented modules, the processor can be configured to have multiple instances (up to two of each) of both the ALU and the FPU as can be seen in **Figure 4.1**. This increases the capacity to have concurrent execution of instructions in the datapath that involve the use of an ALU or FPU. Both the ALU and the FPU are themselves made up of smaller sub modules contained inside the wrapper, some of which are pipelined providing an optimum use of resources. Continuing with configuration capabilities the ALU provides three different types of adder modules that can be easily chosen when configuring the processor. Multiplications and divisions are also contained in their own submodules along with logic and type conversion operators. When it comes to the FPU, great care had to be taken with this more complex unit as there are multiple extra operations to perform before and after the actual calculation takes place. Working with numbers represented in IEEE 754 requires checks on the operands before their usage to ensure compatibility for the operation, and if not, perform the corresponding conversions to make compatible operands available. After the calculation has been performed if any of the operands have been converted then the result may need a final conversion before sending the result to the common data bus (BUS).

Moving on to a different type of operator, we have the **memory unit (MEM)** which allows for the execution of more complicated programs that process large amounts of data, some of which we will see in chapter 6. The memory unit in the processor of this project has 1024 entries each of which stores a 32 bit wide datum. The MEM is pipelined in two separate stages: first the address calculation stage and second the memory operation. By pipelining the MEM we can bring forward the address calculation of an instruction whose data to store won't be available until an unknown number of cycles later. This delay could be caused by data dependencies between the instructions of the program. Seperating the address calculation and memory operation will reduce the memory operation time once the data to store is available in the corresponding reservation station. The memory operator supports three types of read/write operations which are *byte*, *half* and *word*. The type is used when calculating the address so as to ensure that the result is valid for the memory operation type. In the case of an invalid address being calculated, an exception is raised, sent to the common data bus

(BUS) and the memory operation is cancelled. The MEM (like the ALU and FPU) is contained in a wrapper with a gated output register. Inside this wrapper there is extensive logic that provides support for memory disambiguation that allows safe *out-of-order* execution of stores and reads.

The last operator to introduce is the **branch** operator that enables the processor to execute programs with complicated data flow sequences. The processor in this document supports speculative execution which allows instructions after a branch to be issued before the branch has been confirmed. The operation performed by this module is nothing more than a small calculation that provides a logical result indicating whether the branch should be taken or not.

The last piece to the pipeline has been mentioned multiple times when introducing the rest of the modules and that is the common data bus (BUS). The BUS is in charge of managing all the results of the different operators and sending them to the next stage with the use of a round-robin algorithm. The number of instances of the BUS can be configured providing a greater throughput of data leaving the operators.

4.2 Dataflow

Now that we've seen an overview of the different modules that make the processor, we will move on to an example of the dataflow that occurs while the processor is in execution.

Everything begins at the instruction cache (l1I) because this is where the program that we want to run is stored as MIPS32 instructions in hexadecimal representation. The next stage is to decode an instruction which will be the one situated in the cache at the address given by the PC register, which has the data calculated by the New_PC_address module. This module calculates the address of the instruction in the cache to be executed in the following cycle.

The instruction is sent to the instruction decoder (DEC) which checks the different fields of the instruction and decides what resources will be required and also what operands need to be transmitted to the different operators that might be involved in this particular instruction. The decoder also adds a new entry to the re-order buffer (ROB) and sends the available operands to a reservation station so that *out-of-order* execution will be correctly supported.

Once all the operands are ready in the reservation station, the dataflow moves onto the operator for the instruction being executed. When it finishes calculating, the result this is sent to the common data bus (BUS) so that the rest of the modules can check to see if they were waiting for this result to be available.

Finally, now that the instruction has completed all of its operations and provided a result, it moves on to the commit stage; releasing its entry in the ROB and writing the operation result in one of the register banks or the memory unit.

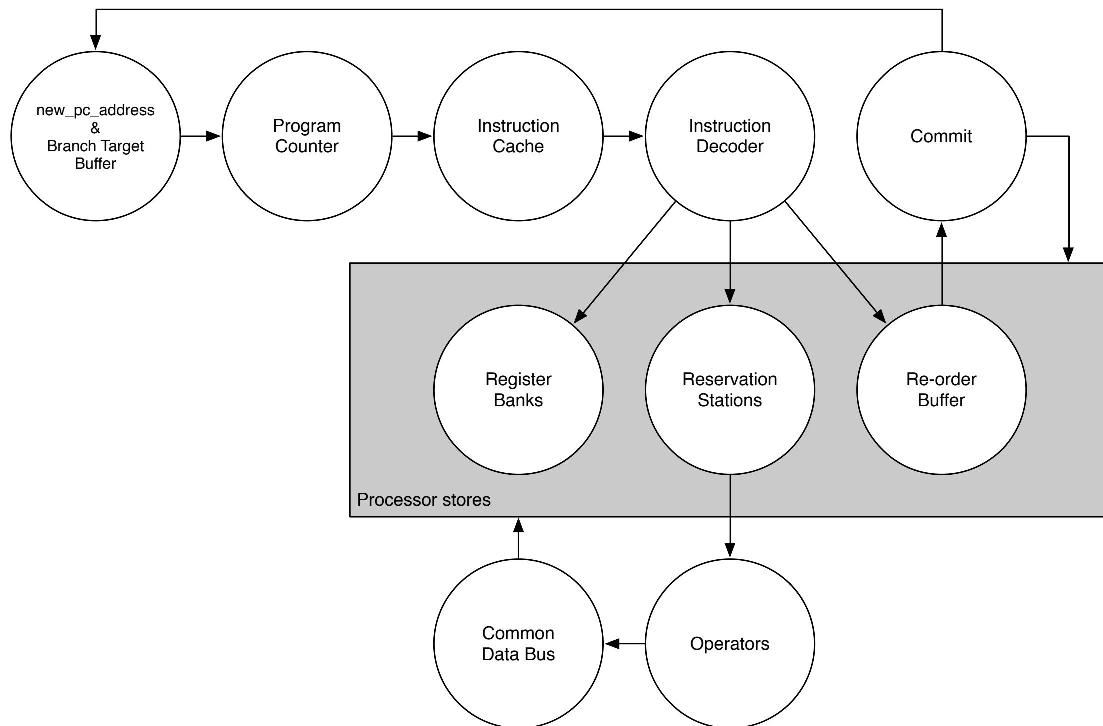

Figure 4.2: An overview of how data flows between different modules of the processor

In **Figure 4.2** we can see a brief overview of how data flows from left-to-right and top-to-bottom between the modules as an instruction is executed. Missing from the diagram are the connections from the common data bus to the modules so that they may retrieve the data they are waiting for.

4.3 Processor Configurations

Previously, we have seen that multiple parts of the processor datapath can be configured to add modules and increase the complexity of these modules. The technique we use to achieve this is by using preprocessor macros so that different sections of our code are conditionally compiled. All the possible configurations are available in our code, we just hide parts of it by using the macros: `‘ifdef x` and `‘endif` where *x* will have to be added to a global header file for the project if we want that section of the code to be compiled. This is accomplished with the `‘define` macro. An example of macro usage in the source code can be found in the listing 4.1 along with the corresponding entry in the global header file so that the `display` sentence will be compiled.

```
// Ths is in the source code
 ‘ifdef codeBlockA
  $display("code block A is executed") ;
 ‘endif
```

```
// This is in the header file
 `define codeBlockA
```

Listing 4.1: Macro usage example

Finally, in listing 4.2 there is an example of how the processor is configured by using macros. The example represents the simplest of the configurations that will be seen in chapter 6.

```
`ifdef CORE_1way_1op_1rs
`define DEC0
`define num_ROB_entries      16       // Number of ROB entries
`define log_num_ROB_entries  4        // log of number of ROB entries
`define BUS0
`define COMMIT0
`define RB_INT
`define RB_FP
`define ALU0
`define ALU0_RS0
`define FPU0
`define FPU0_RS0
`define FPU
`define ALU
`define FPU_RS0
`define ALU_RS0
`define CLA_level2_2stages
`define MEM0
`define MEM0_RS0
`define BRANCH0
`define BRANCH0_RS0
`endif
```

Listing 4.2: Processor configuration using macros

CHAPTER 5

Specific Components

This chapter contains a detailed look at the modules that the author spent most time working with and consequently is the most capable to provide an in-depth look at the inner workings of each of these units of the processor. A diagram showing all the inputs and outputs for each module will be shown along with a description of why the module is included in the architecture and how it delivers on its requirements.

5.1 32-bit Register Bank

Microprocessors were created to fulfill the wish of automating lengthy calculations. All calculations work with data, the type of data may vary but all operations will use some data as an input to a calculation which will then produce a new piece of data, the result. Now you could design a computer where every piece of data has to be provided by human interaction as it is needed by the processor but a much more powerful solution is to allow the processor to store and retrieve enough data so as not to require any interaction while it performs its operations. This capability to store and retrieve data can be achieved by using components that are implemented using different types of storage technology. Each of these technologies will have different speeds at which they can manipulate data but they will also have a trade-off in that faster storage will generally have a higher monetary cost. The balancing act between speed and cost leads to the so called *memory hierarchy* where different components using different storage technologies are connected in a top-down fashion with the fastest at the top. It only makes sense to use the fastest available storage mechanism when directly providing data to the internal units of the processor. This function is performed by a register bank. The register bank will be tightly coupled and contained in the datapath of the processor and will retrieve and store data in a single cycle. The architecture designed in this project has two register banks so as to support

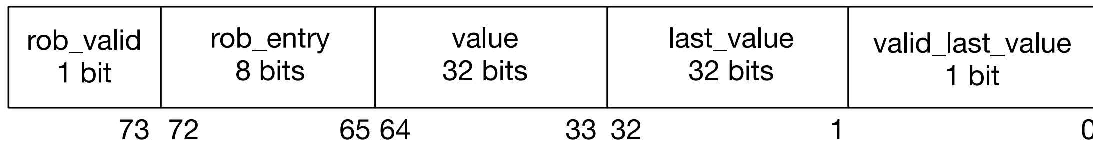

Figure 5.1: The fields that a register contains

two types of data, integer numbers and real numbers in IEEE 754 floating point representation.

Internally, each register is made up of 74 bits and contains multiple fields that can be seen in **Figure 5.1**. The `rob_entry` and `rob_valid` fields represent whether this register is currently associated with an instruction in the re-order buffer (ROB). This will be used in order to know whether data on the common data bus (BUS) corresponds to this register and should be retrieved. The `value` field will contain the 32-bit value of the register while the *last_value field* is used to store the result that will be written to the register by the latest instruction that reserved it. That instruction has reached the *writeback* (WB) stage but it has not committed yet. The usage of the *last_value* field provides support for *register renaming* and *speculation* without requiring to access the ROB to retrieve the last confirmed but not consolidated value of a register.

Figure 5.2: The connections to the RB unit

In **Figure 5.2** we can see the different inputs and outputs that both register banks utilize. The register banks are two separate modules and as such it is possible to configure the processor to have both register banks or one or the other. The register banks are synchronous modules so will require the global clock signal of the processor along with a reset signal which will trigger the erasing of all the stored data. The current design of the architecture has a trade off which is that because the instruction decoder is not pipelined it is necessary to provide the data in the register bank in an asynchronous way solely to the decoder so that the rest of the pipeline will remain in sync when decoding instructions. This is achieved by having two very large buses on each bank that lead to and from the instruction

decoder and carries the entire contents of the register banks so that instructions can be decoded in a single cycle. This connection also allows the decoder to update the associated entries of the re-order buffer (ROB) to the registers that will be used by each instruction. Updates to the values in the register banks are all carried out in a synchronous manner. The banks retrieve values from the common data bus (BUS) when the re-order buffer (ROB) entry of the data on the BUS coincides with that of the `rob_entry` field of a register in either bank. This value is assigned to the `last_value` field of the register rather than the `value` field. It isn't until a signal comes from a commit module indicating that an instruction has been confirmed that the respective register will have the definitive value in its `value` field copied.

We have seen an example of how the register banks function; data is retrieved from the bus, the final value of the data will come from the commit module once the instruction has been cofirmed. But it is important to note that following with the general idea of the project which is to provide a configurable processor, the register banks can be configured to support a variable number of buses and also commit modules. This means that in a single cycle either register bank can store the temporary or final value for multiple registers.

The increased complexity of allowing multiple buses and commit modules in the architecture also introduces some new edge cases that must be correctly treated. The first of which occurs when there are multiple instructions in the confirm phase and they are sending their final values to one of the register banks. But what if it turns out the destination register of more than one of those instructions coincides. The decision of which commit module to listen to is actually quite simple. Each commit module has an instance ID associated and this is tied to the instruction decoders that also have instance IDs, this means that the instructions will be confirmed by different commits but in order of decodification. So if we ensure that we always take the value from the highest instance ID commit module when multiple commits want to store to the same value we will have the most recently decoded instruction and in consequence the most recently updated value.

Another case is when an update to the latest value field of a register comes from the bus but at the same time the instruction decoder is changing the associated re-order buffer entry for this same register. In this case the DEC has priority and will write the new ROB entry but the register bank must also invalidate its latest value field.

The last case is when a *flush* is being performed on the register bank in order to erase all of its unconfirmed values but a new value is ready to be retrieved from the BUS. In this case the value coming from the BUS is ignored and the flush is performed as normal.

To end we will see the differences between the two types of register banks. The integer register bank (RB_INT) contains 34 registers each of which stores a 32 bit value. The first 32 registers are for general use except for the register $0 which

is hardwired to always contain the value 0 and cannot be changed. The remaining registers, $32 and $33 correspond to the registers LO and HI and provide support to instructions that produce results of upto 64 bits in size. Turning to the floating point register bank (RB_FP), its differences with respect to the RB_INT are that it only has 32 registers in total and the register $0 can be modified.

The register bank was one of the first completed modules of the project and underwent a wide range of isolated tests to ensure its functionality was correct. Even so, as the architecture became more complicated and as the project proceeded it was necessary to return to the register banks and make slight changes to its internal logic so as to continue supporting the rest of the datapath.

5.2 Branch Target Buffer

For the architecture designed in this document we have chosen to use dynamic branch prediction (see Section 2.3.2) which allows the avoidance of control hazards and improves the instructions per cycle rate (IPC) of programs with conditional branch instructions. In order to implement the mechanisms that support dynamic branch prediction we have used a Branch Target Buffer (BTB).

Figure 5.3: The connections to the BTB unit

We will begin by looking at the connections that the BTB module requires, all of which can be seen in **Figure 5.3**. The BTB is a synchronous module as it must make a prediction in time with the rest of the pipeline. This means that it is necessary to have the global clock signal of the processor which is provided as an input. As we will see, the BTB also stores data internally in registers so a reset signal is also used so as to erase the contents of these registers when necessary. Next, the module requires the address the program counter (PC) is currently pointing at, this will be used internally to identify the instructions. The last input signal comes from the commit module and has the necessary information to identify a conditional branch instruction that already had a prediction and will contain

whether the branch was taken or not allowing the BTB to know if its prediction was correct.

The outputs from the BTB include two signals for each of the possible instruction decoder present in the current configuration (up to four). The first signal for the decoders is the prediction of the current instruction. The second is an enable signal that is set to low level when a previous instruction to this one has set its prediction bit or when a *flush* command is received from the commit unit, this means the decoder must stop decoding intructions for a cycle. The last output from the BTB is the address of the next instruction to be executed when the BTB predicts that a conditional branch will be taken and will be used by the New_PC_Address module to correctly set the PC in the following cycle.

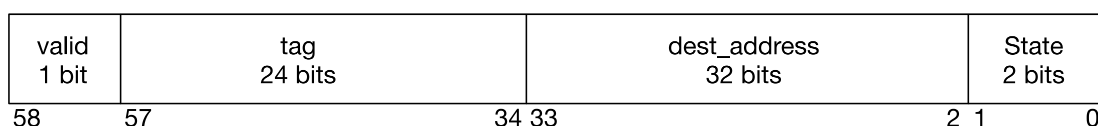

Figure 5.4: The fields that a BTB table entry contains

Internally, the BTB contains a variable size register table. Each register of this table contains several different fields which can be seen in **Figure 5.4**. Each entry in the BTB table represents a conditional branch instruction that has been decoded along with a `valid` bit to show the validity of the entry. In the `dest_address` field the destination address of the branch is stored. The *tag* field is used as an index to associate the instruction with its address in the program. The last field is the *state* the instruction is currently in. By state we mean the state of the automaton that the BTB uses to decide whether a branch is taken or not depending on whether the previous prediction was correct or not. This automaton can be seen in **Figure 5.5** and shows the four states that a branch instruction can be in. States `00` and `01` imply a prediction that the branch will not be taken and the states `10` and `11` provide a prediction that the branch will be taken. The arcs of the automaton are used when the result of whether the branch was actually taken or not arrives at the BTB. By using four states we can allow one misprediction when in a *strong* state (`00` or `11`) before changing the future predictions. The default state when an instruction is first added to the BTB table can be configured externally to the module.

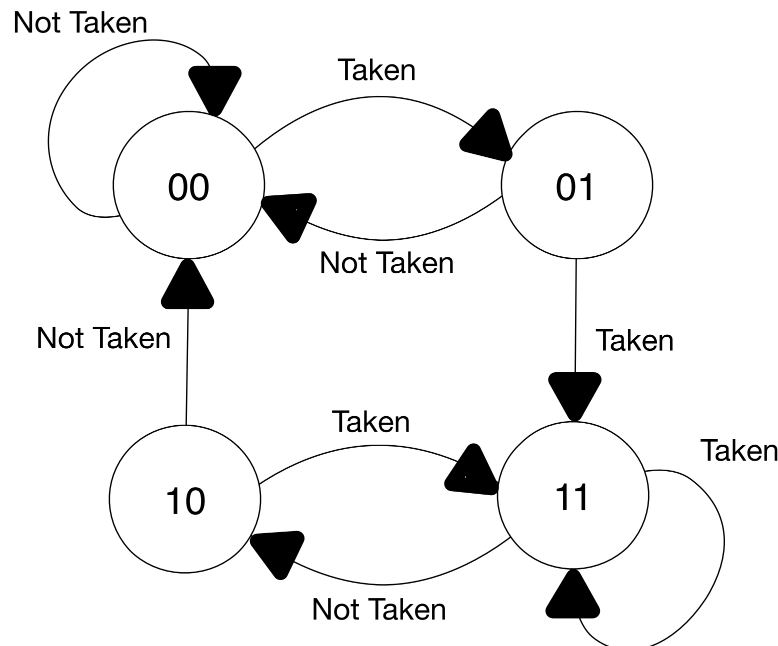

Figure 5.5: The automaton used by the BTB to change its branch prediction

With the branch target buffer module we can greatly improve the IPC of programs that make heavy use of loops with many iterations, examples of the BTB being used will be seen in chapter 6.

5.3 Memory Access Unit

We have seen the unit at the top of the memory hierarchy with the register banks (see Section 5.1), now we will see a module situated directly below the processor registers. This module is the memory access unit (MEM) situated inside the processor pipeline. For the project detailed in this document the MEM contains its own internal memory but it can easily be modified so as to connect to data stores external to the processor. The bulk of the module is the logic necessary to support the different types of read/write operations while also performing them *out-of-order* in a safe manner. This logic is split into two units inside of the MEM which can be seen in **Figure 5.6** and will be explained in detail later on.

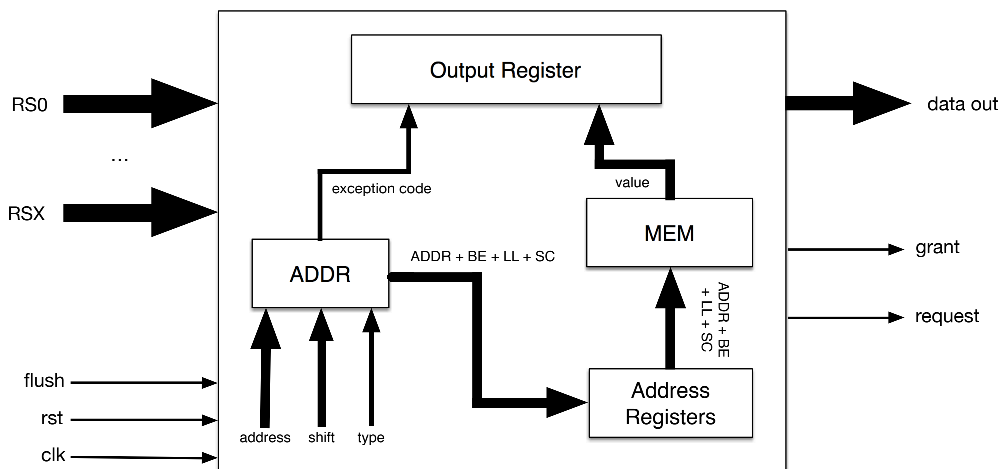

Figure 5.6: The connections to the MEM unit

To begin with the necessary input and output signals required by the memory access unit we have a global clock and reset input signals that allow the unit to stay in sync with the rest of the modules in the pipeline and to erase its contents when necessary. The other input required by the MEM is a bus for each MEM reservation station that contains all the information necessary to perform the memory operations related to the instructions associated with each reservation station. The fields that make up this bus can be consulted in **Appendix A.5**.

The outputs from the memory access unit are all connected to the common data bus (BUS). First we have two signals that are used to control the access to the BUS from MEM. The *request* signal is sent by the MEM to the BUS when it has a result ready to be sent to another unit of the processor. Next, the *grant* signal comes from the BUS and indicates that MEM can discard data at its output and proceed with the next one. This data is sent through the output *data out* of the MEM and contains a 32-bit datum or in the case of an exception having ocurred then an additional field contains the exception code. The final field of *data out* is the ID of the entry in the re-order buffer so that the other modules of the processor can detect whether the data on the BUS is for them or not.

Moving on to the internal structure of the memory access unit, which is also detailed in **Figure 5.6**, we have two units that provide the support to separate memory operations into two phases (pipelining). The first is the address calculation module (ADDR) which is in charge of taking the destination address and *shift* values sent by a reservation station and then using these to calculate the actual address to access from memory.

There are four types of read/write memory operations supported by the memory access unit and the differences are associated with the length of the datum to be manipulated. These sizes are: *byte* (8 bits), *half* (16 bits) and *word* (32 bits). Because of these different sizes, when the address is calculated the ADDR must check whether the resulting address is valid for the type of instruction associated

with this particular address calculation. For example, if an instruction wishes to access a 32 bit datum then the address calculated must be a multiple of four because data types can't be split up between different rows in memory. It is a multiple of four because we use 32-bit wide memory locations and there are four bytes in 32 bits, also each memory address points to a single byte. If the address is invalid then an exception is raised and sent to the BUS, no more memory operations will occur for this instruction.

When an address is calculated, a set of four bits are constructed called *byte enable* bits, **Figure 5.7** gives a view of what will be explained next. These *byte enable* bits represent which type of data the instruction is using, if all the bits are set to 1 then it is a *word* data type because it occupies all four bytes of the memory slot. Data type of size *half* can either have a *byte enable* value of 0011 or 1100 which represents that the address points to the lower or upper half of the memory slot. For *byte* sized data any one of the *byte enable* bits can have value 1 but only one of them, which one is enabled represents which byte of the memory slot the data is in. If after this process all the *byte enable* bits still have value 0 then the address was invalid for the type of data size this instruction uses.

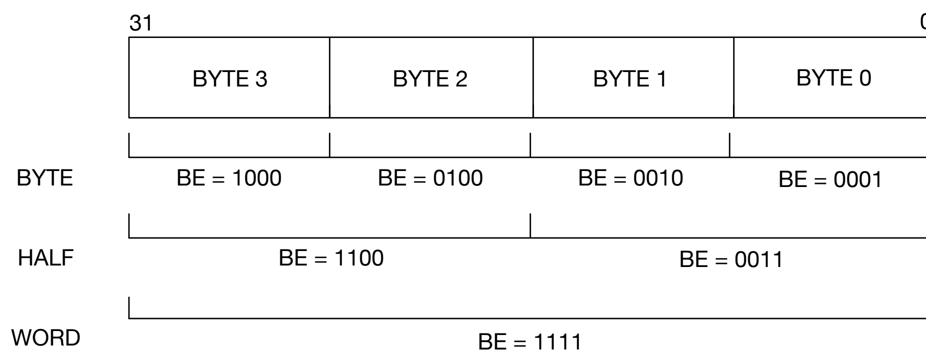

Figure 5.7: The byte enable bits are used to access the correct memory location

As can be seen in **Figure 5.6** there are two more values that leave the ADDR module, these are two bits that are used to indicate if an instruction is a *conditional store* or a *linked load*. These instructions are not currently supported by the rest of the processor and will be talked about in Section 7.2.

The data that leaves the ADDR module is sent to a set of registers, one for each MEM reservation station, where the calculated address is stored and allows memory operations to be *queued* in the case that the operation is for data to be stored but that data is not yet available in the reservation station. This is the base of the support for memory disambiguation and the allowance of *out-of-order* execution of instructions that use the memory access unit.

In the case of store operations, once the data is available and has been sent from a reservation station then the stored address along with the data is sent to

the internal MEM unit which will perform the actual store operation. By using the *byte enable* bits, the internal MEM unit will be able to save the correct part of the input data depending on the size of the data to be stored. This also occurs with load operations, the value is retrieved from the correct position inside the memory slot pointed to by the calculated address and is then sent to the output register of the memory access unit at which point a request is made for the data to be sent to the BUS.

There is one last piece of internal logic to the memory access unit and that is to detect data hazards of type read-after-write (RAW). Because the processor supports *out-of-order* execution it can occur that a *load* instruction is being performed before a preceding *store* operation has updated its value in memory. This means that the *load* operation will retrieve stale data and the program will produce incorrect results. This is solved by having the logic to detect if there is a conflict by which the address to load from coincides with that of a *store* instruction in any of the other reservation stations alloted to the memory access unit. The way this is detected is by having a set of four bits for each reservation station that will be used as a counter. These bits are referred to as *least recently used* (LRU) bits and are incremented every cycle that the reservation station wasn't accessed but any other reservation station was. When a reservation station is accessed, its LRU bits are set to 0 which indicates it is the *most* recently used. The logic involved in the increments of the LRU bits are carried out by the instruction decoder. On the other hand, if a reservation station is reset and any of the other reservation stations haven't been used for longer than this one (their LRU is higher) then they have their LRU bits decremented by one. This is to avoid overflow of the LRU counter and is carried out by the reservation station itself. By knowing when a reservation station was last used we can detect whether a store that is waiting to be executed is an older instruction than the load we wish to resolve first, in this case we delay the load until the store has finished. A detailed example of detecting previous store conflicts will be seen in Section 6.1.2.

To conclude, the memory access unit allows the execution of programs on the processor that use an effectively unlimited amount of data while doing so in a optimum manner through the use of memory disambiguation.

CHAPTER 6

Results

This chapter covers how the implemented architecture has been verified and also some observations we have made when synthesizing different modules and the processor as a whole. Verification involves running a range of programs on the processor designed to produce the different hazards that our architecture must eliminate. We also implement a set of programs that performs extensive data processing, these allow us to draw conclusions on the performance of our processor. Graphs and timetables created by log2timetable (see Section 3.4) are used extensively to quickly and easily draw conclusions. The programs all produce data that must be checked to see if they are correct. In this text these checks will not be shown and it is supposed that the programs gave the correct results. Regarding the generated timetables, during the first two cycles the processor performs a reset of all modules. These cycles will not be shown in the timetables and will begin with the first cycle when instructions are being decoded.

6.1 Design Verification

The processor implemented has the capability to configure different parts of its architecture. Some modules can be duplicated up to four times allowing the concurrent execution of different instructions. These modules are the following: the instruction decoder, the common data bus, the commit module and the reservation stations for the ALU, FPU, MEM and Branch operators. Another configurable part is the table size in the re-order buffer and branch target buffer units. Seven different processor configurations have been chosen for running tests on and they are detailed in **Table 6.1**.

The number of instruction decoders (DEC), common data buses (BUS) and commit (COMMIT) units is the same because if we can insert more instructions per cycle into the pipeline it makes sense that we can receive more results and confirm more instructions per cycle so as to avoid a natural bottleneck in the

Config.	DEC, BUS & COMMIT units	ALU/FPU units	MEM/BRANCH units	RS per operator	ROB entries
A	1	1/1	1/1	1	16
B	1	1/1	1/1	4	32
C	2	1/1	1/1	2	16
D	2	1/1	1/1	4	32
E	2	2/2	1/1	2	16
F	2	2/2	1/1	4	32
G	4	2/2	1/1	4	32

Table 6.1: Breakdown of components included in each processor configuration

pipeline. Also, it is important to note that even though the complexity of the processor is increased as we go down the table, in the case of configuration E the size of the re-order buffer (ROB) is set to 16 so as to better compare the difference that adding an extra ALU and FPU unit to the datapath provides.

6.1.1. Superscalar

To start with we see the results of executing a simple program that contains four arithmetic instructions, we then compare running the same program with a different core configuration. Finally we slightly modify the program so as to insert a data hazard and observe the changes in the timeline.

```
addi $8,  $0, 10          # $8 = $0 + 10
addi $9,  $0, 20          # $9 = $0 + 20
addi $10, $0, 30          # $10 = $0 + 30
addi $11, $0, 40          # $11 = $0 + 40
```

Figure 6.1: Arithmetic program 1 that executes four arithmetic instructions

The program in **Figure 6.1** performs adding operations and stores the result in four different registers and uses four different immediate operands. We begin by using configuration A and in the timetable of **Figure 6.2** we can see how the instructions pass through the different stages of execution. Because this is using configuration A, which has a single DEC then only one instruction enters the pipeline in each cycle. Even so, throughput is optimum as there are no dependencies between instructions. All of the instructions pass through the following stages: the instruction decoding stage (DEC), multiple stages in the arithmetic-logic unit (ADD), sending the result on the common data bus (WBK) and finally confirming the instruction (COM).

instr/cycles	2	3	4	5	6	7	8	9
addi $8, $0, 10	DEC	ADD	ADD	WBK	COM			
addi $9, $0, 20		DEC	ADD	ADD	WBK	COM		
addi $10, $0, 30			DEC	ADD	ADD	WBK	COM	
addi $11, $0, 40				DEC	ADD	ADD	WBK	COM

Figure 6.2: Timetable when executing arithmetic program 1 using configuration A of the processor

Next, in **Figure 6.3** we see the same program but when being run with the processor in the most complicated configuration which is: configuration G. To begin with, we can quickly see that overall program execution time has been reduced by two cycles.

instr/cycles	2	3	4	5	6	7
addi $8, $0, 10	DEC	ADD	ADD	WBK	COM	
addi $9, $0, 20	DEC	ADD	ADD	WBK	COM	
addi $10, $0, 30	DEC		ADD	ADD	WBK	COM
addi $11, $0, 40	DEC		ADD	ADD	WBK	COM

Figure 6.3: Timetable when executing arithmetic program 1 using configuration G of the processor

Also, because of the four decoders this configuration has available, all of the instructions of the program are decoded in just one cycle. Even so, because we only have two ALU units, in the following cycle only the first two instructions begin their operations. The ALU is pipelined so operations take two cycles to complete but this also means that in cycle four all of the program instructions are inside the ALU, but in different stages. In the fifth cycle the result of the first two instructions is sent to the BUS and in the sixth cycle both instructions are confirmed and have been completed. Even though we have the capability to confirm four instructions in a single cycle, because of the constrain incurred by only having two ALU units, the last two instructions are confirmed one cycle later than the first two. With this particular program we have seen a 77% increase in performance by evolving from configuration A to configuration G. In the section on design synthesis we will see how this improved configuration influences the complexity of the implementation on an FPGA.

The program in **Figure 6.4** is almost identical to the previous one and is run using the same configuration G as the previous example. What has changed is that the third instruction is now an addition of the values in two registers rather than a register and immediate value. The registers used in this addition have been chosen carefully so as to introduce a data hazard between instructions two and three. Instruction two produces the value for register $9 but instruction three

consumes the value stored in this same register. In order to avoid the dependency, execution of instruction three must wait for the value to be correctly stored in register $9.

```
addi $8,  $0, 10           # $8 = $0 + 10
addi $9,  $0, 20           # $9 = $0 + 20
add  $10, $9, $9           # $10 = $9 + $9
addi $11, $0, 40           # $11 = $0 + 40
```

Figure 6.4: Arithmetic program 2 that executes four arithmetic instructions and has a data hazard

In the timetable of **Figure 6.5** we have the execution of this new program when running on the processor with the most extensive configuration, which is configuration G.

instr/cycles	2	3	4	5	6	7	8	9
addi $8, $0, 10	DEC	ADD	ADD	WBK	COM			
addi $9, $0, 20	DEC	ADD	ADD	WBK	COM			
add $10, $9, $9	DEC				ADD	ADD	WBK	COM
addi $11, $0, 40	DEC		ADD	ADD	WBK			COM

Figure 6.5: Timetable when executing arithmetic program 2 using configuration G of the processor

The significant changes to the timetable involve the third instruction, the one that was modified. Now, instead of entering the first stage of the ALU in the fourth cycle as before, the instruction waits until the sixth cycle when it is safe to use register $9 that is being updated by the second instruction (via resolving the data hazard through the RS). Even though the third instruction now takes longer because of the delay caused by the data hazard, the fourth instruction is unaffected and performs its operations in the same cycles (out of order). All except for the commit stage which must be performed in the correct order of the program sequence of instructions, this means that the fourth instruction must wait until the previous delayed instruction is finished and at which point it can also be confirmed.

6.1.2. Memory Disambiguation

The next program that will be used is related to memory operations. This program was executed using the configuration G of the processor. The program involves a read-after-write dependency and will show how the processor correctly avoids any errors in the program results by delaying any *load* instructions that read from the same memory location as a previous *store* instruction.

```
addi $8, $0, 10          # $8 = $0 + 10
addi $9, $0, 20          # $9 = $0 + 20
sw   $8, 0($0)           # MEM[0+$0] = $8
lw   $10, 0($0)          # $10 = MEM[0+$0]
```

Figure 6.6: Memory access program 1 that contains a data hazard

The third instruction of the program in **Figure 6.6**, a *store word*, saves the value in $8 to the first addressable location in memory. But the next instruction, a *load word*, accesses this same memory location. The processor must ensure that the `lw` instruction doesn't execute its memory operation until the previous `sw` has been confirmed and its value written to memory.

instr/cycles	2	3	4	5	6	7	8	9	10	11
addi $8, $0, 10	DEC	ADD	ADD	WBK	COM					
addi $9, $0, 20	DEC	ADD	ADD	WBK	COM					
sw $8, 0($0)	DEC	ADR		TWB	WBK	COM	MEM			
lw $10, 0($0)	DEC		ADR					MEM	WBK	COM

Figure 6.7: Timetable when executing memory access program 1 using configuration G of the processor

The timetable in **Figure 6.7** shows the program running on configuration G of the processor. All four instructions are decoded in the first cycle as per normal and both arithmetic instructions begin their operations in the following cycle. While the arithmetic operations are being performed in cycles three and four the memory operations can calculate their corresponding memory address, in the new stage (ADR), as no dependencies exist for these values. To maintain short timetables an example showing this possible dependency has not been included. As there is only one address calculation module inside the memory unit the ADR stage occurs in separate cycles for each of the memory instructions. In the fifth cycle the value that will be stored to memory is sent to the BUS which means that the `sw` instruction can now take this value and prepare for the memory operation. In the sixth cycle both arithmetic instructions are confirmed so this means that the memory module can send to the BUS the message that it now has all the data it needs and has queued the store operation. This message is sent because memory accesses can take a long time on external hardware, by queueing the store operation in a buffer the `sw` instruction can be considered completed and be removed from the re-order buffer. In cycle 8 we see how the store operation is performed in the (MEM) stage following the confirmation of the instruction. This whole time the last instruction, the `lw`, has been patiently waiting for the store to have completed so as to correctly load the fresh data. In the ninth cycle of the program the `lw` begins its load operation from memory and in the last two cycles sends the loaded value to the BUS and then ends by confirming its completion.

There is one inefficiency in this result that could be fixed in future work. The `load` word instruction is waiting to read the value stored in memory by the un-yet completed `sw`. Rather than perform two memory operations when the value needed by the `lw` is already inside the memory unit, a shortcut could be provided allowing the load operation to take the value straight away and skip going to memory for it. However, this was decided not to be implemented as would add more complexity.

6.1.3. Conditional Branches

In order to process large amounts of data with simply structured code it makes sense to use loops and iterate over data sets. In order to perform loops in MIPS32, support for conditional branches is required. In the case of the processor of the present document it also supports dynamic branch prediction (see Section 2.3.2) which is especially useful with loops so that the next iteration may begin before the actual branch instruction has even been confirmed. Two main cases occur when working with a branch predictor, both of which will be explained with the use of two example programs.

It is important to note that the Branch Target Buffer (BTB) used in this project is configured to have a default prediction of a branch *not* being taken.

The program in **Figure 6.8** is a simple example of the branch predictor making a correct prediction of a branch not being taken. The condition in the second instruction will always be false so the `bne` instruction will never be taken. Because this is the first time the BTB has seen this branch instruction, it assigns the default prediction of *not taken*.

```
loop:
addi $8, $0, 1          # $8 = $0 + 1
bne  $8, $8, loop       # if ($8 <> $8) then PC = loop
addi $8, $0, 2          # $8 = $0 + 2
addi $9, $8, 2          # $9 = $8 + 2
addi $10, $8, 4         # $10 = $8 + 4
```

Figure 6.8: Branch program 1 that contains a control hazard

As can be seen in **Figure 6.9**, the instructions after the `bne` immediately enter the decoding phase because the BTB has predicted that these instructions will be executed. Once the BTB reaches the COM stage and confirms that the prediction was correct the following instructions are unaffected because efectively, the branch is not taken. It is also worth observing the presence of multiple data hazards in this program all of which are successfully treated so as to avoid errors in the program result.

instr/cycles	2	3	4	5	6	7	8	9	10	11
addi $8, $0, 1	DEC	ADD	ADD	WBK	COM					
bne $8, $8, loop	DEC				BCH	WBK	COM			
addi $8, $0, 2	DEC	ADD	ADD	WBK			COM			
addi $9, $8, 2	DEC						ADD	ADD	WBK	COM
addi $10, $8, 4		DEC					ADD	ADD	WBK	COM

Figure 6.9: Timetable when executing branch program 1 using configuration G of the processor

To explain the opposite possibility, when a BTB doesn't correctly predict the outcome of a branch instruction, we have the program in **Figure 6.10**. This program has a small change which is the swapping of the `bne` instruction for a `beq` instruction. This change means that the condition that before was always *false* will now always be *true* and because our BTB predicts that a branch won't be taken, this prediction will be incorrect.

```
loop:
addi $8, $0, 1          # $8 = $0 + 1
beq  $8, $8, loop       # if ($8 == $8) then PC = loop
addi $8, $0, 2          # $8 = $0 + 2
addi $9, $8, 2          # $9 = $8 + 2
addi $10, $8, 4         # $10 = $8 + 4
```

Figure 6.10: Branch program 2 that contains a control hazard

Looking at the timetable in **Figure 6.11** we can see how program execution is identical up until the eigth cycle when the branch instruction is confirmed and the processor realizes that the BTB had made an incorrect prediction. The entire pipeline must be *flushed* (FLU) which involves all instructions in execution being cancelled and all associated data in the reservation stations being deleted including their entries in the re-order buffer.

instr/cycles	2	3	4	5	6	7	8	9	10	11
addi $8, $0, 1	DEC	ADD	ADD	WBK	COM					
beq $8, $8, loop	DEC				BCH	WBK	COM			
addi $8, $0, 2	DEC	ADD	ADD	WBK			FLU			
addi $9, $8, 2	DEC				ADD	ADD	FLU			
addi $10, $8, 4		DEC				ADD	FLU			
addi $8, $0, 1									DEC	ADD
beq $8, $8, loop									DEC	
addi $8, $0, 1										DEC
beq $8, $8, loop										DEC

Figure 6.11: Timetable when executing branch program 2 using configuration G of the processor

Effectively, the three instructions after the `bne` are cancelled in the eighth cycle when the branch instruction is confirmed. Next, an empty cycle appears as instruction decoding is halted so that all the necessary registers in the pipeline can be erased. Then in the tenth cycle the first two instructions are decoded again because the branch has changed the program counter (PC) to point to the first instruction of the program. Although the behaviour in this example isn't very useful it is important to note that in the eleventh cycle the first two instructions are again decoded, this is because the BTB failed it's prediction and it has now changed its prediction for this instruction to be taken. This prediction will stay the same until it fails a prediction, which in the case of this program, will never happen.

With these examples we have seen a brief overview of the methods that were used to check the processor works correctly. This was the main goal of the project, to provide a base processor that can be analyzed and discover possible optmizations to improve instruction throughput.

6.1.4. Benchmarks

Next, we show the results when running two benchmarks on the processor and provide the calculated Instructions Per Cycle (IPC) value. Each of the programs are run using all the different configurations of the processor.

These two programs are designed to make heavy use of all the operators of the processor while executing a large number of loops. The source code of these two progams can be found in **Appendix B**. The first of these is an implementation of the well known *axpy* algorithm which involes iterating over two vectors and performing the following operation on each set of elements. $z = a \times x + y$. In our program the vectors have 64 elements each meaning that on some processor configurations the total number of cycles for the program is over 1000. All arithmetic operations are carried out by using floating point operations.

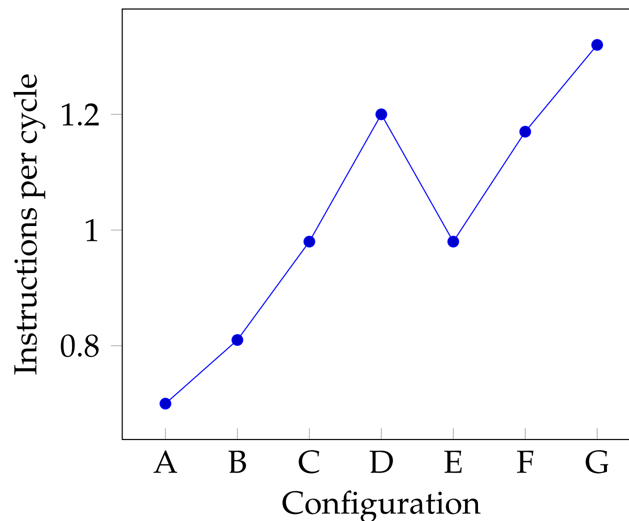

Figure 6.12: Instructions per cycle obtained when executing the *axpy* benchmark on different configurations of the processor

In **Figure 6.12** we can observe a pattern of improved performance as we introduce more complex configurations. All except in the case of configuration E which is caused by a reduction in the size of the re-order buffer (ROB) so as to better observe the difference when moving to a processor with two ALU units and two FPU units. In this particular benchmark, adding more ALU/FPU units doesn't improve instruction throughput. This is caused by the nature of the *axpy* algorithm, each addition has to wait for the previous multiplication to have been completed and then storing the current iterations result also waits for all the calculations to have been completed. So even though there are more operators being used, all of them suffer continuous delays from the the heavily data dependent code that makes up the *axpy* algorithm. It isn't until we increase the size of the ROB that we see an improvement over configuration D with configuration F and more so with configuration G that represents the most complex configuration of the processor. Overall, with the *axpy* benchmark we can observe a speedup between configuration G and A of 1.88. For such heavily data dependent code that is the *axpy* benchamark these results are more than satisfactory.

Moving on to the next benchmark, this one involves performing iterations in order to approximately calculate the *pi* number. The algorithm chosen makes use of a infinite series called the *Nilakantha* series [13]. The program performs 100 iterations of the series and involves extensive use of FPU operations.

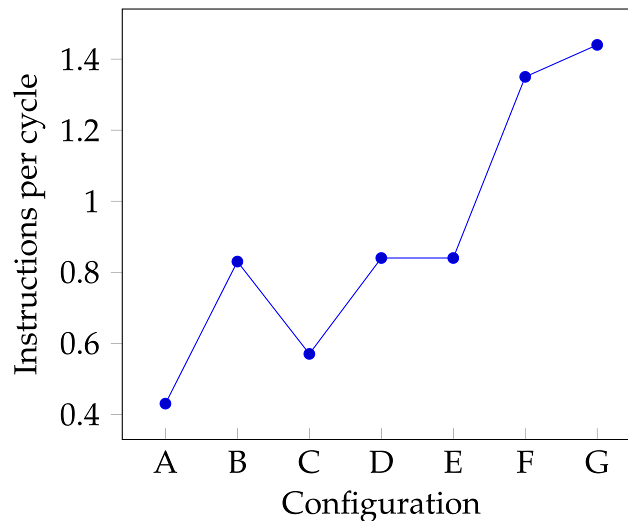

Figure 6.13: Instructions per cycle obtained when executing the *pi* benchmark on different configurations of the processor

In **Figure 6.13** we can see a similar tendency as with the *axpy* benchmark, adding more resources to the processor configuration in general improves instruction throughput. Between configuration B & C we see a drop in throughput, this is caused by the reduction in reservation stations made available to each operator in configuration C. The *pi* benchmark involves many subsequent floating point operations so providing more reservation stations to the FPU will always improve throughput. Between configuration D & E we don't see a drop in performance even though the number of reservation stations has been halved. This is because in configuration E we have doubled the number of FPU units which results in providing enough additional resources to maintain an identical throughput to configuration D with this particular benchmark. Even so, we still aren't seeing better performance than configuration B, it isn't until we start adding more reservation stations that we see an improvement with a large jump in throughput. In configuration G we see a slight improvement from allowing four instructions to be decoded per cycle. In terms of concrete improvements between configurations, the speedup going from configuration A to configuration G is 3.35. Considering the perfect speedup by increasing to four the number of instructions capable of being decoded per cycle is also four, the value obtained by the current architecture is more than promising.

6.2 Design Synthesis

We have successfully verified that the processor implemented functions correctly. Next we can take a look at how this architecture might look on an FPGA. This can be achieved by performing the synthesis step available with the Vivado

program (see Section 3.1). This step will give us the usage percentage of the different resources available on a specific FPGA. In the case of this project we work with an FPGA by Xilinx and in particular the Virtex-7 XC7VX485T [17] model.

The analysis involves the use of graphs and compares all the different configurations of the processor when synthesizing the entire processor as a whole. Later, each module is separately synthesized when used in two different processor configurations.

An FPGA provides multiple types of resources that can be used when programming the board. Here we see the amount used of two different resources. These are Look Up Tables (LUTs) and registers (REGS). LUTs make up the logic of a module while those that store data or are synchronous have a certain number of registers.

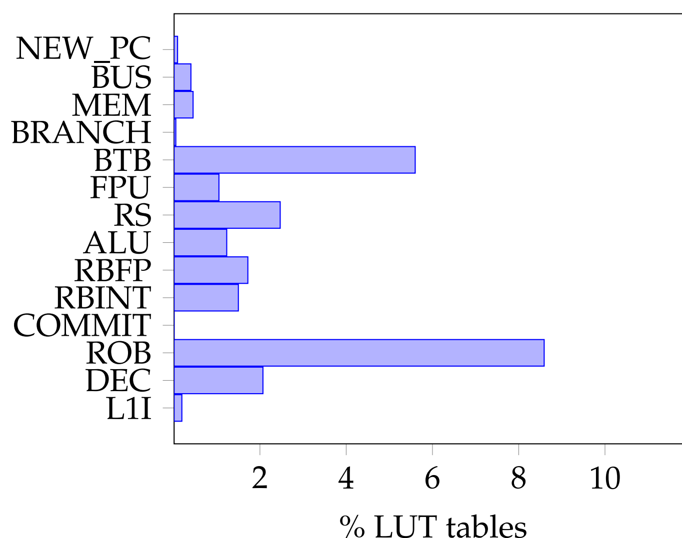

Figure 6.14: Percentage of LUTs used on the FPGA by each module when synthesizing configuration G

Figure 6.14 shows the results of synthesizing all the modules of the processor when using configuration G. The highest usage is from the re-order buffer (ROB) and Branch Target Buffer (BTB) modules. This is because these modules have the most complex internallogic and also have a large number of ports that can be read from and written to by the other units of the processor. This increases the amount of control logic requiered by these modules. The opposite occurs with the instruction cache (L1I) and memory access unit (MEM) even though these modules have large amounts of internal storage that can be accessed by the rest of the processor. The reason these two modules don't have as high a LUT usage is because they are optimized with the use of *block ram* (BRAM). BRAM is a type of resource available directly on the FPGA and as such drastically reduces the logic overhead when using registers and LUTs. In the future we hope to optimize the

ROB and BTB modules with the use of BRAMs. We also observe other modules with minimal LUT usage such as the *COMMIT* and *BRANCH* modules. This is because these units use very simple logic to function which leads to an almost nonexistent LUT usage on the FPGA.

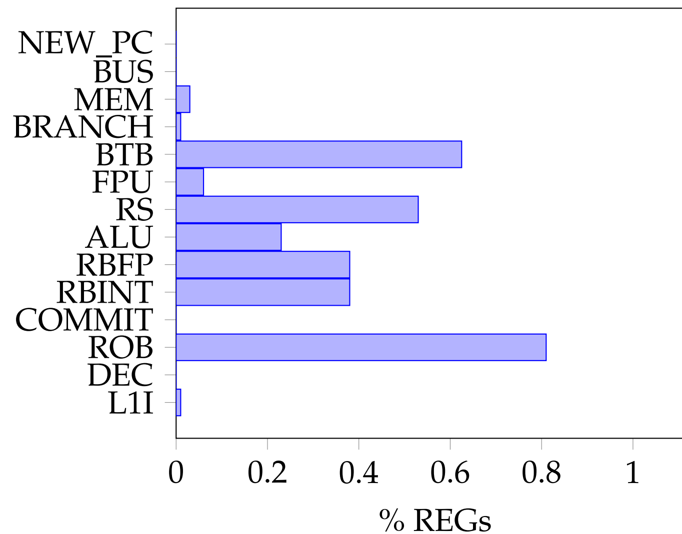

Figure 6.15: Percentage of registers used on the FPGA by each module when synthesizing configuration G

Next we look at the percentage of registers used by the different modules of the architecture when synthesizing configuration G. The results can be seen in **Figure 6.15**. The modules that use the smallest percentage of registers available on the FPGA are the instruction decoder (DEC) and common data bus (BUS). These modules are purely combinational and as such have zero register usage. The ROB and BTB use the largest percentage because they are essentially tables of registers. The same happens with the reservation stations and register banks but on a smaller scale as these modules provide less storage space. These modules are not yet optimized, if they were to use the onboard block rams (BRAMs) of the FPGA they would not have such high register usage. The L1I is already optimized with the use of BRAMs as can be seen by its low register usage in **Figure 6.15**.

We have seen the resource usage of the different modules when synthesizing a single configuration that is configuration G. Now we consider the results when synthesizing the processor as a whole and comparing all the different configurations of the processor shown in **Table 6.1**.

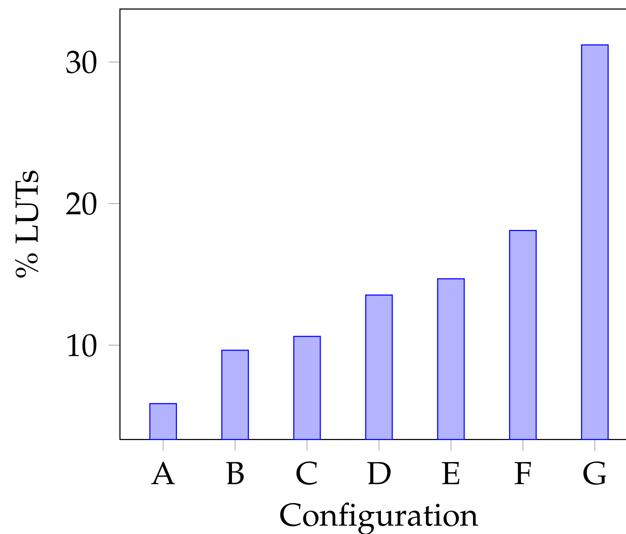

Figure 6.16: Percentage of LUT usage by each configuration of the processor

Figure 6.16 shows how as the complexity of the architecture is increased and more modules are added to the datapath then the LUT usage also increases up until the configuration G where more than 30% of the LUTs available on the FPGA are in use.

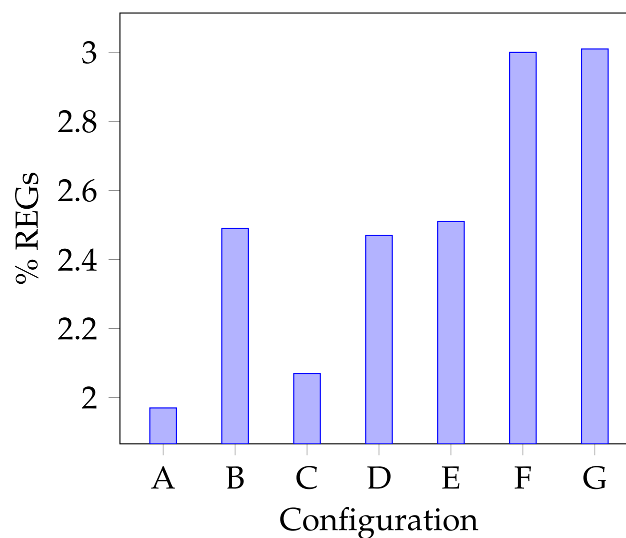

Figure 6.17: Percentage of register usage by each configuration of the processor

In **Figure 6.17** we can see that the register usage between processor configurations is very different to that of LUT usage. The register usage is largely dependent on the variation of the number of reservations stations per operator and in

small part the size of the re-order buffer (ROB). The difference between configurations F & G is minimal because even though the number of instruction decoders is duplicated, these modules are purely combinational. The only increase in registers are from other modules that may need to add some more in order to support the extra decoders.

In this chapter we have seen how the processor correctly functions and the mechanisms it uses to avoid and mitigate data dependencies along with a brief performance analysis. We have also seen the amount of usage that our design has of the resources that an FPGA provides and thus will be able to predict which FPGAs can be used with our project.

CHAPTER 7

Conclusion

This last chapter will serve as a brief revisiting of the work completed along with the problems that appeared during its development and how they were overcome.

7.1 Outcome

To begin with it seems necessary to reiterate the fact that this work was carried out as a group of four people and as such the resulting achievements are the sum of these four students. This allowed the undertaking of a project with a much wider scope than that which an individual would embark on. During the development of the project all members of the group were able to learn how to better work as a team along with overcoming the inherent problems related to a lack of synchronization that often occurs when multiple minds are simultaneously working on the same areas of a project. But by the end of the coding phase the whole team had learnt how to think as one and to forsee problems before they arose. This allowed a vast quantity of objectives to be completed along with some new ones that hadn't previously been contemplated.

Initially, we discovered that in order to use the Vivado IDE (see Section 3.1) we required a license from Xilinx, owners of the software. We had to make a formal request to the GAP so as to provide us with a valid license in order to gain complete access to all the features of Vivado.

Some examples of modules that were found necessary after the design stage had been completed and coding was well under way are: The field_comp submodule belonging to the decoder, the New_PC_addr module and a module that implements the round robin algorithm. This last improvement came about once we had an initial design that could be simulated and correctly execute programs but we quickly saw that operator starvation was occurring because of the static priorities we had assigned to each operator when entering the common data bus

(BUS). So by developing an arbiter that implemented the round-robin algorithm we were able to share the BUS in a fairer way by using a time-sharing scheduler.

During the design phase it had been decided that only a small subset of the instruction set would be supported. But later on it was deemed necessary to provide extra instructions so as to support more complex programs and better measure the performance of the processor.

Once the design had been implemented (along with the necessary modifications discovered during its development) it was of utmost importance to take great care when verifying the validity of the completed product. By simulating the processor and running different programs that contain hazards it was possible, through timetable analysis, to see how our architecture was correctly eliminating the hazards. Also, by running common benchmarks we were able to calculate the efficiency of the core and compare how different configurations altered the performance.

During the verification phase we quickly saw how our design wasn't perfect and we were forced to adapt fast to changes while ensuring all members of the team were made aware of the latest updates. Repeatedly testing different changes in the design allowed us to achieve an in-depth knowledge of the dataflow inside the processor and taught us what to look out for when trying to fix an error in the implementation. We also created a set of tools using `bash` and `python` that facilitated the drawing of conclusions as to whether a program was executing correctly on the processor.

Finally, once we had a working design we were able to synthesize the different modules that make up the core by using the Vivado IDE. We also synthesized the processor (as a whole and with different configurations) allowing us to draw conclusions on how efficient our design is at using the resources provided on the actual FPGA. This step also showed us how our implementations of the ROB & MEM could be optimized to use the onboard BRAMs of the FPGA and greatly reduce their resource usage.

7.2 Extensions and future work

The first step is to perform exhaustive validation and evaluation of the processor. New, more complex programs must be implemented along with additional tools for resource usage analysis. The objective is to find weak points in the processor along with the most efficient configurations.

The next step is to integrate the processor into the PEAK architecture on an FPGA system. There has actually been performed an intital test of this adaption.

Many of the developed modules of the processor support exceptions and launch them when necessary. However, the architecture is missing support to perform the corresponding actions to recover and return to normal program flow

following the emergence of an exception. This would involve creating a new module that would centralize the decision making tasks required to recover from an exception in the processor.

The possibility of abandoning the MIPS32 architecture has also been discussed. This is because its licensing model would impede any commercialization of the processor designed in this project. The architecture could be adapted to use a different instruction set like OpenRISC [14], which provides a more open licensing agreement.

Several of the modules have been split into a pipeline. The instruction decoder module could also be adapted to a pipelined design. This would allow the removal of the very large buses connecting both register banks to the decoder module by adding a new request phase. The request phase would request and receive the needed values from the register bank one at a time rather than having access to all of the registers as in the current architecture. Pipelining the decoder would also require changes in the register banks to support receiving data requests and sending data to the common data bus (BUS).

Currently, Reservation Stations (RS) have different internal structures and interfaces depending on the operator they belong to. A possible improvement would be to use the same structure as the re-order buffer (ROB) and have a generic RS module that can be used by any operator. This would lead to optimum use of the reservation stations.

A small change to the ROB could be done that involves reducing its use of the FPGA register resources. This would be done by switching to use the memory block (BRAM) resources of the FPGA instead.

Although the architecture has very complex and configurable arithmetic-logic and floating point units, support for double precision wasn't able to be included during the project timeline. Providing support for double precision operations would allow a greater range for representing operands and results.

To end with the last operator, the memory module has the beginning of support for two extra instructions which are the conditional store (SC) and the linked load (LL). If the processor were capable of executing these instructions it would provide the mechanisms necessary to safely run alongside other cores while using shared memory. The SC and LL instructions provide the base on which methods of memory-safe concurrency run on and would allow the implemented processor to be used in a multi-core scenario.

APPENDIX A

Datapath buses

This section contains diagrams with the fields that make up the buses connected to the different units the author of this document was assigned and were seen in chapter 5.

A.1 Register Banks

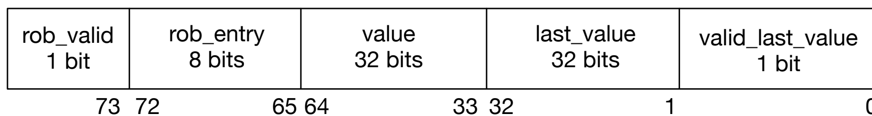

Figure A.1: fromDEC bus encoding

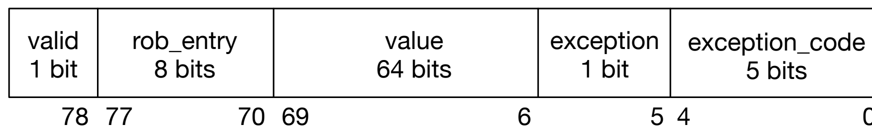

Figure A.2: fromBUSX bus encoding

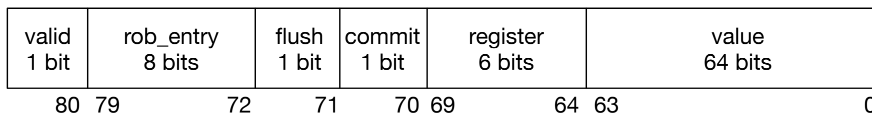

Figure A.3: fromCOMMITX bus encoding

A.2 Branch Target Buffer

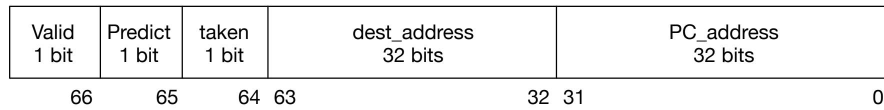

Figure A.4: fromCOMMITX bus encoding

A.3 Memory Access Unit

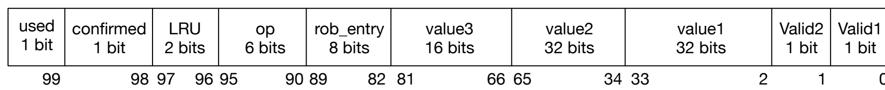

Figure A.5: RSX bus encoding

APPENDIX B

Benchmark source code

This section contains the assembly source code of the two programs used as benchmarks when studying the performance of the processor.

B.1 axpy

The following source code is from the testbench program that implements the *axpy* algorithm.

```
        #; #z = a * x + y
        #; #Vector size: 64 words
        #; #Vector x
.globl x

        .data 0x10000000
x:
        .float 0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0
        .float 10.0,11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0
        .float 20.0,21.0,22.0,23.0,24.0,25.0,26.0,27.0,28.0,29.0
        .float 30.0,31.0,32.0,33.0,34.0,35.0,36.0,37.0,38.0,39.0
        .float 40.0,41.0,42.0,43.0,44.0,45.0,46.0,47.0,48.0,49.0
        .float 50.0,51.0,52.0,53.0,54.0,55.0,56.0,57.0,58.0,59.0
        .float 60.0,61.0,62.0,63.0

.globl y
        #; #Vector y
y:
        .float 100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0
        .float 100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0
        .float 100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0
```

```
        .float 100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0
        .float 100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0
        .float 100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0
        .float 100.0,100.0,100.0,100.0

        #; #Vector z
        #; #64 elements occupy 256 bytes
.globl z
z:      .space 256

        #;# Scalar a
.globl a
a:      .float 1.0

        .text 0x00400000
.globl start
start:
la $t1, x
        la $t2, y
        la $t3, z
        la $t0, a
        lwc1 $f0, 0($t0)
        addi $t4,$0,64
.globl loop
loop:
        lwc1 $f2, 0($t1)
        lwc1 $f4, 0($t2)
        mul.s $f6, $f2, $f0
        add.s $f6, $f6, $f4
        swc1 $f6, 0($t3)
        addi $t1, $t1, 4
        addi $t2, $t2, 4
        addi $t3, $t3, 4
        addi $t4, $t4, -1
        bnez $t4, loop
        .end
```

B.2 pi

The following is the assembly source code from the benchmark program *pi* that aproximates the value of pi by using the Nilakantha series.

```
.text 0x00400000
```

```
.globl start
start:
addi $10, 100
li $2, 2          # $2 = 2
mtc1 $2, $f2      # $f2 = 2
cvt.s.w $f2, $f2  # $f2 = 2.0

li $3, 3          # $3 = 3
mtc1 $3, $f3      # $f3 = 3
cvt.s.w $f3, $f3  # $f3 = 3.0

li $4, 4          # $4 = 4
mtc1 $4, $f4      # $f4 = 4
cvt.s.w $f4, $f4  # $f4 = 4.0

li $5, 2          # $5 = 2
mtc1 $5, $f5      # $f5 = 2
cvt.s.w $f5, $f5  # $f5 = 2.0

li $6, 3          # $6 = 3
mtc1 $6, $f6      # $f6 = 3
cvt.s.w $f6, $f6  # $f6 = 3.0

li $7, 4          # $7 = 4
mtc1 $7, $f7      # $f7 = 4
cvt.s.w $f7, $f7  # $f7 = 4.0

li $8, 0          # $8 = 0
mtc1 $8, $f8      # $f8 = 0
cvt.s.w $f8, $f8  # $f8 = 0.0

li $9, 0          # $9 = 0
mtc1 $9, $f9      # $f9 = 0
cvt.s.w $f9, $f9  # $f9 = 0.0

add.s $f8, $f8, $f3  # $f8 = 3.0

bucle:
mul.s $f9, $f5, $f6   # $f9 = 2 * 3
mul.s $f9, $f9, $f7   # $f9 = 2 * 3 * 4
div.s $f9, $f4, $f9   # $f9 = 4/(2*3*4)
add.s $f8, $f8, $f9   # $f8 = 3.0 + 4/(2*3*4)

add.s $f5, $f5, $f2   # $f5 +2
```

```
add.s $f6, $f6, $f2   # $f6 +2
add.s $f7, $f7, $f2   # $f7 +2

mul.s $f9, $f5, $f6   # $f9 = 4 * 5
mul.s $f9, $f9, $f7   # $f9 = 4 * 5 * 6
div.s $f9, $f4, $f9   # $f9 = 4/(4*5*6)
sub.s $f8, $f8, $f9   # $f8 = $f8 - 4/(4*5*6)

add.s $f5, $f5, $f2   # $f5 +2
add.s $f6, $f6, $f2   # $f6 +2
add.s $f7, $f7, $f2   # $f7 +2
addi $10, -1
bne $10, $0, bucle
```

Bibliography

[1] Mark Gordon Arnold. *Verilog digital computer design: Algorithms into hardware*. Upper Saddle River, Prentice Hall PTR, 1999.

[2] *Mips Architecture and Assembly Language Overview.* University of Illinois at Chicago, [consulted: 22 february 2015, 18:00]. available at: `http://logos.cs.uic.edu/366/notes/MIPS%20Quick%20Tutorial.htm`.

[3] Joseph J. F. Cavanagh. *Digital Computer Arithmetic - Design and Implementation*. McGraw-Hill, Inc., 1985.

[4] Michael D. Ciletti. *Advanced digital design with the Verilog HDL*. Upper Saddle River, Prentice Hall, 2011.

[5] *Pro Git* git-scm, 2014, [consulted: 15 january 2015, 13:00]. available at: `https://progit2.s3.amazonaws.com/en/2015-05-31-24e8b/progit-en.519.pdf`.

[6] Hennessy, J.L., Patterson, D.A. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2012.

[7] Hennessy, J.L., Patterson, D.A. *Computer organization and design: the hardware-software interface*. Morgan Kaufmann, 2012.

[8] *IEEE Standard for Binary Floating-Point Arithmetic.* IEEE Standards Board, 1985, [consulted: 9 january 2015, 17:00]. available at: `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=30711`.

[9] *MIPS (procesador)* Wikipedia: The Free Encyclopedia. [consulted: 2 june 2015, 12:00]. available at: `https://es.wikipedia.org/wiki/MIPS_(procesador)`.

[10] MIPS Technologies, Inc. *MIPS Architecture for Programmers Volume I: Introduction to the MIPS32 Architecture*. Mountain view, & CA, marzo, 2001.

[11] MIPS Technologies, Inc. *MIPS Architecture for Programmers Volume II: The MIPS32 Instruction Set*. Mountain view, & CA, marzo, 2001.

[12] MIPS Technologies, Inc. *MIPS Architecture for Programmers Volume II: The MIPS32 Instruction Set*. Mountain view, & CA, junio, 2003.

[13] *pi* MathIsFun.com [consulted: 5 july 2015, 14:00]. available at: `http://www.mathsisfun.com/numbers/pi.html`.

[14] *OpenRISC* OpenRISC Community, 2014, [consulted: 6 july 2015, 13:00]. available at: `http://openrisc.io`.

[15] *SPIM S20 A MIPS R2000 Simulator* University of Wisconsin Madison, 1997, [consulted: 22 april 2015, 17:00]. available at: `http://pages.cs.wisc.edu/~larus/SPIM/spim_documentation.pdf`.

[16] Shrivastava Purnima, Tiwari Mukesh, Singh Jaikaran y Rathore Sanjay. VHDL Environment for Floating point Arithmetic Logic Unit - ALU Design and Simulation *Research Journal of Engineering Sciences*, Vol. 1(2), 1-6, Sehore, MP, INDIA, agosto, 2012.

[17] *All Programmable 7 Series Product Tables and Product Selection Guide* Xilinx, 2015, [consulted: 5 july 2015, 14:00]. available at: `http://www.xilinx.com/publications/prod_mktg/7-series-product-selection-guide.pdf`.

[18] *Vivado Design Suite User Guide: Using the Vivado IDE.* Xilinx, 2014, [consulted: 10 january 2015, 10:00]. available at: `http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug893-vivado-ide.pdf`.