Designing a microprocessor

Coroian Tudor Florentin

December 29, 2020

Contents

Chapter 1

Introduction

1.1 Context

General description A microprocessor, or CPU (Central Processing Unit), is the main component of any computer system. It can and should be seen as *the brain* of such systems, since, as the name suggests, it *processes* or *computes* the desired values or commands.

As such, the microprocessor is capable of executing arithmetical and logical operations along with data movement instructions and jumping instructions, which are useful when implementing loops or decission branches.

Sizes of microprocessors Microprocessors can have any size[1], but they usually come into *standard sizes*, i.e. in powers of two:

- 8-bit processors
- 16-bit processors
- 32-bit processors
- 64-bit processors

However, they can come in all sizes and some of them do. Usually, the embeded chips come in *non-standard sizes* such as 4-bit, 12-bit, 18-bit or 24-bit. This number of bits represents the amount of physical memory that can be addressed by the CPU and also the amount of bits that can be processed by one read/write operation.[1]

1.2 Project proposal

General requierment Design a CPU capable of executing at least 10 instruction.

Specifications The component will be simulated in the IDE provided by Vivado, using VHDL as the hardware description language, and it will be programmed on a Basys3 FPGA board. It will be able to execute a hardcoded program which will have the aim to show the correctness of each instruction.

[1]Here, size refferes to the lenght of the instructions that are found in the instruction set of the CPU

Design decissions Before starting the description of the design process and the design process itself, a few decissions regarding the CPU should be made in order to have a clear goal in mind which will result in a clear plan of action. As such, in what follows, the design decissions will be listed:

List 1.2.1: Design decissions

1. The datapath of the CPU will use registers as its ALU (Arithmetic Logic Unit) architecture.
2. The size of the CPU will be 32 bits, more than enough to encode the minimun number of instructions requiered.
3. The CPU will use the RISC (Reduced Instruction Set Computer) architecture.
4. The CPU will have a Single-Cycle datapath.

Remark 1.2.1: Other decissions

Any other decission in the design is considered to be a small one and it will be adressed in the *Design* chapter of this documentation.

1.3 Project plan

1.3.1 Bibliographic study

Gather information about the design of the CPU as well as deciding upon most characteristics of it.

1.3.2 Analysis

Choose a set of instruction to be implemented and available for execution on the CPU. Make sure to provide the minimum number of instructions required as well as have at least one jump instruction. Provide a documentation for each instruction.
Decide upon a structure for each type of instruction, i.e. AL-type, SC-type, I-type, J-type.

1.3.3 Design and Implement

Design the ROM and RAM used to load/store data and the program itself.
Design the Register File. (*RF*).
Design the Instruction Fetch Unit (*IFU*).
Design the Instruction Decode Unit and the Operand Fetch Unit (*IDU*).
Design the Control Unit (*CU*).
Desgin the Execution Unit (*EXU*).
Design the Memory Unit (*MEMU*).
Design the ALU Control Unit (*ALUC*).
Design the Current Program State Register (*CPSR*).
Design the circuitry necessary for the debouncing of the buttons of the board.
Design the circuitry necessary for the display of the data inside the CPU.

Design the testbench units for each of the components.

Remark 1.3.1: Separation

Each unit will be implemented as a separate file in order to make it easier to follow the flow of the code

Remark 1.3.2: The Write Back Unit (*WBU*)

Since the WBU consists of only one multiplexor, it will be implemented inside the MEMU.

1.3.4 Write program

Write a program to show the correctness of the design and encode it in the project.

1.3.5 Test

Test each component individually and solve any issues that may appear.
Connect everything together and test the whole CPU by running the program written previously.

1.3.6 Alternative route

Implement a mini-assembler capable of interpreting instructions written in Assembly Language and converting them into binary coded instructions.

Chapter 2

Bibliographic study

2.1 Overview

This document describes the design and use of a general purpose processing unit. This is what is usually referred to as *microprocessor*, since it can be easily integrated in a larger system and interract with the peripherals (if any) of that system.[1]

2.2 Types of use

In general, CPUs can be used for a wide array of applications. As such, this CPU will be designed for the purpose of general computing. This type of processor is the one usually used in computers.

2.3 Instruction Set Architecture

ARM and MIPS Each processor comes with its own ISA. For this processor, the ISA will be designed in a similar manner with the ARM architecure[4] and the MIPS architecture[5].

RISC vs CISC In the same manner, a difference can be made between RISC and CISC (Complex Instruction Set Architecture), which are two different types of instruction set architectures. The RISC architecture has *reduced instructions* which means that each instruction does so little that the time in which it is done is low. The CISC architecture has *complex instructions* which means that the instructions are more advanced and do more than a single RISC instructions, albeit in a longer time.[1]

Why ARM? In ARM state, all instructions are conditionally executed according to the state of the CPSR (Current Program Status Register) condition codes and the instruction's condition field. This field (bits 31:28) determines the circumstances under which an instruction is to be executed. If the state of the C, N, Z and V flags fulfils the conditions encoded by the field, the instruction is executed, otherwise it is ignored.
There are sixteen possible conditions, each represented by a two-character suffix that can be appended to the instruction's mnemonic. For example, a Branch (B in assembly language) becomes BEQ for "Branch if Equal", which means the Branch will only be taken if the Z flag is set. [4]

2.4 Data and program memory

Options When it comes to the memory, there are two types of approaches: the Von Neumann Architecture and the Harvard Architecture[1]. The difference between these two approaches is the separation of the data memory from the program memory. More explicitly, in the Von Neumann approach, instructions are represented in the same way as data, whereas in the Harvard approach, instructions and data occupy separate memory modules.

Drawbacks In a Von-Neumann architecture, the same memory and bus are used to store both data and instructions that run the program. Since you cannot access program memory and data memory simultaneously, the Von Neumann architecture is susceptible to bottlenecks and system performance is affected. [6]
If a Von Neumann machine wants to perform an operation on some data in memory, it has to move the data across the bus into the CPU. When the computation is done, it needs to move outputs of the computation to memory across the same bus. The amount of data the bus can transfer at one time (speed and bandwidth) plays a large part in how fast the Von Neumann architecture can be. As such, this CPU will use the Harvard Architecture.

Why Harvard? There are at least two memory address spaces to work with, so there is a memory register for machine instructions and another memory register for data. Computers designed with the Harvard architecture are able to run a program and access data independently, and therefore simultaneously. Harvard architecture has a strict separation between data and code. Thus, Harvard architecture is more complicated but separate pipelines remove the bottleneck that Von Neumann creates. [6]

2.5 Components

Some common components of the microprocessor are:

- the Control Unit which generates the control signal
- the Arithmetic Logic Unit (ALU) which does the computing itself
- registers for data transfer or temporary data storage
- a control unit for the ALU
- memory for the instructions of the program
- memory for permanent data storage

Additionally, my version of CPU will have a Current State Program Register to store the 4 flags (*zero, overflow, carry, negative*).

These components along with the ones presented in the previous chapter are the ones used in the design of this CPU.

2.6 Endianness

What is endiannes? The terms endian and endianness refer to the convention that decides the ordering of bytes when processor stores a word data from its register to memory or the other way around, loads a word data from memory to its register.
Another distinction can be made when in comes to the endianness of the microprocessor. As such, we distinguish between *Little Endian*[1] and *Big Endian*[2].

The difference Little-endian machines let you read the lowest-byte first, without reading the others. You can check whether a number is odd or even (last bit is 0) very easily, which is cool if you're into that kind of thing. Big-endian systems store data in memory the same way we humans think about data (left-to-right), which makes low-level debugging easier.[7] As a consequence, since I want the debugging process to be easier more than to access the lower byte first, this CPU will use the Big Endian approach.

2.7 Memory addressing

2.7.1 Types of memory addressing

There are multiple ways in which the instructions of the CPU is able to fetch its operands. These ways will be described next. The approach in what regards memory addressing will be specified in the next subsection as well as the reason for it.

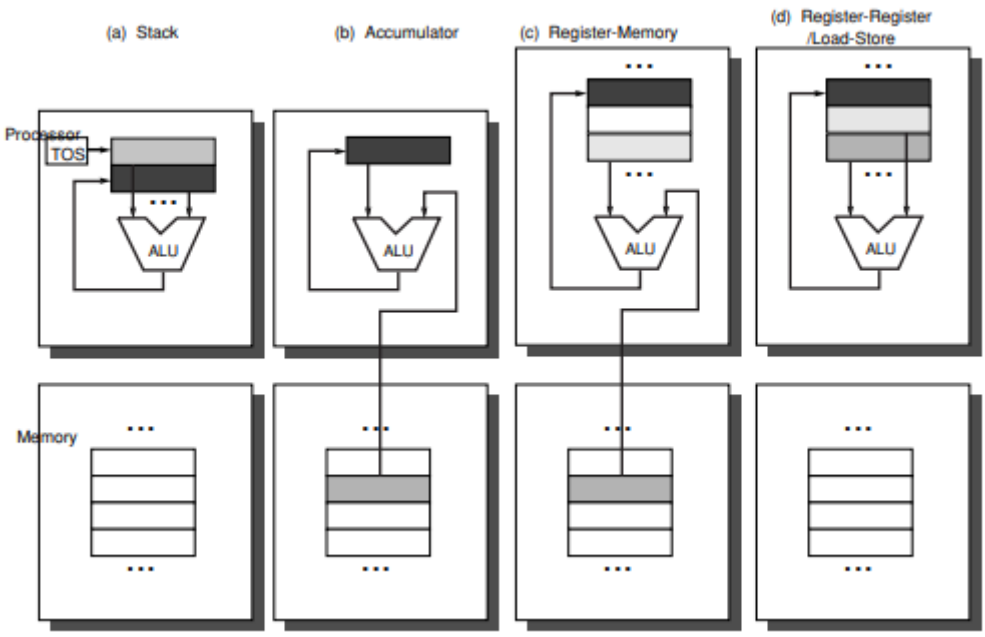

Figure 2.1: Types of memory addresing
[8]

[1]The Most Significant Bit (MSB) is stored at a higher address.
[2]The MSB is stored at a lower address.

0 - address machines Also known as *Stack* addressing mode, this type of machines operate solely on the stack. In this manner, their instructions have no operand, thus the name *0 - address machines*. The way in which they operate on the stack is by pushing the operands onto the stack, perforimng the operation, then pushing the result back on the stack.

1 - address machines Also known as *Accumulator* addressing mode, this type of machines operate on a defined register in which the result is stored. In this sense, one operand is the accumulator itself and the other one is taken from the memory.

2 - address machines Also known as *Register - Memory* addressing mode, this type of machines operate with a register file. This means that one operand is taken from a register in the register file and the other operand is taken from memory. In this case, the result will be stored in another register in the register file.

3 - address machines Also known as *Register - Register* addressing mode, this type of machines operate mainly on the register file. As such, the operands of the instruction are registers in the register file, as well as the result. This allows for a better reuse of data.

More operands Some CISC machines permit a variety of addressing modes that allow more than 3 operands (registers or memory accesses), such as the VAX "POLY" polynomial evaluation instruction.[9]

2.7.2 The choice

For this project alone, I chose to implement the CPU as a 3-address machine because it allows for a better data reusability and clearer assembly code.

Remark 2.7.1: Type of addressing

Due to the fact that some of the arithmetical and logical instruction will be designed so that they can be performed on 3 registers, this can be also considered a 4-address machine, i.e. 3 registers for the operands as 1 register for the result.

Chapter 3

Analysis

3.1 Instruction Set

3.1.1 Types of instructions

Similar to the MIPS architecture, this CPU will have several types of instructions, namely:

List 3.1.1: Types of instructions

- AL-type instructions - This type of instructions includes arithmetical and logical instructions.
- SC-type instructions - This type of instructions include instructions such as compare or shift. In other words, shift and comparing instructions.
- I-type instructions - This type of instructions include the Load and Store and also other operations with immediate values.
- J-type instructions - This type of instructions are the jump and branch instructions, with the posibility to link.

The difference from the classical R, J, I type instructions of the MIPS architecture, is that the shift and compare instructions have been separated from the arithmetical and logical instructions. This allows for some new, 3 register instructions, i.e. operations on 3 registers.

Just like mentioned before, all the instructions will have 4 condition bits, similar to the ARM achitecture. This opens the way to some new types of instructions that will only be executed if the condition mentioned is met. However, there is the option to execute these instructions without checking for any condition.

3.1.2 Instruction format

Inspired by both MIPS and ARM architectures, I designed a special format for each one of the four instruction types. They will be presented next and each field will be described in detail.

3.1.2.1 AL-type instructions

The meaning of each field, as well as the number of bits for each one, will be presented next.

Table 3.1.1: AL-type instruction encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				Source					Target					Dest					Source2					Func				Opc			

Table 3.1: AL-type instruction format

- `Cond : 4 bits` - The condition field. As mentioned before, this field opens up the posibility that the instruction will be execute if certain conditions are met. The encoding of those conditions will be presented in the next section.
- `Source : 5 bits` - The field for the source register.
- `Target : 5 bits` - The field for the target register.
- `Dest : 5 bits` - The field for the destionation register. The result will be stored here.
- `Source2 : 5 bits` - The field for another source register. As mentioned before, this allows for the existence of some interesting 3-term operations.
- `Func : 4 bits` - The function field. This field selects the arithmetical or logical function to be executed.
- `Opc : 4 bits` - The operation code field. This field selects the family of arithmetical and logical functions. As such, this field will always be set to `0000` for any of the arithmetical or logical functions.

3.1.2.2 SC-type instructions

The meaning of each field, as well as the number of bits for each one, will be presented next.

Table 3.1.2: SC-type instruction encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				Source					Target					0	0	0	0	sd		sa				Func				Opc			

Table 3.2: SC-type instruction format

- `Cond : 4 bits` - This is the condition field. As mentioned before, this field opens up the posibility that the instruction will be execute if certain conditions are met. The encoding of those conditions will be presented in the next section.
- `Source : 5 bits` - The field for the source register.
- `Target : 5 bits` - The field for the target register.
- `sd : 2 bits` - The shift direction field. This allows for numbers to be shifted left or right and arithmetically or logically.
- `sa : 4 bits` - The shift amount field. This allows for numbers to be shifted by at most a whole word, *i.e. 2 bytes*.
- `Func : 4 bits` - The function field. This field selects the shift or compare function to be executed.
- `Opc : 4 bits` - The operation code field. This field selects the family of shift and compare functions. As such, this field will always be set to `0001` for any of the shift and compare functions.

Remark 3.1.1: Splitting the R-type functions

As mentioned before, I chose to split the shift and compare functions from the arithmetical and logical functions in order to have the posibility of implementing some 3-term operations and add some versatility to the CPU.

Remark 3.1.2: The empty fields

Another aspect worth mentioning is that the fields from 14 up to 17 will not be interpreted in this instruction and thus they can be set to `0`.

Remark 3.1.3: The shift operation

The shift operation is chosen based on the value of the `sd` field. As such, we distinguish between:

- `00`: Shift left logical
- `01`: Shift right logical
- `10`: Shift left arithmetical
- `11`: Shift right arithmetical

When writing code in assemble language, we distinguish between these instructions by their name:

- `sll`: Shift left logical
- `srl`: Shift right logical
- `sla`: Shift left arithmetical
- `sra`: Shift right arithmetical

3.1.2.3 I-type instructions

The meaning of each field, as well as the number of bits for each one, will be presented next.

Table 3.1.3: I-type instructions encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				Source					Target					Immediate																Opc		

Table 3.3: I-type instruction format

- `Cond : 4 bits` - This is the condition field. As mentioned before, this field opens up the posibility that the instruction will be execute if certain conditions are met. The encoding of those conditions will be presented in the next section.
- `Source : 5 bits` - The field for the source register.
- `Target : 5 bits` - The field for the target register.
- `Immediate : 14 bits` - The field for the immediate value. This will be extended to 32 bits during the *Instrucion Decode* phase.
- `Opc : 4 bits` - The operation code field. This field selects between the different I-type instructions.

3.1.2.4 J-type instructions

The meaning of each field, as well as the number of bits for each one, will be presented next.

Table 3.1.4: J-type instruction encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cond				L	Address																							Opc			

Table 3.4: J-type instruction format

- `Cond : 4 bits` - This is the condition field. As mentioned before, this field opens up the posibility that the instruction will be execute if certain conditions are met. The encoding of those conditions will be presented in the next section.
- `L : 1 bit` - The link field. If this bit is set to one, then the jump operation will be executed as a jump and link operation.
- `Address : 23 bits` - The address field. This contains the address which the jump operation will place in the Program Counter.
- `Opc : 4 bits` - The operation code field. This field selects between the different J-type instructions.

3.1.3 The list of instructions

In what follows, all the instructions that will be designed for this CPU will be presented. However, they will be separated in four categories, namely the mandatory 10 instructions (also called the basic ones) and three other sets of instructions which will be implemented if the time will allow such a task.

3.1.3.1 The first 10 instructions

It comes as no surprise that we need to have two instructions that operate on the memory, specifically a LOAD and a STORE instruction. Next, a JUMP instruction is required as well as a COMPARE instruction to allow for some basic loop functionalities of the assembly code.

The rest of the 6 instruction are:

- The ADD instruction
- The AND instruction
- The SFT instruction
- The XOR instruction
- The SUB instruction
- The MUL instruction

In addition to these ten instructions, I will design and implement an 11th instruction, namely the ADDI instruction, in order to ease the loading of immediate values into the Register File.

The use of these instructions together with their RTL (Register Transfer Level) abstracts will pe presented in the next chapter.

3.1.3.2 Aditional instructions

These additional instructions are linked to their design, mainly because I chose to approach it from the ARM's architecture perspective. The condition bits allow for some new and interesting instructions. Some of these instructions may be:

- The LOADCS instruction - *load if carry is set*
- The ADDZ instruction - *add if the zero bit is set*
- The XORGT instruction - *xor if the zero bit is clear and the negative bit equals the overflow bit*
- The SUBV instruction - *subtract if the overflow bit is set*

However, this kind instructions will be implemented if the first 10 instrucions are implemented successfully and work well.

In the same manner, each arithmetical and logical instruction will have a variant in which it will be able to perform the same operation but on 3 operands instead of only 2. In order to distinguish between those operations, they will have a T appended to the end of the instruction (eg. ADDT, SUBT, MULT, XORT, ANDT, a.s.o.).

3.1.3.3 One CISC instruction

In addition to designing the basic set of instruction, the conditional instructions and the 3 operands instruction, it would be useful to design a complex operation (i.e. a CISC one). This operation will be the LOOP operation. It would work in a similar manner to the one already existing in Assembly Language, by decremeneting a value stored in a register, checking the zero flag and then jumping to the begining of the loop or continuing with the next instruction. However, as stated before, this is going to be implementing only if the time allows it, since it's above the scope of the project.

3.2 The condition field

3.2.1 What is the condition field?

As presented before, each instruction will be preceded by 4 bits which are called *the condition bits*. This results in 16 possible conditions for each instruction. Also, if all of the four bits are '0', the instruction is going to be executed, regardless of the flags in the CPSR (Current Program Status Register).

Also, it is worth mentioning at this point that each instruction will affect the CPSR, mainly the flags of this register. This is done in order to propagate the effects of an instruction to the next one. This aspect will be presented more thoroughly in the *Design* chapter, when presenting the design of the *Register File*.

3.2.2 Condition codes

Table 3.2.1: The condition codes

Code	Suffix	Flags	Meaning
0000	-	ignored	always execute
0001	EQ	Z set	equal
0010	NE	Z clear	not equal
0011	CS	C set	unsigned higher or same
0100	CC	C clear	unsigned lower
0101	MI	N set	negative
0110	PL	N clear	positive or zero
0111	VS	V set	overflow
1000	VC	V clear	no overflow
1001	HI	C set and Z clear	unsigned higher
1010	LS	C clear or Z set	unsigned lower or same
1011	GE	N equals V	greater or equal
1100	LT	N not equal to V	less than
1101	GT	Z clear AND (N equals V)	greater than
1110	LE	Z set OR (N not equal to V)	less than or equal
1111	T	ignored	3 operands

Table 3.5: Condition code summary
[4]

3.2.3 Usage and interpretation

In assembly language, the differentiation between condtional instructions and normal instructions (i.e. instructions that are always executed) is done by adding the suffix indicated in the table.

In machine language (a.k.a. binary representation of the instructions) the differentiation is done at the *Instruction Decode* step, when the condition bits dictate what flags of the CPSR will be checked.

Chapter 4

Design

4.1 RTL Abstract

4.1.1 Preview

For this project, I deicded to design four instruction sets. They are, as follows: The Lite Instruction Set, The Triple-Operand Instruction Set, The Conditional Instruction Set, The CISC Instruction Set. Their names indicate the type of instructions of which they are composed.

The Lite Instruction Set is the set containing the 10 (with one shift instruction divided into 4) basic instructions requiered by the project.

The Triple-Operand Instruction Set is the set containing operations that can be performed on three operands.

The Conditional Instruction Set is the set containing the operations that will check the flags before being executed.

The CISC Instruction Set is the set containing only one instruction, mainly the LOOP instruction presented before.

These instruction sets grow in complexity in the same order in which they are presented. In this regard, implementing anyone of them will be done if and only if the previous instruction sets were implemented.

In what follows, each of these will be presented and explained.

4.1.2 The Lite Instruction Set

Table 4.1.1: LIS

Instruction	RTL Abstract	Program Counter
ADD $rs, $rt, $rd	$RF[rd] \leftarrow RF[rs] + RF[rt]$	$PC \leftarrow PC + 1$
SUB $rs, $rt, $rd	$RF[rd] \leftarrow RF[rs] - RF[rt]$	$PC \leftarrow PC + 1$
XOR $rs, $rt, $rd	$RF[rd] \leftarrow RF[rs] \oplus RF[rt]$	$PC \leftarrow PC + 1$
AND $rs, $rt, $rd	$RF[rd] \leftarrow RF[rs] \wedge RF[rt]$	$PC \leftarrow PC + 1$
SLL $rs, $rt, n	$RF[rt] \leftarrow RF[rs] << n$	$PC \leftarrow PC + 1$
SRL $rs, $rt, n	$RF[rt] \leftarrow RF[rs] >> n$	$PC \leftarrow PC + 1$
SLA $rs, $rt, n	$RF[rt] \leftarrow RF[rs] << n \& LSB$	$PC \leftarrow PC + 1$
SRA $rs, $rt, n	$RF[rt] \leftarrow MSB \& RF[rs] >> n$	$PC \leftarrow PC + 1$
MUL $rs, $rt	$RF[1] \leftarrow \text{LOW}[RF[rs] * RF[rt]]$ $RF[2] \leftarrow \text{HIGH}[RF[rs] * RF[rt]]$	$PC \leftarrow PC + 1$
JMP imm	-	$PC \leftarrow PC + 1 + Ext(imm)$
COMP $rs, $rt	$rez \leftarrow RF[rs] - RF[rt]$ $rez == 0 \Rightarrow Z \leftarrow 1$ $rez < 0 \Rightarrow N \leftarrow 0, V \leftarrow 1$ $rez > 0 \Rightarrow Z \leftarrow 0, N \leftarrow V \leftarrow 0$	$PC \leftarrow PC + 1$
LOAD imm($rs), $rt	$RF[rt] \leftarrow M[RF[rs] + Ext(imm)$	$PC \leftarrow PC + 1$
STORE imm($rs), $rt	$M[RF[rs] + Ext(imm) \leftarrow RF[rt]$	$PC \leftarrow PC + 1$
ADDI $rs, $rt, imm	$RF[rt] \leftarrow RF[rs] + Ext(imm)$	$PC \leftarrow PC + 1$

Table 4.1: Lite Instruction Set - RTL Abstract

One mention needs to be made about the COMP instruction. In order to compare the two arguments of the function, the instruction performs a subtraction, without storying the result anywhere. After performing the subtraction, the sign bit is checked and the flags of the CPSR are set accordingly.

Another mention should be made about the MUL instruction. When multypling two 32-bit numbers, the result will be of 64-bit length. Due to this fact, the lower 32 bits of the result will be stored on the second register and the higher 32 bits of the result will be stored on the third register of the register file (i.e. `RF[1]` and `RF[2]`).

4.1.3 The Triple-Operand Instruction Set

Table 4.1.2: TOIS

Instruction	RTL Abstract
ADDT $rs1, $rs2, $rt, $rd	$RF[rd] \leftarrow RF[rs1] + RF[rs2] + RF[rt]$
SUBT $rs1, $rs2, $rt, $rd	$RF[rd] \leftarrow RF[rs1] - RF[rs2] - RF[rt]$
XORT $rs1, $rs2, $rt, $rd	$RF[rd] \leftarrow RF[rs1] \oplus RF[rs2] \oplus RF[rt]$
ANDT $rs1, $rs2, $rt, $rd	$RF[rd] \leftarrow RF[rs1] \wedge RF[rs2] \wedge RF[rt]$
ORT $rs1, $rs2, $rt, $rd	$RF[rd] \leftarrow RF[rs1] \vee RF[rs2] \vee RF[rt]$
NANDT $rs1, $rs2, $rt, $rd	$RF[rd] \leftarrow \neg(RF[rs1] \wedge RF[rs2] \wedge RF[rt])$
NORT $rs1, $rs2, $rt, $rd	$RF[rd] \leftarrow \neg(RF[rs1] \vee RF[rs2] \vee RF[rt])$
MULT $rs1, $rs2, $rt, $rd	$RF[1] \leftarrow \text{LOW}[RF[rs] * RF[rt]]$ $RF[rd] \leftarrow \text{MID}[RF[rs] * RF[rt]]$ $RF[2] \leftarrow \text{HIGH}[RF[rs] * RF[rt]]$

Table 4.2: Triple-Operand Instruction Set - RTL Abstract

All the instructions listed above are self explanatory and are made clear enough by their RTL Abstract. They perform the same operation (as well as some more) as the ones presented previously.

In the same manner as before, the MUL instruction now takes 3 operands. Therefore, the result will have at most 96 bits and will be storred on three registers. I chose to design this instruction in such a way that the middle 32 bits of the result are storred in a register specified by the programmer via the *$rd* parameter. The high and low parts of the result are storred in the same manner as before.

4.1.4 The Conditional Instruction Set

Table 4.1.3: CIS

Instruction	RTL Abstract	Program Counter
ADDEQ $rs, $rt, $rd	$RF[rd] \leftarrow RF[rs] + RF[rt]$	$PC \leftarrow PC + 1$
ADDNE $rs, $rt, $rd	$RF[rd] \leftarrow RF[rs] + RF[rt]$	$PC \leftarrow PC + 1$
ADDGT $rs, $rt, $rd	$RF[rd] \leftarrow RF[rs] + RF[rt]$	$PC \leftarrow PC + 1$
ADDLT $rs, $rt, $rd	$RF[rd] \leftarrow RF[rs] + RF[rt]$	$PC \leftarrow PC + 1$
ADDGE $rs, $rt, $rd	$RF[rd] \leftarrow RF[rs] + RF[rt]$	$PC \leftarrow PC + 1$
ADDLE $rs, $rt, $rd	$RF[rd] \leftarrow RF[rs] + RF[rt]$	$PC \leftarrow PC + 1$
JMPEQ imm	-	$PC \leftarrow PC + 1 + Ext(imm)$
JMPNE imm	-	$PC \leftarrow PC + 1 + Ext(imm)$
JMPGT imm	-	$PC \leftarrow PC + 1 + Ext(imm)$
JMPLT imm	-	$PC \leftarrow PC + 1 + Ext(imm)$
JMPGE imm	-	$PC \leftarrow PC + 1 + Ext(imm)$
JMPLE imm	-	$PC \leftarrow PC + 1 + Ext(imm)$

Table 4.3: Conditional Instruction Set - RTL Abstract

Remark 4.1.1: The link option

Each one of the JUMP instructions has another variant with -L appended to the end. This means that the jump to another address will be performed with the LINK action. The llinking is done by storing the next address generated by the Instruction Fetch Unit in the first register of the register file (i.e. `RF[0]`).

Each of these functions are self explanatory and the meaning of the suffixes used for each of them was presented before.

4.1.5 The CISC Instruction Set

Table 4.1.4: CISCIS

Instruction	RTL Abstract	Program Counter
LOOP tag	$RF[0] \leftarrow RF[0] - 1$	$if\ RF[0] > 0\ then\ PC \leftarrow Ext(tag) + 4$ $else\ PC \leftarrow PC + 1$

Table 4.4: CISC Instruction Set - RTL Abstract

The LOOP instruction requieres the programmer to store the number of times the code should loop back to the TAG address in the first register file, i.e. *RF[0]*. The CPU will decrement the value of this register and if it reaches 0, the CPU will NOT loop back to the TAG address.

This behaviour is similar to the one of the LOOP instruction from Assembly Language.

4.2 Instruction encoding

4.2.1 Preview

The instructions presented before should be translated in a language that the microprocessor can understand. Such a language is the Binary Language. In other words, each instruction should be translated into a series of ones and zeros.

In what follows, all the instructions presented before will be listed together with their encodings.

4.2.2 Notations

In order to make the reading of these instructions easier, a series of notations will be used. These notations are:

List 4.2.1: Notations

- _ (underscore) : will be used to separate the fields of the instructions
- X : will be used to indicate that those bits won't be read for that instruction
- s : will be used to indicate the address of the source register
- t : will be used to indicate the address of the target register
- d : will be used to indicate the address of the destination register
- S (capital s) : will be used to indicate the address of the second source register
- i : will be used to indicate the binary representation of the immediate value
- a : will be used to indicate the binary representation of the shift ammount

4.2.3 The Lite Instruction Set

Table 4.2.1: Lite instructions encoding

Instruction	Encoding
ADD	0000_sssss_ttttt_ddddd_XXXXX_0000_0000
SUB	0000_sssss_ttttt_ddddd_XXXXX_0001_0000
XOR	0000_sssss_ttttt_ddddd_XXXXX_0010_0000
AND	0000_sssss_ttttt_ddddd_XXXXX_0011_0000
MUL	0000_sssss_ttttt_XXXXX_XXXXX_0100_0000
SLL	0000_sssss_ttttt_0000_00_aaaa_0000_0001
SRL	0000_sssss_ttttt_0000_01_aaaa_0000_0001
SLA	0000_sssss_ttttt_0000_10_aaaa_0000_0001
SRA	0000_sssss_ttttt_0000_11_aaaa_0000_0001
CMP	0000_sssss_ttttt_000000_XXXX_0001_0001
LOAD	0000_sssss_ttttt_iiiiiiiiiiiii_0010
STORE	0000_sssss_ttttt_iiiiiiiiiiiii_0011
ADDI	0000_sssss_ttttt_iiiiiiiiiiiii_0101
JMP	0000_0_iiiiiiiiiiiiiiiiiiiii_0100
JMPL	0000_1_iiiiiiiiiiiiiiiiiiiii_0100

Table 4.5: Lite Instruction Set - encoding

4.2.4 The Triple-Operand Instruction Set

Table 4.2.2: Triple-Operand instructions encoding

Instruction	Encoding
ADDT	1111_sssss_ttttt_ddddd_SSSSS_0000_0000
SUBT	1111_sssss_ttttt_ddddd_SSSSS_0001_0000
XORT	1111_sssss_ttttt_ddddd_SSSSS_0010_0000
ANDT	1111_sssss_ttttt_ddddd_SSSSS_0011_0000
MULT	1111_sssss_ttttt_ddddd_SSSSS_0100_0000
ORT	1111_sssss_ttttt_ddddd_SSSSS_0101_0000
NANDT	1111_sssss_ttttt_ddddd_SSSSS_0110_0000
NORT	1111_sssss_ttttt_ddddd_SSSSS_0111_0000

Table 4.6: Triple-Operand Instruction Set - encoding

4.2.5 The Conditional Instruction Set

Table 4.2.3: Conditional instructions encoding

Instruction	Encoding
ADDEQ	0001_sssss_ttttt_ddddd_XXXXX_0000_0000
ADDNE	0010_sssss_ttttt_ddddd_XXXXX_0000_0000
ADDGT	1101_sssss_ttttt_ddddd_XXXXX_0000_0000
ADDLT	1100_sssss_ttttt_ddddd_XXXXX_0000_0000
ADDGE	1011_sssss_ttttt_ddddd_XXXXX_0000_0000
ADDLE	1110_sssss_ttttt_ddddd_XXXXX_0000_0000
JMPEQ	0001_0_iiiiiiiiiiiiiiiiiiii_0100
JMPEQL	0001_1_iiiiiiiiiiiiiiiiiiii_0100
JMPNE	0010_0_iiiiiiiiiiiiiiiiiiii_0100
JMPNEL	0010_1_iiiiiiiiiiiiiiiiiiii_0100
JMPGT	1101_0_iiiiiiiiiiiiiiiiiiii_0100
JMPGTL	1101_1_iiiiiiiiiiiiiiiiiiii_0100
JMPLT	1100_0_iiiiiiiiiiiiiiiiiiii_0100
JMPLTL	1100_1_iiiiiiiiiiiiiiiiiiii_0100
JMPGE	1011_0_iiiiiiiiiiiiiiiiiiii_0100
JMPGEL	1011_1_iiiiiiiiiiiiiiiiiiii_0100
JMPLE	1110_0_iiiiiiiiiiiiiiiiiiii_0100
JMPLEL	1110_1_iiiiiiiiiiiiiiiiiiii_0100

Table 4.7: Conditional Instruction Set - encoding

4.2.6 The CISC Instruction Set

Table 4.2.4: CISC instructions encoding

Instruction	Encoding
LOOP	0000_0_iiiiiiiiiiiiiiiiiiii_0101

Table 4.8: CISC Instruction Set - encoding

4.3 Function encoding

In order to differentiate between functions which have the same `Operation code`, I designed the `Function` field, which is composed of the bits 4 to 7. Here, function means any arithmetical, logical, shift or comparison operation.

Table 4.3.1: Function codes

Function	Code
Addition	0000
Subtraction	0001
Exclusive OR	0010
And	0011
Multiplication	0100
Or	0101
Nand	0110
Nor	0111
Shift Left Logical	0000
Shift Right Logical	0000
Shift Left Arithmetic	0000
Shift Right Arithmetic	0000
Compare	0001

Table 4.9: Codes for each function

Remark 4.3.1: Overlaping

Even if it seems like the Shift Left Logical, Shift Right Logical, Shift Left Arithmetic, Shift Right Arithmetic and Compare operations are encoded in the same manner as the Addition and Subtraction operations, the difference is done in the `Operation code` field.

Remark 4.3.2: Identical function codes

These function codes are the same, regardless of the instruction set in which each instruction is found.

4.4 Operation encoding

In order to differentiate between operations, I have designed the `Operation code` field, which is composed of the bits from 0 to 4. These operation codes are going to be presented next.

Table 4.4.1: Operation codes

Operation	Code
AL-type instructions	0000
SC-type instructions	0001
Load	0010
Store	0011
Addi	0101
Jmp	0100

Table 4.10: Codes for each operation

4.5 Block diagrams

4.5.1 Top Level Block Diagram

In the figure below, the block diagram of the whole CPU (i.e. top level) is presented.

Figure 4.5.1: Top Level Block Diagram

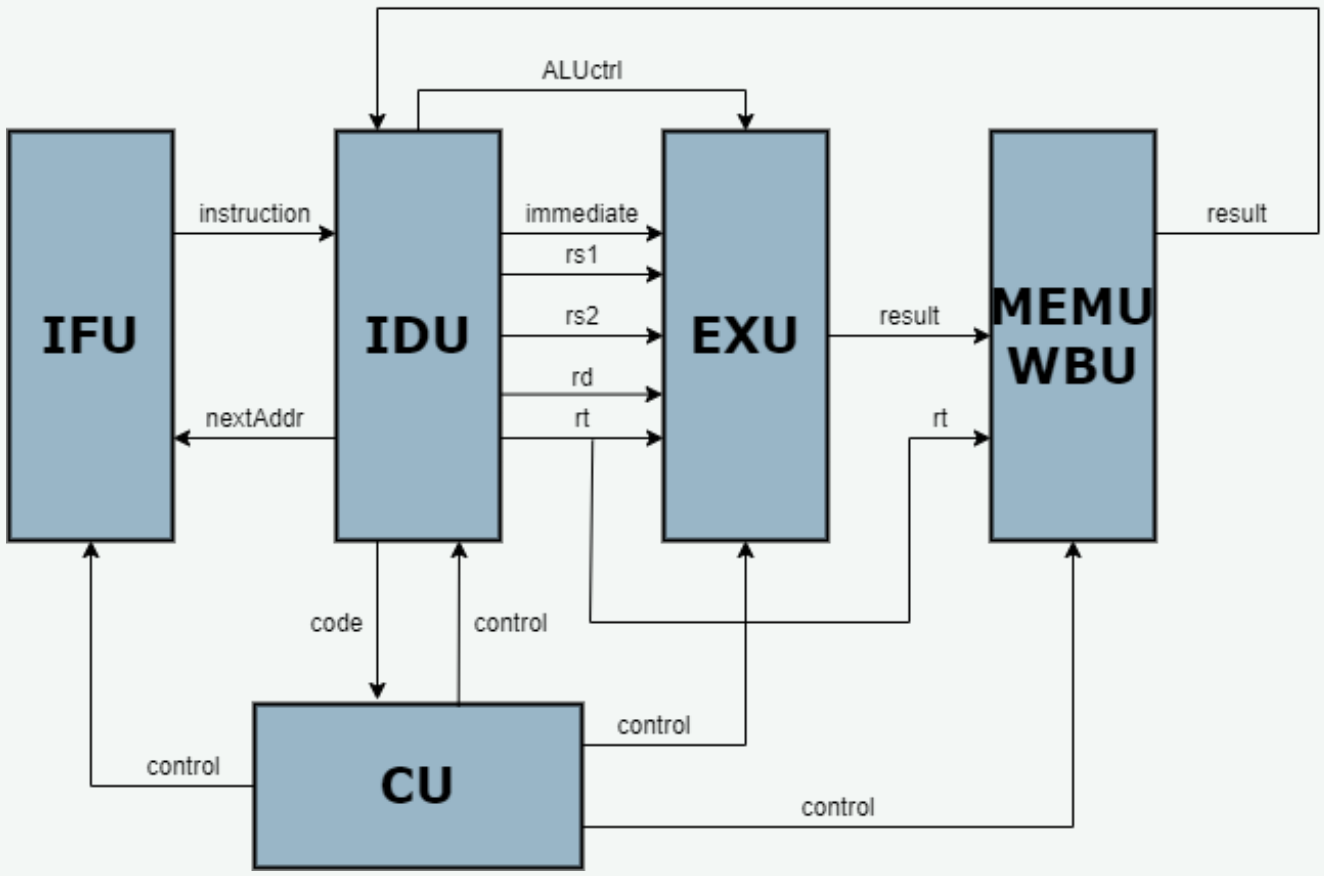

Figure 4.1: Top Level Block Diagram

Data flow The data flow can be easily followed from left to right. The instruction is passed from the *Instruction Fetch Unit* to the *Instruction Decode Unit*, which will decode the instruction and pass the operands to the *Execution Unit*. Here, the actual instruction is performed and the result is passed to the *Memory Unit* which also includes the *Write Back Unit*. The result is stored in memory (if necessary) or is sent back to the *Instruction Decode Unit* to be stored in one of the registers in the *Register File*.

Remark 4.5.1: Components

Each of the components presented here will have its block diagram presented in the next sections.

4.5.2 Memory

For this CPU, I chose to use 3 types of memory: ROM, RAM and a Register File. Each one of them will be presented next.

4.5.2.1 Register File

The *Register File* is used to store operands for the different instructions. Since I want to have the possibility to implement the Triple-Operand Instruction Set, the Register File has 3 input `Addresses` and 3 `Data Out` ports for future use.

In the figure bellow, the block diagram for the Register File is presented.

Figure 4.5.2: Register File Block Diagram

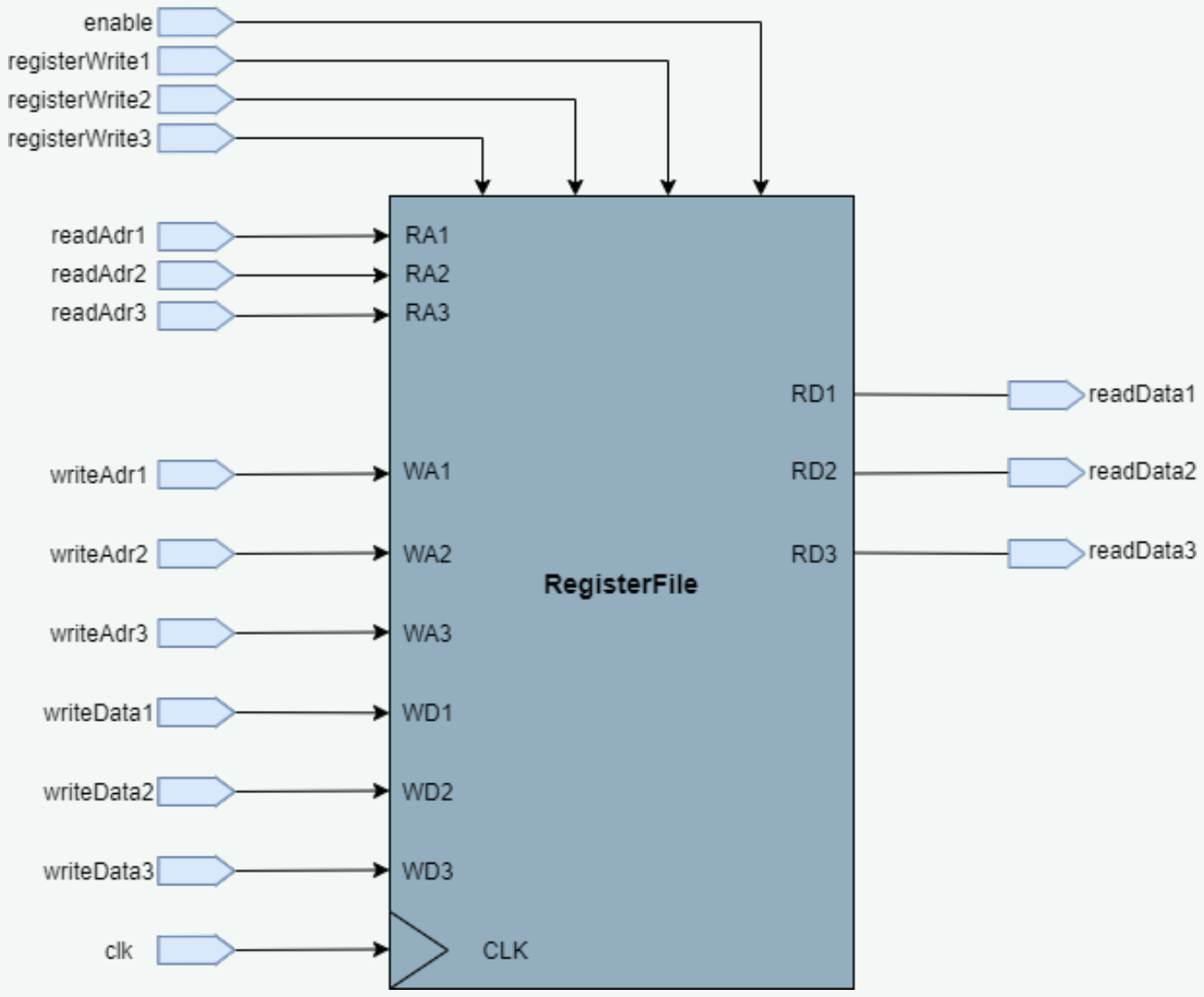

Figure 4.2: Register File Block Diagram

The block diagram This block diagram matches the entity implemented in the `RegisterFile.vhd` file, or in the *Register File Entity* listing.

Remark 4.5.2: Size of input/output

It's not specified on the block diagram, but the sizes of the read and write addresses is 5 bits and the sizes of the read and write data is 32 bits.

Since the registers can be addressed on 5 bits, this leaves room for a total of 32 registers, out of which 31 can be used by the programmer. The first register (i.e. `RF[0]`) will keep the return address for the JUMP AND LINK instructions.

As it can be seen from the diagram, the register file has 3 read ports and 3 write ports. The three read ports were designed in order to accomodate the triple operdand instructions. Moreover, the 3 write ports, along with the 3 write enabling signals, are used to store the result of the multiplication of two operands and three operands or to store the address for the jump and link instructions.

Remark 4.5.3: The enable signal

In the previous figure, the register file was presented with an enable signal. This signal enables the writing and reading in and from the register file for the conditional instructions. In other words, if the contents of the CPSR do not equal the condition bits of the instruction, then the register file will not be enabled for writing, nor reading.

4.5.2.2 ROM

The *ROM* is used to store the program that will be executed by the CPU. It simply takes in an `Address` and returns the `Data` stored at that address. Each instruction will be encoded (as presented before) on 32 bits.

In the figure bellow, the block diagram for the READ ONLY MEMORY is presented.

Figure 4.5.3: Read Only Memory Block Diagram

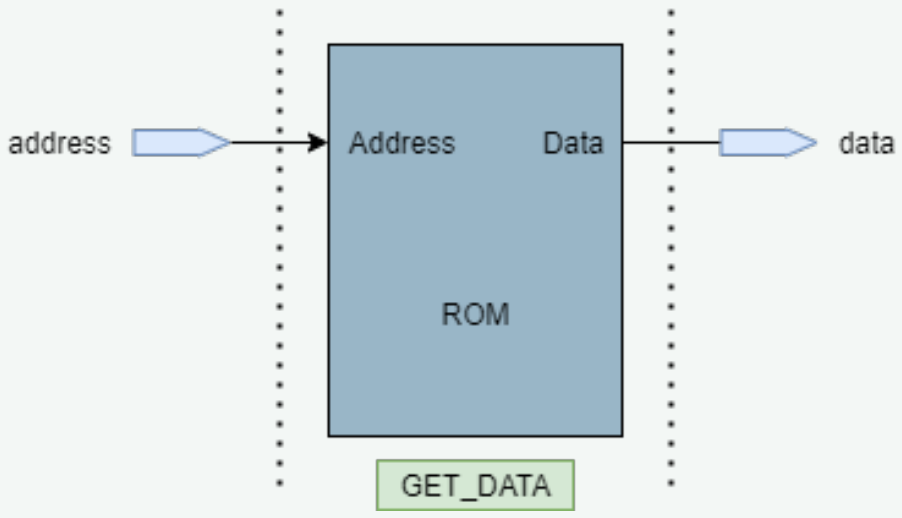

Figure 4.3: Read Only Memory Block Diagram

Remark 4.5.4: Size of input/output

It's not specified on the block diagram, but both the `Address` input and the `Data` output are implemented on 32 bits.

The block diagram This block diagram matches the entity implemented in the `ROM.vhd` file, or in the *ROM Entity* listing.

4.5.2.3 RAM

The *RAM* is used to store data outside of the registers. This should be seen as a long-term data storage. The memory is only accessed by the LOAD and STORE instructions.

In the figure bellow, the block diagram for the RANDOM ACCESS MEMORY is presented.

Figure 4.5.4: Random Access Memory Block Diagram

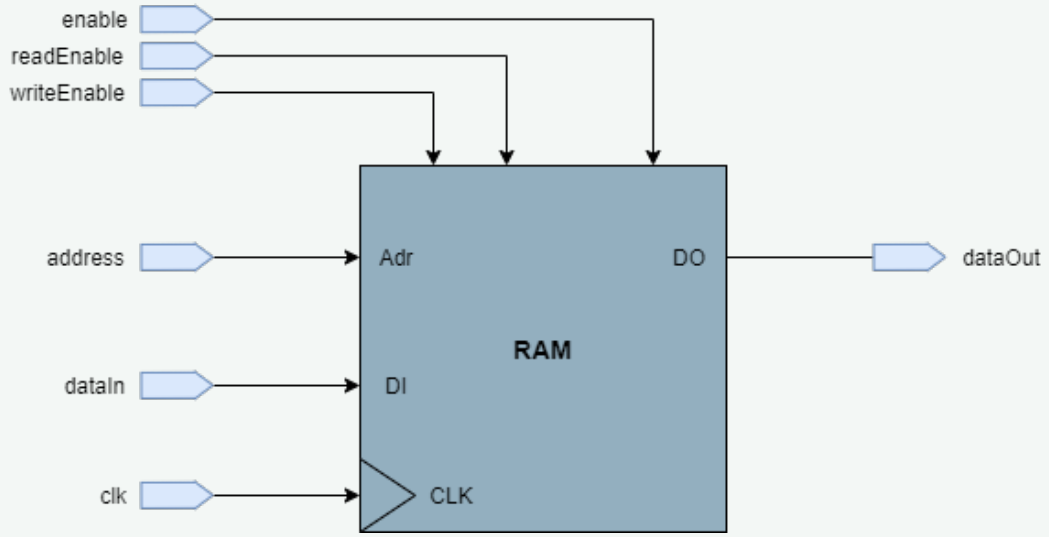

Figure 4.4: Random Access Memory Block Diagram

The block diagram This block diagram matches the entity implemented in the `RAM.vhd` file, or in the *RAM Entity* listing.

Remark 4.5.5: The enable signal

As it can be seen in the figure, the memory has an aditional enable signal along with the `readEnable` and `writeEnable`. This signal acts in a similar manner to the one from the register file. It enables the reading and writing from and into the memory only if the contents of the CPSR match the condition field of the instruction.

4.5.3 Miscellaneous

4.5.3.1 Debouncer

The debouncer is a circuit which ensures that a stable signal is propagated from the buttons of the *Basys3 FPGA Board* to the other components of the CPU.

When pressing a button, the mechanical parts make it so that the signal does not transition between '0' and '1' correctly, i.e. it will oscilate multiple times between these values. To overcome this, we use a DEBOUNCER.

In the figure below, the block diagram for the DEBOUNCER is presented.

Figure 4.5.5: Debouncer Block Diagram

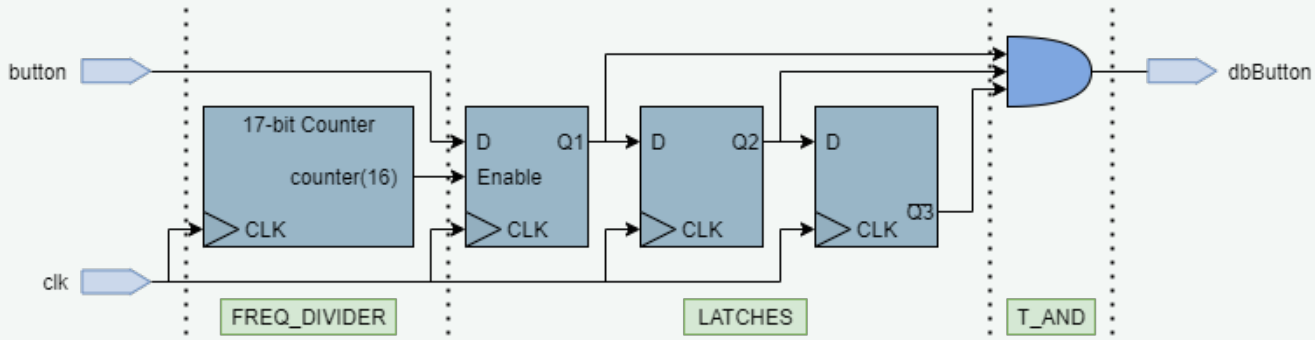

Figure 4.5: Debouncer Block Diagram

The data flow can be easily followed from left to right. Each level is denoted in the same manner as the corresponding processes in the `Debouncer.vhd` file, or in the *Debouncer Architecture* listing.

The *Frequency Divider* is nothing more than a 17-bit counter, which takes the clock signal from the board, at 100MHz and converts it into a signal with the same duty-cycle, but a frequency of about 763Hz. This, in turn, enables the first *D FlipFlop*. These flip-flops, together with the *AND Gate*, ensure that the signal from the button is transmitted forward, only if it remained stable for 3 clock cycles after the counter has finished counting.

4.5.3.2 Seven Segment Display Controller

The seven segment display controller is a circuit used to control the *7Seg Display* of the *Bassy3 FPGA Board*. This display is composed out of 4 digits. Each digit has one corresponding anode, which sums up to a total of 4 anodes.

In order to minimize the number of pins on the FPGA Board, the 7Seg Display has a common cathode configuration, which means that there are only 8 cathodes instead of the expected 32. In consequence, we can display only one digit at a time. However, through the circuit bellow, we can develop a workaround.

In the figure bellow, the block diagram for the SEVEN SEGMENT DISPLAY CONTROLLER is presented.

Figure 4.5.6: Seven Segment Display Controller Block Diagram

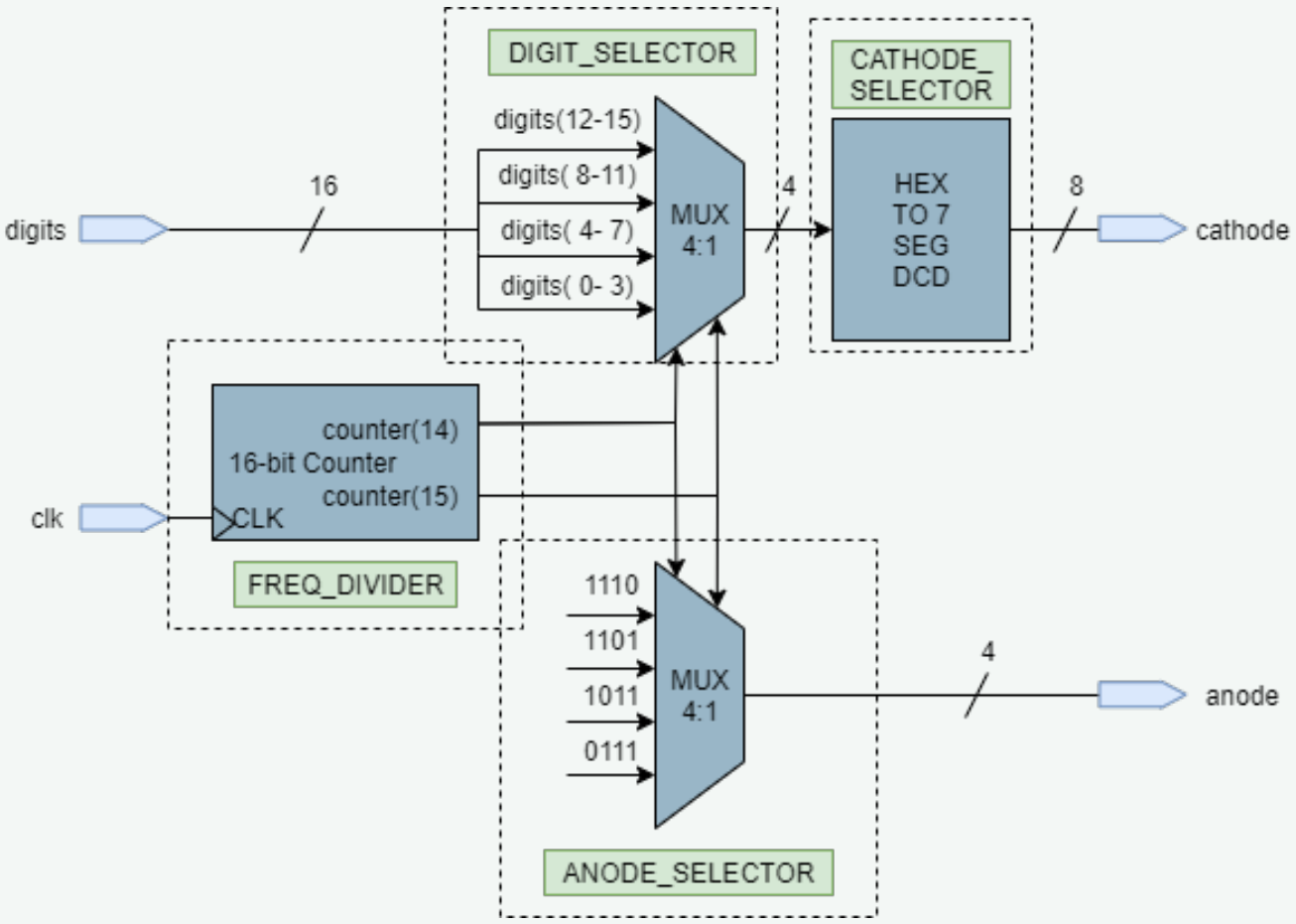

Figure 4.6: Seven Segment Display Controller Block Diagram

The data flow can be easily followed from left to right. Each level is denoted in the same manner as the corresponding processes in the `ControllerSevenSegmentDisplay.vhd` file, or in the *Seven Segment Display Architecture* listing.

The *Frequency Divider* is nothing more than a 16-bit counter, which takes the clock signal from the board, at 100MHz and converts it into a signal with the same duty-cycle, but a frequency of about 1.52kHz. This ensures that each digit is shown individually at small enoguh time intervals that our brain perceives them as being essentially simultaneous.

The last two bits of the counter (i.e. `counter(15 downto 14)`) control the multiplexor which activates the anodes and the multiplexor which renders one of the four digits (each corresponding to a certain anode).

The `digits` signal is encoded on 16 bits. This means that each digit is interpreted in hexadecimal by the decoder at the *Cathode Selector* level.

The overall output of the circuit consists of the `anode` selector, encoded on 4 bits, and the `cathode` selector, encoded on 8 bits.

In the image below, the representation of each digit can be seen.

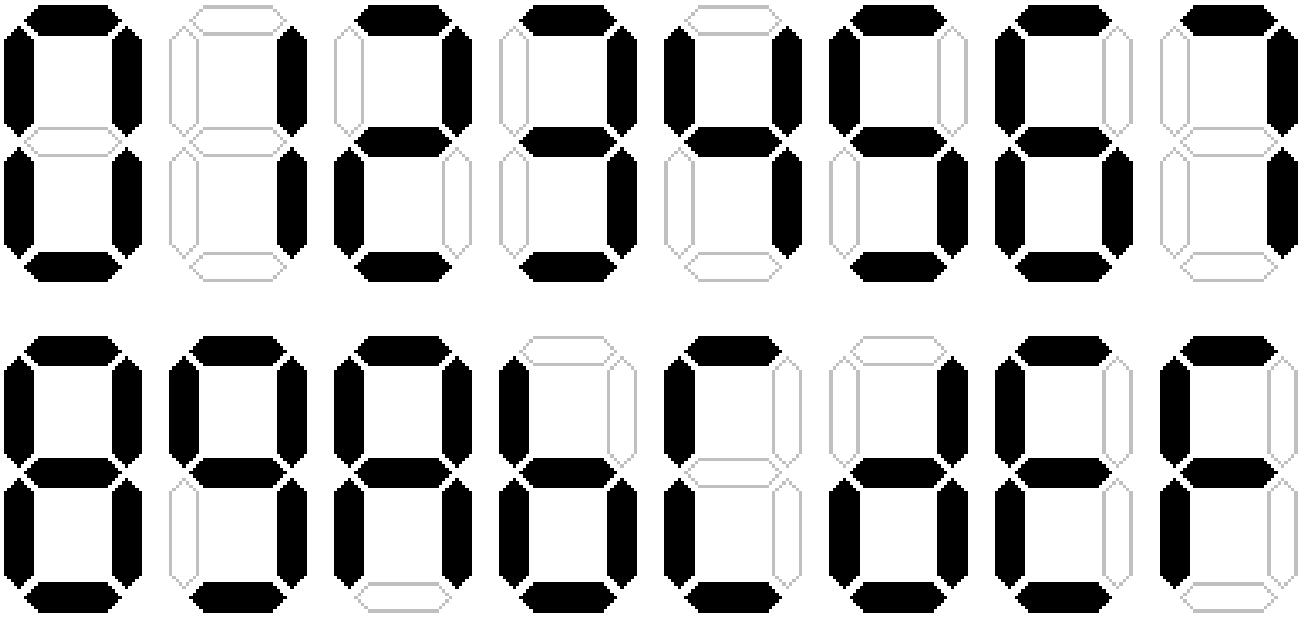

Figure 4.7: Seven Segment Display Digits

Remark 4.5.6: Use of these components

Both the `Debouncer` and the `Seven Segment Display Controller` are used solely for the *Synthesis* part of the testing process.

4.5.3.3 CPSR

The *Current Program State Register*, or CPSR, is a simple 4-bit register which is used to store the flags that are set and checked by the instructions.

In the figure bellow, the block diagram for the Current Program State Register is presented.

Figure 4.5.7: Current Program State Register Diagram

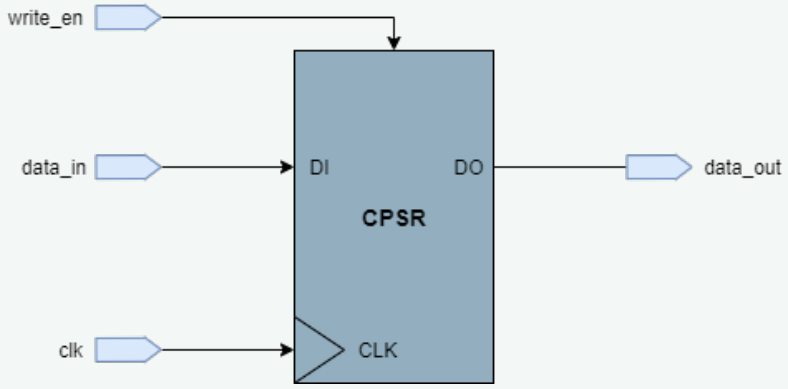

Figure 4.8: Current Program State Register Diagram

Remark 4.5.7: Size of input/output

It's not specified on the block diagram, but the sizes of the `data_in` and `data_out` are 4 bits each.

The structure of the Current Program State Register is presented bellow.

Table 4.5.1: Flags of the CPSR

3	2	1	0
Z	V	C	N

Table 4.11: Flags of the CPSR

The meaning of each flag:

- `Z` - the zero flag
- `V` - the overflow flag
- `C` - the carry flag
- `N` - the negative flag

4.5.3.4 ALU

For this CPU, I had to design my own *Arithmetic Logic Unit*, or ALU, in order to accomodate for the triple operand instructions. As such, in the figure bellow, the block diagram of the ARITHMETIC LOGIC UNIT is presented.

Figure 4.5.8: Arithmetic Logic Unit Diagram

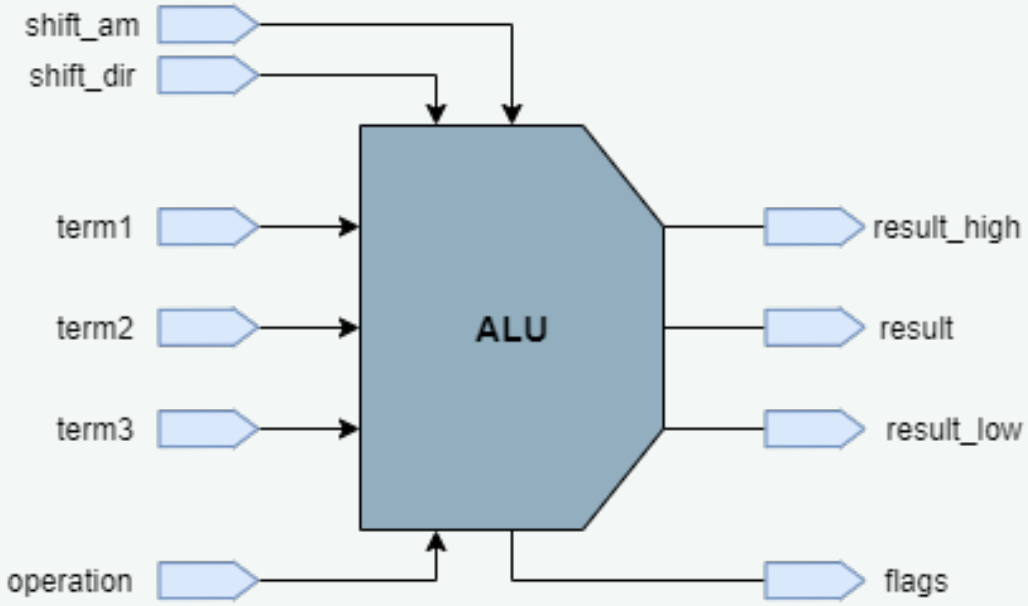

Figure 4.9: Arithmetic Logic Unit Diagram

Remark 4.5.8: Size of input/output

It's not specified on the block diagram, but the three inputs (namely `term1`, `term2` and `term3`) along with the three outputs (`result_high`, `result` and `result_low`) are all sized at 32 bits. The `shift_am` signal is on 4 bits and the `shift_dir` is on 2 bits. Moreover, both the `operation` input and the `flags` output are encoded on 4 bits.

The arithmetic logic unit in the figure above is capable of performing operations on 3 operands instead of just two, as well as setting the flags of the CPSR according to the result obtained. In the same manner, for the MUL instruction, the ALU outputs the result in split in 2 or 3 parts (depending on its size).

More details about the way in which the ALU executes the operations will be given in the *Implementation* chapter.

4.5.3.5 ALU Controller

Due to the fact that multiple instructions need to perform the addition, I had to design a control unit for the ALU. This was also meant to solve the issue of having operations performed on 2 or 3 operands, i.e. the need to distinguish between them, since they have the same operation code and function code.

In the figure bellow, the block diagram for the ALU Controller is presented.

Figure 4.5.9: Arithmetic Logic Unit Controller Diagram

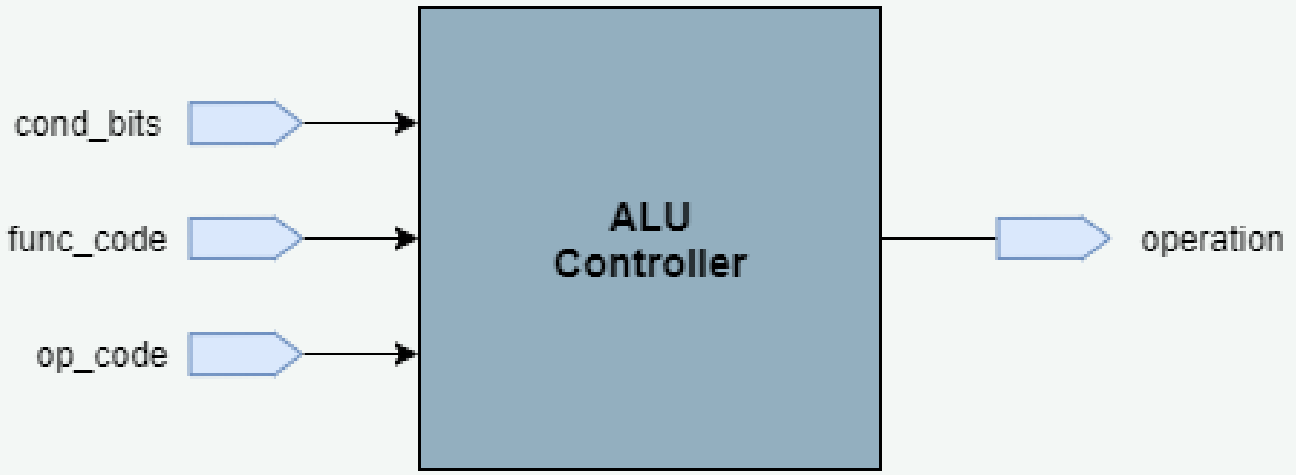

Figure 4.10: Arithmetic Logic Unit Controller Block Diagram

Remark 4.5.9: Size of input/output

It's not specified on the block diagram, but all the inputs and outputs of the ALU controller are on 4 bits.

More details about the implementation of this component and how it works will be presented in the *Implementation* chapter.

4.5.4 Instruction Fetch Unit

The Instruction Fetch component is used to fetch the instructions from the instruction memory. Its inputs are `jump_address, jump_ctrl, reset, enable, clk` and its outputs are `next_address, instruction`.

In the figure below, the block diagram for the Instruction Fetch Unit is presented.

Figure 4.5.10: Instruction Fetch Unit Block Diagram

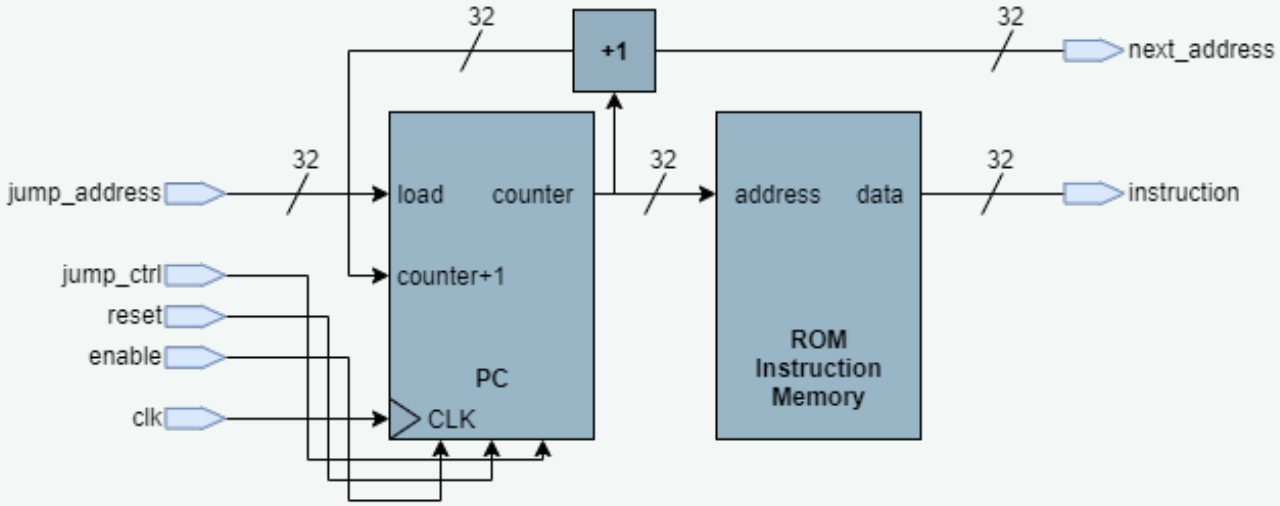

Figure 4.11: Instruction Fetch Unit Block Diagram

As it can be seen in the picture, the Program Counter is nothing more then a register with two load ports and multiple enableing signals. More explanations about this register will be provided in the *Implementation Chapter*.

The Instruction Memory is the ROM presented before. At each 32 bit address, an instruction will be stored. Overall, we can fit a program of 4,294,967,296 instructions at most.

4.5.5 Instruction Decode Unit

In the figure below, the block diagram for the Instruction Decode Unit is presented.

Figure 4.5.11: Instruction Decode Unit Block Diagram

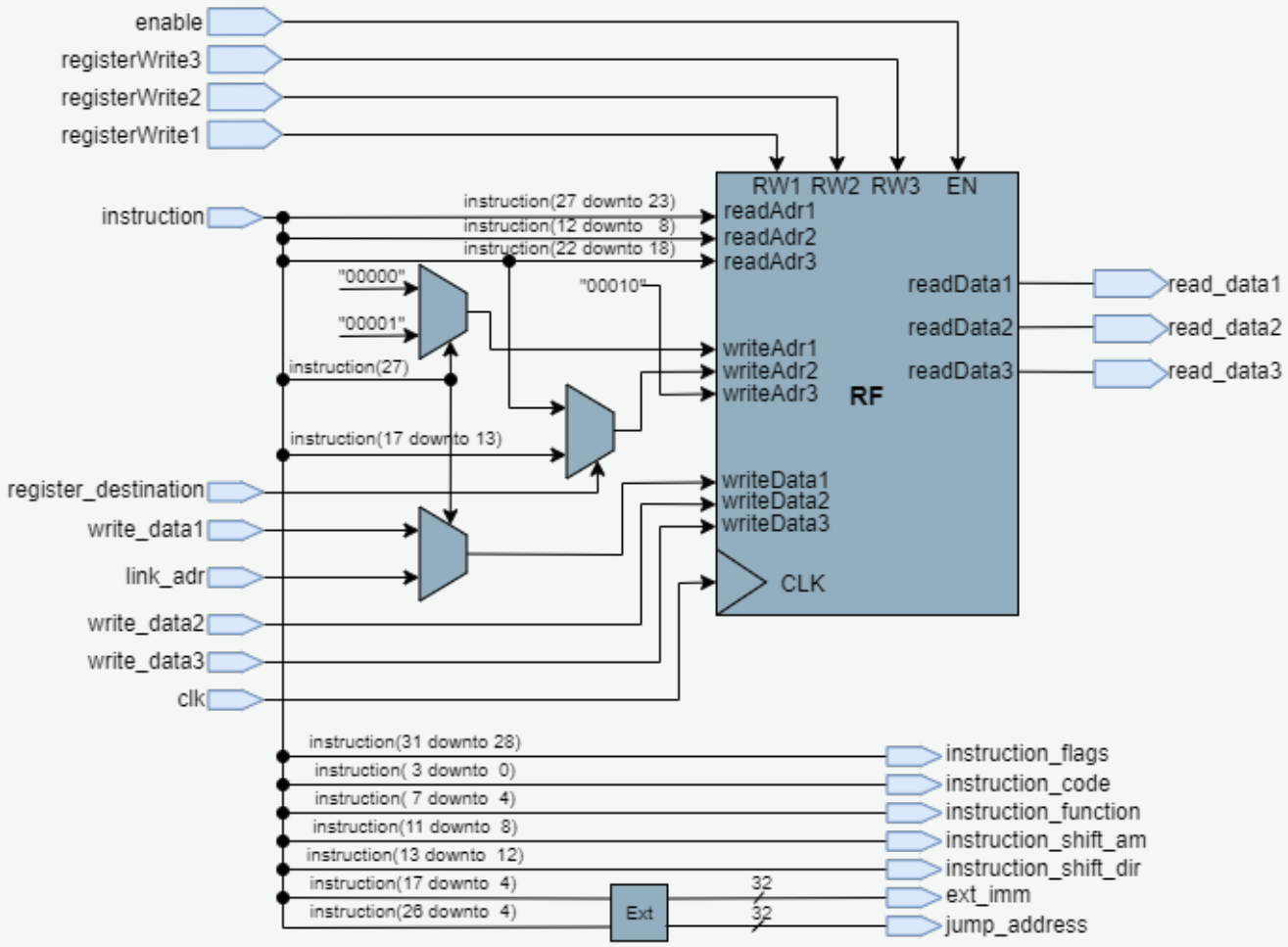

Figure 4.12: Instruction Decode Unit Block Diagram

As it can be seen in the picture, the central component of the IDU is the register file. Multiplexors and control signal have been used to switch between the different data that should be stored in the registers as well as the corresponding addresses.

In a few words, the IDU breaks down the instruction into the appropiate fields and operates the register file. It accomodates the link option for the jump instructions as well as the write back mechanism for the multiplation instruction.

Remark 4.5.10: The link bit

The 28th bit of the instruction (i.e. `instruction(27)`) is the link bit. It is used to switch between the two addresses (namely `"00000"` and `"00001"`) and the two data (namely the high part of the multiplication result or the next instruction address).

In the block diagram of the IDU, the small component denoted `Ext` is the *Extension* unit. It performs the extension of the immediate values to 32 bits, by padding the value with zeroes. Dus, it is a *Zero extension unit*. In future updates of the CPU, this can be turned into a *Sign extenstion unit*. For the scope of this project, the CPU will only operate with unsigned values.

4.5.6 Control Unit

In the figure below, the block diagram for the Control Unit is presented.

Figure 4.5.12: Control Unit Block Diagram

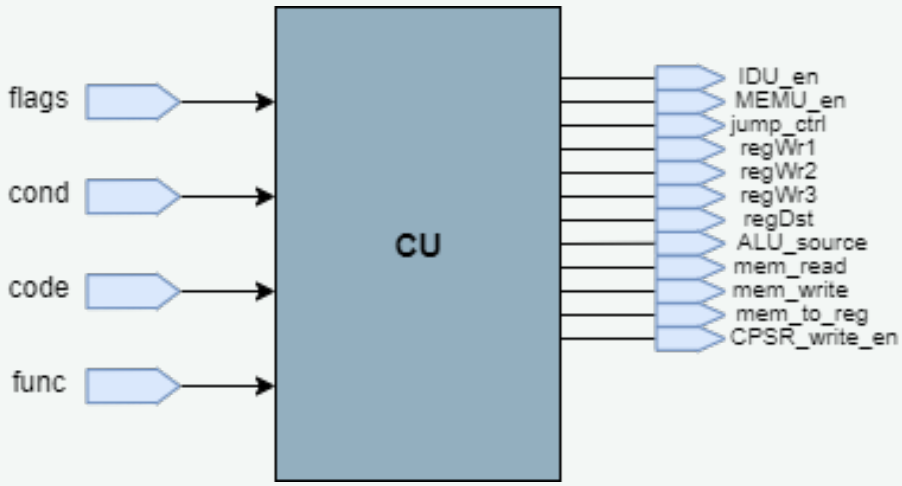

Figure 4.13: Control Unit Block Diagram

The control unit does nothing more than generate the control signals for the whole microprocessor. It does so depending on the flags from the CPSR, the condition bits of the instruction, the operation code of the instruction and the function code of the instruction.

More details about the mechanisms behind this component will be presented in the *Implementation* chapter.

Remark 4.5.11: Size of input/output

It's not specified on the block diagram, but all the inputs of the control unit are on 4 bits and all the outputs are single bit signals.

4.5.7 Execution Unit

In the figure below, the block diagram for the Execution Unit is presented.

Figure 4.5.13: Execution Unit Block Diagram

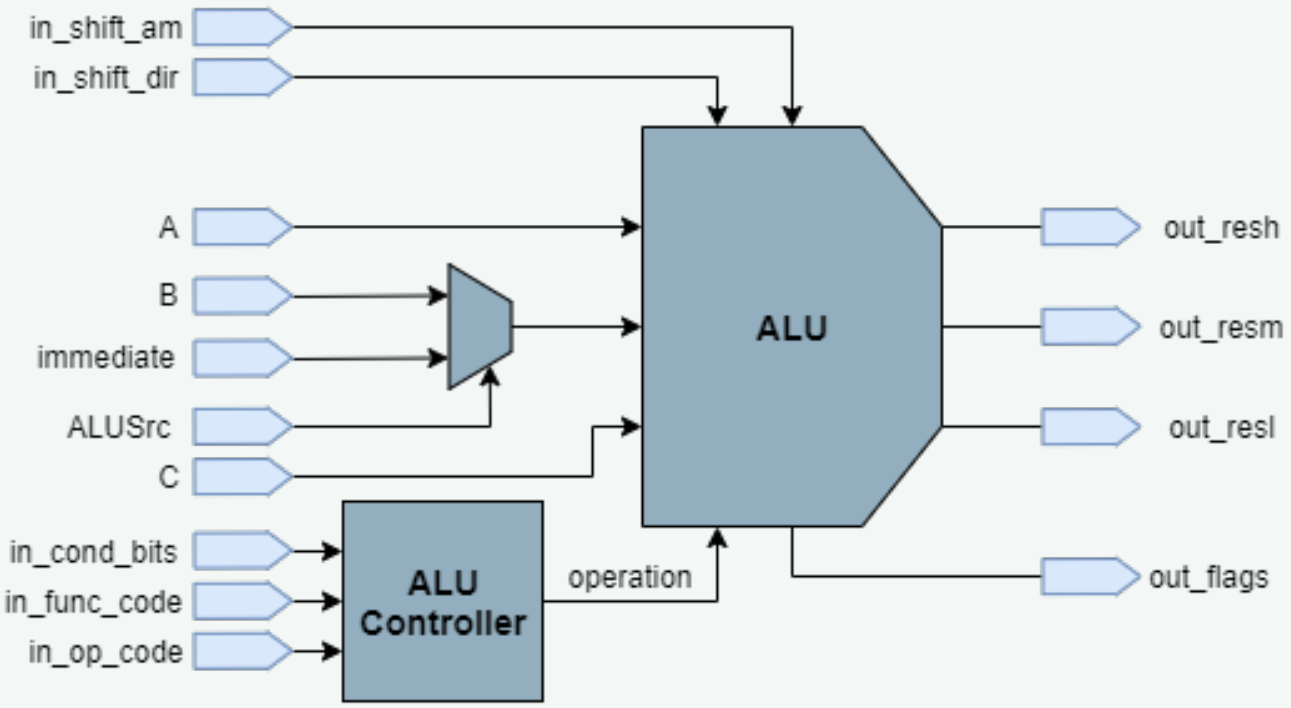

Figure 4.14: Execution Unit Block Diagram

As it can be seen in the picture above, the *Execution Unit* is simply a bigger component that contains the *ALU Controller* presented before and the *Arithmetic Logic Unit*. What's new is the multiplexor that switches between wich input is taken as the second term. It has to choose between the contents of the register file at the correpsonding address and the immediate value from the instruction.

Remark 4.5.12: Size of input/output

It's not specified on the block diagram, but each of the three inputs and the immediate value, together with the results, are encoded on 32 bits. The `out_flags` signal is encoded on 4 bits, as well as the inputs of the *ALU Controller*.

More details about the mechanism behind this block diagram will be presented in the *Implementation* chapter.

4.5.8 Memory Unit and Write Back Unit

In the figure below, the block diagram for the Memory Unit and the Write Back Unit is presented.

Figure 4.5.14: Memory Unit and Write Back Block Diagram

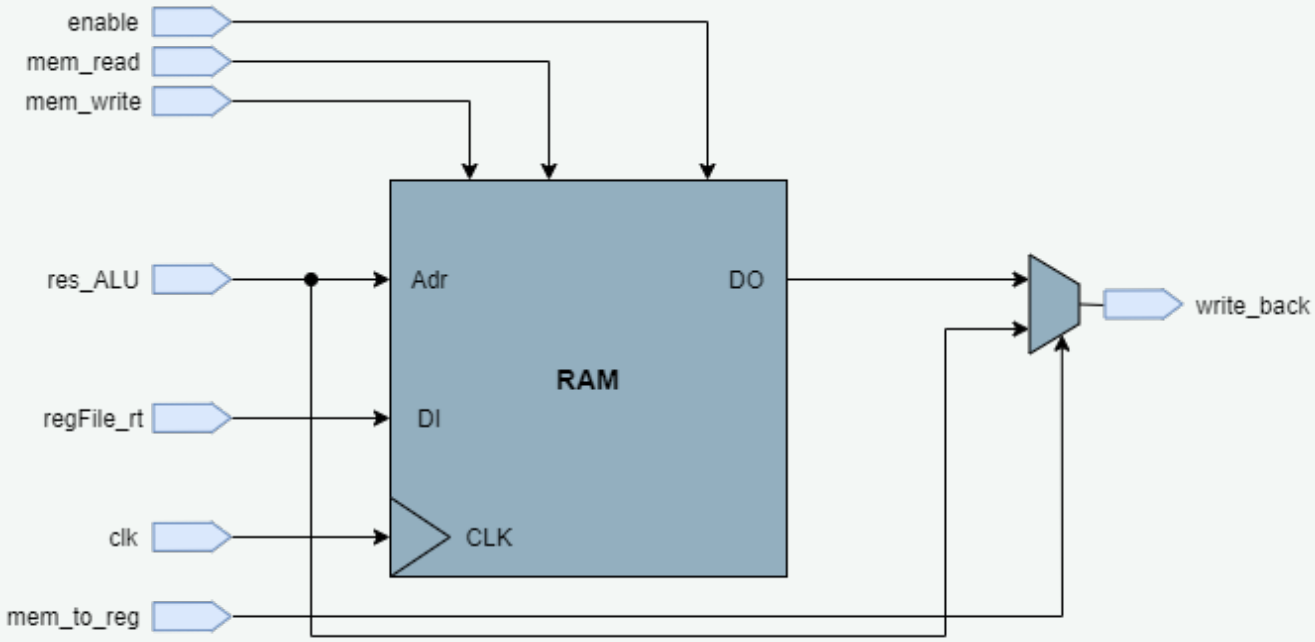

Figure 4.15: Memory Unit and Write Back Block Diagram

As it can be seen in the picture above, the *Memory Unit* and the *Write Back Unit* have been merged together. The write back unit consists of nothing more than a multiplexor that switches between which signal is written back to the register file. For the load operation, the data comming out of the RAM will be written back in the register file. For any other instruction, the result from the ALU is written back in the register file directly.

The memory unit consists of just the RAM that was presented before.

The control signal for the write back unit is the `mem_to_reg` signal, generated by the control unit presented before.

Remark 4.5.13: Size of input/output

It's not specified on the block diagram, but all the inputs and outputs of this component are the same as presented in the *RAM* subsection.

More details about the mechanism of behind this component will be presented in the *Implementation* chapter.

4.5.9 Complex Top Level Diagram

Figure 4.5.15: Top Level (detailed) Block Diagram

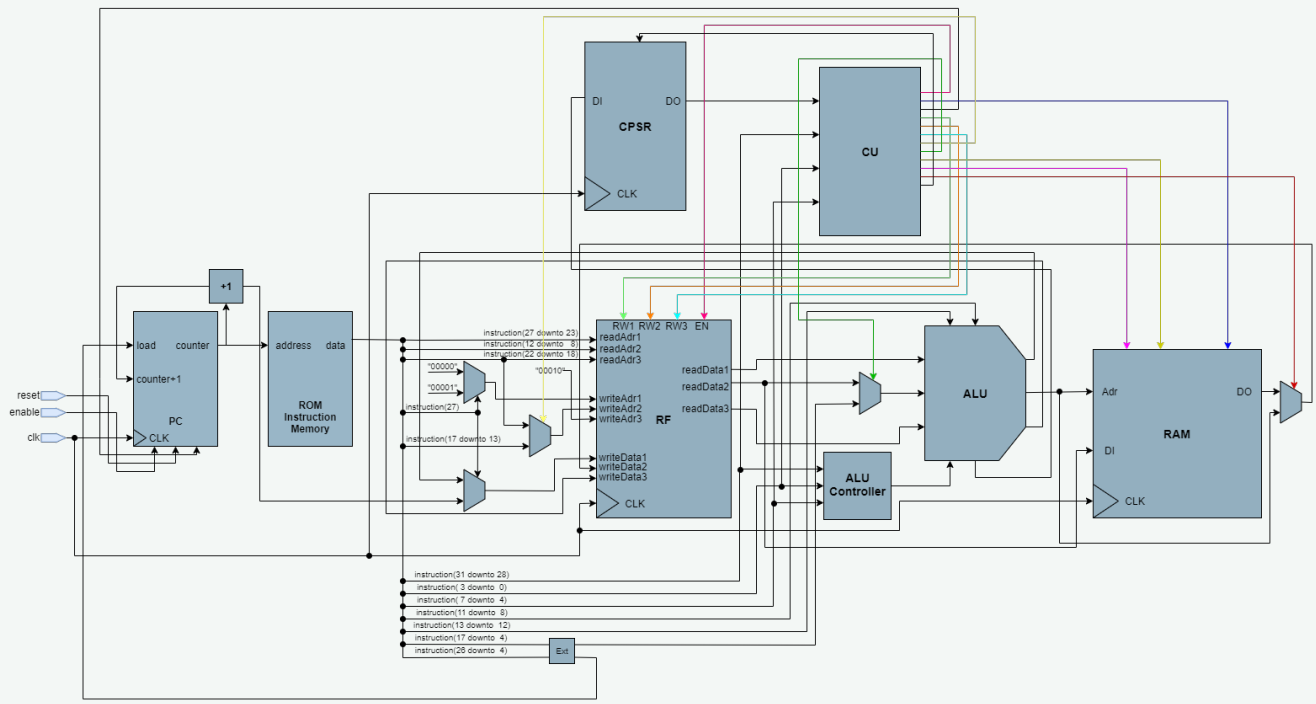

Figure 4.16: Top Level (detailed) Block Diagram

The previous figure contains the top level block diagram of the CPU. It presents the components that were described so far and the way in which they are connected. The control signals (i.e. the outputs of the control unit) are colored so they can be easily distinguished in the scheme.

Chapter 5

Implementation

5.1 Memory

5.1.1 Register File

As mentioned before, the register file has 3 input addresses and 3 data outputs in order to accomodate for the *Triple-Operand Instruction Set*. In the listing below, the enitity for the register file is presented. It matches the block diagram of the register file presented before.

Listing 5.1.1: Register File - Entity

```
entity RegisterFile is
    port ( clk            :  in STD_LOGIC;
           enable         :  in STD_LOGIC;
           registerWrite1:  in STD_LOGIC;
           registerWrite2:  in STD_LOGIC;
           registerWrite3:  in STD_LOGIC;
           readAdr1       :  in STD_LOGIC_VECTOR ( 4 downto 0);
           readAdr2       :  in STD_LOGIC_VECTOR ( 4 downto 0);
           readAdr3       :  in STD_LOGIC_VECTOR ( 4 downto 0);
           writeAdr1      :  in STD_LOGIC_VECTOR ( 4 downto 0);
           writeAdr2      :  in STD_LOGIC_VECTOR ( 4 downto 0);
           writeAdr3      :  in STD_LOGIC_VECTOR ( 4 downto 0);
           writeData1     :  in STD_LOGIC_VECTOR (31 downto 0);
           writeData2     :  in STD_LOGIC_VECTOR (31 downto 0);
           writeData3     :  in STD_LOGIC_VECTOR (31 downto 0);
           readData1      : out STD_LOGIC_VECTOR (31 downto 0);
           readData2      : out STD_LOGIC_VECTOR (31 downto 0);
           readData3      : out STD_LOGIC_VECTOR (31 downto 0));
end RegisterFile;
```

As it can be seen in the listing, the address of a register is encoded on 5 bits, which allows for a total of 32 registers. The data in these registers is of 32 bits width as well.

The `writeAdr` and `writeData` ports are used in conjunction with the Write Back Unit, which provides the CPU with the functionality to write the result of an instruction back in the register file.

As stated before, the register file has 3 input port for writting data and also 3 for the address where that data will be written.

In the listing below, the architecture for the register file is presented.

Listing 5.1.2: Register File - Architecture

```
type regArray is array ( 0 to 31) of STD_LOGIC_VECTOR(31 downto 0);
signal RF : regArray :=(others => X"0000_0000");
begin
REG_FILE: process (clk)
begin
    if enable = '1' then
        if clk = '1' and clk'event then
            if registerWrite1 = '1' then
                RF(conv_integer(writeAdr1)) <= writeData1;
            end if;
            if registerWrite2 = '1' then
                RF(conv_integer(writeAdr2)) <= writeData2;
            end if;
            if registerWrite3 = '1' then
                RF(conv_integer(writeAdr3)) <= writeData3;
            end if;
        end if;
    end if;
end process;
READ_DATA: process(enable)
begin
    if enable = '1' then
        readData1 <= RF(conv_integer(readAdr1));
        readData2 <= RF(conv_integer(readAdr2));
        readData3 <= RF(conv_integer(readAdr3));
    end if;
end process;
```

From the architecture of the register file, we can easily notice that the writing procedure is done synchronously and each writing port is enabled by one writing signal. Moreover, the reading from this component is done asynchronously. However, the `enable` signal is the one that drives the whole component. If it is clear (i.e. it has the logic value 0), the component is not active, thus no reading or writing will be done from or in the register file.

5.1.2 ROM

The entity implementation of the ROM is similar to the block diagram that was already presented in the *Desgin* chapter.

Listing 5.1.3: ROM - Entity

```
entity ROM is
    port ( address :  in STD_LOGIC_VECTOR(31 downto 0);
           data    : out STD_LOGIC_VECTOR(31 downto 0));
end ROM;
```

In the listing bellow, the architecture for the ROM is presented.

Listing 5.1.4: ROM - Architecture

```
architecture Behavioral of ROM is
type ROM_t is array(0 to 65535) of STD_LOGIC_VECTOR(31 downto 0);
signal ROM1 : ROM_t := (others => X"0000_0000");
signal ROM2 : ROM_t := (others => X"0000_0000");
begin
GET_DATA: process ( address )
begin
    if conv_integer(address(31 downto 16)) = 0 then
        data <= ROM1(conv_integer(address(15 downto 0)));
    else
        data <= ROM2(conv_integer(address(15 downto 0)));
    end if;
end process;
end Behavioral;
```

It might seem odd that there are two signals instantiated inside the ROM. This is due to the fact that Vivado will not allow for arrays of size 2^{32}, thus I instantiated the signal twice. If the incoming address has the first 16 bits different from 0, then it will be directed to the second signal. In the opposite case, it will be direct to the first.

I chose to have the instrution memory on 32 bits in order to be able to use the same extension operation for the immediate value of the jump operation. This decision also helps in future developement. For example, when implementing the circuitry necessary for executing the branch instruction, I will be able to use the same ALU from the execution unit, because the address will be 32 bits long.

5.1.3 RAM

The enitity for the RAM matches the block diagram presented before. As mentioned in the *Design* chapter, this memory has an `enable` input that activates the whole component. The other inputs are self explanatory.

Listing 5.1.5: RAM - Entity

```
entity RAM is
    port ( clk         :  in STD_LOGIC;
           enable      :  in STD_LOGIC;
           writeEnable :  in STD_LOGIC;
           readEnable  :  in STD_LOGIC;
           address     :  in STD_LOGIC_VECTOR(31 downto 0);
           dataIn      :  in STD_LOGIC_VECTOR(31 downto 0);
           dataOut     : out STD_LOGIC_VECTOR(31 downto 0));
end RAM;
```

This entity is the one used in the `MEMU` implementation.

In the listing bellow, the architecture for the RAM is presented.

Listing 5.1.6: RAM - Architecture

```
architecture Behavioral of RAM is
type RAM_t is array ( 0 to 65535) of STD_LOGIC_VECTOR(31 downto 0);
signal RAM1 : RAM_t := (others => X"0000_0000");
signal RAM2 : RAM_t := (others => X"0000_0000");
begin
WRITE: process (clk)
begin
    if enable = '1' then
        if conv_integer(address(31 downto 16)) = 0 then
            if clk = '1' and clk'event then
                if writeEnable = '1' then
                    RAM1(conv_integer(address(15 downto 0))) <= dataIn;
                end if;
            end if;
        else
            if clk = '1' and clk'event then
                if writeEnable = '1' then
                    RAM2(conv_integer(address(15 downto 0))) <= dataIn;
                end if;
            end if;
        end if;
    end if;
end process;
READ: process (clk)
begin
    if enable = '1' then
        if conv_integer(address(31 downto 16)) = 0 then
            if clk = '0' and clk'event then
                if readEnable = '1' then
                    dataOut <= RAM1(conv_integer(address(15 downto 0)));
                end if;
            end if;
        else
            if clk = '0' and clk'event then
                if readEnable = '1' then
                    dataOut <= RAM2(conv_integer(address(15 downto 0)));
                end if;
            end if;
        end if;
    end if;
end process;
end Behavioral;
```

I used the same procedure of splitting the memory in two parts, in the same manner as I did for the ROM. All the explanations about why and how this was done were given before.

5.2 Miscellaneous

5.2.1 Debouncer

The explanations about how this component is working were given in the *Design* chapter, when I described the DEBOUNCER.

Listing 5.2.1: Debouncer - Entity

```
entity Debouncer is
    port ( clk      :  in STD_LOGIC;
           button   :  in STD_LOGIC_VECTOR(4 downto 0);
           dbButton : out STD_LOGIC_VECTOR(4 downto 0));
end Debouncer;
```

Listing 5.2.2: Debouncer - Architecture

```
architecture Behavioral of Debouncer is
signal q1: STD_LOGIC_VECTOR(4 downto 0) := (others => '0');
signal q2: STD_LOGIC_VECTOR(4 downto 0) := (others => '0');
signal q3: STD_LOGIC_VECTOR(4 downto 0) := (others => '0');
signal counter: STD_LOGIC_VECTOR(16 downto 0) := (others => '0');
begin
FREQ_DIVIDER: process (clk, counter)
begin
    if clk = '1' and clk'event then
        counter <= counter + 1;
    end if;
end process;
LATCHES: process (clk, counter)
begin
    if clk = '1' and clk'event then
        if counter(16) = '1' then
            q1 <= button;
            q2 <= q1;
            q3 <= q2;
        end if;
    end if;
end process;
T_AND: dbButton <= q1 and q2 and (not q3);
end Behavioral;
```

5.2.2 Seven Segment Display Controller

The explanations about how this component is working were given in the *Design* chapter, when I described the SEVEN SEGMENT DISPLAY CONTROLLER.

Listing 5.2.3: Seven Segment Display Controller - Entity

```
entity ControllerSevenSegmentDisplay is
    port ( clk     :  in STD_LOGIC;
           digits  :  in STD_LOGIC_VECTOR (15 downto 0);
           anode   : out STD_LOGIC_VECTOR ( 3 downto 0);
           cathode : out STD_LOGIC_VECTOR ( 7 downto 0));
end ControllerSevenSegmentDisplay;
```

Listing 5.2.4: Seven Segment Display Controller - Architecture

```
architecture Behavioral of ControllerSevenSegmentDisplay is
signal counter : STD_LOGIC_VECTOR(15 downto 0) :=(others => '0');
signal decoder : STD_LOGIC_VECTOR( 3 downto 0) :=(others => '0');
begin
FREQ_DIVIDER: process (clk, counter)
begin
    if clk = '1' and clk'event then
        counter <= counter + 1;
    end if;
end process;
ANODE_SELECTOR: process (counter)
begin
    case counter(15 downto 14) is
        when "00"   => anode <= "1110"; -- first display
        when "01"   => anode <= "1101"; -- second display
        when "10"   => anode <= "1011"; -- third display
        when "11"   => anode <= "0111"; -- fourth display
        when others => anode <= "1111"; -- error
    end case;
end process;
DIGIT_SELECTOR: process (counter, digits)
begin
    case counter(15 downto 14) is
        when "00"   => decoder <= digits( 3 downto  0); -- first
        when "01"   => decoder <= digits( 7 downto  4); -- second
        when "10"   => decoder <= digits(11 downto  8); -- tirhd
        when "11"   => decoder <= digits(15 downto 12); -- fourth
        when others => decoder <= "0000";               -- error
    end case;
end process;
CATHODE_SELECTOR: process(decoder)
begin
    case decoder is
        when "0000" => cathode <= "11000000"; -- 0
        when "0001" => cathode <= "11111001"; -- 1
        when "0010" => cathode <= "10100100"; -- 2
        when "0011" => cathode <= "10110000"; -- 3
        when "0100" => cathode <= "10011001"; -- 4
        when "0101" => cathode <= "10010010"; -- 5
        when "0110" => cathode <= "10000010"; -- 6
        when "0111" => cathode <= "11111000"; -- 7
        when "1000" => cathode <= "10000000"; -- 8
        when "1001" => cathode <= "10010000"; -- 9
        when "1010" => cathode <= "10001000"; -- A
        when "1011" => cathode <= "10000011"; -- B
        when "1100" => cathode <= "11000110"; -- C
        when "1101" => cathode <= "10100001"; -- D
        when "1110" => cathode <= "10000110"; -- E
        when "1111" => cathode <= "10001110"; -- F
        when others => cathode <= "11111111"; -- segments off
    end case;
end process;
end Behavioral;
```

5.2.3 CPSR

The CPSR is nothing more than a simple register, whose bits have a special meaning that was already presented.

Listing 5.2.5: Current Program State Register - Entity

```
entity CPSR is
    Port ( clk      :  in STD_LOGIC;
           write_en :  in STD_LOGIC;
           data_in  :  in STD_LOGIC_VECTOR( 3 downto 0);
           data_out : out STD_LOGIC_VECTOR( 3 downto 0));
end CPSR;
```

Listing 5.2.6: Current Program State Register - Architecture

```
architecture Behavioral of CPSR is
signal Z, V, C, N : STD_LOGIC;
begin
SET_FLAGS: process (clk)
begin
    if clk = '1' and clk'event and write_en = '1' then
        Z <= data_in(3);
        V <= data_in(2);
        C <= data_in(1);
        N <= data_in(0);
    end if;
end process;
GET_FLAGS: data_out <= Z & V & C & N;
end Behavioral;
```

5.2.4 ALU

Bellow, the entity for the ALU is listed.

Listing 5.2.7: Arithmetic Logic Unit - Entity

```
entity ALU is
    Port ( term1       :  in STD_LOGIC_VECTOR (31 downto 0);
           term2       :  in STD_LOGIC_VECTOR (31 downto 0);
           term3       :  in STD_LOGIC_VECTOR (31 downto 0);
           operation   :  in STD_LOGIC_VECTOR ( 3 downto 0);
           result_high : out STD_LOGIC_VECTOR (31 downto 0);
           result      : out STD_LOGIC_VECTOR (31 downto 0);
           result_low  : out STD_LOGIC_VECTOR (31 downto 0);
           flags_out   : out STD_LOGIC_VECTOR ( 3 downto 0);
        -- flags_out(3) = Z (zero)
        -- flags_out(2) = V (overflow)
        -- flags_out(1) = C (carry)
        -- flags_out(0) = N (negative)
           shift_am    :  in STD_LOGIC_VECTOR ( 3 downto 0);
           shift_dir   :  in STD_LOGIC_VECTOR ( 1 downto 0));
end ALU;
```

The architecture of the ALU may seem overwhelming at first, but each operation is implemented clearly. This component does more than just perform computations, it also generates flags based on the results.

Listing 5.2.8: Arithmetic Logic Unit - Architecture

```
architecture Behavioral of ALU is
signal Z_flag    : STD_LOGIC := '0';
signal V_flag    : STD_LOGIC := '0';
signal C_flag    : STD_LOGIC := '0';
signal N_flag    : STD_LOGIC := '0';
signal pad_left  : STD_LOGIC_VECTOR(14 downto 0) := (others => '0');
signal pad_right : STD_LOGIC_VECTOR(14 downto 0) := (others => '0');
begin
COMPUTE: process(operation, term1, term2, term3, shift_am, shift_dir)
    variable v_add_2 : UNSIGNED(32 downto 0);
    variable v_add_3 : UNSIGNED(33 downto 0);
    variable v_mul_2 : UNSIGNED(63 downto 0);
    variable v_mul_3 : UNSIGNED(95 downto 0);
    variable v_res_u : UNSIGNED(31 downto 0);
begin
    pad_left  <= (others => term1( 0));
    pad_right <= (others => term1(31));
    Z_flag <= '0';
    V_flag <= '0';
    C_flag <= '0';
    N_flag <= '0';
    v_add_2 := RESIZE(UNSIGNED(term1), 33) + UNSIGNED(term2);
    v_add_3 := RESIZE(UNSIGNED(term1), 34) + UNSIGNED(term2) + UNSIGNED(term3);
    v_mul_2 := UNSIGNED(term1) * UNSIGNED(term2);
    v_mul_3 := UNSIGNED(term1) * UNSIGNED(term2) * UNSIGNED(term3);
    case operation is
        when "0000" => result <= STD_LOGIC_VECTOR(v_add_2(31 downto 0));--add
                       V_flag <= STD_LOGIC(v_add_2(32));
                       C_flag <= STD_LOGIC(v_add_2(32));
                       if conv_integer(STD_LOGIC_VECTOR(v_add_2(31 downto 0)))
                          = 0 then
                           Z_flag <= '1';
                       else Z_flag <= '0';
                       end if;
        when "0001" => result <= term1 - term2;--sub
                       v_res_u := UNSIGNED(term1) - UNSIGNED(term2);
                       if conv_integer(STD_LOGIC_VECTOR(v_res_u)) = 0 then
                           Z_flag <= '1';
                       else Z_flag <= '0';
                       end if;
                       if to_integer(UNSIGNED(term1)) < to_integer(UNSIGNED(
                          term2)) then
                           V_flag <= '1';
                           N_flag <= '1';
                       else V_flag <= '0';
                            N_flag <= '0';
                       end if;
        when "0010" => result <= term1 xor term2;--xor
                       v_res_u := UNSIGNED(term1) xor UNSIGNED(term2);
                       if conv_integer(STD_LOGIC_VECTOR(v_res_u)) = 0 then
                           Z_flag <= '1';
```

```
                else Z_flag <= '0';
                end if;
        when "0011" => result <= term1 and term2;--and
                v_res_u := UNSIGNED(term1) and UNSIGNED(term2);
                if conv_integer(STD_LOGIC_VECTOR(v_res_u)) = 0 then
                    Z_flag <= '1';
                else Z_flag <= '0';
                end if;
        when "0100" => result_high <= STD_LOGIC_VECTOR(v_mul_2(63 downto 32));
           --mul
                result <= (others => '0');
                result_low  <= STD_LOGIC_VECTOR(v_mul_2(31 downto  0));
                if conv_integer(STD_LOGIC_VECTOR(v_mul_2(63 downto 32)))
                    = 0 then
                    V_flag <= '0';
                else V_flag <= '1';
                end if;
        when "0101" => --shift
            case shift_dir is
                when "00" => --sll
                    case shift_am is
                        when "0000" => result <= (others => '0');
                        when "0001" => result <= term1(30 downto 0) & "0";
                        when "0010" => result <= term1(29 downto 0) & "00";
                        when "0011" => result <= term1(28 downto 0) & "000";
                        when "0100" => result <= term1(27 downto 0) & "0000";
                        when "0101" => result <= term1(26 downto 0) & "00000";
                        when "0110" => result <= term1(25 downto 0) & "000000";
                        when "0111" => result <= term1(24 downto 0) &
                            "0000000";
                        when "1000" => result <= term1(23 downto 0) &
                            "00000000";
                        when "1001" => result <= term1(22 downto 0) &
                            "000000000";
                        when "1010" => result <= term1(21 downto 0) &
                            "0000000000";
                        when "1011" => result <= term1(20 downto 0) &
                            "00000000000";
                        when "1100" => result <= term1(19 downto 0) &
                            "000000000000";
                        when "1101" => result <= term1(18 downto 0) &
                            "0000000000000";
                        when "1110" => result <= term1(17 downto 0) &
                            "00000000000000";
                        when "1111" => result <= term1(16 downto 0) &
                            "000000000000000";
                        when others => result <= X"FFFF_FFFF";
                    end case;
                when "01" => --srl
                    case shift_am is
                        when "0000" => result <= (others => '0');
                        when "0001" => result <= "0" & term1(31 downto 1);
                        when "0010" => result <= "00" & term1(31 downto 2);
                        when "0011" => result <= "000" & term1(31 downto 3);
                        when "0100" => result <= "0000" & term1(31 downto 4);
                        when "0101" => result <= "00000" & term1(31 downto 5);
```

```
                    when "0110" => result <= "000000" & term1(31 downto 6);
                    when "0111" => result <= "0000000" & term1(31 downto 7)
                        ;
                    when "1000" => result <= "00000000" & term1(31 downto
                        8);
                    when "1001" => result <= "000000000" & term1(31 downto
                        9);
                    when "1010" => result <= "0000000000" & term1(31 downto
                         10);
                    when "1011" => result <= "00000000000" & term1(31
                        downto 11);
                    when "1100" => result <= "000000000000" & term1(31
                        downto 12);
                    when "1101" => result <= "0000000000000" & term1(31
                        downto 13);
                    when "1110" => result <= "00000000000000" & term1(31
                        downto 14);
                    when "1111" => result <= "000000000000000" & term1(31
                        downto 15);
                    when others => result <= X"FFFF_FFFF";
                end case;
            when "10" => --sla
                case shift_am is
                    when "0000" => result <= (others => '0');
                    when "0001" => result <= term1(30 downto 0) & pad_left
                        (0);
                    when "0010" => result <= term1(29 downto 0) & pad_left(
                         1 downto 0);
                    when "0011" => result <= term1(28 downto 0) & pad_left(
                         2 downto 0);
                    when "0100" => result <= term1(27 downto 0) & pad_left(
                         3 downto 0);
                    when "0101" => result <= term1(26 downto 0) & pad_left(
                         4 downto 0);
                    when "0110" => result <= term1(25 downto 0) & pad_left(
                         5 downto 0);
                    when "0111" => result <= term1(24 downto 0) & pad_left(
                         6 downto 0);
                    when "1000" => result <= term1(23 downto 0) & pad_left(
                         7 downto 0);
                    when "1001" => result <= term1(22 downto 0) & pad_left(
                         8 downto 0);
                    when "1010" => result <= term1(21 downto 0) & pad_left(
                         9 downto 0);
                    when "1011" => result <= term1(20 downto 0) & pad_left
                        (10 downto 0);
                    when "1100" => result <= term1(19 downto 0) & pad_left
                        (11 downto 0);
                    when "1101" => result <= term1(18 downto 0) & pad_left
                        (12 downto 0);
                    when "1110" => result <= term1(17 downto 0) & pad_left
                        (13 downto 0);
                    when "1111" => result <= term1(16 downto 0) & pad_left
                        (14 downto 0);
                    when others => result <= X"FFFF_FFFF";
                end case;
```

```
            when "11" => --sra
                case shift_am is
                    when "0000" => result <= (others => '0');
                    when "0001" => result <= pad_right(0) & term1(31 downto
                        1);
                    when "0010" => result <= pad_right( 1 downto 0) & term1
                       (31 downto  2);
                    when "0011" => result <= pad_right( 2 downto 0) & term1
                       (31 downto  3);
                    when "0100" => result <= pad_right( 3 downto 0) & term1
                       (31 downto  4);
                    when "0101" => result <= pad_right( 4 downto 0) & term1
                       (31 downto  5);
                    when "0110" => result <= pad_right( 5 downto 0) & term1
                       (31 downto  6);
                    when "0111" => result <= pad_right( 6 downto 0) & term1
                       (31 downto  7);
                    when "1000" => result <= pad_right( 7 downto 0) & term1
                       (31 downto  8);
                    when "1001" => result <= pad_right( 8 downto 0) & term1
                       (31 downto  9);
                    when "1010" => result <= pad_right( 9 downto 0) & term1
                       (31 downto 10);
                    when "1011" => result <= pad_right(10 downto 0) & term1
                       (31 downto 11);
                    when "1100" => result <= pad_right(11 downto 0) & term1
                       (31 downto 12);
                    when "1101" => result <= pad_right(12 downto 0) & term1
                       (31 downto 13);
                    when "1110" => result <= pad_right(13 downto 0) & term1
                       (31 downto 14);
                    when "1111" => result <= pad_right(14 downto 0) & term1
                       (31 downto 15);
                    when others => result <= X"FFFF_FFFF";
                end case;
            when others => result <= X"FFFF_FFFF";
        end case;
    when "0110" => v_res_u := UNSIGNED(term1) - UNSIGNED(term2);
                   if to_integer(UNSIGNED(term1)) > to_integer(UNSIGNED(
                      term2)) then--cmp
                       N_flag <= V_flag;
                   elsif to_integer(UNSIGNED(term1)) < to_integer(UNSIGNED(
                      term2)) then
                       N_flag <= not V_flag;
                   else
                       Z_flag <= '1';
                   end if;
    when "0111" => result <= X"FFFF_FFFF";
    when "1000" => result <= STD_LOGIC_VECTOR(v_add_3(31 downto 0)); --addt
                   V_flag <= STD_LOGIC(v_add_3(33)) or STD_LOGIC(v_add_3
                      (32));
                   C_flag <= STD_LOGIC(v_add_3(33)) or STD_LOGIC(v_add_3
                      (32));
                   if conv_integer(STD_LOGIC_VECTOR(v_add_3(31 downto 0)))
                      = 0 then
                       Z_flag <= '1';
```

```
                    else Z_flag <= '0';
                    end if;
        when "1001" => result <= term1 - term2 - term3;--subt
                    v_res_u := UNSIGNED(term1) - UNSIGNED(term2) - UNSIGNED(
                        term3);
                    if conv_integer(STD_LOGIC_VECTOR(v_res_u)) = 0 then
                        Z_flag <= '1';
                    else Z_flag <= '0';
                    end if;
                    if to_integer(v_res_u) < 0 then
                        V_flag <= '1';
                        N_flag <= '1';
                    else V_flag <= '0';
                         N_flag <= '0';
                    end if;
        when "1010" => result <= term1 xor term2 xor term3;--xort
                    v_res_u := UNSIGNED(term1) xor UNSIGNED(term2) xor
                        UNSIGNED(term3);
                    if conv_integer(STD_LOGIC_VECTOR(v_res_u)) = 0 then
                        Z_flag <= '1';
                    else Z_flag <= '0';
                    end if;
        when "1011" => result <= term1 and term2 and term3;--andt
                    v_res_u := UNSIGNED(term1) and UNSIGNED(term2) and
                        UNSIGNED(term3);
                    if conv_integer(STD_LOGIC_VECTOR(v_res_u)) = 0 then
                        Z_flag <= '1';
                    else Z_flag <= '0';
                    end if;
        when "1100" => result_high <= STD_LOGIC_VECTOR(v_mul_3(95 downto 64));
            --mult
                    result      <= STD_LOGIC_VECTOR(v_mul_3(63 downto 32));
                    result_low  <= STD_LOGIC_VECTOR(v_mul_3(31 downto  0));
                    if conv_integer(STD_LOGIC_VECTOR(v_mul_3(95 downto 64)))
                        = 0
                     and conv_integer(STD_LOGIC_VECTOR(v_mul_3(63 downto 32)
                        )) = 0 then
                        V_flag <= '0';
                    else V_flag <= '1';
                    end if;
        when "1101" => result <= term1 or term2 or term3;--ort
                    v_res_u := UNSIGNED(term1) or UNSIGNED(term2) or
                        UNSIGNED(term3);
                    if conv_integer(STD_LOGIC_VECTOR(v_res_u)) = 0 then
                        Z_flag <= '1';
                    else Z_flag <= '0';
                    end if;
        when "1110" => result <= not(term1 or term2 or term3);--nort
                    v_res_u := not(UNSIGNED(term1) or UNSIGNED(term2) or
                        UNSIGNED(term3));
                    if conv_integer(STD_LOGIC_VECTOR(v_res_u)) = 0 then
                        Z_flag <= '1';
                    else Z_flag <= '0';
                    end if;
        when "1111" => result <= not(term1 and term2 and term3);--nandt
                    v_res_u := not(UNSIGNED(term1) and UNSIGNED(term2) and
```

```
                        UNSIGNED(term3));
                    if conv_integer(STD_LOGIC_VECTOR(v_res_u)) = 0 then
                        Z_flag <= '1';
                    else Z_flag <= '0';
                    end if;
        when others => result <= X"FFFF_FFFF";
    end case;
end process;
GET_FLAGREG: flags_out   <= Z_flag & V_flag & C_flag & N_flag;
end Behavioral;
```

5.2.5 ALU Controller

In the listing bellow, the entity for the ALU Controller is presented.

Listing 5.2.9: ALU Control Unit - Entity

```
entity ALU_Controller is
    Port ( flags     :  in STD_LOGIC_VECTOR( 3 downto 0);
           op_code   :  in STD_LOGIC_VECTOR( 3 downto 0);
           func_code :  in STD_LOGIC_VECTOR( 3 downto 0);
           operation : out STD_LOGIC_VECTOR( 3 downto 0));
end ALU_Controller;
```

In the listing bellow, the architecture for the ALU Controller is presented.

Listing 5.2.10: ALU Control Unit - Architecture

```
architecture Behavioral of ALU_Controller is
begin
GEN_OP: process ( op_code, func_code, flags )
begin
    case flags is
        when "1111" =>
            if op_code = "0000" then
                case func_code is
                    when "0000" => operation <= "1000";
                    when "0001" => operation <= "1001";
                    when "0010" => operation <= "1010";
                    when "0011" => operation <= "1011";
                    when "0100" => operation <= "1100";
                    when "0101" => operation <= "1101";
                    when "0110" => operation <= "1110";
                    when "0111" => operation <= "1111";
                    when others => operation <= "0111";
                end case;
            end if;
        when others =>
            case op_code is
                when "0000" =>
                    case func_code is
                        when "0000" => operation <= "0000";
                        when "0001" => operation <= "0001";
                        when "0010" => operation <= "0010";
                        when "0011" => operation <= "0011";
                        when "0100" => operation <= "0100";
```

```
                        when others => operation <= "0111";
                    end case;
                when "0001" =>
                    case func_code is
                        when "0000" => operation <= "0101";
                        when "0001" => operation <= "0110";
                        when others => operation <= "0111";
                    end case;
                when "0010" => operation <= "0000";
                when "0011" => operation <= "0000";
                when "0101" => operation <= "0000";
                when others => operation <= "0111";
            end case;
    end case;
end process;
end Behavioral;
```

5.3 Instruction Fetch Unit

In the listing below, the enitity for the instruction fetch unit is presented. It matches the block diagram of the instruction fetch unit presented before.

Listing 5.3.1: Instruction Fetch Unit - Entity

```
entity IFU is
    port ( clk           :  in STD_LOGIC;
           reset         :  in STD_LOGIC;
           enable        :  in STD_LOGIC;
           jump_address  :  in STD_LOGIC_VECTOR(15 downto 0);
           jump_ctrl     :  in STD_LOGIC;
           instruction   : out STD_LOGIC_VECTOR(31 downto 0);
           next_address  : out STD_LOGIC_VECTOR(31 downto 0));
end IFU;
```

As it can be seen from the listing, this component has three enableing input signals, namely `reset, enable` and `jump_ctrl`. The `reset` signal is self explanatory: it reset the Program Counter back to the address `X"0000_0000"`. The `enable` signal will come from a button of the FPGA board. This signal will be debounced before it will be used in the instruction fetch component. Finally, the `jump_ctrl` signal will come from the Control Unit which was presented in the *Design Chapter*.

In the listing below, the architecture for the instruction fetch unit is presented.

Listing 5.3.2: Instruction Fetch Unit - Architecture

```
architecture Behavioral of IFU is
component ROM is
    port ( address :  in STD_LOGIC_VECTOR(31 downto 0);
           data    : out STD_LOGIC_VECTOR(31 downto 0));
end component ROM;
signal counter :STD_LOGIC_VECTOR(31 downto 0):=(others => '0');
signal ROM_data:STD_LOGIC_VECTOR(31 downto 0):=(others => '0');
begin
PROGRAM_COUNTER: process (clk, reset, enable)
begin
```

```
    if clk = '1' and clk'event then
        if    reset = '1' then counter <= (others => '0');
        elsif enable = '1' then counter <= counter + 1;
        elsif jump_ctrl = '1' then counter <= jump_address;
        end if;
    end if;
end process;
GET_INSTRUCTION: instruction <= ROM_data;
GET_NEXT_ADDRSS: next_address <= counter + 1;
ROM_COMP: ROM port map (address => counter, data => ROM_data);
end Behavioral;
```

As it can be seen from the listing, the Program Counter is just a register with multiple control signals. At each clock cycle, the program counter can either increment its value, load the jump address or reset.

The component provides the instruction stored at that address in the instruction memory, as well as the next address in the flow of code.

5.4 Instruction Decode Unit

In the listing below, the enitity for the instruction decode unit is presented. It matches the block diagram of the instruction decode unit presented before.

Listing 5.4.1: Instruction Decode Unit - Entity

```
entity IDU is
    Port ( instruction          :  in STD_LOGIC_VECTOR(31 downto 0);
           clk                  :  in STD_LOGIC;
           enable               :  in STD_LOGIC;
           register_write1      :  in STD_LOGIC;
           register_write2      :  in STD_LOGIC;
           register_write3      :  in STD_LOGIC;
           register_destination :  in STD_LOGIC;
           write_data1          :  in STD_LOGIC_VECTOR(31 downto 0);
           write_data2          :  in STD_LOGIC_VECTOR(31 downto 0);
           write_data3          :  in STD_LOGIC_VECTOR(31 downto 0);
           link_adr             :  in STD_LOGIC_VECTOR(31 downto 0);
           read_data1           : out STD_LOGIC_VECTOR(31 downto 0);
           read_data2           : out STD_LOGIC_VECTOR(31 downto 0);
           read_data3           : out STD_LOGIC_VECTOR(31 downto 0);
           ext_imm              : out STD_LOGIC_VECTOR(31 downto 0);
           jump_address         : out STD_LOGIC_VECTOR(31 downto 0);
           instruction_flags    : out STD_LOGIC_VECTOR( 3 downto 0);
           instruction_code     : out STD_LOGIC_VECTOR( 3 downto 0);
           instruction_function : out STD_LOGIC_VECTOR( 3 downto 0);
           instruction_shift_am : out STD_LOGIC_VECTOR( 3 downto 0);
           instruction_shift_dir: out STD_LOGIC_VECTOR( 1 downto 0));
end IDU;
```

In the listing below, the architecture for the instruction decode unit is presented.

Listing 5.4.2: Instruction Decode Unit - Architecture

```
architecture Behavioral of IDU is
component RegisterFile is
    port ( clk           :  in STD_LOGIC;
           enable        :  in STD_LOGIC;
           registerWrite1:  in STD_LOGIC;
           registerWrite2:  in STD_LOGIC;
           registerWrite3:  in STD_LOGIC;
           readAdr1      :  in STD_LOGIC_VECTOR ( 4 downto 0);
           readAdr2      :  in STD_LOGIC_VECTOR ( 4 downto 0);
           readAdr3      :  in STD_LOGIC_VECTOR ( 4 downto 0);
           writeAdr1     :  in STD_LOGIC_VECTOR ( 4 downto 0);
           writeAdr2     :  in STD_LOGIC_VECTOR ( 4 downto 0);
           writeAdr3     :  in STD_LOGIC_VECTOR ( 4 downto 0);
           writeData1    :  in STD_LOGIC_VECTOR (31 downto 0);
           writeData2    :  in STD_LOGIC_VECTOR (31 downto 0);
           writeData3    :  in STD_LOGIC_VECTOR (31 downto 0);
           readData1     : out STD_LOGIC_VECTOR (31 downto 0);
           readData2     : out STD_LOGIC_VECTOR (31 downto 0);
           readData3     : out STD_LOGIC_VECTOR (31 downto 0));
end component RegisterFile;
signal write_address1 : STD_LOGIC_VECTOR( 4 downto 0) := (others => '0');
signal write_data_1   : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal write_address2 : STD_LOGIC_VECTOR( 4 downto 0) := (others => '0');
begin
CHOOSE_DEST_N_DATA: process(instruction(27))
begin
    if instruction(27) = '1' and instruction( 3 downto 0) = "0100" then
        write_address1 <= "00000";
        write_data_1 <= link_adr;
    else
        write_address1 <= "00001";
        write_data_1 <= write_data1;
    end if;
end process;
CHOOSE_DEST2: process(register_destination)
begin
    if register_destination = '0' then
        write_address2 <= instruction(17 downto 13);
    else
        write_address2 <= instruction(22 downto 18);
    end if;
end process;
GET_FUNCTION : instruction_function <= instruction( 7 downto  4);
GET_CODE : instruction_code     <= instruction( 3 downto  0);
GET_SHIFT_AM : instruction_shift_am <= instruction(11 downto  8);
GET_FLAGS : instruction_flags    <= instruction(31 downto 28);
GET_SHIFT_DIR: instruction_shift_dir<= instruction(13 downto 12);
RF: RegisterFile port map (clk => clk,
                           enable => enable,
                           registerWrite1 => register_write1,
                           registerWrite2 => register_write2,
                           registerWrite3 => register_write3,
                           readAdr1   => instruction(27 downto 23), --rs1
```

```
                            readAdr2   => instruction(12 downto  8), --rs2
                            readAdr3   => instruction(22 downto 18), --rt
                            writeAdr1  => write_address1,
                            writeAdr2  => write_address2,
                            writeAdr3  => "00010",
                            writeData1 => write_data_1,
                            writeData2 => write_data2,
                            writeData3 => write_data3,
                            readData1  => read_data1,
                            readData2  => read_data2,
                            readData3  => read_data3);
GET_IMM : ext_imm <= "0000000000000000" & instruction(17 downto 4);
GET_JMPA: jump_address <= "000000000" & instruction(26 downto 4);
end Behavioral;
```

The complex part when implementing the IDU was having to make it compatible with the types of instructions I chose to design.

In order to accomodate for the link option of the jump instructions, the `CHOOSE_DEST_N_DATA` process switches between writing either the current address in the first register or the high part of the multiplication result in the second register.

The other complex design choice was making the IDU able to take in 3 results at a time for the multiplication instruction. Other then this, the component behaves exactly as expected from an IDU, breaking down the instruction into the corresponding fields.

5.5 Control Unit

In the listing below, the entity for the control unit is presented.

Listing 5.5.1: Control Unit - Entity

```
entity CU is
    Port ( flags          :  in STD_LOGIC_VECTOR( 3 downto 0);
           func           :  in STD_LOGIC_VECTOR( 3 downto 0);
           cond           :  in STD_LOGIC_VECTOR( 3 downto 0);
           code           :  in STD_LOGIC_VECTOR( 3 downto 0);
           IDU_en         : out STD_LOGIC;
           MEMU_en        : out STD_LOGIC;
           jump_ctrl      : out STD_LOGIC;
           regWr1         : out STD_LOGIC;
           regWr2         : out STD_LOGIC;
           regWr3         : out STD_LOGIC;
           regDst         : out STD_LOGIC;
           ALU_source     : out STD_LOGIC;
           mem_read       : out STD_LOGIC;
           mem_write      : out STD_LOGIC;
           mem_to_reg     : out STD_LOGIC;
           CPSR_write_en  : out STD_LOGIC);
end CU;
```

In the listing below, the architecture for the control unit is presented.

Listing 5.5.2: Control Unit - Architecture

```
architecture Behavioral of CU is
begin
GEN_CONTROL_SIGN: process (cond, code, func, flags)
begin
    if cond = "0000" then
        if code = "0000"then
            if func = "0000" then -- Multiply two numbers
                jump_ctrl <= '0';
                IDU_en <= '1';
                MEMU_en <= '0';
                regWr1 <= '1';
                regWr2 <= '0';
                regWr3 <= '1';
                regDst <= '0';
                ALU_source <= '0';
                mem_read <= '0';
                mem_write <= '0';
                mem_to_reg <= '0';
                CPSR_write_en <= '1';
            else -- Any other AL-type op on 2 nb
                jump_ctrl <= '0';
                IDU_en <= '1';
                MEMU_en <= '0';
                regWr1 <= '0';
                regWr2 <= '1';
                regWr3 <= '0';
                regDst <= '0';
                ALU_source <= '0';
                mem_read <= '0';
                mem_write <= '0';
                mem_to_reg <= '0';
                CPSR_write_en <= '1';
            end if;
        elsif code = "0001" then
            if func = "0000" then -- Any shift operation
                jump_ctrl <= '0';
                IDU_en <= '1';
                MEMU_en <= '0';
                regWr1 <= '0';
                regWr2 <= '1';
                regWr3 <= '0';
                regDst <= '0';
                ALU_source <= '0';
                mem_read <= '0';
                mem_write <= '0';
                mem_to_reg <= '0';
                CPSR_write_en <= '0';
            else -- Compare two numbers
                jump_ctrl <= '0';
                IDU_en <= '1';
                MEMU_en <= '0';
                regWr1 <= '0';
                regWr2 <= '0';
```

```
            regWr3 <= '0';
            regDst <= '0';
            ALU_source <= '0';
            mem_read <= '0';
            mem_write <= '0';
            mem_to_reg <= '0';
            CPSR_write_en <= '1';
        end if;
    elsif code = "0010" then -- Load operation
            jump_ctrl <= '0';
            IDU_en <= '1';
            MEMU_en <= '1';
            regWr1 <= '0';
            regWr2 <= '1';
            regWr3 <= '0';
            regDst <= '1';
            ALU_source <= '1';
            mem_read <= '1';
            mem_write <= '0';
            mem_to_reg <= '1';
            CPSR_write_en <= '0';
    elsif code = "0011" then -- Store operation
            jump_ctrl <= '0';
            IDU_en <= '1';
            MEMU_en <= '1';
            regWr1 <= '0';
            regWr2 <= '0';
            regWr3 <= '0';
            regDst <= '0';
            ALU_source <= '1';
            mem_read <= '0';
            mem_write <= '1';
            mem_to_reg <= '0';
            CPSR_write_en <= '0';
    elsif code = "0100" then -- Jump operation
            jump_ctrl <= '1';
            IDU_en <= '1';
            MEMU_en <= '0';
            regWr1 <= '1';
            regWr2 <= '0';
            regWr3 <= '0';
            regDst <= '0';
            ALU_source <= '0';
            mem_read <= '0';
            mem_write <= '0';
            mem_to_reg <= '0';
            CPSR_write_en <= '0';
    elsif code = "0101" then -- Add immediate op
            jump_ctrl <= '0';
            IDU_en <= '1';
            MEMU_en <= '0';
            regWr1 <= '0';
            regWr2 <= '1';
            regWr3 <= '0';
            regDst <= '1';
            ALU_source <= '1';
```

```
                mem_read <= '0';
                mem_write <= '0';
                mem_to_reg <= '0';
                CPSR_write_en <= '1';
        end if;
    elsif cond = "1111" then
        if code = "0000" then
            if func = "0100" then -- Multiply three numbers
                jump_ctrl <= '0';
                IDU_en <= '1';
                MEMU_en <= '0';
                regWr1 <= '1';
                regWr2 <= '1';
                regWr3 <= '1';
                regDst <= '0';
                ALU_source <= '0';
                mem_read <= '0';
                mem_write <= '0';
                mem_to_reg <= '0';
                CPSR_write_en <= '1';
            else -- Any other AL-type op on 3 nb
                jump_ctrl <= '0';
                IDU_en <= '1';
                MEMU_en <= '0';
                regWr1 <= '0';
                regWr2 <= '1';
                regWr3 <= '0';
                regDst <= '0';
                ALU_source <= '0';
                mem_read <= '0';
                mem_write <= '0';
                mem_to_reg <= '0';
                CPSR_write_en <= '1';
            end if;
        end if;
    else
        if cond = flags then
            if code = "0000" and func = "0000" then -- Add after checking flags
                jump_ctrl <= '0';
                IDU_en <= '1';
                MEMU_en <= '0';
                regWr1 <= '0';
                regWr2 <= '1';
                regWr3 <= '0';
                regDst <= '0';
                ALU_source <= '0';
                mem_read <= '0';
                mem_write <= '0';
                mem_to_reg <= '0';
                CPSR_write_en <= '1';
            elsif code = "0100" then -- Conditional jump
                jump_ctrl <= '1';
                IDU_en <= '1';
                MEMU_en <= '0';
                regWr1 <= '1';
                regWr2 <= '0';
```

```
                regWr3 <= '0';
                regDst <= '0';
                ALU_source <= '0';
                mem_read <= '0';
                mem_write <= '0';
                mem_to_reg <= '0';
                CPSR_write_en <= '0';
            end if;
        else -- Error
            jump_ctrl <= '0';
            IDU_en <= '0';
            MEMU_en <= '0';
            regWr1 <= '0';
            regWr2 <= '0';
            regWr3 <= '0';
            regDst <= '0';
            ALU_source <= '0';
            mem_read <= '0';
            mem_write <= '0';
            mem_to_reg <= '0';
            CPSR_write_en <= '0';
        end if;
    end if;
end process;
end Behavioral;
```

5.6 Execution Unit

In the listing below, the entity for the execution unit is presented.

Listing 5.6.1: Execution Unit - Entity

```
entity EXU is
    Port ( A, B, C       :  in STD_LOGIC_VECTOR(31 downto 0);
           in_shift_am   :  in STD_LOGIC_VECTOR( 3 downto 0);
           in_shift_dir  :  in STD_LOGIC_VECTOR( 1 downto 0);
           in_flags      :  in STD_LOGIC_VECTOR( 3 downto 0);
           in_op_code    :  in STD_LOGIC_VECTOR( 3 downto 0);
           in_func_code  :  in STD_LOGIC_VECTOR( 3 downto 0);
           immediate     :  in STD_LOGIC_VECTOR(31 downto 0);
           out_resh      : out STD_LOGIC_VECTOR(31 downto 0);
           out_resm      : out STD_LOGIC_VECTOR(31 downto 0);
           out_resl      : out STD_LOGIC_VECTOR(31 downto 0);
           out_flags     : out STD_LOGIC_VECTOR( 3 downto 0);
           ALUSrc        :  in STD_LOGIC);
end EXU;
```

In the listing below, the architecture for the control unit is presented.

Listing 5.6.2: Execution Unit - Architecture

```
architecture Behavioral of EXU is
component ALU is
    Port ( term1       :  in STD_LOGIC_VECTOR (31 downto 0);
           term2       :  in STD_LOGIC_VECTOR (31 downto 0);
           term3       :  in STD_LOGIC_VECTOR (31 downto 0);
           operation   :  in STD_LOGIC_VECTOR ( 3 downto 0);
           result_high : out STD_LOGIC_VECTOR (31 downto 0);
           result      : out STD_LOGIC_VECTOR (31 downto 0);
           result_low  : out STD_LOGIC_VECTOR (31 downto 0);
           flags_out   : out STD_LOGIC_VECTOR ( 3 downto 0);
        -- flags_out(3) = Z (zero)
        -- flags_out(2) = V (overflow)
        -- flags_out(1) = C (carry)
        -- flags_out(0) = N (negative)
           shift_am    :  in STD_LOGIC_VECTOR ( 3 downto 0);
           shift_dir   :  in STD_LOGIC_VECTOR ( 1 downto 0));
end component ALU;
component ALU_Controller is
    Port ( flags     :  in STD_LOGIC_VECTOR( 3 downto 0);
           op_code   :  in STD_LOGIC_VECTOR( 3 downto 0);
           func_code :  in STD_LOGIC_VECTOR( 3 downto 0);
           operation : out STD_LOGIC_VECTOR( 3 downto 0));
end component ALU_Controller;
signal ALU_term2 : STD_LOGIC_VECTOR(31 downto 0);
signal ALU_op    : STD_LOGIC_VECTOR( 3 downto 0);
begin
TERM2: process (ALUSrc, B, immediate)
begin
    if ALUSrc = '0' then
        ALU_term2 <= B;
    else
        ALU_term2 <= immediate;
    end if;
end process;
ALUC: ALU_Controller port map ( flags     =>  in_flags,
                                op_code   =>  in_op_code,
                                func_code =>  in_func_code,
                                operation => ALU_op);
ALUI: ALU port map ( term1       => A,
                     term2       => ALU_term2,
                     term3       => C,
                     operation   => ALU_op,
                     result_high => out_resh,
                     result      => out_resm,
                     result_low  => out_resl,
                     flags_out   => out_flags,
                     shift_am    => in_shift_am,
                     shift_dir   => in_shift_dir);
end Behavioral;
```

5.7 Memory Unit and Write Back Unit

In the listing below, the entity for the memory unit is presented.

Listing 5.7.1: Memory Unit - Entity

```
entity MEMU is
    port ( mem_read   :  in STD_LOGIC;
           mem_write  :  in STD_LOGIC;
           mem_to_reg :  in STD_LOGIC;
           clk        :  in STD_LOGIC;
           enable     :  in STD_LOGIC;
           res_ALU    :  in STD_LOGIC_VECTOR(31 downto 0);
           regFile_rt :  in STD_LOGIC_VECTOR(31 downto 0);
           write_back : out STD_LOGIC_VECTOR(31 downto 0));
end MEMU;
```

In the listing below, the architecture for the control unit is presented.

Listing 5.7.2: Memory Unit - Architecture

```
architecture Behavioral of MEMU is
component RAM is
    port ( clk         :  in STD_LOGIC;
           enable      :  in STD_LOGIC;
           writeEnable :  in STD_LOGIC;
           readEnable  :  in STD_LOGIC;
           address     :  in STD_LOGIC_VECTOR(31 downto 0);
           dataIn      :  in STD_LOGIC_VECTOR(31 downto 0);
           dataOut     : out STD_LOGIC_VECTOR(31 downto 0));
end component RAM;
signal out_dataOut : STD_LOGIC_VECTOR(31 downto 0);
begin
TO_WRITE: process (mem_to_reg)
begin
    if mem_to_reg = '0' then
        write_back <= res_ALU;
    else
        write_back <= out_dataOut;
    end if;
end process;
CRAM: RAM port map ( clk => clk,
                     enable => enable,
                     writeEnable => mem_write,
                     readEnable => mem_read,
                     address => res_ALU,
                     dataIn => regFile_rt,
                     dataOut => out_dataOut);
end Behavioral;
```

Chapter 6

Testing and validation

6.1 Overview

In order to ease the debugging process, I tested each component individually. I tried to cover every corner case I could think of, in order to make sure they were working properly.

In the following section I will presented the code for each testbench, a table with the inputs, expected outputs and actual outputs and a picture with the wave form of the working unit under test. The testbench listings will only contain the signal assignments, due to the fact that most of that code is repetitive. I will skip over the instatiation of the component and their declarations in each architecture.

Remark 6.1.1: Signal identification

For a better understanding of the testbenches, the signals used in each have the same name as the inputs and outputs of the component that is being tested, with the suffix `in_` or `out_`.

For the bigger components of the CPU (i.e. IFU, IDU, EXU, MEMU), I will present only the testbench code and the waveform since each of these components is nothing more than an augmentation of the components that were already tested.

6.2 Simulation

6.2.1 Memory

6.2.1.1 Register File

Listing 6.2.1: Register File Testbench

```
GENERATE_CLOCK: in_clk <= not in_clk after 1 ps;
in_readAdr1   <= "00001",
                 "10000" after  5 ps,
                 "00001" after 11 ps;
in_readAdr2   <= "00010",
                 "10001" after  5 ps,
                 "00010" after 11 ps;
in_readAdr3   <= "00011",
```

```
                  "10010" after  5 ps,
                  "00011" after 11 ps;
in_writeAdr1   <= "00001";
in_writeAdr2   <= "00010";
in_writeAdr3   <= "00011";
in_writeData1  <= X"0000_0000",
                  X"FFFF_FFFE" after  7 ps,
                  X"0000_0000" after 11 ps;
in_writeData2  <= X"0000_0000",
                  X"FFFF_FFFD" after  9 ps,
                  X"0000_0000" after 11 ps;
in_writeData3  <= X"0000_0000",
                  X"FFFF_FFFC" after  9 ps,
                  X"0000_0000" after 11 ps;
in_regWr1  <= '1' after  7 ps,
              '0' after  9 ps;
in_regWr2  <= '1' after  9 ps,
              '0' after 11 ps;
in_regWr3  <= '1' after  9 ps,
              '0' after 11 ps;
```

Figure 6.2.1: Register File Waveform

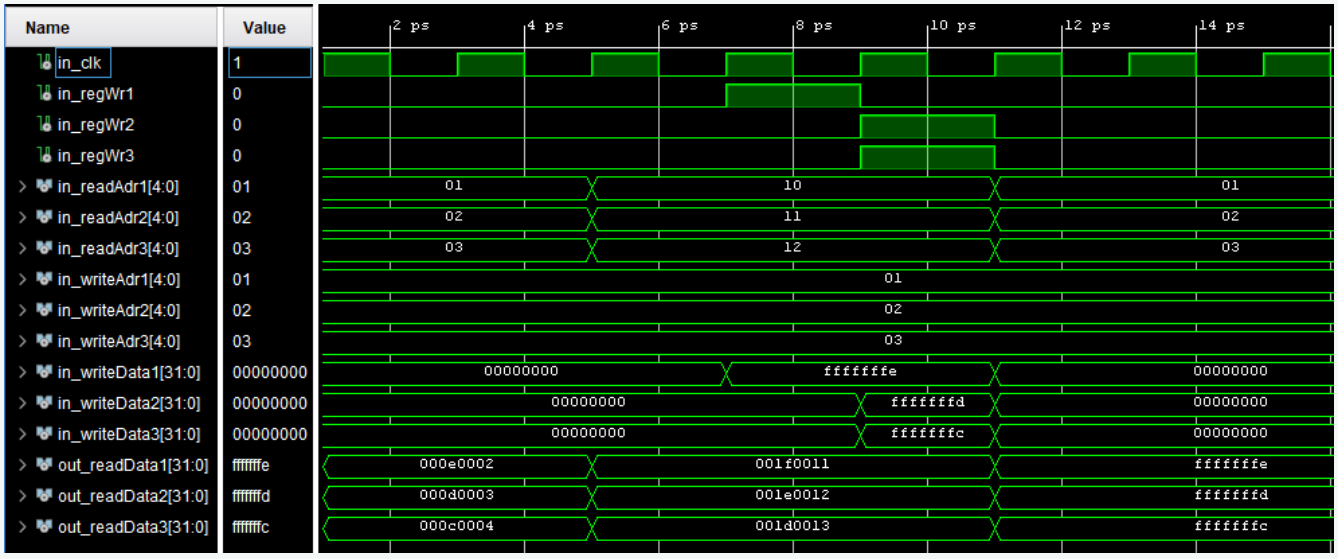

Figure 6.1: Register File Waveform

Table 6.2.1: Resgister File Testing

This test covers a total of 4 scenarios.

Scenario: Read

INPUT: RA1=x01, RA2=x02, RA3=x03
EXPECTED OUTPUT: RD1=x000E0002, RD2=x000D0003, RD3=x000C0004
ACTUAL OUTPUT: RD1=x000E0002, RD2=x000D0003, RD3=x000C0004

Scenario: Read from a different address

INPUT: RA1=x11, RA2=x12, RA3=x13
EXPECTED OUTPUT: RD1=x001F0011, RD2=x001E0012, RD3=x001D0013
ACTUAL OUTPUT: RD1=x001F0011, RD2=x001E0012, RD3=x001D0013

Scenario: Write

INPUT: WA1=x01, WA2=x02, WA3=x03
WD1=xFFFFFFFE, WD2=xFFFFFFFD, WD3=xFFFFFFFC
EXPECTED OUTPUT: -
ACTUAL OUTPUT: -

Scenario: Read from the written address

INPUT: RA1=x01, RA2=x02, RA3=x03
EXPECTED OUTPUT: RD1=xFFFFFFFE, RD2=xFFFFFFFD, RD3=xFFFFFFFC
ACTUAL OUTPUT: RD1=xFFFFFFFE, RD2=xFFFFFFFD, RD3=xFFFFFFFC

Table 6.1: Resgister File Testing

6.2.1.2 ROM

Listing 6.2.2: Read Only Memory Testbench

```
in_addrs <= X"0000_0000",
            X"0000_0001" after 1 ps,
            X"0000_0002" after 2 ps;
```

Figure 6.2.2: Read Only Memory Waveform

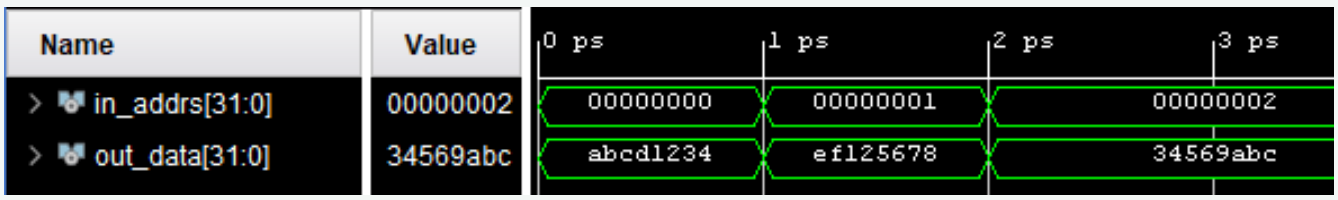

Figure 6.2: Read Only Memory Waveform

Table 6.2.2: Read Only Memory Testing

This test covers a total of 2 scenarios.

Scenario: Read

INPUT: addrs=x00000000
EXPECTED OUTPUT: data=xABCD1234
ACTUAL OUTPUT: data=xABCD1234

Scenario: Read from a different address

INPUT: addrs=x00000001
EXPECTED OUTPUT: data=xEF125678
ACTUAL OUTPUT: data=xEF125678

Table 6.2: Read Only Memory Testing

6.2.1.3 RAM

Listing 6.2.3: Random Access Memory Testbench

```
GENERATE_CLOCK: in_clk <= not in_clk after 1 ps;
in_writeEnable <= '1', '0' after 3 ps;
in_readEnable <= '0', '1' after 2 ps;
in_address <= X"0000_1234";
in_dataIn <= X"2345_BCDE";
```

Figure 6.2.3: Random Access Memory Waveform

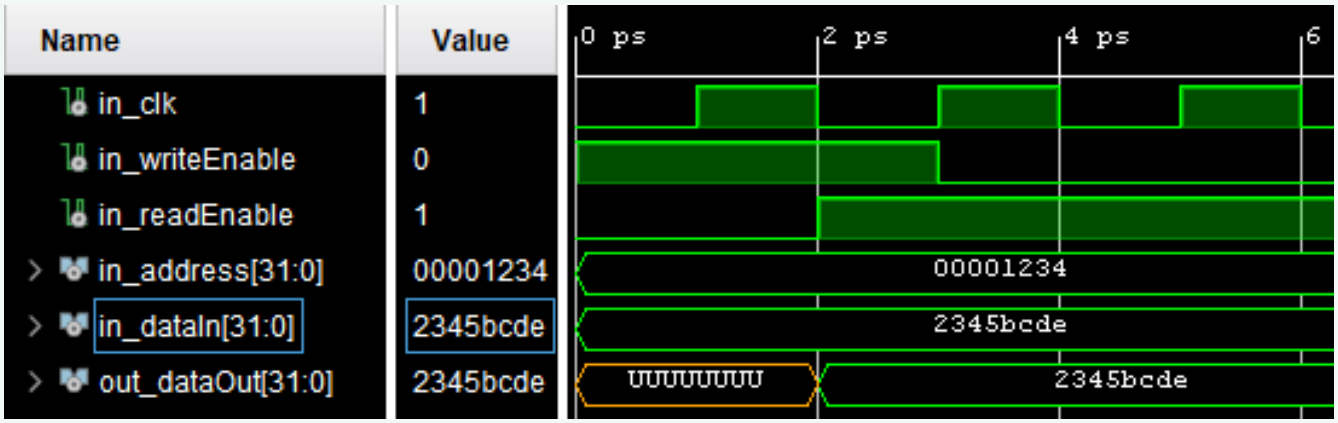

Figure 6.3: Random Access Memory Waveform

Table 6.2.3: Random Access Memory Testing

This test covers a total of 2 scenarios.

Scenario: Write

INPUT: address=x00001234
dataIn=x2345BCDE
EXPECTED OUTPUT: -
ACTUAL OUTPUT: -

Scenario: Read from the written address

INPUT: address=x00001234
EXPECTED OUTPUT: dataOut=x2345BCDE
ACTUAL OUTPUT: dataOut=x2345BCDE

Table 6.3: Random Access Memory Testing

6.2.2 Miscellaneous

6.2.2.1 Seven Segment Display Controller

Listing 6.2.4: Seven Segment Display Controller Testbench

```
GENERATE_CLOCK: in_clk <= not in_clk after 1 ps;
in_digits <= X"ABCD",
             X"1234" after 1 ns;
```

Figure 6.2.4: Seven Segment Display Controller Waveform

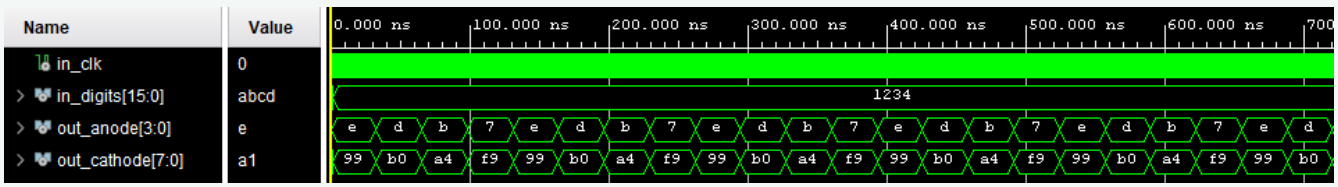

Figure 6.4: Seven Segment Display Controller Waveform

Table 6.2.4: Seven Segment Display Controller Testing

This test covers a total of 1 scenario.

Scenario: Display number

INPUT: address=x1234
EXPECTED OUTPUT: cycles through the same 4 values for the anode and cathode
ACTUAL OUTPUT: cycles through the same 4 values for the anode and cathode

Table 6.4: Seven Segment Display Controller Testing

6.2.2.2 Arithmetic Logic Unit

Listing 6.2.5: Arithmetic Logic Unit Testbench

```
in_shift_am <= X"4";
in_shift_dir <= "00" after  5 ps,
                "01" after  6 ps,
                "10" after  7 ps,
                "11" after  8 ps;
in_operation <= X"0",
                X"1" after  1 ps,
                X"2" after  2 ps,
                X"3" after  3 ps,
                X"4" after  4 ps,
                X"5" after  5 ps,
                X"6" after  9 ps,
                X"8" after 10 ps,
                X"9" after 11 ps,
                X"A" after 12 ps,
                X"B" after 13 ps,
                X"C" after 14 ps,
                X"D" after 15 ps,
                X"E" after 16 ps,
                X"F" after 17 ps;
in_term1 <= X"1111_2222",
            X"0000_0088" after  1 ps,
            X"FFFF_FFFF" after  2 ps,
            X"2345_2345" after  3 ps,
            X"4" after  4 ps,
            X"2345_2345" after 10 ps;
in_term2 <= X"2222_3333",
            X"0000_00BB" after  1 ps,
            X"0000_0002" after  2 ps,
            X"2345_2345" after  3 ps,
            X"3" after  4 ps,
            X"CCCC_CCCC" after 10 ps;
in_term3 <= X"2222_3333" after 10 ps,
            X"2345_2345" after 11 ps,
            X"0000_0002" after 12 ps,
            X"2345_2345" after 13 ps,
            X"1111_2222" after 14 ps;
```

Figure 6.2.5: Arithmetic Logic Unit Waveform 1

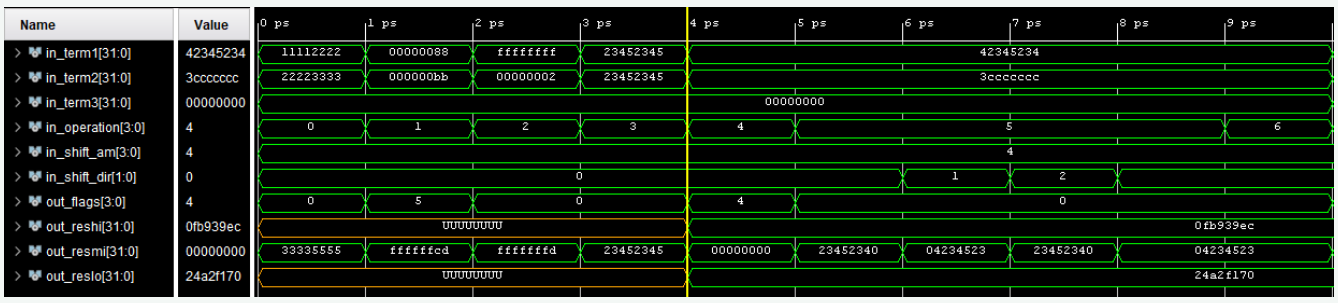

Figure 6.5: Arithmetic Logic Unit Waveform 1

Figure 6.2.6: Arithmetic Logic Unit Waveform 2

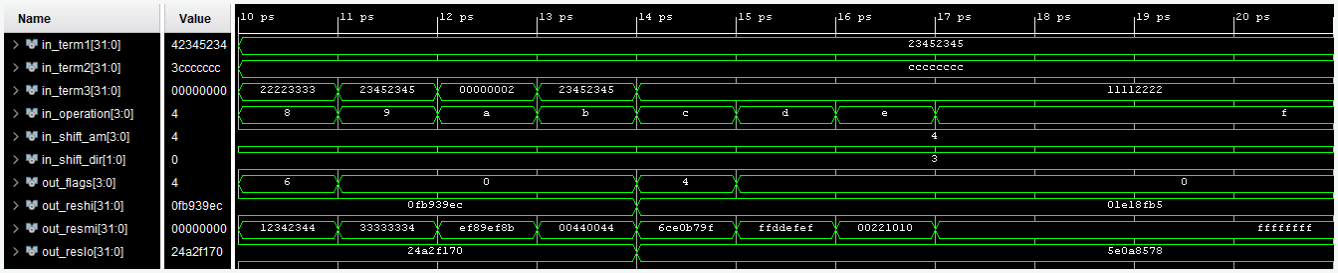

Figure 6.6: Arithmetic Logic Unit Waveform 2

Table 6.2.5: Arithmetic Logic Unit Testing

This test covers a total of 37 scenarios.

Scenario: Add - normal

INPUT: term1=x77, term2=x55, term3=x0
EXPECTED OUTPUT: regM=xCC, flags=x0
ACTUAL OUTPUT: regM=xCC, flags=x0

Scenario: Add - overflow

INPUT: term1=xFFFFFFFF, term2=x3, term3=x0
EXPECTED OUTPUT: regM=x2, flags=x6
ACTUAL OUTPUT: regM=x2, flags=x6

Scenario: Add - zero

INPUT: term1=x0, term2=x0, term3=x0
EXPECTED OUTPUT: regM=x0, flags=x8
ACTUAL OUTPUT: regM=x0, flags=x8

Scenario: Subtraction - normal

INPUT: term1=xCC, term2=x55, term3=x0
EXPECTED OUTPUT: regM=x77, flags=x0
ACTUAL OUTPUT: regM=x77, flags=x0

Scenario: Subtraction - negative

INPUT: term1=x88, term2=xBB, term3=x0
EXPECTED OUTPUT: regM=xFFFFFFCD, flags=x5
ACTUAL OUTPUT: regM=xFFFFFFCD, flags=x5

Scenario: Subtraction - zero

INPUT: term1=xAAAA, term2=xAAAA, term3=x0
EXPECTED OUTPUT: regM=x0, flags=x8
ACTUAL OUTPUT: regM=x0, flags=x8

Scenario: Xor - normal

INPUT: term1=xACAC, term2=x1111, term3=x0
EXPECTED OUTPUT: regM=xBDBD, flags=x0
ACTUAL OUTPUT: regM=BDBD, flags=x0

Scenario: Xor - zero

INPUT: term1=xAAAA, term2=xAAAA, term3=x0
EXPECTED OUTPUT: regM=x0, flags=x8
ACTUAL OUTPUT: regM=x0, flags=x8

Scenario: And - normal

INPUT: term1=xAAAA, term2=x4242, term3=x0
EXPECTED OUTPUT: regM=x202, flags=x0
ACTUAL OUTPUT: regM=x202, flags=x0

Scenario: And - zero

INPUT: term1=xFFFF, term2=x0, term3=x0
EXPECTED OUTPUT: regM=x0, flags=x8
ACTUAL OUTPUT: regM=x0, flags=x8

Scenario: Multiplication - normal

INPUT: term1=x77, term2=x11, term3=x0
EXPECTED OUTPUT: regH=x0, regL=x7E7, flags=x0
ACTUAL OUTPUT: regH=x0, regL=x7E7, flags=x0

Scenario: Multiplication - overflow

INPUT: term1=xCBADAAAA, term2=x22223333, term3=x0
EXPECTED OUTPUT: regH=x1B284677, regL=x6B2CDDDE, flags=x6
ACTUAL OUTPUT: regH=x1B284677, regL=x6B2CDDDE, flags=x6

Scenario: Shift left logical by 4

INPUT: term1=xBBBB, term2=x0, term3=x0
EXPECTED OUTPUT: regM=xBBB0, flags=x0
ACTUAL OUTPUT: regM=xBBB0, flags=x0

Scenario: Shift right logical by 4

INPUT: term1=xBBBB, term2=x0, term3=x0
EXPECTED OUTPUT: regM=x0BBB, flags=x0
ACTUAL OUTPUT: regM=x0BBB, flags=x0

Scenario: Shift left arithmetic by 4

INPUT: term1=xBBBB, term2=x0, term3=x0
EXPECTED OUTPUT: regM=xBBBF, flags=x0
ACTUAL OUTPUT: regM=xBBBF, flags=x0

Scenario: Shift right arithmetic by 4

INPUT: term1=xBBBB, term2=x0, term3=x0
EXPECTED OUTPUT: regM=xFBBB, flags=x0
ACTUAL OUTPUT: regM=xFBBB, flags=x0

Scenario: Compare - less

INPUT: term1=x234, term2=x432, term3=x0
EXPECTED OUTPUT: flags=x1
ACTUAL OUTPUT: flags=x1

Scenario: Compare - equal

INPUT: term1=x234, term2=x432, term3=x0
EXPECTED OUTPUT: flags=x8
ACTUAL OUTPUT: flags=x8

Scenario: Compare - greater

INPUT: term1=x234, term2=x432, term3=x0
EXPECTED OUTPUT: flags=x0
ACTUAL OUTPUT: flags=x0

Scenario: Add 3 terms - normal

INPUT: term1=x77, term2=x55, term3=x22
EXPECTED OUTPUT: regM=xEE, flags=x0
ACTUAL OUTPUT: regM=xEE, flags=x0

Scenario: Add 3 terms - overflow

INPUT: term1=xEEEEEEEE, term2=x11111111, term3=x22222222
EXPECTED OUTPUT: regM=x22222221, flags=x6
ACTUAL OUTPUT: regM=x22222221, flags=x6

Scenario: Add 3 terms- zero

INPUT: term1=x0, term2=x0, term3=x0
EXPECTED OUTPUT: regM=x0, flags=x8
ACTUAL OUTPUT: regM=x0, flags=x8

Scenario: Subtraction 3 terms - normal

INPUT: term1=xABAB, term2=x123, term3=x432
EXPECTED OUTPUT: regM=xA656, flags=x0
ACTUAL OUTPUT: regM=xA656, flags=x0

Scenario: Subtraction 3 terms - negative

INPUT: term1=x123, term2=xABAB, term3=x321
EXPECTED OUTPUT: regM=FFFF5257, flags=x5
ACTUAL OUTPUT: regM=FFFF5257, flags=x5

Scenario: Subtraction 3 terms - zero

INPUT: term1=xAAAA, term2=x8888, term3=x2222
EXPECTED OUTPUT: regM=x0, flags=x8
ACTUAL OUTPUT: regM=x0, flags=x8

Scenario: Xor 3 terms - normal

INPUT: term1=xEEEE, term2=x1111, term3=x8563
EXPECTED OUTPUT: regM=x7A9C, flags=x0
ACTUAL OUTPUT: regM=x7A9C, flags=x0

Scenario: Xor 3 terms - zero

INPUT: term1=x5858, term2=x1234, term3=x4A6C
EXPECTED OUTPUT: regM=x0, flags=x8
ACTUAL OUTPUT: regM=x0, flags=x8

Scenario: And 3 terms - normal

INPUT: term1=x12, term2=x14, term3=x78
EXPECTED OUTPUT: regM=x10, flags=x0
ACTUAL OUTPUT: regM=x10, flags=x0

Scenario: And 3 terms - zero

INPUT: term1=x23, term2=x44, term3=x0
EXPECTED OUTPUT: regM=x0, flags=x8
ACTUAL OUTPUT: regM=x0, flags=x8

Scenario: Multiplication - normal

INPUT: term1=x77, term2=x44, term3=x11
EXPECTED OUTPUT: regH=x0, regM=0, regL=x2195C, flags=x0
ACTUAL OUTPUT: regH=x0, regM=0, regL=x2195C, flags=x0

Scenario: Multiplication - overflow

INPUT: term1=xAAAAAAAA, term2=x58585858, term3=x25862586
EXPECTED OUTPUT: regH=x8A20AA3, regM=F2BFEEBC, regL=x049E06A0, flags=x6
ACTUAL OUTPUT: regH=x8A20AA3, regM=F2BFEEBC, regL=x049E06A0, flags=x6

Scenario: Or 3 terms - normal

INPUT: term1=x55, term2=x1234, term3=x321
EXPECTED OUTPUT: regM=x1375, flags=x0
ACTUAL OUTPUT: regM=x1375, flags=x0

Scenario: Or 3 terms - zero

INPUT: term1=x0, term2=x0, term3=x0
EXPECTED OUTPUT: regM=x0, flags=x8
ACTUAL OUTPUT: regM=x0, flags=x8

Scenario: Nor 3 terms - normal

INPUT: term1=x55, term2=x1234, term3=x321
EXPECTED OUTPUT: regM=x1054, flags=x0
ACTUAL OUTPUT: regM=x1054, flags=x0

Scenario: Nor 3 terms - zero

INPUT: term1=x0, term2=x0, term3=x0
EXPECTED OUTPUT: regM=x0, flags=x8
ACTUAL OUTPUT: regM=x0, flags=x8

Scenario: Nand 3 terms - normal

INPUT: term1=x1234, term2=x4321, term3=x3321
EXPECTED OUTPUT: regM=xFFFFCEFE, flags=x0
ACTUAL OUTPUT: regM=xFFFFCEFE, flags=x0

Scenario: Nand 3 terms - zero

INPUT: term1=x0, term2=x25, term3=xFFFFFFFF
EXPECTED OUTPUT: regM=x0, flags=x8
ACTUAL OUTPUT: regM=x0, flags=x8

Table 6.5: Seven Segment Display Controller Testing

6.2.2.3 Arithmetic Logic Unit Controller

Listing 6.2.6: Arithmetic Logic Unit Controller Testbench

```
in_flags <= "1111",
            "0000" after 3 ps;
in_op_code <= "0000",
              "0001" after 3 ps,
              "0010" after 5 ps,
              "0011" after 6 ps,
              "0101" after 7 ps;
in_func_code <= "0000",
                "0001" after 1 ps,
                "0010" after 2 ps,
                "0000" after 3 ps,
                "0001" after 4 ps;
```

Figure 6.2.7: Arithmetic Logic Unit Controller Waveform

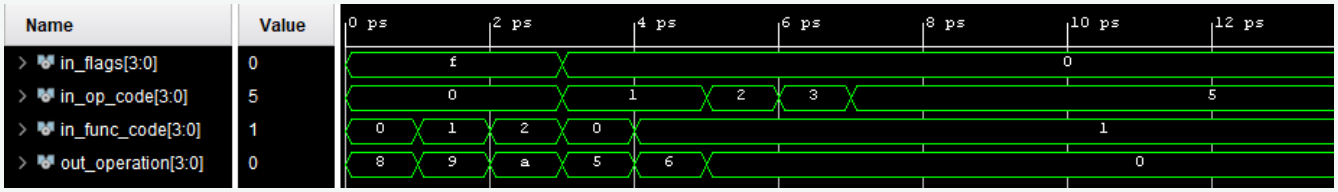

Figure 6.7: Arithmetic Logic Unit Controller Waveform

Table 6.2.6: Arithmetic Logic Unit Controller Testing

This test covers a total of 1 scenario.

Scenario: Get operation

INPUT: flags=xF, op_code=x0, func=x0
EXPECTED OUTPUT: operation=x0
ACTUAL OUTPUT: operation=x0

Table 6.6: Arithmetic Logic Unit Controller Testing

6.2.3 Instruction Fetch Unit

Listing 6.2.7: Instruction Fetch Unit Testbench

```
GENERATE_CLOCK : in_clk <= not in_clk after 1 ps;
GENERATE_BUTTON: in_enable <= '1',
                              '0' after 15 ps,
                              '1' after 21 ps;
GENERATE_RESET : in_reset <= '0',
                             '1' after 5 ps,
                             '0' after 7 ps,
                             '1' after 13 ps,
                             '0' after 15 ps;
GENERATE_JUMP_A: in_jump_address <= X"0000_000A";
GENERATE_JUMP_C: in_jump_ctrl <= '0',
                                 '1' after 23 ps,
                                 '0' after 25 ps;
```

Figure 6.2.8: Instruction Fetch Unit Waveform

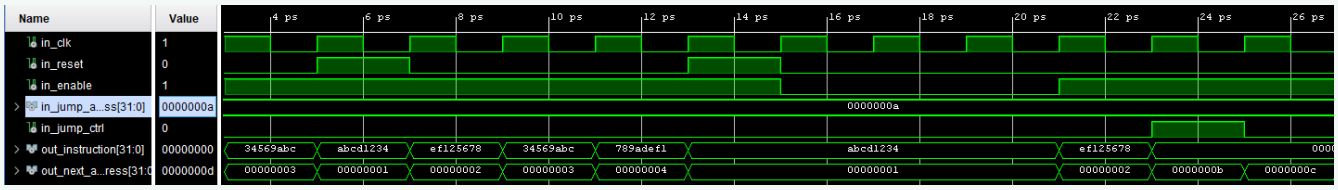

Figure 6.8: Instruction Fetch Unit Waveform

6.2.4 Instruction Decode Unit

Listing 6.2.8: Instruction Decode Unit Testbench

```
GENERATE_CLOCK : in_clk <= not in_clk after 1 ps;
in_instruction <= B"0000_00001_00010_00000_00000_0011_0000",
                  B"0000_00010_00011_0000_10_1101_0000_0001" after 5 ps,
                  B"0000_00001_00010_10101010101010_0010" after 9 ps,
                  B"0000_1_00000000000001111110111_0100" after 13 ps,
                  B"1111_00011_00000_00010_00001_0110_0000" after 17 ps,
                  B"1011_00001_00010_00011_00000_0000_0000" after 21 ps;
```

Figure 6.2.9: Instruction Decode Unit Waveform

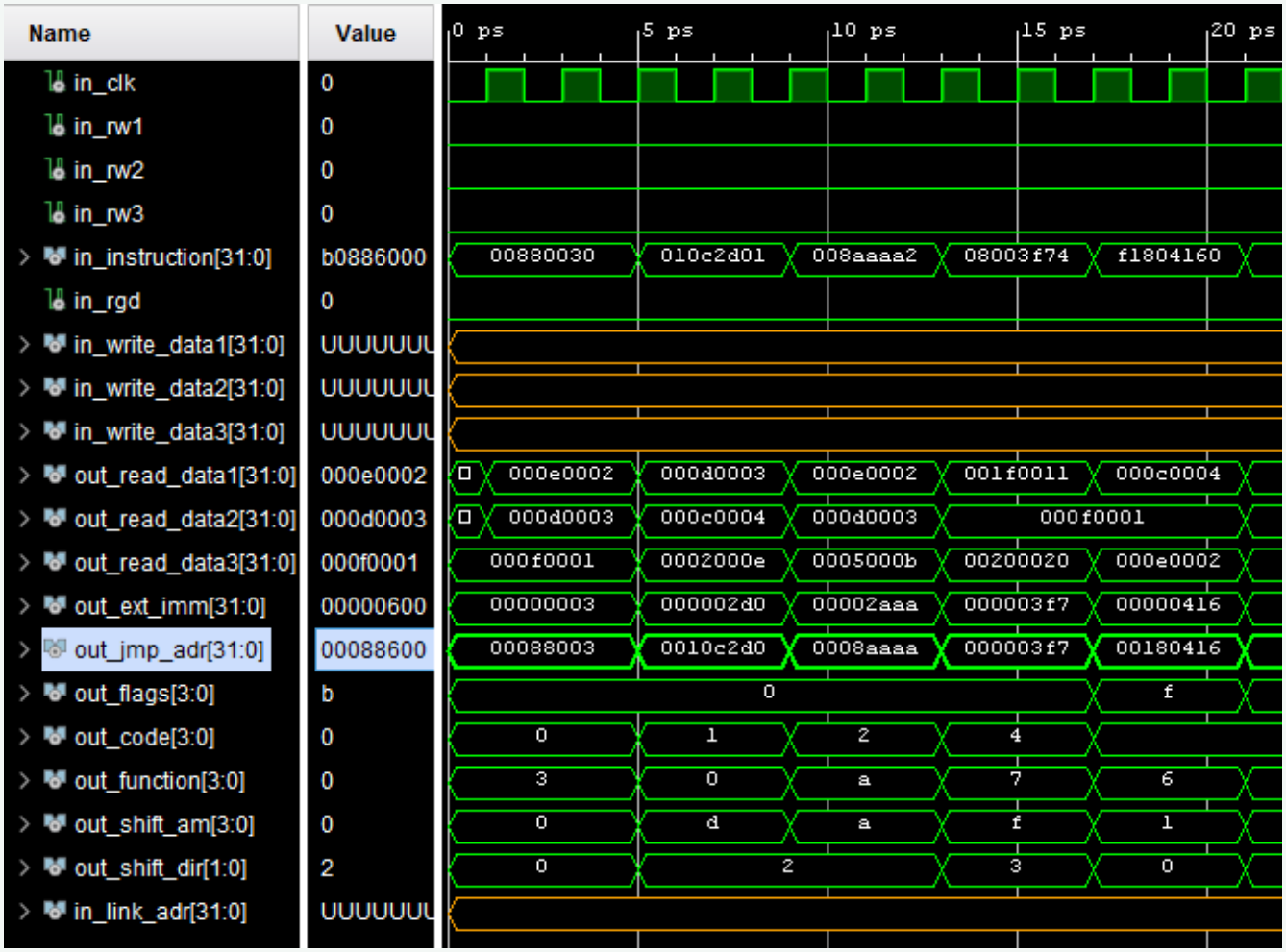

Figure 6.9: Instruction Decode Unit Waveform

6.2.5 Control Unit

Listing 6.2.9: Control Unit Testbench

```
in_flags <= "1100",
            "0000" after 6 ps;
in_cond <= "1111",
           "1100" after 5 ps;
in_code <= "0000";
in_func <= "0100",
           "0000" after 5 ps;
```

Figure 6.2.10: Control Unit Waveform

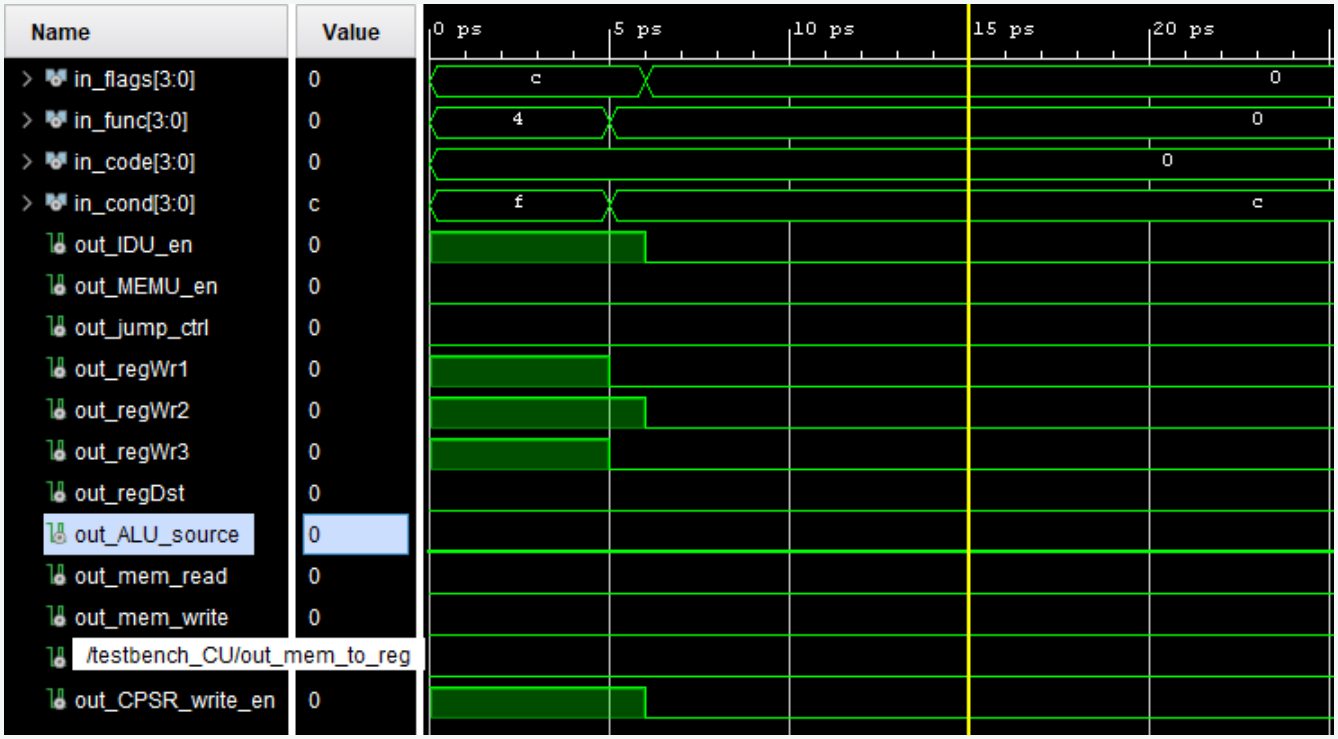

Figure 6.10: Control Unit Waveform

6.2.6 Execution Unit

Listing 6.2.10: Execution Unit Testbench

```
in_in_flags <= "0000";
in_in_op_code <= "0000";
in_in_func_code <= "0000";
in_term1 <= X"1234_1234";
in_term2 <= X"1234_1234";
in_immediate <= X"ABCD_ABCD";
in_ALUSrc <= '0', '1' after 5 ps;
```

Figure 6.2.11: Execution Unit Waveform

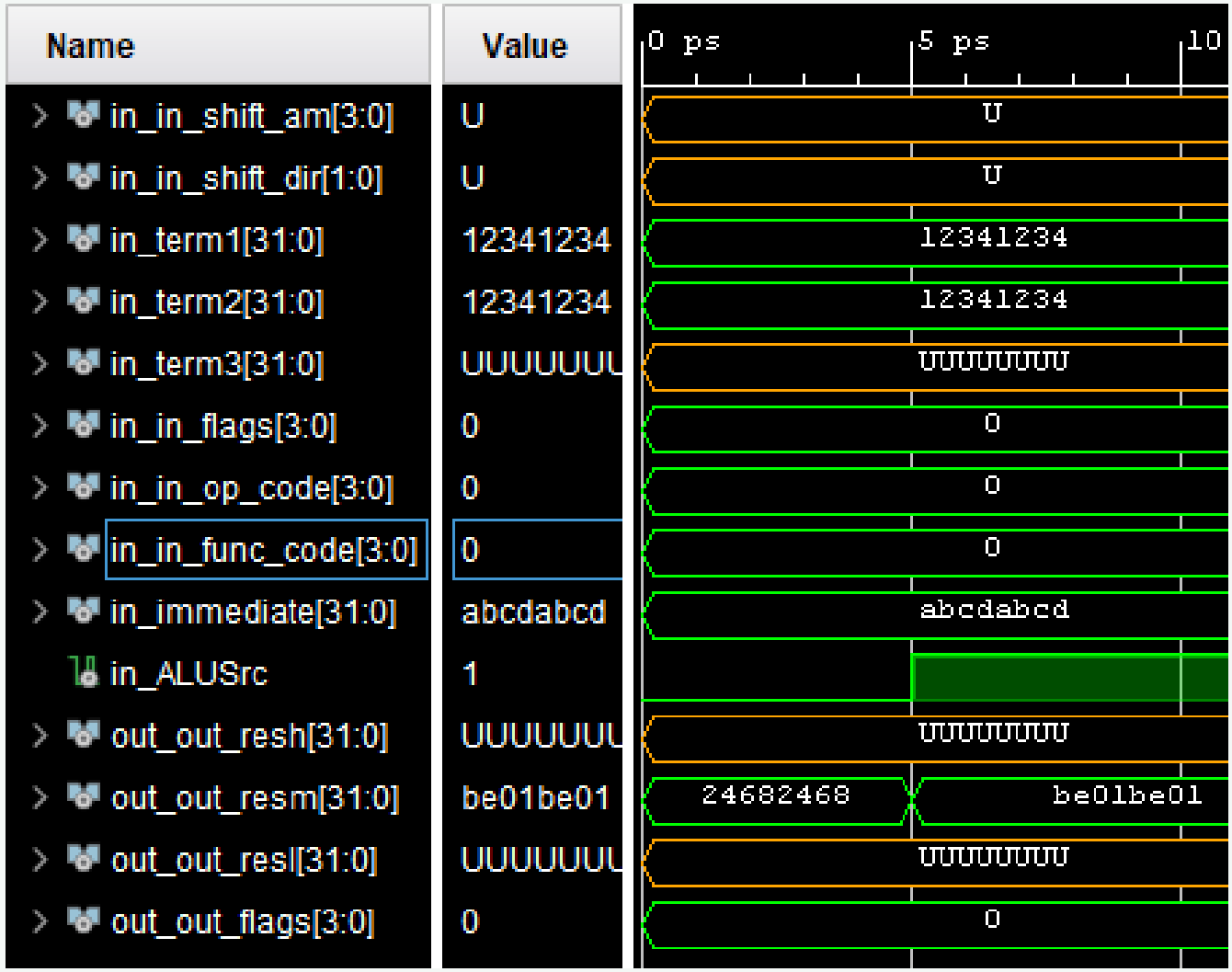

Figure 6.11: Execution Unit Waveform

6.2.7 Memory Unit

Listing 6.2.11: Memory Unit Testbench

```
GENERATE_CLOCK : in_clk <= not in_clk after 1 ps;
in_mem_read <= '1';
in_mem_write <= '1';
in_mem_to_reg <= '0', '1' after 5 ps;
in_res_ALU <= X"1234_ABCD";
in_regFile_rt <= X"ABCD_5678";
```

Figure 6.2.12: Memory Unit Waveform

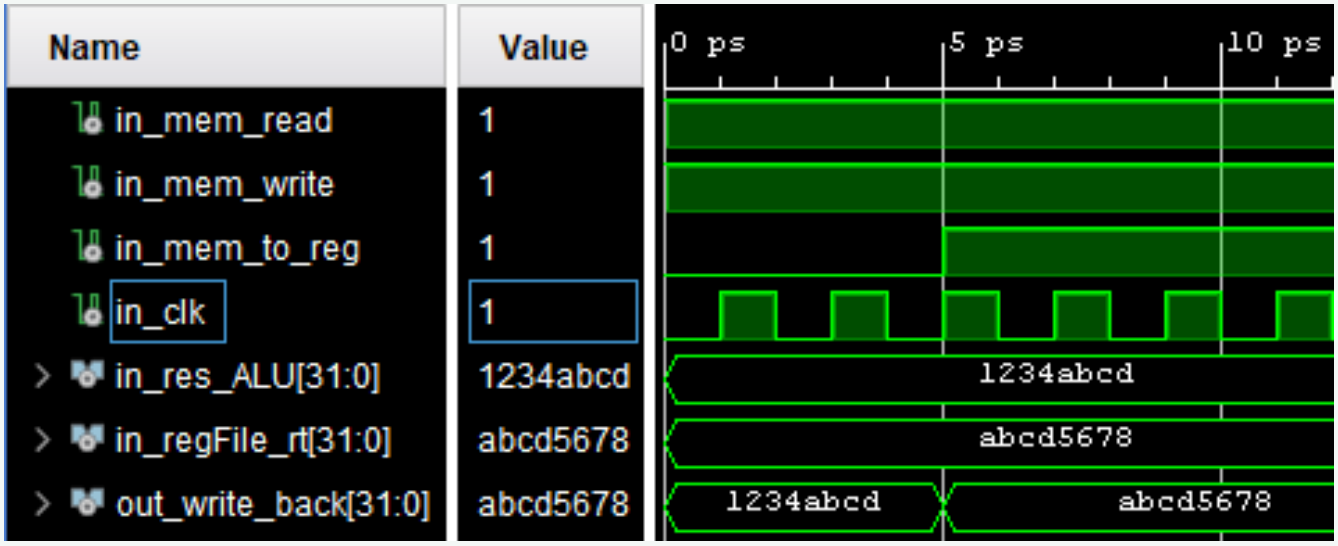

Figure 6.12: Memory Unit Waveform

Chapter 7

Conclusions

Desiging a basic CPU is a relatively easy task. The design steps are already known and are easy to follow. I wanted to make my CPU more complex by combining features from both the MIPS processor and the ARM processor. As such, the instruction fields and types are similar to the ones in the MIPS CPU and the advantage of condition fields for each instructions are taken from the ARM CPU. I also decided to design instructions capable of operating with 3 operands. All these tasks proved to be successful.

The components of the CPU are almost the same as the ones in the MIPS processor, namely IFU, IDU, EXU, MEMU and a CU. My CPU has a controller for the ALU in the EXU, instead of the signals being generated in the CU. I consider that provides the reader with a clearer view of the whole circuit. Other than the components mentioned before, I designed the CPSR which holds the flags for the conditional instructions.

Due to time limitations, I did not succed at implementing the LOOP CISC operation, nor the mini-assembler. However, the design of this CPU started with their existence in mind and at each step I made sure to leave an "empty socket" for these modules to connect in. As such, both the implementation of the LOOP operation and the mini-assambler can be seen as future improvements of my CPU.

Bibliography

[1] *http : //www.csit − sun.pub.ro/courses/cn*1*CA/MicroprocessorDesign_*20111005*.pdf*

[2] *https : //ro.wikipedia.org/wiki/Arhitectur*%*C*4%83*_RISC*

[3] *https : //www.geeksforgeeks.org/differences − between − single − cycle − and − multiple − cycle − datapath/*

[4] *https : //iitd − plos.github.io/col*718*/ref/arm − instructionset.pdf*

[5] *http : //www.math − cs.gordon.edu/courses/cps*311*/handouts −* 2019*/MIPS*%20*ISA.pdf*

[6] *https : //www.microcontrollertips.com/difference − between − von − neumann − and − harvard − architectures/*

[7] *https : //levelup.gitconnected.com/little − endian − vs − big − endian − eb*2*a*2*c*3*a*9135

[8] *http : //www.csit − sun.pub.ro/courses/cn*2*/Carte_H&P/H*%20*and*%20*P/chapter_*2*.pdf*

[9] *https : //en.wikipedia.org/wiki/Instruction_set_architecture#Number_of_operands*

List of Tables

List of Figures

Listings from VHDL