Universiteit Leiden

Opleiding Informatica

Design, Analysis, and Optimization of an Embedded Processor

Name:	Ruben Meerkerk
Studentnr:	s1219677
Date:	31/08/2016
1st supervisor:	Dr. T.P. Stefanov
2nd supervisor:	Dr. K.F.D. Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Design, Analysis, and Optimization of an Embedded Processor

Ruben Meerkerk

Abstract

In this thesis, we design, implement, and compare two processor designs, each at a different level of abstraction, for the same processor. The goal is to analyse which design level of abstraction leads to a more efficient and effective processor design. The two abstraction levels we investigate are gate-level and RTL. The gate-level designs were made using Schematics and the RTL (register-transfer level) designs using a HDL (hardware description language) called VHDL (VHSIC Hardware Description Language). The processor we designed, is specified in this thesis. The processor specifications are the same for both designs, but the design choices we made, differ for some components. The processor designs we made, were implemented using software tools called Xilinx ISE Design Suite. We compared the two implemented processor designs on their maximum clock frequency, the amount of hardware resources used, and the amount of effort needed to design the processor. We managed to optimize the Schematic design to use less hardware resources than the VHDL design, but the VHDL design required less effort to make and had a faster clock speed. This leads us to the conclusion that higher abstraction levels are more suitable for designing an embedded processor.

Acknowledgements

This thesis is the result of research performed at the the Leiden Institute of Advanced Computer Science (LIACS) - Leiden University. I would like to thank my supervisor Todor Stefanov, who was always willing to help me whenever I had questions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Designing an embedded processor can be done at different levels of abstraction. This is illustrated in Figure 1.1. A design made at a higher level of abstraction is known to have a lot of advantages:

- the structure of the design is often more clear,
- less work (and therefore less time) is needed to make the design,
- the design is more flexible for large changes.

One of the levels of abstraction mentioned in Figure 1.1 is called RTL [Vah09], which stands for register-transfer level. RTL models a synchronous digital circuit in terms of the flow of digital signals between registers, and the logical operations performed on these signals. The Register-transfer-level abstraction is mainly used in hardware description languages (HDLs) like Verilog and VHDL (VHSIC Hardware Description Language) [Smi16] to create high-level representations of a circuit, from which lower-level representations and ultimately actual implementation can be derived. Design at the RTL level is a typical practice in modern digital design.

Below RTL is the gate-level of abstraction [gat16]. Designing at gate-level requires the use of logic gates. A logic gate is an idealized or physical device implementing a Boolean function. The Boolean function being implemented can have multiple inputs, but has only one output. It has become less common to design embedded processors at gate level. The reason for this is that a lot more work is required to design complex circuits using logic gates than RTL.

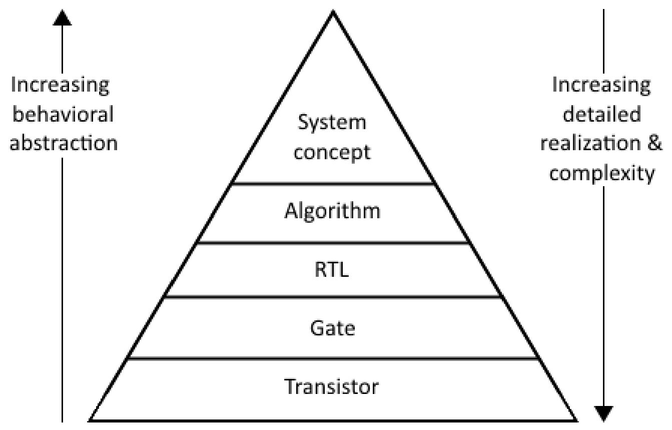


Figure 1.1: Behavioural level of abstraction pyramid. [Smi16]

1.1 Problem Description

In this thesis, we will design one and the same embedded processor two times starting at two different levels of abstraction. The levels of abstraction are RTL and gate level. The goal is to find out which level of abstraction is more suitable for designing an efficient embedded processor, with the least amount of effort. Also, we should investigate how the level of abstraction affects the quality of the final implementation of the processor.

We will design the embedded processor at RTL level using a hardware description language called VHDL (VHSIC Hardware Description Language). The VHDL code will be synthesized and implemented using Xilinx ISE Design Suite [ise12]. At the gate level, we will use Schematics, which will be synthesized and implemented with the same software. After implementing both designs, we will compare them using reports generated by the Xilinx ISE software. We will look at the amount of effort needed to design the processor, the amount of hardware resources used, and the maximum clock frequency at which the processor can run.

1.2 Thesis Contributions

In this thesis, we show how we designed an embedded processor, starting with a specification describing the functional behaviour of the processor. With the same specification, we made two different designs starting at two different abstraction levels, i.e. RTL and gate level. After designing and testing the processor designs, we were able to synthesize and implement the two designs, which made it possible for us to evaluate and compare the two designs. We looked at the amount of effort it took for each design to be made, how much hardware is required for each design, and the clock frequency of each design. By doing this, we were able to conclude which abstraction level for designing an embedded processor would be more suitable.

1.3 Related Work

In the past several similar studies have been done around this topic. For example a paper written by Pecheux et al. [PLV05] shows the differences between the two mixed-signal hardware description languages, VHDL-AMS and Verilog-AMS. They did not design a processor, like we did, but an airbag system, to compare the two hardware description languages.

A different paper written by Jackson and Hannah [JH93] researches and compares different adder designs using Verilog. The paper gave some insight on the use of Verilog, which is a hardware description language similar to VHDL.

The paper written by Sanders [Sau89] compares declarative languages and imperative languages. It also describes the benefits of synthesizing from a hardware description language. But unlike this thesis, it compares two different methods of design with the same abstraction level. It still closely relates to this thesis by making comparisons between different methods of designing hardware, even though the same abstraction level is used.

1.4 Thesis Overview

The remaining part of the thesis is organized as follows:

Chapter 2 explains how hardware can be designed using the software Xilinx ISE Design Suite at different levels of abstraction.

In Chapter 3, we specify how the processor should work. We describe what instructions the processor must be able to perform and how to perform them.

Chapter 4 shows how the embedded processor is designed using Schematics and VHDL. In this chapter, we also explain some design choices made during the design process to optimize the processor.

In Chapter 5, we evaluate the two processor designs by comparing them with each other. We compare them on clock speed, hardware resource usage, and amount of effort to design.

This thesis ends with Chapter 6. Here, we conclude our thesis with the major findings of our research.

Chapter 2

Background

This chapter explains how in general hardware can be designed at both RTL (high abstraction level) and gate level (low abstraction level). The next two sections (Section 2.1 and 2.2) will illustrate this by showing how a 5 to 1 multiplexer can be designed at each level of abstraction. The last section (Section 2.3) of this chapter givea a brief introduction to the software tool Xilinx ISE Design Suite.

As explained in the Chapter 1, RTL stands for register-transfer level. RTL models hardware by describing the flow of digital signals between registers and the logical operations performed on these signals. Designing at RTL level of abstraction is often realized using a hardware description language (HDL). When we design hardware with a HDL, we simulate both data transfer and data storage by assigning the data to variables. These variables can model both signals and registers. We can also define other variables that represent components. These component variables require signal/register variables as their input or output signals. A HDL often also contain various statements and data structures that are similar to a programming language.

Gate level is a level of abstraction below RTL and above the transistor level (see Figure 1.1). Designing hardware at gate level requires the use of logic gates. Logic gates are a physical implementation of Boolean functions, which can have multiple logical inputs, but only a single logical output. Examples of logic gates are AND, OR, XOR and NOT. By connecting the inputs and outputs of various logic gates, we can design various hardware components like adders, multiplexers and even registers.

A multiplexer is a hardware component that selects the value of one of its inputs to be its output. In Figure 2.1 we see the 5 to 1 multiplexer. The single bit inputs A, B, C, D, and E are the values the 5 to 1 multiplexer will choose from. The 3-bit signal S selects which of those single bit input values will be used as output

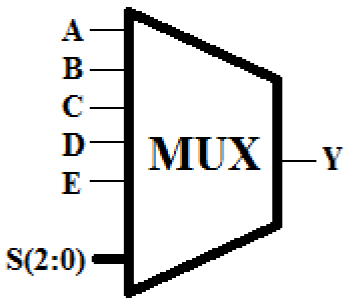


Figure 2.1: A 5 to 1 single bit multiplexer.

$S_2S_1S_0$	Y
000	A
001	B
010	C
011	D
100	E
xxx	x

Table 2.1: The expected behaviour of a 5 to 1 multiplexer

Y. Table 2.1 shows the expected behaviour of our multiplexer. It shows which value output Y will take for almost every value of S. The values of S that are not in Table 2.1 are represented by 'xxx'. The value of Y for these values does not matter to us, which is why we placed 'x' as value for Y. The symbol 'x' means "do not care" and is used when the behaviour for a hardware component for certain cases is undefined.

2.1 Gate-level design of a 5 to 1 multiplexer using Schematics

We make our gate level design of the multiplexer with Schematics. Schematic designs are made with logic gates, wires, and hardware components. We can make these hardware components ourselves using Schematics, but there are also a lot of standard components available in the symbol library of the ISE Design Suite. Nevertheless, every design we make using Schematics is still considered to be a gate-level design.

In Figure 2.2, we show a Schematic design of a 5 to 1 multiplexer. The design was purely made using logic gates. Designing hardware component on a low level of abstraction means that optimizations have to be made manually. To optimize this design, we choose to select input E whenever $S_2 = 1$. This behaviour, according to Table 2.1, is allowed, since all cases where $S_2 = 1$ either mean that $Y = E$ or undefined behaviour. Since we only used logic gates, we can simply describe the behaviour of this design using Boolean algebra. The Boolean equation equivalent to this design is:
$Y = S_2E + S_2'(S_1'S_0'A + S_1'S_0B + S_1S_0'C + S_1S_0D)$.

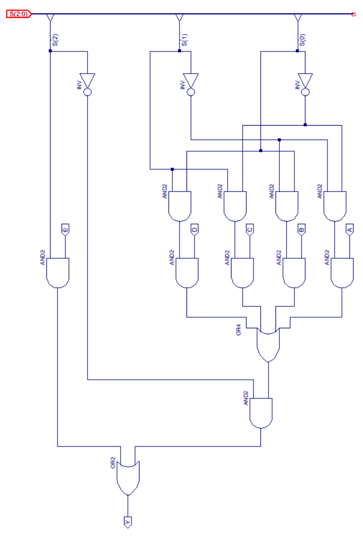


Figure 2.2: A 5 to 1 single bit multiplexer designed using Schematics.

Designing a hardware component at a low level of abstraction (gate level), means that all optimizations that need to be made, have to come from the designer. The Schematic design in Figure 2.2 clearly states which logic gates are used and how they interact with one another. The synthesizer tool could optimize the design by restructuring the hardware into a component with the same behaviour as the Schematic design. Unfortunately the behaviour of the Schematic design is not identical to the behaviour described in Table 2.1. The behaviour of the Schematic design is defined for all cases, while Table 2.1 shows case that are undefined. This means that the synthesizer tool could only optimize 5 to 1 multiplexers, that behave the same way as the Schematic Design (selecting input E whenever $S_2 = 1$), which is not necessarily optimal for the actual 5 to 1 multiplexer.

2.2 RTL-level design of a 5 to 1 multiplexer in VHDL

For the high-level abstraction design at RTL of our embedded processor we use VHDL [Smi16]. VHDL is a HDL that can be used to design hardware. A HDL is very similar to a programming language. It uses variables to model signals and registers. The data stored in these variables can be manipulated using boolean or arithmetic operators. In VHDL arrays can be defined and their indexes can be defined in both increasing and decreasing order, which is not common for most programming languages.

Figure 2.3 shows how a 5 to 1 multiplexer can be designed using VHDL. Lines 6 to 11 describe the input ports $(A, B, C, D, E,$ and $S)$ and line 11 describes the output port (Y) of the multiplexer. Note that input port S is defined as a std_logic_vector, while all other ports (both input and output) are std_logic. This is because S is

```
 1 library IEEE;
 2 use IEEE.STD_Logic_1164.all;
 3 use IEEE.Numeric_STD.all;
 4
 5 entity MULTIPLEXER is
 6   port( A in std_logic;
 7         B in std_logic;
 8         C in std_logic;
 9         D in std_logic;
10         E in std_logic;
11         S: in std_logic_vector(2 downto 0);
12         Y: out std_logic);
13 end entity MULTIPLEXER;
14
15 architecture Behaviour of MULTIPLEXER is
16 begin
17   Process(A,B,C,D,E,S)
18   begin
19     case S is
20       when "000" => Y <= A;
21       when "001" => Y <= B;
22       when "010" => Y <= C;
23       when "011" => Y <= D;
24       when "100" => Y <= E;
25       when others => Y <= '-';
26     end case;
27   end process;
28 end architecture Behaviour;
```

Figure 2.3: A 5 to 1 single bit multiplexer designed using VHDL.

three bits wide, while the others are single bit inputs. Lines 17 to 27 describe the behaviour of the multiplexer. We start describing its behaviour by defining a process that starts whenever the mentioned input signals at line 17 change value. In this case all input values of the multiplexer are mentioned on that line. The process itself is described on lines 19 to 26. Line 19 shows the beginning of a case statement, which uses input S to select a statement between lines 20 and 25. Lines 20 to 25 describe which value will be selected for Y, depending on the value of S.

Using VHDL we describe how the multiplexer should behave. The VHDL code in Figure 2.3 does never state how the output should be generated at gate level. It only tells us which input value $(A, B, C, D$ or $E)$ to give as output (Y), depending on the value of the selector (S). We also did not specify what value should be returned when the binary value of S exceeds 100_2. In fact we only specified that the value does not matter to us, using the "do not care" value '-'. In VHDL we have to specify the value of an output for all possible values of the inputs, otherwise it will be interpreted as a register. Since we do not want our multiplexer to use any registers, we specify a value for the output (even for cases that might not exist). Since we specify that we do not care what the value will be, the synthesizer will have more room to select this value for optimizing the multiplexer.

2.3 Xilinx ISE Design Suite

To realize the actual designs, we use a software tool called Xilinx ISE Design Suite [ise12]. Using Xilinx we can design both Schematics (gate level) and write VHDL code (RTL). Besides designing the processor we can also use the Xilinx tools to synthesize and implement our designs. It is essential for us to do that, so we can compare the two designs. The software also has tools for testing the designs before the synthesis and final implementation. With VHDL, we can design a test-bench for our processor and its components to verify the correctness of their behaviour by simulations. We will not have to make separate tests for the two designs, because both are supposed to behave the same way.

It is worth noting that Xilinx has a wide range of components like adders, registers, and multiplexers, which we do not have to design ourselves with logic gates. This does not change the fact that we still need to design a lot of components ourselves. In fact, we will later see that besides designing the larger components for our processor, we will also need small variations on existing components.

Chapter 3

Processor Specification

Before we design the processor on different levels of abstraction, we have to specify it. We start by giving a general idea of what the processor should look like using a block diagram. Then we specify how the processor should work by providing its instruction set architecture.

3.1 Block Diagram of the Processor

The embedded processor [Ste15] that we design is a simple 8-bit processor, with a Harvard Architecture. A Harvard Architecture means that the instructions used by the processor are stored in a different memory component from the memory component that is used to store and load data. The processor is depicted in Figure 3.1 and shows that there are two memory components, one called Instruction Memory and the other called Data Memory. As the names suggest, the Instruction Memory is used to store instructions, while the Data Memory is used to store data.

Even though the Control Unit works mostly with 16-bit signals, the processor is still considered to be an 8-bit processor, because the Arithmetic and Logic Unit (ALU), located in the Data Path, performs its operations with 8-bit operands. The operands used by the ALU are stored in the Register File, which in our processor consists of four 8-bit registers (R0 to R3). The register file is a very small, but also very fast temporary storage unit, which is used by the ALU for operands and intermediate results. A much larger but slower storage component is the Data Memory. It is the task of the Data Path to move data to and from the Data Memory.

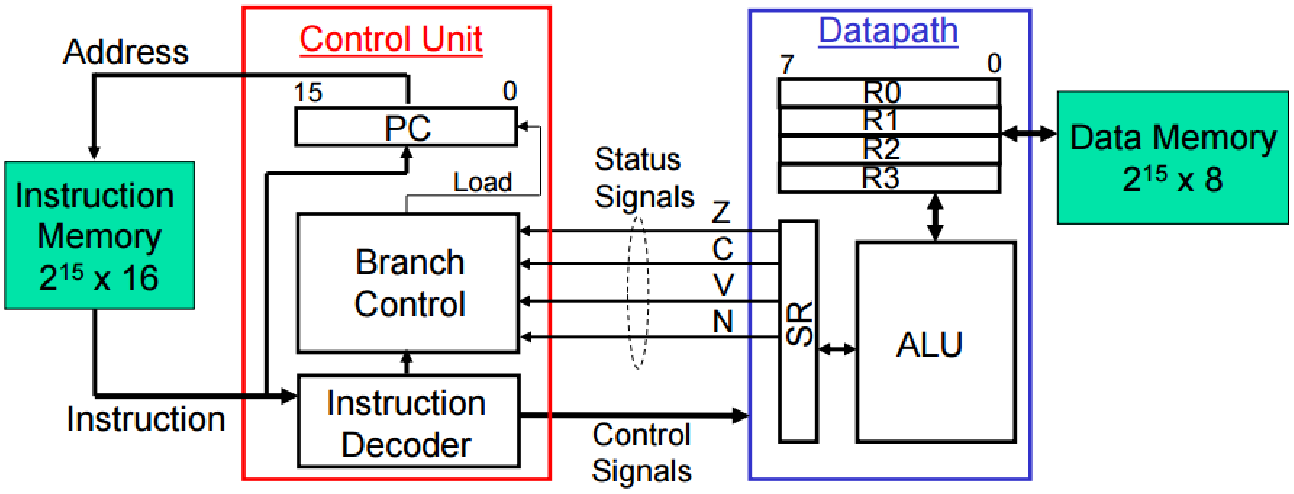


Figure 3.1: Block diagram of the embedded processor.

The Control Unit is responsible for fetching instructions from the Instruction Memory. A component called Program Counter (PC) keeps track of the address of the instruction to be executed. Each clock cycle an instruction is fetched using the address stored in the program counter. Every fetched instruction has to be decoded by the Instruction Decoder. After decoding an instruction, the Branch Control decides whether the Program Counter (PC) should load a new address or increment the address that is already stored. The Instruction Decoder also tells the Data Path what kind of operation it has to perform using several control signals.

3.2 Instruction Set Architecture

An instruction is a set of bits that instructs the processor to perform a certain operation. The set of all instructions used by the processor is called an instruction set. A thorough description of the instruction set for a processor is called instruction set architecture (ISA).

Any instruction set architecture has the following three major components:

- Programming Model
- Instruction Specifications
- Instruction Formats

The meaning of each of these components is explained in the next three subsections.

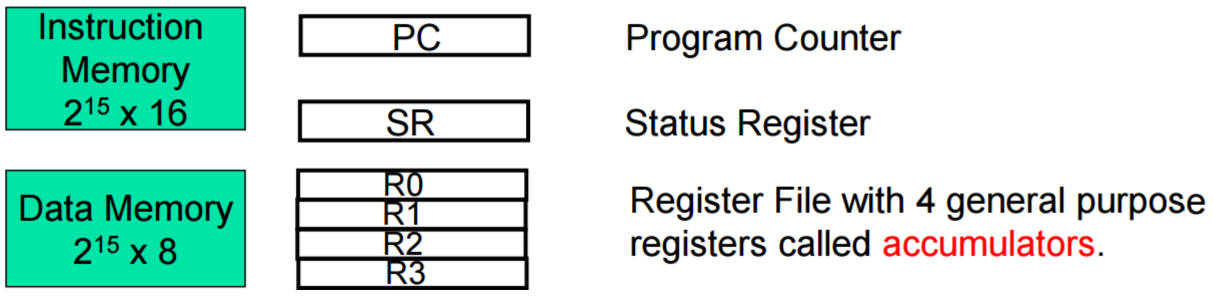

Instruction Memory 2^{15} x 16
Data Memory 2^{15} x 8
PC
SR
R0
R1
R2
R3
Program Counter
Status Register
Register File with 4 general purpose registers called accumulators.

Figure 3.2: The programming model of the processor.

3.2.1 Programming Model

The Programming Model is the processor structure as viewed by a user who programs the processor in a language that directly specifies the instructions to be executed. Such language is called assembly language.

Assembly language is a programming language that uses mnemonic names for its instructions and operands. It is almost on the same level of abstraction as the machine language and has (almost) a one-to-one correspondence with machine instructions. This means that it is very easy to make conversions between the two languages, which will be useful when we want to test our processor.

To be able to create a program for the processor, the programmer has to know about the resources available in the processor. The full Programming Model of our processor is shown in Figure 3.2. The Programming Model of our processor tells the programmer that our processor has two memories, one for instruction storage and one for data storage. The model also shows that there are four 8-bit registers available for operations and temporary data storage. On top of that, we also make sure that the Program Counter and Status Register are included in our Programming Model.

3.2.2 Instruction Specifications

Instruction Specifications describe each of the distinct instructions that can be executed by a processor. Table 3.1 lists the Instruction Specifications for our processor.

The first column of Table 3.1 shows the instruction types, Data Manipulation, Data Movement, and Control Flow. Data Manipulation instructions are instructions that can change the values stored in the Register File. These instructions can be divided into two different operation types (see the second column of Table 3.1), Register-format Arithmetic & Logic Operations and Register-format Shift Operations. The first operation type revolves around all instructions that affect at least one of the four different status bits. The second

Instruction Type	Operation Type	Mnemonic	Operation	Status Bits
Data Manipulation Instructions	**Register-format Arithmetic & Logic Operations**	LDR Rj, Ri	Rj ← Ri	Z, N
		INC Rj, Ri	Rj ← Ri + 1	Z, N
		DEC Rj, Ri	Rj ← Ri - 1	Z, N
		ADD Rj, Ri	Rj ← Rj + Ri	C, V, Z ,N
		ADDC Rj, Ri	Rj ← Rj + Ri + C	C, V, Z, N
		SUB Rj, Ri	Rj ← Rj + Ri′ + 1	C, V, Z, N
		AND Rj, Ri	Rj ← Rj ∧ Ri	Z, N
		OR Rj, Ri	Rj ← Rj ∨ Ri	Z, N
		XOR Rj, Ri	Rj ← Rj ⊕ Ri	Z, N
		NOT Rj, Ri	Rj ← Rj′	Z, N
	Register-format Shift Operations	SHL Rj, Ri	Rj ← Ri << 1	NO effect
		SHR Rj, Ri	Rj ← Rj >> 1	NO effect
Data Movement Instructions	**Memory Write (from registers)**	ST (Rj), Ri	Mem[R0|Rj] ← Ri	NO effect
	Memory Read (to registers)	LD Rj, (Ri)	Rj ← Mem[R0|Ri]	NO effect
	Immediate Transfer Operations	LDI Rj, #const8	Rj ← const8	NO effect
		STI (Rj), #const8	Mem[R0|Rj] ← const8	NO effect
Control Flow Instructions	**Branches**	BZ #offset11	PC ← PC + offset11	NO effect
		BNZ #offset11	PC ← PC + offset11	NO effect
		BC #offset11	PC ← PC + offset11	NO effect
		BNC #offset11	PC ← PC + offset11	NO effect
		BV #offset11	PC ← PC + offset11	NO effect
		BNV #offset11	PC ← PC + offset11	NO effect
		BN #offset11	PC ← PC + offset11	NO effect
		BNN #offset11	PC ← PC + offset11	NO effect
	Jump	JMP Rj, Ri	PC ← Rj|Ri	NO effect

Table 3.1: The Instruction Specifications for the processor.

operation type revolves around the two shift operations that do manipulate the register values, but do not affect any of the status bits. As the operation type names suggest, both sets of instructions will make use of the register format, which will be explained in the next section (Section 3.2.3).

Data Movement instructions are instructions that either load data to the Register File or stores data from it. These instructions can be divided into three different operation types, Memory Write, Memory Read, and Immediate Transfer. Instructions of operation type, Memory Write and Memory Read, store data into and load data from Data Memory, respectively. Instructions of operation type Immediate Transfer, will either load a constant value into the Register File or store it into Data Memory.

Control Flow instructions are instructions that can change the flow of control. These instructions can be divided into two different operation types, Branch and Jump. Branch instructions have to meet certain conditions to change the flow of control, while a Jump instruction changes the flow of control unconditionally.

3.2.3 Instruction Formats

Instruction formats determine the meaning of the bits used to encode each instruction. All formats for our processor will contain 16 bits, because our instruction memory is 16-bit wide. We have a total of 25 distinct instructions, which means that we need at least 5 bits to encode them all. To encode the instructions we use a 5-bit Opcode inside our formats. Some operations require one or two registers. Since our Register File holds 4 registers in total, we only need 2 bits to encode a single register.
There are three types of instruction formats, we use for our processor. We refer to these formats as register format, immediate format and branch format.

The register format, shown in Figure 3.3, is used for all instructions that require two registers. This means that arithmetic, logic, shift, some memory, and jump instructions (see Table 3.1) will use the register format. Because this format only uses a 5-bit Opcode and 4 bits for the two registers, there are 7 bits left that will not be used by these instructions.

15 11	10 9	8 2	1 0
Opcode	Source and/or Destination (Rj)	x x x x x x x	Source (Ri)

Figure 3.3: The register format.

The immediate format is shown in Figure 3.4. The immediate format is used for two instructions, Load Immediate (LDI) and Store Immediate (STI) — see Table 3.1. These instructions both require one register and an 8-bit constant. This means that together with the Opcode we need 15 bits from our format. This leaves one bit (bit 8) unused.

15 11	10 9	8	7 0
Opcode	Destination (Rj)	x	Immediate Operand (const8)

Figure 3.4: The immediate format.

The branch format, shown in Figure 3.5, is used for all branch instructions — see Table 3.1. As shown in Table 3.1, each branch instruction requires an 11-bit offset. This means that together with the 5-bit Opcode, this format uses all 16 bits of the instruction format.

15 11	10 0
Opcode	Immediate Operand (offset11)

Figure 3.5: The branch format.

Chapter 4

Designing the Processor at Different Levels of Abstraction

Before we can use our processor specification, described in Chapter 3, to design the processor, we have to make some design choices that we will apply to both the VHDL (high level of abstraction) design and the Schematic (low level of abstraction) design. These choices can revolve around the top design view of the processor (see Section 4.1) or the different components we use in our processor (see Section 4.2). This chapter will also show the correctness of our two processor designs using a test program (see Section 4.3).

4.1 Top design view of the processor

Figure 4.1 shows the design choices we made for the top level of our processor. We can see how several signals are connected between the different components within the Control Unit and the Data Path, which were not shown yet in Figure 3.1 of Chapter 3.

From the Control Unit, there are several control signals (DA, AA, BA, MB, FS, MD, WR, MW, and SL) and an 8-bit constant value (Constant), that go to the Data Path. The control signals all originate from the Instruction Decoder and have the task to instruct the Data Path what operations have to be performed. The signal, Constant, originates from Instruction Memory and has the task to deliver a constant value to the Data Path, that can either be loaded into the Register File or stored into the Data Memory.
The Control Unit also communicates with the Instruction Memory using the Program Counter (PC). The Program Counter generates an address that is used to fetch the next 16-bit instruction from the Instruction

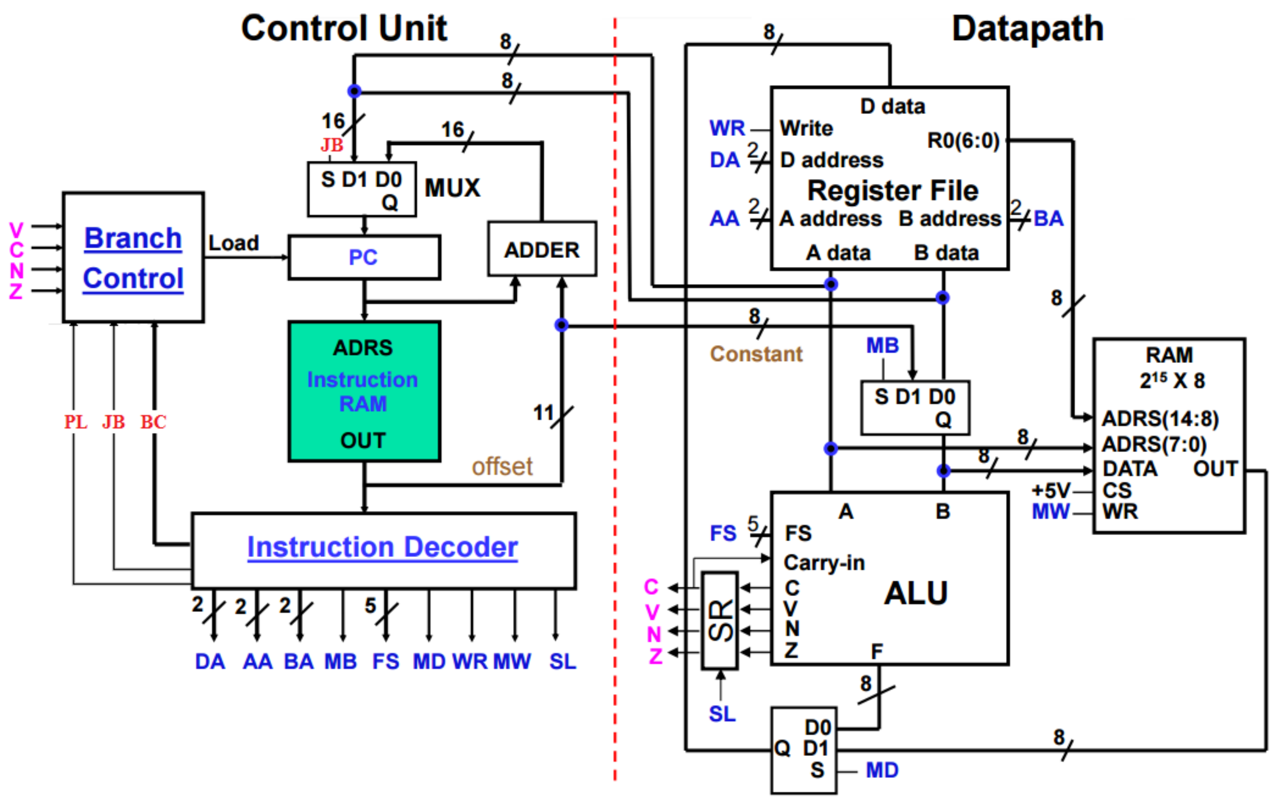


Figure 4.1: A top level view of the design choices we have made for our processor.

Memory. This instruction goes to the Instruction Decoder to determine the operations that have to be performed. Part of the instruction is also used as signal offset (see Figure 4.1), which is used to get signal Constant.

From the Data Path, there are two 8-bit signals and four single bit signals, that go to the Control Unit. The two 8-bit signals originate from the Register File and are concatenated by the Control Unit to make an address that can be used for a jump instruction. The four single bit signals originating from the Status Register (SR) are used by the Control Unit to determine whether a branch condition is met or not.

The Data Path also has several signals going in and out the Data Memory. A 7-bit signal and an 8-bit signal, originating from the Register File, make up the memory address (ADRS(14:8) and ADRS(7:0)), that can be used for both loading and storing data. Another 8-bit signal, originating from a multiplexer, is used to transfer the data that has to be stored. The smallest signal (MW), going from the Data Path to the Data Memory, decides whether new data will be stored or not. Using the two signals that make up the memory address, the Data Memory can transfer the data on that address to the Data Path. In Figure 4.1 this is shown with the 8-bit signal going from the out port, OUT, of the Data Memory to one of the multiplexers of the Data Path.

4.1.1 High-level VHDL design

In Appendix A the VHDL designs of our processor are listed. First, the top level processor is shown. The architecture of the processor (lines 18 to 28) connects the Control Unit with the Data Path using various signals. Figure 4.1 shows the processor connected to two blocks of memory. We see here at lines 9 to 14 the signals that will be connected to these memory components.

In Section A.1 of Appendix A we can see how the Control Unit is designed. Lines 12 to 30 show all input and output signals of the Control Unit, which we had to connect in the processor as shown in Figure 4.1. Lines 45 to 47 show how the PC, Instruction Decoder, and Branch Control are connected with each other. Before that we see some operations from lines 39 to 44. Three things happen on these lines. First, we concatenate two input signals A and B to generate our jump address. The second thing we do, is to generate the immediate constant, which will be used by the Data Path to load a constant value into a register. At last, we generate an offset that will be used for branching. When a branch is taken we want to skip a certain amount of instructions. As we can see in the code (line 43), the offset is added to the old address and used as input for the PC whenever we execute a branch instruction (JB = 0), as shown at line 44.

In Section A.2 of Appendix A we show how we designed the Data Path. Lines 11 to 31 show all input and output signals of the Data Path, which we had to connect in the processor as shown in Figure 4.1. Lines 40 & 41 show how the Register File and ALU are connected to each other. Lines 42 to 66 show the various tasks the Data Path must perform using the Register File and ALU. Lines 42 & 43 generate two halves of an address that the Control Unit will use for jump instructions. Lines 44 to 48 describe how the Data Path stores and loads data to and from the Data Memory. Lines 49 to 66 describe the behaviour of the Status Register (SR).

4.1.2 Low-level Schematic design

In Appendix B, the Schematic designs of our processor are shown. First, we see the top level of the processor. The processor uses two component blocks, the Control Unit and the Data Path. Figure 4.1 shows the processor connected to two blocks of memory. We can see in our Schematic design the input and output signals that will be connected to these memory components.

In Section B.1 of Appendix B we show how we designed the Control Unit as Schematic. We see a lot of input and output signals, which we used in the processor to connect it with the Data Path. We also show how the

PC, Instruction Decoder, and Branch Control are connected with each other, just like we show in Figure 4.1. The input signals A and B are concatenated to generate a jump address. This jump address will be chosen by the multiplexer (mux2_1_16) if the current instruction is not a branch instruction (JB = 1). The branch address is generated using the immediate offset from the fetched branch instruction. This offset is added to the address used to fetch the instruction using an adder (see ADD16). The result is the branch address. The branch address will only be selected by the multiplexer (mux2_1_16) if the current instruction is a branch instruction (JB = 0). Whether the PC will load the jump/branch address is decided by the Branch Control, by using the Load signal.

In Section B.2 of Appendix B we show how we designed the Data Path. We see all the input and output signals of the Data Path, which we had to connect in the processor as shown in Figure 4.1. We also see how the Register File and ALU are connected to each other.
We see two output signals A and B from the Register File. These are the two halves of the jump instruction which will be used by the Instruction Decoder. We also use signals A and B to store data into the Data Memory. Signal A concatenated with R0 make up the address of the data and either signal B or the input signal, Const, will be stored at that address. Signals A and B are also used as operands for the ALU and input signal FS to select the operation.
Connected to the output of the ALU, we see a register (FD4RE), which represents the Status Register (SR). This register contains the four status bits. Our processor will only change the contents of the status register, when an operation affects a status bit. As we specified in Chapter 3 (see Table 3.1) only the arithmetic and logic operations should affect the status bits. All status bits are used by the Control Unit (more precisely the Branch Control), but the Carry bit is also used by the ALU for one specific operation (ADDC).
Besides storing data into the Data Memory, we can also load data from it to the Register File. We use the rightmost multiplexer (mux2_1_8) to decide whether we want to load data from the memory or from the ALU.

4.2 Design of processor components

In the previous section, we have shown the design choices we made at the top-level design view of our processor. In the next subsections, we will show the design choices we made for the different hardware components comprising our processor. Some components were designed using the same choices and optimizations for both VHDL and Schematics, while other components required some more optimizations, when using Schematics.

4.2.1 Register File

The Register File (see Figure 4.1) has a very straightforward purpose. It handles the data that the processor has to load into the registers. The Register File in our processor consists of four 8-bit registers called R0, R1, R2 and R3. Figure 4.1 shows several signals connected to the Register File. The signals D, WR and DA are used for storing data. Signal D transports data to the Register File, signal WR decides whether the data will be stored or not and signal DA decides which one of the four registers will be used for storage. Signals AA, BA, A, and B are used for reading data from the registers. Signals AA and BA select a register, whose data will be transported by signals A and B, respectively. Signal R0 is a special signal with a less straightforward task. Signal R0 uses data from register R0 to make up the most significant part of the memory address, that the processor uses for data storage. The least significant part of the memory address comes from signal A, whose data can come from any of the four registers (even R0).

Section A.2.1 of Appendix A shows our design of the Register File using VHDL. After defining all input and output ports in lines 13 to 21, we start describing the behaviour of the Register File. At lines 26 and 27, we define an array of registers. Lines 29, 30 and 31 describe what value the output signals should have. Since we use an array for our registers, we can just use the input signals AA and BA as indices to describe this behaviour. Lines 32 to 43 describe what values should be stored into the the registers, while using signal DA or variable *i* as index.

Section B.2.1 of Appendix B shows our design of the Register File using Schematics. This method of designing has the disadvantage that we have to add every single register (FD8RE) to our design and connect them to the right components in the correct way. Since our Register File only uses four registers, this will not require that much effort, but in practice Register Files would require more registers, which leads to more work. This extra effort was avoided in VHDL by using an array of registers. Nevertheless, this particular design is not that much more complex than the VHDL design. The decoder (D2_4E) at the left is simply meant to select a register to store the value input signal D. The multiplexers (mux4_1_8) on the right have the purpose of choosing the right register data to transfer to the output signals A and B. And on the top right, we see that the output Ro always gets the value of the top register (Ro). These things are still quite easy to spot in this design, but later on we will see a Schematic design that requires a bit more work to understand.

Instruction	FS	Operation	Status Bits Set
INC	00000	F = B + 1	C = 0, V = 0, N, Z
ADD	00001	F = A + B	C, V, N, Z
ADDC	00010	F = A + B + Cin	C, V, N, Z
SUB	00011	F = A + B′ + 1	C, V, N, Z
DEC	00100	F = B - 1	C = 0, V = 0, N, Z
LDR	00101	F = B	C = 0, V = 0, N, Z
SHR	00110	$F = B >> 1$	
SHL	00111	$F = B << 1$	
AND	01000	$F = A \wedge B$	C = 0, V = 0, N, Z
OR	01001	$F = A \vee B$	C = 0, V = 0, N, Z
XOR	01010	$F = A \oplus B$	C = 0, V = 0, N, Z
NOT	01011	F = B′	C = 0, V = 0, N, Z
LDI	01111	F = B	

Table 4.1: A table of all operations performed by the ALU.

4.2.2 ALU

Calculations are done by the ALU. The ALU in Figure 4.1 works very straightforward. The ALU gets two 8-bit input signals, which represent two numbers and an input signal that represents a carry-bit. These three numbers will be used to perform a certain calculation, which will be selected by the 5-bit input signal called FS. The output signals of the ALU consist of the 8-bit value F and the four status bits C, V, N, Z. These status bits stand for Carry, Overflow, Negative and Zero respectively.

Table 4.1 shows all Data Manipulation instructions and the Load Immediate instruction, we introduced earlier in Chapter 3—see Table 3.1. Table 4.1 shows our design choice to associate the different instructions to certain values of signal FS. Note that in column Status Bits Set we see for some instructions the status bits Carry and Overflow being set to zero. The reason for this is that for some operations the ALU should not detect carry or overflow, but at the same time it should detect and store zero and negative in the status register. Since we chose to load the four status bits simultaneously to the Status Register, we decided to always set the carry and overflow bit to zero for these instructions.

Section A.2.2 of Appendix A shows the VHDL code for our ALU design. Using VHDL to design the ALU required little effort. We can see at lines 27 to 41 how we use a 9-bit signal to temporarily store the outcome of the selected operation. Selecting the operation can be easily done with VHDL by using the input signal FS in an select statement. The lines 43 to 93 describe how the four different status bits are set. We set the status bits in the same way as we do with output signal F.

Section B.2.2 of Appendix B shows how we designed the ALU using Schematics. We can see clearly that this design is by far much more complex. It is not a surprise that it took more effort designing the ALU using a low level of abstraction (Schematics) than using a high level of abstraction (VHDL).

Instruction	FS	Input 1	Input 2	Carry-in
INC	00000	B	1	0
ADD	00001	B	A	0
ADDC	00010	A	B	Cin
SUB	00011	A	B′	1
DEC	00100	B	11111111	0
LDR	00101	B	0	0

Table 4.2: Selecting the adder input.

Besides having to make a design that requires this large amount of signals and components, we also had to make some optimizations beforehand to prevent the use of even more components. Our strategy of optimizing the ALU was to use a single adder component to perform the first six operations in Table 4.1, while trying to use as little as possible multiplexers to choose which input signals the adder has to receive. Realizing that addition is a commutative operation, we were able to reduce the amount of multiplexers needed to a minimum. Table 4.2 shows which variable was selected for each input.

There were some other optimizations made for the ALU that we have not mentioned. These optimizations often revolved around selecting the right values using as few multiplexers as possible. We did not mention them because they are often very simply solved using truth tables.

4.2.3 Program Counter

The Program Counter is responsible for setting the correct instruction from the program memory. Each clock cycle it can either load a new instruction address or increment the previously used address. The new 16-bit address will be supplied by the output signal from the multiplexer (MUX) in Figure 4.1. The signal Load determines whether the Program Counter should load a new address or not. The 16-bit output signal of the Program Counter is the instruction address that will select which instruction will be fetched.

The VHDL code for our Program Counter can be found at Section A.1.1 of Appendix A. Lines 10 to 13 define the input and output ports. Lines 19 to 31 describe how the output signal ADDRS is set depending on the input signals. Note that we use signal 'address' (which will be interpreted as a register) to store the current 16-bit address, so we can use it during the next clock cycle (line 19).

The Schematic design of our Program Counter in Section B.1.1 of Appendix B, is not much different from the

Instruction	Opcode bits	Control Signals					Branch/Jump Signals		
	$I_{15}I_{14}I_{13}I_{12}I_{11}$	MB	MD	WR	MW	SL	PL	JB	$BC_2BC_1BC_0$
INC	00000	0	0	1	0	1	0	x	xxx
ADD	00001	0	0	1	0	1	0	x	xxx
ADDC	00010	0	0	1	0	1	0	x	xxx
SUB	00011	0	0	1	0	1	0	x	xxx
DEC	00100	0	0	1	0	1	0	x	xxx
LDR	00101	0	0	1	0	1	0	x	xxx
SHR	00110	0	0	1	0	0	0	x	xxx
SHL	00111	0	0	1	0	0	0	x	xxx
AND	01000	0	0	1	0	1	0	x	xxx
OR	01001	0	0	1	0	1	0	x	xxx
XOR	01010	0	0	1	0	1	0	x	xxx
NOT	01011	0	0	1	0	1	0	x	xxx
ST	01100	0	x	0	1	0	0	x	xxx
LD	01101	x	1	1	0	0	0	x	xxx
STI	01110	1	x	0	1	0	0	x	xxx
LDI	01111	1	0	1	0	0	0	x	xxx
BNZ	10000	x	x	0	0	0	1	0	000
BNC	10001	x	x	0	0	0	1	0	001
BNV	10010	x	x	0	0	0	1	0	010
BNN	10011	x	x	0	0	0	1	0	011
BZ	10100	x	x	0	0	0	1	0	100
BC	10101	x	x	0	0	0	1	0	101
BV	10110	x	x	0	0	0	1	0	110
BN	10111	x	x	0	0	0	1	0	111
	11xxx	x	x	x	x	x	x	x	xxx
JMP	11111	x	x	0	0	0	1	1	xxx

Table 4.3: This table shows the opcode (5 most significant bits) of the instruction fetched and the decoded values that result from it. The symbol 'x' means that the bit value does not matter.

VHDL design. Our Schematic design uses a register to store either the old address incremented by one or the address called DATA. It is very likely that both our designs of the program counter will be exactly the same after synthesis.

4.2.4 Instruction Decoder

Each time the Program Counter generates a new instruction address, the Instruction Decoder has to take the newly fetched instruction and decode it. The Instruction Decoder will not only generate the control signals for the Data Path, but also jump/branch signals for the Control Unit, which are illustrated in Figure 4.1 as PL, JB and BC. Table 4.3 shows how we decode the Opcode bits into the output signals of the Instruction Decoder. Signals DA, AA, and BA are not mentioned in the table, because they are derived using the instruction formats in Chapter 3. We decided that AA and DA will be our two register source/destination bits from the register format, while BA will be the two register source bits.

Instruction	BC	Branch if
BNZ	000	Z=0
BNC	001	C=0
BNV	010	V=0
BNN	011	N=0
BZ	100	Z=1
BC	101	C=1
BV	110	V=1
BN	111	N=1

Table 4.4: The conditions for a branch operation.

The VHDL design of our Instruction Decoder is shown at Section A.1.2 in Appendix A. Lines 9 to 21 define all input and output ports. Lines 26 to 96 describe how the output values should be generated from the instruction. Using Table 4.3 we were able to determine that the values of some signals could be simply derived by copying certain bits from the instruction. These signals can be seen at lines 26 to 32. The other signals are selected using an if-else statement. We simply stated, for each opcode, the value that the signal should get. Any optimizations were left for the synthesizer to do.

The Schematic design of our Instruction Decoder is shown at Section B.1.2 in Appendix B. Before we could make the design we had to find a way to translate the opcode into each signal. To do that we used truth tables to get the optimal way of doing this. Our expectations were that the optimizations, we performed on gate level for our Instruction Decoder, would be the same as the optimizations made by the synthesizer of our RTL design.

4.2.5 Branch Control

The Branch Control, shown in Figure 4.1, has the task to decide whether the Program Counter should load a new instruction address or increment the current address. In order to make this decision, the Branch Control component uses the four status-bit signals and the three jump/branch signals PL, JB and BC. The output signal Load is used to inform the Program Counter on this decision.

If PL=0, then the instruction was not a jump or a branch instruction, which automatically means that Load=0. If PL=1 and JB=1, then the instruction was a jump, which automatically means that Load=1. If PL=1 and JB=0, then the instruction is a branch, which means that the value of Load depends on the value of BC and the status bits. Table 4.4 shows when a branch will be performed. So given that PL=1 and JB=0 and the branch will be taken, then Load=1.

```
clock_process :process
begin
            clock <= '0';
            wait for clock_period/2;
            clock <= '1';
            wait for clock_period/2;
end process;
```

Figure 4.2: VHDL code that describes the clock behaviour in a testbench.

The VHDL code in Section A.1.3 of Appendix A shows how we designed the Branch Controller at RTL level of abstraction. After defining the input and output ports in lines 5 to 12, we describe the behaviour of the Branch Control in lines 17 to 42. We simply describe the behaviour, explained above, in VHDL by using if-else statements.

The Schematic design of our Branch Controller is not very complex. As we can see in Section B.1.3 of Appendix B we only needed three multiplexers (M2_1), an XNOR gate (XNOR2), an AND gate (AND2), and an OR gate (OR2). We do not expect that the VHDL design will be much different in comparison, after we synthesized both designs.

4.3 Design Testing

After making the two processor designs, we have to test them for correctness. We test our designs using VHDL. With VHDL, we can make a test environment called a testbench. The testbench generates stimuli for the input signals of a hardware component depending on the amount of time that has passed since the beginning of the test. This means that the same input signal can have multiple values during the test. For example, a testbench often has to simulate the behaviour of a clock signal by alternating its value after every specified period of time. Some of the components, we had to design, also use a clock cycle as input, which means that their test files also contained some code that described the alternating behaviour of the clock cycle. Figure 4.2 shows the code snippet that describes the clock behaviour of the Data Path.

After we write the VHDL code for our testbench we can simulate our design and generate the testbench waveform. When we look at the testbench waveform of a hardware component we can see the values of all our input and output signals during the whole simulation. We can then determine if the value of the output signals correspond to the values we expected them to have at that point in time.

Fortunately both our designs were made from the same Processor specifications, which meant that we could test both designs with almost identical VHDL code. Some components did use different names for the same input/output signals, which meant that we had to copy and edit a test in those cases. Knowing that both processors should behave the same way (except for undefined behaviour), we often compared the two testbench results with each other after examining them individually. When the two waveforms matched each other, it would give us a final indicator of the correctness of our designs.

Besides testing the individual components and the processor using generated input signals, we also used a test program [Ste15]. The original test program is written in C++ as shown in Figure 4.3. The test program executes a for-loop to sum all numbers from 1 to 20. This means that the outcome of the test program will be that X = 120. To use this test program, we first translated it into assembly, which is shown in Figure 4.4. At the end of the assembly program at line 13, it is specified that the answer 120 will be stored at memory address 0x3cff. Of course to actually test the two processors we have to translate the assembly code into binary, as shown in Table 4.5. To use the binary test program and be able to store the answer, we had to add Instruction Memory and Data Memory into our designs, outside the design hierarchy. This means that these memory components were neither synthesized nor implemented in the final designs. The binary code was added "manually" into the Instruction Memory.

In their current state, both designs of the processor are capable of producing the same correct results, which is storing value $120_{10} = 11010010_2$ into address $3cff_{16} = 111010111111111_2$. We can see the result of the two simulations in Figure 4.5a and Figure 4.5b. If we look at both figures, we can see in the first column names of different signals. The signal *printinstr* tells us which instruction is currently executed, *printiaddrs* shows the address of the instruction, *printmaddrs* is the address used for data storage, and *printdata* shows us what value is being stored. After 1875ns *printdata* and *printmaddrs* both show the correct result. The time at which the answer is stored can be exactly predicted, but also easily guessed. The assembly code goes through a for-loop exactly 20 times. Multiply it with the number of instructions (eight) used by the for-loop and the clock period (10ns), and we get a minimum of 1600ns before the end of the program is reached. The exact calculation is as follows:

$$time = rise\_time + reset\_time + (program\_size - 1 + for\_loop\_size \times (for\_loop\_rate - 1) +$$

$$for\_loop\_check) \times clock\_period = 5ns + 50ns + (28 - 1 + 8 \times (20 - 1) + 3) \times 10ns = 1875ns$$

.

For which:

- *rise_time* = amount of time before the first rising edge appeared in the simulation,

```
int X = 0;
for(int i = 1; i <= 20; i++){
  X = X + i;
}
```

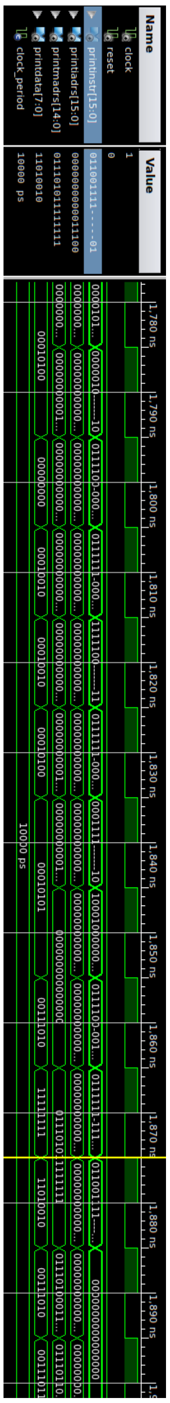
Figure 4.3: The test program written in C++.

```
0x0010: LDI R1, #0x00 // X = 0"
0x0011: LDI R2, #0x01 // R2 stores the loop index i
0x0012: LDI R3, #0x14 // R3 stores the upper loop bound 20
0x0013: SUB R3, R2 // R3 = 20 - i
0x0014: BNC #0x006 // if(i > 20) goto #0x001A = #0x0014 + #0x006
0x0015: ADD R1, R2 // X = X + i
0x0016: INC R2 // i++
0x0017: LDI R0, #0x00 // R0 = 0x0000
0x0018: LDI R3, #0x12 // R3 = 0x0012
0x0019: JMP R0, R3 // Go back to loop test (address 0x0012 = R0|R3)
0x001A: LDI R0, #0x3c // R0 = 0x003c
0x001B: LDI R3, #0xff // R3 = 0x00ff
0x001C: ST  (R3), R1 // Store X in data memory at address 0x3cff
```

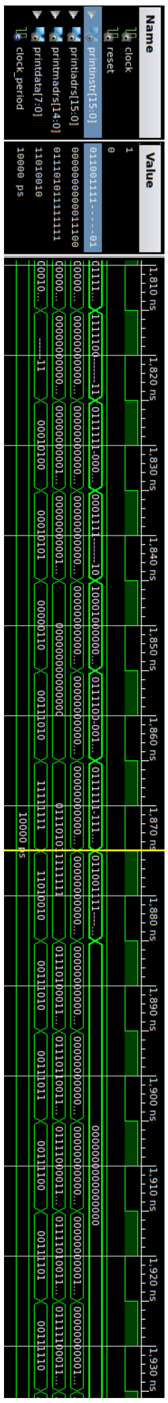
Figure 4.4: The test program translated from C++ to assembly.

- *reset_time* = amount of time the reset signal was activated,
- *program_size* = highest instruction address = $1C_{16}$ = 28_{10} (not the amount of lines, since the program starts at 0x10, while the processor starts at 0x00),
- *for_loop_size* = amount of instructions needed for the for-loop used for each loop of the for-loop,
- *for_loop_rate* = amount of loops taken,
- *for_loop_check* = amount of instructions needed for the check of the for-loop,
- *clock_period* = amount of time between two rising edges.

Besides using a calculation to predict the moment the program ends, we can also examine the values of signals *printinstr* and *printmaddrs*. These signals should have 8 different values that repeat every period of 80ns, 20 times. These values should also correspond to the values in Figure 4.5.



(a) The testbench waveform of the Schematics design of the processor.



(b) The testbench waveform of the VHDL design of the processor.

Figure 4.5: The testbench waveforms of the two processor designs.

Address	Instruction
0x0010	0111101X00000000
0x0011	0111110X00000001
0x0012	0111111X00010100
0x0013	0001111XXXXXXX10
0x0014	1000100000000110
0x0015	0000101XXXXXXX10
0x0016	0000010XXXXXXX10
0x0017	0111100X00000000
0x0018	0111111X00010010
0x0019	1111100XXXXXXX11
0x001A	0111100X00111010
0x001B	0111111X11111111
0x001C	011001111XXXXX01

Table 4.5: The test program translated from assembly to binary code. Symbol ‘X’ means that the bit value does not matter.

Chapter 5

Processor Design Evaluation

This chapter evaluates the two processor designs by comparing them with each other. Section 5.1 explains how the implementation and its evaluation reports were generated. Section 5.2 takes the generated reports and compares them with one another. The two processor designs will be evaluated on the amount of hardware resources used, the maximum clock speed, and the amount of effort needed to design them. At the end we will conclude which abstraction level of designing an embedded processor gave the highest quality processor with the least amount of effort.

5.1 Implementation and Experimental Setup

After writing the VHDL code and drawing Schematics of the embedded processor, and thorough testing, we synthesized and implemented both designs. Before we implemented the designs, we made sure that the "Design Goals and Strategies" were set the same way for both designs, because we do not want the software tools to optimize one design in a different way from the other. For both designs, the Design Goal was set to 'Balanced' and the Strategy to 'Xilinx default'. Synthesizing and implementing the processor designs resulted in several files being generated by the Xilinx ISE tools, which gave a lot of information on the quality of our two designs. Both of our designs can be found in the tarball "Report.tar.gz" delivered with the thesis. This includes test files and generated reports.

The implementation process was always preceded by the synthesize process. The implementation process consists of translation, mapping, and place&route. These sub-processes together with the synthesize process generated the files "processor.syr", "processor.bld", "processor.mrp", "processor.par", "processor.twx". "Processor.syr" is the synthesize report, which reports about optimizations made and makes a number of timing estimates. “Processor.bld” is the translation report, which actually does not give us any useful information on the designed processor’s properties. “Processor.mrp” is the mapping report, which reports about the hardware utilization of different components. “Processor.par” is the place&route report. “Processor.twx” is the static timing report, which gives more detailed and precise information about timing constraints.

5.2 Experimental Results

By comparing reports between the two designs, we can draw conclusions. The static timing reports can tell us more about the performance and clock speed of the two designs.The report contains multiple tables. Each table contains different information about the time needed for a signal to travel from one point to the other. The average of all tables show a less amount of time for the Schematic processor design. Nevertheless, according to the report the VHDL design has a higher frequency. The clock frequency of the Schematic design has to be 168 MHz in order to work properly, while the VHDL can work at a frequency of nearly 221 MHz. In the case of performance speed, it seems that the VHDL design performed better than the Schematic design. The same frequencies (timing constraints, which can be converted to frequencies) are found at the end of the place&route report.

When we compare the mapping report, we can tell which design used less hardware resources. Table 5.1 shows how much resources were used for both designs. The first column shows all components used in both designs. The second row shows the three types of hardware resources used in the designs. A Look Up Table (LUT) is a programmable logic gate. The input signals select values that are stored inside the LUT, to use them as output. The values that are stored in the LUT can be altered, which makes it programmable. A slice is a collection of a set number of LUTs, flip-flops and multiplexers.

Note that each cell of the table contains two numbers divide by the symbol ‘/’. The number on the right side tells us how many resources were used to realize the design, while the number on the left side tells us how many of these resources were not used by a component lower in the design hierarchy. The VHDL Data Path uses 49 Slices, but components lower in the design hierarchy (Register File and ALU) use 26 of those Slices. This means that 23 of those 49 Slices were added into the whole design by the Data Path itself. To compare the VHDL with the Schematics designs we only need to look at the numbers on the right side.

Note that the Schematic design had more components lower down the hierarchy, which are not shown in the

	Resources Used					
	Slices		Slice Registers		LUTs	
Design / Components	VHDL	Schematics	VHDL	Schematics	VHDL	Schematics
Processor	0/61	0/54	0/52	0/52	0/160	0/138
Data Path	23/49	0/26	4/36	0/36	29/123	0/82
Register File	5/5	0/4	32/32	0/32	20/20	0/18
ALU	21/21	1/22	0/0	0/0	74/74	3/48
Control Unit	4/12	0/28	0/16	0/16	16/37	0/56
PC	5/5	0/18	16/16	0/16	17/17	0/32
Instruction Decoder	2/2	5/5	0/0	0/0	2/2	6/6
Branch Control	1/1	1/1	0/0	0/0	2/2	1/2

Table 5.1: The utilization of hardware resources for each component.

table. Because we did not design these components in VHDL, we have nothing to compare with these smaller components. Even if we did, we would have to increase the size of the table quite substantially. Leaving out these components is the reason why the numbers on the left of the Schematic design are generally closer to zero compared to the VHDL components. For comparison, we do not need to look at the numbers on the left, but if we wanted to optimize our designs, we could use them to pinpoint where in the hierarchy optimization is needed.

When we examine Table 5.1, there are a few things we can notice quite quickly. The amount of Slice Registers used for both our VHDL and Schematic designs are exactly the same. This is not very surprising because we made very clear which components would be using registers and how many bits those register needed to store. If any of these components would have used more slice registers, then that would have indicated a mistake in our design.
Something that surprised us was that the number of Slices and LUTs used in the Schematic designs are generally lower than in the VHDL designs. Overall the Schematic design uses 54 Slices and 138 LUTs while the VHDL processor design uses 61 Slices and 160 LUTs.

As mentioned earlier, the whole processor requires less hardware resources in our Schematic design than in our VHDL design. However, this is not the case when we look at the Control Unit and Data Path. The Data Path itself (not its subcomponents) also seems to be responsible for the major difference in Slice usage. While the Data Path seems to tell us the same thing, the Control Unit required less resources when designed with VHDL. The subcomponents (PC, Instruction Decoder and Branch Control) of the Control Unit seem to be responsible for this, because when we remove their resource usage from the Control Units resource usage, we get the same amount of resources for both designs.

Something we did not expect was the Register File to use different a amount of resources for the different designs. We especially would not have guessed that the Schematic design is more optimized, because we did not actually try to optimize it in any particular way.

We were very happy to see that the work we have put into the ALU of the Schematic design paid off. We were not sure that the optimizations we had made, would compete with the software-made optimizations. As Table 5.1 shows, our extra manual optimizations resulted in a more resource optimal ALU. The amount of LUTs used by the ALU is probably the main reason for the overall resource optimization results. But the number of slices used in the Schematic design was slightly higher.

As we mentioned earlier, the Control Unit uses more resources when designed using Schematics. It seems this is mostly caused by the PC. Both the amount of Slices and LUTs used in the Schematic design are substantially higher than in the VHDL design. Looking at the original report, we can tell that the high amount of Slices is mostly caused by the register (see Section B.1.1 of Appendix B) used in our design. The Adder we used, in our design, required the same amount of LUTs as our Multiplexer, but also required 4 Slices. This makes the Multiplexer, which is the only component we had to design ourselves, the least resource dependable. It is quite surprising that two standard Xilinx components, which seem to be impossible to leave out of our design, cause so much resource utilization. What is more surprising is how the VHDL design managed to somehow use less resources, because it does not seem possible that we could have optimized our PC any better in the Schematic design.

The Instruction Decoder also requires more resources in our Schematic designs. The difference is far less than with the PC. We expected that we were able to optimize this component ourselves to the fullest, but apparently there is room for improvement.

The Branch Control seems to be completely optimized in both VHDL and Schematic designs. We base this observation on the fact that both designs required the same amount of resources and the fact that both designs use very little amount of resources. From all components shown in Table 5.1 this was definitely the least resource polluting component.

Finally we can conclude that the VHDL design resulted in a processor with a higher clock speed and for some components with less used hardware resources. Even though the Schematic design of our processor is slower, overall it required less hardware resources. We were even able to manually optimize the ALU, which is the largest component (not counting the top level components) in our processor. However, to optimize the

ALU, we had to put a lot more effort into our Schematic designs. The amount of effort needed to design the ALU and many other components in Schematics was significantly larger. A lot of smaller components had to be designed for some components. We also had to put more effort into optimizing other components. Most of the effort was needed to connect the many different signals in the ALU. The extra effort only made the Schematic design less resource dependent, which surprisingly did not affect the clock speed. Even though after all that extra effort, we put into our Schematic designs, the clock speed was still lower compared to the VHDL design.

Chapter 6

Conclusions and Future Recommendations

The focus of our research was to compare two levels of abstraction for designing a relatively simple embedded processor. We have made a design starting at a low level of abstraction using Schematics and at a high level of abstraction using VHDL. We compared the two design approaches on their ability to produce high quality designs of an embedded processors.

In our evaluation, we concluded that the VHDL (high level of abstraction) design required less effort to make. We also found that more resources were used for the VHDL design. The VHDL design required 61 Slices and 160 LUTs, whereas the Schematic design required 54 Slices and 138 LUTs. The amount of Slice Registers used by both designs was 52. The VHDL design may have used more resources, but the clock signal of the VHDL design had a maximum frequency of nearly 221 MHz, whereas the clock signal of the Schematic design had a maximum frequency of 168 MHz.

Since we had a limited amount of time doing the research, we were only able to design the processor using two design methods at different levels of abstraction. This means that we have produced very limited amount of results. More reliable results could have been produced by designing the processor using more low and high level of abstraction designs.

To obtain our results we used only one processor. Designing more different processors could mean more different results. It could turn out that certain types of processors should be designed using high abstraction level approaches, while others might be of higher quality when designed at a lower levels of abstraction. Given more time we could have tried designing a range of processors, each designed multiple times using different design approaches of both high and low level. Doing this would have definitely given more reliable results.

Besides increasing the sample space of our research, it would also be useful to have different people designing processors. An individual person could have a biased amount of skills towards one design approach over another. This could mean that this individual's lack of experience in using VHDL, could have influenced the resulting processor design in its performance and quality. Having different people do the research would most likely decrease this bias.

Other research done before, already indicated that high abstraction levels are more suitable for designing large hardware components like embedded processors. Designers and manufacturers prefer to design their hardware with the least amount of effort and the most amount of quality. Higher usage of hardware resources can be compensated by higher quality. In this case we measure the quality of the two processor designs through their clock speed. Since the higher level of abstraction (VHDL) design required less effort and a higher clock speed, it is safe to conclude that high abstraction levels are more suitable for the design of embedded processors than low abstraction levels.

Appendices

Appendix A

Processor Designed using VHDL

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity processor is
    generic (
        BYTE: integer := 8; -- amount of bits stored in a register
        RAL: integer := 2 -- amount of bits needed to describe register address
    );
    Port ( Instruction : in  STD_LOGIC_VECTOR (15 downto 0);
           Load : in  STD_LOGIC_VECTOR (7 downto 0);
           IADRS : out  STD_LOGIC_VECTOR (15 downto 0);
           MADRS : out  STD_LOGIC_VECTOR (14 downto 0);
           Store : out  STD_LOGIC_VECTOR (7 downto 0);
           RAMWR : out  STD_LOGIC;
           clock,reset : in  STD_LOGIC);
end processor;

architecture Behavioral of processor is
signal WR,MB,MW,MD,SL,C,V,N,Z : STD_LOGIC;
signal A,B,Const : STD_LOGIC_VECTOR (BYTE - 1 downto 0);
signal FS : STD_LOGIC_VECTOR (5 - 1 downto 0);
signal DA,AA,BA : STD_LOGIC_VECTOR (RAL - 1 downto 0);
begin
```

```
    DP : entity work.Datapath port map(Const,WR,DA,AA,BA,MB,FS,MW,MD,SL,Load,A,B,C
        ,V,N,Z,MADRS,Store,RAMWR,clock,reset);
    CU : entity work.ControlUnit port map(Instruction,C,V,N,Z,A,B,IADRS,Const,DA,
        AA,BA,MB,FS,MD,WR,MW,SL,clock,reset);
end Behavioral;
```

A.1 Control Unit

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity ControlUnit is
    generic (
        BYTE: integer := 8; -- amount of bits stored in a register
        TWOBYTE: integer := 16; -- amount of bits stored in a register
        RAL: integer := 2 -- amount of bits needed to describe register address
    );
    Port ( INSTR : in STD_LOGIC_VECTOR (TWOBYTE - 1 downto 0);
           C : in STD_LOGIC;
           V : in STD_LOGIC;
           N : in STD_LOGIC;
           Z : in STD_LOGIC;
           A : in STD_LOGIC_VECTOR (BYTE - 1 downto 0);
           B : in STD_LOGIC_VECTOR (BYTE - 1 downto 0);
           ADRS : out STD_LOGIC_VECTOR (TWOBYTE - 1 downto 0);
           Const : out STD_LOGIC_VECTOR (BYTE - 1 downto 0);
           DA : out STD_LOGIC_VECTOR (1 downto 0);
           AA : out STD_LOGIC_VECTOR (1 downto 0);
           BA : out STD_LOGIC_VECTOR (1 downto 0);
           MB : out STD_LOGIC;
           FS : out STD_LOGIC_VECTOR (4 downto 0);
           MD : out STD_LOGIC;
           WR : out STD_LOGIC;
```

```
            MW : out  STD_LOGIC;
            SL : out  STD_LOGIC;
            clock, reset : in  STD_LOGIC);
end ControlUnit;

architecture Behavioral of ControlUnit is
signal PL, JB, Load : STD_LOGIC;
signal BC : STD_LOGIC_VECTOR (3 - 1 downto 0);
signal Data, AB, ADD, address, Q : STD_LOGIC_VECTOR (TWOBYTE - 1 downto 0);
signal offset : STD_LOGIC_VECTOR (10 downto 0);
begin
  AB <= A & B;
  ADRS <= address;
  offset <= INSTR(10 downto 0);
  Const <= offset(BYTE - 1 downto 0);
  ADD <= address + ("00000" & offset);
  DATA <= AB when JB = '1' else ADD;
  PC : entity work.PC port map(Load,Data,address,clock,reset);
  ID : entity work.InstructionDecoder port map(INSTR,DA,AA,BA,MB,FS,MD,WR,MW,SL,
      PL,JB,BC);
  BrC : entity work.BranchControl port map(PL,JB,BC,C,V,N,Z,Load);
end Behavioral;
```

A.1.1 PC

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity PC is
    generic (
        TWOBYTE: integer := 16
    );
    Port ( Load : in  STD_LOGIC;
           Data : in  STD_LOGIC_VECTOR (TWOBYTE - 1 downto 0);
```

```
            ADRS : out  STD_LOGIC_VECTOR (TWOBYTE - 1 downto 0);
            clock , reset : in  STD_LOGIC);
end PC;

architecture Behavioral of PC is
signal address : std_logic_vector (TWOBYTE - 1 downto 0);
begin
  ADRS <= address;
  Process(clock ,reset)
  begin
    if reset = '1' then
      address <= (others => '0');
    elsif clock 'event and clock = '1' then
      if Load = '0' then
        address <= address + 1;
      else
        address <= Data;
      end if;
    end if;
  end process;
end Behavioral;
```

A.1.2 Instruction Decoder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity InstructionDecoder is
    generic (
        TWOBYTE: integer := 16; -- amount of bits stored in a register
        RAL: integer := 2 -- amount of bits needed to describe register address
    );
    Port ( INSTR : in  STD_LOGIC_VECTOR (TWOBYTE - 1 downto 0);
           DA : out  STD_LOGIC_VECTOR (RAL - 1 downto 0);
           AA : out  STD_LOGIC_VECTOR (RAL - 1 downto 0);
           BA : out  STD_LOGIC_VECTOR (RAL - 1 downto 0);
```

```
              MB : out  STD_LOGIC;
              FS : out  STD_LOGIC_VECTOR (5 - 1 downto 0);
              MD : out  STD_LOGIC;
              WR : out  STD_LOGIC;
              MW : out  STD_LOGIC;
              SL : out  STD_LOGIC;
              PL : out STD_LOGIC;
              JB : out STD_LOGIC;
              BC : out STD_LOGIC_VECTOR (3 - 1 downto 0));
end InstructionDecoder;

architecture Behavioral of InstructionDecoder is
begin
  FS <= INSTR(TWOBYTE - 1 downto TWOBYTE - 5);
  DA <= INSTR(TWOBYTE - 6 downto TWOBYTE - 7);
  AA <= INSTR(TWOBYTE - 6 downto TWOBYTE - 7);
  BA <= INSTR(1 downto 0);
  PL <= INSTR(TWOBYTE - 1);
  JB <= INSTR(TWOBYTE - 2);
  BC <= INSTR(TWOBYTE - 3 downto TWOBYTE - 5);
  process(INSTR)
  begin
  if INSTR(TWOBYTE - 1 downto TWOBYTE - 5) <= "00101" then
    MB <= '0';
    MD <= '0';
    WR <= '1';
    MW <= '0';
    SL <= '1';
  elsif INSTR(TWOBYTE - 1 downto TWOBYTE - 5) <= "00111" then
    MB <= '0';
    MD <= '0';
    WR <= '1';
    MW <= '0';
    SL <= '0';
  elsif INSTR(TWOBYTE - 1 downto TWOBYTE - 5) <= "01011" then
```

```
    MB <= '0';
    MD <= '0';
    WR <= '1';
    MW <= '0';
    SL <= '1';
  elsif INSTR(TWOBYTE - 1 downto TWOBYTE - 5) = "01100" then
    MB <= '0';
    MD <= '-';
    WR <= '0';
    MW <= '1';
    SL <= '0';
  elsif INSTR(TWOBYTE - 1 downto TWOBYTE - 5) = "01101" then
    MB <= '-';
    MD <= '1';
    WR <= '1';
    MW <= '0';
    SL <= '0';
  elsif INSTR(TWOBYTE - 1 downto TWOBYTE - 5) = "01110" then
    MB <= '1';
    MD <= '-';
    WR <= '0';
    MW <= '1';
    SL <= '0';
  elsif INSTR(TWOBYTE - 1 downto TWOBYTE - 5) = "01111" then
    MB <= '1';
    MD <= '0';
    WR <= '1';
    MW <= '0';
    SL <= '0';
  elsif INSTR(TWOBYTE - 1 downto TWOBYTE - 5) <= "10111" then
    MB <= '-';
    MD <= '-';
    WR <= '0';
    MW <= '0';
    SL <= '0';
  elsif INSTR(TWOBYTE - 1 downto TWOBYTE - 5) = "11111" then
    MB <= '-';
    MD <= '-';
    WR <= '0';
    MW <= '0';
    SL <= '0';
  else
    MB <= '-';
    MD <= '-';
    WR <= '-';
    MW <= '-';
    SL <= '-';
  end if;
  end process;
end Behavioral;
```

A.1.3 Branch Control

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity BranchControl is
    Port ( PL : in  STD_LOGIC;
           JB : in  STD_LOGIC;
           BC : in  STD_LOGIC_VECTOR (2 downto 0);
           C : in  STD_LOGIC;
           V : in  STD_LOGIC;
           N : in  STD_LOGIC;
           Z : in  STD_LOGIC;
           Load : out  STD_LOGIC);
end BranchControl;

architecture Behavioral of BranchControl is
begin
  process(PL,JB,BC,V,C,N,Z)
  begin
```

```
      if PL = '0' then
        Load <= '0';
      elsif JB = '1' then
        Load <= '1';
      elsif BC = "000" and Z = '0' then
        Load <= '1';
      elsif BC = "001" and C = '0' then
        Load <= '1';
      elsif BC = "010" and V = '0' then
        Load <= '1';
      elsif BC = "011" and N = '0' then
        Load <= '1';
      elsif BC = "100" and Z = '1' then
        Load <= '1';
      elsif BC = "101" and C = '1' then
        Load <= '1';
      elsif BC = "110" and V = '1' then
        Load <= '1';
      elsif BC = "111" and N = '1' then
        Load <= '1';
      else
        Load <= '0';
      end if;
   end process;
end Behavioral;
```

A.2 Data Path

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity Datapath is
      generic (
        BYTE: integer := 8; -- amount of bits stored in a register
        RAL: integer := 2 -- amount of bits needed to describe register address
    );
    Port ( Const : in  STD_LOGIC_VECTOR (BYTE - 1 downto 0);
           WR : in  STD_LOGIC;
           DA : in  STD_LOGIC_VECTOR (RAL - 1 downto 0);
           AA : in  STD_LOGIC_VECTOR (RAL - 1 downto 0);
           BA : in  STD_LOGIC_VECTOR (RAL - 1 downto 0);
           MB : in  STD_LOGIC;
           FS : in  STD_LOGIC_VECTOR (5 - 1 downto 0);
           MW : in  STD_LOGIC;
           MD : in  STD_LOGIC;
           SL : in  STD_LOGIC;
           RAMOUT : in STD_LOGIC_VECTOR (BYTE - 1 downto 0);
           A : out STD_LOGIC_VECTOR (BYTE - 1 downto 0);
           B : out STD_LOGIC_VECTOR (BYTE - 1 downto 0);
           C : out  STD_LOGIC;
           V : out  STD_LOGIC;
           N : out  STD_LOGIC;
           Z : out  STD_LOGIC;
           RAMADRS : OUT STD_LOGIC_VECTOR (14 downto 0);
           RAMDATA : OUT STD_LOGIC_VECTOR (BYTE - 1 downto 0);
           RAMWR : OUT STD_LOGIC;
           clock, reset : in  STD_LOGIC);
end Datapath;

architecture Behavioral of Datapath is
signal carry : STD_LOGIC;
signal sA, sB, F, QB, QD : STD_LOGIC_VECTOR (BYTE - 1 downto 0);
signal R0 : STD_LOGIC_VECTOR (6 downto 0);
signal SR : STD_LOGIC_VECTOR (3 downto 0);
begin
  Data : entity work.Regfile port map(QD,sA,sB,DA,AA,BA,WR,R0,clock,reset);
  Arithm : entity work.ALU port map(sA,QB,FS,carry,SR(0),SR(1),SR(2),SR(3),F);
  A <= sA;
  B <= sB;
  RAMWR <= MW;
  RAMADRS <=  R0 & sA;
  QB <= sB when MB = '0' else Const;
  QD <= F when MD = '0' else RAMOUT;
  RAMDATA <= QB;
  process(clock, reset)
  begin
    if reset = '1' then
      carry <= '0';
      C <= '0';
      V <= '0';
      N <= '0';
      Z <= '0';
    elsif clock'event and clock = '1' then
      if SL = '1' then
        carry <= SR(0);
        C <= SR(0);
        V <= SR(1);
        N <= SR(2);
        Z <= SR(3);
      end if;
    end if;
  end process;
end Behavioral;
```

A.2.1 Register File

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.std_logic_unsigned.all;

entity Regfile is
    generic (
        BYTE: integer := 8; -- amount of bits stored in a register
        RAL: integer := 2; -- amount of bits needed to describe register address
             (Register Address Length)
        RCOUNT: integer := 4 -- amount of registers used (2^RAL)
    );
    Port (
            D : in   STD_LOGIC_VECTOR (BYTE - 1 downto 0);
            A : out  STD_LOGIC_VECTOR (BYTE - 1 downto 0);
            B : out  STD_LOGIC_VECTOR (BYTE - 1 downto 0);
            DA : in  STD_LOGIC_VECTOR (RAL - 1 downto 0);
            AA : in  STD_LOGIC_VECTOR (RAL - 1 downto 0);
            BA : in  STD_LOGIC_VECTOR (RAL - 1 downto 0);
            WR : in  STD_LOGIC;
            R0 : out STD_LOGIC_VECTOR (6 downto 0);
            clock, reset : in  STD_LOGIC);
end Regfile;

architecture Behavioral of Regfile is
signal s1,s2,s3,s4,s5 : STD_LOGIC_VECTOR (7 downto 0);
type Regarray is array(0 to RCOUNT - 1) of STD_LOGIC_VECTOR (BYTE - 1 downto 0);
signal R : Regarray;
begin
  R0 <= R(0)(6 downto 0);
  A <= R(to_integer(unsigned(AA)));
  B <= R(to_integer(unsigned(BA)));
  process(clock, reset)
  begin
    if reset = '1' then
      for i in 0 to RCOUNT - 1 loop
        R(i) <= (others => '0');
      end loop;
    elsif clock'event and clock = '1' then
      if WR = '1' then
        R(to_integer(unsigned(DA))) <= D;
      end if;
    end if;
  end process;
end Behavioral;
```

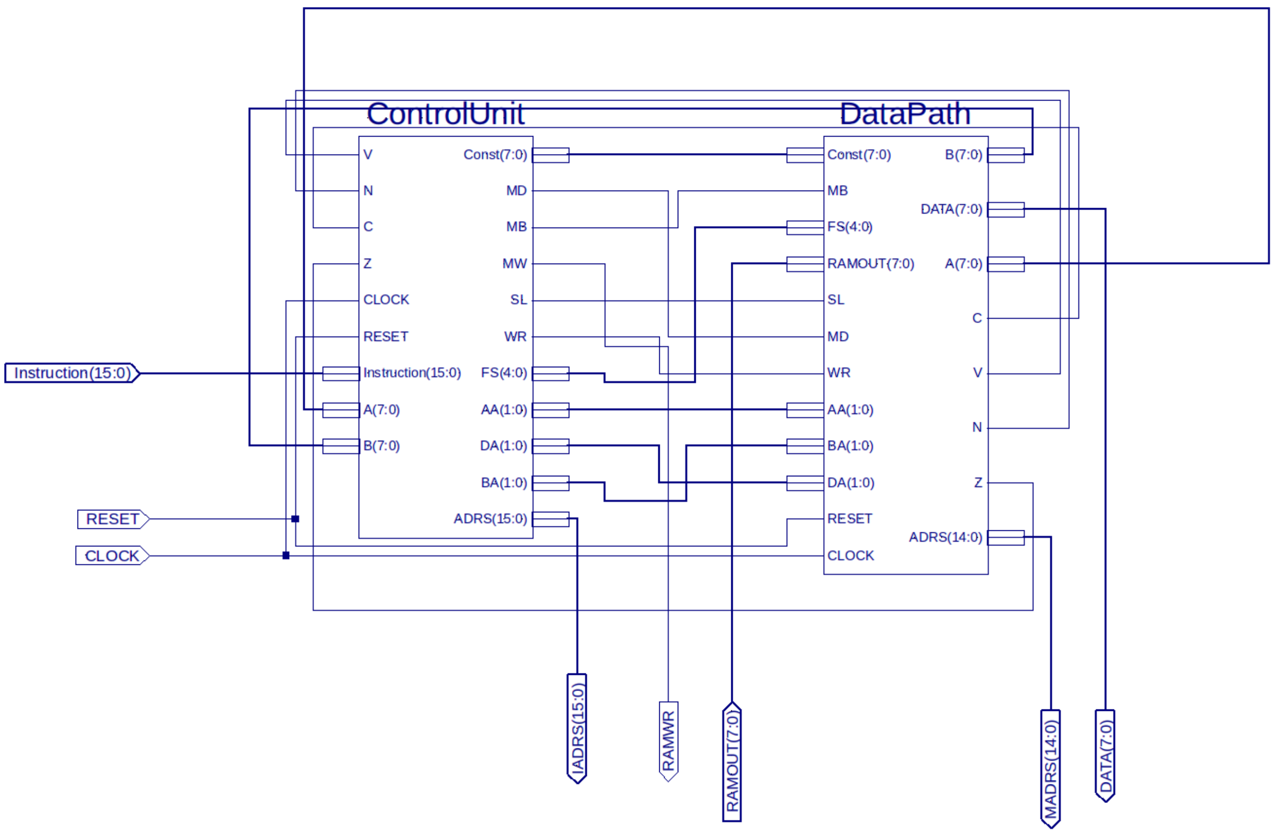
A.2.2 ALU

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity ALU is
    generic (
        BYTE: integer := 8 -- amount of bits stored in a register
    );
    Port ( A : in  STD_LOGIC_VECTOR (BYTE - 1 downto 0);
           B : in  STD_LOGIC_VECTOR (BYTE - 1 downto 0);
           FS : in  STD_LOGIC_VECTOR (5 - 1 downto 0);
           Cin : in  STD_LOGIC;
           C : out  STD_LOGIC;
           V : out  STD_LOGIC;
           N : out  STD_LOGIC;
           Z : out  STD_LOGIC;
           F : out  STD_LOGIC_VECTOR (BYTE - 1 downto 0));
end ALU;

architecture Behavioral of ALU is
signal temp : STD_LOGIC;
signal s : STD_LOGIC_VECTOR (BYTE downto 0);
signal Zero : STD_LOGIC_VECTOR (BYTE - 1 downto 0);
begin
    Zero <= (others => '0');
    with FS select s <=
      ("0" & B) + 1 when "00000",
      ('0' & A) + ('0' & B) when "00001",
      ('0' & A) + ('0' & B) + (Zero & Cin) when "00010",
      ('1' & A) - ('0' & B) when "00011", --the '1' in front of A helps
```

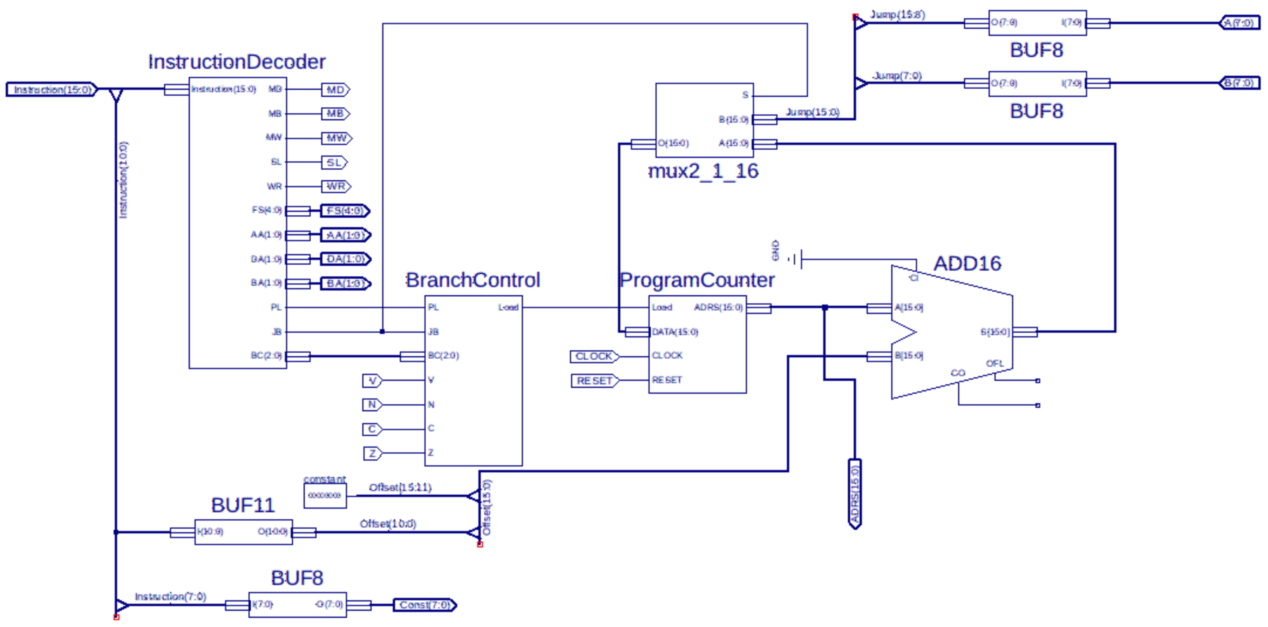
```
            detecting carry
        ('0' & B) - 1 when "00100",
        ('0' & B) when "00101",
        "00" & B(BYTE - 1 downto 1) when "00110",
        '0' & B(BYTE - 2 downto 0) & '0' when "00111",
        ('0' & A) and ('0' & B) when "01000",
        ('0' & A) or ('0' & B) when "01001",
        ('0' & A) xor ('0' & B) when "01010",
        not ('0' & B) when "01011",
        ('0' & B) when "01111",
        (others => '-') when others;
    F <= s(BYTE - 1 downto 0);
    with FS select C <=
        '0' when "00000",
        s(BYTE) when "00001",
        s(BYTE) when "00010",
        s(BYTE) when "00011",
        '0' when "00100",
        '0' when "00101",
        '0' when "01000",
        '0' when "01001",
        '0' when "01010",
        '0' when "01011",
        '-' when others;
    with FS select V <=
        '0' when "00000",
        (A(BYTE - 1) xnor B(BYTE - 1)) and (s(BYTE-1) xor A(BYTE - 1)) when "00001
            ",
        (A(BYTE - 1) xnor B(BYTE - 1)) and (s(BYTE-1) xor A(BYTE - 1)) when "00010
            ",
        (A(BYTE - 1) xor B(BYTE - 1)) and (s(BYTE-1) xor A(BYTE - 1)) when "00011"
            ,
        '0' when "00100",
        '0' when "00101",
        '0' when "01000",
        '0' when "01001",
        '0' when "01010",
        '0' when "01011",
        '-' when others;
    with FS select N <=
        s(BYTE-1) when "00000",
        s(BYTE-1) when "00001",
        s(BYTE-1) when "00010",
        s(BYTE-1) when "00011",
        s(BYTE-1) when "00100",
        s(BYTE-1) when "00101",
        s(BYTE-1) when "01000",
        s(BYTE-1) when "01001",
        s(BYTE-1) when "01010",
        s(BYTE-1) when "01011",
        '-' when others;
    temp <= '1' when Zero = s(BYTE -1 downto 0)
                else '0';
    with FS select Z <=
        temp when "00000",
        temp when "00001",
        temp when "00010",
        temp when "00011",
        temp when "00100",
        temp when "00101",
        temp when "01000",
        temp when "01001",
        temp when "01010",
        temp when "01011",
        '-' when others;
end Behavioral;
```

Appendix B

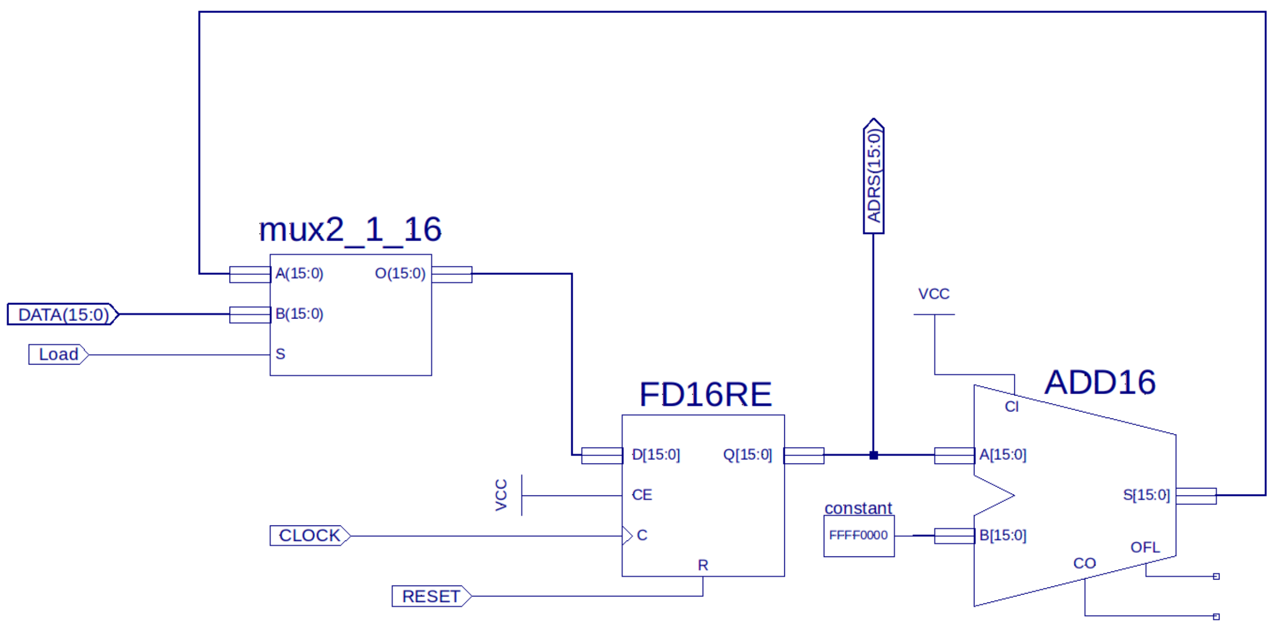
Processor Designed using Schematics



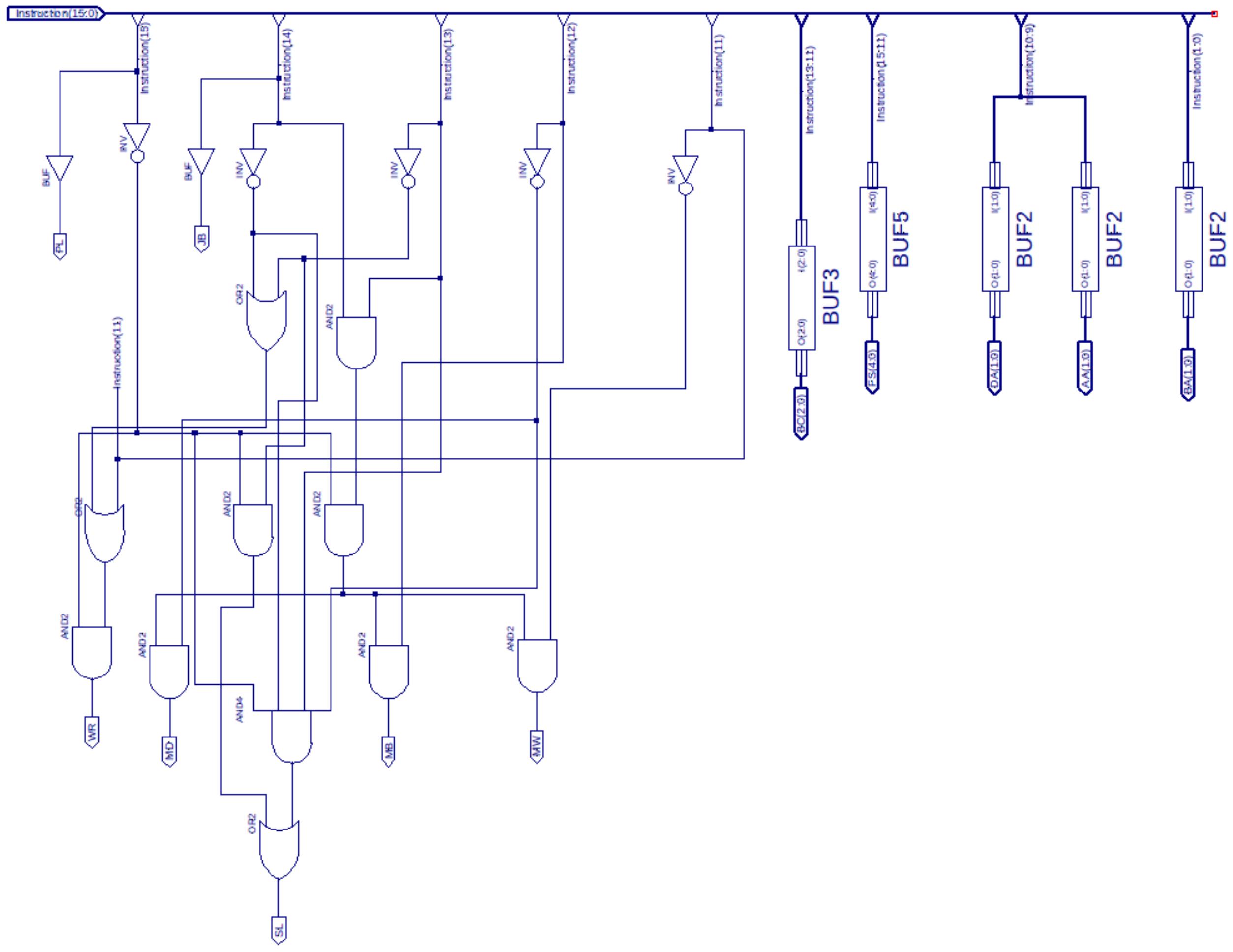
B.1 Control Unit



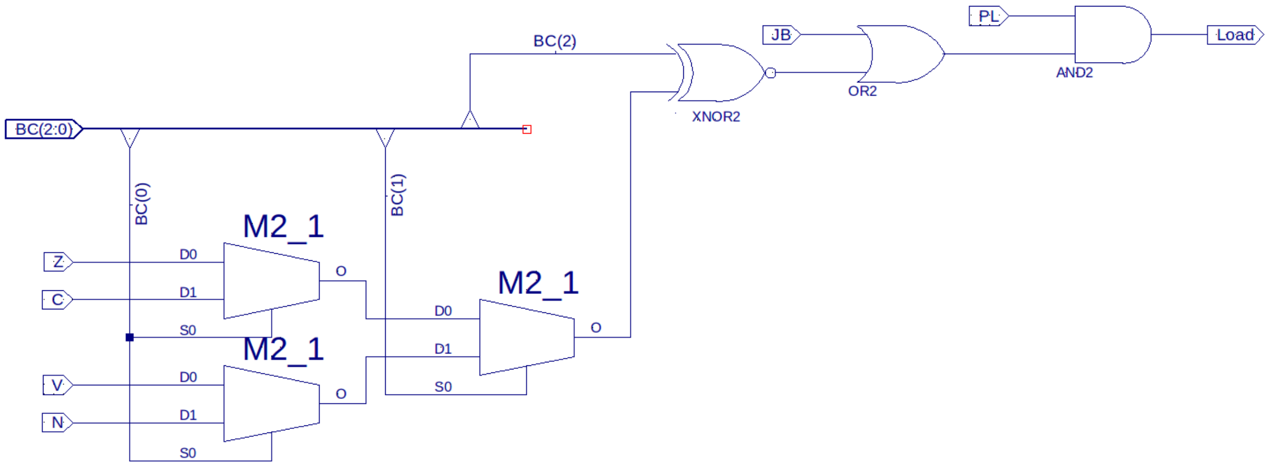
B.1.1 PC



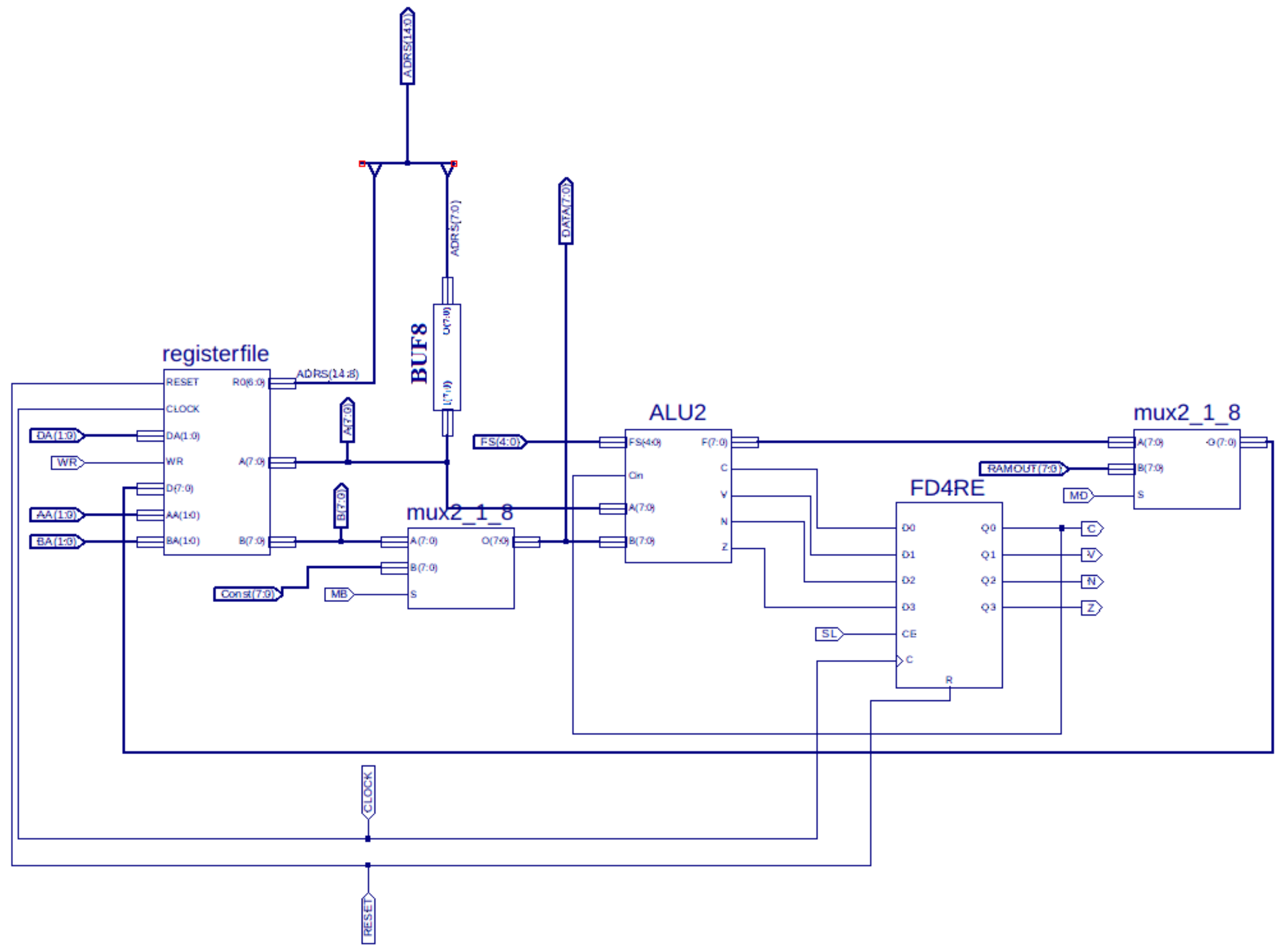
B.1.2 Instruction Decoder



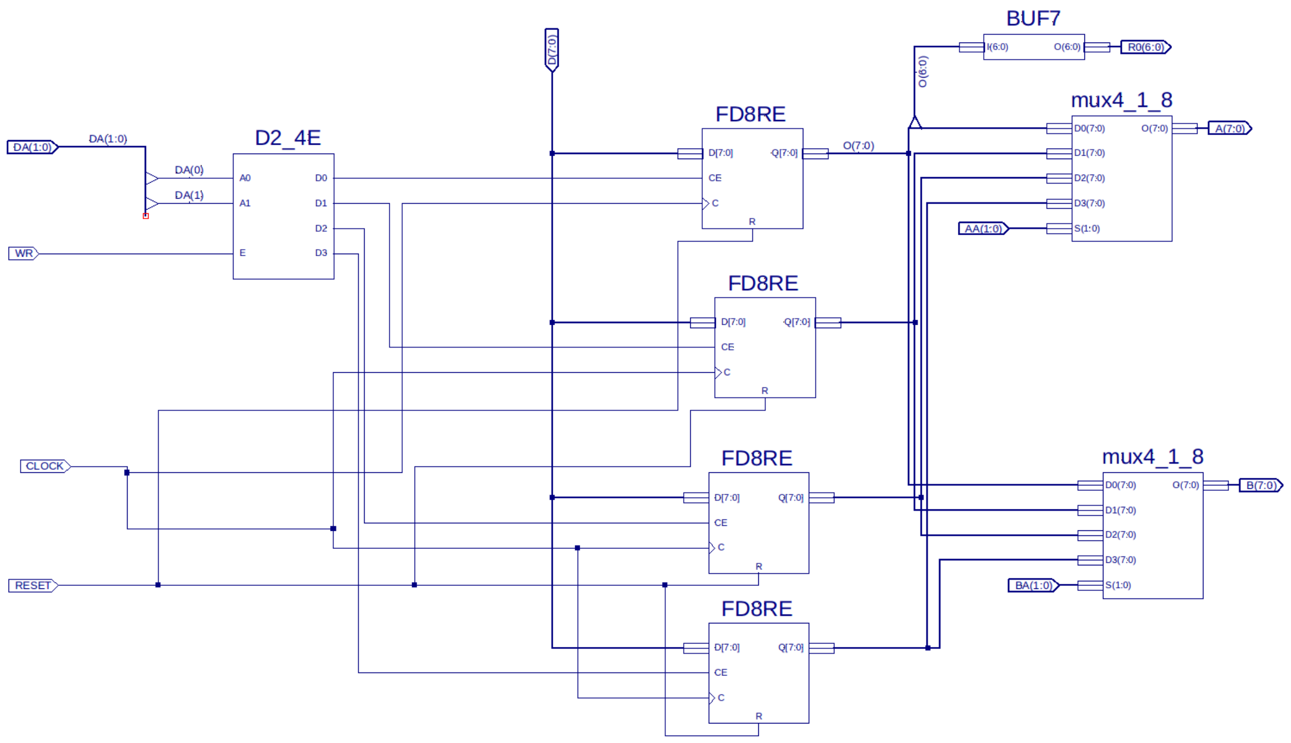
B.1.3 Branch Control



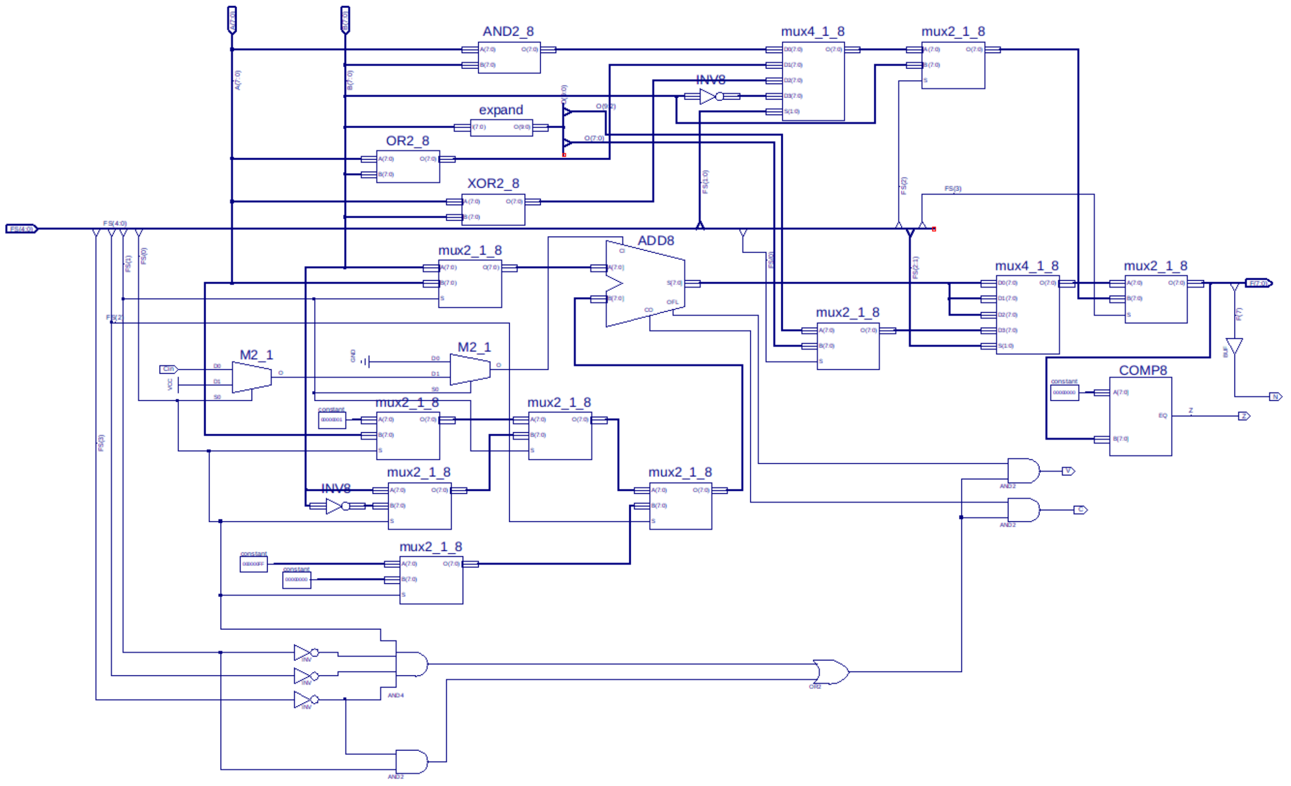
B.2 Data Path


ADRS(14:0)
ADRS(7:0)
DATA(7:0)
BUF8
registerfile
RESET
CLOCK
DA(1:0)
WR
D(7:0)
AA(1:0)
BA(1:0)
R0(6:0)
A(7:0)
B(7:0)
ADRS(14:8)
mux2_1_8
Const(7:0)
MB
S
O(7:0)
ALU2
FS(4:0)
Cin
F(7:0)
C
V
N
Z
FD4RE
D0
D1
D2
D3
CE
Q0
Q1
Q2
Q3
R
SL
mux2_1_8
RAMOUT(7:0)
MD
CLOCK
RESET

B.2.1 Register File



B.2.2 ALU



Bibliography

[gat16] Logic gate. https://en.wikipedia.org/wiki/Logic_gate, 2016.

[ise12] ISE In-Depth Tutorial. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ise_tutorial_ug695.pdf, 2012.

[JH93] D. J. Jackson and S. J. Hannah. Modelling and comparison of adder designs with Verilog HDL. *Southeastern Symposium on System Theory*, 1(25), 1993.

[PLV05] F. Pecheux, C. Lallement, and A. Vachoux. VHDL-AMS and Verilog-AMS as alternative hardware description languages for efficient modeling of multidiscipline systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(2), 2005.

[Sau89] J. Saunders. Logic synthesis from a hardware description language. *IEE Colloquium on 'High Level Modelling and Design for ASICs'*, 1(120), 1989.

[Smi16] Douglas J Smith. HDL Chip Design A practical guide for designing, synthesizing and simulating ASICs and FPGAs using VHDL or Verilog. http://ebook.pldworld.com/_eBook/FPGA%EF%BC%8FHDL/-Eng-/HDL%20Chip%20Design.%20A%20Practical%20Guide%20for%20Designing,%20Synthesizing%20and%20Simulating%20ASICs%20and%20FPGAs%20Using%20VHDL%20or%20Verilog%20(Douglas%20Smith).pdf, 2016.

[Ste15] Todor Stefanov. Processor Design Basics: Instruction Set Architecture. http://liacs.leidenuniv.nl/~stefanovtp/courses/DITE/lectures/DITE13.pdf, 2015.

[Vah09] Frank Vahid. Digital Design with RTL Design, Verilog and VHDL. *University of California, Riverside*, 1(2), 2009.