Design of a Teaching Instruction Set Processor in VHDL [1]

Diarmaid O'Cearuil
00632066

B.A. (Mod.) Computer Science
Final Year Project

Supervisor: Michael Manzke

May 4, 2005

[1] This document has been fully typeset in LaTeX

Abstract

The goal of this project was to design a micro-programmed microprocessor with its own instruction set as a teaching tool for 2nd year computer science students. Following on from Laura Redmond's project last year, I started by changing the existing functional components and then adding new ones. I changed and made additions to the control circuitry in order to micro-program the new functional components.
The next part of the project was to download the design onto an FPGA.
The final part of the project was to design a GUI that would be a tutorial for the students and allow the lecturer/student to create a new microprocessor project with the option of not including certain components so that the students would have to design these themselves.
The report outlines the difficulties of simulating a micro-programmed microprocessor and transfering the design to the board.

ACKNOWLEDGEMENTS

I would like to thank Michael Manzke, my supervisor for all the support and help he gave me throughout the project.
I also wish to extend thanks to Ross Brennan and Eoin Creedon for their help with the board.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter serves as an introduction to the Teaching Instruction Set Processor project. The background and motivations for this project will be outlined. This chapter will also discuss the approach taken to designing the microprocessor and the tools used to do so. At the end of the chapter there is a section dedicated to Laura Redmond's work on the project.

1.1 Project Motivation

The main aim of this project was that it could be effectively used as a teaching tool for Computer Science students studying the 2BA4 *Microprocessor Systems* module. The project is an extension to Ross Brennan's final year project two years ago in which he replaced the Motorola M68008 processor chip on an FPGA prototype project board with the LEON SPARC VHDL RISC implementation of a processor. Last year, Laura Redmond worked on Ross' board to develop a more creative and extensive instruction set. My aim was to expand on Laura's project and create an even more developed instruction set and time permitting, an interactive GUI to provide a tutorial and an interactive approach to learning about Instruction sets. FPGAs [1] and HDLs [2] have been around for quite a long time and it is important that their usefulness for teaching students Hardware Design be investigated.

1.2 Project Breakdown

The majority of the project consisted of writing and testing a microprogrammed microprocessor using VHDL [3]. I followed Laura's example by designing from a 'bottom up' perspective wherever possible i.e. starting with basic components and using them to design more complex components. Then, when thoroughly tested I would download the design onto the FPGA. I also developed a GUI, that would act as a tutorial for a student using the Microprocessor. It would also allow a Lecturer or student to create a new Microprocessor project file, with the option of removing certain components from it, hence the student would have to design the component themselves, appreciating the surrounding logic.

[1] Field Programmable Gate Array: A general purpose chip which can used to carry out a specific hardware funtion
[2] Hardware Description Languages
[3] Very High Speed Integrated Circuit Hardware Description Language

1.3 Software

This section focuses on the software tools used for this project.

1.3.1 VHDL

VHDL was chosen by Laura because it used prevalently in the Computer Science department. Xilinx's Project Navigator tool (version 6.2) was used to write the VHDL and synthesize it. Modelsim was used for testing the design. Synthesizable VHDL is discussed later, in chapter 4.

1.3.2 Impact

Impact was the software used to generate the bit file that would be later loaded onto the FPGA.

1.3.3 VB6

The GUI was written in Visual Basic 6.

1.4 Hardware

The microprocessor design was downloaded onto a Virtex II XC2V1000 Xilinx chip in an FPGA. See figure 1.1.

1.5 Laura Redmond's Microprocessor Project

Laura successfully implemented a microprocessor with a fairly complex and extensive instruction set. Figure 1.2 shows Laura's microprocessor from a tree perspective. If one thinks of the Microprocessor in terms of layers that instantiate other layers, then the microprocessor is the top layer. It is made up of the control unit and the datapath layers. The datapath layer instantiates the ALU and register file layers which in turn, instantiate more layers. The control unit instantiates all the control hardware. The register file consists of 4 registers which store 8-bit values, a decoder and a multiplexer. The ALU is made up of a logic unit, a shifter and a ripple-carry adder [4]. The control unit is the part of the microprocessor that sends the required signals to the Datapath, enabling it to perform the required operation on the required operands. The control unit will be discussed in greater detail in Chapter 3.

1.6 Phases of the Project

Since I inherited a large amount of complex code it was a necessity to plan the steps in which I would tackle each part of the project. I decided to go with a 'bottom up' approach, adopting Laura's ethos, making the project easier to design and test. The main phases were as follows:

[4]The functionality of Laura's Register File, ALU and Datapath are explained in more detail in Chapter 2

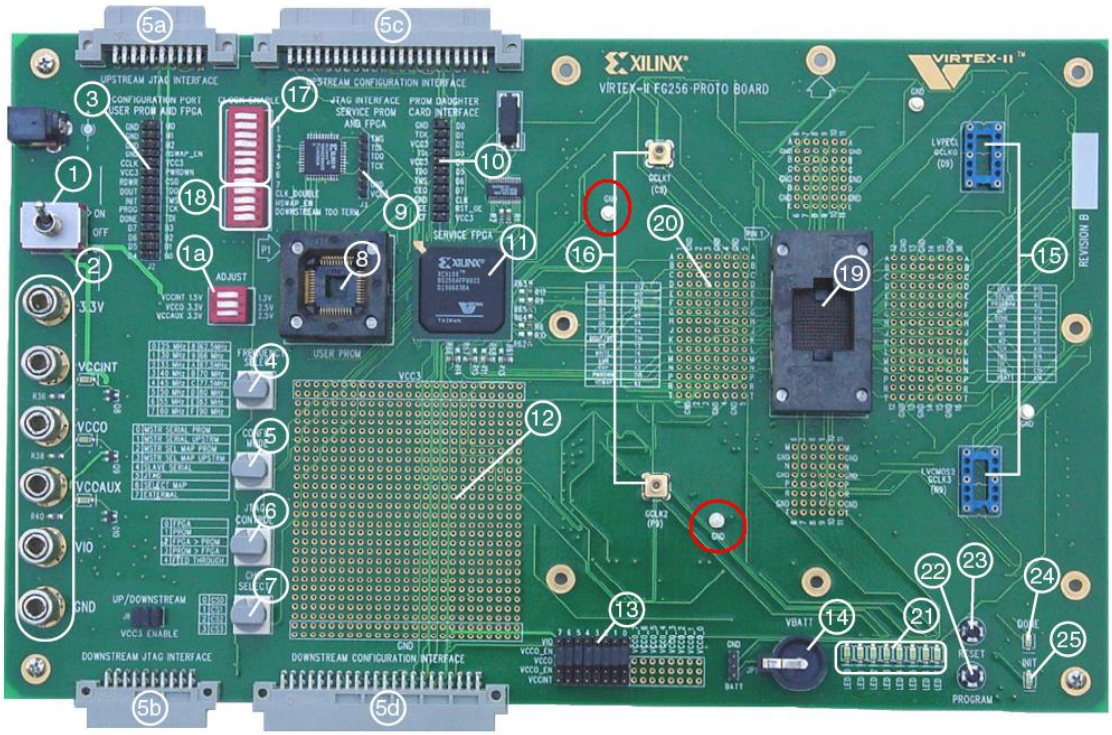

Figure 1.1: FPGA Prototype Project Board

[15] Taken form Laura Redmond's Report

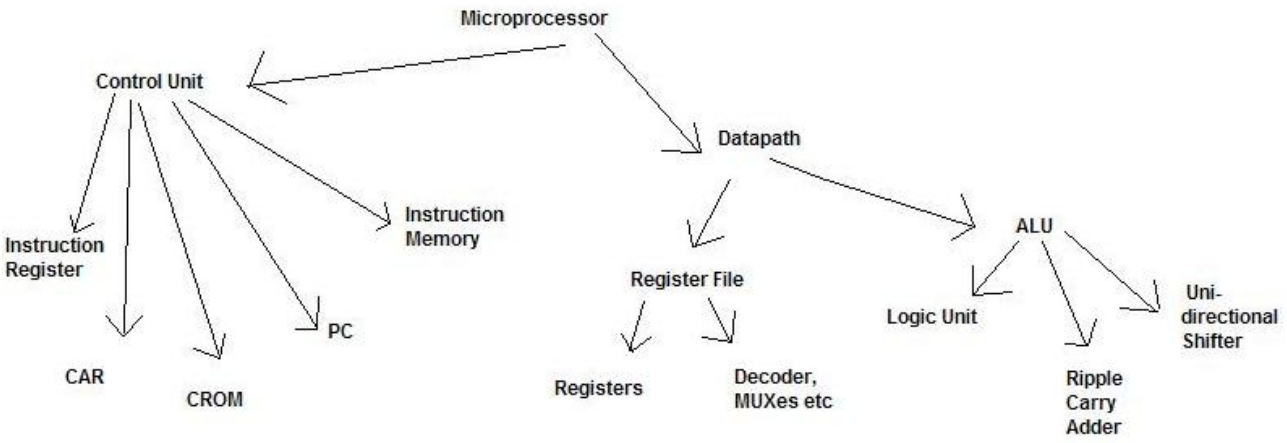

Figure 1.2: Hierarchy of Laura's Microprocessor Project

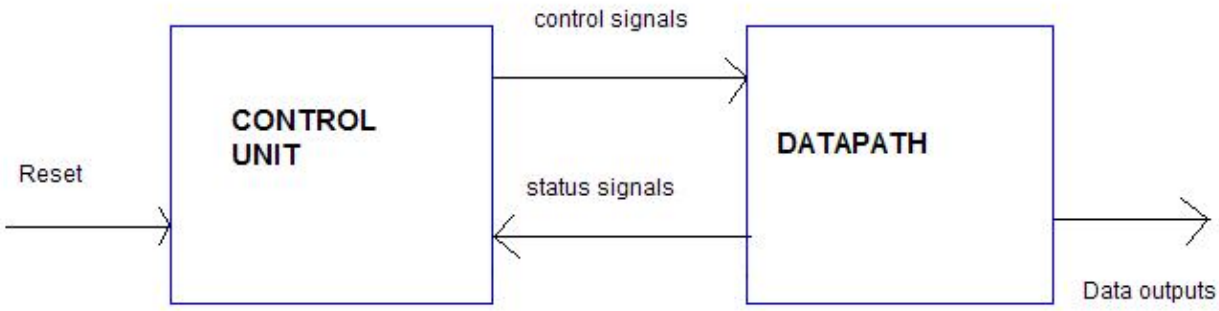

Figure 1.3: Microprocessor

Phase 1

I reviewed all of Laura's work and made sure that everything worked correctly. I also read up on the topic.

Phase 2

The existing ALU components were updated and tested, individually and together.

Phase 3

The size of the registers in the register file were increased. The size of the decoder and MUXes were upgraded also.

Phase 4

New components were designed, tested and integrated into the ALU.

Phase 5

The entire datapath was tested.

Phase 6

Laura's controlwords were revised and changed wherever necessary (due to changes in the datapath), and were then tested.

Phase 7

New controlwords for were added for the new components of the ALU.

Phase 8

Conditional branches were added and the microprocessor was thoroughly tested. [5].

Phase 9

The GUI was written.

Phase 10

The design was put on the board.

[5]Problems in this phase will be discussed in detail in Chapter 3

Chapter 2

Hardware Design

2.1 Overview

This chapter outlines the functionality and implementation of layer that performs the operations and stores the data, the datapath. See figure 2.1.

2.2 The Register File

In the register file, the destination register (the register being written to) is decided upon using a decoder. Since the size of the number of registers was expanded to 8, changes had to made to the entire register file: The decoder takes in a 3-bit binary value, instead of a 2-bit value from an opcode instruction identifying the destination register. The resultant output of the decoder is used as read/write select for each of the registers. Likewise, the AA and BA signals (coming from the opcode) were changed from 2-bit to 3-bit values. These signals are used as select lines for the A and B busses respectively. The multiplexers were expanded from 4:1 to 8:1.
The size of the data stored in the register file was updated from 8 bits to 32 bits. The motivation for this was that since I had planned to integrate a multiplier and a floating point unit into the design that larger values than 8 bits would be needed to get decent, useful results from them. These changes produced few problems. See figure 2.2.

2.3 Changes to the ALU

Laura's ALU consisted of a unidirectional shifter, a logic unit, a ripple carry adder and 2 multiplexers to choose the necessary inputs and outputs. The select lines for these multiplexers came from the control instructions issued by the control unit. The following changes were made to these components:

2.3.1 Shifter

Laura's shifter took in an 8-bit value and shifted it left depending on a 3 bit value coming from the control unit. It was changed to a barrel-shifter capable of shifting a 32-bit value left or right depending

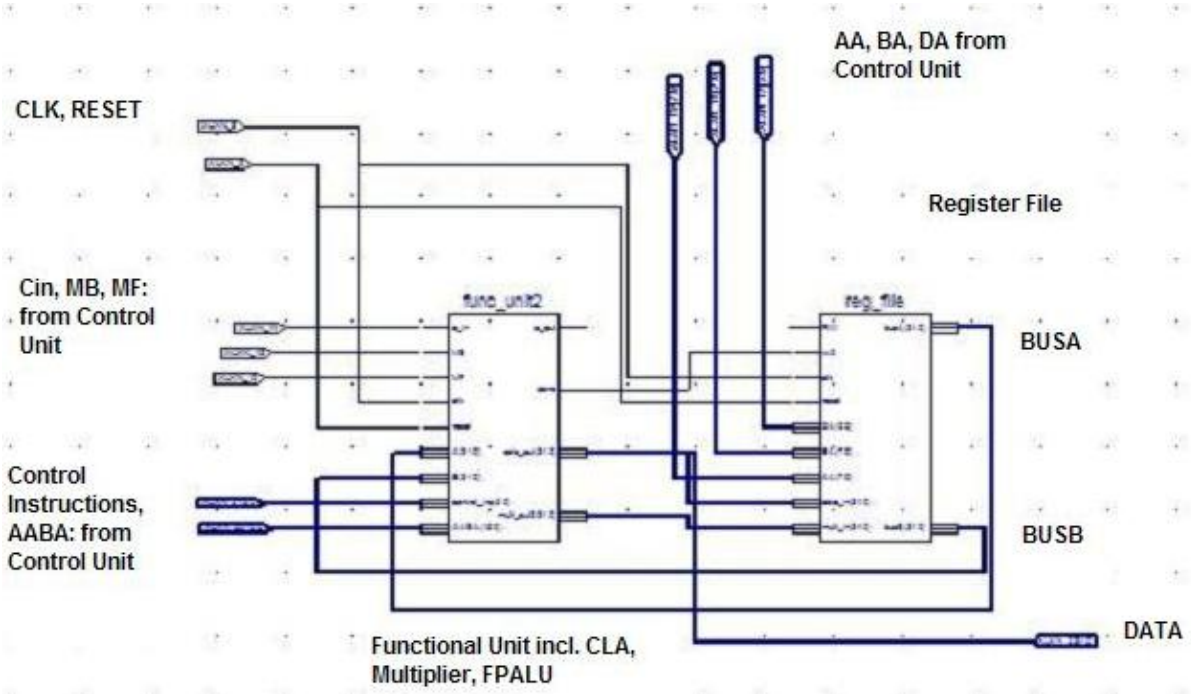

Figure 2.1: Schematic Representation Of Datapath

on another control signal from the control unit. The process of updating the shifter was an easy enough task but was time-consuming.

2.3.2 Adder

The ripple-carry adder was changed to a carry-look-ahead adder. The reason for this is that although ripple-carry design works perfectly, it has a huge gate delay associated with it. For an n-bit ripple-carry adder there would be 2n + 2 gate delays. Hence for a 32 bit adder there would be an inefficient 66 gate delays. The solution to this was to design an adder with the carry-look-ahead (CLA) technique. The CLA would consist of more complex hardware but would reduce the number of gate delays significantly.
The implementation of the CLA proved time-consuming. Three boolean equations had to be written for each bit in the adder, with the logic for each one getting more complicated as the number of bits increased. This meant that a simple mistake, like a 'typo' was difficult to find and hence, correct.

2.3.3 Logic Unit

The logic unit is the hardware where the logical operations are performed. The only changes that needed to be made here were to update it to 32-bits. Later on, the size of the multiplexer, which chooses the output of the logic unit, was reduced from 8:1 to 4:1. This was because there were four unused inputs to the multiplexer and that could lead to undesirable situations when later testing the complete Microprocessor.

2.4 Additions to the ALU

Once all the alterations to the to the ALU were thoroughly tested I added in new components. They are described below:

2.4.1 Multiplier

Laura's microprocessor executed single-cycle instructions only. I thought it would be good to expand it to be able to execute multi-cycle instructions. It was decided that a multiplier component would be the most useful multi-cycle instruction to include. The multiplier was designed using Finite State Machines, roughly following an example from Mano [8] that takes 64 clock cycles to execute an instruction. Since the result of multiplying two 32-bit numbers would more often than not require more than 32 bits, the output of the unit would have to be 64 bits as to reduce the risk of inaccurate results. A special 'temporary' register was added to the register file to hold the upper 32 bits before the 64 bits were stored in memory. There were a few difficulties integrating the new temporary register. Since I also wished for this register to be used by other functions I had to integrate two chip-enable signals into the design. The enable signal for multiplications would be the 'done' signal from the multiplier. The difficult part regarding the actual multiplier came later when sequencing and timing became a difficult issue. These problems will be discussed in Chapter 3.

2.4.2 Floating Point Adder

I obtained a copy of a floating point adder from a classmate. It was a project in the 3BA5 module last year, however I had deleted mine so I had to seek out another copy of it. The FP adder executes instructions in 7 clock cycles. This component also yielded timing and sequencing problems later in the project.

A few changes had to be made to it. The Adder was designed using schematics (See Figure 2.2 or figure 2.3 for example schematics). Schematics are a feature of VHDL that allow the designer to connect components using a Graphical User Interface. Basically, the designer can compile synthesized components into symbols which can be 'drag and dropped' into the GUI and then interconnected with wires, and input and output markers. These schematics can then be synthesized themselves to see if the components port map correctly, like in a port map and can also be tested with testbench waveforms.

My project used port maps and not schematics. Therefore I had to modify every layer in the floating point adder. This was more time consuming than difficult. I had to reacquaint myself with floating point arithmetic and its implementation in hardware as well as code every schematic in to a port map and thoroughly test it. Changes had to be made to most of the sub-components so that they integrated properly with the rest of the ALU and datapath. Later on in the project, this unit had to be modified again to insert a done signal, that would tell the control unit when a floating point computation was complete (see Chapter 3 for further details).

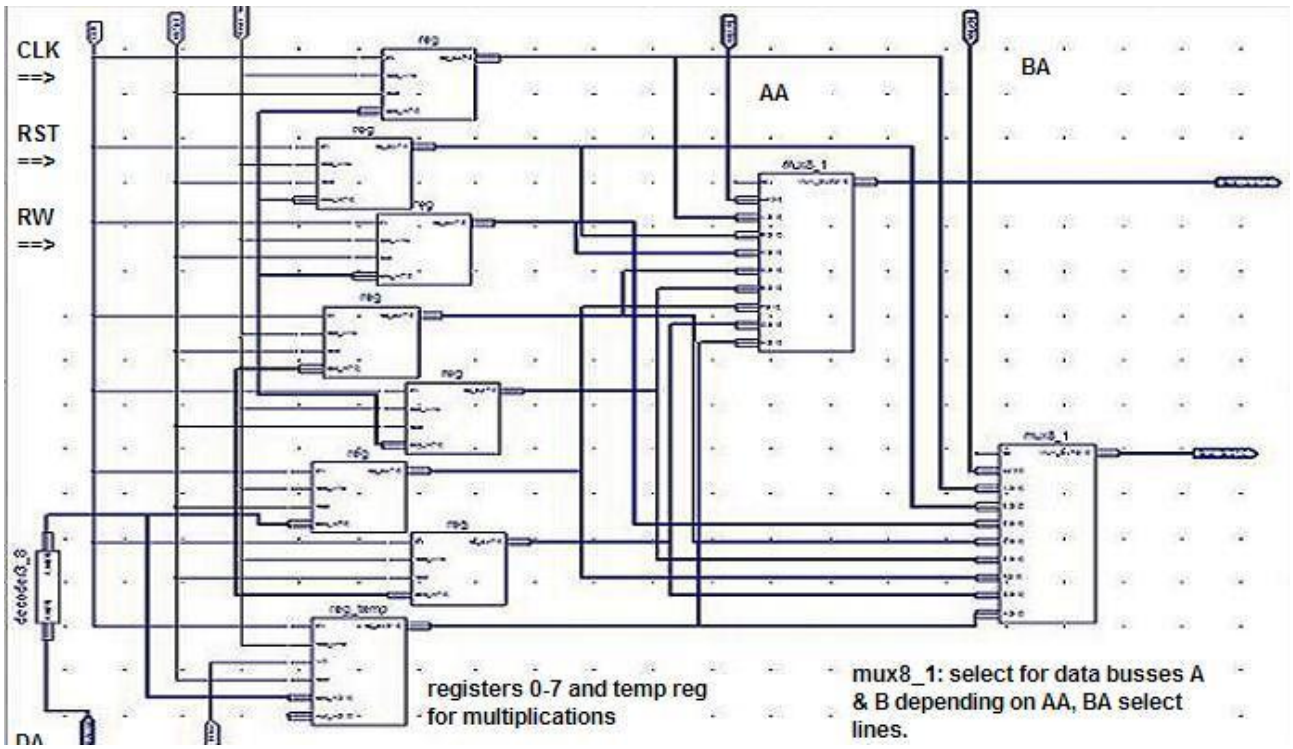

Figure 2.2: Schematic Representation of the Register File with incorporated changes and additions

2.4.3 Memory Unit

A local memory unit was added as well. The main reason for this was so that the values in the memory could be set before the Microprocessor was run ande values could be loaded into the registers at the beginning so that I could check the outputs from the testbenches and be sure of accurate results. Otherwise, arithmetic would be performed on zeros or on all ones. [1]

2.4.4 Miscellaneous additions

I added one of the inputs to the ALU directly to the 8:1 multiplexer which selects the output of the unit. This was done to enable MOVE instructions. See Figure 2.3 for a schematic of the ALU.

[1] what I mean by this is that the only values that could have been in the registers would be zero when reset goes high or all 1's when the NOT instruction is issued

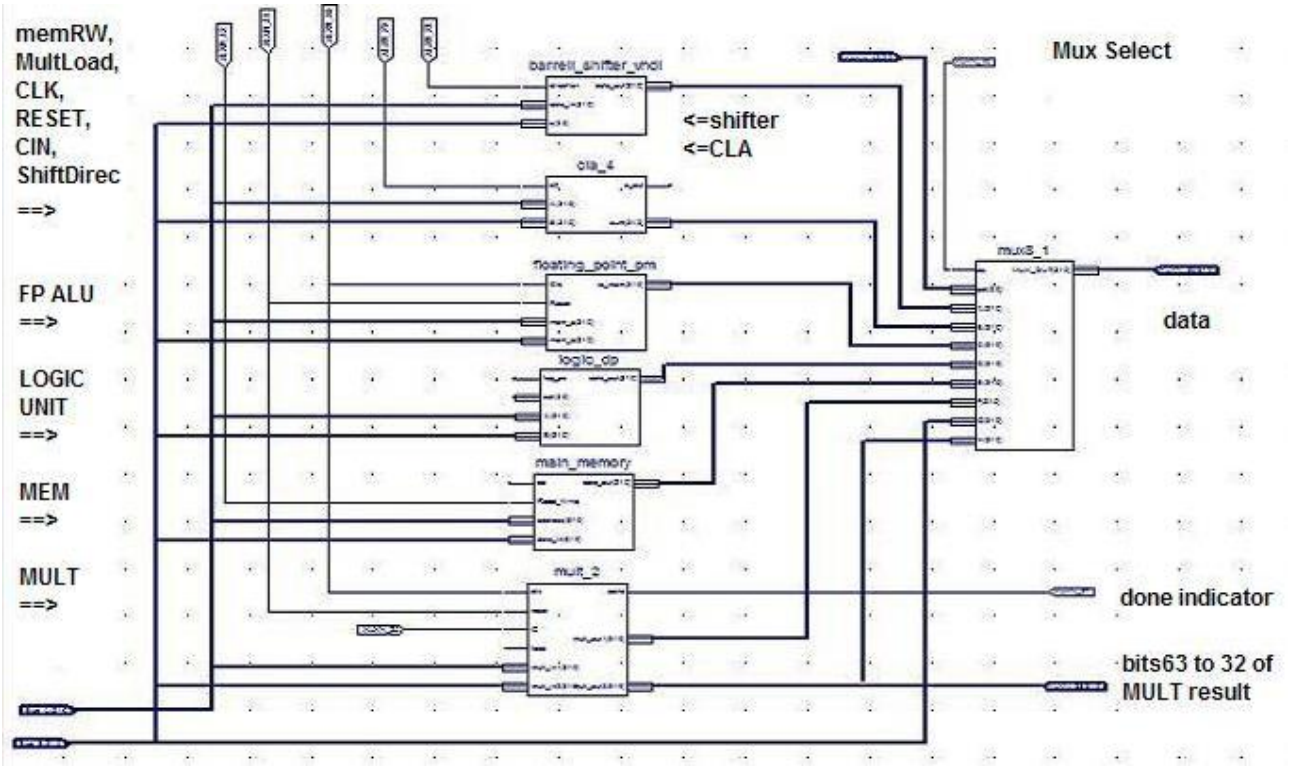

Figure 2.3: Schematic Representation of the ALU with incorporated changes and additions

Chapter 3

Micro-programming the Design

3.1 Overview

The control unit is the part of the microprocessor that supervises the sequence of operations. In this chapter I will discuss the implementation of the control unit and the problems that arose.

3.2 Laura's Control Unit

Figure 3.1 is the state machine for Laura's micro-programmed microprocessor. When the address is "00000" (Instruction Fetch), the instruction register is loaded with the instruction from the instruction memory pointed to by the PC, and the PC is incremented. The immediate next instruction is Execute, which checks the value in the instruction register and then executes that instruction. After that instruction is executed, the next state is Instruction Fetch again.
The control unit layer instantiated a program counter (PC), an instruction memory (IM), an instruction register (IR), a multiplexer (MUXC), a control address register (CAR) and a control ROM (CROM). The instruction memory stored the instruction opcodes which made up the simple test program. The CROM held the controlwords which controlled the sequencing of the control unit and output results of the data path. The PC and CAR pointed to address zero of the IM and CROM respectively on reset going high.Instruction Fetch (IF) is the controlword at address zero, (See Laura's controlword table in table 3.1). Therefore, the instruction at address zero in the IM is loaded into the IR as determined by the IL control bit in the IF controlword. This instruction also increments the PC (via the 1 value in the PI control bit). The next address pointed to by the CAR is determined by the MC bit in the controlword. If it.s a 1, the opcode in the instruction memory is the address value ready to be loaded into the CAR, otherwise the address in the NA field of the controlword is used as the address pointer in the CROM. This decision is carried out by the above-mentioned MUXC. In the IF controlword the NA field is always chosen (MC is low). This NA points to the EX controlword which chooses the OPC address (MC is high). On the next clock cycle this OPC address is loaded into the CAR (the CAR is clocked) and the instruction we fetched in the IF state will now execute. At this point, the control unit entity will output the fetched instruction's corresponding controlword. Laura later added the hardware needed for unconditional branches and jumps.

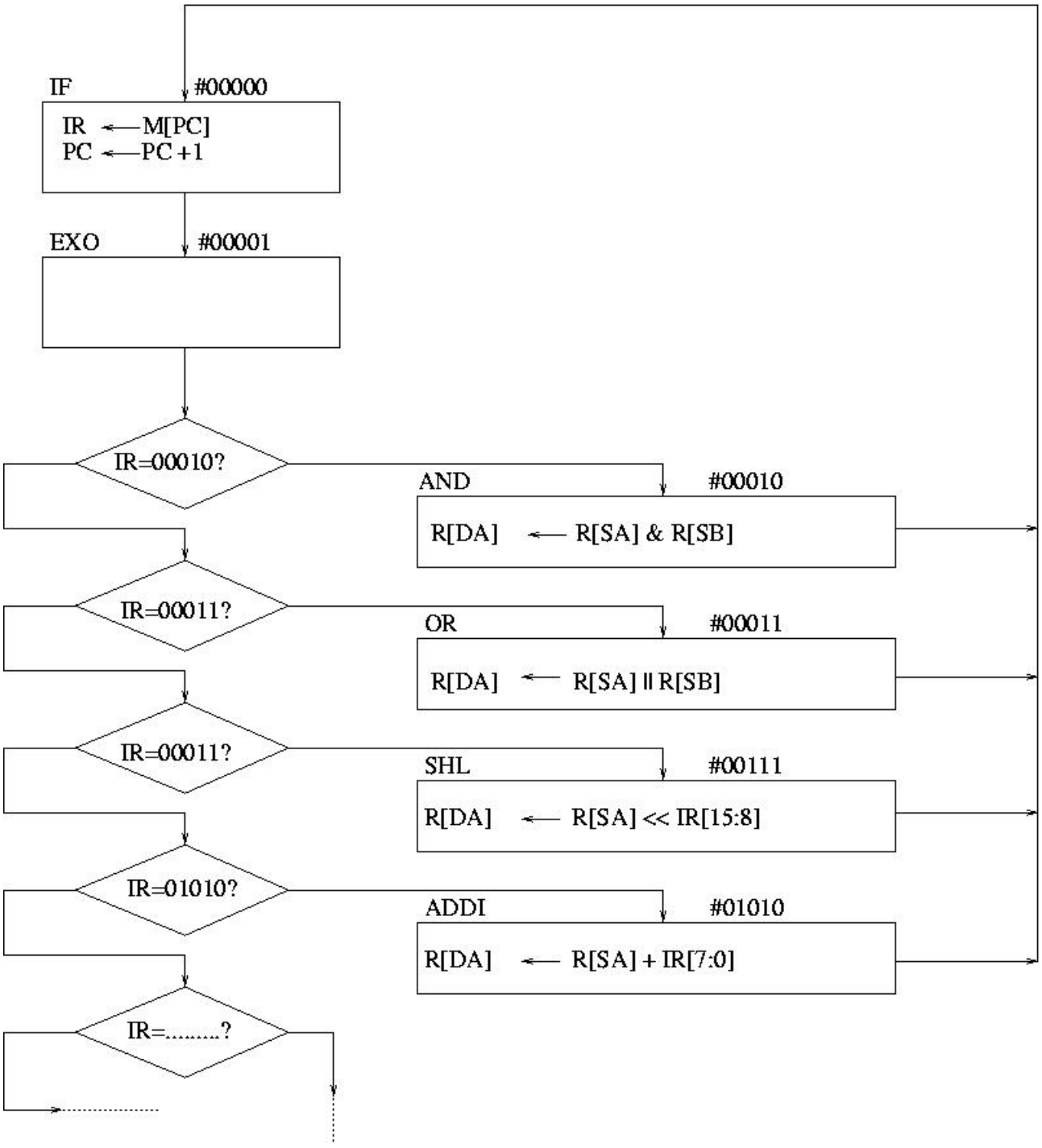

Figure 3.1: State Machine of a micro-programmed microprocessor

[15] Taken from Laura's Final Year Project Report

		31	30	29	28	27	26 - 22	21	20	19	18	17	16	15	14	13 - 9	8	7	6 - 2	1	0
Address	Instruction	-	-	-	-	ENABLE	NABRA	DV	DC	RW	MF	CIN	MW	MR	MB	CONTINS	PI	PL	NASEQ	MC	IL
00000	IF	-	-	-	-	0	00001	-	0	1	-	-	-	0	-	11—	1	0	00001	0	1
00001	EXO	-	-	-	-	1	00000	-	0	1	-	-	-	-	-	11—	0	0	00000	1	0
00010	AND	-	-	-	-	1	00000	-	0	0	-	-	1	1	1	01000	0	0	00000	0	0
00011	OR	-	-	-	-	1	00000	-	0	0	-	-	1	1	1	01001	0	0	00000	0	0
00100	NOT	-	-	-	-	1	00000	-	0	0	-	-	1	1	1	01010	0	0	00000	0	0
00101	BUF	-	-	-	-	1	00000	-	0	0	-	-	1	1	1	01011	0	0	00000	0	0
00110	XOR	-	-	-	-	1	00000	-	0	0	-	-	1	1	1	01100	0	0	00000	0	0
00111	SHIFT	-	-	-	-	1	00000	-	0	0	-	-	1	1	1	00000	0	0	00000	0	0
01000	STO	-	-	-	-	1	00000	-	0	0	-	-	0	1	1	01101	0	0	00000	0	0
01001	LD	-	-	-	-	1	00000	-	0	0	-	-	1	0	1	01110	0	0	00000	0	0
01010	ADDI	-	-	-	-	1	00000	-	0	0	-	0	1	1	0	10—	0	0	00000	0	0
01011	NOTI	-	-	-	-	1	00000	-	0	0	-	-	1	1	0	01010	0	0	00000	0	0
01100	ADD	-	-	-	-	1	00000	-	0	0	-	0	1	1	1	10—	0	0	00000	0	0
01101	SUB	-	-	-	-	1	00000	-	0	0	-	1	1	1	1	10—	0	0	00000	0	0
01110	NOP	-	-	-	-	1	00000	-	0	0	-	-	-	-	-	11—	0	0	00000	0	0
01111	JMP	-	-	-	-	1	00000	-	0	1	-	-	1	1	1	11—	0	0	00000	0	0
10000	JMP2ITSELF	-	-	-	-	1	00001	-	1	0	-	-	-	-	-	11—	0	0	00001	0	0
10001	BRAC1	-	-	-	-	1	10010	-	0	1	-	-	-	-	-	11—	0	0	00000	0	0
11010	BRAC2	-	-	-	-	1	10100	-	0	1	-	-	-	-	-	11—	0	0	—–	0	0
11011	MOVE	-	-	-	-	1	00000	-	0	0	-	-	1	1	1	01011	0	0	00000	0	0
11100	BRA3	-	-	-	-	1	00001	-	0	1	-	-	-	-	-	11—	0	0	00001	0	1
11101	MOVEI	-	-	-	-	1	00000	-	0	0	-	-	1	1	0	01011	0	0	00000	0	0
11110	SUBI	-	-	-	-	1	00000	-	0	0	-	1	1	1	0	10—	0	0	00000	0	0

Table 3.1: Laura's Final ControlWord table

Instruction Name	Opcode	Action
LOAD1	– 4A – F9	R2 $\leftarrow$ M[01]
LOAD2	– 4B – FA	R3 $\leftarrow$ M[02]
NOP	– ?E – –	No operation
NOTR7	– 27 FF FF	R7 $\leftarrow$ NOT(R7)
SHL	– 3F 0F FF	R7 $\leftarrow$ R7 $<< \times$ 8
SHR	– BF FF FF	R7 $\leftarrow$ R7 $>> \times$ 8
MOVE	– AA – 01	R2 $\leftarrow$ R1

Table 3.2: Initial Test Program

3.3 Change/Additions to ControlWords

Since I had made changes to the ALU, changes had to be made to nearly all of the existing controlwords. This proved to be straight forward enough as there were no timing issues due to all the instructions being single-cycle ones. A SHR, a new LOAD, STORE and MOVE controlword were added. Table 3.2 shows a sample test program that was successfully run.

		31 - 29	28	27 - 23	22	21	20	19	18	17	16	15	14 - 9	8	7	6 - 2	1	0
Address	Instruction	BRA INS	ENABLE	NABRA	DELAY	DC	RW	BRA RST	CIN	MW	MR	MB	CONTINS	PI	PL	NASEQ	MC	IL
00000	IF	- - -	0	00001	1	0	1	1	-	-	-	-	011—	1	0	00001	0	1
00001	EXO	- - -	1	00000	1	0	1	-	-	-	-	-	011—	0	0	00000	1	0
00010	AND	- - -	1	00000	0	-	0	-	-	-	-	1	001000	0	0	00000	0	0
00011	OR	- - -	1	00000	0	-	0	-	-	-	-	1	001001	0	0	00000	0	0
00100	NOT	- - -	1	00000	0	-	0	-	-	-	-	1	001010	0	0	00000	0	0
00110	XOR	- - -	1	00000	0	-	0	-	-	-	-	1	001010	0	0	00000	0	0
00111	SHL	- - -	1	00000	0	0	0	-	-	1	1	1	000000	0	0	00000	0	0
10111	SHR	- - -	1	00000	0	0	0	-	-	1	1	1	000100	0	0	00000	0	0
01000	STORE	- - -	1	00000	0	0	0	-	-	0	1	1	100010	0	0	00000	0	0
01001	LOAD	- - -	1	00000	0	0	0	-	-	1	0	1	100011	0	0	00000	0	0
01010	ADDI	- - -	1	00000	0	0	0	-	0	-	-	0	010—	0	0	00000	0	0
01011	NOTI	- - -	1	00000	0	-	0	-	-	-	-	0	001100	0	0	00000	0	0
01100	ADD	- - -	1	00000	0	-	0	-	0	-	-	1	010—	0	0	00000	0	0
01101	SUB	- - -	1	00000	0	-	0	-	1	-	-	1	010—	0	0	00000	0	0
01110	NOP	- - -	1	00000	0	0	1	-	-	-	-	-	111—	0	0	00000	0	0
10101	MOVE	- - -	1	00000	0	0	0	-	-	1	1	1	110—	0	0	00000	0	0
01111	JMP	- - -	1	00000	0	0	1	-	-	-	-	-	111—	0	1	00000	0	0
10000	JMP2ITSELF	- - -	1	00001	0	0	1	-	-	-	-	-	011—	0	0	00001	0	0
10001	BRAC1	- - -	1	10010	0	1	1	-	-	-	-	-	01—-	0	0	00000	-	0
11010	BRAC2	- - -	1	10100	0	1	-	-	-	-	-	-	——	0	0	——	0	0

Table 3.3: Control Word Table (1st Draft)

Branch Condition	BRA INS	Mnemonic	Condition	Status Bit
Branch If Higher	000	BHI	A $<$ B	C + Z = 0
Branch If Higher or Equal	001	BHE	A $\geq$ B	C = 0
Branch If Lower	010	BLT	A $<$ B	C = 1
Branch If Lower or Equal	011	BLE	A $\leq$ B	C + Z = 1
Branch If Equal	100	BEQ	A == B	Z = 1
Branch If Not Equal	101	BNE	A != B	Z = 0

Table 3.4: List of Conditional Branches

3.4 Conditional Branches

A conditional branch is a branch that is either taken or not taken due to status bits that come from the datapath. The status bits are the carry and zero flags, set if there was a carry/borrow or a computation equal to zero performed, respectively. A branch control component was designed that took these two inputs and a 3-bit input, BRA INS from the controlword and outputted a signal that acted as a select line for a MUX which would choose either the next address in the PC or the next NABRA address. When I began testing the branch instructions I came into a number of problems, which will be discussd in detail later in this chapter.

3.5 Multiplier and Floating Point Adder Controlwords

Since the multiplier and floating point instructions would need more than a single clock cycle I needed to come up with a way of delaying the next instruction until the the desired computation was complete. I returned to the above mentioned units in the datapath and changed them, including 'done' signals which would be outputs of the datapath layer and inputs to the control unit layer. This was an easy enough task for the multiplier as there was already an internal signal that was essentially a done signal. However, there was no such signal in the floating point unit. The unit was remodelled to propagate the start signal through a series of flip-flops until it reached the end of the pipelined floating point adder, thus acting as a done signal. In the control unit, these done signals would be inputs to a modified branch control component. In the controlword for the Multiplier instruction the NABRA field would be set to the address of a 'hidden' MULTNOP controlword. [1] This hidden controlword would evaluate the 'branch out' signal from the Branch control unit and whenever high, would set the next instruction to be executed to be the address in the NABRA field, which is itself. When the done signal goes high the branch out signal will go low, allowing the normal flow to continue while also allowing the right value to be written to the register file.
Similarly for floating point addition, there is a FPNOP controlword that will loop until the done signal from the datapath goes high. However, for the FPNOP the signal was staying high indefinitely,

[1] By hidden, I mean that it would be invisible to the programmer, so that it could not be issued directly from instruction memory

```
case bra_ins is
    when "000" =>    -- 000 Branch Higher (BHI)
        if (c or z)= '0' then
            branch_out <= '1';
        else
            branch_out <= '0';
        end if;
    when "001" =>   -- 001 Branch if Higher or Equal (BHE)
        if c = '0' then
            branch_out <= '1';
        else
            branch_out <= '0';
        end if;
    when "010" =>   -- 010 Branch if Lower Than (BLT)
        if c = '1' then
            branch_out <= '1';
        else
            branch_out <= '0';
        end if;
    when "011" =>   -- 011 Branch if Lower or Equal (BLE)
        if (c or z) = '1' then
            branch_out <= '1';
        else
            branch_out <= '0';
        end if;
    when "100" => -- 100 Branch if Equal  (BEQ)
        if z = '1' then
            branch_out <= '1';
        else
            branch_out <= '0';
        end if;
    when "101" =>    -- 101 Branch if Not Equal (BNE)
        if z = '0' then
            branch_out <= '1';
        else
            branch_out <= '0';
        end if;
    when others =>
        branch_out <= '0';
    end case;
```

Figure 3.2: Excerpt of Source Code from Branch Control component detailing the logic behind the branch signal

causing the processor to always branch. After a lot of testing, I discovered that the enable signal for the unit, which was being propagated through the unit as a done signal was being kept high by the FPNOP operation, hence the loop. The appropriate changes were made to the FPNOP controlword and the microprocessor functioned correctly when executing the floating point add operation. See figure 3.3 for a testbench waveform of the floating point add instruction.

3.6 Problems

3.6.1 Branch Reset

When the branch control component was fully tested I port mapped it into the control unit layer. However, problems were arising when ever a branch was being taken. When the IF controlword was being executed the 'branch out' was remaining high and was interfering with the flow of the rest of the program. In order to remedy this, I inserted a branch reset input into the branch control unit, that would reset the branch control unit whenever the next instruction was being fetched.

3.6.2 Delay Register

The branch reset signal only fixed the problem in a few cases. When a conditional branch was taken, the next sequential instruction was not being executed. This was because the branch out signal was staying high therefore the NABRA field was always being selected, creating an infinite loop. A normal branch reset signal would not suffice as it would reset the bit that needed to be examined. After many possible solutions failed to resolve the problems I added a new field to the controlword format, 'delay'. The delay bit is set to 1 in the conditional branch controlwords. A register was added to the control unit, called the delay register. The delay signal would come into the register and be stored there for one clock cycle then outputted to the branch control component where it would reset the branch output only after the original value was read. This problem was resolved. However, the solution created knock on affects.

3.6.3 Floating Point Adder

In figure 3.3 the floating point unit is working as expected. This was before the Delay Register was implemented. Once it was implemented, the floating point unit ceased to work properly in the microprocessor layer. However, it worked in the ALU and datapath layers. Due to timing constraints I was unable to locate the source of the error and correct it.
See Appendix A for a detailed table of the function of all the fields in the controlwords.

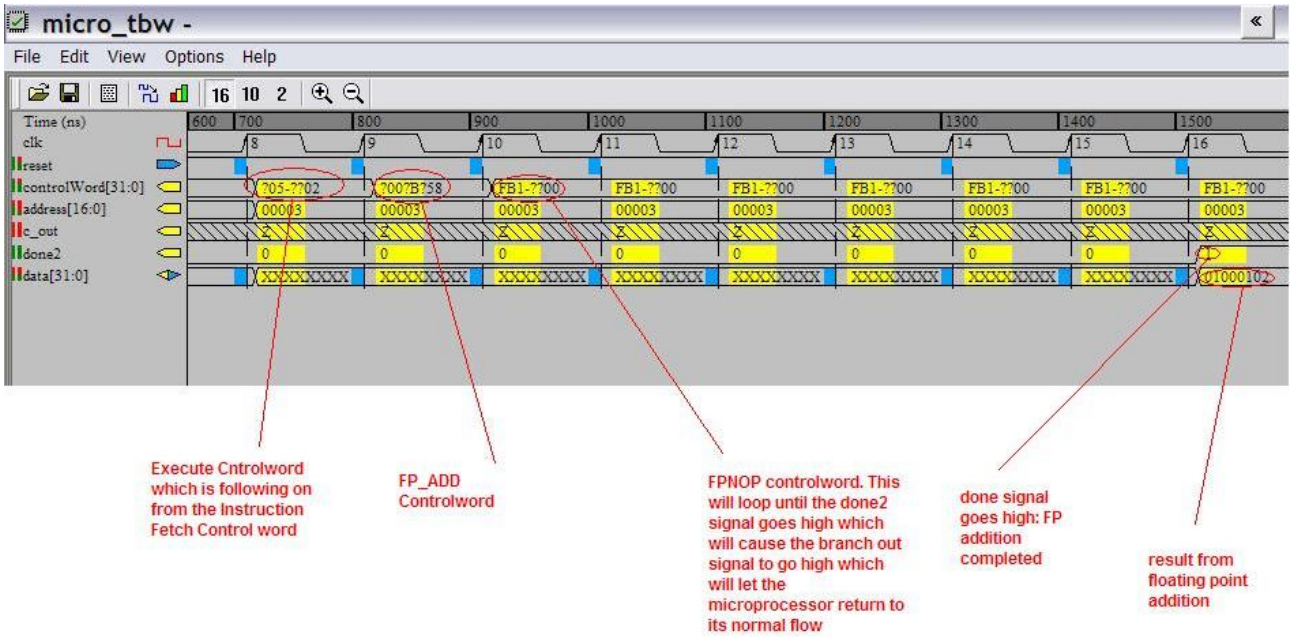

Figure 3.3: Testbench waveform that shows the done signal for floating point addition going high and then the result being written to the databus

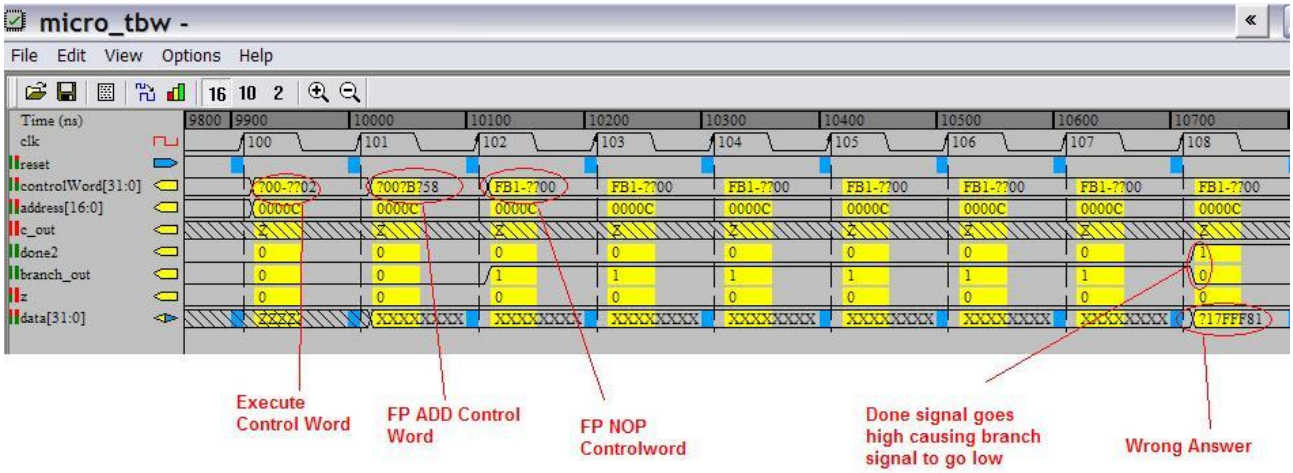

Figure 3.4: Testbench waveform that shows the broken Floating Point Adder. See Figure 3.3 for when it was working

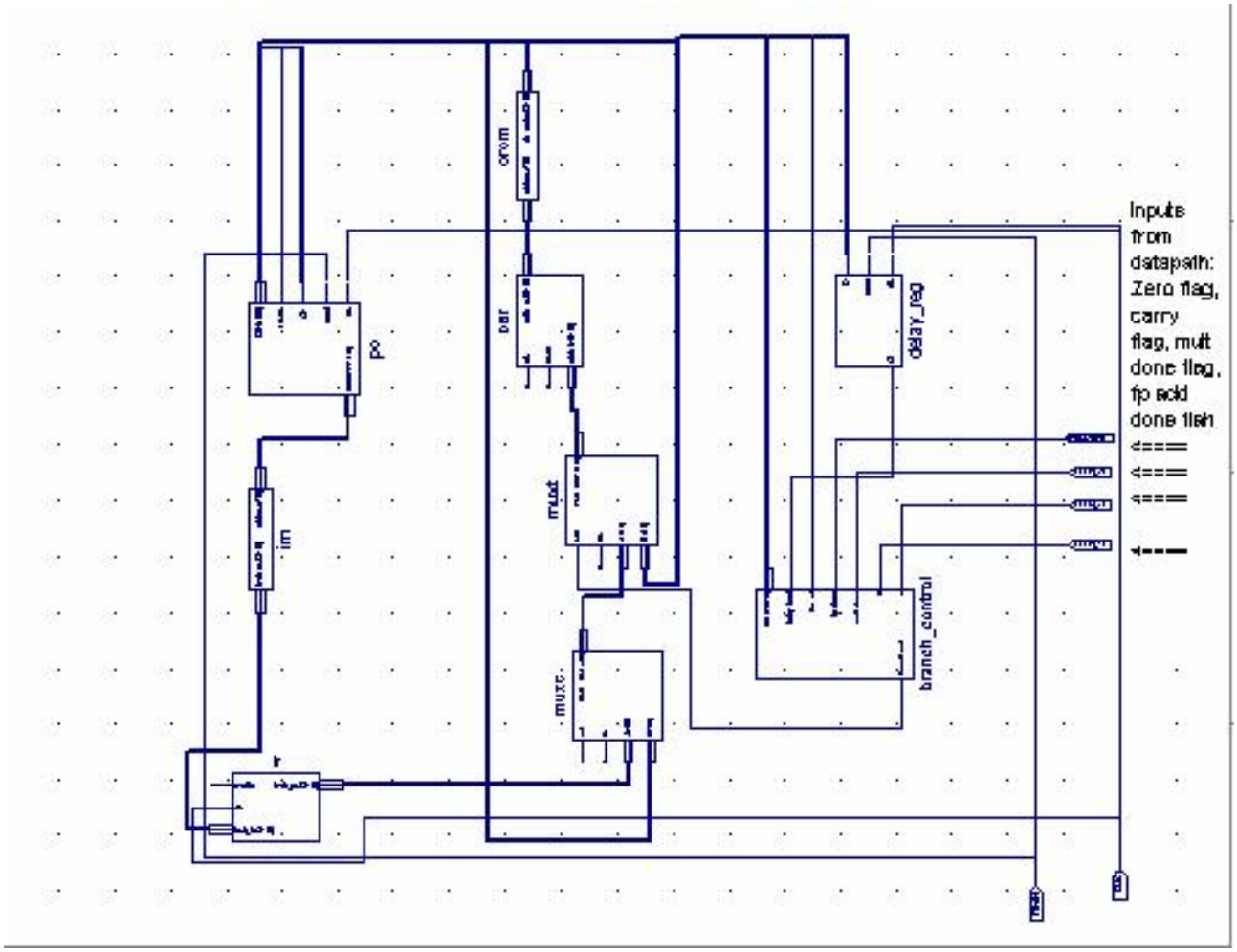

Figure 3.5: Schematic Representaion of Control Unit

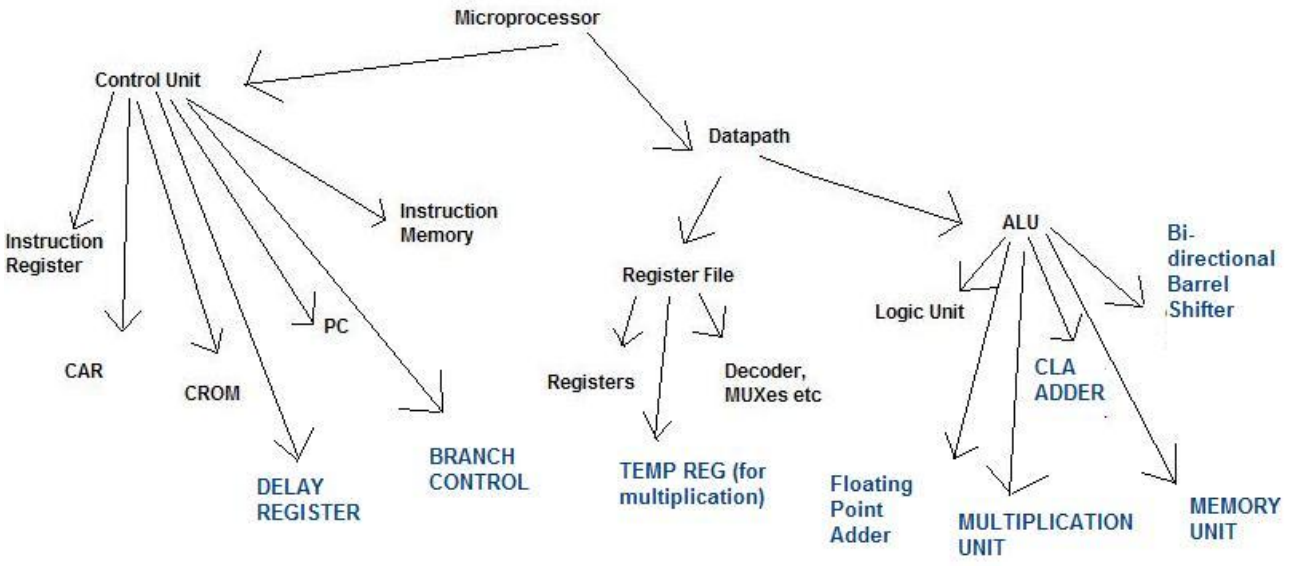

Figure 3.6: Tree representation of the completed Microprocessor project

		31 - 29	28	27 - 23	22	21	20	19	18	17	16	15	14 - 9	8	7	6 - 2	1	0
Address	Instruction	BRA INS	ENABLE	NABRA	DELAY	DC	RW	BRA RST	CIN	MW	MR	MB	CONTINS	PI	PL	NASEQ	MC	IL
00000	IF	- - -	0	00001	1	0	1	1	-	-	-	-	011—	1	0	00001	0	1
00001	EXO	- - -	1	00000	1	0	1	-	-	-	-	-	011—	0	0	00000	1	0
00010	AND	- - -	1	00000	0	-	0	-	-	-	-	1	001000	0	0	00000	0	0
00011	OR	- - -	1	00000	0	-	0	-	-	-	-	1	001001	0	0	00000	0	0
00100	NOT	- - -	1	00000	0	-	0	-	-	-	-	1	001010	0	0	00000	0	0
00110	XOR	- - -	1	00000	0	-	0	-	-	-	-	1	001010	0	0	00000	0	0
00111	SHL	- - -	1	00000	0	0	0	-	-	1	1	1	000000	0	0	00000	0	0
10111	SHR	- - -	1	00000	0	0	0	-	-	1	1	1	000100	0	0	00000	0	0
01000	STORE	- - -	1	00000	0	0	0	-	-	0	1	1	100010	0	0	00000	0	0
01001	LOAD	- - -	1	00000	0	0	0	-	-	1	0	1	100011	0	0	00000	0	0
01010	ADDI	- - -	1	00000	0	0	0	-	0	-	-	0	010—	0	0	00000	0	0
01011	NOTI	- - -	1	00000	0	-	0	-	-	-	-	0	001100	0	0	00000	0	0
01100	ADD	- - -	1	00000	0	-	0	-	0	-	-	1	010—	0	0	00000	0	0
01101	SUB	- - -	1	00000	0	-	0	-	1	-	-	1	010—	0	0	00000	0	0
01110	NOP	- - -	1	00000	0	0	1	-	-	-	-	-	111—	0	0	00000	0	0
10101	MOVE	- - -	1	00000	0	0	0	-	-	1	1	1	110—	0	0	00000	0	0
01111	JMP	- - -	1	00000	0	0	1	-	-	-	-	-	111—	0	1	00000	0	0
10000	JMP2ITSELF	- - -	1	00001	0	0	1	-	-	-	-	-	011—	0	0	00001	0	0
10001	BRAC1	- - -	1	10010	0	1	1	-	-	-	-	-	01—-	0	0	00000	-	0
10010	BRAC2	- - -	1	10100	0	1	-	-	-	-	-	-	———	0	0	——	0	0
10011	FPADD	- - -	1	00000	0	0	0	-	-	1	1	1	0111–	0	0	10110	0	0
10100	MULT	- - -	1	00000	0	0	0	-	-	1	1	1	101111	0	0	11000	0	0
11001	BHI	0 0 0	1	00100	1	0	1	0	0	-	-	-	-11—	0	1	00000	0	0
11010	BHE	0 0 1	1	00100	1	0	1	0	0	-	-	-	-11—	0	1	00000	0	0
11011	BLT	0 1 0	1	00100	1	0	1	0	0	-	-	-	-11—	0	1	00000	0	0
11100	BLE	0 1 1	1	00100	1	0	1	0	0	-	-	-	-11—	0	1	00000	0	0
11001	BEQ	1 0 0	1	00100	1	0	1	0	0	-	-	-	-11—	0	1	00000	0	0
11110	BNE	1 0 1	1	00000	1	0	1	0	0	-	-	-	-11—	0	1	00000	0	0
11000	MULTNOP	1 1 0	1	11000	0	0	1	-	-	-	-	-	101—	0	0	00000	0	0
10110	FPNOP	1 1 1	1	10110	0	0	1	-	-	-	-	-	011—	0	0	00000	0	0

Table 3.5: Final Control Word Table

Instruction Name	Opcode	Action
LOADR0	– 48 – 01	R0 $\leftarrow$ M[01]
LOADR1	– 49 – 02	R1 $\leftarrow$ M[02]
LOADR2	– 4A – 03	R2 $\leftarrow$ M[03]
LOADR3	– 4B – 04	R3 $\leftarrow$ M[04]
LOADR4	– 4C – 05	R4 $\leftarrow$ M[05]
LOADR5	– 4D – 06	R5 $\leftarrow$ M[06]
LOADR6	– 4E – 07	R6 $\leftarrow$ M[07]
LOADR7	– 4F – 08	R7 $\leftarrow$ M[08]
SHL	– 39 -? -?	R1 $\leftarrow$ R2 $<<$ $\times$ 8
BEQ	92 5? ?? 80	PC $\leftarrow$ 4 if z = 1
SHR	– B9 -? -?	R1 $\leftarrow$ R2 $>>$ $\times$ 8
MULT	– A0 -? -?	R0 $\leftarrow$ R2 $\times$ R3
FPADD	– 9A -? -?	R2 $\leftarrow$ R2 + R3
BNE	BF 5? ?? 80	PC $\leftarrow$ 0 if z = 0

Table 3.6: Final Test Program

Chapter 4

Synthesizable VHDL, the Board and the GUI

In this chapter I will discuss synthesizing the design, downloading it to the board[1], and the teching GUI.

4.1 Synthesizable VHDL

Synthesis is the translation of a high level design into a specified hardware. It translates a register transfer level model of hardware (written in a HDL) into an optimized technology-specific gate level implementation. That is to say, the synthesis tool will take the designer's code and remodel it (maintaining the desired logic) in order to optimize it. Because of this it was important that my design be presented well, free of latches and unsupported VHDL constructs so that the optimization by the synthesis tool did not affect the desired logic of the microprocessor.
It was imperative that the design was synthesizable. It is worth a mention that only a small fraction (around 10 per cent) of the VHDL code constructs are synthesizable. Aside from declaration constructs, the only VHDL language constructs used in the design are process, case,if-then-else and concurrent signal assignment. Only code that can be synthesized can be converted by the compiler into a valid net-list of ports, which then can be translated and mapped onto physical hardware. Not only would a synthesizable design be downloadable to the target project board but would prove that an efficient design is in place.

4.1.1 HDL programming styles

There are 3 main styles of HDL programming:

- Behavioral coding
  - no specific hardware details supplied
  - bus sizes, clocks, resets etc. not specified
  - no target technology

[1] The FPGA

  - not synthesizabe

- RTL coding
  - more detailed coding
  - bus sizes, clocks, resets etc defined
  - target technology not supplied
  - is synthesizabe

- Structural Coding
  - more detailed description of how circuit operates
  - bus sizes, clocks, resets etc. defined
  - target technology defined
  - is synthesizabe

My approach to the design was Structural coding. I decided not to make use of any of the modules in the technology library since I thought it would be better to design every component of the microprocessor myself.

4.1.2 Useful Rules For Synthesis

As part of my 4S1 *Integrated Systems Design* module this year, I learned several partitioning rules for Synthesis. A number of these, detailed below, were useful for designing the microprocessor in an efficient way so that the synthesize tool could enhance my design for speed and area. I adhered to them rigourously wherever possible.

- No hierarchy in combinational paths
  - optimization is limited if hierarchy in combinational paths
  - hierarchy boundaries prevent sharing of common terms
  - example: Combinational modules (CROM, CAR etc.) in the Control Unit are seperated from the non-combinational ones

- No 'glue' logic between layers
  - optimization is limited if glue logic exists between layers i.e a path between module1 and module2 consisting of gates
  - glue logic prevents sharing of common terms
  - example: Multiplexers between control Unit and datapath are part of the datapath layer

- Separate designs with different goals

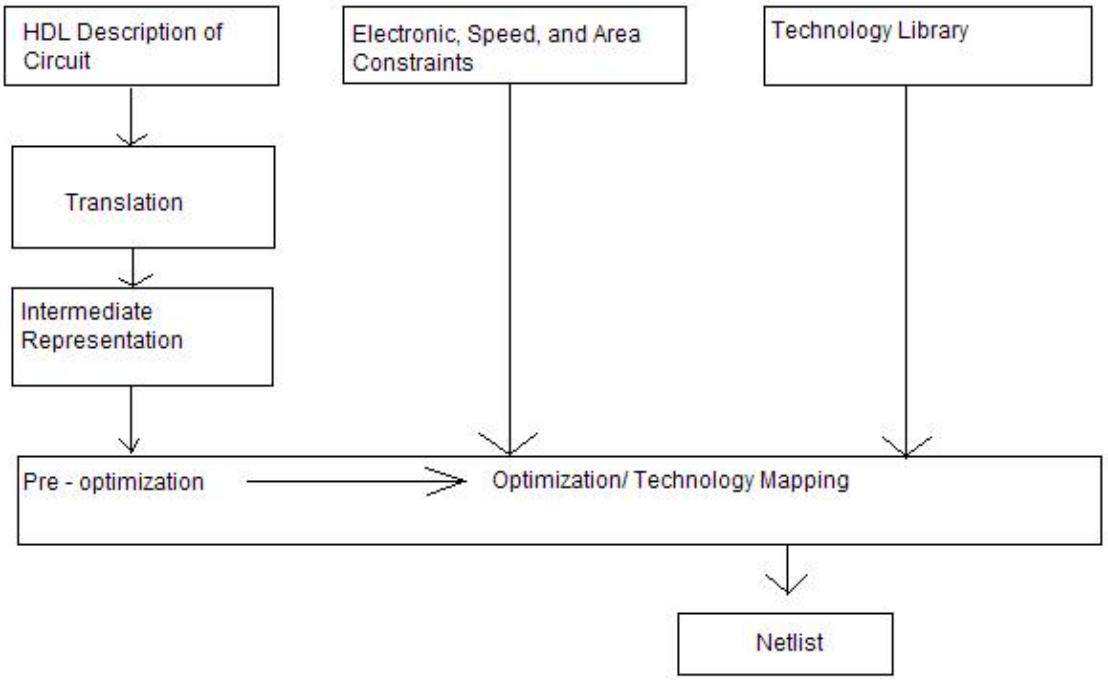

Figure 4.1: High-Level Flow for Synthesis Tool

  - designer should use different modules for different components to partition design into blocks
  - easier for synthesis tool
  - example: All logic operations grouped in one module because they have similar goals
  - example: A different module for the multiplier and floating point unit

- Isolate state machines
  - FSM optimization tool can't be used if non-state machine logic exists in process
  - separating the FSM logic allows the FSM optimization tool for optimal state assignment and area reduction
  - example: The multiplier module

4.2 The Board

The board in question is a Virtex II XC2V1000 Xilinx chip in an FPGA. An FPGA is a configurable application specific integrated circuit that is programmable to perform a certain hardware task. Its main components are

- a regular array of configurable logic blocks (CLBs) that can implement combinational and sequential logic

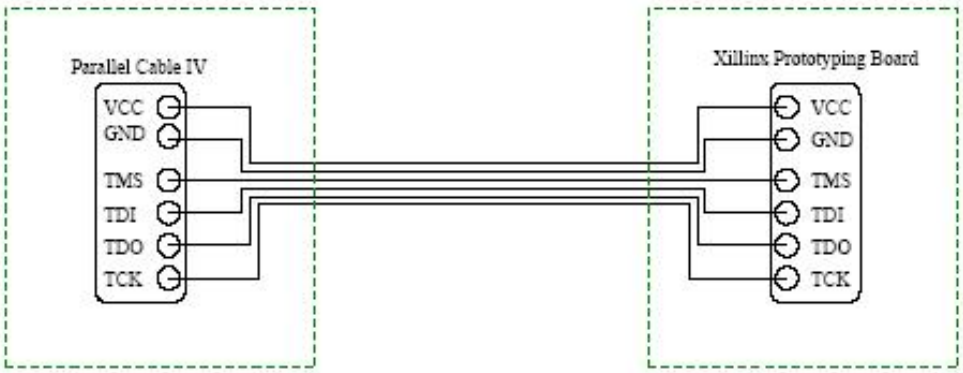

Figure 4.2: Download Cable Interface
[2] taken from Ross Brennan's final year project report

- a matrix of programmable interconnects, that surround the CLBs
- programmable I/O cells surrounding the core

Each CLB consists of a look-up table, MUXes and a flip-flop and can be programmed to implement boolean functions of a few variables. The I/O blocks can be configured to connect to the wiring of the CLB and interconnects.

4.3 Downloading the design to the Board

I used Xilinx Impact to generate the bit file that would be downloaded to the board (Figure1.1). It was configured in Serial slave mode and was downloaded to the board via a parallel cable.

4.4 Problems with testing the Board

The RESET button[2] on the board was broken so I had to come up with another method for displaying the results. I designed a reset counter module which counted up with every clock pulse. When the count got to 2, the reset button went high[3] and after a huge number of clock pulses it reset the count and set reset low.
However, the LEDs on the board appeared not to be lighting up. After I ran a test program on it I discovered that the reason for this was that the clock pulse was too fast. I wrote a clock divider module that would take the clock as an input and after a sufficient count had been accumulated, it

[2] effectively the Start button
[3] reset was active low i.e the microprocessor was reset when the reset signal went low

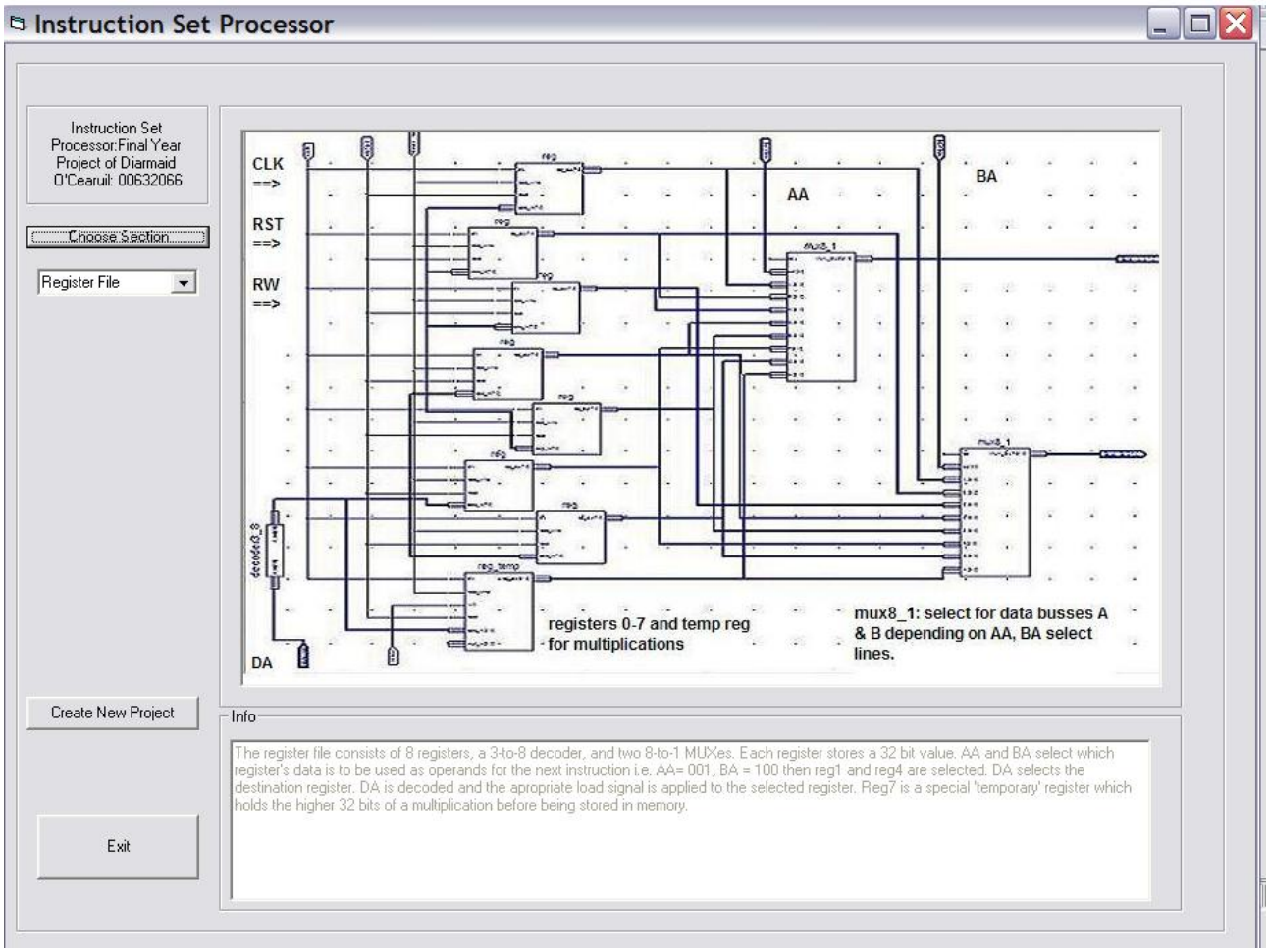

Figure 4.3: Tutorial part of GUI

would output a new clock to the system. However this did not solve the problem and due to timing constraints I was unable to come up with an adequate solution.

4.5 The GUI

The GUI has two sections. The first part of the GUI provides the user with a list of components from the microprocessor. When a component is selected i.e. the register file, a schematic of the layer is displayed, detailing the components and that make up the unit and the connections between them. At the bottom there is a text box that gives a deatailed explanation of the register file and how it all works. See figure 4.3.

The second section (figure 4.4) allows the user to make their own VHDL microprocessor project. Two windows in the GUI allow the user to choose the target folder, where the project is located, and destination folder where the new project will be stored. On the bottom right-hand side of the GUI there a total of nine options for which component(s) to have absent from the new project folder. When the user has selected the components to be absent and clicks the 'Create New Project' button the new project is created. Since the program is filtering through a large number of files (more than 2500) this takes a few moments.

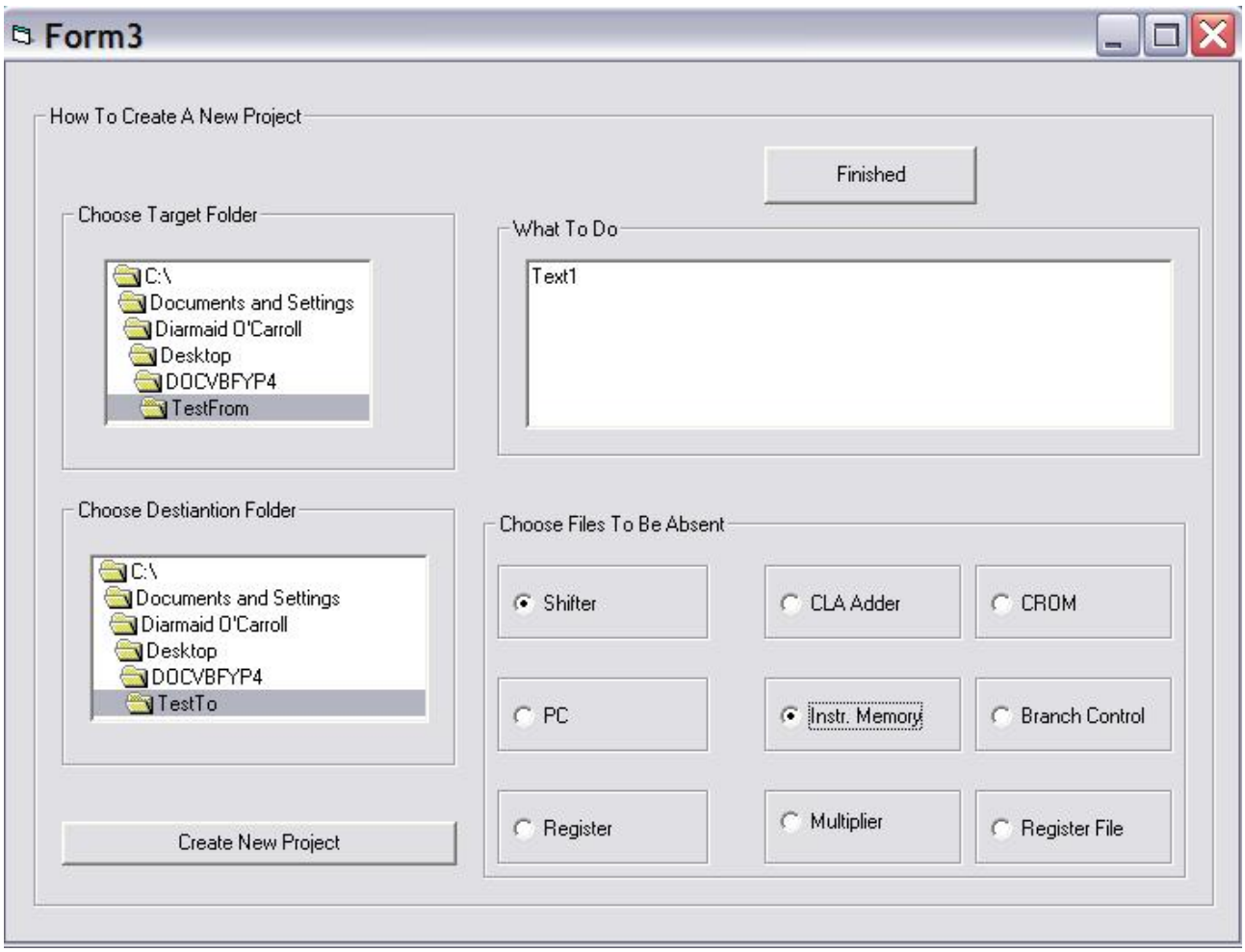

Figure 4.4: GUI: Creating a new project file

Figure 4.5: Warnings when opening modified project

4.5.1 Using the GUI

Here we will step through the process of a lecturer creating a new project file for the students. The lecturer will open up the GUI and proceed to the second part and choose to remove the following components from the project: The instruction Memory, the branch control unit, and the barrel shifter. Then he will click the *Create a New Project* button and when the creation is finished will assign the job of designing the components to the students. A student will open the new project file. Figure 4.5 shows the warnings that will appear as the project is opened.

The three selected components absent from the project are highlighted in the *Sources In Project* plane by red question marks. When the student clicks on one of the these, like in figure 4.6, marks a window appears inviting them to create a new component where the old one was. If it the component in question was the branch control unit, like in figure 4.6 the student would have to open up the 'control unit' layer and look at the declaration of the inputs and outputs of the branch control unit[4] in order to start designing the new component. Then the student would have to appreciate the other components that make up the control unit layer so that they can efficiently design the branch control unit.

When finished designing the component, a testbench waveform is already available for them to test their design. This test bench consists of input and output vectors and when matched with a fully functional branch control unit will display the results with no warnings. This way the student can design and test the missing components and be sure that their designs are working and efficient.

[4]the inputs and outputs are declared here because the control unit layer instantiates the branch control unit

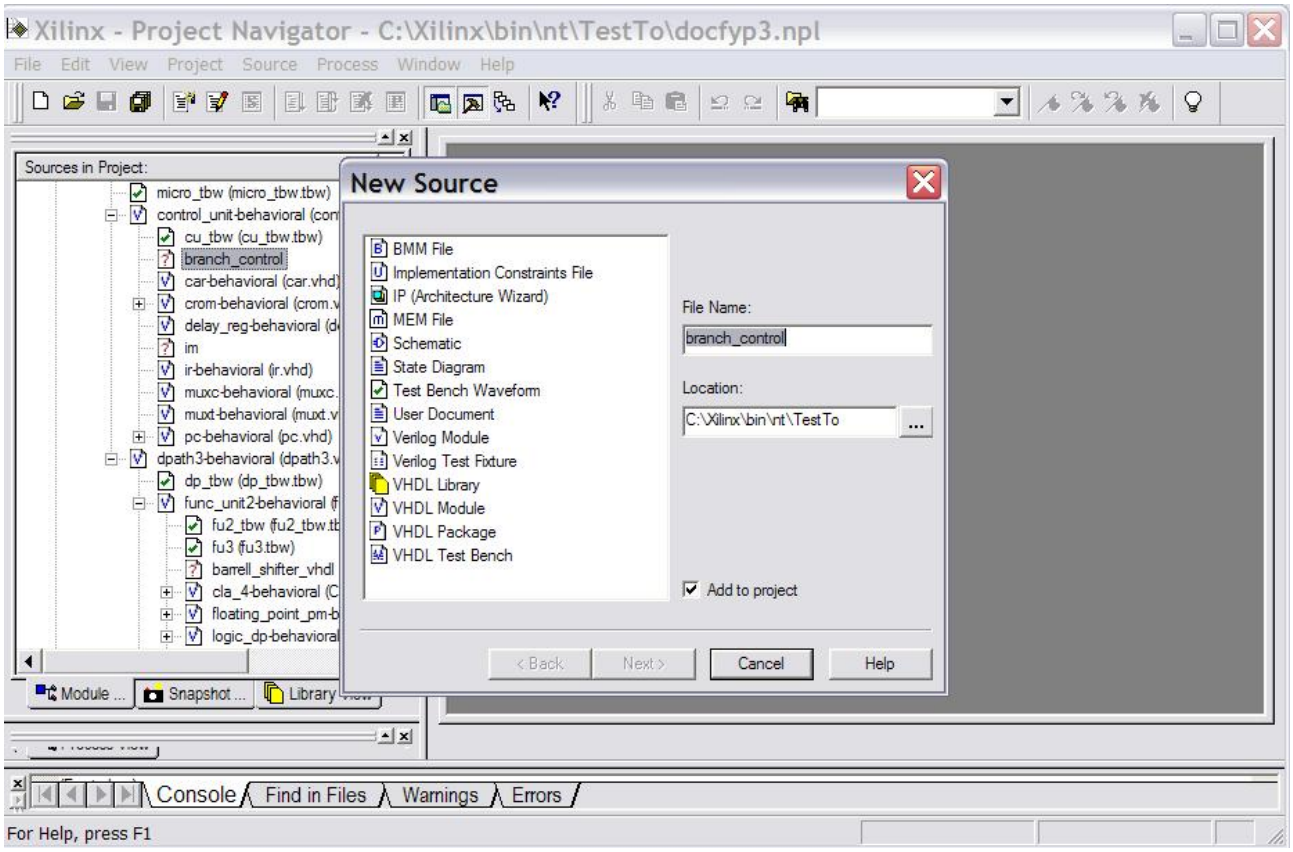

Figure 4.6: Modified Project viewed through Project Navigator

Chapter 5

Results, Conclusions and Future Work

5.1 Results

The results of the project shows that a simulated complex instruction-set processor could be represented in VHDL and synthesized with an XST VHDL synthesizer.
The project, with a little further work done to it, could be used as a teaching tool in a few areas:

- for the 2BA4 *Microprocessor Systems* project. This project goes into detail on the internal components of a microprocessor and would be ideal for students to see how each of those components work and how they interact with each other. The project could be modified to hook up the Virtex chip on the FPGA to other components such as RAM and EEPROM, to provide a better indication of how the microprocessor interacts with its surrounding components. The GUI also provides another avenue for the students to learn about microprocessor design.
- for the 1BA3 module. Students learn how to program assembly language and conditional branches are an integral part of it. This project could be used to teach how the branching is carried out in hardware and help the students gain a deeper understanding of what they are programming.
- for the 1BA4 module. In the *Digital Logic Design* module the students learn the basics of digital logic from logic tutors. These tutors are huge cases containing gates and flip-flops. They are quite awkward and have a limited size. This project, or a similar project could be tailored to be a teaching tool for those students.

5.2 Conclusions

As well as proving useful for teaching needs, the project demonstrates the usefulness of Hardware Description Languages and FPGAs, not only from a design and synthesis perspective but also as teaching tools. VHDL is already used in the *Computer Architecture* I module and in the 3BA5 module. It could easily be integrated into the 1BA3, 1BA4 and 3BA4 modules.
The project also shows why FPGAs are gaining popularity in industry. In a matter of minutes a

simulated design can be placed on a board and then fully tested. For a full-custom[1] or semi-custom design[2], when the design is finished and thoroughly tested the design is sent off to a factory to made into hardware. This is expensive in terms of time (up to 8 weeks) and money, especially if a fault is found on the hardware chip after. In the case of an FPGA, the design can be tested on-board and if any errors arise then the design can be debugged and replaced on the FPGA.

5.3 Future Work

More complex instructions could be made if more ALU units like an integer division unit was designed and added. Three other final year projects have been done this year that could be integrated including an optimised floating point adder, an optimised floating point multiplier and a project that implements Tomosulo's method.
The GUI could be expanded to include the subcomponents of each layer in the design and any other new layers. A different GUI could be implemented as a teaching tool for the control words, allowing the students to construct their own control words and testing them out on the system.

[1] A design where every single component is designed by the designers in order to maximise speed and area
[2] A design consisting mostly of 'primitive' components from a standard library

Bibliography

[1] Michael Manzke & Ross Brennan. On the Introduction of Reconfigurable Hardware into Computer Architecture Education. http://www4.ncsu.edu/ efg/wcae/2003/submissions/manzke.pdf.

[2] Ross Brennan. Design & Evaluation of an FPGA based Microprocessor Board. http://www.cs.tcd.ie/Michael.Manzke/fyp2002-2003/ross_brennan_report.pdf.

[3] Colm Brewer. Final Year Project: Design of an Optimised Floating Point Adder, 2005.

[4] Edsko deVries. Professional Typesetting Using Latex. http://www.edsko.net.

[5] Shonagh Hurley. Final Year Project: Implementation of Tommasulo's Method, 2005.

[6] Mike Johnson. An Alternative to RISC: The Intel 80x86, 2003. http://www.Xilinx.com.

[7] P.R.M Oliver & N. Kantaris. *Using Visual Basic - 2nd Edition Updated*. Babani Computer Books, 2003.

[8] M. Morris Mano & Charles R. Kyme. *Logic and Computer Design Fundamentals - 2nd Edition Updated*. Prentice Hall, 2001.

[9] Ms. S. Launders. 4S1 Integrated Systems Design Notes.

[10] Michael Manzke. Computer Architecture and Microprocessor Systems - 2ba4 Notes. http://www.cs.tcd.ie/Michael.Manzke/2ba4.html.

[11] Michael Manzke. Computer Engineering- 3ba5 Notes. http://www.cs.tcd.ie/Michael.Manzke/3ba5.html.

[12] John McCarthy. 4S1 Integrated Systems Design Notes.

[13] Mark McTaggart. Final Year Project: Design of an Optimised Floating Point Multiplier, 2005.

[14] Ollie Pugh. 3BA5 Floating Point Adder assignment part 1, 2004.

[15] Laura Redmond. Final Year Project: Design of a Teachng Instruction Set Processor, 2004. http://www.cs.tcd.ie/Michael.Manzke/fyp2003-2004/laura_redmond_report.pdf.

[16] Xilinx. Virtex II Prototype Platform Users Guide, 2001. http://www.Xilinx.com.

Appendix A

Control Word Table

Bit(s)	Field name	Signal	Function
0	IL	0	Disable Instruction Register
		1	Enable Instruction Register
1	MC	0	Select NASEQ in MUXC
		1	Select opcode in MUXC (to execute next instruction in memory)
6 - 2	NASEQ	XXXXX	The address of the next instruction to be executed immediately
7	PL	0	Hold PC
		1	Load PC
8	PI	0	Hold PC
		1	Increment PC
14 - 9	Control Instructions	0000XX	Shift Left
		0001XX	Shift Right
		001X00	AND
		001X01	OR
		001X10	XOR
		001X11	NOT
		010XXX	CLA ADD
		0111XX	Start FP Add
		0110XX	FP Add
		100X0X	Output HiZ
		100X10	Store
		100X11	Load
		101XXX	Multiply and Enable temp register
		110XXX	Move
15	MB	0	MuxB select: select immediate value
		1	MuxB select: select value register file
16	MR	X	No Function
17	MW	X	No Function
18	CIN	0	Add in CLA adder
		0	Subtract in CLA adder
19	Branch Reset	0	Do not reset branch control unit
		1	Reset Branch control unit
20	RW	0	Enable write to register file
		1	Disable write to register file
21	DC	X	No function
22	Delay	0	Do not reset Branch control unit after one clock cycle
		1	Reset Branch control unit after one clock cycle
27 - 23	NABRA	XXXXX	Address of the next instruction to be executed if branch taken
28	Enable	0	Read from external chip(micro-processor disabled)
		1	Do not read from external chip (micro-processor enabled)
31 - 29	Branch Instruction	000	BHI
		001	BHE
		010	BLT
		011	BNE
		100	BEQ
		101	BNE
		110	Wait for Mult Done
		111	Wait for FP Done

Table A.1: Control Word Fields

Appendix B

Terminology

- *Register*: A kind of placeholder for data
- *Flip-Flop*: Basic element of memory storage
- *Bus*: A shared transfer path between registers, driven by selection logic
- *Port*: A pin on an IC
- *Port Map*: A VHDL statement that instantiates components
- *Schematic*: A feature of VHDL where components can be connected through schematics
- *Bit file*: The file that is downloaded to the FPGA, representing the design
- *Controlword*: A 32 bit signal, sent by the CROM to other coponents, that indicate what instruction is to be perform
- *ALU*: Arithmetic/Logic Unit: the hardware that performs the operations
- *Register File*: The layer containing the registers
- *Datapath*: The layer that connects the ALU and Register File
- *Control Unit*: The layer that determines and then sends the control signals to the datapath
- *Testbench Waveform*: A waveform that takes in the input vectors of a component and outputs the simulated results
- *Opcode*: The part of the instruction that tells the CROM the desired controlword
- *Gate Delay*: The delay associated with signals propagating through gates