by Zhyang Ong zhiyang@ieee.org and Andrew Mattheisen amattheisen@gmail.com

EE577B Troy Processor Project Report

TABLE OF CONTENTS

1 Introduction 4
1.1 Arithmetic Logic Unit (ALU) Module 5
1.1.1 Inputs and Outputs 5
1.1.2 Functional Test Results 5
1.1.3 Synthesis Results 5
1.1.4 Post-Synthesis Functional Test Results 5
1.2 Control Module 5
1.2.1 Inputs and Outputs 5
1.2.2 Functional Test Results 6
1.2.3 Synthesis Results 7
1.2.4 Post-Synthesis Functional Test Results 7
1.3 Data Memory Module (provided – see constraints above) 8
1.3.1 Inputs and Outputs 8
1.3.2 Functional Test Results 8
1.3.3 Synthesis Results 8
1.3.4 Post-Synthesis Functional Test Results 8
1.4 Instruction Memory Module (provided – see constraints above) 8
1.4.1 Inputs and Outputs 8
1.4.2 Functional Test Results 9
1.4.3 Synthesis Results 9
1.4.4 Post-Synthesis Functional Test Results 9
1.5 Program Counter Module 9
1.5.1 Inputs and Outputs 9
1.5.2 Functional Test Results 9
1.5.3 Synthesis Results 9
1.5.4 Post-Synthesis Functional Test Results 10
1.6 Register File Module 10
1.6.1 Inputs and Outputs 10
1.6.2 Functional Test Results 11
1.6.3 Synthesis Results 12
1.6.4 Post-Synthesis Functional Test Results 12
1.7 Mux Module 14
1.7.1 Inputs and Outputs 14
1.7.2 Functional Test Results 14
2 CPU Pipeline 15
2.1 CPU Module 22
2.1.1 Inputs and Outputs 22
2.1.2 Functional Test Results 23
2.1.3 Synthesis Results 25
2.1.4 Post-Synthesis Functional Test Results 25
3 Instruction set implementation 26
4 Adder/Subtractor design 38
4.1 Functional Test Results 38
4.2 Synthesis Results 39
4.3 Post-Synthesis Functional Test Results 39

5 Shifter Design 39
5.1 Functional Test Results 39
5.2 Synthesis Results 39
5.3 Post-Synthesis Functional Test Results 39
6 Multiplier Design 39
6.1 Functional Test Results 39
6.2 Synthesis Results 39
6.3 Post-Synthesis Functional Test Results 40
7 Hazard management and data forwarding 41
7.1 Functional Test Results 45
8 Optimization 46
9 Synthesis Report 46
10 Final specifications of your designs 46
11 Possible Future Enhancements 46
12 Conclusion 47
13 References 48

1 Introduction

This report documents the process of designing and testing a 128 bit microprocessor to implement a limited set of instructions. The goal of this project is to create a microprocessor and maximize its speed. The design process is outlined below:

- Develop a conceptual design
- Divide the design into modules
- Implement each module with Register Transfer Logic (RTL) style Verilog Hardware Description Language (HDL)
- Test each module functionally using NcVerilog
- Synthesize all modules developed using Synopsys Design Compiler
- Verify each module synthesizes correctly via post synthesis functional testing
- Integrate all modules and test functionality
- Synthesize the Central Processing Unit (CPU) module
- Verify the top level design synthesized correctly via post synthesis functional testing

Note: The process described above must be iterated to develop an optimal solution.

The instruction following instruction set has been implemented:

- Wide word addition (WADD)
- Wide word AND (WAND)
- Wide word load from data memory using immediate addressing (WLD)
- Wide word signed multiplication of the even bytes or double-bytes (a double-byte is 16 bits) (WMULES)
- Wide word signed multiplication of the odd bytes or double-bytes (WMULOS)
- Wide word unsigned multiplication of the even bytes or double-bytes (WMULEU)
- Wide word unsigned multiplication of the odd bytes or double-bytes (WMULOU)
- Wide word move (WMV)
- Wide word NOT (WNOT)
- Wide word OR (WOR)
- Wide word byte permute (WPRM)
- Wide word shift logical left (WSLL)
- Wide word shift logical left immediate (WSLLI)
- Wide word shift arithmetic right (WSRA)
- Wide word shift arithmetic right immediate (WSRAI)
- Wide word shift logical right (WSRL)
- Wide word shift logical right immediate (WSRLI)
- Wide word store to data memory using immediate addressing (WST)
- Wide word subtraction (WSUB)
- Wide word XOR (WXOR)

Note: branch and jump commands were not implemented - this greatly simplifies the design.

The design of the microprocessor will be constrained by the following:

- The design must interface to a predefined Instruction Memory RTL style Verilog HDL module (this module is included in section xx).

- The design must interface to a predefined Data Memory RTL style Verilog HDL module (this module is included in section xx).
- Top level functional testing must be accomplished using a predefined Verilog HDL test bench (this module is included in section xx).
- The multiplier will not use the "*" symbol available in RTL style Verilog HDL (this is because the synopsis design compiler available at USC uses a sub-optimal multiplier when it encounters the "*" symbol in RTL style Verilog HDL).

Benchmarking is a common practice of measuring speed in the industry. Benchmarking will be used to measure the speed of the processor developed. Several benchmarks will be provided to test our microprocessor. We will measure the execution time the processor requires to run each benchmark program and report the results.

The basic modules used to construct the microprocessor are described in the following subsections.

1.1 Arithmetic Logic Unit (ALU) Module

1.1.1 Inputs and Outputs

Inputs: [0:4] aluop, [0:1] ww, [0:127] reg_a, [0:127] reg_b
Outputs: [0:127] result

1.1.2 Functional Test Results

The ALU is composed of addition (see section 3), shift (see section 4), and multiplication (see section 5) logic.

1.1.3 Synthesis Results

The ALU is composed of addition (see section 3), shift (see section 4), and multiplication (see section 5) logic.

1.1.4 Post-Synthesis Functional Test Results

The ALU is composed of addition (see section 3), shift (see section 4), and multiplication (see section 5) logic.

1.2 Control Module

1.2.1 Inputs and Outputs

Inputs: [0:31] instruction
Outputs: [0:4] aluop, [0:4] rrdaddra, [0:4] rrdaddrb, [0:4] rwraddrd, [0:2] regop, [0:1] ww, [0:20] maddr, memEn, memWrEn, [0:15] wbyteen, [0:127] immediate, (reginmuxop, aluinmuxop will also be included to control multiplexers in the pipeline as described in the next section)

1.2.2 Functional Test Results

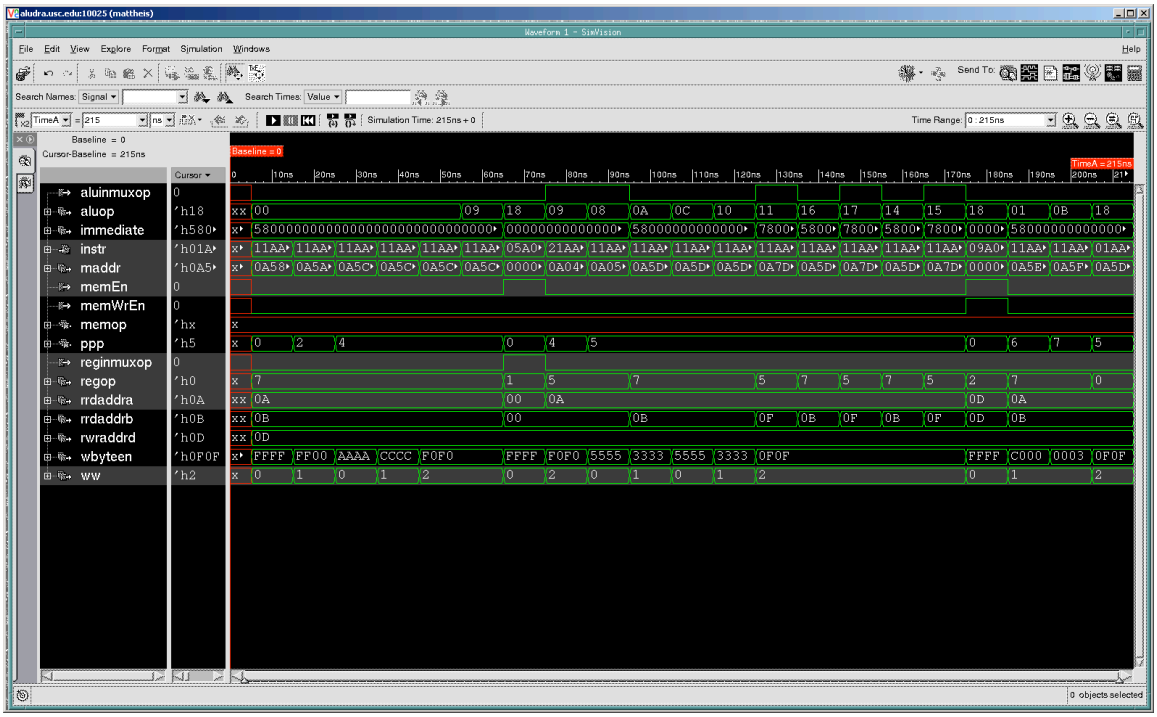
Functional testing of the control logic was performed to ensure that the control logic was generating the expected signals for a given input stream of instructions. The input stream of instructions tested *each possible instruction at the time of implementation.

> *The multiplication instructions had not implemented before the original Functional testing of the control logic, and therefore are not tested at the module level.

The following instructions were tested and the output of the control module was carefully scrutinized to the internal logic was correct. As can be seen in the waveforms below, the control logic is very important – it provides signals to all other modules to control their operation.

The following instructions were checked:

```
32'b00010001101010100101100000000000; // ADD
32'b00010001101010100101101001000000; // ADD
32'b00010001101010100101110000000000; // ADD
32'b00010001101010100101110001000000; // ADD
32'b00010001101010100101110010000000; // ADD
32'b00010001101010100101110010001001; // AND
32'b00000101101000000000000000000001; // LD
32'b00100001101010100000010010001001; // VMV
32'b00010001101010100000010100001000; // NOT
32'b00010001101010100101110101001010; // OR
32'b00010001101010100101110100001100; // PRM
32'b00010001101010100101110101010000; // SLL
32'b00010001101010100111110110010001; // SLLI
32'b00010001101010100101110110010110; // SRA
32'b00010001101010100111110110010111; // SRAI
32'b00010001101010100101110110010100; // SRL
32'b00010001101010100111110110010101; // SRLI
32'b00001001101000000000000000000001; // ST
32'b00010001101010100101111001000001; // SUB
32'b00010001101010100101111101001011; // XOR
32'b00000001101010100101110110001011; // NOP
```



1.2.3 Synthesis Results

The following results were found in the control.area report:

Number of ports:	226
Number of nets:	129
Number of cells:	98
Number of references:	9
Combinational area:	2548.000000
Noncombinational area:	0.000000
Total cell area:	2548.000000

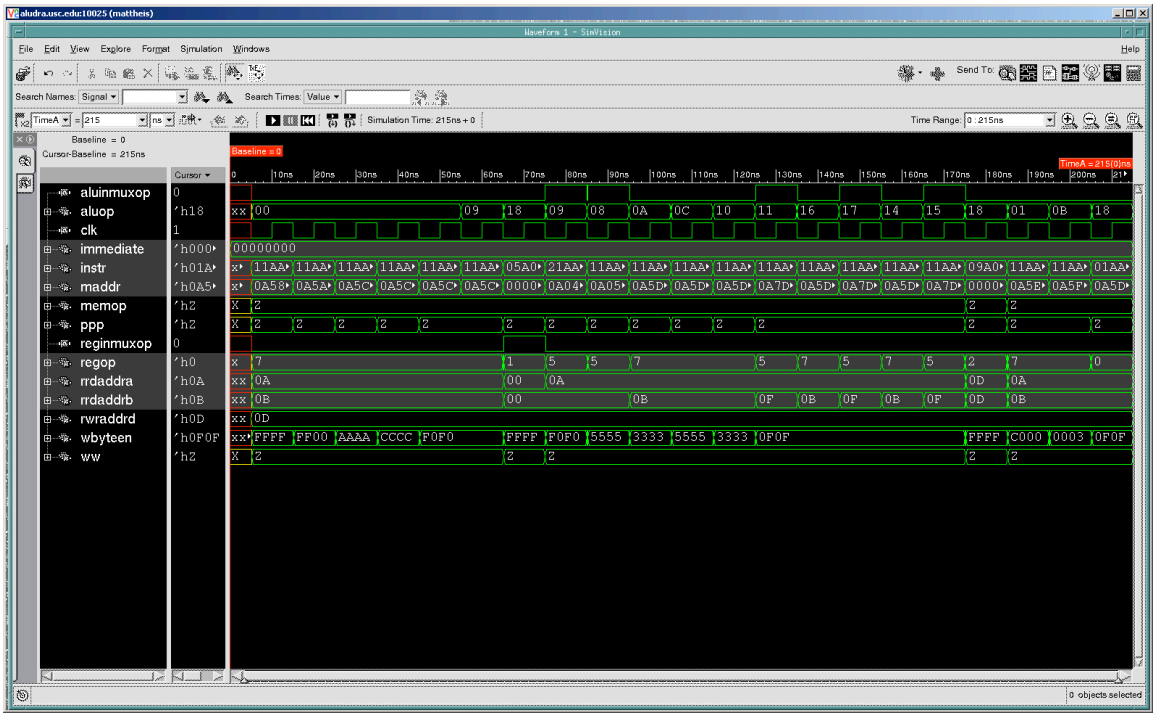
The following timing information was extracted from the timing report:

Max delay:	0.95 delay units.

The check_design report indicated that some input nets were not used and that some output nets were shorted together, or shorted to logic 0. We looked into these and found that all warnings were expected.

1.2.4 Post-Synthesis Functional Test Results

Using the same instruction set, the same results were obtained from the netlist generated by the synthesis tool (see below).



1.3 Data Memory Module (provided – see constraints above)

1.3.1 Inputs and Outputs

Inputs: clk, wren, memEn, [0:7] memAddr, [0:127] dataIn
Outputs: [0:127] dataOut

1.3.2 Functional Test Results

There was no need to test the data memory module individually since it was provided to us and we were unable to alter it.

1.3.3 Synthesis Results

The data memory module was unsynthesizeable, as advertized when it was provided to us, therefore we did not attempt to synthesize it.

1.3.4 Post-Synthesis Functional Test Results

Since no synthesis was performed, no post synthesis functional testing was performed either.

1.4 Instruction Memory Module (provided – see constraints above)

1.4.1 Inputs and Outputs

Inputs: [0:7] addr
Outputs: [0:31]instruction

1.4.2 Functional Test Results

There was no need to test the instruction memory module individually since it was provided to us and we were unable to alter it.

1.4.3 Synthesis Results

The instruction memory module was unsynthesizeable, as advertized when it was provided to us, therefore we did not attempt to synthesize it.

1.4.4 Post-Synthesis Functional Test Results

Since no synthesis was performed, no post synthesis functional testing was performed either.

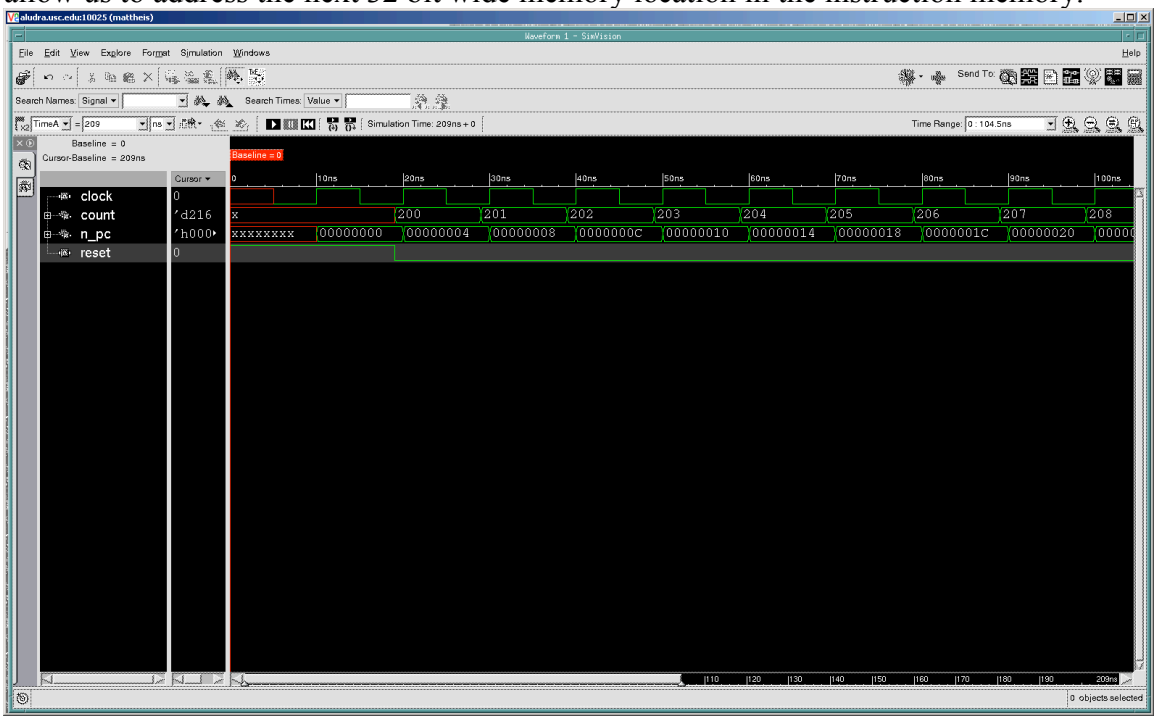
1.5 Program Counter Module

1.5.1 Inputs and Outputs

Inputs: clk and reset
Outputs: [0:20] program_counter

1.5.2 Functional Test Results

The following waveform was generated by the program counter. As you can see, the value of the program counter increments by 4 each positive clock edge – which will allow us to address the next 32 bit wide memory location in the instruction memory.



1.5.3 Synthesis Results

The following results were found in the control.area report:

Number of ports: 34

Number of nets:	102
Number of cells:	66
Number of references:	4
Combinational area:	3576.000000
Noncombinational area:	3072.000000
Total cell area:	6648.000000

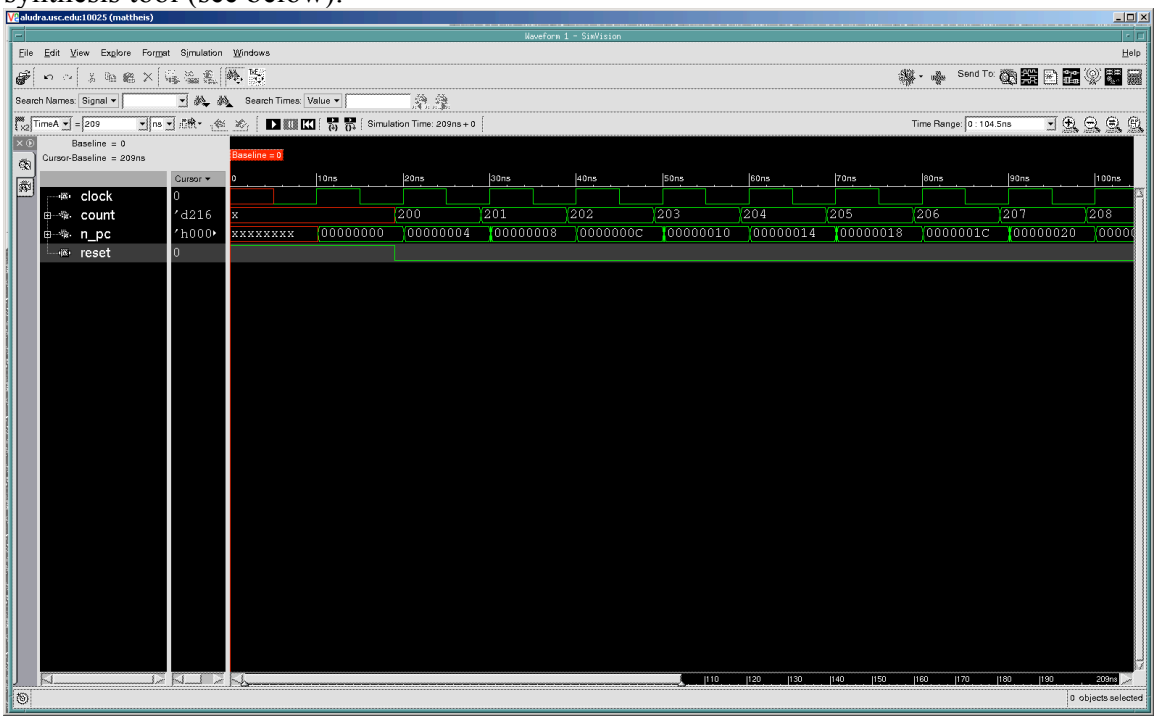
The following timing information was extracted from the timing report:

Max delay:	5.0 delay units.

The check_design report indicated that one of the synthesized cells did not drive any nets. We kept this in mind while testing the functionality of the synthesized code and found that the warning could be ignored (see next section).

1.5.4 Post-Synthesis Functional Test Results

Using the same inputs, the same results were obtained from the netlist generated by the synthesis tool (see below).



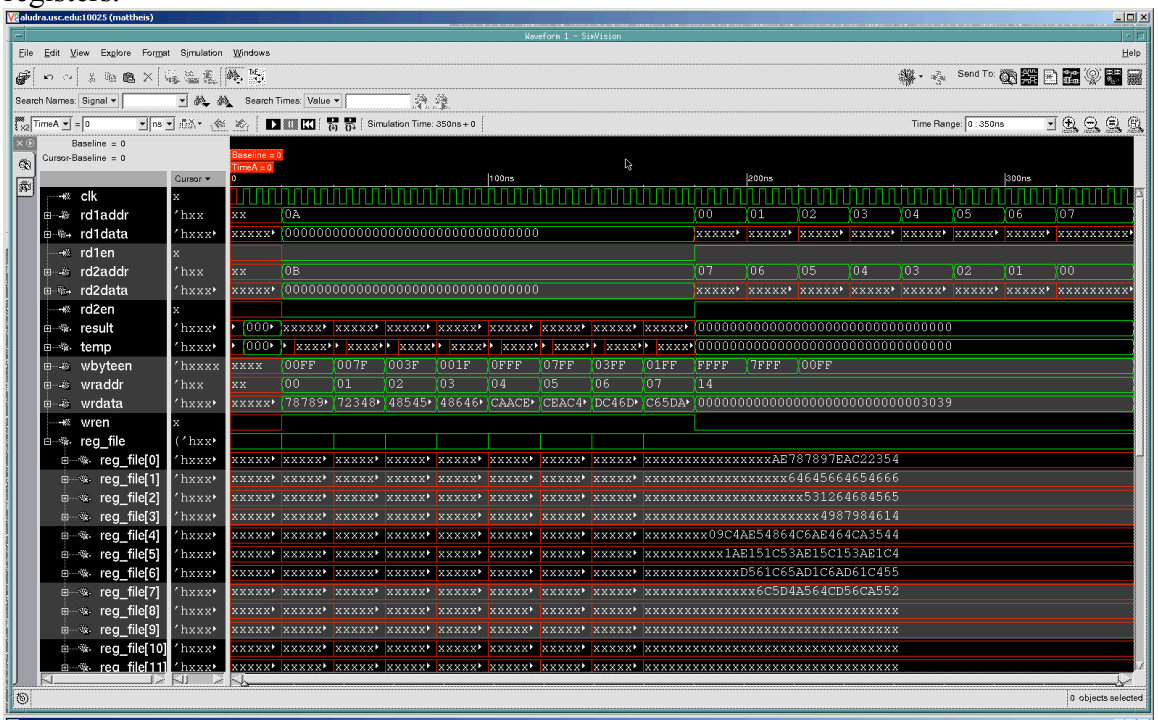
1.6 Register File Module

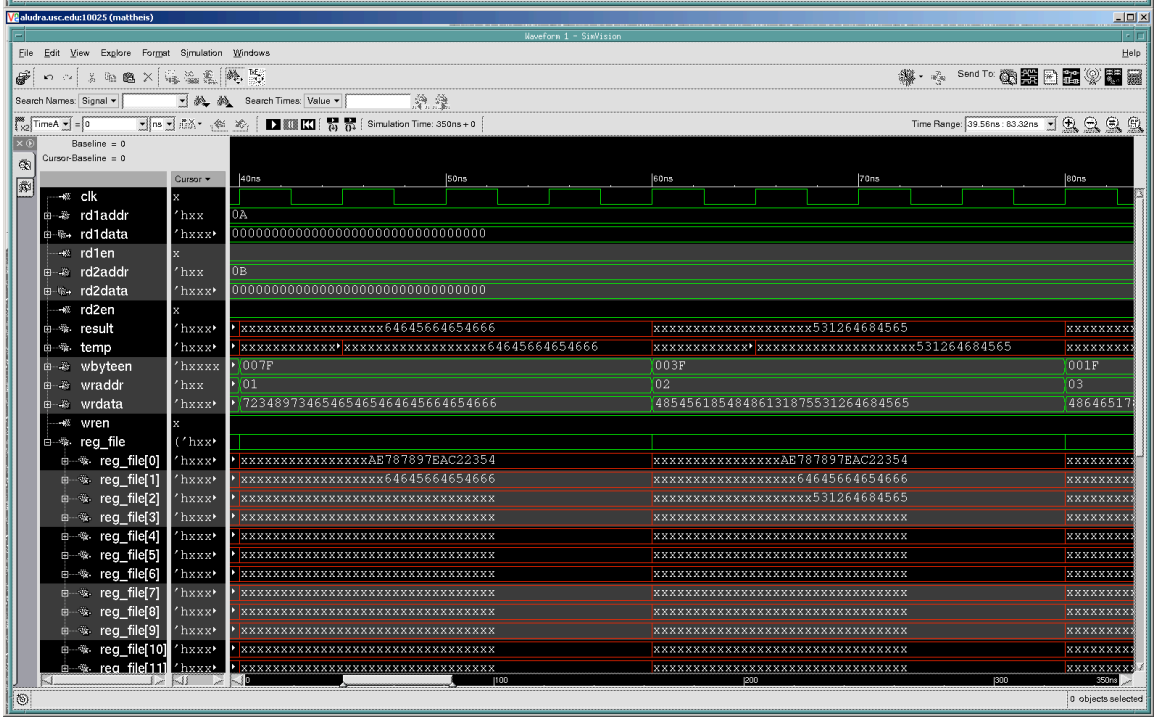
1.6.1 Inputs and Outputs

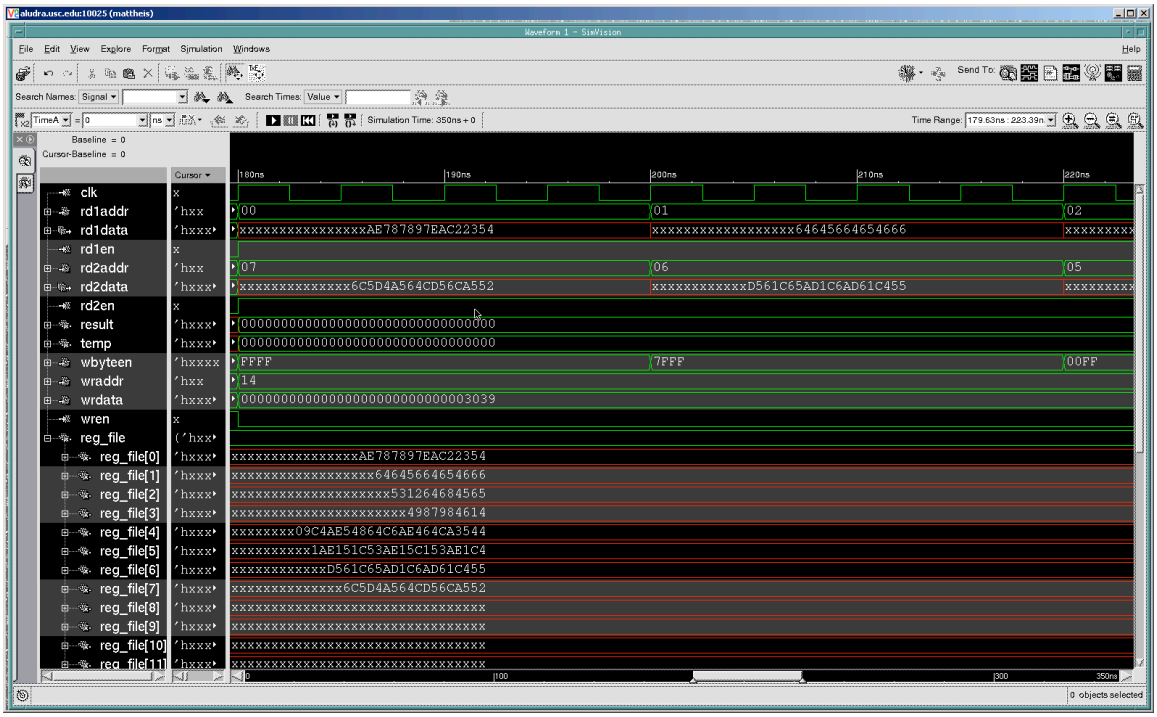
Inputs: clk, [0:127] wrdata, wren, rd1en, rd2en, [0:4] wraddr, [0:4] rd1addr, [0:4] rd2addr, [0:15] wbyteen
Outputs: [0:127] rd1data, [0:127] rd2data

1.6.2 Functional Test Results

It is difficult to fit the results into a single screen shot, but below are a few screenshots showing that several registers are written to then later the correct values are read from the registers.







1.6.3 Synthesis Results

The following results were found in the control.area report:

Number of ports:	24
Number of nets:	421
Number of cells:	405
Number of references:	10
Combinational area:	9600.000000
Noncombinational area:	6912.000000
Total cell area:	16512.000000

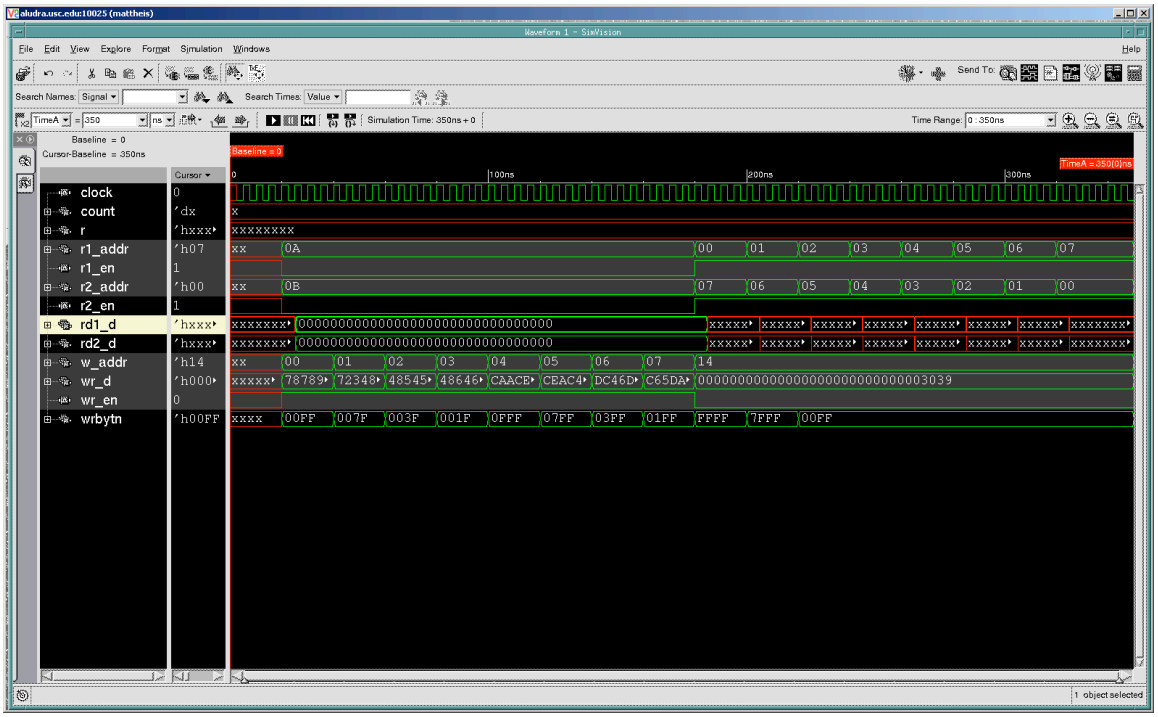
The following timing information was extracted from the timing report:

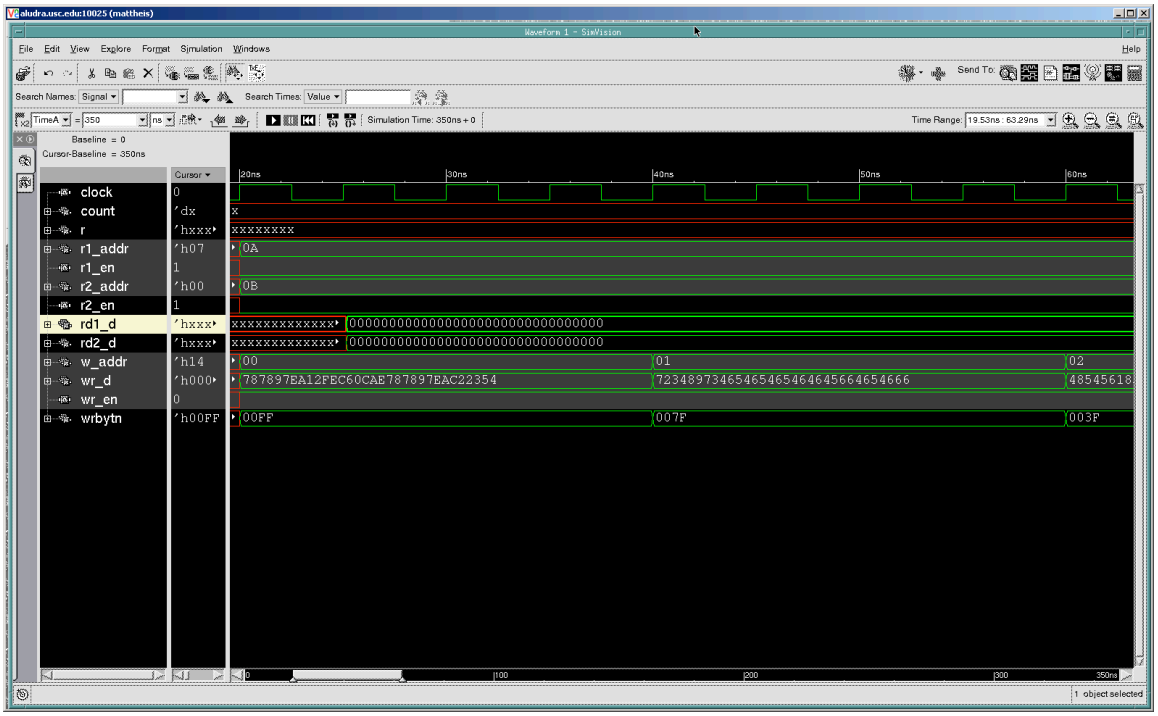
Max delay:	2.25 delay units.

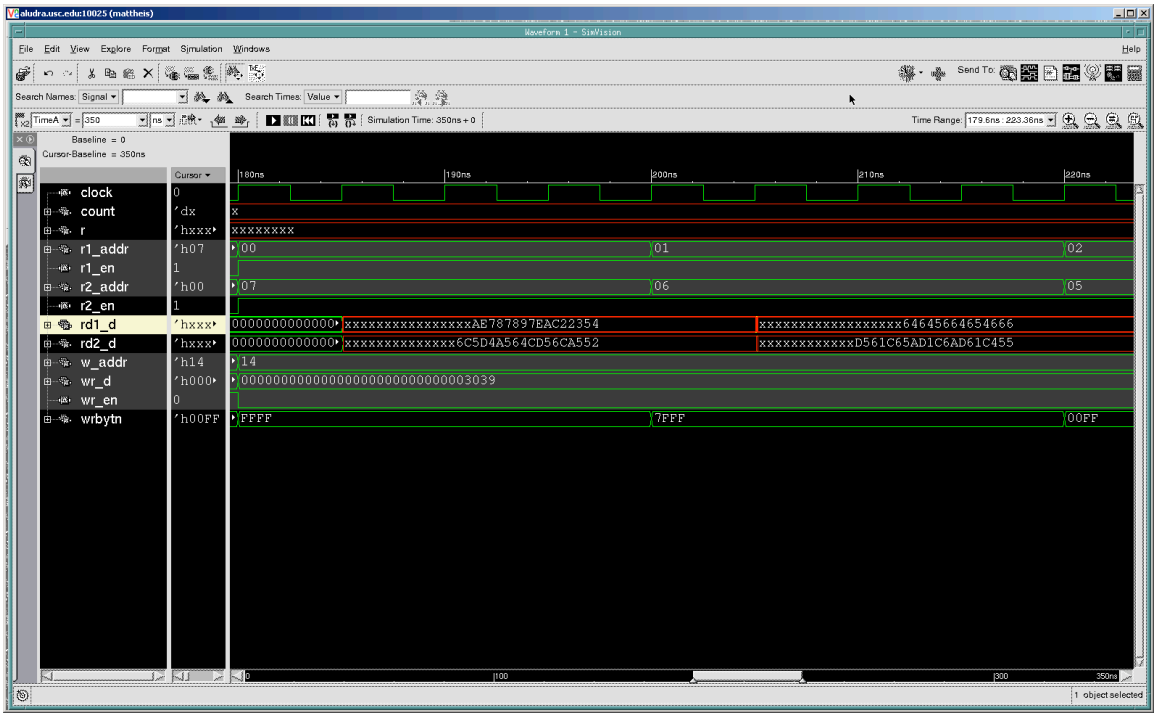
The check_design report was empty, indicating no errors and no warnings.

1.6.4 Post-Synthesis Functional Test Results

Using the same inputs, the same results were obtained from the netlist generated by the synthesis tool (see below).


aludra.usc.edu:10025 (mattheis)
Waveform 1 - SimVision
File Edit View Explore Format Simulation Windows Help
clock
count
r
r1_addr
r1_en
r2_addr
r2_en
rd1_d
rd2_d
w_addr
wr_d
wr_en
wrbytn
1 object selected


aludra.usc.edu:10025 (mattheis)
Waveform 1 - SimVision
File Edit View Explore Format Simulation Windows Help
clock
count
r
r1_addr
r1_en
r2_addr
r2_en
rd1_d
rd2_d
w_addr
wr_d
wr_en
wrbytn
1 object selected



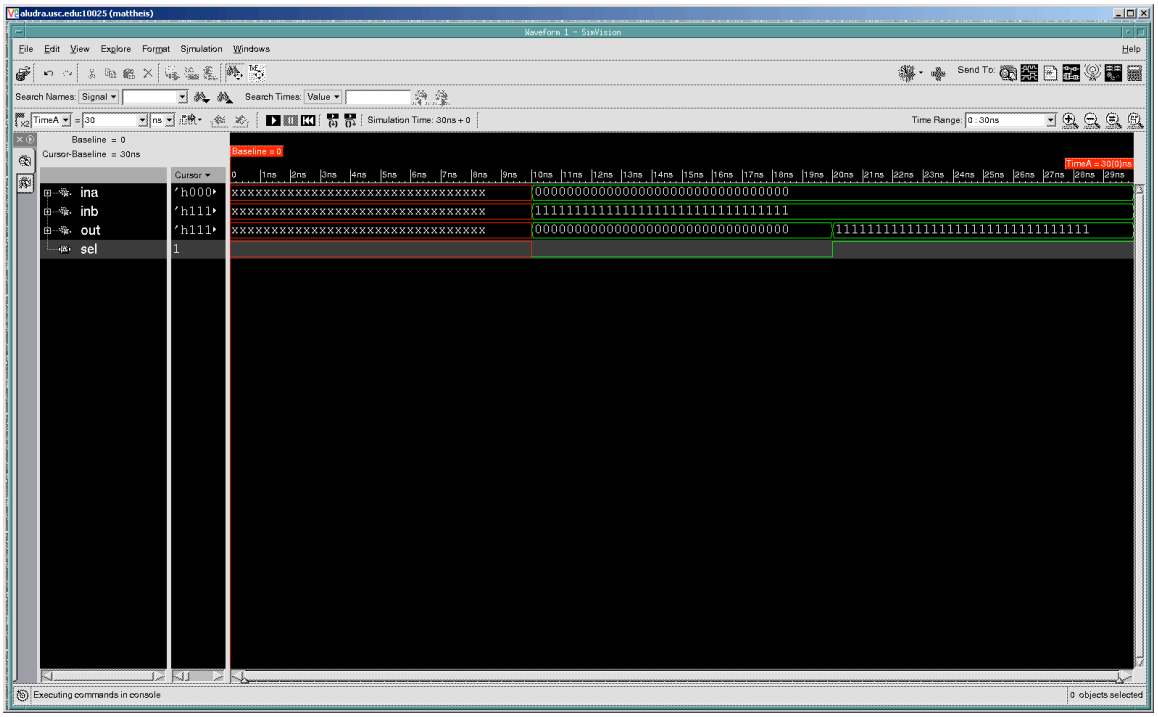
1.7 Mux Module

1.7.1 Inputs and Outputs

Inputs: [0:127] dataIn1, [0:127] dataIn2, select
Outputs: [0:127] dataOut

1.7.2 Functional Test Results

As can be seen below, the mux functions as expected.



Due to the simplicity of the module (out<=sel?inb:ina;), synthesis and post synthesis testing were not required.

2 CPU Pipeline

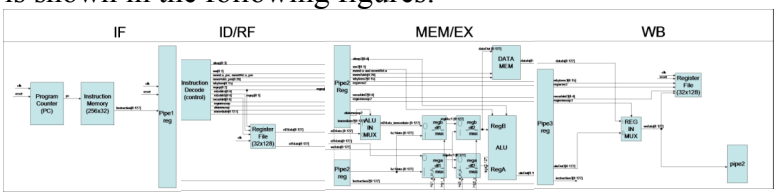
Minimizing execution time is the goal of this project. Therefore a pipeline was implemented to increase the throughput. The benchmark instruction set is expected to be long, so the small additional latency should be overshadowed by the increase in throughput. Traditionally, processors are pipelined into the following 5 stages:

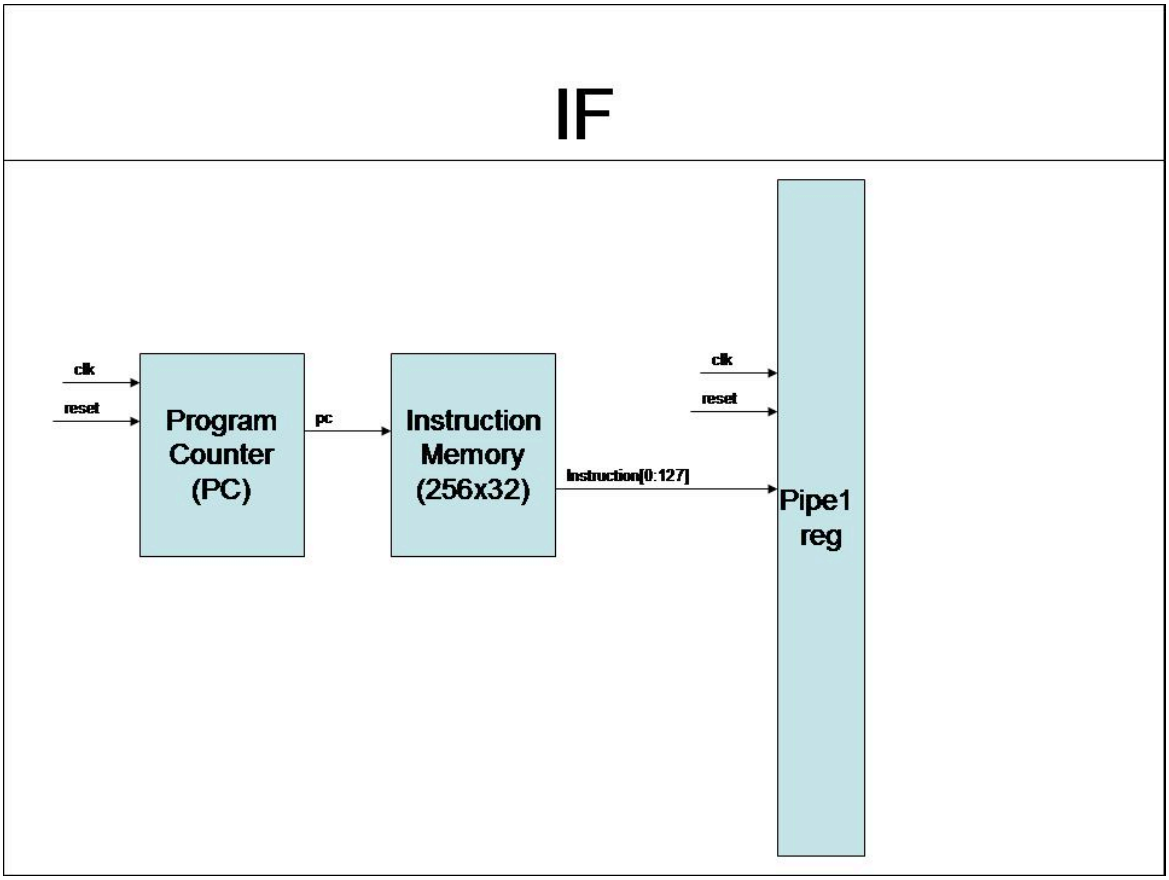
- Instruction Fetch (IF) – An instruction is fetched from the instruction memory
- Instruction Decode and Register Fetch (ID/RF) – The instruction is decoded and the applicable registers are fetched from the register file
- Execution (EX) – The ALU is used to perform the required operation on the data from the registers
- MEMory access (MEM) – the data memory is written to or read from
- WriteBack – the result of the data memory or ALU operation are written back to the register file.

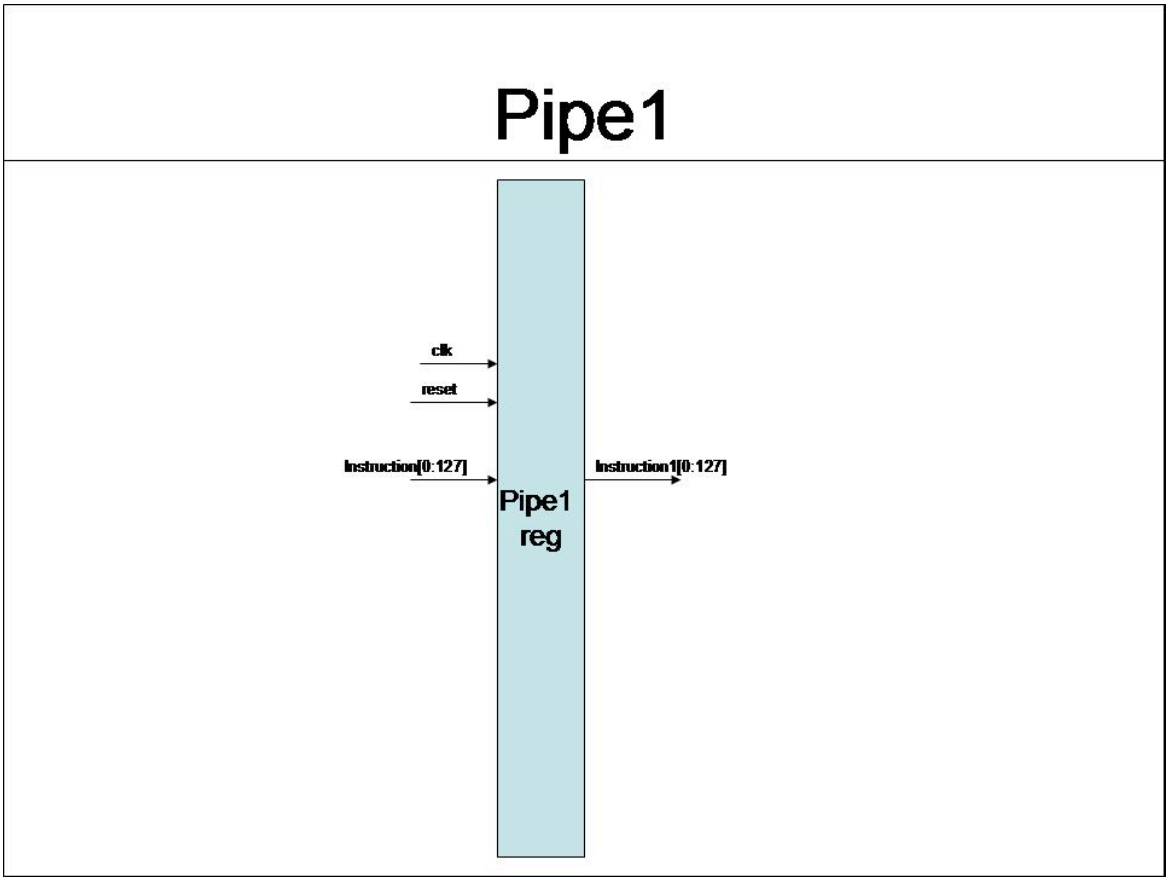
For this project, we were able to combine the MEM and EX stages of the pipeline since each of the instructions in the instruction set EITHER access memory OR use the ALU (but not both, as non-immediate memory access would). Therefore, we implemented the following 4-stage pipeline:

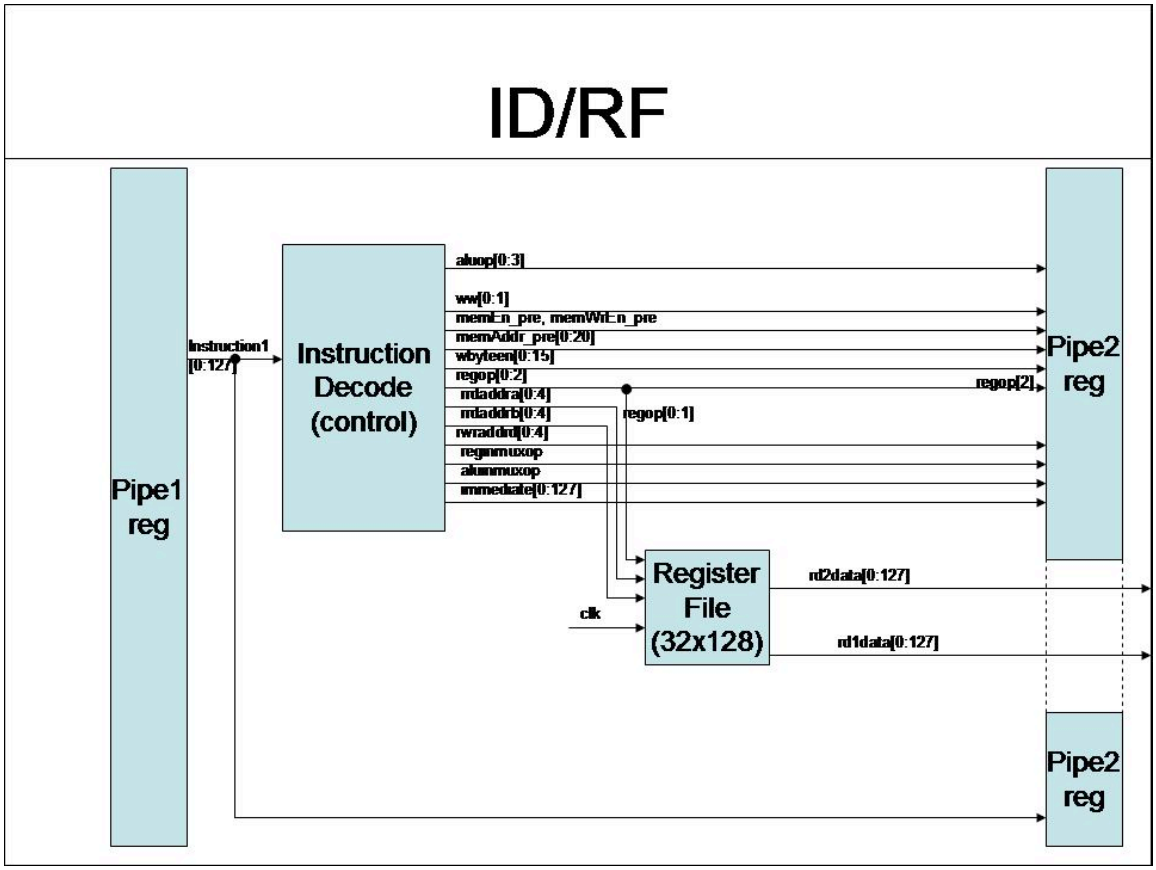
- Stage 1 – IF
- Stage 2 – ID/RF
- Stage 3 – EX/MEM
- Stage 4 – WB

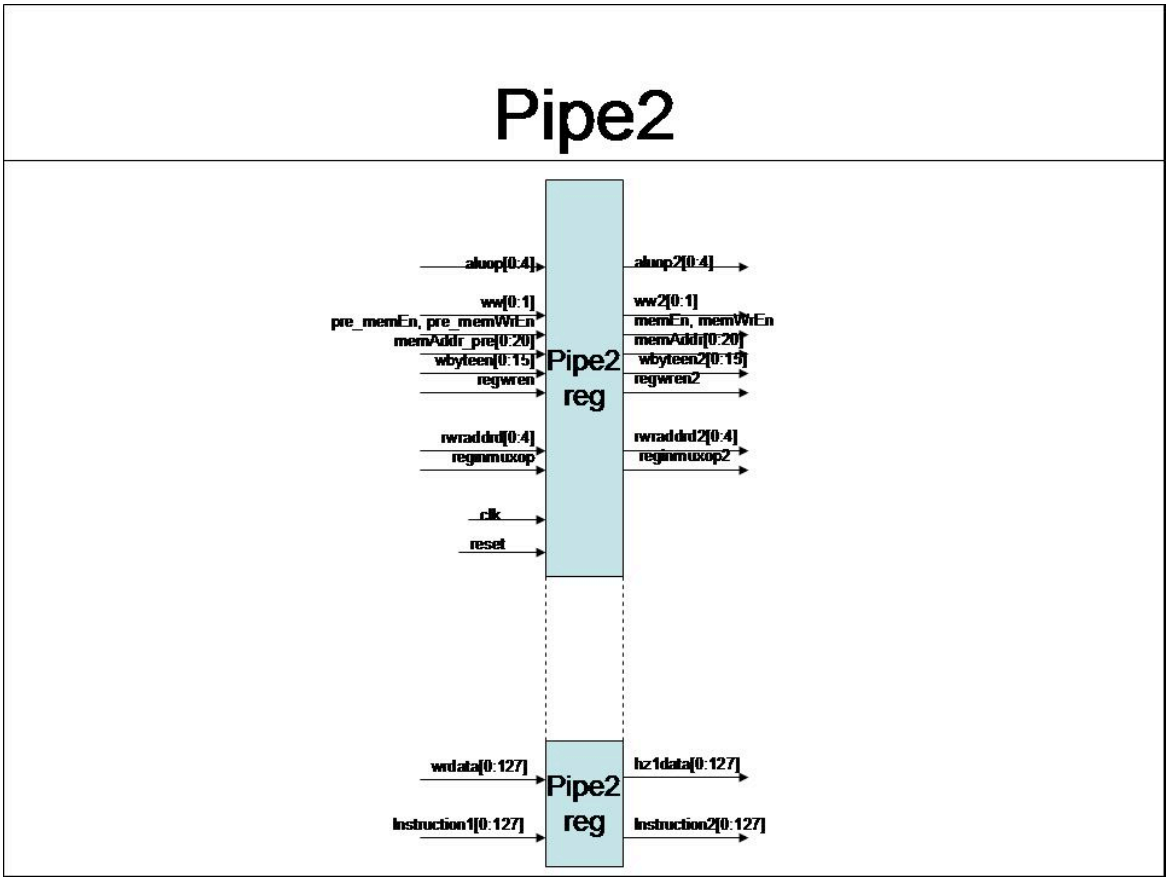
Flip Flops (FF) were used between stages 1 and 2, between stages 2 and 3, and between stages 3 and 4 to create the pipeline. The FFs between stages 1 and 2 are contained in a module called Pipe1. The FFs between stages 2 and 3 are contained in a module called Pipe2. The FFs between stages 3 and 4 are contained in a module called Pipe3. The entire pipeline is shown in the following figures:

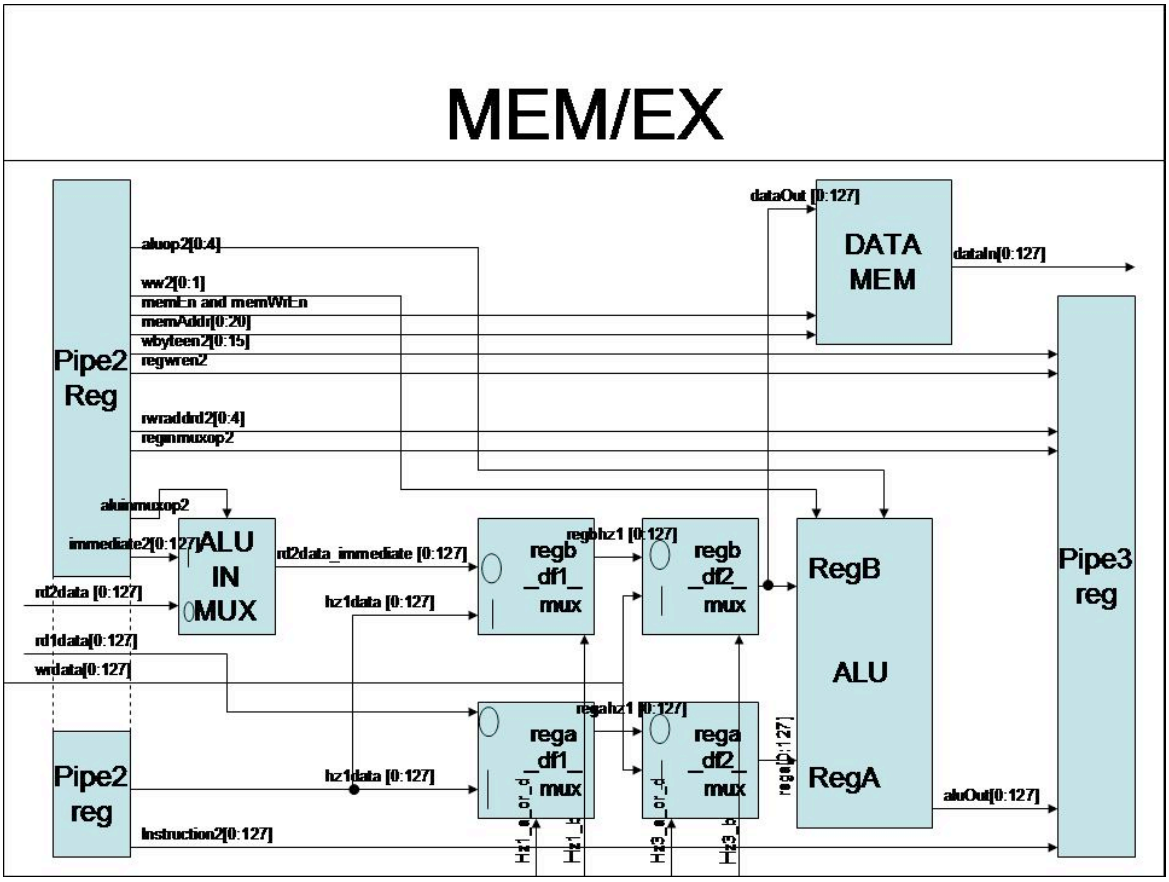




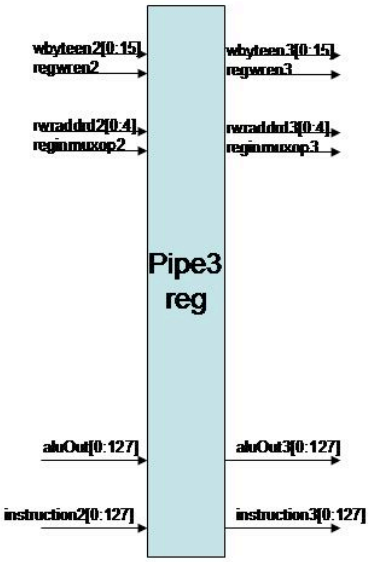

Pipe1
clk
reset
Instruction[0:127]
Instruction1[0:127]
Pipe1
reg

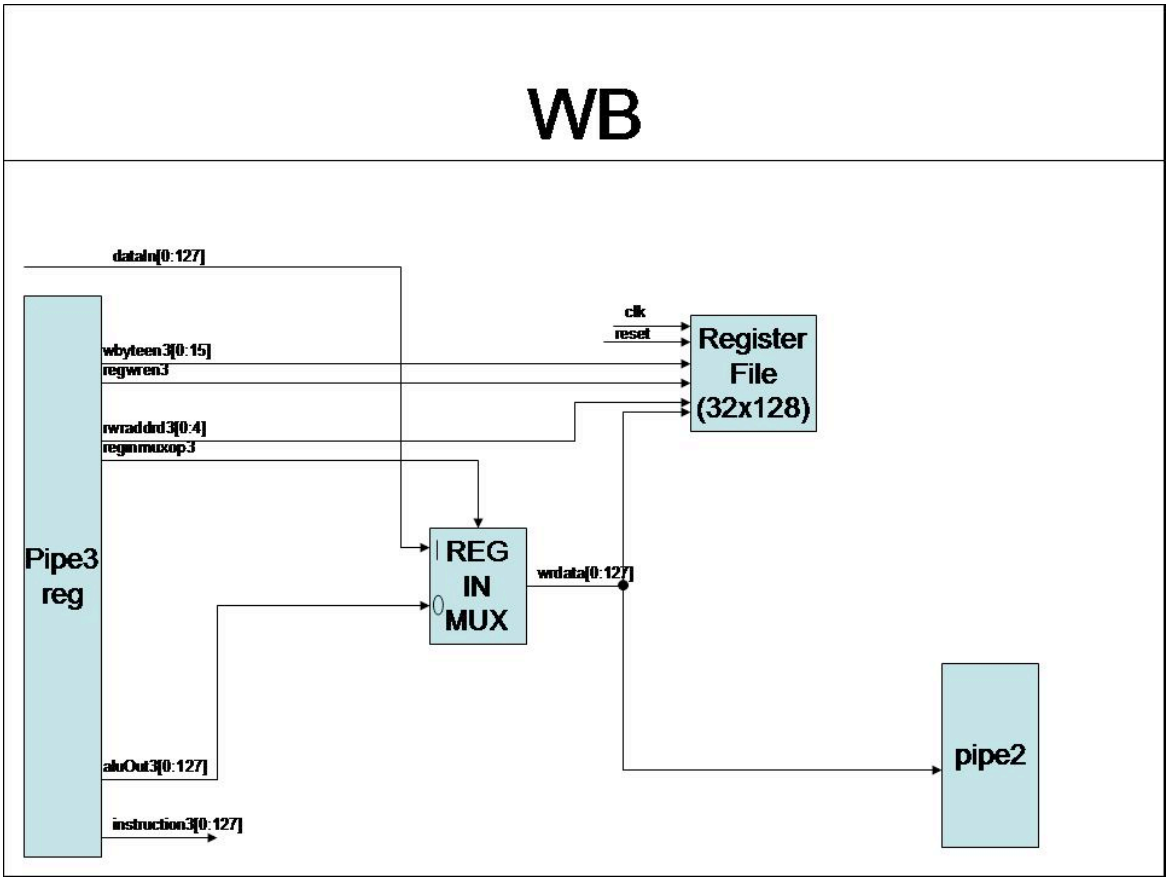

ID/RF
Pipe1 reg
Instruction1 [0:127]
Instruction Decode (control)
aluop[0:3]
ww[0:1]
memEn_pre, memWrEn_pre
memAddr_pre[0:20]
wbyteen[0:15]
regop[0:2]
rdaddra[0:4]
rdaddrb[0:4]
rwraddrd[0:4]
regmuxop
alummuxop
immediate[0:127]
regop[2]
regop[0:1]
Pipe2 reg
Register File (32x128)
clk
rd2data[0:127]
rd1data[0:127]
Pipe2 reg


Pipe2
aluop[0:4]
ww[0:1]
pre_memEn, pre_memWrEn
memAddr_pre[0:20]
wbyteen[0:15]
regwren
rwraddd[0:4]
regimmuxop
clk
reset
Pipe2 reg
aluop2[0:4]
ww2[0:1]
memEn, memWrEn
memAddr[0:20]
wbyteen2[0:15]
regwren2
rwraddd2[0:4]
regimmuxop2
wrdata[0:127]
Instruction1[0:127]
Pipe2 reg
hz1data[0:127]
Instruction2[0:127]


MEM/EX
Pipe2 Reg
aluop2[0:4]
ww2[0:1]
memEn and memWrEn
memAddr[0:20]
wbyteen2[0:15]
regwren2
rwraddrd2[0:4]
regwmuxop2
aluimmuxop2
immediate2[0:127]
ALU IN MUX
rd2data_immediate [0:127]
rd2data [0:127]
hz1data [0:127]
rd1data[0:127]
wrdata[0:127]
regbhz1 [0:127]
regb _df1_ mux
regb _df2_ mux
regahz1 [0:127]
rega _df1_ mux
rega _df2_ mux
Pipe2 reg
hz1data [0:127]
Instruction2[0:127]
dataOut [0:127]
DATA MEM
datain[0:127]
RegB
ALU
RegA
regb[0:127]
aluOut[0:127]
Pipe3 reg

Pipe3


wbyteen2[0:15]
regwren2
rwraddd2[0:4]
regimuxop2
wbyteen3[0:15]
regwren3
rwraddd3[0:4]
regimuxop3
Pipe3
reg
aluOut[0:127]
instruction2[0:127]
aluOut3[0:127]
instruction3[0:127]



It should be noted that the register file in stage 2 is the same register file in stage 4. In stage 2, only the read capability of the register file is exercised. In stage 4, only the write capability is exercised. Therefore an instruction in stage 2 can use the read side of the register file while an instruction in stage 4 of the pipeline can simultaneously access the write side of the register file.

As with any pipeline, it can only be operated at the speed of the slowest stage. In the pipeline design shown above, the MEM/EX stage (stage 3) is the slowest stage. Since the module representing the data memory is provided, there is no chance to optimize it. However, the ALU module is not provided, and therefore can be carefully designed to minimize latency. This design will be described in the following sections (3-6).

When a processor is pipelined, the possibility for data hazards is created. Section 7 addresses these hazards and shows how data forwarding was used to eliminate all data hazards.

2.1 CPU Module

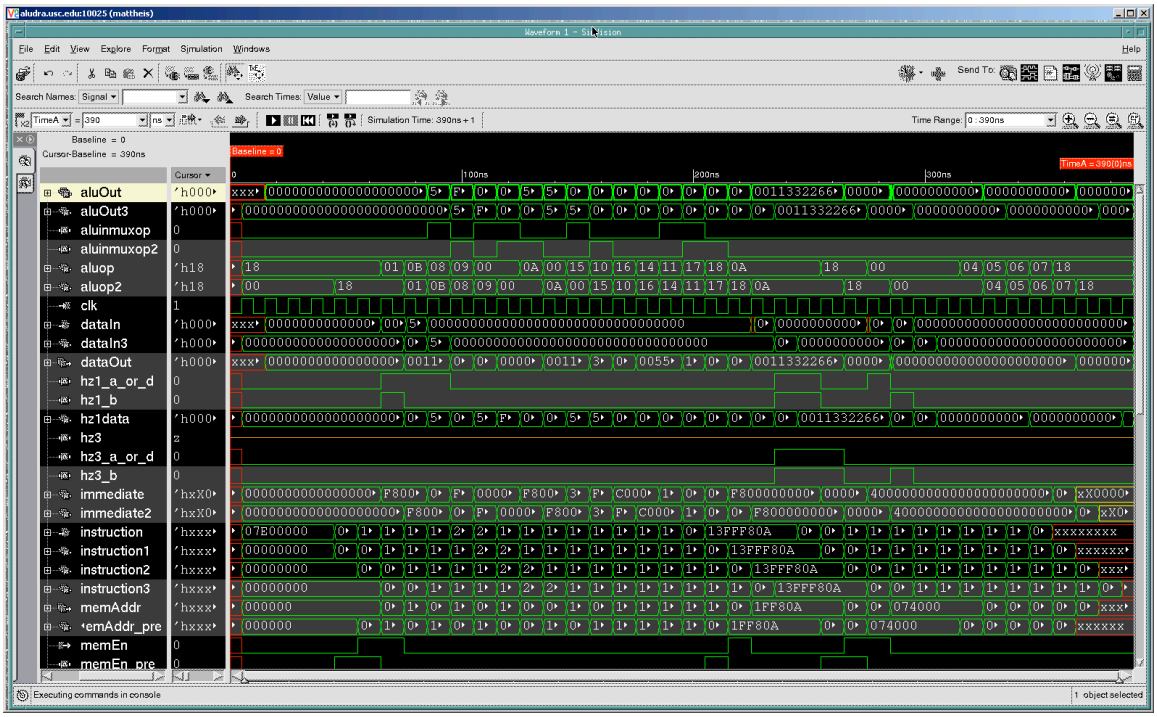
2.1.1 Inputs and Outputs

Inputs: clk, reset, [0:31] instruction, [0:31] pc, [0:127] dataIn,
Outputs: [0:127] dataOut, memEn, memWrEn, memAddr

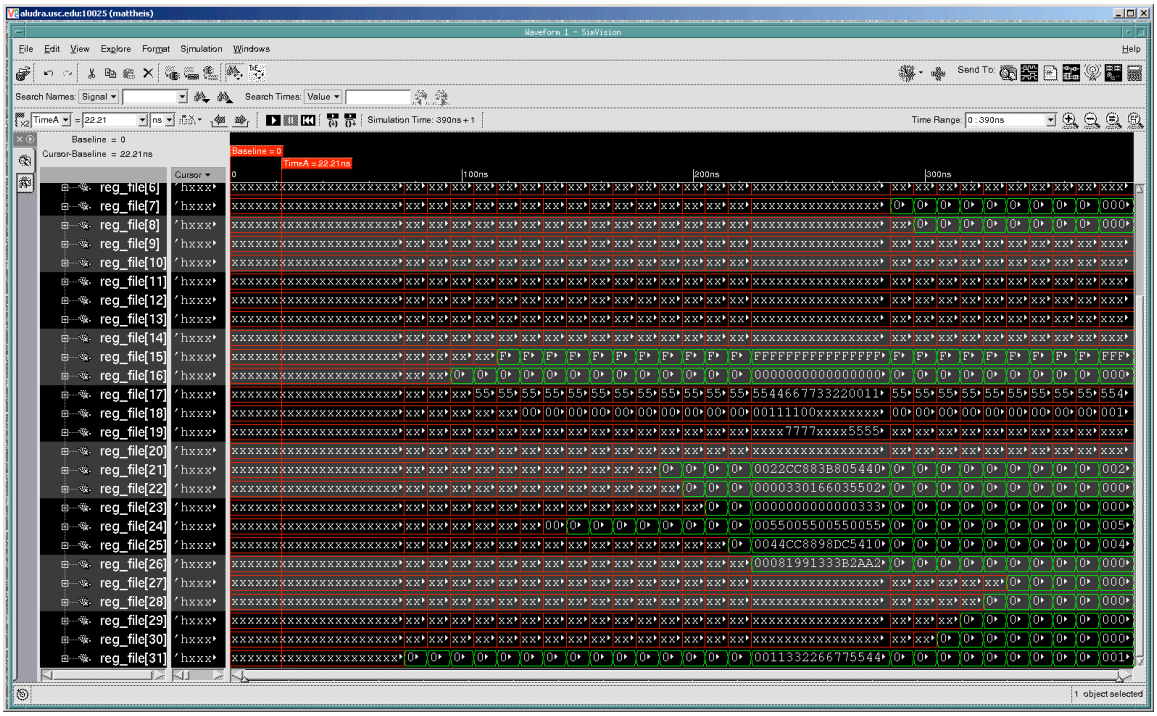
2.1.2 Functional Test Results

The following instruction set was used to test the cpu module:

```
07E00000  // wld wr31, 0 ;
04A00001  // wld wr5, 1 ;
121FF881  // wsubaw wr16, wr31, wr31 ;
1225FA4B  // wxoruh wr17, wr5, wr31 ;
11F00008  // wnotab wr15, wr16 ;
1245FC89  // wandew wr18, wr5, wr31 ;
23100400  // wmveb wr24, wr16 ;
23050500  // wmvob wr24, wr5 ;
1265FD4A  // woroh wr19, wr5, wr31 ;
129FFB40  // wadddh wr20, wr31, wr31 ;
102F3815  // wsrliab wr1, wr15, 7 ;
12BFF850  // wsllah wr21, wr31, wr31 ;
12DFC016  // wsraab wr22, wr31, wr24 ;
12FFC094  // wsrlaw wr23, wr31, wr24 ;
133F1011  // wslliab wr25, wr31, 2 ;
135F0857  // wsraiah wr26, wr31, 1 ;
0AA00002  // wst wr21, 0x02 ;
13FFF80A  // worab wr31, wr31, wr31 ;
13FFF80A  // worab wr31, wr31, wr31 ;
13FFF80A  // worab wr31, wr31, wr31 ;
13FFF80A  // worab wr31, wr31, wr31 ;
04E00003  // wld wr7, 3
05000004  // wld wr8, 4
13C74000  // addab wr30, wr7, wr8 ;
13A74000  // addab wr29, wr7, wr8 ;
13874000  // addab wr28, wr7, wr8 ;
13674000  // addab wr27, wr7, wr8 ;
13C74044  // mules wr30, wr7, wr8 ;
13A74045  // muleu wr29, wr7, wr8 ;
13874046  // mulos wr28, wr7, wr8 ;
13674047  // mulou wr27, wr7, wr8 ;
00000000  // NOP ;
```



The outputs were carefully scrutinized to ensure that re result of each operation was stored in the correct location.



And the data memory was correct after the run:
Memory location # 0 : 0011332266775544ccddffeeaabb9988
Memory location # 1 : 55555555555555555555555555555555
Memory location # 2 : 0022cc883b805440a0008000d8008800
Memory location # 3 : 00000000000000000000000000000202
Memory location # 4 : 00000000000000000000000000000303
Memory location # 5 : xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
…
Memory location # 63 : xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

At this point hazard detection had already been implemented as can be seen by the signals hz1_a_or_d, hz1_b, hz3_a_or_d, and hz3_b. Detailed disussion about the hazards in the above functional test will be discussed in section 7.

2.1.3 Synthesis Results

See section 9.

2.1.4 Post-Synthesis Functional Test Results

The post synthesis functional test results show that the cpu has the same functionality after synthesis.

As part of this project, the following 6 random instruction sets were generated to determine the speed of the pipelined processor:

<INCLUDE HERE>

3 Instruction set implementation

The instruction set introduced in the introduction is described in detail in this section. The rest of this section has been copied with permission from the USC document for EE577B Fall 2007 isa_ov.pdf.

Chapter 1 - Troy WideWord Processor Instruction Set Overview

Notation

This chapter gives an instruction set overview and the following chapter gives a detailed instruction description. All instructions are 32 bits. Big-Endian byte and bit labeling is used, meaning that bit/byte 0 is the most significant. Other conventions are listed in the table below.

TABLE 1. Instruction Glossary

Symbol	Meaning	Symbol	Meaning
$A \leftarrow B$	Assignment	x ^ y	x bitwise exclusive ORed with y
{x,y}	Bit string concatenation	~x	bitwise inversion of x
{y{x}}	x replicated y times	MEM[EA]	Memory contents at effective address EA
x[y:z]	Selection of bits y through z from x	0x*value*	Hexadecimal value
x & y	x bitwise ANDed with y	0b*value*	Binary value
x | y	x bitwise ORed with y	(rX)	Contents of general-purpose register X

The following table gives the rules of precedence and associativity for the pseudocode operators. All operators on the same line have equal precedence, and all operators on a given line have higher precedence than those on the lines below them.

TABLE 2. Precedence of Pseudocode Operators

Operator	Associativity
MEM[n]	left to right
x[y:z]	left to right
{y{x}}	left to right
~	right to left
×, ÷	left to right
+, -	left to right
{x,y}	left to right
=, !=, <, <=, >, >=	left to right
^, &	left to right
|	left to right
←	none

Page 2 of 26

Instruction Formats

As shown in Figure 1, most Troy WideWord instructions use a three-operand format to specify two 128-bit source registers (wrA and wrB) and a 128-bit destination register (wrD). Load and store use a different instruction format shown in Figure 2. Note that these classifications are generalizations, and some instructions will vary somewhat from the format for which it is classified. For instance, the *shift immediate* instructions are classified as W-type instructions, although they specify a 5-bit immediate shift amount in the place of wrB. The end result is that not all W-type instructions can be decoded in exactly the same way; the wrB field may be a register specifier or it may be a 5-bit immediate.

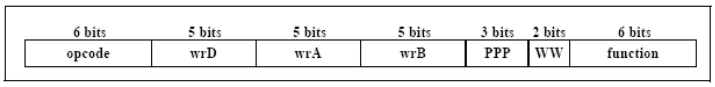


Figure 1 Format W for WideWord Arithmetic/Logical Operations

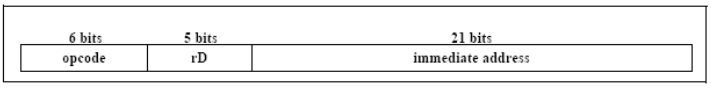


Figure 2 Format M for Wide-Word Load/Store Operations

The control fields are defined as follows:

WW (width)

The 2-bit *WW* field sets the width of the WideWord operands to eight, sixteen, or thirty-two bits, which primarily affects the shift operations, configuration of the carry chain for additions and subtractions, and the multiplication operations.The encoding of these bits is listed in the following table:

WW Value	Operand Width	Assembler Mnemonic
00	8 bits	b
01	16 bits	h
10	32 bits	w
11	Reserved	NA

PPP (participation)

The 3-bit *PPP*, or participation, field specifies what kind of selective execution, if any, governs the operation. Recall that under selective execution only certain subfields commit their results during the writeback stage. The subfields participating are specified by the decoding of *PPP*. The encoding of the *PPP* bits is listed in the following table:

PPP Value	Participation Definition	Assembler Mnemonic
000	All subfields participate	a
001	*reserved*	*none*
010	Upper 64-bits participate	u
011	Lower 64-bits participate	d
100	Subfields with even index participate	e
101	Subfields with odd index participate	o
110	Only most significant subfield (subfield 0) participates	m
111	Only least significant subfield participates	l

For those participation modes that refer to the even/odd indices or most/least significant subfields, the exact bits that participate depend also on the operand width specified by the *WW* field. The following table shows the possible subfield indices for the different values of *WW* (recall that Big-Endian labeling is used so that subfield 0 is always the most significant regardless of operand size):

WW Value	Subfield Indices within a WideWord for Differing Operand Widths															
	most sig.															least sig.
00	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
01	0		1		2		3		4		5		6		7	
10	0				1				2				3			

This table may be useful for visualizing which subfield(s) participate based on the selective execution mode and operand width value. For instance, an "upper" participate mode using 32-bit operands means that only words 0 and 1 participate.

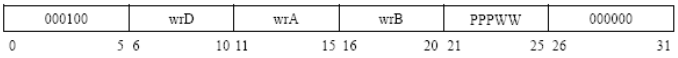
Alphabetical list of instructions

TABLE 3. Preliminary Encoding of DIVA Instruction Set

		Encoding					
Instruction	Format	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
WADD	W	000100	wrD	wrA	wrB	PPPWW	000000
WAND	W	000100	wrD	wrA	wrB	PPPWW	001001
WLD	M	000001	wrD	immediate_address			
WMULES	W	000100	wrD	wrA	wrB	PPPWW	000100
WMULEU	W	000100	wrD	wrA	wrB	PPPWW	000101
WMULOS	W	000100	wrD	wrA	wrB	PPPWW	000110
WMULOU	W	000100	wrD	wrA	wrB	PPPWW	000111
WMV	W	001000	wrD	wrA	00000	PPPWW	000000
WNOT	W	000100	wrD	wrA	00000	PPPWW	001000
WOR	W	000100	wrD	wrA	wrB	PPPWW	001010
WPRM	W	000100	wrD	wrA	wrB	PPP00	001100
WSLL	W	000100	wrD	wrA	wrB	PPPWW	010000
WSLLI	W	000100	wrD	wrA	shift_amount	PPPWW	010001
WSRA	W	000100	wrD	wrA	wrB	PPPWW	010110
WSRAI	W	000100	wrD	wrA	shift_amount	PPPWW	010111
WSRL	W	000100	wrD	wrA	wrB	PPPWW	010100
WSRLI	W	000100	wrD	wrA	shift_amount	PPPWW	010101
WST	M	000010	wrD	immediate_address			
WSUB	W	000100	wrD	wrA	wrB	PPPWW	000001
WXOR	W	000100	wrD	wrA	wrB	PPPWW	001011

waddx - WideWord Add

wadd*pw* wrD, wrA, wrB



Variable values in the following equations are as follows:

WW Value	size
00	8
01	16
10	32

for i = 0 to (128 - size) by size

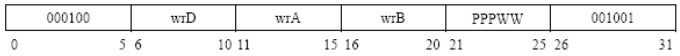
if PPP bits enable writeback for this subfield

$$wrD[i:(i+(size-1))] \leftarrow (wrA)[i:(i+(size-1))] + (wrB)[i:(i+(size-1))]$$

The WW field determines if the 128-bit contents of wrA and wrB are treated as 32 bytes, 16 half-words, or 8 words. The aggregate sums of the aligned data fields of wrA and wrB are placed into wrD, subject to participation.

wandx - WideWord AND

wand*pw* wrD, wrA, wrB



Variable values in the following equations are as follows:

WW Value	size
00	8
01	16
10	32

for i = 0 to (128 - size) by size

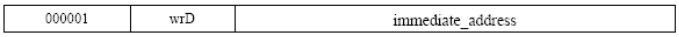
if PPP bits enable writeback for this subfield

$$wrD[i:(i+(size-1))] \leftarrow (wrA)[i:(i+(size-1))] \ \& \ (wrB)[i:(i+(size-1))]$$

The 128-bit contents of wrA are ANDed with the 128-bit contents of wrB, and the result is placed into wrD, subject to participation. The WW field simply effects how participation applies for this operation.

wld - Load WideWord Register

wld wrD, immediate_address



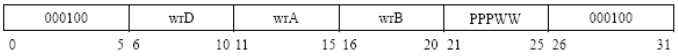
$EA \leftarrow \{\text{immediate\_address}, 4\{0\}\}$

$wrD \leftarrow \text{MEM}[EA]$

The immediate_address is assumed to be in units of 128-bit wide words. Thus, to obtain an effective address, EA, in units of bytes, 4 zeros are appended to the immediate_address. The 128-bit value at the memory location specified by EA is then loaded into wrD.

wmules - WideWord Multiply Even Signed

wmules*pw* wrD, wrA, wrB



WW Value	input size	output size
01	8	16
10	16	32

In the equation below, the variable *size* refers to the input size given in the table above.

for i = 0 to (128 - 2 × size) by 2 × size

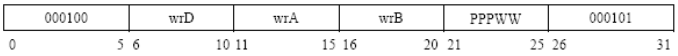
if PPP bits enable writeback for this subfield

$$wrD[i:(i+(2\times size-1))] \leftarrow (wrA)[i:(i+(size-1))] \times (wrB)[i:(i+(size-1))]$$

Each even-numbered signed-integer byte or half-word of wrA is multiplied by the corresponding signed-integer byte or half-word of wrB, where the WW field determines if the 128-bit contents of wrA and wrB are treated as bytes or half-words. The resulting signed halfword or word products are placed, in the same order, into wrD, subject to participation.

wmuleu - WideWord Multiply Even Unsigned

wmuleu*pw* wrD, wrA, wrB



WW Value	input size	output size
01	8	16
10	16	32

In the equation below, the variable *size* refers to the input size given in the table above.

for i = 0 to (128 - 2 × size) by 2 × size

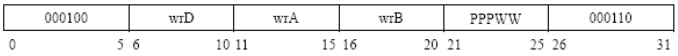
if PPP bits enable writeback for this subfield

$$wrD[i:(i+(2\times size-1))] \leftarrow (wrA)[i:(i+(size-1))] \times (wrB)[i:(i+(size-1))]$$

Each even-numbered unsigned-integer byte or half-word of wrA is multiplied by the corresponding unsigned-integer byte or half-word of wrB, where the WW field determines if the 128-bit contents of wrA and wrB are treated as bytes or half-words. The resulting unsigned half-word or word products are placed, in the same order, into wrD, subject to participation.

wmulos - WideWord Multiply Odd Signed

wmulos*pw* wrD, wrA, wrB



WW Value	input size	output size
01	8	16
10	16	32

In the equation below, the variable *size* refers to the input size given in the table above.

for i = 0 to (128 - 2 × size) by 2 × size

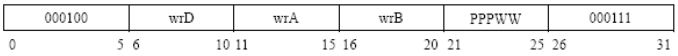
if PPP bits enable writeback for this subfield

$$wrD[i:(i+(2\times size-1))] \leftarrow (wrA)[(i+size):(i+(2\times size-1))] \times (wrB)[(i+size):(i+(2\times size-1))]$$

Each odd-numbered signed-integer byte or half-word of wrA is multiplied by the corresponding signed-integer byte or half-word of wrB, where the WW field determines if the 128-bit contents of wrA and wrB are treated as bytes or half-words. The resulting signed halfword or word products are placed, in the same order, into wrD, subject to participation.

wmulou - WideWord Multiply Odd Unsigned

wmulou*pw* wrD, wrA, wrB



WW Value	input size	output size
01	8	16
10	16	32

In the equation below, the variable *size* refers to the input size given in the table above.

for i = 0 to (128 - 2 × size) by 2 × size

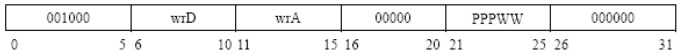
if PPP bits enable writeback for this subfield

$$wrD[i:(i+(2\times size-1))] \leftarrow (wrA)[(i+size):(i+(2\times size-1))] \times (wrB)[(i+size):(i+(2\times size-1))]$$

Each odd-numbered unsigned-integer byte or half-word of wrA is multiplied by the corresponding unsigned-integer byte or half-word of wrB, where the WW field determines if the 128-bit contents of wrA and wrB are treated as bytes or half-words. The resulting unsigned half-word or word products are placed, in the same order, into wrD, subject to participation.

wmv*x* - Move from WideWord to WideWord

wmv*p* wrD, wrA



Variable values in the following equations are as follows:

WW Value	size
00	8
01	16
10	32

for i = 0 to (128 - size) by size

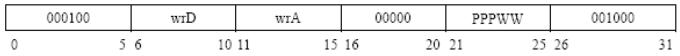
if PPP bits enable writeback for this subfield

$wrD[i:(i+(size-1))] \leftarrow (wrA)[i:(i+(size-1))]$

The entire 128-bit contents of wrA are transferred to wrD, subject to the participation mode specified by PPP. The WW field simply effects how participation applies for this operation.

wnot*x* - WideWord NOT

wnot*pw* wrD, wrA



Variable values in the following equations are as follows:

WW Value	size
00	8
01	16
10	32

for i = 0 to (128 - size) by size

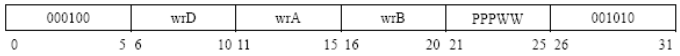
if PPP bits enable writeback for this subfield

$wrD[i:(i+(size-1))] \leftarrow \sim(wrA)[i:(i+(size-1))]$

The 128-bit contents of wrA are bitwise inverted, and the result is placed into wrD, subject to participation. The WW field simply effects how participation applies for this operation.

worx - WideWord OR

worpw wrD, wrA, wrB



Variable values in the following equations are as follows:

WW Value	size
00	8
01	16
10	32

for i = 0 to (128 - size) by size

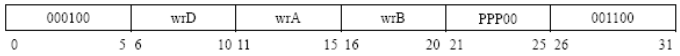
if PPP bits enable writeback for this subfield

$wrD[i:(i+(size-1))] \leftarrow (wrA)[i:(i+(size-1))] \mid (wrB)[i:(i+(size-1))]$

The 128-bit contents of wrA are ORed with the 128-bit contents of wrB, and the result is placed into wrD, subject to participation. The WW field simply effects how participation applies for this operation.

wprmx - WideWord Permute

wprmp wrD, wrA, wrB



for i = 0 to 120 by 8

$s \leftarrow (wrB)[(i+4):(i+7)]$

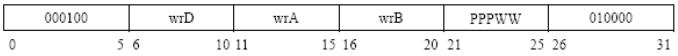
if PPP bits enable writeback for this byte subfield

$wrD[i:(i+7)] \leftarrow (wrA)[s \times 8:((s \times 8)+7)]$

The contents of wrA are the source vector for this permutation operation. Bits 4 to 7 of each byte element of the contents of wrB are used to select a byte element from the source vector for each byte element of the result. The result is placed into wrD, subject to participation.

wsllx - WideWord Shift Left Logical

wsll*pw* wrD, wrA, wrB



Variable values in the following equations are as follows:

WW Value	size	bits
00	8	3
01	16	4
10	32	5

for i = 0 to (128 - size) by size

$s \leftarrow (wrB)[(i + size - bits):(i + size - 1)]$

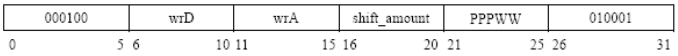
if PPP bits enable writeback for this subfield

$wrD[i:(i + (size - 1))] \leftarrow \{(wrA)[(i + s):(i + size - 1)], s\{0\}\}$

The WW field determines if the 128-bit contents of wrA and wrB are treated as 32 bytes, 16 half-words, or 8 words. The contents of each data field of wrA are shifted left by the number of bits specified by the low order bits of the corresponding data field contained as contents of wrB, inserting zeros into the low order bits of each data field of the result. The result is placed into wrD, subject to participation.

wsllix - WideWord Shift Left Logical Immediate

wslli*pw* wrD, wrA, shift_amount



Variable values in the following equations are as follows:

WW Value	size	bits
00	8	3
01	16	4
10	32	5

$s \leftarrow$ shift_amount$[(5 - bits):4]$

for i = 0 to (128 - size) by size

if PPP bits enable writeback for this subfield

$wrD[i:(i + (size - 1))] \leftarrow \{(wrA)[(i + s):(i + size - 1)], s\{0\}\}$

The WW field determines if the 128-bit contents of wrA are treated as 32 bytes, 16 half-words, or 8 words. The contents of each data field of wrA are shifted left by the number of bits specified by the appropriate bits of the shift_amount, inserting zeros into the low order bits of each data field of the result. The result is placed into wrD, subject to participation.

wsrax - WideWord Shift Right Arithmetic

wsra*pw* wrD, wrA, wrB

000100	wrD	wrA	wrB	PPPWW	010110
0 5	6 10	11 15	16 20	21 25	26 31

Variable values in the following equations are as follows:

WW Value	size	bits
00	8	3
01	16	4
10	32	5

for i = 0 to (128 - size) by size

$s \leftarrow (wrB)[(i+size-bits):(i+size-1)]$

if PPP bits enable writeback for this subfield

$rD[i:(i+(size-1))] \leftarrow \{s\{(wrA)[i]\}, (wrA)[i:(i+size-s-1)]$

The WW field determines if the 128-bit contents of wrA and wrB are treated as 32 bytes, 16 half-words, or 8 words. The contents of each data field of wrA are shifted right by the number of bits specified by the low order bits of the corresponding data field contained as contents of wrB, sign-extending the high-order bits of each data field of the result. The result is placed into wrD, subject to participation.

wsraix - WideWord Shift Right Arithmetic Immediate

wsrai*pw* wrD, wrA, shift_amount

000100	wrD	wrA	shift_amount	PPPWW	010111
0 5	6 10	11 15	16 20	21 25	26 31

Variable values in the following equations are as follows:

WW Value	size	bits
00	8	3
01	16	4
10	32	5

$s \leftarrow$ shift_amount$[(5-bits):4]$

for i = 0 to (128 - size) by size

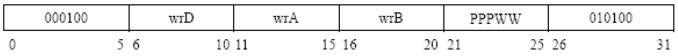
if PPP bits enable writeback for this subfield

$rD[i:(i+(size-1))] \leftarrow \{s\{(wrA)[i]\}, (wrA)[i:(i+size-s-1)]$

The WW field determines if the 128-bit contents of wrA are treated as 32 bytes, 16 half-words, or 8 words. The contents of each data field of wrA are shifted right by the number of bits specified by the appropriate bits of the shift_amount, sign-extending the high-order bits of each data field of the result. The result is placed into wrD, subject to participation.

wsrlx - WideWord Shift Right Logical

wsrl*pw* wrD, wrA, wrB



Variable values in the following equations are as follows:

WW Value	size	bits
00	8	3
01	16	4
10	32	5

for i = 0 to (128 - size) by size

$s \leftarrow (wrB)[(i+size-bits):(i+size-1)]$

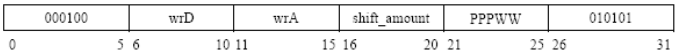
if PPP bits enable writeback for this subfield

$wrD[i:(i+(size-1))] \leftarrow \{s\{0\}, (wrA)[i:(i+size-s-1)]\}$

The WW field determines if the 128-bit contents of wrA and wrB are treated as 32 bytes, 16 half-words, or 8 words. The contents of each data field of wrA are shifted right by the number of bits specified by the low order bits of the corresponding data field contained as contents of wrB, inserting zeros into the high-order bits of each data field of the result. The result is placed into wrD, subject to participation.

wsrlix - WideWord Shift Right Logical Immediate

wsrli*pw* wrD, wrA, shift_amount



Variable values in the following equations are as follows:

WW Value	size	bits
00	8	3
01	16	4
10	32	5

$s \leftarrow$ shift_amount$[(5-bits):4]$

for i = 0 to (128 - size) by size

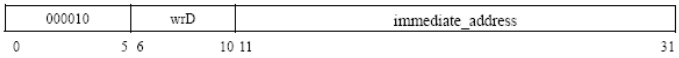
if PPP bits enable writeback for this subfield

$wrD[i:(i+(size-1))] \leftarrow \{s\{0\}, (wrA)[i:(i+size-s-1)]\}$

The WW field determines if the 128-bit contents of wrA are treated as 32 bytes, 16 half-words, or 8 words. The contents of each data field of wrA are shifted right by the number of bits specified by the appropriate bits of the shift_amount, inserting zeros into the high-order bits of each data field of the result. The result is placed into wrD, subject to participation.

wst - Store WideWord Register

wst wrD, immediate_address



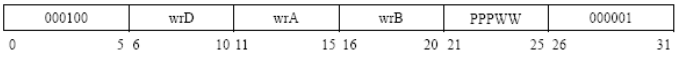
$EA \leftarrow \{\text{immediate\_address}, 4\{0\}\}$

$MEM[EA] \leftarrow (wrD)$

The immediate_address is assumed to be in units of 128-bit wide words. Thus, to obtain an effective address, EA, in units of bytes, 4 zeros are appended to the immediate_address. The 128-bit contents of wrD are stored at the memory location specified by EA.

wsub*x* - WideWord Subtract

wsub*pw* wrD, wrA, wrB



Variable values in the following equations are as follows:

WW Value	size
00	8
01	16
10	32

for i = 0 to (128 - size) by size

if PPP bits enable writeback for this subfield

$wrD[i:(i+(size-1))] \leftarrow (wrA)[i:(i+(size-1))] + \sim(wrB)[i:(i+(size-1))] + 1$

The WW field determines if the 128-bit contents of wrA and wrB are treated as 32 bytes, 16 half-words, or 8 words. The aggregate differences of the aligned data fields of wrA and wrB are placed into wrD, subject to participation.

wxor*x* - WideWord Exclusive-OR

wxor*pw* wrD, wrA, wrB

000100	wrD	wrA	wrB	PPPWW	001011
0 5	6 10	11 15	16 20	21 25	26 31

Variable values in the following equations are as follows:

WW Value	size
00	8
01	16
10	32

for i = 0 to (128 - size) by size

if PPP bits enable writeback for this subfield

$wrD[i:(i+(size-1))] \leftarrow (wrA)[i:(i+(size-1))] \ ^{\wedge} \ (wrB)[i:(i+(size-1))]$

The 128-bit contents of wrA are exclusive-ORed with the 128-bit contents of wrB, and the result is placed into wrD, subject to participation. The WW field simply effects how participation applies for this operation.

4 Adder/Subtractor design

The Synopsys DesignWare intellectual property (IP) library consists of high performance IP blocks for system development and integration to reduce development time, and time-to-market (Cohen et al, 1996). Given the short design time of this processor project and the high performance requirement, we decided to utilize the virtual microarchitecture library from DesignWare IP to implement the adder design (Synopsys, 2001). This would save us time from implementing advanced adder designs, using Verilog, in structural RTL. It would also save us numerous man-hours from iterating the design process to verify that the design functions correctly, and optimize the design for high performance within our aggressive schedule.

(Koren, 1993)

Some adder designs
http://www.ece.iit.edu/~jstine/book/

http://ieeexplore.ieee.org/search/wrapper.jsp?arnumber=835116

Simulators for various adder designs, along with other arithmetic circuits, can be obtained at http://www.ecs.umass.edu/ece/koren/arith/simulator/.

4.1 Functional Test Results

See section 8.

4.2 Synthesis Results

The following results were found in the control.area report:

Number of ports:	391
Number of nets:	4270
Number of cells:	3623
Number of references:	43
Combinational area:	17115.000000
Noncombinational area:	0.000000
Total cell area:	17115.000000

The following timing information was extracted from the timing report:

Max delay:	7.11 delay units.

The check_design report indicated that six of the synthesized cells did not drive any nets. We kept this in mind while testing the functionality of the synthesized code and found that the warnings could be ignored.

4.3 Post-Synthesis Functional Test Results

See section 10.

5 Shifter Design

5.1 Functional Test Results

See section 8.

5.2 Synthesis Results

TBD – redesigned and removed latches

5.3 Post-Synthesis Functional Test Results

See section 10.

6 Multiplier Design

6.1 Functional Test Results

See section 8.

6.2 Synthesis Results

The following results were found in the control.area report:

Number of ports:	391
Number of nets:	32234
Number of cells:	27146
Number of references:	222
Combinational area:	1410694.000000
Noncombinational area:	0.000000

Total cell area: 1410694.000000

The following timing information was extracted from the timing report:
Max delay: 37.5 delay units.
This is by far the worst delay of any of the ALU functions. Therefore this is on the critical path in the pipelined CPU. See Section 11 for more about this.

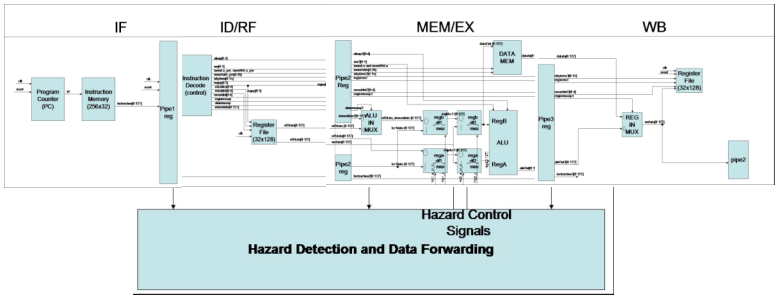
The check_design report indicated that 777 of the synthesized cells did not drive any nets. We kept this in mind while testing the functionality of the synthesized code and found that the warnings could be ignored.

6.3 Post-Synthesis Functional Test Results

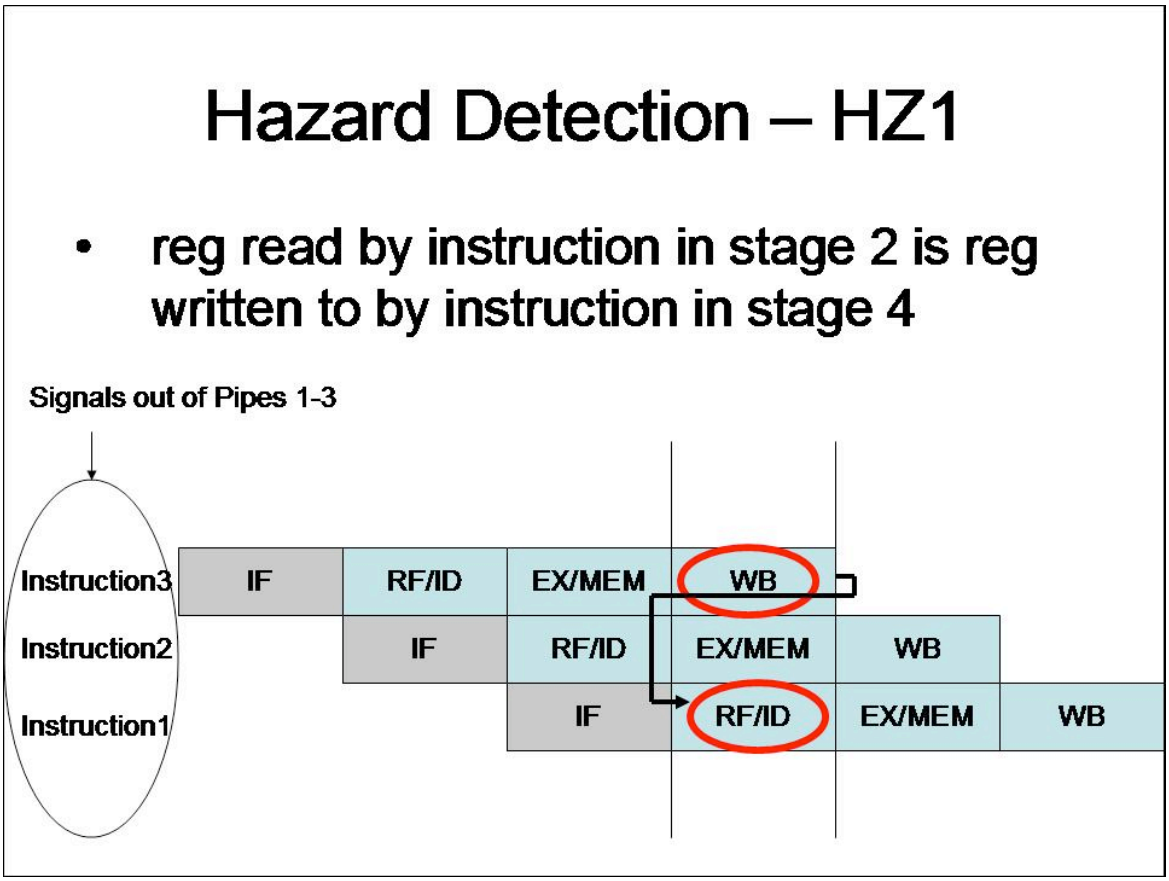
See section 10.

7 Hazard management and data forwarding

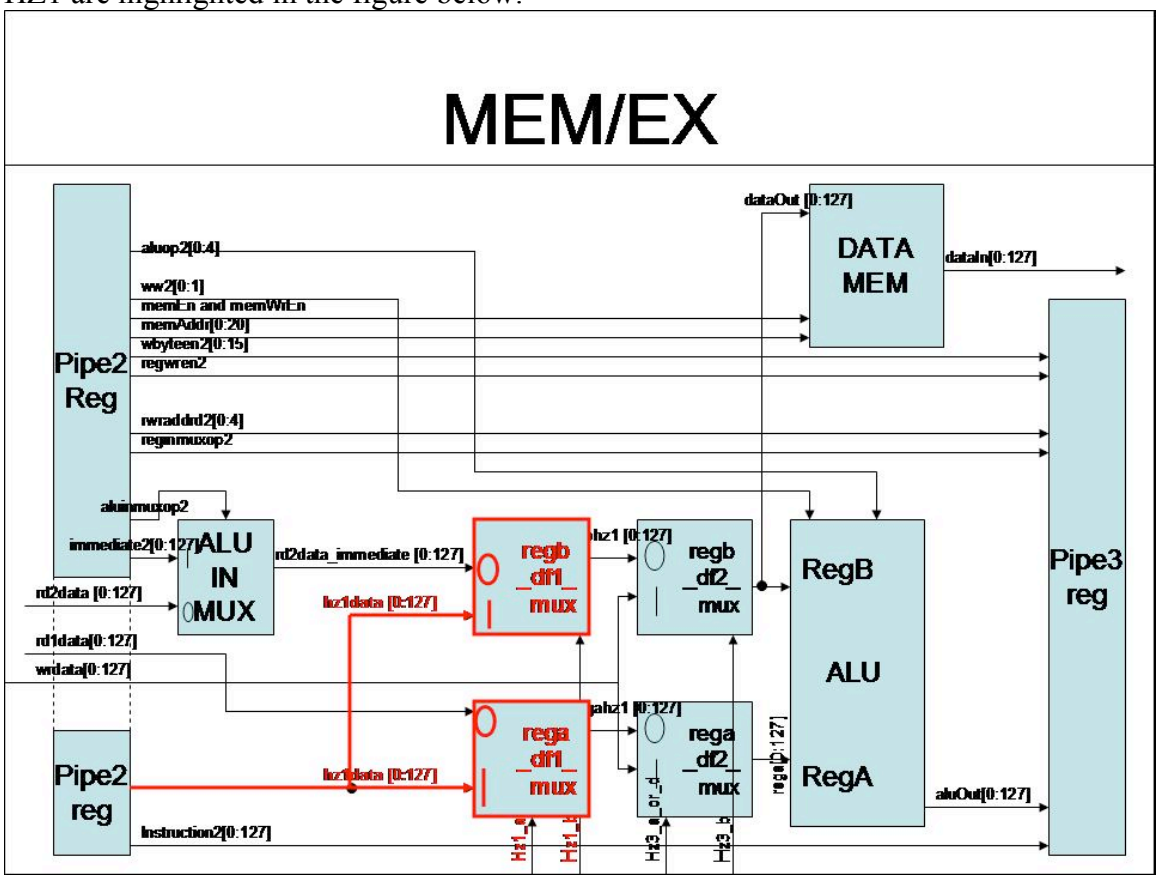
Hazard detection occurs due to the pipelining process when multipleinstructions in the pipeline are accessing the same storage location (register file memory location). The hazard detection module inspects theinstruction signals in all stages of the pipeline at the same time as shown below:

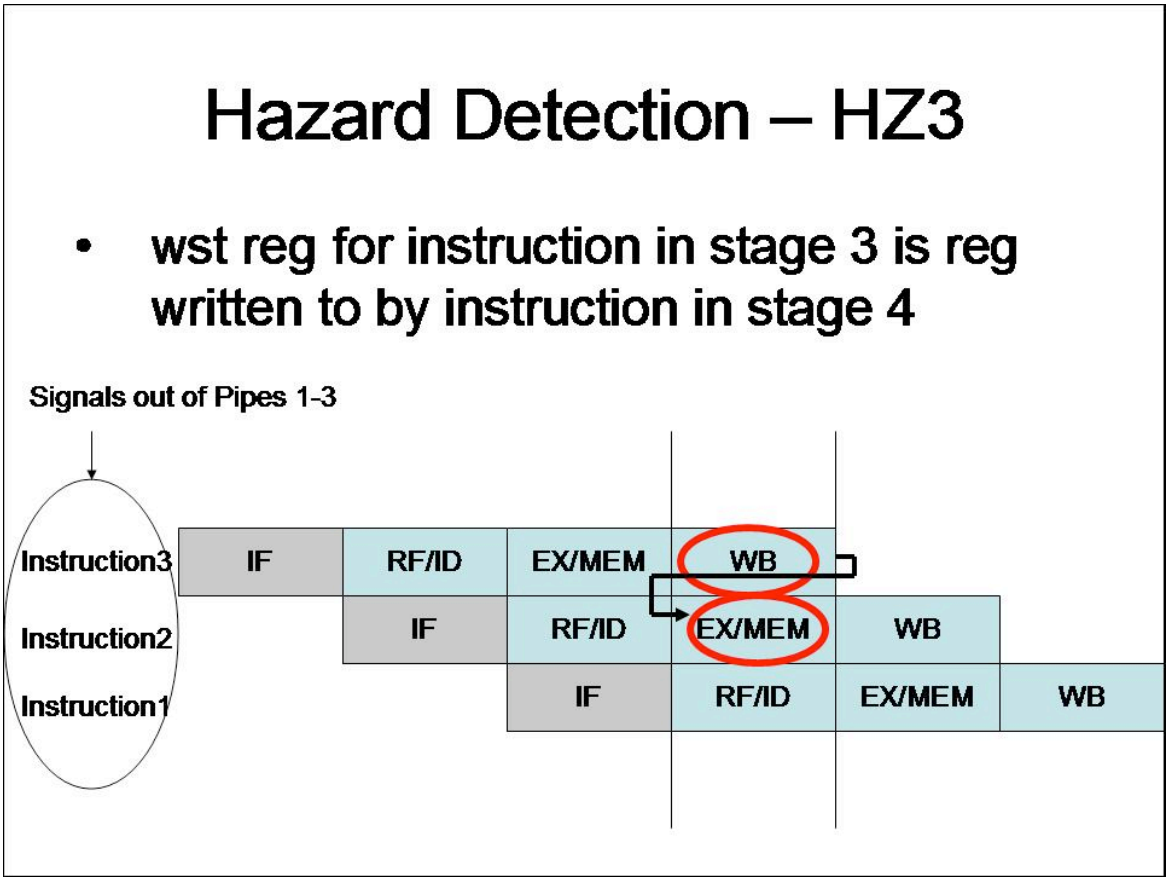


There are two types of data hazards in the 4 stage pipeline described in section 2. These hazards are described below.

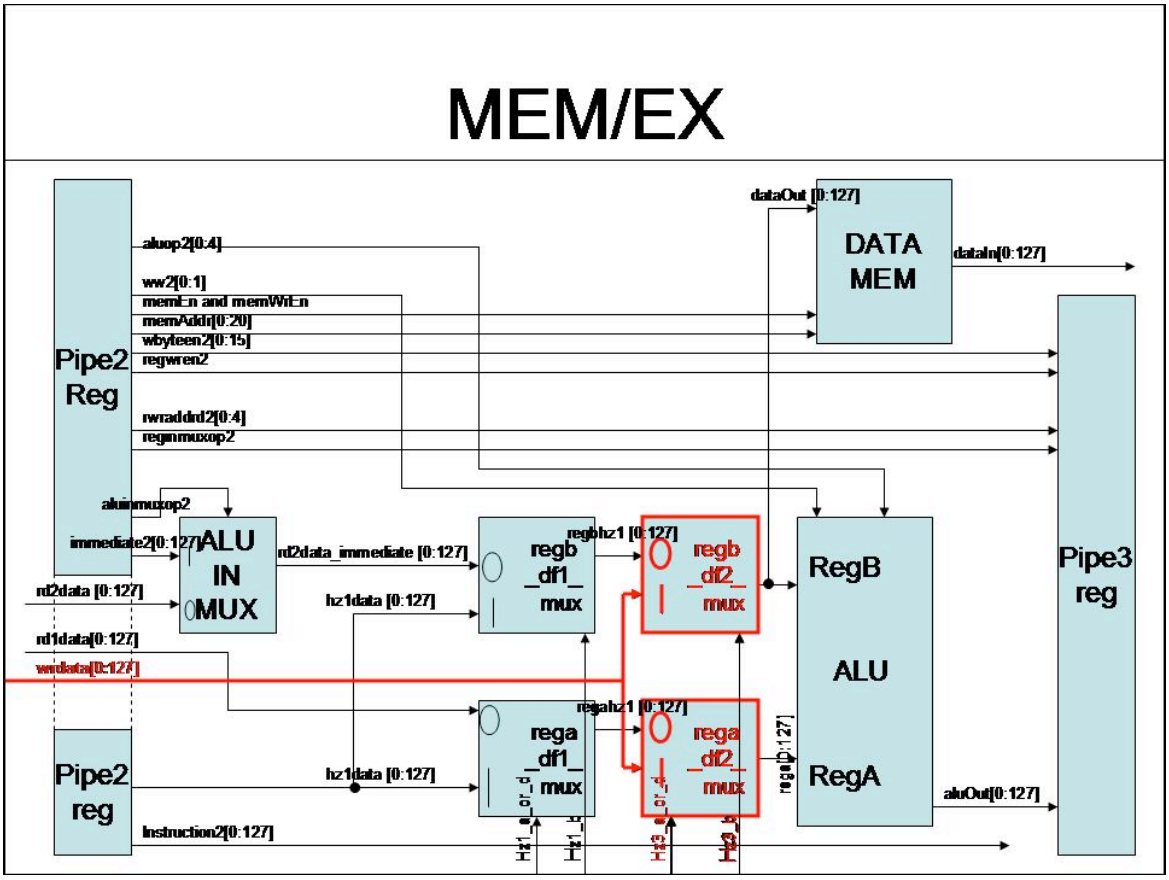


Data forwarding is used to overcome HZ1. The data paths used to forward data and avert HZ1 are highlighted in the figure below:





Data forwarding is used to overcome HZ3. The data paths used to forward data and avert HZ3 are highlighted in the figure below:

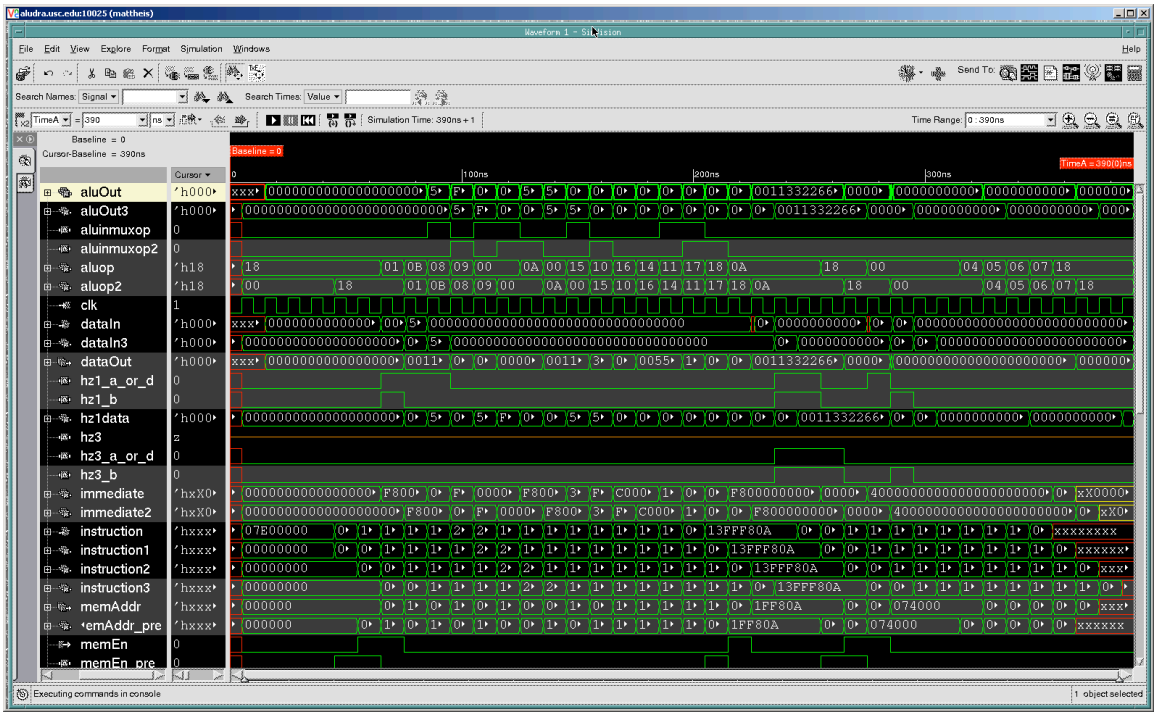

MEM/EX
Pipe2 Reg
aluop2[0:4]
ww2[0:1]
memEn and memWrEn
memAddr[0:20]
wbyteen2[0:15]
regwren2
rwraddrd2[0:4]
regnmuxop2
aluinmuxop2
immediate2[0:127]
ALU IN MUX
rd2data [0:127]
rd2data_immediate [0:127]
hz1data [0:127]
rd1data[0:127]
wrdata[0:127]
regb _df1_ mux
regbhz1 [0:127]
regb _df2_ mux
rega _df1_ mux
regahz1 [0:127]
rega _df2_ mux
Pipe2 reg
Instruction2[0:127]
dataOut [0:127]
DATA MEM
datain[0:127]
RegB
ALU
RegA
regab[0:127]
aluOut[0:127]
Pipe3 reg

7.1 Functional Test Results

From section 2, the following hazards exist:

HEX INSTR	BINARY INSTRUCTION	TYPE	REGD	REGA	REGB	HAZARD DETECTION SIGNALS
07E00000	000001_11111_00000_00000_00000000000	wld	wr31,		0	
04A00001	000001_00101_00000_00000_00000000001	wld	wr5,		1	
121FF881	000100_10000_11111_11111_00010000001	wsubaw	wr16,	wr31,	wr31	HZ1_a_or_d, HZ1_b
1225FA4B	000100_10001_00101_11111_01001001011	wxoruh	wr17,	wr5,	wr31	HZ1_a_or_d
11F00008	000100_01111_10000_00000_00000001000	wnotab	wr15,	wr16		HZ1_a_or_d
1245FC89	000100_10010_00101_11111_10010001001	wandew	wr18,	wr5,	wr31	
23100400	001000_11000_10000_00000_10000000000	wmveb	wr24,	wr16		
23050500	001000_11000_00101_00000_10100000000	wmvob	wr24,	wr5		
1265FD4A	000100_10011_00101_11111_10101001010	woroh	wr19,	wr5,	wr31	
129FFB40	000100_10100_11111_11111_01101000000	wadddh	wr20,	wr31,	wr31	
102F3815	000100_00001_01111_00111_00000010101	wsrliab	wr1,	wr15,	7	
12BFF850	000100_10101_11111_11111_00001010000	wsllah	wr21,	wr31,	wr31	
12DFC016	000100_10110_11111_11000_00000010110	wsraab	wr22,	wr31,	wr24	
12FFC094	000100_10111_11111_11000_00010010100	wsrlaw	wr23,	wr31,	wr24	
133F1011	000100_11001_11111_00010_00000010001	wslliab	wr25,	wr31,	2	
135F0857	000100_11010_11111_00001_00001010111	wsraiah	wr26,	wr31,	1	
0AA00002	000010_10101_00000_00000_00000000010	wst	wr21,		2	
13FFF80A	000100_11111_11111_11111_00000001010	worab	wr31,	wr31,	wr31	
13FFF80A	000100_11111_11111_11111_00000001010	worab	wr31,	wr31,	wr31	HZ1_a_or_d, HZ1_b, HZ3_a_or_d, HZ3_b
13FFF80A	000100_11111_11111_11111_00000001010	worab	wr31,	wr31,	wr31	HZ1_a_or_d, HZ1_b, HZ3_a_or_d, HZ3_b
04E00003	000001_00111_00000_00000_00000000011	wld	wr7		3	HZ3_a_or_d, HZ3_b
05000004	000001_01000_00000_00000_00000000100	wld	wr8		4	
13C74000	000100_11110_00111_01000_00000000000	addab	wr30	wr7	wr8	HZ1_a_or_d
13A74000	000100_11101_00111_01000_00000000000	addab	wr29	wr7	wr8	HZ1_b, HZ3_b
13874000	000100_11100_00111_01000_00000000000	addab	wr28	wr7	wr8	
13674000	000100_11011_00111_01000_00000000000	addab	wr27	wr7	wr8	
13C74044	000100_11110_00111_01000_00001000100	mules	wr30	wr7	wr8	
13A74045	000100_11101_00111_01000_00001000101	muleu	wr29	wr7	wr8	
13874046	000100_11100_00111_01000_00001000110	mulos	wr28	wr7	wr8	
13674047	000100_11011_00111_01000_00001000111	mulou	wr27	wr7	wr8	
00000000	000000_00000_00000_00000_00000000000					

From section 2, the following waveform was generated depicting the hazard detection signals hz1_a_or_d, hz1_b, hz3_a_or_d, and hz3_b. It can be seen that the waveform matches the expected result above.



8 Optimization

ME

9 Synthesis Report

ME
(nicely compiled not the output of the tool)

10 Final specifications of your designs

ME
(claims)

11 Possible Future Enhancements

If more time was allowed to complete this project, we would have investigated other multiplication algorithms and even pipelined the multiplier because it is the slowest function of the ALU. If the multiply operation took multiple clocks to accomplish and all most other instructions only took a single clock in the MEM/EX stage, then overall the program's execute time for the benchmarks would decrease because the clock frequency that controls the pipeline could be increased.

12 Conclusion

Whatever you do, reduce the workload for the next class you teach – this semester required more hours per week for school than previous semesters when I took 2 classes, and was miserable - seriously. I am so glad it is finally over.

13 References