LAB 10:

Final Project: The RISC-Y Processor

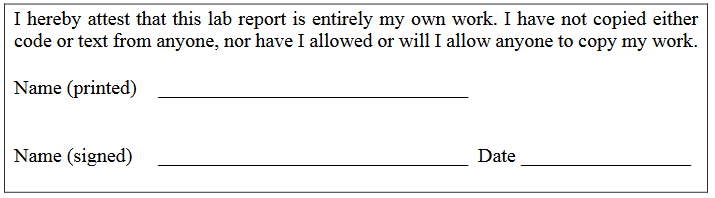
Professor Ronald Mehler

ECE 526L

Spring 2018

Garen Nikoyan

5/10/2018



**Objective:**

The objective of this lab is to create a complete microcontroller, consisting of almost every other lab that was throughout the semester. This microcontroller can be referred to as a RISC-Y processor, since it has a reduced instruction set.

**Methodology:**

Most of the top level module was instantiated modules from previous labs, such as the ROM, RAM, scalable register, and counter. These modules were used to create five different subsystems, each with a specific purpose.

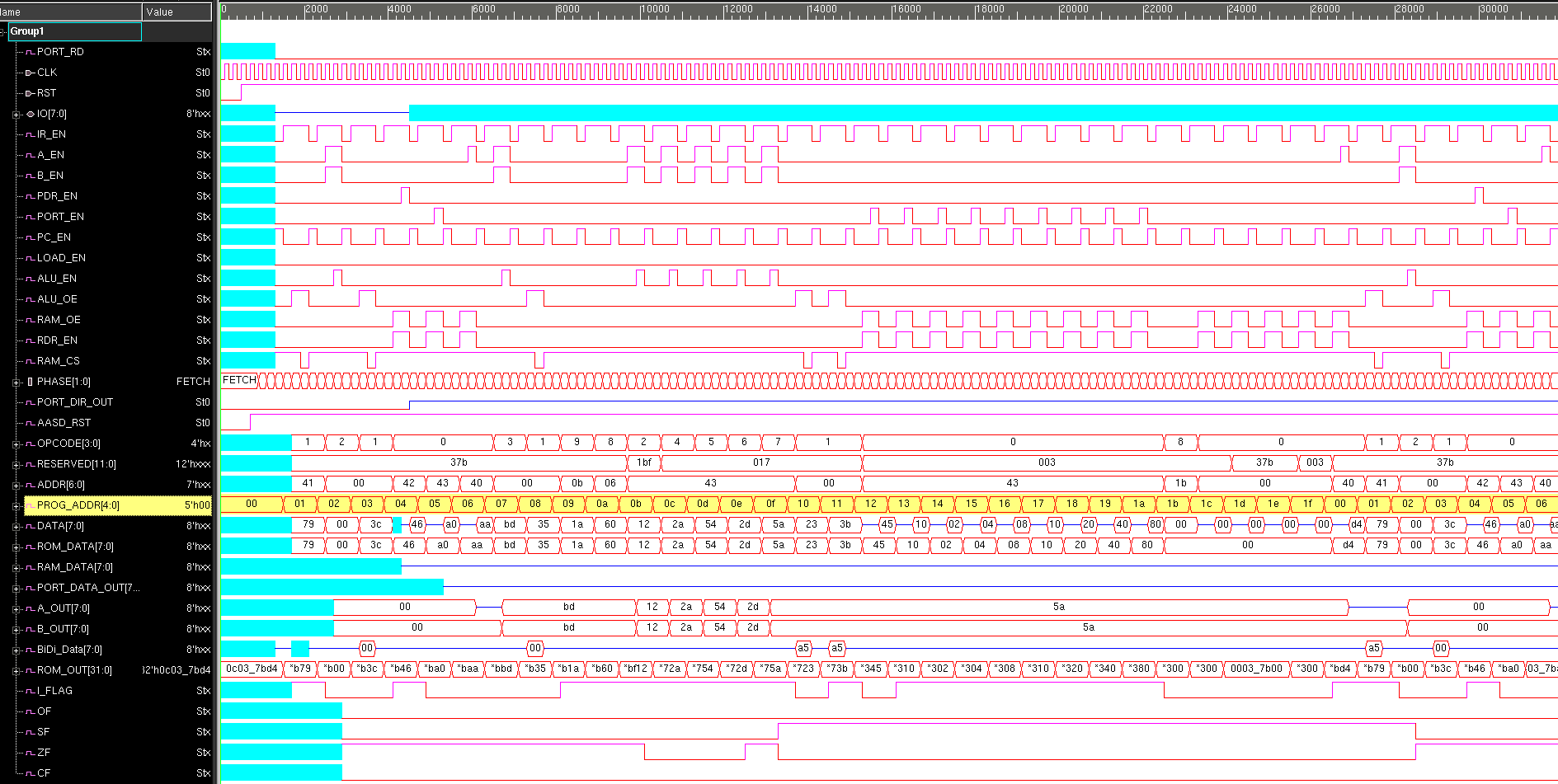
The first subsystem is the memory subsystem. This subsystem consists of the Program counter, ROM, memory instruction register, RAM, RAM data register, and a bidirectional data bus. This subsystem is one of the most important, as it passes the opcodes, addresses, and rom data to the rest of the module. The second subsystem is the IO subsystem, which consists of Port direction register, Port Data Register, two buffers, and an IO data bus. This takes the ROM data into the Port Data Register, and outputs the Port Read Data to the RAM. The third subsystem is the ALU, which consists of two registers, for A and B data, and the ALU. This subsystem is important because it outputs the conditional flags, which are important for branching properly. The Memory to Data Bus subsystem is probably the simplest. It is just a multiplexer that takes in ROM data and RAM data, and outputs ROM data if RAM OE is low, and RAM data if RAM OE is high. The last subsystem is the most important, the sequence controller subsystem. It consists of the phase generator and sequence controller from Lab 9. This will take the opcode, address, and flags, and decide what should be done. If there is an error in this, it could easily break the whole processor.

The test bench was made to test a few opcodes, most importantly, the branches. In my experience, branching is a very powerful tool, and a processor without it would be severely handicapped.

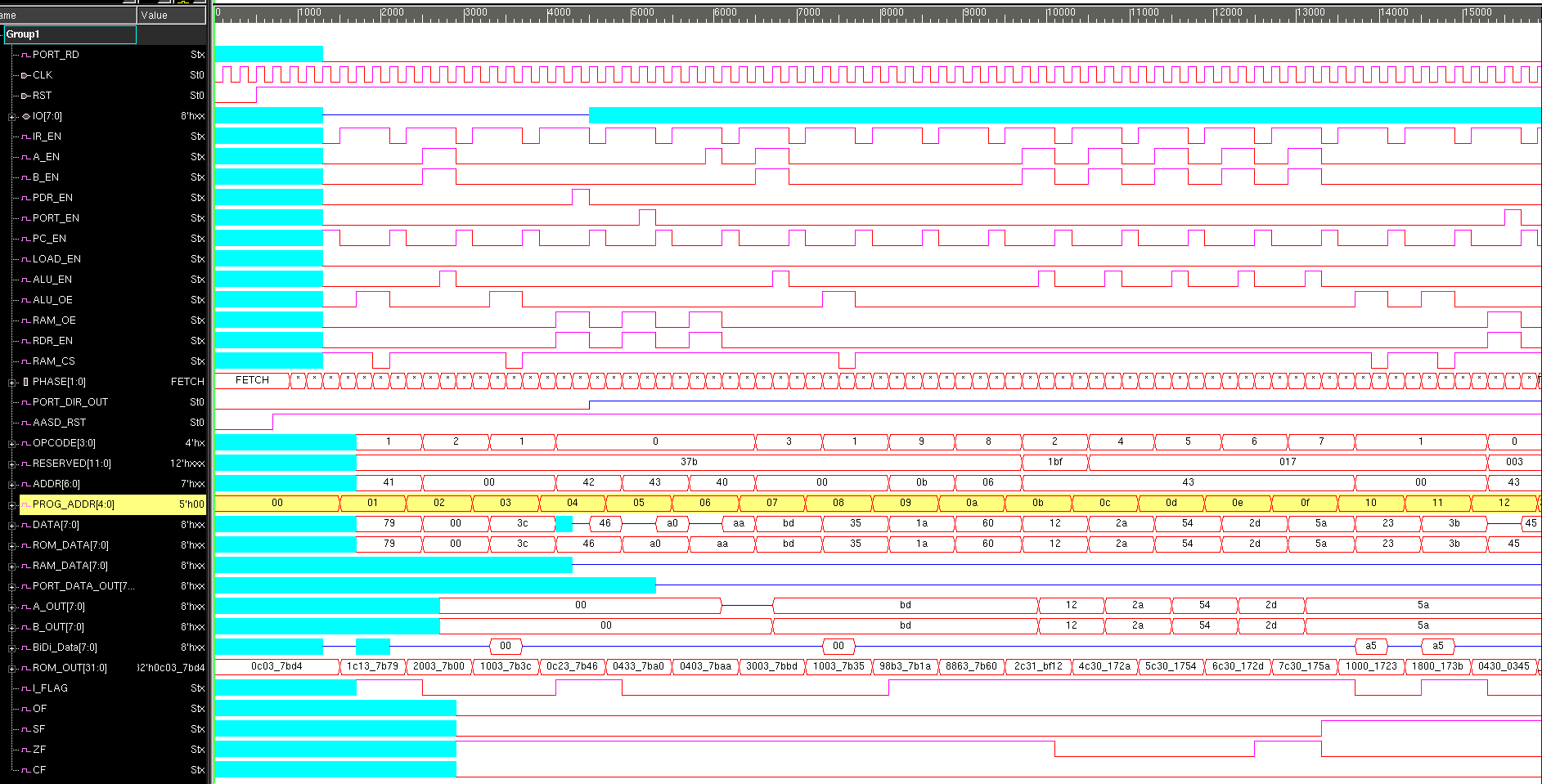
**Analysis:**

After countless hours of troubleshooting, I was not able to make my processor work correctly. There seems to have been an error in the sequence controller, as well the memory and IO subsystems, causing the bidirectional data, RAM data, and Port Direction (PORT\_DIR\_OUT) lines to Z. Besides these errors, there was also an error in the branching. The address pointer for the ROM did not properly branch. In hindsight, this might have been an error in the way the flags were set.

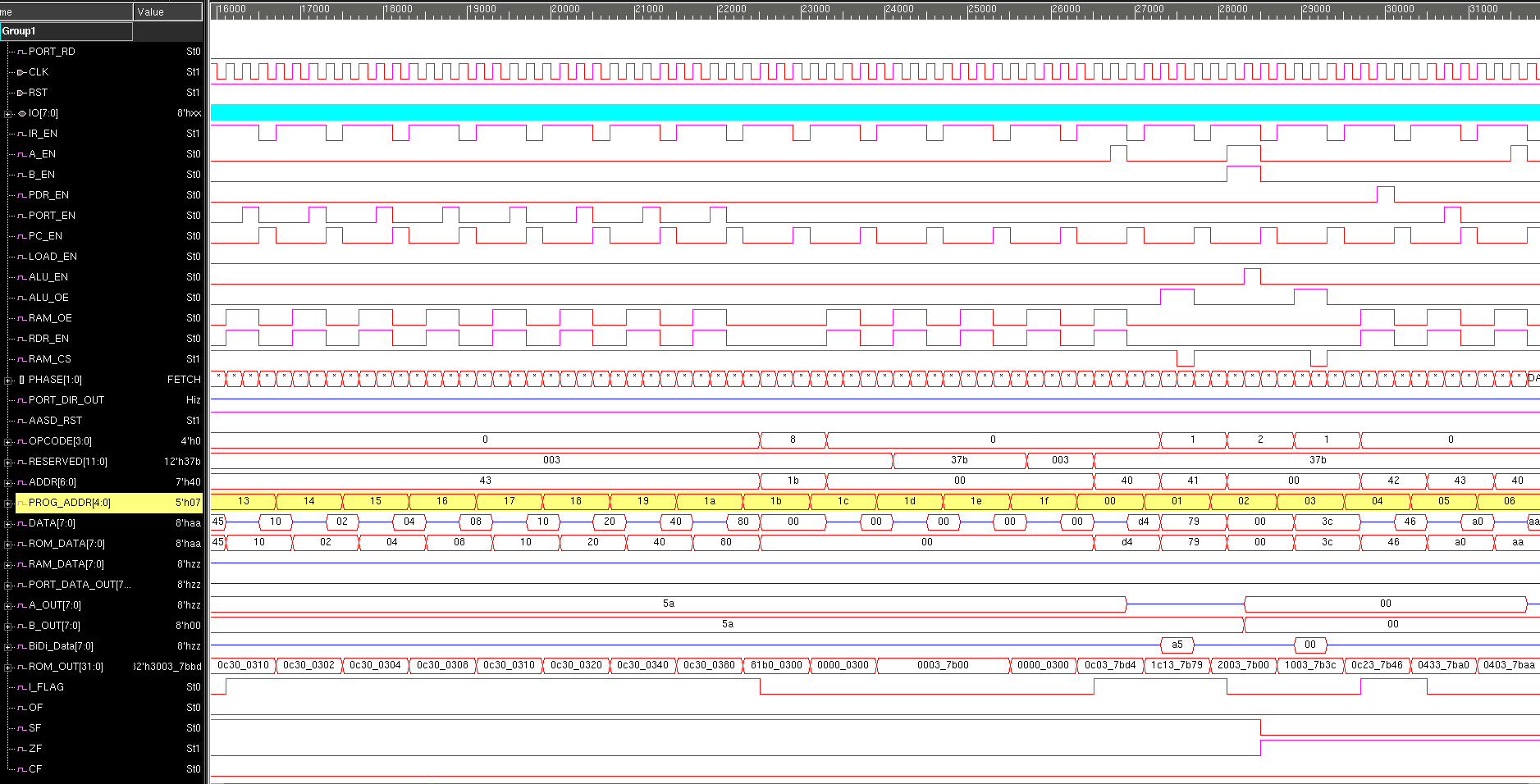
Log files were not included as I only used the waveforms to debug my circuit.



*Figure 1: Full Waveforms*

**

*Figure 2: Zoomed in, first half*

**

*Figure 3: Zoomed in, second half*

**Modules:**

**RISCY\_CPU.sv**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\* ECE526L Experiment #10 Garen Nikoyan, Spring 2018

\*\*\* Final Project: The RISC-Y Processor

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\* Filename: RISCY\_CPU.sv Created by: Garen Nikoyan, 5/2/2018 \*\*\*

\*\*\* -Revision History

\*\*\* 5/2/2018: First draft

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\* This module instantiates various modules from previous labs in and

\*\*\* attempt to model a RISC processor. There are five main subsystems:

\*\*\* Memory, IO, ALU, Memory to Data Bus, and Sequence Controller.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

`timescale 1ns/100ps

`default\_nettype none

package cycle\_package;

typedef enum [1:0] {FETCH = 0, DECODE = 1, EXECUTE = 2, UPDATE = 3} CYCLE;

endpackage

// \*\*\*\*\*\*\*\*\*\* Top Level CPU

module RISCY(CLK,RST,IO);

import cycle\_package::\*;

input CLK, RST;

inout [7:0] IO;

wire IR\_EN, A\_EN, B\_EN, PDR\_EN, PORT\_EN, PORT\_RD, PC\_EN, LOAD\_EN, ALU\_EN, ALU\_OE, RAM\_OE, RDR\_EN, RAM\_CS;

CYCLE PHASE;

wire PORT\_DIR\_OUT, AASD\_RST;

wire [3:0] OPCODE;

wire [11:0] RESERVED;

wire [6:0] ADDR;

wire [4:0] PROG\_ADDR;

wire [7:0] DATA, ROM\_DATA, RAM\_DATA, PORT\_DATA\_OUT, A\_OUT, B\_OUT;

wire [7:0] BiDi\_Data, ALU\_DATA;

wire [31:0] ROM\_OUT;

wire I\_FLAG, OF, SF, ZF, CF;

// Memory Subsystem

RAM RAM8(.DATA(BiDi\_Data), .ADDR(ROM\_DATA[4:0]), .OE(RAM\_OE), .CS(RAM\_CS), .WS(CLK));

SCALE\_REG #(8) REG8(.REG\_OUT(RAM\_DATA), .DATA\_IN(BiDi\_Data), .EN(RDR\_EN), .CLOCK(CLK));

ROM ROM32(.DATA(ROM\_OUT), .ADDR(PROG\_ADDR), .OE(1'b1), .CS(1'b0));

ProgCount #(5) PrgCnt(.Count(PROG\_ADDR), .CLOCK(CLK), .RESET(AASD\_RST), .Enable(PC\_EN), .Load(LOAD\_EN), .Data(ADDR[4:0]));

SCALE\_REG #(32) REG32(.REG\_OUT({OPCODE,I\_FLAG, ADDR, RESERVED, ROM\_DATA}), .DATA\_IN(ROM\_OUT), .EN(IR\_EN), .CLOCK(CLK));

// IO Subsystem

PORT\_DIR\_REG PDR(.REG\_OUT(PORT\_DIR\_OUT), .DATA\_IN(DATA[0]), .PDR\_EN(PDR\_EN),.CLOCK(CLK), .RESET(AASD\_RST));

SCALE\_REG #(8) IOREG(.REG\_OUT(PORT\_DATA\_OUT), .DATA\_IN(DATA), .EN(PORT\_EN), .CLOCK(CLK));

Tri\_State\_Buffer Port\_Data(.OUT(IO), .A(PORT\_DATA\_OUT), .SEL(PORT\_DIR\_OUT));

Tri\_State\_Buffer Port\_Read(.OUT(BiDi\_Data), .A(IO), .SEL(PORT\_RD));

// ALU Subsystem

SCALE\_REG #(8) A\_Reg(.REG\_OUT(A\_OUT), .DATA\_IN(DATA), .EN(A\_EN), .CLOCK(CLK));

SCALE\_REG #(8) B\_Reg(.REG\_OUT(B\_OUT), .DATA\_IN(DATA), .EN(B\_EN), .CLOCK(CLK));

ALU #(8) ALU0(.CLK(CLK), .EN(ALU\_EN), .OE(ALU\_OE), .OPCODE(OPCODE), .A(A\_OUT), .B(B\_OUT), .ALU\_OUT(BiDi\_Data), .CF(CF), .OF(OF), .SF(SF), .ZF(ZF));

// Memory to Data Bus Subsystem

scale\_mux #(8) MUX(.A(ROM\_DATA), .B(RAM\_DATA), .SEL(RAM\_OE), .OUT(DATA));

// Sequence Controller Subsystem

AASD AASD0(.AASD\_RST(AASD\_RST), .CLOCK(CLK), .RESET(RST));

phaser Phsr(.CLOCK(CLK), .RESET(AASD\_RST), .EN(1'b0), .PHASE(PHASE));

sequence\_controller SeqCtrl(.ADDR(ADDR), .OPCODE(OPCODE), .PHASE(PHASE), .I\_FLAG(I\_FLAG), .ZF(ZF), .NF(SF), .OF(OF), .CF(CF), .IR\_EN(IR\_EN), .A\_EN(A\_EN), .B\_EN(B\_EN), .PDR\_EN(PDR\_EN), .PORT\_EN(PORT\_EN), .PORT\_RD(PORT\_RD), .PC\_EN(PC\_EN), .PC\_LOAD(LOAD\_EN), .ALU\_EN(ALU\_EN), .ALU\_OE(ALU\_OE), .RAM\_OE(RAM\_OE), .RDR\_EN(RDR\_EN), .RAM\_CS(RAM\_CS));

endmodule

**sequence\_controller.sv**

`timescale 1ns/100ps

`default\_nettype none

package cycle\_package;

typedef enum [1:0] {FETCH =0, DECODE = 1, EXECUTE = 2, UPDATE = 3} CYCLE;

endpackage

module sequence\_controller(ADDR,OPCODE,PHASE,I\_FLAG,ZF,NF,OF,CF,IR\_EN,A\_EN,B\_EN,PDR\_EN,PORT\_EN,PORT\_RD,PC\_EN,PC\_LOAD,ALU\_EN,ALU\_OE,RAM\_OE,RDR\_EN,RAM\_CS);

import cycle\_package::\*;

input [6:0] ADDR;

input [3:0] OPCODE;

input CYCLE PHASE;

input I\_FLAG,ZF,NF,OF,CF;

output reg IR\_EN,A\_EN,B\_EN,PDR\_EN,PORT\_EN,PORT\_RD,PC\_EN,PC\_LOAD,ALU\_EN,ALU\_OE,RAM\_OE,RDR\_EN,RAM\_CS;

reg branches;

// Instruction set

localparam LOAD = 0,

STORE = 1,

ADD = 2,

SUB = 3,

AND = 4,

OR = 5,

XOR = 6,

NOT = 7,

B = 8,

BZ = 9,

BN = 10,

BV = 11,

BC = 12;

always @(PHASE,I\_FLAG,ZF,NF,OF,CF,ADDR,OPCODE)

case (PHASE)

// FETCH

FETCH:

begin

IR\_EN=1'b1;

PC\_EN=1'b0;

end

// DECODE

DECODE:

begin

case (OPCODE)

LOAD:

begin

RAM\_OE <= 1'b1;

RAM\_CS <= 1'b1;

RDR\_EN <= 1'b1;

end

STORE:

begin

if(ADDR==67) PORT\_RD <= 1'b1;

else ALU\_OE <= 1'b1;

end

ADD:

begin

A\_EN = 1'b1;

B\_EN = 1'b1;

end

SUB:

begin

A\_EN = 1'b1;

B\_EN = 1'b1;

end

AND:

begin

A\_EN = 1'b1;

B\_EN = 1'b1;

end

OR:

begin

A\_EN = 1'b1;

B\_EN = 1'b1;

end

XOR:

begin

A\_EN = 1'b1;

B\_EN = 1'b1;

end

NOT:

begin

A\_EN = 1'b1;

B\_EN = 1'b1;

end

endcase

end

// EXECUTE

EXECUTE:

begin

case (OPCODE)

LOAD:

begin

case (ADDR)

64: A\_EN = 1'b1;

65: B\_EN =1'b1;

66: PDR\_EN <=1'b1;

67: PORT\_EN <=1'b1;

endcase

end

STORE:

begin

RAM\_CS <= 1'b0;

end

ADD:

begin

ALU\_EN <=1'b1;

end

SUB:

begin

ALU\_EN <=1'b1;

end

AND:

begin

ALU\_EN <=1'b1;

end

OR:

begin

ALU\_EN <=1'b1;

end

XOR:

begin

ALU\_EN <=1'b1;

end

NOT:

begin

ALU\_EN <=1'b1;

end

endcase

end

// UPDATE

UPDATE: begin

case(OPCODE)

B: PC\_LOAD=1'b1;

BZ: PC\_LOAD=ZF;

BN: PC\_LOAD=NF;

BV: PC\_LOAD=OF;

BC: PC\_LOAD=CF;

default: PC\_LOAD=1'b0;

endcase

IR\_EN <= 1'b0;

A\_EN = 1'b0;

B\_EN = 1'b0;

PDR\_EN <= 1'b0;

PORT\_EN <= 1'b0;

PORT\_RD <= 1'b0;

ALU\_EN <= 1'b0;

ALU\_OE <= 1'b0;

RAM\_OE <= 1'b0;

RDR\_EN <= 1'b0;

PC\_EN=1'b1;

PC\_LOAD=1'b0;

RAM\_CS <= 1'b1;

end

// DEFAULT

default: begin

IR\_EN <= 1'b0;

A\_EN = 1'b0;

B\_EN = 1'b0;

PDR\_EN <= 1'b0;

PORT\_EN <= 1'b0;

PORT\_RD <= 1'b0;

ALU\_EN <= 1'b0;

ALU\_OE <= 1'b0;

RAM\_OE <= 1'b0;

RDR\_EN <= 1'b0;

PC\_EN=1'b0;

PC\_LOAD=1'b0;

RAM\_CS <= 1'b1;

end

endcase

endmodule

**phaser.sv**

`timescale 1ns/100ps

`default\_nettype none

package cycle\_package;

typedef enum [1:0] {FETCH =0, DECODE = 1, EXECUTE = 2, UPDATE = 3} CYCLE;

endpackage

module phaser(CLOCK, RESET, EN, PHASE);

import cycle\_package::\*;

input CLOCK, RESET, EN;

output CYCLE PHASE;

always @(posedge CLOCK, negedge RESET) begin

if(!RESET)

PHASE <= PHASE.first();

else if(!EN)

PHASE <= PHASE.next();

else

PHASE <= PHASE;

end

endmodule

**AASD.sv**

`timescale 1ns/100ps

`default\_nettype none

package cycle\_package;

typedef enum [1:0] {FETCH =0, DECODE = 1, EXECUTE = 2, UPDATE = 3} CYCLE;

endpackage

module AASD(AASD\_RST,CLOCK,RESET);

output reg AASD\_RST;

input CLOCK,RESET;

reg Out1; //this is the wire for the output of the first FF

always @ (posedge CLOCK or negedge RESET) //allows 4 synchronous/asynchronous

if(!RESET)begin//asynchronous reset

Out1 <= 1'b0;

AASD\_RST <= 1'b0;

end

else begin

Out1 <= 1'b1;

AASD\_RST <= Out1;

end

endmodule

**RAM.sv**

`timescale 1ns/100ps

`default\_nettype none

package cycle\_package;

typedef enum [1:0] {FETCH =0, DECODE = 1, EXECUTE = 2, UPDATE = 3} CYCLE;

endpackage

module RAM(DATA, ADDR, OE, CS, WS);

inout [7:0] DATA;

input [4:0] ADDR;

input OE, CS, WS;

reg [7:0] memory [0:31];

// if OE=1 and CS=0, DATA=memory at ADDR; if not, DATA=z

// assign is used because of inout type

assign DATA = (OE && !CS) ? memory[ADDR] : 8'bz;

always@(posedge WS) begin

if (!OE && !CS)

memory[ADDR] <= DATA;

else

memory[ADDR] <= memory[ADDR]; // to prevent latches in hardware

end

endmodule

**ROM.sv**

`timescale 1ns/100ps

`default\_nettype none

package cycle\_package;

typedef enum [1:0] {FETCH =0, DECODE = 1, EXECUTE = 2, UPDATE = 3} CYCLE;

endpackage

module ROM(DATA, ADDR, OE, CS);

output reg [31:0] DATA;

input wire [4:0] ADDR;

input OE, CS;

reg [31:0] memory [0:31];

// if OE=1 and CS=0, DATA=memory at ADDR; if not, DATA=z

always@\* DATA = (OE && !CS) ? memory[ADDR] : 32'bz;

endmodule

**SCALE\_REG.sv**

`timescale 1ns/100ps

`default\_nettype none

package cycle\_package;

typedef enum [1:0] {FETCH =0, DECODE = 1, EXECUTE = 2, UPDATE = 3} CYCLE;

endpackage

module SCALE\_REG(REG\_OUT,DATA\_IN,EN,CLOCK);

parameter RegSize = 1;

output reg [ RegSize - 1 : 0] REG\_OUT;

input reg [ RegSize - 1 : 0] DATA\_IN;

input CLOCK,EN;

always @ (posedge CLOCK) begin

if(EN)

REG\_OUT <= DATA\_IN;

else

REG\_OUT <= REG\_OUT;

end

endmodule

**PORT\_DIR\_REG.sv**

`timescale 1ns/100ps

`default\_nettype none

package cycle\_package;

typedef enum [1:0] {FETCH =0, DECODE = 1, EXECUTE = 2, UPDATE = 3} CYCLE;

endpackage

module PORT\_DIR\_REG(REG\_OUT,DATA\_IN,PDR\_EN,CLOCK,RESET);

output reg REG\_OUT;

input reg DATA\_IN; // DATA[0]

input CLOCK,RESET,PDR\_EN;

always @ (posedge CLOCK or negedge RESET) begin

if(PDR\_EN)

REG\_OUT <= DATA\_IN;

else if (!RESET)

REG\_OUT <= 0;

else

REG\_OUT <= REG\_OUT;

end

endmodule

**Tri\_State\_Buffer.sv**

`timescale 1ns/100ps

`default\_nettype none

package cycle\_package;

typedef enum [1:0] {FETCH =0, DECODE = 1, EXECUTE = 2, UPDATE = 3} CYCLE;

endpackage

module Tri\_State\_Buffer(OUT, A, SEL);

input [7:0] A;

input SEL;

inout [7:0] OUT;

assign OUT = SEL ? A : 8'bz; //first tri state buffer

endmodule

**ProgCount.sv**

`timescale 1ns/100ps

`default\_nettype none

package cycle\_package;

typedef enum [1:0] {FETCH =0, DECODE = 1, EXECUTE = 2, UPDATE = 3} CYCLE;

endpackage

module ProgCount(Count, CLOCK, RESET, Enable, Load, Data);

parameter PCSize = 1;

output reg [PCSize-1:0] Count;

input [PCSize-1:0] Data;

input CLOCK, Enable, RESET, Load;

always @ (posedge CLOCK or negedge RESET)

if(!RESET)

Count <= 0;

else if(Enable)

begin

if(Load)

Count <= Data;

else

Count <= Count+ 1;

end

endmodule

**scale\_mux.sv**

`timescale 1ns/100ps

`default\_nettype none

package cycle\_package;

typedef enum [1:0] {FETCH =0, DECODE = 1, EXECUTE = 2, UPDATE = 3} CYCLE;

endpackage

module scale\_mux(A, B, SEL, OUT);

parameter Size = 1;

output reg [Size-1:0] OUT;

input [Size-1:0] A, B;

input SEL;

always @(SEL, A, B)

OUT = SEL ? B : A; // if SEL=1, OUT=B, if SEL=0, OUT=A

// if SEL=x, then A=B causes OUT=A=B, and A!=B, OUT=x

endmodule

**ALU.sv**

`timescale 1ns/100ps

`default\_nettype none

package cycle\_package;

typedef enum [1:0] {FETCH =0, DECODE = 1, EXECUTE = 2, UPDATE = 3} CYCLE;

endpackage

module ALU(CLK, EN, OE, OPCODE, A, B, ALU\_OUT, CF, OF, SF, ZF);

parameter WIDTH = 8;

output reg [ WIDTH-1 : 0] ALU\_OUT;

output reg CF, OF, SF, ZF;

input [3:0] OPCODE;

input [ WIDTH-1 : 0] A, B;

input reg CLK, EN, OE;

reg [ WIDTH-1 : 0] temp;

localparam ADDalu = 4'b0010,

SUBalu = 4'b0011,

ANDalu = 4'b0100,

ORalu = 4'b0101,

XORalu = 4'b0110,

NOTalu = 4'b0111;

always @(OE, temp) ALU\_OUT = (OE) ? temp : 8'bz;

always @(posedge CLK) begin

if(EN) begin

case (OPCODE)

ADDalu: begin

temp = A + B;

// Carry Flag

if (A+B> 2\*\*WIDTH -1) CF=1;

else CF=0;

// Signed Flag

if (temp[WIDTH-1]==1) SF=1;

else SF=0;

// Zero Flag

if(temp==0) ZF=1;

else ZF=0;

// Overflow Flag

if(A>0 && B>0 && temp<0) OF=1;

else if (A<0 && B<0 && temp>0) OF=1;

else OF=0;

end

SUBalu: begin

temp = A - B;

// Carry Flag

if (A<B) CF=1;

else CF=0;

// Signed Flag

if (temp[WIDTH-1]==1) SF=1;

else SF=0;

// Zero Flag

if(temp==0) ZF=1;

else ZF=0;

// Overflow Flag

if(A>0 && B>0 && temp<0) OF=1;

else if (A<0 && B<0 && temp>0) OF=1;

else OF=0;

end

ANDalu: begin

temp = A&B;

// Signed Flag

if(temp[WIDTH-1]==1) SF=1;

else SF=0;

// Zero Flag

if(temp==0) ZF=1;

else ZF=0;

end

ORalu: begin

temp = A|B;

// Signed Flag

if(temp[WIDTH-1]==1) SF=1;

else SF=0;

// Zero Flag

if(temp==0) ZF=1;

else ZF=0;

end

XORalu: begin

temp = A^B;

// Signed Flag

if(temp[WIDTH-1]==1) SF=1;

else SF=0;

// Zero Flag

if(temp==0) ZF=1;

else ZF=0;

end

NOTalu: begin

temp = ~A;

// Signed Flag

if(temp[WIDTH-1]==1) SF=1;

else SF=0;

// Zero Flag

if(temp==0) ZF=1;

else ZF=0;

end

default: temp = 1'bx;

endcase

end

end

endmodule

**Testbench:**

**RISCY\_TB.sv**

`timescale 1ns/100ps

module RISCY\_TB();

reg CLK, RST;

wire [7:0] IO;

reg [7:0] Data\_In;

integer i;

RISCY UUT(CLK, RST, IO);

initial CLK = 1'b0;

always #10 CLK = !CLK;

assign IO = UUT.PORT\_RD ? Data\_In: 'bz;

initial begin

$vcdpluson;

$readmemh("ROM\_DATA.txt", UUT.ROM32.memory);

end

initial begin

RST = 1'b0; Data\_In = 8'h55;

#50 RST = 1'b1;

#2000

$display("\t RAM Address || Contents");

for(i=0; i<32; i=i+1) begin

#20 $display("\t %2h | %2h ", i, UUT.ROM32.memory[i]);

end

$finish;

end

endmodule

**ROM\_DATA.txt**

00001100000000110111101111010100

00011100000100110111101101111001

00100000000000110111101100000000

00010000000000110111101100111100

00001100001000110111101101000110

00000100001100110111101110100000

00000100000000110111101110101010

00110000000000110111101110111101

00010000000000110111101100110101

10011000101100110111101100011010

10001000011000110111101101100000

00101100001100011011111100010010

01001100001100000001011100101010

01011100001100000001011101010100

01101100001100000001011100101101

01111100001100000001011101011010

00010000000000000001011100100011

00011000000000000001011100111011

00000100001100000000001101000101

00001100001100000000001100010000

00001100001100000000001100000010

00001100001100000000001100000100

00001100001100000000001100001000

00001100001100000000001100010000

00001100001100000000001100100000

00001100001100000000001101000000

00001100001100000000001110000000

10000001101100000000001100000000

00000000000000000000001100000000

00000000000000110111101100000000

00000000000000110111101100000000

00000000000000000000001100000000

**Conclusion:**

This lab, besides being the most difficult, was also the most enlightening. Trying to instantiate so many modules while keeping track of all the connections proved to be very difficult, and tracking down errors required a very deep understanding of not only the hardware description, but also of what was expected. In conclusion, this lab tied what I think was the most important takeaways from this class all into one, by including such a large number of modules into one design.