Lab B

Processor Design

TCES 330 Digital Systems Design

Spring 2015

Authors:

James Brewer \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Jared Herdlevar \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Submission Date: May 6, 2015

Table of Contents

[Lab B 3](#_Toc421273661)

[Requirements 3](#_Toc421273662)

[Processor Architecture 3](#_Toc421273663)

[Datapath Module 3](#_Toc421273664)

[Data RAM LPM 3](#_Toc421273665)

[Register File (RF) 4](#_Toc421273666)

[ALU 4](#_Toc421273667)

[Control Unit 4](#_Toc421273668)

[Instruction ROM LPM 4](#_Toc421273669)

[State Machine Controller 5](#_Toc421273670)

[Program Execution 5](#_Toc421273671)

[Design 6](#_Toc421273672)

[Datapath Design 6](#_Toc421273673)

[Control Unit 7](#_Toc421273674)

[Observations 9](#_Toc421273675)

[*Control Unit* 9](#_Toc421273676)

[PC 9](#_Toc421273677)

[IR 9](#_Toc421273678)

[ROM 9](#_Toc421273679)

[Controller 9](#_Toc421273680)

[*Datapath* 10](#_Toc421273681)

[RAM 10](#_Toc421273682)

[16-bit 2-to-1 MUX 10](#_Toc421273683)

[Register File 10](#_Toc421273684)

[ALU 11](#_Toc421273685)

[Conclusion 11](#_Toc421273686)

[Appendix 12](#_Toc421273687)

[Top-Level Module 12](#_Toc421273688)

[Level 1 Modules 13](#_Toc421273689)

[Level 2 Modules 15](#_Toc421273690)

[Level 3 Modules 17](#_Toc421273691)

[Level 4 Modules 26](#_Toc421273692)

Figure 1 – Processor structure overview 3

Figure 2 – The outline of the six instructions 4

Figure 3 – An outline of a general controller state machine 5

Figure 4 – The detailed connections of the various modules in the processor 6

Figure 5 – Compilation Report 9

Figure 6 – State machine viewer results. 11

# Lab B

The purpose of this laboratory assignment is to experience designing a processor from several hierarchal components, including a controller, datapath, and their internal modules.

## Requirements

### Processor Architecture

The processor should include six available instructions to operate on values stored in RAM. The overall processor should include a hierarchy of modules as illustrated below.

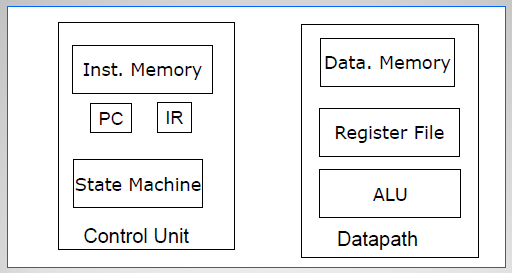


Figure – Processor structure overview.

### Datapath Module

The datapath module needed to consist of a 256 x 16-bit RAM LPM, a data write 2-1 MUX, a 16 X 16-bit Register File, and an ALU consisting of various instructions. The datapath needed several input connections to communicate with the Control Unit and various debug outputs.

### Data RAM LPM

The RAM needed to be initialized through the Quartus MegaWizard. Only one address port was needed to either read or write an address in memory, to be controlled by a read enable or a write enable. Initialization of the RAM would come from a memory initialization file (MIF) to be filled as needed.

The RAM output leads to a 2-to-1 16-bit wide MUX that would send data to the Register File.

### Register File (RF)

The register file was implemented from Lab 5, which itself was converted to be a 16 x 16-bit set of output enabled registers, and writing to the registers was essentially the same as writing to RAM (choosing an address and then enabling the module to write to the register)

### ALU

The ALU was again implemented from Lab 5. For the purpose of this lab, the ALU consisted of 8 different operations including: Null, Add, Subtract, Pass, XOR, OR, AND, and Plus 1. The ALU would then output back to the RF input mux to be stored in a register.

### Control Unit

The control unit module needed to consist of an instruction set ROM module, a PC register, an Instruction register (IR), and a controller designed as a state machine. The control unit would then control the various address and enable ports of the datapath.

### Instruction ROM LPM

The ROM module would be implemented as a MegaWizard function like the RAM module, to be initialized with its own MIF. It would take an address from the PC, and then output that instruction to the IR. The module would be created with 32 X 16-bit storage, therefore being able to store up to 32 instructions. The instructions would be of the form shown in the figure below:

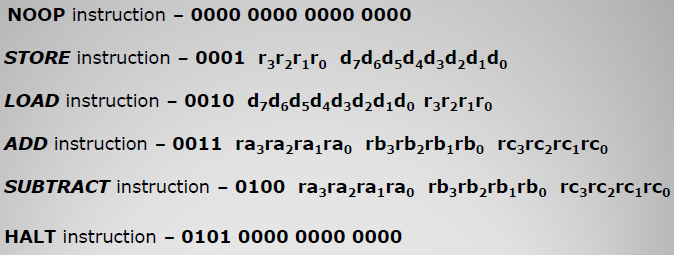


Figure – The outline of the six instructions.

### State Machine Controller

The last module of the control unit would need to be a state machine that changes the outputs of the control unit based on the instructions given in the IR. At least one state is needed for each of the six instructions, as well as a state to fetch the instruction from the IR, and another state to decode it. One of the instructions (LOAD) was described to need two clock cycles to operate properly, so two Load states would be needed to complete the instruction. A basic conceptual diagram of the state machine is shown below. The full state machine sets various outputs to the datapath based on the instructions in the IR, or be default for the state.

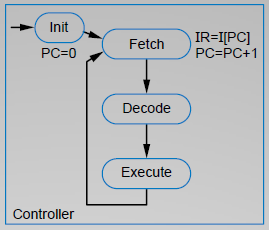


Figure – An outline of a general controller state machine. The decode state would output to as many execute states as there are instructions.

### Program Execution

As per the lab write-up, we were to execute the following program with the processor using available means:

RF[0] = D[A] - D[1A] + D[3] - D[8A]; //Adding and Subtracting

D[BB] = RF[0]; //Storing a value in RAM

HALT //Halting the processor

The data RAM should then include the following values at the given addresses:

D[3] = 0x10AA

D[A] = 0xB0C5

D[1A] = 0x00DC

D[8A] = 0x00E9

## Design

For the design of the processor, we decided to tackle the creation of modules and build on the project from there, starting with the datapath (which uses previous files) and then the control unit, the overall processor and then the board interface. We followed the general connection design from the lecture slides going through the structure of the processor, as shown in the next figure.

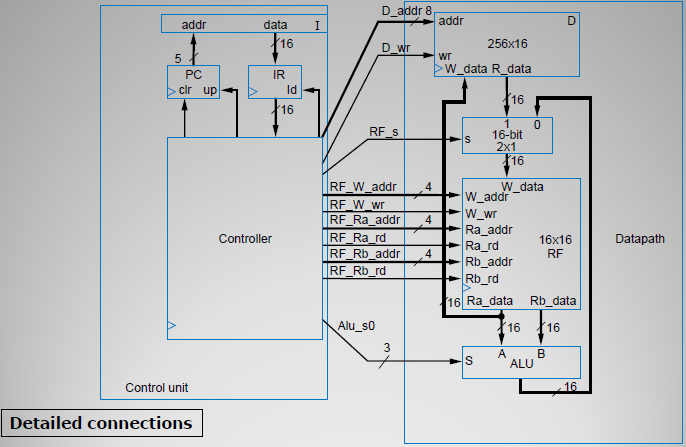


Figure – The detailed connections of the various modules in the processor.

### Datapath Design

We started by creating the RAM LPM through the MegaWizard (ramlpm.v). Following the design in the lecture slides, it’s a 1-port module due to only having one address input for either reading or writing to or from memory (D\_addr). It has an enable port (D\_wr) for writing to memory from the write port (W\_data), and a direct connection output for reading data (R\_data). We then created the MIF (ram.mif) with the necessary values. MIFs appear to use decimal values, so we needed to convert the hexadecimal data to decimal.

We also needed to create a 16-bit wide 2-to-1 MUX (Mux16w2\_1.v) for controlling data input to the RF. We implemented this by using a generation block on the Mux2\_1 module and creating 16 of them for each bit of data that needed to be passed through. The selector for the circuit.

We then used the RegisterFile module created in Lab 5 (RegisterFile.v), which contains three 4-bit decoders (DecoderN) for selecting which register to write to and which register(s) to read for outputs A and B. Each write and read connection has its own enable connection. We implement 16 RegisterOEN modules initialized with 16-bits each to create the internal register set.

The final module we implemented was the ALU from Lab 5 as well. For variety, we used an if-else block from the 3-bit input to decide what to assign the output, Q.

We finished the datapath module by referencing several wire connections for debug output, and specifying the inputs as shown above in Fig. 4

### Control Unit

We began the control unit similarly to the datapath by creating the ROM module (romlpm.v). Similarly, just a 1-port module is necessary for reading one instruction at a time. We created the MIF using a translated instruction set from the sample program specified in the requirements.

The program consists of nine different instructions. The instructions were executed in the following order:

1. Load RF[1] with D[A]
2. Load RF[2] with D[1A]
3. Load RF[3] with D[3]
4. Load RF[4] with D[8A]
5. Subtract RF[2] from RF[1] and store the value in RF[5]
6. Add RF[5] to RF[3] and store the value in RF[6]
7. Subtract RF[4] from RF[6] and store the value in RF[0]
8. Store RF[0] at memory location D[BB]
9. Halt

When encoded, the instructions are encoded as follows in binary (and decimal for Quartus):

1. 0010 0000 1010 0001 = 8353
2. 0010 0001 1010 0010 = 8610
3. 0010 0000 0011 0011 = 8243
4. 0010 1000 1010 0100 = 10404
5. 0100 0001 0010 0101 = 16677
6. 0011 0101 0011 0110 = 13622
7. 0100 0110 0100 0000 = 17984
8. 0001 0000 1011 1011 = 4283
9. 0101 XXXX XXXX XXXX = 20480

We then created the PC and IR modules, which were extremely simple to design. The PC module essentially increments the address given to the ROM. Every time the state machine returns to the FETCH state the incrementation is triggered by setting PC\_up. The IR module essentially holds the value taken from ROM until the next instruction is needed.

Finally, we created the controller state machine module (Controller.v). We used the three basic states (Init, Fetch, and Decode) for controller operation, from which the decoder is connected to the six other states for each of the instructions. We had to split two instructions up to utilize more than one clock cycle. We already knew about the LOAD instruction, but during initial tests while designing, STORE also needed a secondary state to properly write the data to RAM.

Following the guidelines in the lecture slides, our states set the following outputs (on the connections shown in Fig. 4) based on information in the IR (with the format given in Fig. 2). For any math functions, only one state is needed, as we can directly output from the ALU to another register in the RF.

* Init
  + Clear the PC
* Fetch
  + Increment the PC
  + Load the value from the IR
* Decode
  + Set the next state based on IR[17:15]
* NOOP
  + Essentially do nothing
* LOAD\_A
  + Set the address of the data being loaded.
  + Set the RF mux to output from the RAM
  + Set the RF address to write to the specified register
* LOAD\_B
  + Maintain the outputs from LOAD\_A
  + Allow the RF to write the data to the register
* STORE
  + Set the address of the RAM to write to
  + Set the address of the register to read from
  + Enable reading from the register through port A
* STORE2
  + Maintain the outputs from STORE
  + Enable writing to RAM
* ADD
  + Set the RF address to write to, and enable write
  + Set the RF channel A address to read from, and enable read
  + Set the RF channel B address to read from, and enable read
  + Set the ALU to A + B (1)
  + Set the RF input mux to the ALU output (0)
* SUB
  + Set the RF addresses and enables as in ADD
  + Set the ALU to A – B (2)
* HALT
  + Do nothing, except loop back to HALT

We would implement the full state machine using two Always blocks: one to control the state changes (Clock sensitive), and the other to change the outputs (State sensitive). As the clock triggers, it goes to the next state (also choosing which instruction to execute), and because the state changed, this triggers the outputs being set for that state. This lead to an overall cleaner module, and even eliminated the need for a “next state” register variable.

## Observations

### *Processor*

The processor and interfacing module compiled successfully, still only using less than 1% of the boards logic components as shown in the compilation report.

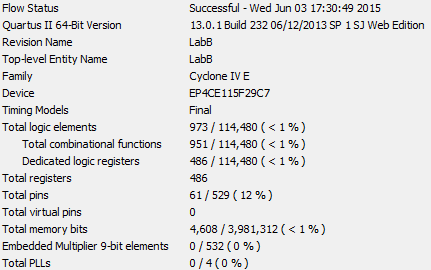


Figure – Compilation Report

### *Control Unit*

### PC

As the PC had a clock input, we assumed that it should also have a reset. However, this appears to be redundant with the ‘Clr’ input to PC. When the processor is reset, the PC is reset, and the controller returns to the Init state. In this state, the clear is asserted, which again clears the PC at the next positive edge of the clock. The ROM address output from PC is a 5 bit number, which allows for 32 instructions. Once the PC counter has reached 31, it begins again at zero.

### IR

The instruction register acts like a stop light for the instructions coming out of the ROM and into the controller. The PC tells the ROM what instruction to output, and the IR will not let the instruction pass to the controller until ld is asserted from the controller.

### ROM

The use of ROM was new for this lab. It has similar properties to RAM, however once the values are loaded from the MIF, the values cannot be changed. It can only be used to read instructions from. The ROM is 32 words by 16 bits. The PC can count 0-31 in order to access all the words from the ROM in order, and the word length is 16 bits because that is how much data each instruction needs to hold.

### Controller

We ended up implementing the controller state machine slightly different than earlier state machines in this class. One block is controlled by the positive edge of the clock, and decides what the next state will be. The other always block is controlled by when the state changes. Inside there is a case statement that changes the outputs of the controller depending on what the current state is. This reduces the need for another reg variable to set the next state.

Another important thing learned is that at the beginning of the state always block, at least some outputs need to be set to 0 before entering the case statement, specifically the read/write enable outputs. Otherwise, the compiler will infer latches that cause unpredictable behavior. That way the outputs are always set to an unused state, unless overridden in the case statement that follows.

Also, to allow for the correct memory address and register to be accessed, as second store state was added to make sure the proper values were written to RAM. Attempting it with only one state proved not to work correctly. The first store state sets the address in memory and enables reading from an RF register. The second state maintains these values, and also enables the RAM to be written to.

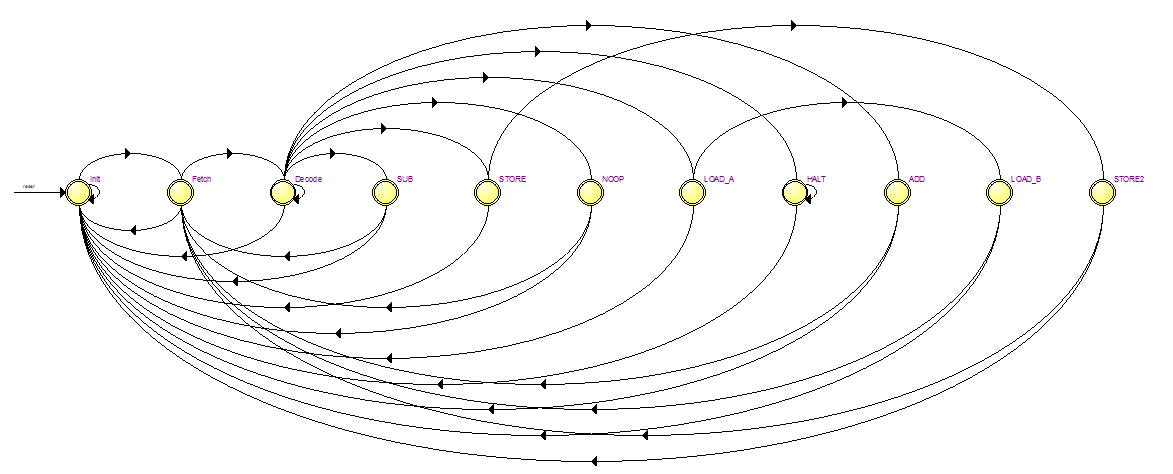


Figure – State machine viewer result for the controller module.

### *Datapath*

### RAM

The RAM was easy to set up through the MegaWizard, and operated as expected. However, sometimes while running the hardware and activating the Memory Viewer, Quartus would crash when closing the viewer.

### 16-bit 2-to-1 MUX

Again this was an easy module to set up like smaller multi-bit wide muxes we made in the past. Early on however, the inputs were flipped in some versions of implementing the module, which seemed to be a source of confusion for other teams as well. Reading the diagram from left to right, the 1 input (RAM) came before the 0 input (ALU).

### Register File

The register file worked early on as expected from finishing Lab 5. In our early drafts on how to implement the sample program, we almost expected that we could operate the ADD and SUB instructions almost like ARM assembly could, that is, being able to write to a register that you’re also trying to read from (i.e. ADD R0, R0, R1). However, this was definitely not the case with this register file. If there was an extra register from the ALU output, and a secondary state for either of the math instructions, this could probably be done. Instead, we needed to put the result in an empty register that we weren’t going to use again.

### ALU

Like the RF, the ALU was already tested in Lab 5 before starting this project. Like mentioned in the RF observation, if there was an extra register or a way to hold the ALU output value, this would add extra functionality to the datapath module. Most of the ALU functions went unused, but could be easily implemented in the instruction RAM. OR, AND, XOR, ADD1, and even MOV (with register values only) could all be added in some way.

## Conclusion

For this lab assignment, we were able to successfully implement a basic processor through a combination of Verilog concepts. We were able to reach a greater understanding of how processors are designed, and a basic outline of the modules that they include. From previously working with assembly code, we realized that this was practically the same approach, albeit on a smaller scale. The control unit stores various instructions in memory, and the controller state machine operates through them as it steps through memory. The program loaded in ROM can be initialized differently, based on what type of program needed to be ran.

## Appendix

### Top-Level Module

#### LabB.v

/\*

Jared Herdlevar

James Brewer

TCES 330

Lab B

Top level module

Key 0 is the clock

Key 1 is reset

\*/

module LabB(KEY, SW, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7);

input [1:0]KEY;

input [17:15]SW;

output [6:0]HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7;

wire [4:0]addr;

wire [3:0]State;

wire [15:0]Alua, Alub, Aluo, rf0, muxo, IR;

//Processor(Clk, Reset, addr, IR, State, Alua, Alub, Aluo, rf0, muxo);

Processor processor1(KEY[0], ~KEY[1], addr, IR, State, Alua, Alub,

Aluo, rf0, muxo);

// these registers determine what the 7seg display

reg [3:0] disp4, disp5, disp6, disp7;

always @(SW[17:15]) begin

case (SW[17:15])

0 : begin

disp7 = State; // current state

disp6 = addr[3:0]; // pc counter

disp5 = addr[4]; // pc counter

disp4 = 0; // empty

end

1: begin

disp4 = Alua[15:12]; // alua

disp5 = Alua[11:8]; // alua

disp6 = Alua[7:4]; // alua

disp7 = Alua[3:0]; // alua

end

2: begin

disp4 = Alub[15:12]; // alub

disp5 = Alub[11:8]; // alub

disp6 = Alub[7:4]; // alub

disp7 = Alub[3:0]; // alub

end

3: begin

disp4 = Aluo[15:12]; // alu output

disp5 = Aluo[11:8]; // alu output

disp6 = Aluo[7:4]; // alu output

disp7 = Aluo[3:0]; // alu output

end

5: begin

disp4 = rf0[15:12]; // register 0 contents

disp5 = rf0[11:8]; // register 0 contents

disp6 = rf0[7:4]; // register 0 contents

disp7 = rf0[3:0]; // register 0 contents

end

6: begin

disp4 = muxo[15:12]; // mux output

disp5 = muxo[11:8]; // mux output

disp6 = muxo[7:4]; // mux output

disp7 = muxo[3:0]; // mux output

end

endcase

end

// display hex numbers to 7seg displays

Hex7Seg h0(disp4, HEX7);

Hex7Seg h1(disp5, HEX6);

Hex7Seg h2(disp6, HEX5);

Hex7Seg h3(disp7, HEX4);

Hex7Seg h4(IR[15:12], HEX3);

Hex7Seg h5(IR[11:8], HEX2);

Hex7Seg h6(IR[7:4], HEX1);

Hex7Seg h7(IR[3:0], HEX0);

endmodule

### Level 1 Modules

#### Hex7Seg.v

// Jared Herdlevar and James Brewer

// Hex display output decoding (0-15) to (0-F)

module Hex7Seg(SW, HEX);

input [0:3]SW;

output reg[0:6]HEX;

// As the input changes, decode the output

always @(SW)

case(SW)

4'b0000 : HEX = 7'b1000000;

4'b0001 : HEX = 7'b1111001;

4'b0010 : HEX = 7'b0100100;

4'b0011 : HEX = 7'b0110000;

4'b0100 : HEX = 7'b0011001;

4'b0101 : HEX = 7'b0010010;

4'b0110 : HEX = 7'b0000010;

4'b0111 : HEX = 7'b1111000;

4'b1000 : HEX = 7'b0000000;

4'b1001 : HEX = 7'b0011000;

4'b1010 : HEX = 7'b0001000;

4'b1011 : HEX = 7'b0000011;

4'b1100 : HEX = 7'b1000110;

4'b1101 : HEX = 7'b0100001;

4'b1110 : HEX = 7'b0000110;

4'b1111 : HEX = 7'b0001110;

default : HEX = 7'b1111111;

endcase

endmodule

#### Processor.v

/\*

Jared Herdlevar

James Brewer

TCES 330

Lab B

This processor module exists to hold the datapath and controlunit.

Outputs from the processor module are to send the debugging info to

the top level module to display on 7seg

\*/

module Processor(Clk, Reset, addr, IR, State, Alua, Alub, Aluo, rf0, muxo);

input Clk, Reset;

output [4:0]addr;

output [15:0]IR, Alua, Alub, Aluo, muxo, rf0;

output [3:0]State;

// wires connect the control unit outputs to the datapath inputs

wire [7:0]D\_addr;

wire D\_wr, RF\_s, RF\_W\_wr, RF\_Ra\_rd, RF\_Rb\_rd;

wire [3:0]RF\_W\_addr, RF\_Ra\_addr, RF\_Rb\_addr, Alu\_s0;

//ControlUnit(Clk, Reset, addr, State, D\_wr, RF\_s, RF\_W\_wr, RF\_Ra\_rd,

//RF\_Rb\_rd, D\_addr, RF\_W\_addr, RF\_Ra\_addr, RF\_Rb\_addr, Alu\_s0);

ControlUnit cu(Clk, Reset, D\_addr, D\_wr, RF\_s, RF\_W\_wr, RF\_Ra\_rd,

RF\_Rb\_rd, RF\_W\_addr, RF\_Ra\_addr, RF\_Rb\_addr, Alu\_s0,

State, IR, addr);

//Datapath(Clk, D\_addr, D\_wr, RF\_s, RF\_W\_wr, RF\_Ra\_rd, RF\_Rb\_rd,

//RF\_W\_addr, RF\_Ra\_addr, RF\_Rb\_addr, Alu\_s0, Alua, Alub,

//Aluo, rf0, muxo);

Datapath dp(Clk, D\_addr, D\_wr, RF\_s, RF\_W\_wr, RF\_Ra\_rd, RF\_Rb\_rd,

RF\_W\_addr, RF\_Ra\_addr, RF\_Rb\_addr, Alu\_s0, Alua, Alub,

Aluo, rf0, muxo, Reset);

Endmodule

### Level 2 Modules

#### ControlUnit.v

/\*

Jared Herdlevar

James Brewer

TCES 330

Lab B

This is the module that holds the PC, IR, statemachine and rom

\*/

module ControlUnit(Clk, Reset, D\_addr, D\_wr, RF\_s, RF\_W\_wr, RF\_Ra\_rd,

RF\_Rb\_rd, RF\_W\_addr, RF\_Ra\_addr, RF\_Rb\_addr, Alu\_s0, thestate, theIR, addr);

input Clk, Reset;

output [7:0]D\_addr;

output D\_wr, RF\_s, RF\_W\_wr, RF\_Ra\_rd, RF\_Rb\_rd;

output [3:0]RF\_W\_addr, RF\_Ra\_addr, RF\_Rb\_addr, thestate;

output [2:0]Alu\_s0;

output [15:0]theIR;

output [4:0]addr;

// wires used to connect modules within the control unit

wire clr, up, ld;

wire [15:0]data, IR;

assign theIR = IR; // used to send to top level module to display on

7seg

// holds the instructions

romlpm rom(addr, Clk, data);

// determines address to grab instruction

PC pc(Clk, Reset, clr, up, addr);

// holds the next instruction

IR ir(Clk, Reset, ld, data, IR);

// Is a statemachine that determines outputs to the datapath

Controller cont(Clk, Reset, IR, clr, up, ld, D\_addr, RF\_W\_addr,

RF\_Ra\_addr, RF\_Rb\_addr, D\_wr, RF\_s, RF\_W\_wr, RF\_Ra\_rd, RF\_Rb\_rd, Alu\_s0, thestate);

Endmodule

#### Datapath.v

/\*

Jared Herdlevar

James Brewer

TCES 330

Lab B

The datapath module that holds the ram, alu, registers and mux.

\*/

module Datapath(Clk, D\_addr, D\_wr, RF\_s, RF\_W\_wr, RF\_Ra\_rd, RF\_Rb\_rd,

RF\_W\_addr, RF\_Ra\_addr, RF\_Rb\_addr, Alu\_s0, Alua,

Alub, Aluo, rf0, muxo, Reset);

input Clk, D\_wr, RF\_s, RF\_W\_wr, RF\_Ra\_rd, RF\_Rb\_rd;

input [3:0]RF\_W\_addr, RF\_Ra\_addr, RF\_Rb\_addr;

input [7:0]D\_addr;

input [2:0]Alu\_s0;

input Reset;

output [15:0]Alua, Alub, Aluo, muxo;

output reg [15:0]rf0;

initial begin

rf0 = 0; // initialize for modelsim

end

// wires used to connect modules within datapath.

wire [15:0]Ra\_data, Rb\_data, ALU\_out, R\_data;

// used to send to top level module to dislpay on 7seg

assign Alua = Ra\_data;

assign Alub = Rb\_data;

assign Aluo = ALU\_out;

//Ram holds data in memory

ramlpm ram(D\_addr, Clk, Ra\_data, D\_wr, R\_data);

//The mux decides if the input to the registers will be from ram or the

alu output.

Mux16w2\_1 mux16(RF\_s, ALU\_out, R\_data, muxo);

// The registers

RegisterFile registefile(Clk, Reset, muxo, RF\_W\_addr, RF\_W\_wr,

RF\_Ra\_addr, RF\_Rb\_addr, RF\_Ra\_rd, RF\_Rb\_rd, Ra\_data, Rb\_data);

//The arithmetic logic unit

ALU alu1(Ra\_data, Rb\_data, Alu\_s0, ALU\_out);

// this is used to store the current value of register 0.

// whenever the register write address is set to 0, and write

// is enabled, store that value in rf0.

always @(posedge Clk) begin

if (RF\_W\_addr == 4'b0000 & RF\_W\_wr)

rf0 = muxo;

end

endmodule

### Level 3 Modules

#### PC.v

/\*

Jared Herdlevar

James Brewer

TCES 330

Lab B

The program counter

Clk the clock

Resets the counter

clr clears the counter

addr the address to grab in rom

\*/

module PC(Clk, Reset, clr, up, addr);

input Clk, Reset, clr, up;

output reg[4:0]addr;

initial // initialize for modelsim

addr = 0;

always @(posedge Clk) begin

if (Reset | clr) // if reset or cleared set address to 0

addr = 0;

else if (up)

addr = addr + 1; // since addr is 5 bits, it will roll over

to 0

end

endmodule

#### IR.v

/\*

Jared Herdlevar

James Brewer

TCES 330

Lab B

The instruction register

Clk is the clock

Reset resets the IR

ld is enable to grab next instruction

data is the data coming into the register

IR is the data leaving the IR

\*/

module IR(Clk, Reset, ld, data, IR);

input Clk, Reset, ld;

input [15:0]data;

output reg [15:0]IR;

initial // initialize for model sim

IR = 0;

always @(posedge Clk) begin

if (Reset)

IR = 0;

else if (ld) // if enabled output the incoming data

IR = data;

end

endmodule

#### Controller.v

/\*

Jared Herdlevar

James Brewer

TCES 330

Lab B

Processor controller.

Clk = system clock

Reset = synchronous reset

IR = instruction

up = pc incrementer

ld = load instruction register with new instruction

D\_wr = memory write enable

RF\_s = multiplexor selector

RF\_W\_wr = register write enable

RF\_Ra\_rd = register read enable

RF\_Rb\_rd = register read enable

D\_addr = memory address

RF\_W\_addr = register write address

RF\_Ra\_addr = register read address

RF\_Rb\_addr = register read address

Alu\_s0 = alu selector

State = current state

\*/

module Controller(Clk, Reset, IR, clr, up, ld, D\_addr, RF\_W\_addr, RF\_Ra\_addr,

RF\_Rb\_addr, D\_wr, RF\_s, RF\_W\_wr, RF\_Ra\_rd, RF\_Rb\_rd, Alu\_s0, thestate);

input Clk, Reset;

input [15:0]IR;

output reg clr, up, ld, D\_wr, RF\_s, RF\_W\_wr, RF\_Ra\_rd, RF\_Rb\_rd;

output reg[3:0]RF\_W\_addr, RF\_Ra\_addr, RF\_Rb\_addr;

output reg[7:0]D\_addr;

output reg[2:0]Alu\_s0;

output [3:0]thestate;

reg [3:0]state;

initial begin // initialize for modelsim

clr = 0;

up = 0;

ld = 0;

D\_wr = 0;

RF\_s = 0;

RF\_W\_wr = 0;

RF\_Ra\_rd = 0;

RF\_Rb\_rd = 0;

RF\_W\_addr = 0;

RF\_Ra\_addr = 0;

RF\_Rb\_addr = 0;

D\_addr = 0;

Alu\_s0 = 0;

end

assign thestate = state; // assign output state so Quartus recognizes

this as a FSM

localparam [3:0]Init = 4'b0000, // enumerate the states

Fetch = 4'b0001,

Decode = 4'b0010,

NOOP = 4'b0011,

LOAD\_A = 4'b0100,

LOAD\_B = 4'b0101,

STORE = 4'b0110,

ADD = 4'b0111,

SUB = 4'b1000,

HALT = 4'b1001,

STORE2 = 4'b1010;

// When ever the state changes, change the outputs

always @(state) begin

clr <= 0;

up <= 0;

ld <= 0;

D\_wr <= 0;

RF\_W\_wr <= 0;

case (state)

Init: begin // intitialize

clr <= 1;

end

Fetch: begin // fetch instruction

up <= 1;

ld <= 1;

end

Decode: begin // decode instruction

end

NOOP: begin // do nothing

end

LOAD\_A: begin // load from memory

D\_addr <= IR[11:4];

RF\_s <= 1;

RF\_W\_addr <= IR[3:0];

end

LOAD\_B: begin // load into register

D\_addr <= IR[11:4];

RF\_s <= 1;

RF\_W\_addr <= IR[3:0];

RF\_W\_wr <= 1;

end

STORE: begin // load up the mem address

D\_addr <= IR[7:0];

RF\_Ra\_addr <= IR[11:8];

RF\_Ra\_rd <= 1;

end

STORE2: begin // write to the mem address

D\_addr <= IR[7:0];

D\_wr <= 1;

RF\_Ra\_addr <= IR[11:8];

RF\_Ra\_rd <= 1;

end

ADD: begin // add 2 registers and store in a third

RF\_W\_addr <= IR[3:0];

RF\_W\_wr <= 1;

RF\_Ra\_addr<=IR[11:8];

RF\_Ra\_rd <= 1;

RF\_Rb\_addr <= IR[7:4];

RF\_Rb\_rd <= 1;

Alu\_s0 <= 1;

RF\_s <= 0;

end

SUB: begin // subtract 2 registers and store in a 3rd

RF\_W\_addr <= IR[3:0];

RF\_W\_wr <= 1;

RF\_Ra\_addr<=IR[11:8];

RF\_Ra\_rd <= 1;

RF\_Rb\_addr <= IR[7:4];

RF\_Rb\_rd <= 1;

Alu\_s0 <= 2;

RF\_s <= 0;

end

HALT: begin // stop

end

default: begin

end

endcase

end

// at posedge of clock change states

always @(posedge Clk) begin

if (Reset)

state = Init;

else

case (state) // grabs the nextstate and stores in state

Init: begin

state = Fetch;

end

Fetch: begin

state = Decode;

end

Decode: begin

if (IR[15:12] == 4'b0000)

state = NOOP;

else if (IR[15:12] == 4'b0001)

state = STORE;

else if (IR[15:12] == 4'b0010)

state = LOAD\_A;

else if (IR[15:12] == 4'b0011)

state = ADD;

else if (IR[15:12] == 4'b0100)

state = SUB;

else if (IR[15:12] == 4'b0101)

state = HALT;

else begin

end

end

NOOP: begin

state = Fetch;

end

LOAD\_A: begin

state = LOAD\_B;

end

LOAD\_B: begin

state = Fetch;

end

STORE: begin

state = STORE2;

end

STORE2: begin

state = Fetch;

end

ADD: begin

state = Fetch;

end

SUB: begin

state = Fetch;

end

HALT: begin

state = HALT;

end

default: begin

state = Init;

end

endcase

end

endmodule

#### Mux16w2\_1.v

/\*

Jared Herdlevar

James Brewer

TCES 330

Lab B

A 16-bit wide 2-to-1 MUX

\*/

module Mux16w2\_1(s, x, y, m);

input s;

input [15:0] x, y;

output [15:0] m;

genvar i;

//Generates 16 Muxes

generate

for (i = 0; i <= 15; i = i + 1) begin: mux

Mux2\_1 Mx(s, x[i], y[i], m[i]);

end

endgenerate

endmodule

#### RegisterFile.v

// TCES 330, Spring 2015

// Implements a register file

// Size: 16 16-bit registers

// Jared Herdlevar and James Brewer

// Changes to enable reading 2 ports.

module RegisterFile( Clk, Reset, W\_data, W\_addr, W\_en, Ra\_addr, Rb\_addr, Ra\_en, Rb\_en, Ra\_data, Rb\_data );

input Clk; // system clock

input Reset; // reset signal

input [15:0] W\_data; // data to write

input [3:0] W\_addr; // write address

input W\_en; // write enable

input [3:0] Ra\_addr, Rb\_addr; // read address

input Ra\_en, Rb\_en; // read enable

output [15:0] Ra\_data, Rb\_data;// output data

wire [15:0] W\_d, R\_d, Rb\_d;

// write decoder

DecoderN #(.N(4)) WriteDecoder( W\_addr, W\_en, W\_d );

// read decoder

DecoderN #(.N(4)) ReadDecoder( Ra\_addr, Ra\_en, R\_d );

DecoderN #(.N(4)) ReadDecoder1( Rb\_addr, Rb\_en, Rb\_d );

// registers with output enable

// reference: module RegisterOEN( Clk, Rst, Ld, Ia, Ib, Oea, Oeb, Qa, Qb );

RegisterOEN #(.N(16)) Reg0( Clk, Reset, W\_d[0], W\_data, R\_d[0], Rb\_d[0], Ra\_data, Rb\_data );

RegisterOEN #(.N(16)) Reg1( Clk, Reset, W\_d[1], W\_data, R\_d[1], Rb\_d[1], Ra\_data, Rb\_data );

RegisterOEN #(.N(16)) Reg2( Clk, Reset, W\_d[2], W\_data, R\_d[2], Rb\_d[2], Ra\_data, Rb\_data );

RegisterOEN #(.N(16)) Reg3( Clk, Reset, W\_d[3], W\_data, R\_d[3], Rb\_d[3], Ra\_data, Rb\_data );

RegisterOEN #(.N(16)) Reg4( Clk, Reset, W\_d[4], W\_data, R\_d[4], Rb\_d[4], Ra\_data, Rb\_data );

RegisterOEN #(.N(16)) Reg5( Clk, Reset, W\_d[5], W\_data, R\_d[5], Rb\_d[5], Ra\_data, Rb\_data );

RegisterOEN #(.N(16)) Reg6( Clk, Reset, W\_d[6], W\_data, R\_d[6], Rb\_d[6], Ra\_data, Rb\_data );

RegisterOEN #(.N(16)) Reg7( Clk, Reset, W\_d[7], W\_data, R\_d[7], Rb\_d[7], Ra\_data, Rb\_data );

RegisterOEN #(.N(16)) Reg8( Clk, Reset, W\_d[8], W\_data, R\_d[8], Rb\_d[8], Ra\_data, Rb\_data );

RegisterOEN #(.N(16)) Reg9( Clk, Reset, W\_d[9], W\_data, R\_d[9], Rb\_d[9], Ra\_data, Rb\_data );

RegisterOEN #(.N(16)) Reg10( Clk, Reset, W\_d[10], W\_data, R\_d[10], Rb\_d[10], Ra\_data, Rb\_data );

RegisterOEN #(.N(16)) Reg11( Clk, Reset, W\_d[11], W\_data, R\_d[11], Rb\_d[11], Ra\_data, Rb\_data );

RegisterOEN #(.N(16)) Reg12( Clk, Reset, W\_d[12], W\_data, R\_d[12], Rb\_d[12], Ra\_data, Rb\_data );

RegisterOEN #(.N(16)) Reg13( Clk, Reset, W\_d[13], W\_data, R\_d[13], Rb\_d[13], Ra\_data, Rb\_data );

RegisterOEN #(.N(16)) Reg14( Clk, Reset, W\_d[14], W\_data, R\_d[14], Rb\_d[14], Ra\_data, Rb\_data );

RegisterOEN #(.N(16)) Reg15( Clk, Reset, W\_d[15], W\_data, R\_d[15], Rb\_d[15], Ra\_data, Rb\_data );

Endmodule

#### ALU.v

// Jared Herdlevar and James Brewer

// This ALU has eight functions:

// if s == 0 the output is 0

// if s == 1 the output is A + B

// if s == 2 the output is A – B

// if s == 3 the output is A (pass-through)

// if s == 4 the output is A ^ B

// if s == 5 the output is A | B

// if s == 6 the output is A & B

// if s == 7 the output is A + 1;

// the additional functions are for future expansion

module ALU( A, B, Sel, Q );

input [2:0] Sel; // function select

input [15:0] A, B; // input data

output reg [15:0] Q; // ALU output (result)

initial

Q = 0;

// if any of the inputs change...

always @(A or B or Sel) begin

if (Sel == 0)

Q = 0;

else if (Sel == 1)

Q = A + B;

else if (Sel == 2)

Q = A - B;

else if (Sel == 3)

Q = A;

else if (Sel == 4)

Q = A ^ B;

else if (Sel == 5)

Q = A | B;

else if (Sel == 6)

Q = A & B;

else

Q = A + 1;

end

endmodule

### Level 4 Modules

#### Mux2\_1.v

/\*

Jared Herdlevar

James Brewer

TCES 330

Lab B

A simple 2-to-1 MUX

\*/

module Mux2\_1 (s, x, y, m);

input s, x, y;

output m;

assign m = (s & y) | (~s & x);

endmodule

#### RegisterOEN.v

// TECES 330, Spring 2015

// R. Gutmann

// Generic Register with output enable

// Used to build register files

// Jared Herdlevar and James Brewer

// Changes to enable reading 2 ports.

module RegisterOEN( Clk, Rst, Ld, I, Oea, Oeb, Qa, Qb );

parameter N = 32;

input Clk;

input Rst;

input Ld; // load signal

input Oea, Oeb; // output enable (this is new)

input [N-1:0] I; // data to load

output reg[N-1:0] Qa, Qb; // switched output

reg [N-1:0] Q; // standard (unswitched) output

initial begin

Qa = 0;

Qb = 0;

end

// Register Procedure

always @(posedge Clk) begin

if (Rst == 1)

Q <= 0;

else if (Ld == 1)

Q <= I;

end

always @(Oea, Oeb)begin

// Output

Qa = Oea ? Q : {N{1'bZ}};

Qb = Oeb ? Q : {N{1'bZ}};

end

endmodule

#### DecoderN.v

// TCES 330, Spring 2015

// Implement an N to 2\*\*N decoder

module DecoderN( W, E, Y );

parameter N = 2;

localparam M = 2\*\*N;

input [N-1:0] W; // the inputs

input E; // enable

output reg [M-1:0] Y; // the outputs

initial begin

Y = 0;

end

always @(W, E) begin

if ( E ) begin

Y = 1'b1 << W; // shift a '1' into position

end

else begin

Y = 0;

end

end // always

endmodule