**Project 3 – Single Cycle MIPS Processor**

Programmer: Victor Espinoza

SID: 010657450

CECS 440, Section 5; Tu/Th 9:30 - 11:45 A.M.

Lab Section 6

Due: Tuesday, March 24, 2015

**Introduction:**

The purpose of this lab was to combine our register file and ALU modules (made in previous labs), together and use them to help create a Single Cycle MIPS Processor. We used these modules, along with the Controller module and Test bench module provided to us by the professor, to help us create the Single Cycle MIPS Processor. Aside from these modules, I needed to implement a Program Counter, an Instruction Memory, a Data Memory, and other various multiplexors, adders, and logical gates in order to implement the processor. The processor module only has two input variables: clock and reset. It does not contain any output variables. All of the other variables contained within the processor are local to the Single Cell Processor module. It was somewhat challenging wiring all of the parts of the lab correctly, but drawing out the processor diagram and labeling all of the wires really helped ease the burden of wiring everything properly. I have attached my diagram of the Single Cycle MIPS Processor after the Verification Description portion of this lab report.

**Project Description:**

This lab combines our previous labs of the register file and the ALU and implements them into a single cycle MIPS Processor. This means that one instruction is executed every clock cycle. In order to accommodate being able to execute all of the instructions that we implemented in one clock cycle, I needed to stretch out the clock according to the instruction that took the longest amount of time to execute. The critical path for this processor (the instruction with the longest delay) was the load instruction, so this instruction is what determined the clock period. At the beginning of each clock cycle, the Program Counter receives the appropriate address and then it fetches the instruction corresponding to that particular address. Each address in the Program Counter is byte addressable (word-aligned), meaning that it goes up in increments of 4 (0x00, 0x04, 0x08, 0x0C, 0x10, etc...). In order to align and interface the Program Counter address with the physical addresses of the Instruction and Data Memory, I needed to shift the address right by 2. This allows us to get the physical address of the word-addressable memories. Depending on the type of instruction being executed, the processor then takes the different paths associated with each type of instruction. There are three different types of instructions being implemented by our processor: Register Type Instructions, Load/Store Type Instructions, and Branch Type Instructions. Each instruction takes a slightly different path through the processor. The type of instruction being executed is determined in the Instruction Memory File. Inside of this file, the machine code that corresponds to a specific instruction is stored. At the beginning of each clock cycle, we fetch the instruction located in Instruction Memory corresponding to the Program Counter's Address and then execute it. Each instruction is made up of 32 bits. Bits 31:26 make up the operation code (opcode) for each instruction. These bits help identify what operation/instruction is going to be executed. Bits 25:21 comprise the address of the source register, rs. Bits 20:16 contain the address of yet another source register, rt. It is important to note that Load instructions use these bits as the address of the destination register instead of a source register. For Load/Store Type Instructions and Branch Type Instructions, the lower 16 bits (bits 15:0) are used for either storing a constant value or for storing a target address. For Register Type Instructions, the lower 16 bits are further divided into more instruction fields: bits 15:11 hold the destination register address, rd, bits 10:6 hold the shift amount value for the instruction, and finally bits 5:0 hold the function code, which extends the opcode and helps to further identify which instruction is being executed within the Arithmetic Logic Unit (ALU).

**Verification Description:**

The test bench for this lab was extremely simple. First, the local variables were declared and then the unit under test (the Single Cycle Processor) was instantiated. Next, I initialized the Data Memory within the Processor to contain all 0 values. Then it was established that the clock would complement itself every #10 and the Instruction Memory File was read into the InstrMem variable. Finally, the clk and rstb variables were initialized to being low and a reset occurred for #100, then the processor began to execute the instructions. The first instruction in the Instruction Memory was lui $r1, 0xFEDB. This is a load upper immediate instruction, meaning that the 0xFEDB will get loaded into the upper 16 bits of the destination register and the lower 16 bits will be filled in with 0's (0xFEDB0000). The ALUCtl variable of 0x12 signified that a load upper immediate instruction was to be executed within the ALU. The fact that my ALU\_out wire variable contained this value proves that this instruction executed properly. The second instruction in the Instruction Memory was ori $r1, $r1, 0xA987, meaning that an bitwise OR immediate instruction was to be executed within the ALU. This was signified by an ALUCtl value of 0x10. OR-ing the value of 0xFEDB0000 with 0x0000A987 should result in the value of 0xFEDBA987, which is exactly what my ALU\_out variable contained. The third instruction in the Instruction Memory was lui $r2, $r2, 0x1234. This instruction was just like the first instruction, instead now we are storing the value of 0x12340000 in register 2 instead of register 1. My ALU\_out variable had a value of 0x12340000, which was expected. The fourth instruction was ori $r2, $r2, 0x5678. This instruction was similar to the second instruction, instead now we are Or-ing the value in register 2 with 0x00005678. My ALU\_out variable contained a 0x12345678 value, which is as expected. The fifth instruction was add $r3, $r2, $r1. This instruction adds the value stored in register 2 (0x12345678) with the value stored in register 1 (0x FEDBA987) and stores that value in register 3. The ALUCtl variable for this instruction was 0x01, which indicates that a signed add instruction was going to be taking place. This resulted in a value of 0x110FFFFF, which matches the value that was displayed in my ALU\_out variable. The sixth and final instruction within the Instruction Memory was beq $r0, $r0, 0xFFE8. This instruction is a Branch if Equal instruction. To execute this instruction, we first subtract the two source registers from each other and if they result of that subtraction is zero, then the zero flag is set. Being that we are subtracting register 0 from itself in this instruction, the result will be 0 and hence the zero flag will be set. The next step is finding out the effective address that we are going to be branching to. In order to do this, we sign-extend the offset address provided in the branch instruction (0xFFE8), which results in 0xFFFFFFE8. We then shift this value left by 2 (resulting in 0xFFFFFFA0) and then add that to the value stored in the PC\_plus\_4 variable (0x00000018). Once this is finished, we then end up with a value of 0xFFFFFFB8, which is the instruction that we end up branching to.

The Program Counter is now at address location 0xFFFFFFB8. Since there is no instruction within the Instruction Memory available at this address, we do not execute any instructions. The Program Counter then starts counting up addresses in increments of 4 and the same is true for these instruction as well: there are no instructions that correspond to these addresses in the Instruction Memory. This goes on until the Program Counter rolls over to the value of 0x00000000 (where the first instruction in the Instruction Memory is located). At this point, the Processor has looped back to the beginning and it executes the same six instructions that it previously executed. In this way, our processor is a self-contained processor that loops back to itself. The processor will continue to infinitely loop back to itself and re-execute the same set of instructions that was provided to it inside of the Instruction Memory.

**Processor Diagram/Source Code:**

`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date: 15:07:19 03/07/2015

// Design Name: SingleCycle

// Module Name: C:/Users/John Tramel/Desktop/SingleCycle/SingleCycle/

// SingleCycle\_tb.v

// Project Name: SingleCycle

// Target Device:

// Tool versions:

// Description:

//

// Verilog Test Fixture created by ISE for module: SingleCycle

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

////////////////////////////////////////////////////////////////////////////////

module SingleCycle\_tb;

// Inputs

reg clk;

reg rstb;

//Local variables

integer i;

// Instantiate the Unit Under Test (UUT)

SingleCycle uut (

.clk(clk),

.rstb(rstb)

);

initial begin

for(i=0; i<1024; i=i+1)

uut.DataMem[i] <= 32'b0;

end

always #10 clk = ~clk;

initial $readmemh("memfileh",uut.InstrMem);

initial begin

// Initialize Inputs

clk = 0;

rstb = 0;

// Wait 100 ns for global reset to finish

#100 rstb = 1;

// Add stimulus here

end

endmodule

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

//

// Author: Victor Espinoza

// Email: victor.alfonso94@gmail.com

// Project #: Project 3 - Single Cycle MIPS Processor

// Course: CECS 440

// Create Date: 11:02:28 03/10/2015

//

// Module Name: SingleCycle

// File Name: SingleCycle.v

// Description: This top level module combines our previous labs of the register

// file and the ALU and implements them into a single cycle MIPS

// processor. This means that one instruction is executed every

// clock cycle. In order to accommodate being able to execute all

// of the instructions that we implemented in one clock cycle, I

// needed to stretch out the clock according to the instruction

// that took the longest amount of time to execute. The critical

// path for this processor (the instruction with the longest

// delay) was the load instruction, so this instruction is what

// determined the clock period. At the beginning of each clock

// cycle, the Program Counter receives the appropriate address

// and then it fetches the instruction corresponding to that

// particular address. Each address in the Program Counter is

// byte addressable (word-aligned), meaning that it goes up in

// increments of 4 (0x00, 0x04, 0x08, 0x0C, 0x10, etc...). In

// order to align and interface the Program Counter address with

// the physical addresses of the Instruction and Data Memory, I

// needed to shift the address right by 2. This allows us to get

// the physical address of the word-addressable memories. Depending

// on the type of instruction being executed, the processor then

// takes the different paths associated with each type of

// instruction. There are three different types of instructions

// being implemented by our processor: register type instructions,

// Load/Store type instructions, and Branch type instructions.

// Each instruction takes a slightly different path through the

// processor.

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module SingleCycle(clk, rstb);

//Establish Inputs

input clk, rstb;

//Local wire variables

wire [31:0] PC\_PLUS\_4;

wire [31:0] next\_PC;

wire [31:0] add\_PC\_addr;

wire [31:0] rd\_data1;

wire [31:0] rd\_data2;

wire [31:0] Immed\_Value\_Extended;

wire [31:0] rd\_data2\_muxed;

wire [31:0] wr\_data\_muxed;

wire [31:0] ALU\_out;

wire [4:0] ALUCtl;

wire [4:0] wr\_addr\_muxed;

wire ZF;

wire RegDst;

wire BranchE;

wire BranchNE;

wire MemRead;

wire MemtoReg;

wire MemWrite;

wire ALUSrc;

wire RegWrite;

wire branch\_mux\_sel;

//Local reg variables

reg [31:0] PC;

reg [31:0] Instr;

reg [31:0] data\_mem\_out;

reg [31:0] InstrMem [0:1023];

reg [31:0] DataMem [0:1023];

//////////////////////////////////////////////////////////////////////////////////

//Start Program Counter Section of Processor

always @(posedge clk, negedge rstb)

if (!rstb)

PC <= 32'b0;

else

PC <= next\_PC;

//Finished with Program Counter Section of Processor

//////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////

//Start Instruction Memory Section of Processor

always@(\*)

Instr = InstrMem[PC>>2];

//Finished Instruction Memory Section of Processor

//////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////

//Start Controller Section of Processor

//Controller Module Instantiation

//module Controller (InstHi, InstLo, RegDst, BranchE, BranchNE, MemRead, MemtoReg,

// MemWrite, ALUSrc, RegWrite, ALUCtl);

Controller Control(

.InstHi(Instr[31:26]),

.InstLo(Instr[5:0]),

.RegDst(RegDst),

.BranchE(BranchE),

.BranchNE(BranchNE),

.MemRead(MemRead),

.MemtoReg(MemtoReg),

.MemWrite(MemWrite),

.ALUSrc(ALUSrc),

.RegWrite(RegWrite),

.ALUCtl(ALUCtl)

);

//Finished with Controller Section of Processor

//////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////

//Start Register File Section of Processor

//write register address mux 1 0

assign wr\_addr\_muxed = (RegDst) ? Instr[15:11] : Instr[20:16];

//Register File Module Instantiation

//module register\_file(clk, rstb, wr\_e, wr\_addr, wr\_data, rd\_addr1, rd\_addr2,

// rd\_data1,rd\_data2);

register\_file Registers(

.clk(clk),

.rstb(rstb),

.wr\_e(RegWrite),

.wr\_addr(wr\_addr\_muxed),

.wr\_data(wr\_data\_muxed),

.rd\_addr1(Instr[25:21]),

.rd\_addr2(Instr[20:16]),

.rd\_data1(rd\_data1),

.rd\_data2(rd\_data2)

);

//Finished with Register File Section of Processor

//////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////

//Start ALU Section of Processor

//Sign Extend Immediate Value

assign Immed\_Value\_Extended = {{16{Instr[15]}}, Instr[15:0]}; //Sign-extended

//Immediate value

//ALU data2 select mux 1 0

assign rd\_data2\_muxed = (ALUSrc) ? Immed\_Value\_Extended : rd\_data2;

//ALU Module Instantiation

//module ALU(sel, A, B, ZF, Y);

ALU ALU\_UNIT(

.sel(ALUCtl),

.A(rd\_data1),

.B(rd\_data2\_muxed),

.ZF(ZF),

.Y(ALU\_out)

);

//Finished with ALU Section of Processor

//////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////

//Start PC Select Section of Processor

//assign PC+4 wire

assign PC\_PLUS\_4 = PC + 4;

//assign add\_PC\_addr wire

assign add\_PC\_addr = PC\_PLUS\_4 + (Immed\_Value\_Extended<<2);

//assign the branch\_mux\_sel wire

assign branch\_mux\_sel = (BranchE & ZF) | (BranchNE & ~ZF);

//next PC select mux 1 0

assign next\_PC = (branch\_mux\_sel) ? add\_PC\_addr : PC\_PLUS\_4;

//Finished PC Select Section of Processor

//////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////

//Start Data Memory Section of Processor

//read data from data memory if MemRead is asserted

always @(\*)

data\_mem\_out = (MemRead) ? DataMem[ALU\_out>>2]:32'b0;

//write to data memory if MemWrite is asserted

always @(posedge clk, negedge rstb)

if (MemWrite)

DataMem[ALU\_out>>2] <= rd\_data2;

//Data Memory write data select mux 1 0

assign wr\_data\_muxed = (MemtoReg) ? data\_mem\_out : ALU\_out;

//Finished with Data Memory Section of Processor

//////////////////////////////////////////////////////////////////////////////////

endmodule

// SingleCycle Controller

// CECS 440 California State University Long Beach

// John Tramel

// 3/7/2015

// Controller for Single Cycle MIPs Processor

`timescale 1ns/1ns

module Controller (InstHi, InstLo, RegDst, BranchE, BranchNE, MemRead, MemtoReg,

MemWrite, ALUSrc, RegWrite, ALUCtl);

input [ 5:0] InstHi; // Instruction[31:26]

input [ 5:0] InstLo; // Instruction[ 5:0]

output RegDst; // Register Write Adress Select

output BranchE; // Branch Control for =Zero

output BranchNE; // Branch Control for !=Zero

output MemRead; // Read Enable for Data Memory

output MemtoReg; // Write Data Select for Register File

output MemWrite; // Write Enable for Data Memory

output ALUSrc; // ALU B Mux Select

output RegWrite; // Write Enable for Register File

output [ 4:0] ALUCtl; // Control to ALU

//////////// Declare Data Types ///////////////

reg [ 4:0] ALUCtl; // Control to ALU

reg [ 7:0] Ctl; // Control Bits

///////////////// Complete Set of Implemeted Instructions ///////////

assign {RegDst,BranchE,BranchNE,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite} = Ctl;

always @(\*)

casez({InstHi,InstLo})

({6'h00,6'h00}): {ALUCtl,Ctl} = {5'h00,8'h00}; // NOP

({6'h00,6'h20}): {ALUCtl,Ctl} = {5'h01,8'h81}; // ADD

({6'h00,6'h21}): {ALUCtl,Ctl} = {5'h02,8'h81}; // ADDU

({6'h00,6'h22}): {ALUCtl,Ctl} = {5'h03,8'h81}; // SUB

({6'h00,6'h23}): {ALUCtl,Ctl} = {5'h04,8'h81}; // SUBU

({6'h00,6'h24}): {ALUCtl,Ctl} = {5'h05,8'h81}; // AND

({6'h00,6'h25}): {ALUCtl,Ctl} = {5'h06,8'h81}; // OR

({6'h00,6'h26}): {ALUCtl,Ctl} = {5'h07,8'h81}; // XOR

({6'h00,6'h27}): {ALUCtl,Ctl} = {5'h08,8'h81}; // NOR

({6'h00,6'h2A}): {ALUCtl,Ctl} = {5'h09,8'h81}; // SLT

({6'h00,6'h2B}): {ALUCtl,Ctl} = {5'h0A,8'h81}; // SLTU

({6'h08,6'h??}): {ALUCtl,Ctl} = {5'h0B,8'h01}; // ADDI

({6'h09,6'h??}): {ALUCtl,Ctl} = {5'h0C,8'h01}; // ADDIU

({6'h0A,6'h??}): {ALUCtl,Ctl} = {5'h0D,8'h03}; // SLTI

({6'h0B,6'h??}): {ALUCtl,Ctl} = {5'h0E,8'h82}; // SLTIU

({6'h0C,6'h??}): {ALUCtl,Ctl} = {5'h0F,8'h03}; // ANDI

({6'h0D,6'h??}): {ALUCtl,Ctl} = {5'h10,8'h03}; // ORI

({6'h0E,6'h??}): {ALUCtl,Ctl} = {5'h11,8'h03}; // XORI

({6'h0F,6'h??}): {ALUCtl,Ctl} = {5'h12,8'h03}; // LUI

({6'h04,6'h??}): {ALUCtl,Ctl} = {5'h03,8'h40}; // BEQ

({6'h05,6'h??}): {ALUCtl,Ctl} = {5'h03,8'h20}; // BNE

({6'h23,6'h??}): {ALUCtl,Ctl} = {5'h01,8'h9B}; // LW

({6'h2B,6'h??}): {ALUCtl,Ctl} = {5'h01,8'h84}; // SW

default: {ALUCtl,Ctl} = {5'h00,8'h00}; // NOP

endcase

endmodule

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

//

// Author: Victor Espinoza

// Email: victor.alfonso94@gmail.com

// Project #: Project 1 - Register File

// Course: CECS 440

// Create Date: 11:05:58 2/1/2015

//

// Module Name: register\_file

// File Name: register\_file.v

//

// Description: This module creates a register file that is capable of writing

// to and reading from an array of 32 different registers. Which

// registers get written to and which registers have their values

// read from depend on the various 5-bit data address inputs. In

// order to write a value into a register, three conditions must

// be met: 1) A register address must be provided, 2) Data to

// store into the register must be provided, and most importantly,

// 3) The wr\_e input must be asserted on the active edge of the

// clock. If the write enable input is not asseted, then the

// register file does not write data into any register. Only once

// this input is asserted is the register file allowed to write

// data into one of its registers. The reset input clears all of

// the data within the register file and the rd\_data1 and rd\_data2

// output ports display the data values of the register referenced

// by their respective addresses (rd\_addr1 and rd\_addr2).

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module register\_file(clk, rstb, wr\_e, wr\_addr, wr\_data, rd\_addr1, rd\_addr2, rd\_data1

,rd\_data2);

//Initialize Inputs

input clk;

input rstb;

input wr\_e;

input [4:0] wr\_addr;

input [31:0] wr\_data;

input [4:0] rd\_addr1;

input [4:0] rd\_addr2;

//Initialize Outputs

output [31:0] rd\_data1;

output [31:0] rd\_data2;

//Initialize local variables

reg [31:0] REG [0:31];

//assign the output data to its respective register

assign rd\_data1 = (rd\_addr1 != 5'b0) ? REG[rd\_addr1] : 32'b0;

assign rd\_data2 = (rd\_addr2 != 5'b0) ? REG[rd\_addr2] : 32'b0;

//Behavioral section for writing to the register

integer i;

always @(posedge clk, negedge rstb) begin

if(!rstb)

begin

//if the reset bit is asserted, then go

//through the register file and clear all

//of the data that was written to the

//individual registers.

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

REG[i] <= 0;

end

else

begin

if(wr\_e)

//write the data into the desired register

//if and only if wr\_e is asserted.

REG[wr\_addr] <= wr\_data;

end

end

endmodule

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

//

// Author: Victor Espinoza

// Email: victor.alfonso94@gmail.com

// Project #: Project 2 - Arithmetic Logic Unit (ALU)

// Course: CECS 440

// Create Date: 11:05:58 2/19/2015

//

// Module Name: ALU

// File Name: ALU.v

//

// Description: This module creates an ALU that performs various manipulations

// on two inputs. The size of the A input was always 32-bits. The

// value being passed into this input was either signed or

// unsigned. The size of the B input was either 32-bits wide or

// 16-bits wide (for immediate data). The values within this

// input were also either signed or unsigned depending on which

// instruction was being called. Signed and unsigned are different

// interpretations of the way values are perceived. In signed

// values, any value greater than 7FFF\_FFFF (32-bit value for the

// lab), is perceived to be negative. With unsigned values, however,

// there is no such thing as a negative number. The value just keeps

// on getting bigger and bigger. The initial values for both the A and

// B inputs for our ALU module are both unsigned. In order to perform

// the signed instructions in the ALU using these initially unsigned

// values, I had to convert the values from unsigned to signed. I

// did this by creating local integer variables, which are by

// default signed entities. I then moved the A and B inputs into

// these integer values and that converted the values into signed

// numbers. For the immediate data, I needed to extend the 16-bit

// value into a 32-bit value in order to be able to perform

// operations between my A and B inputs. If the instruction being

// executed was a signed instruction, then I sign-extended the

// most significant bit of the immediate data into the upper 16-bits

// of my integer value and then I added the immediate data, B[15:0]

// to the lower 16 bits of my integer value. If the instruction being

// executed was an unsigned instruction, then I simply set the upper

// 16-bits of a reg variable (unsigned entity) to zeros and set the

// lower 16-bits to the immediate data, B[15:0]. After initializing

// all of these variables, then I entered a case statement.

// This case statement decided which instruction to perform based

// on the sel input that goes into my ALU. The sel input, being 5-bits

// wide, is capable of selecting between 32 different instructions.

// For the purpose of this lab, however, I only needed to implement

// 18 instructions. The instructions within the ALU consisted of nop,

// add, addu, sub, subu, and, or, xor, nor, slt, sltu, addi, addiu,

// slti, sltu, andi, ori, and xori. These instructions either came in

// the form of unsigned, signed, bitwise, unsigned immediate, signed

// immediate, or bitwise immediate operations. Once I finished

// executing the desired instruction, my last step was to update the

// Zero Flag (ZF) by performing a reduction OR with the Y output and

// negating the result. This told me whether the output contained a

// value of zero or not. That is the gist of how my ALU module works.

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module ALU(sel, A, B, ZF, Y);

//Define Inputs

input [31:0] A, B;

input [4:0] sel;

//Define Outputs

output [31:0] Y;

reg [31:0] Y;

output ZF;

wire ZF;

//Local Variables

reg [31:0] B\_Immed\_Unsigned;

integer Aint, Bint, Yint;

always @(\*) begin

Aint = A; //Signed A value

Bint = B; //Signed B Value

B\_Immed\_Unsigned = {{16'b0}, B[15:0]}; //Unsigned Immediate value

case(sel)

5'b00000: ;//no op

5'b00001: begin //ADD Signed

Yint = Aint + Bint; // this will produce a signed result

Y = Yint; // the 32 bit signed results is in Y

end

5'b00010: Y = A + B; //ADD Unsigned

5'b00011: begin //Subtract Signed

Yint = Aint - Bint; // this will produce a signed result

Y = Yint; // the 32 bit signed results is in Y

end

5'b00100: Y = A - B; //Subtract Unsigned

5'b00101: Y = A & B; //Bitwise AND

5'b00110: Y = A | B; //Bitwise OR

5'b00111: Y = A ^ B; //Bitwise XOR

5'b01000: Y = ~(A | B); //Bitwise NOR

5'b01001: begin //Set to 1 if less than Signed

if (Aint < Bint)

Yint = 32'b01;

else

Yint = 32'b00;

Y = Yint;

end

5'b01010: begin //Set to 1 if less than Unsigned

if(A < B)

Y = 32'b01;

else

Y = 32'b00;

end

5'b01011: begin //Add Immediate

Yint = Aint + Bint;

Y = Yint; // the 32 bit signed results is in Y

end

5'b01100: Y = A + B\_Immed\_Unsigned; //Add Unsigned Immediate

5'b01101: begin //Set to 1 if less than Immediate

if (Aint < Bint)

Yint = 32'b01;

else

Yint = 32'b00;

Y = Yint;

end

5'b01110: begin //Set to 1 if less than Unsigned Immediate

if(A < B\_Immed\_Unsigned)

Y = 32'b01;

else

Y = 32'b00;

end

5'b01111: Y = A & B\_Immed\_Unsigned; //Bitwise AND Immediate

5'b10000: Y = A | B\_Immed\_Unsigned; //Bitwise OR Immediate

5'b10001: Y = A ^ B\_Immed\_Unsigned; //Bitwise XOR Immediate

5'b10010: Y = {B[15:0],16'b0}; //Load Upper Immediate

default: Y = 32'bx;

endcase

end//end always block

assign ZF = ~(|Y); //Assign the appropriate value to the zero

//flag.

endmodule