**Project 2 - Arithmetic Logic Unit**

Programmer: Victor Espinoza

SID: 010657450

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

Lab Section 6

Due: Thursday, February 26, 2015

**Introduction:**

The purpose of this lab was to create an Arithmetic Logic Unit (ALU). The ALU basically contains all of the underlying logic behind the instructions that I was required to implement for this project. Since there are no clock inputs for the ALU, this means that all of the operations performed within the ALU are combinational. Combinational logic is defined as performing arithmetic operations on bits of data (bitwise logic) and 0/1 logic (True/false logic). The fact that the ALU was using combinational logic also meant that I needed to use blocking assignments instead of non-blocking assignments within my always block. The different instructions that were implemented into my ALU were no operation (nop), add, add unsigned (addu), subtract (sub), subtract unsigned (subu), bitwise and, bitwise or, bitwise xor, bitwise nor, set to 1 if less than signed (slt), set to 1 if less than unsigned (sltu), add immediate (addi), add unsigned immediate (addiu), store to 1 if less than immediate (slti), store to 1 if less than unsigned immediate (sltiu), bitwise and immediate (andi), bitwise or immediate (ori), and bitwise xor immediate (xori).

**Project Description:**

In order to decide which instruction to perform, I used a 5-bit input called sel. I then created a case statement based on sel and executed the appropriate instruction based on the result of select. Since sel is a 5-bit number, there was a total of 32 different possible instructions to execute, but for the purpose of this lab I only needed to implemented the use of 18 different instructions. I also had two more inputs in my ALU module, A and B. The size of the A input was always 32-bits and the value contained inside of it was either signed or unsigned. The size of the B input was either 32-bits or 16-bits (for immediate data). The values within this input were also either signed or unsigned depending on which instruction was being executed. Finally, my ALU contained two outputs; a 1-bit value that represented the zero flag, ZF, and a 32-bit value, Y, that represented the result of performing the instructions within the ALU. That is what makes up my ALU.

Now I will elaborate on the concept of signed vs. unsigned values. Signed and unsigned values are different interpretations of the way in which 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, negative numbers cannot be represented. In other words, there is no such thing as a negative number for unsigned values; the value just keeps on getting bigger and bigger until it overflows to the beginning (0000\_0000). That being said, the initial values for both the A and B inputs for the ALU module were both unsigned. In order to perform the signed instructions in the ALU using these 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 from unsigned numbers into signed numbers. For the immediate data, I needed to extend the 16-bits of data going into the B input 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 my local 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 where I decided which instruction to execute (explained in previous paragraph). Once I finished executing the desired instruction, then my last step was to update the Zero Flag (ZF) by performing a reduction OR operation with the Y output and negating the result. This told me whether the output contained within the Y variable had a value of zero or not. That is the gist of how my ALU module works.

**Verification Description:**

For my test bench, after declaring all of the inputs, outputs, and local variables and instantiating the ALU file being tested, I first initialized all of my inputs to zero. After that, I then enter a for loop that lets me test each of the 18 individual instructions to verify their correctness. In the test bench, I came up with a series of three different test cases for each instruction in order test how the instructions handled different values. This helped me to verify that the instructions were working correctly under different circumstances. Thanks to the professor's test bench example, I was able to generate a test bench that is self checking, meaning that it knows the desired answer for each test case and displays the appropriate messages to let me know if an error occurred while testing the desired instruction. This made it alot easier to verify the correctness of my ALU module because humans often make many errors when trying to verify their work, and creating an automated message that lets the person testing the module know whether the instruction passed all of its test cases or not is way more efficient that going through the problems by hand every time you want to verify a module.

**Source Code:**

-Starts on next page

**ALU File:**

`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 consist 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;

reg ZF;

//Local Variables

integer Bint\_Immed\_Extended;

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

integer Aint, Bint, Yint;

always @(\*) begin

Aint = A; //Signed A value

Bint = B; //Signed B Value

Bint\_Immed\_Extended = {{16{B[15]}}, B[15:0]};//Sign-extended Immediate

//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; //Bitwize AND

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

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

5'b01000: Y = ~(A | B); //Bitwize 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\_Immed\_Extended;

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\_Immed\_Extended)

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; //Bitwize AND Immediate

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

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

default: ;

endcase

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

end

endmodule

**ALU File Test Bench:**

`timescale 1ns / 1ps

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

//

// Author: Victor Espinoza

// Email: victor.alfonso94@gmail.com

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

// Course: CECS 440

// Create Date: 1:29:37 2/21/2015

//

// Module Name: ALU\_tb

// File Name: ALU\_tb.v

//

// Description: This module creates a test bench used to test each individual

// instruction within the ALU and make sure that it is working

// properly. The instructions inside of the ALU consist of nop,

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

// addiu, slti, sltiu, andi, ori, and xori. In the test bench,

// I come up with a series of three different test cases for

// each instruction in order to verify that the instructions

// work correctly under different circumstances. Thanks to the

// professor's testbench example, I was able to generate a test

// bench that is self checking, meaning that it knows the desired

// answer for each test case and displays the appropriate messages

// to let me know if an error occurred while testing the desired

// instruction. This made it alot easier to verify the correctness

// of my ALU module because a message was displayed that let me know

// whether the instruction passed all of its test cases or not.

//

//

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

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

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

module ALU\_tb;

// Inputs

wire [ 4:0] sel;

wire [31:0] A;

wire [31:0] B;

// Outputs

wire [31:0] Y;

wire ZF;

// Variables

integer i,j,k;

assign A = j;

assign B = k;

assign sel = i;

// Instantiate the Unit Under Test (UUT)

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

ALU uut (

.sel(sel),

.A(A),

.B(B),

.ZF(ZF),

.Y(Y)

);

initial begin

// Initialize Inputs

i = 0;

j = 0;

k = 0;

// Add stimulus here: value = i, A = j, B = k

for (i=0; i<18; i=i+1)

case(i)

5'h0: begin // nop

$display("NOP");

j = 1234;

k = 5678;

#100 if (Y!=32'h0) $display("Error on NOP. sb: 0x00000000 is: %h",Y);

end

5'h1: begin // Add Signed

$display("Add");

//numbers less than 0000\_FFFF addition

j = 1200;

k = 6400;

#100 if (Y!=7600)

$display("Error on Add. A: %h B: %h sb: 7600 is:%h", A, B, Y);

//negative number addition

j = -250;

k = -750;

#100 if (Y!=32'hFFFFFC18)

$display("Error on Add. A: %h B: %h sb: 32'hFFFFFC18 is: %h",

A, B, Y);

//negative and positive number addition

j = -1024;

k = 2048;

#100 if (Y!=1024)

$display("Error on Add. A: %h B: %h sb: 1024 is: %h", A, B, Y);

end

5'h2: begin // Add Unsigned

$display("Add Unsigned");

//numbers bigger than 0000\_FFFF addition

j = 123456;

k = 678910;

#100 if (Y!=32'H000C3E3E)

$display("Error on Addu. A: %h B: %h sb: 32'H000C3E3E is: %h",

A, B, Y);

//numbers on different sides of the spectrum

j = 32'h7FFFFFFF;

k = 32'h00000012;

#100 if (Y!=32'H80000011)

$display("Error on Addu. A: %h B: %h sb: 32'H80000011 is: %h",

A, B, Y);

//overflow example

j = 32'hFFFFFFF0;

k = 32'h000000FF;

#100 if (Y!=32'H000000EF)

$display("Error on Addu. A: %h B: %h sb: 32'H000000EF is: %h",

A, B, Y);

end

5'h3: begin // Signed Subtract

$display("Subtract");

//causes a negative result

j = 123456;

k = 678910;

#100 if (Y!=32'HFFF78642)

$display("Error on Sub. A: %H B: %H sb: 32'HFFF78642 is: %H",

A, B, Y);

//causes a positive result

j = 32'hFFFFFFFA;

k = 32'hFFFFFFA5;

#100 if (Y!=32'H00000055)

$display("Error on Sub. A: %H B: %H sb: 32'H00000055 is: %H",

A, B, Y);

//causes a negative result

j = 32'hFFFFFFF0;

k = 32'h0000001B;

#100 if (Y!=32'HFFFFFFD5)

$display("Error on Sub. A: %H B: %H sb: 32'HFFFFFFD5 is: %H",

A, B, Y);

end

5'h4: begin // Unsigned Subtract

$display("Subtract Unsigned");

//numbers greater than 0000\_FFFF

j = 32'HFFFFFFFF;

k = 32'HAAAAAAAA;

#100 if (Y!=32'H55555555)

$display("Error on Subu. A: %H B: %H sb: 32'H55555555 is: %H",

A, B, Y);

//numbers close to each other

j = 32'hFFFFFFFA;

k = 32'hFFFFFF50;

#100 if (Y!=32'H000000AA)

$display("Error on Subu. A: %H B: %H sb: 32'H000000AA is: %H",

A, B, Y);

//numbers less than 0000\_FFFF

j = 32'h0000DDDD;

k = 32'h00000123;

#100 if (Y!=32'H0000DCBA)

$display("Error on Subu. A: %H B: %H sb: 32'H0000DCBA is: %H",

A, B, Y);

end

5'h5: begin // Bitwise AND

$display("AND");

//numbers have some bits in common

j = 32'HFEDC64A4;

k = 32'H0000161D;

#100 if (Y!=32'H00000404)

$display("Error on And. A: %H B: %H sb: 32'H00000404 is: %H",

A, B, Y);

//numbers have no bits in common

j = 32'h76543210;

k = 32'h01234567;

#100 if (Y!=32'H00000000)

$display("Error on And. A: %H B: %H sb: 32'H00000000 is: %H",

A, B, Y);

//all numbers have same bits in common

j = 32'hDDDDDDDD;

k = 32'hAAAAAAAA;

#100 if (Y!=32'H88888888)

$display("Error on And. A: %H B: %H sb: 32'H88888888 is: %H",

A, B, Y);

end

5'h6: begin // Bitwise OR

$display("OR");

//last 4 digits change

j = 32'HFEDCBA98;

k = 32'H00004567;

#100 if (Y!=32'HFEDCFFFF)

$display("Error on Or. A: %H B: %H sb: 32'HFEDCFFFF is: %H",

A, B, Y);

//most of digits change

j = 32'h76543210;

k = 32'h01234567;

#100 if (Y!=32'H77777777)

$display("Error on Or. A: %H B: %H sb: 32'H77777777 is: %H",

A, B, Y);

//all of the digits change

j = 32'h88888888;

k = 32'h22222222;

#100 if (Y!=32'HAAAAAAAA)

$display("Error on Or. A: %H B: %H sb: 32'HAAAAAAAA is: %H",

A, B, Y);

end

5'h7: begin // Bitwise XOR

$display("XOR");

//some bits are masked

j = 32'HFEDCBA09;

k = 32'H00001111;

#100 if (Y!=32'HFEDCAB18)

$display("Error on Xor. A: %H B: %H sb: 32'HFEDCAB18 is: %H",

A, B, Y);

//all bits are set

j = 32'h88888888;

k = 32'h77777777;

#100 if (Y!=32'HFFFFFFFF)

$display("Error on Xor. A: %H B: %H sb: 32'HFFFFFFFF is: %H",

A, B, Y);

//all bits are cleared

j = 32'h55555555;

k = 32'h55555555;

#100 if (Y!=32'H00000000)

$display("Error on Xor. A: %H B: %H sb: 32'H00000000 is: %H",

A, B, Y);

end

5'h8: begin // Bitwise NOR

$display("NOR");

//two different ways to complement value; either nor it with 0 or

//with itself

j = 32'HFEDCBA98;

k = 32'H0000BA98;

#100 if (Y!=32'H01234567)

$display("Error on Nor. A: %H B: %H sb: 32'H01234567 is: %H",

A, B, Y);

//inputs result in a value of 88888888 because this bit is the only

//bit that is not set when performing a nor operation on these two

//inputs.

j = 32'h76543210;

k = 32'h01234567;

#100 if (Y!=32'H88888888)

$display("Error on Nor. A: %H B: %H sb: 32'H88888888 is: %H",

A, B, Y);

//result is the complement of the inputs.

j = 32'hAAAAAAAA;

k = 32'hAAAAAAAA;

#100 if (Y!=32'H55555555)

$display("Error on Nor. A: %H B: %H sb: 32'H55555555 is: %H",

A, B, Y);

end

5'h9: begin // Set to 1 if Less Than Signed

$display("Set to 1 if < Signed");

//test a negative value to make sure that it sets

j = 32'H87654321;

k = 32'H7FFFFFFF;

#100 if (Y!=32'H00000001)

$display("Error on Set<1. A: %H B: %H sb: 32'H00000001 is: %H",

A, B, Y);

//test a value that is bigger than the second register

j = 32'h76543210;

k = 32'h01234567;

#100 if (Y!=32'H00000000)

$display("Error on Set<1. A: %H B: %H sb: 32'H00000000 is: %H",

A, B, Y);

//test a positive value that is smaller than the second register

j = 32'h12345678;

k = 32'h5A5A5A5A;

#100 if (Y!=32'H00000001)

$display("Error on Set<1. A: %H B: %H sb: 32'H00000001 is: %H",

A, B, Y);

end

5'hA: begin // Set to 1 if Less Than Unsigned

$display("Set to 1 if < unsigned");

//Use the same test cases as the ones in the slt instruction to prove

//the difference between the slt and sltu instructions.

//this value should be zero because 87654321 is bigger than 7FFFFFFF

//because the value is unsigned.

j = 32'H87654321;

k = 32'H7FFFFFFF;

#100 if (Y!=32'H00000000) $display

("Error on Set<1 unsigned. A: %H B: %H sb: 32'H00000000 is: %H",

A, B, Y);

//j value is bigger than the second register, so Y should equal 0.

j = 32'h76543210;

k = 32'h01234567;

#100 if (Y!=32'H00000000) $display

("Error on Set<1 unsigned. A: %H B: %H sb: 32'H00000000 is: %H",

A, B, Y);

//j value is smaller than the second register, so Y should be 1.

j = 32'h12345678;

k = 32'h5A5A5A5A;

#100 if (Y!=32'H00000001) $display

("Error on Set<1 unsigned. A: %H B: %H sb: 32'H00000001 is: %H",

A, B, Y);

end

5'hB: begin // Add Immediate Signed

$display("Add Immediate Signed");

//Add register A with a 16 bit immediate value (6400).

//The msb of k, 0, is sign extended to the upper 16 bits of the

//number and then the 16-bit immediate date is added to this to

//create our 32 bit B value.

j = 1200;

k = 6400;

#100 if (Y!=7600)

$display("Error on Addi. A: %h B: %h sb: 7600 is: %h", A, B, Y);

//negative number addition. Only the lower 16 bits of the k value

//are read into the B register. The msb of k, 1, is sign extended

//to the upper 16 bits of the number and then the 16-bit immediate

//date is added to this to create our 32 bit B value.

j = -250;

k = -750;

#100 if (Y!=32'hFFFFFC18)

$display("Error on Addi. A: %h B: %h sb: 32'hFFFFFC18 is: %h",

A, B, Y);

//negative and positive number addition. Only the lower 16 bits of

//the k value are read into the B register. The msb of k, 0, is sign

//extended to the upper 16 bits of the number and then the 16-bit

//immediate date is added to this to create our 32 bit B value.

j = -1024;

k = 2048;

#100 if (Y!=1024)

$display("Error on Addi. A: %h B: %h sb: 1024 is: %h", A, B, Y);

end

5'hC: begin // Add Immediate Unsigned

$display("Add Immediate Unsigned");

//For these values, only the lower 16 bits of the k value are read

//into register B. The upper 16 bits are set to all zeroes no matter

//what the value in the lower 16 bits, showing that the unsigned add

//immediate instruction is different from that of the signed

//immediate add instruction.

//add the same number together. Note, we only read the lower 16 bits

//of the k value.

j = 32'H12345678;

k = 32'H12345678;

#100 if (Y!=32'H1234ACF0)

$display("Error on Addu. A: %h B: %h sb: 32'H1234ACF0 is: %h",

A, B, Y);

//add A register with a 16-bit value.

j = 32'h7FFFFFFF;

k = 32'h00000012;

#100 if (Y!=32'H80000011)

$display("Error on Addu. A: %h B: %h sb: 32'H80000011 is: %h",

A, B, Y);

//cause an overflow to occur, which causes the numbers to roll over

//and start count from the beginning (00000000).

j = 32'hFFFFFFF0;

k = 32'h000000FF;

#100 if (Y!=32'H000000EF)

$display("Error on Addu. A: %h B: %h sb: 32'H000000EF is: %h",

A, B, Y);

end

5'hD: begin // Set to 1 if Less Than Signed

$display("Set to 1 if < Immediate Signed");

//test a negative value to make sure that it sets

//The msb of k, 1, is sign extended to the upper 16 bits of the

//number and then the 16-bit immediate date is added to this to

//create our 32 bit B value.

j = 32'H87654321;

k = 32'H7FFFFFFF;

#100 if (Y!=32'H00000001) $display

("Error on Set<1 Immediate. A: %H B: %H sb: 32'H00000001 is: %H",

A, B, Y);

//test a number bigger than B to make sure that y is set to 0.

//The msb of k, 1, is sign extended to the upper 16 bits of the

//number and then the 16-bit immediate date is added to this to

//create our 32 bit B value. J, being a positive number, is bigger

//than k, which is interpreted as a negative number once it is sign

//extended. This makes the output 0.

j = 32'h76543210;

k = 32'h0123F567;

#100 if (Y!=32'H00000000) $display

("Error on Set<1 Immediate. A: %H B: %H sb: 32'H00000000 is: %H",

A, B, Y);

//test a positive value that is smaller than the immediate value

//this sets Y equal to 1.

j = 32'h00005678;

k = 32'h23456789;

#100 if (Y!=32'H00000001) $display

("Error on Set<1 Immediate. A: %H B: %H sb: 32'H00000001 is: %H",

A, B, Y);

end

5'hE: begin // Set to 1 if Less Than Immediate Unsigned

$display("Set to 1 if < Immediate");

//The upper 16 bits of k are set to zeros and then the 16-bit

//immediate date is added to this to create our 32 bit B value.

//Compare a value that is larger than B, resulting in Y equaling 0.

j = 32'H87654321;

k = 32'H7FFFFFFF;

#100 if (Y!=32'H00000001)

$display("Error on Set<1 immediate unsigned.",

" A: %H B: %H sb: 32'H00000000 is: %H", A, B, Y);

//test a number smaller than k to make sure that Y equals 1.

j = 32'h0000AAAA;

k = 32'H0000FFFF;

#100 if (Y!=32'H00000001)

$display("Error on Set<1 immediate unsigned.",

" A: %H B: %H sb: 32'H00000001 is: %H",

A, B, Y);

//test another number smaller than k to make sure that Y equals 1.

j = 32'h0000E678;

k = 32'h0000F789;

#100 if (Y!=32'H00000001)

$display("Error on Set<1 immediate unsigned.",

" A: %H B: %H sb: 32'H00000001 is: %H", A, B, Y);

end

5'hF: begin // Bitwise AND Immediate

$display("AND Immediate");

//Since we are performing operations on the bits, we will use the

//unsigned representation of the immediate values. To do this, we

//set the upper 16 bits of k to zeros and then add the 16-bit

//immediate date to create the 32 bit B value.

//numbers have some bits in common

j = 32'HFEDC64A4;

k = 32'H0000161D;

#100 if (Y!=32'H00000404)

$display("Error on Andi. A: %H B: %H sb: 32'H00000404 is: %H",

A, B, Y);

//numbers have no bits in common

j = 32'h76543210;

k = 32'h01234567;

#100 if (Y!=32'H00000000)

$display("Error on Andi. A: %H B: %H sb: 32'H00000000 is: %H",

A, B, Y);

//all numbers have same bits in common

j = 32'hDDDDDDDD;

k = 32'hAAAAAAAA;

#100 if (Y!=32'H00008888)

$display("Error on Andi. A: %H B: %H sb: 32'H00008888 is: %H",

A, B, Y);

end

5'h10: begin // Bitwise OR Immediate

$display("OR Immediate");

//Since we are performing operations on the bits, we will use the

//unsigned representation of the immediate values. To do this, we

//set the upper 16 bits of k to zeros and then add the 16-bit

//immediate date to create the 32 bit B value.

//last 4 digits change

j = 32'HFEDCBA98;

k = 32'H00004567;

#100 if (Y!=32'HFEDCFFFF)

$display("Error on Ori. A: %H B: %H sb: 32'HFEDCFFFF is: %H",

A, B, Y);

//last 4 digits change

j = 32'h76543210;

k = 32'h01234567;

#100 if (Y!=32'H76547777)

$display("Error on Ori. A: %H B: %H sb: 32'H76547777 is: %H",

A, B, Y);

//nothing changes

j = 32'h88888888;

k = 32'hAAAA8888;

#100 if (Y!=32'H88888888)

$display("Error on Ori. A: %H B: %H sb: 32'H88888888 is: %H",

A, B, Y);

end

5'h11: begin // Bitwise XOR Immediate

$display("XOR Immediate");

//Since we are performing operations on the bits, we will use the

//unsigned representation of the immediate values. To do this, we

//set the upper 16 bits of k to zeros and then add the 16-bit

//immediate date to create the 32 bit B value.

//some bits are masked

j = 32'HFEDCBA09;

k = 32'H00001111;

#100 if (Y!=32'HFEDCAB18)

$display("Error on Xori. A: %H B: %H sb: 32'HFEDCAB18 is: %H",

A, B, Y);

//all bits in last 4 digits are set

j = 32'h88888888;

k = 32'h77777777;

#100 if (Y!=32'H8888FFFF)

$display("Error on Xori. A: %H B: %H sb: 32'H8888FFFF is: %H",

A, B, Y);

//all bits are cleared

j = 32'h55555555;

k = 32'h55555555;

#100 if (Y!=32'H55550000)

$display("Error on XoriA: %H B: %H sb: 32'H55550000 is: %H",

A, B, Y);

end

default: begin // nop for default operation

$display("NOP");

//do nothing for the no-op.

j = 1234;

k = 5678;

#100 if (Y!=32'h0) $display("Error on NOP. sb: 0x00000000 is: %h",Y);

end

endcase

end//for loop

endmodule