**Project 4 – Single Cycle MIPS with Sorting Code**

Programmer: Victor Espinoza

SID: 010657450

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

Lab Section 6

Due: Tuesday, April 7, 2015

**Introduction:**

The purpose of this lab was to build off of the Single Cycle MIPS Processor that I created in the previous lab and use it to solve a problem. For this problem, I was given an assortment of values filling data memory addresses 0x200 to 0x3FF. I then had to add the 32 largest values within this memory space together and store that value in data memory address 0x100. After this, I needed to add the 32 smallest values within this memory space together and store that value into data memory address 0x101. Finally, I needed to AND all of the values together, OR all of the values together, and perform a checksum on all of the data located within this address range and store those values in data memory addresses 0x102, 0x103, and 0x104, respectively. In order to do this, I needed to write the machine code to sort the data values first and then perform the respective operations on the data afterwards.

**Project Description:**

I split the machine code into three sections: sorting the data memory, adding the 32 highest and the 32 lowest values in the data memory together, and then ANDing, ORing, and performing a checksum on all of the values. I first wrote down the respective assembly code for each instruction and then converted them into machine code. Here is a description of my sorting algorithm.

For this particular lab, I used a bubble sorting algorithm to sort the values within the data memory (located within addresses 0x200 through 0x3FF) in descending order. This particular sorting algorithm passes through the list of data (data memory values in this case) two values at a time and then swaps those values if they are in the wrong position. Once it passes through the whole list, then it is known for a fact that the last element in the list is sorted into the correct position. After this, the sorting algorithm then repeats the process and then by the end of that pass the second to last element will be in the correct position. This process is repeated until all of the elements in the list are positioned in the correct location. In this way, this algorithm is distinguished as a comparison sort. This algorithm is very straightforward and simple to implement, but it is extremely inefficient and slow compared to the other sorting algorithms. For this reason, it is not very practical to use this sorting algorithm when the list contains a large amount of data within it. This particular sorting algorithm works really well if the list of data elements is almost sorted, but it is not very efficient if the data within the list is randomly stored or in reverse order. For this lab, I begin by starting at the base addresses of 0x200 and 0x201. This is the physical address of the data memory, so in order to match the addresses to the MIPS Program Counter address range, we need to shift these values left by 2, so the MIPS address range becomes 0x800 through 0xFFC. I then load the values located in these addresses (0x800 and 0x804 in MIPS address range) into two different registers using the load word (lw) instruction. I then determine if the first value is smaller than the second value using the set less than (slt) instruction. If this instruction is true, I then store the contents of each register into the appropriate address using the store word (sw) instruction to overwrite the values. I then increment these addresses and then compare addresses 0x804 and 0x808. I repeat this process until I reach the end of the data value range (0xFFC) and then I know that the last element is in the correct place. Now I simply decrement the inner counter value because the last value that the algorithm sorted is now in the correct place and no longer needs to be compared and then I then go back to the base address of 0x800 and start over using the branch not-equal (bne) instruction. When I am finished with this pass, then the second to last element is in the correct place. Once I am finished repeating this process 0x1FF times, then the data within the data memory range of 0x200 to 0x3FF (0x800 through 0xFFC for MIPS) is finally sorted in descending order. That is the gist of how my sorting algorithm works.

The next section of my assembly/machine code dealt with adding the 32 highest values in the data memory together and adding the lowest 32 values of the data memory together. The way that I sorted the data located within the data memory, the highest values were stored at the beginning of the data memory range (0x200/0x800) and the lowest values were stored at the end of the data memory range (0x3FF/0xFFC). Being that I sorted the values using the set less than signed instruction, the highest values in the data memory were those that were closer to 0x7FFFFFFF and the lowest values in the data memory were those that were closer to 0x80000000. In order to execute this part of the program, I first initialized all of my counters and zeroed out the registers that I was going to be using to store the sum of these values using the add immediate (addi) instruction. I then used the load word (lw) instruction to load the largest values into one register (R13) and the smallest values into another register (R14). I then added these values to the cumulative sum of the register holding the highest values (R11) and the register holding the smallest values (R12) using the add instruction. I then incremented the address of the largest values and decremented the address of the smallest values and then repeated the process until the largest 32 values were added together and the smallest 32 values were added together. As in the previous section, this loop was achieved using the branch not-equal (bne) instruction.

In order to calculate the correct value to use as an offset for the branch instruction, I needed to first determine the value of the next PC address. After determining this value, I then subtracted it from the value of the intended address that I wanted to branch to. This was the value that needed to be used as an offset while branching to that location. In my processor, however, I shift the address offset left by 2 when branching. In order to take this into account, I needed to shift the desired offset value right by 2 in the machine code so that when the MIPS processor inevitably shifted the offset left by 2, the offset value would be correct.

The final section of my assembly/machine code dealt with ANDing, ORing, and performing a checksum on all of the 512 data memory values that we were given. The process was very similar to how I performed the last section, although now I looped through the section 512 times instead of 32 times. I first initialized all of my counters to their respective values. Then I needed to create a seed value of 0xFFFFFFFF for the ANDing of all the values together. I did this by first loading in the upper 16 bits of register 19 with F's (0xFFFF0000) using the load upper immediate (lui) instruction. After doing this, I then ORed in the lower 16 bits worth of F's into register 19 using the OR immediate instruction (ori) and now R19 had a value of 0xFFFFFFFF. I then needed to create a seed value of 0x00000000 for the ORing of all the values together. I achieved this by simply adding the contents of R0, which are always 0x00000000, with the immediate data 0x0000, which resulted in my register (R20) having a value of 0x00000000. Next, I cleared out the contents of R21 in order to use this register to store the checksum value. Finally, I then ANDed, ORed, and performed a checksum on all of the values given to us in the data memory by looping 512 times using the branch not-equal (bne) instruction. Now all that was left to do was to store all of these calculated values into the appropriate data memory locations (data memory locations 0x100 - 0x104) using the store word instruction. I ended up with the following values:

32 Largest Values sum: f78155dd

32 Smallest Values sum: 23e4fe34

AND Every Location: 00000000

OR Every Location: ffffffff

Checksum: 58ae5216

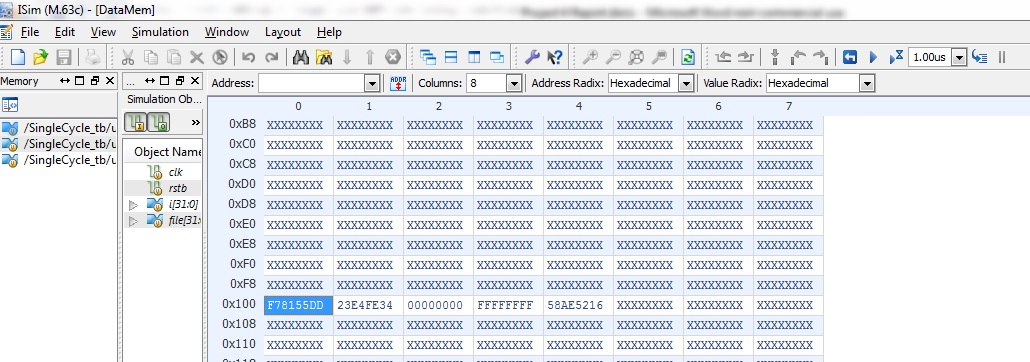
**Verification Description:**

In order to verify the correctness of my project, I used a combination of looking at the waveforms, writing data to a file, and looking at the physical addresses of the data memory to make sure that everything was sorted correctly. Looking at the waveforms was effective, but there were so many iterations and it turned out to be really inefficient looking at each instruction being executed every clock cycle. In order to remedy this, I was able to write helpful messages to a file every time a RegWrite or a MemWrite occurred (thanks to the help of the video that the professor posted!) so that I knew the address of the Program Counter and the register and contents that were being written to it. I also displayed the output of the final result to the console/wrote those values into the file when the program was done executing. I also made use of the memory column of the ISim program so that I could physically see the contents within the data memory. This made it alot easier for me to verify the correctness of my program and make sure that my sorting algorithm was working properly. Using all of these methods together, I was able to verify that my machine code was in fact correct. Starting on the next page are screenshots of the data memory followed by the excel sheet that contains all of the assembly/machine code that I wrote for this program and the source code for my MIPS Processor:

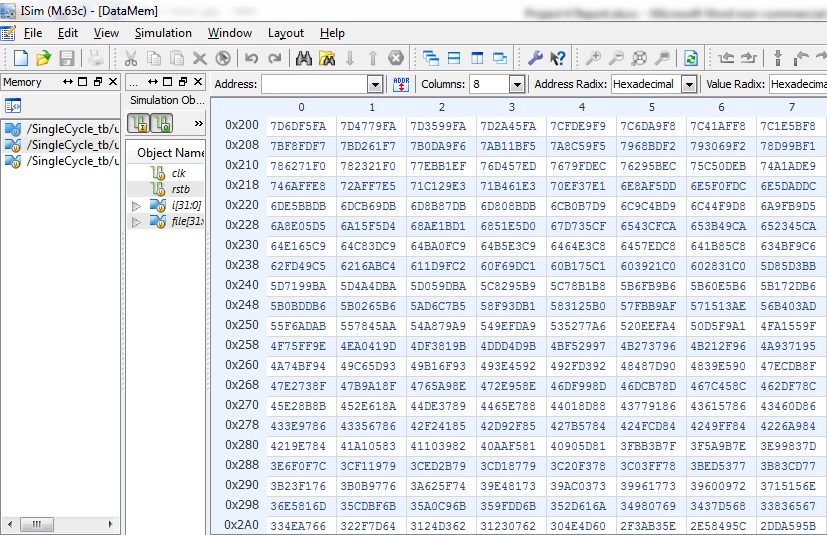
**Data Memory Contents/Machine Code/Source Code:**

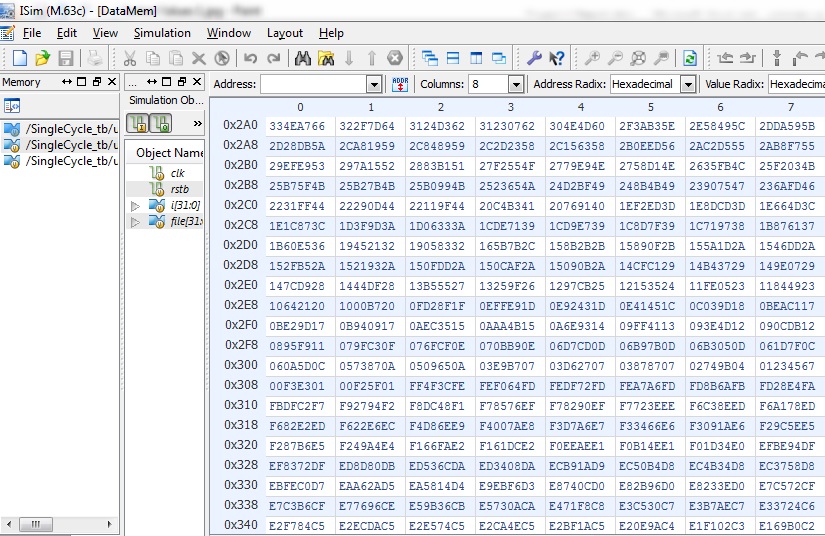
\*Begins on next page

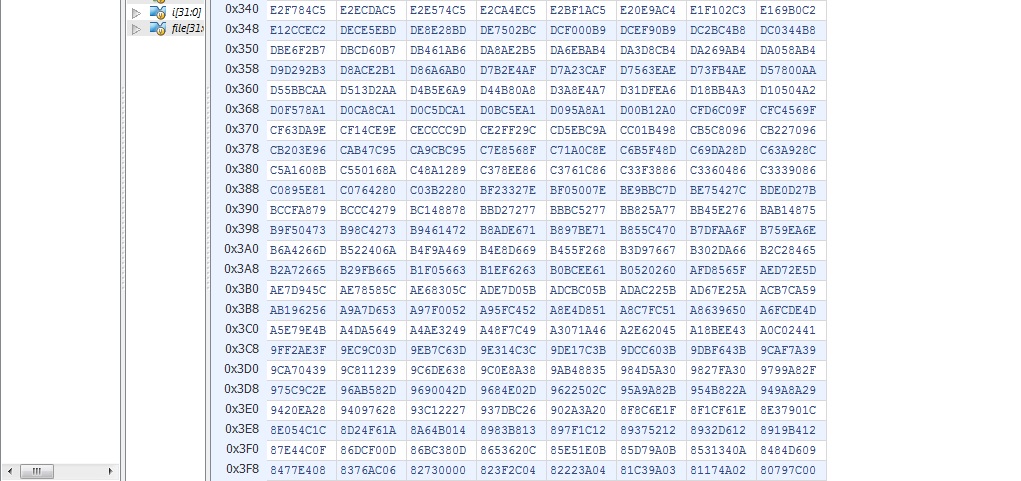
**Data Memory Outputs:**

****

**Sorted Data Memory:**







**Machine Code/Source Code:**

\*Starts on next page

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Addr** | **Instruction** | **Hex Machine Code** |  |  |  |  |  |  |  |  |  |
| 00 | addi $r1, $r0, 0x0001 | 20010001 |  |  |  |  | | |  |  |  |
| 04 | addi $r10, $r0, 0x01FF | 200A01FF |  |  |  |  | | |  |  |  |
| 08 | addi $r11, $r0, 0x01FF | 200B01FF |  |  |  |  | | |  |  |  |
| 0C | addi $r2, $r0, 0x0000 | 20020000 |  |  |  |  | | |  |  |  |
| 10 | addi $r3, $r0, 0x0000 | 20030000 |  |  |  |  | | |  |  |  |
| 14 | addi $r4, $r0, 0x0800 | 20040800 |  |  |  |  | | |  |  |  |
| 18 | lw $r5, $r4, 0x0000 | 8C850000 |  |  |  |  | | |  |  |  |
| 1C | lw $r6, $r4, 0x0004 | 8C860004 |  |  |  |  | | |  |  |  |
| 20 | slt $r7, $r5, $r6 | 00A6382A |  |  |  |  |  |  |  |  |  |
| 24 | bne $r7, $r1, 0x0008 | 14E10002 |  |  |  |  | | |  |  |  |
| 28 | sw $r5, $r4, 0x0004 | AC850004 |  |  |  |  | | |  |  |  |
| 2C | sw $r6, $r4, 0x0000 | AC860000 |  |  |  |  | | |  |  |  |
| 30 | addi $r4, $r4, 0x004 | 20840004 |  |  |  |  | | |  |  |  |
| 34 | addi $r3, $r3, 0x0001 | 20630001 |  |  |  |  | | |  |  |  |
| 38 | bne $r3, $r11, 0xFFDC | 146BFFF7 |  |  |  |  | | |  |  |  |
| 3C | addi $r2, $r2, 0x0001 | 20420001 |  |  |  |  | | |  |  |  |
| 40 | sub $r11, $r11, $r1 | 01615822 |  |  |  |  |  |  |  |  |  |
| 44 | bne $r2, $r10, 0xFFC8 | 144AFFF2 |  |  |  |  | | |  |  |  |
| 48 | addi $r8, $r0, 0x0800 | 20080800 |  |  |  |  | | |  |  |  |
| 4C | addi $r9, $r0, 0x0FFC | 20090FFC |  |  |  |  | | |  |  |  |
| 50 | addi $r10, $r0, 0x0020 | 200A0020 |  |  |  |  | | |  |  |  |
| 54 | addi $r15, $r0, 0x0000 | 200F0000 |  |  |  |  | | |  |  |  |
| 58 | addi $r11, $r0, 0x0000 | 200B0000 |  |  |  |  | | |  |  |  |
| 5C | addi $r12, $r0, 0x0000 | 200C0000 |  |  |  |  | | |  |  |  |
| 60 | lw $r13, $r8(0x0000) | 8D0D0000 |  |  |  |  | | |  |  |  |
| 64 | lw $r14, $r9(0x0000) | 8D2E0000 |  |  |  |  | | |  |  |  |
| 68 | add $r11, $r11, $r13 | 016D5820 |  |  |  |  |  |  |  |  |  |
| 6C | add $r12, $r12, $r14 | 018E6020 |  |  |  |  |  |  |  |  |  |
| 70 | addi $r8, $r8, 0x0004 | 21080004 |  |  |  |  | | |  |  |  |
| 74 | addi $r9, $r9, 0xFFFC | 2129FFFC |  |  |  |  | | |  |  |  |
| 78 | addi $r15, $r15, 0x0001 | 21EF0001 |  |  |  |  | | |  |  |  |
| 7C | bne $r10, $r15, 0xFFE0 | 154FFFF8 |  |  |  |  | | |  |  |  |
| 80 | addi $r16, $r0, 0x0000 | 20100000 |  |  |  |  | | |  |  |  |
| 84 | addi $r17, $r0, 0x0800 | 20110800 |  |  |  |  | | |  |  |  |
| 88 | addi $r18, $r0, 0x0200 | 20120200 |  |  |  |  | | |  |  |  |
| 8C | lui $r19, 0xFFFF | 3C13FFFF |  |  |  |  | | |  |  |  |
| 90 | ori $r19, $r19, 0xFFFF | 3673FFFF |  |  |  |  | | |  |  |  |
| 94 | addi $r20, $r0, 0x0000 | 20140000 |  |  |  |  | | |  |  |  |
| 98 | addi $r21, $r0, 0x0000 | 20150000 |  |  |  |  | | |  |  |  |
| 9C | lw $r22, $r17(0x0000) | 8E360000 |  |  |  |  | | |  |  |  |
| A0 | and $r19, $r19, $r22 | 02769824 |  |  |  |  |  |  |  |  |  |
| A4 | or $r20, $r20, $r22 | 0296A025 |  |  |  |  |  |  |  |  |  |
| A8 | addu $r21, $r21, $r22 | 02B6A821 |  |  |  |  |  |  |  |  |  |
| AC | addi $r17, $r17, 0x0004 | 22310004 |  |  |  |  | | |  |  |  |
| B0 | addi $r16, $r16, 0x0001 | 22100001 |  |  |  |  | | |  |  |  |
| B4 | bne $r16, $r18, 0xFFE4 | 1612FFF9 |  |  |  |  | | |  |  |  |
| B8 | addi $r30, $r0, 0x0400 | 201E0400 |  |  |  |  | | |  |  |  |
| BC | sw $r11, $r30(0x0000) | AFCB0000 |  |  |  |  | | |  |  |  |
| C0 | sw $r12, $r30(0x0004) | AFCC0004 |  |  |  |  | | |  |  |  |
| C4 | sw $r19, $r30(0x0008) | AFD30008 |  |  |  |  | | |  |  |  |
| C8 | sw $r20, $r30(0x000C) | AFD4000C |  |  |  |  | | |  |  |  |
| CC | sw $r21, $r30(0x0010) | AFD50010 |  |  |  |  | | |  |  |  |
| D0 | beq $r0, $r0, 0xFFFC | 1000FFFF |  |  |  |  | | |  |  |  |

`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 begin

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

$readmemh("DataMem", uut.DataMem);

end

integer file;

initial begin

//open the file that you are going to be writing messages to

file = $fopen("outputFileMessages");

// Initialize Inputs

clk = 0;

rstb = 0;

// Wait 100 ns for global reset to finish

#100;

@(posedge clk)rstb = 1;

end

always@(negedge clk) begin

$fwrite(file,"PC = %h ", uut.PC);

if(uut.RegWrite)

$fdisplay(file,"Reg = %h Value = %h",uut.wr\_addr\_muxed, uut.wr\_data\_muxed);

else

$fdisplay(file, " ");

if(uut.MemWrite)begin

$fdisplay(file, "Wr\_addr being changed = %h Wr\_data being changed= %h", (uut.ALU\_out>>2), uut.DataMem[uut.ALU\_out>>2]);

end

if(uut.PC == 32'h000000D0) begin

$display(" 32 Largest Values sum: %h", uut.DataMem[10'h100]);

$fdisplay(file, " 32 Largest Values sum: %h", uut.DataMem[10'h100]);

$display("32 Smallest Values sum: %h", uut.DataMem[10'h101]);

$fdisplay(file, "32 Smallest Values sum: %h", uut.DataMem[10'h101]);

$display(" AND Every Location: %h", uut.DataMem[10'h102]);

$fdisplay(file, " AND Every Location:%h", uut.DataMem[10'h102]);

$display(" OR Every Location: %h", uut.DataMem[10'h103]);

$fdisplay(file, " OR Every Location: %h", uut.DataMem[10'h103]);

$display(" Checksum: %h", uut.DataMem[10'h104]);

$fdisplay(file, " Checksum: %h", uut.DataMem[10'h104]);

$stop;

end

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)

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 @(\*)

begin

{ALUCtl,Ctl} = {5'h00,8'h00};

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'h03}; // ADDI

({6'h09,6'h??}): {ALUCtl,Ctl} = {5'h0C,8'h03}; // 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'h1B}; // LW

({6'h2B,6'h??}): {ALUCtl,Ctl} = {5'h01,8'h86}; // SW

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

endcase

end

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