UNIVERSITY OF NEVADA LAS VEGAS. DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING LABORATORIES.

| Class: | CPE300L | | | Semester: | Fall 2021 |
|---|---|---|---|---|---|
| | | | | | |
| Points | | Document author: | Jerrod Batu | | |
| | | Author's email: | batuj1@unlv.nevada.edu | | |
| | | | | | |
| | | Document topic: | Postlab 10 | | |
| Instructor's comments: | | | | | |

## 1. Introduction / Theory of Operation

In this lab, I will become familiar with designing a complete CPU. This will be done by implementing and experimenting with single cycle implementation of a limited subset of MIPS instructions.

a. The **soft processor** is a processor that can be implemented entirely by using logic synthesis. In fact, most systems utilize a single soft processor, and they are implemented in FPGA fabric. They can be easily modified and altered to specific needs, features, customizable instructions, et.c Multiple cores can be used for a soft processor, and one down side to these processors is the speed of the fabric. Compared to **hard processors**, they are several disadvantages to the soft processor. Hard processors can use up to 1GHz or more speed while soft processors are limited to 250MHz and less. In addition, hard processors are much faster as they are optimized by not being limited by the fabric speed. One downside to these hard processors is that they are fixed and cannot be modified. Also, hard processors are implemented to an integrated circuit, but they are connected to the FPGA fabric.

b. The main **difference** between single cycle MIPS processor and multicycle MIPS processor are within the clock cycles. Although single cycle MIPS processors are simple, and can be easily integrated, they are inefficient due to all of the instructions having the same clock cycle length. Basically, the instructions all take the same amount of time independent from what they actually process. Multicycle MIPS processors are much faster because the amount of cycles per instruction is not limited by the critical path delay; therefore, it has a faster clock rate.

## 2. Prelab

https://docs.google.com/document/d/15eEaIPVXo2YMEBWTmh03Y7VzLegkVnQv/edit?usp=sharing&ouid=102808507017671072128&rtpof=true&sd=true

This is the link to my prelab 10.

**3. Results of Experiments**

Experiment 1

```
module jal (input clk, reset,output [31:0] writedata, dataadr,output memwrite,output [6:0] seg1,seg2);                    //
seg1 and seg2 added for 7seg display

        wire [31:0] pc, instr, readdata;

        // instantiate processor and memories
        mips mips (clk, reset, pc, instr, memwrite, dataadr,
                                writedata, readdata);
        imem imem (pc[7:2], instr);
        dmem dmem (clk, memwrite, dataadr, writedata,readdata);
        seg7 display (pc[7:0], seg1, seg2);
                                                        // 7 seg display instantiation
endmodule

module mips (input clk, reset,
                        output [31:0] pc,
                        input [31:0] instr,
                        output memwrite,
                        output [31:0] aluout, writedata,
                        input [31:0] readdata);

        wire memtoreg, branch, alusrc, regdst, regwrite, jr, jump, ori, lui, jal, zero, pcsrc;
        //ori and jr and lui and jal wire added
        wire [2:0] alucontrol;

        controller c(instr[31:26], instr[5:0], zero,memtoreg, memwrite, pcsrc,
                                alusrc, regdst, regwrite, jr, jump, ori, lui, jal, alucontrol);
                        // ori and jr and lui and jal wire added to controller
        datapath dp(clk, reset, memtoreg, pcsrc,
                                alusrc, regdst, regwrite, jr, jump, ori, lui, jal,
                                        // ori and jr and lui and jal wire added to DP
                                alucontrol,
                                zero, pc, instr,
                                aluout, writedata, readdata);
endmodule

module imem (input [5:0] a, output [31:0] rd);

        reg [31:0] RAM[63:0]; // limited memory

        initial
```

```
        begin
            $readmemh ("memfile.dat",RAM);
        end
    assign rd = RAM[a]; // word aligned
endmodule

module dmem (input clk, we,
                input [31:0] a, wd,
                output [31:0] rd);

    reg [31:0] RAM[63:0];

    assign rd = RAM[a[31:2]]; // word aligned
    always @ (posedge clk)
        if (we)
            RAM[a[31:2]] <= wd;
endmodule

module controller (input [5:0] op, funct,
                        input zero,
                        output memtoreg, memwrite,
                        output pcsrc, alusrc,
                        output regdst, regwrite,
                        output jr, jump, ori, lui, jal,
                            // ori and jr and lui and jal signal added to controller
                        output [2:0] alucontrol);

    wire [1:0] aluop;
    wire branch;

    maindec md(op, funct, memtoreg, memwrite, branch,
                            // funct port added to main decoder
                alusrc, regdst, regwrite, jr, jump, ori, lui,   jal,
                    // ori and jr and lui and jal signal added to list of ports for main decoder
                aluop);
    aludec ad (funct, aluop, alucontrol);
    assign pcsrc = branch & zero;
endmodule

module datapath (input clk, reset,
                input memtoreg, pcsrc,
                input alusrc, regdst,
                input regwrite, jr, jump, ori, lui, jal,
                        // ori and jr and lui and jal signal added
```

```
                              input [2:0] alucontrol,
                              output zero,
                              output [31:0] pc,
                              input [31:0] instr,
                              output [31:0] aluout, writedata,
                              input [31:0] readdata);

wire [4:0] writereg, raout;
                                                              // ramux output added
wire [31:0] pcnext, pcnextbr, pcplus4, pcbranch, pcnextjr;
                                  // pcnextjr wire added
wire [31:0] signimm, signimmsh, zeroimm, ezimm;
                                                // zero immediate and extend zero wire added
wire [31:0] srca, srcb;
wire [31:0] result, zeout, ezout, jalout;
                                                // zero extend mux and extend zero and jal mux output wire added

// next PC logic
flopr #(32) pcreg(clk, reset, pcnext, pc);
adder pcadd1 (pc, 32'b100, pcplus4);
sl2 immsh(signimm, signimmsh);
adder pcadd2(pcplus4, signimmsh, pcbranch);
mux2 #(32) pcbrmux(pcplus4, pcbranch, pcsrc, pcnextjr);
                                          // pcbrmux output is now pcnextjr
mux2 #(32) pcjrmux (pcnextjr, srca, jr, pcnextbr);
                                          // pcjrmux added
mux2 #(32) pcmux(pcnextbr, {pcplus4[31:28], instr[25:0], 2'b00},jump, pcnext);

// register file logic
regfile rf(clk, regwrite, instr[25:21],
instr[20:16], raout, jalout, srca, writedata);
                                          // change result input to extend zero mux output
mux2 #(5) wrmux(instr[20:16], instr[15:11],regdst, writereg);
mux2 #(5) ramux (writereg,5'b11111, jal, raout);
                                          // ramux to jump to ra register
mux2 #(32) resmux(aluout, readdata, memtoreg, result);
extzero ez(instr[15:0], ezimm);
                                                              // extend zero added to DP
signext se(instr[15:0], signimm);
mux2 #(32) ezmux(result,ezimm,lui,ezout);
                                                //extend zero mux added to DP
mux2 #(32) jalmux (ezout, pcplus4, jal, jalout);
zeroext ze(instr[15:0], zeroimm);
                                                        // zero extend mux added to DP
```

```
	// ALU logic
	mux2 #(32) zemux (signimm, zeroimm, ori, zeout);
						// zero extend mux added to DP
	mux2 #(32) srcbmux(writedata, zeout, alusrc, srcb);
						// d1 changes to be output of zero extend mux
	alu alu(srca, srcb, alucontrol, aluout, zero);
endmodule

module maindec (input [5:0] op, funct, output memtoreg, memwrite, output branch, alusrc,
		// funct field added to main decoder
				output regdst, regwrite, output jr, jump, ori, lui, jal, output [1:0] aluop);
		// ori and jr and lui and jal output signal added

	reg [12:0] controls;
									// controls register becomes 13-bits because of ori and
jr and jal signal
	assign {regwrite, regdst, alusrc, branch, memwrite, memtoreg, jr, jump, ori, lui, jal, aluop}  = controls;    // ori and
jr and lui and jal output added to list of controls

	always @ (* )
		case(op)
			6'b000000 :  begin
										//if statement to differentiate between RTYPE
and JR
						if (funct == 6'b001000)
							controls <= 13'b0000001000010;      //JR
							//jr control signal added with same op code as Rtyp
						else
							controls <= 13'b1100000000010;      //Rtyp
							//now becomes 10-bits wide
					end
			6'b100011 :  controls <= 13'b1010010000000;			//LW
					//now becomes 10-bits wide
			6'b101011 :  controls <= 13'b0010100000000;			//SW
					//now becomes 10-bits wide
			6'b000100 :  controls <= 13'b0001000000001;			//BEQ
					//now becomes 10-bits wide
			6'b001000 :  controls <= 13'b1010000000000;			//ADDI
						//now becomes 10-bits wide
			6'b000010 :  controls <= 13'b0000000100000;			//J
						//now becomes 10-bits wide
			6'b001101 :  controls <= 13'b1010000010011;			//ORI
						//ori signal added to controls
```

```
                6'b001111 :  controls <= 13'b1000000001000;                  //LUI
                                        //lui signal added to controls
                6'b000011 :  controls <= 13'b1000000100100;                  //JAL
                                        //jal signal added to controls
                default:  controls <= 13'bXXXXXXXXXXXXX;             //???
                                        //becomes 10-bits wide
     endcase
endmodule

module aludec (input [5:0] funct,
                 input [1:0] aluop,
                 output reg [2:0] alucontrol);

     always @ (*)
          case (aluop)
                2'b00: alucontrol <= 3'b010; // add
                2'b01: alucontrol <= 3'b110; // sub
                default: case(funct) // RTYPE
                     6'b100000: alucontrol <= 3'b010; // ADD
                     6'b100010: alucontrol <= 3'b110; // SUB
                     6'b100100: alucontrol <= 3'b000; // AND
                     6'b100101: alucontrol <= 3'b001; // OR
                     6'b101010: alucontrol <= 3'b111; // SLT
                     6'b000100: alucontrol <= 3'b101; // SLLV
                                        //SLLV added to alu decoder
                     default: alucontrol <= 3'bxxx; // ???
                endcase
                2'b11: alucontrol <= 3'b001;   // ori
                                        //ori control added to alu decoder
          endcase
endmodule

module flopr # (parameter WIDTH = 8)
     (input clk, reset,
      input [WIDTH-1:0] d,
      output reg [WIDTH-1:0] q);

      always @ (posedge clk, posedge reset)
          if (reset) q <= 0;
          else q <= d;
endmodule

module adder (input [31:0] a, b, output [31:0] y);
```

```
	assign y = a + b;
endmodule

module sl2 (input [31:0] a, output [31:0] y);

	// shift left by 2
	assign y = {a[25:0], 2'b00};
endmodule

module mux2 # (parameter WIDTH = 8)
	(input [WIDTH-1:0] d0, d1, input s,
	 output [WIDTH-1:0] y);

	assign y = s ? d1 : d0;
endmodule

module regfile (input clk, input we3,
					input [4:0] ra1, ra2, wa3,
					input [31:0] wd3,
					output [31:0] rd1, rd2);

	reg [31:0] rf[31:0];

	// three ported register file
	// read two ports combinationally
	// write third port on rising edge of clock
	// register 0 hardwired to 0
	always @ (posedge clk)
		if (we3) rf[wa3] <= wd3;
			assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
			assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule

module signext (input [15:0] a,
				output [31:0] y);

	assign y = {{16{a[15]}}, a};
endmodule

module zeroext (input [15:0] a,
										// zero extend created
				output [31:0] y);

	assign y = {16'b0000000000000000, a};
```

```
endmodule

module extzero (input [15:0] a,
                                                                                    //extend zero circuit added
                                    output [31:0] y);
         assign y = {a, 16'b0000000000000000};
endmodule

module alu (a,b,sel, out, zero);
   input [31:0] a,b;
   input [2:0] sel;
   output reg [31:0] out;
          output reg zero;

          initial
                  begin
                           out = 0;
                           zero =1'b0;
                  end

          always @ (*)
                  begin
      case(sel)
         3'b000:
                                             begin
                                                      out=a & b;
                                                      if (out == 0)
                                                               zero = 1;
                                                      else
                                                               zero = 0;
                                             end
         3'b001:
                                             begin
                                                      out= a | b;
                                                      if (out == 0)
                                                               zero = 1;
                                                      else
                                                               zero = 0;
                                    end
         3'b110:
                                             begin
                                                      out=a-b;
                                                      if (out == 0)
                                                               zero = 1;
```

```
                                        else
                                                zero = 0;
                                end
        3'b010:
                                begin
                                        out=a+b;
                                        if (out == 0)
                                                zero = 1;
                                        else
                                                zero = 0;
                                end
        3'b111:
                            begin
                                        if ( a < b)
                                                out = 1;
                                        else
                                                out=0;
                                end
                    3'b101:
                                                                                    //add check for left shift
                            begin
                                 out = b << a;
                                                                            //shift logic added to ALU
                                 if (out == 0)
                                         zero = 1;
                                 else
                                         zero = 0;
                            end
      endcase
   end
endmodule

// 7 Seg Display
module seg7 (out, segments1, segments2);
                                                // seg7 module created
        input [7:0] out;
        output reg [6:0] segments1, segments2;

        always @ (out) begin
                    case (out)
                          0 : begin                                                              // 0
                                        segments1 = 7'b0000001;
                                        segments2 = 7'b0000001;
                                 end
```

```
4 : begin                                        // 4
        segments1 = 7'b1001100;
        segments2 = 7'b0000001;
    end
8 : begin                                        // 8
        segments1 = 7'b0000000;
        segments2 = 7'b0000001;
    end
12 : begin                                       // c
        segments1 = 7'b0110001;
        segments2 = 7'b0000001;
    end
16 : begin                                       // 10
        segments1 = 7'b0000001;
        segments2 = 7'b1001111;
    end
20 : begin                                       // 14
        segments1 = 7'b1001100;
        segments2 = 7'b1001111;
    end
24 : begin                                       // 18
        segments1 = 7'b0000000;
        segments2 = 7'b1001111;
    end
28 : begin                                       // 1c
        segments1 = 7'b0110001;
        segments2 = 7'b1001111;
    end
32 : begin                                       // 20
        segments1 = 7'b0000001;
        segments2 = 7'b0010010;
    end
36 : begin                                       // 24
        segments1 = 7'b1001100;
        segments2 = 7'b0010010;
    end
40 : begin                                       // 28
        segments1 = 7'b0000000;
        segments2 = 7'b0010010;
    end
44 : begin                                       // 2c
        segments1 = 7'b0110001;
        segments2 = 7'b0010010;
    end
```

```
				48 : begin								// 30
							segments1 = 7'b0000001;
							segments2 = 7'b0000110;
						end
				52 : begin								// 34
							segments1 = 7'b1001100;
							segments2 = 7'b0000110;
						end
				56 : begin								// 38
							segments1 = 7'b0000000;
							segments2 = 7'b0000110;
						end
				60 : begin								// 3c
							segments1 = 7'b0110001;
							segments2 = 7'b0000110;
						end
				64 : begin								// 40
							segments1 = 7'b0000001;
							segments2 = 7'b1001100;
						end
				68 : begin								// 44
							segments1 = 7'b1001100;
							segments2 = 7'b1001100;
						end
				default:
							begin
								segments1 = 7'b1111111;
								segments2 = 7'b1111111;
							end
			endcase
		end
endmodule
```

This is my verilog code for the JAL instruction added to the single cycle MIPS processor. Along with the JAL instruction, ORI, JR, SLLV, and LUI instructions are also added as they were done for homework 9 in the CPE 300 lecture. All of the comments would describe the changes made to the single cycle MIPS processor.

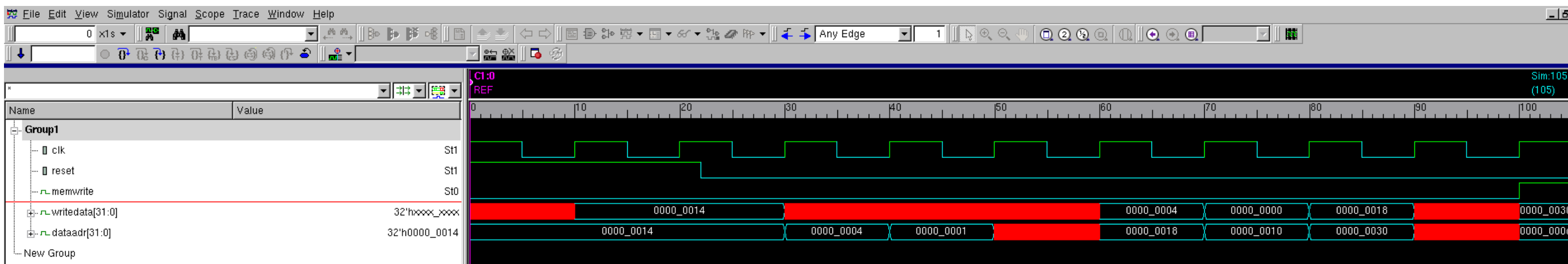
Experiment 2

a.

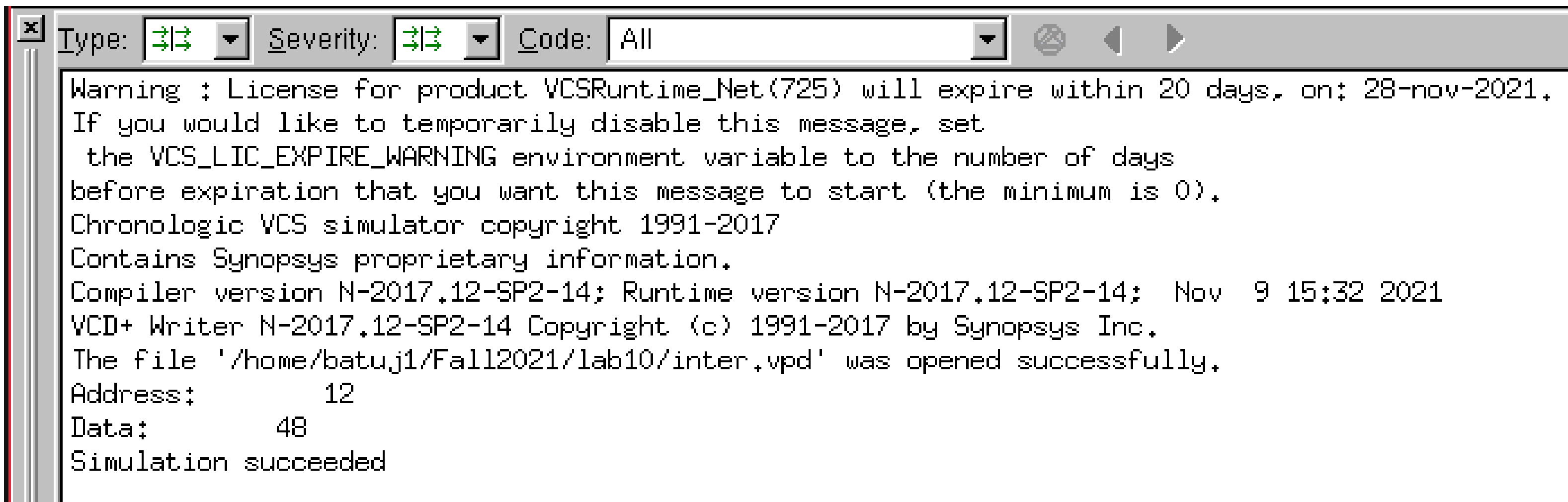
| 20040014 |
|---|
| 20080004 |
| 200c0001 |
| 0c100006 |
| 01821004 |
| 08100008 |
| 00881020 |
| 03e00008 |
| ad020008 |

This is my MIPS code that would provide instructions for (a + b) * 2 to be properly outputted. The two numbers I chose are 20 (0x14) for a and 4 (0x4) for b. The final result displayed is 48 as (20 + 4) * 2 = 48.

b.



These are my VCS waveforms for the single cycle MIPS processor including the JAL instruction.




This is my VCS console for the single cycle MIPS processor including the JAL instruction. I modified the testbench to display the address and its contents in order to get a successful simulation.

c. https://drive.google.com/file/d/1XlJFuOLnNe-Wgr99oDKa-gshAbW-2DH8/view?usp=sharing
This is the link to my video describing the operation of (a + b) * 2 with the DE2 implementation of the single cycle MIPS processor including the JAL instruction.

Experiment 3

a.

```
20010001
20020001
20030040
20040001
00821004
10620001
08000004
00821006
1022fffc
08000007
```

This is my MIPS code that would provide instructions for a rotating light to be properly outputted onto the DE2 board. I utilized the SLLV instruction for the LED light to rotate to the left, and I created a SRLV instruction for the LED light to rotate to the right. The one second interval is where a BEQ instruction is used to check to shift left or right depending on what state the LED is at.

b. https://drive.google.com/file/d/1XmyXf7R3e-20DHnC6A8Z6JevMc5kaHCh/view?usp=sharing
This is the link to my video explaining how my single cycle MIPS processor code will implement MIPS code for a rotating light on the DE2 board.

## 4. Answers to questions

Question 1:

There are several ways to implement delay for experiment 3. The more instructions placed in between the branch and jump instructions, the more delay there would be. For example, by placing 2 addi instructions in between branch and jump that would be a 4 second delay. By placing 5 addi instructions in between branch and jump, there would be a 10 second delay. These instructions would not be taken into consideration because they would hold 0 value; therefore, one of the best methods to implement a 10 second delay would be to include any 0 value instructions that would occur between the branch and jump instructions.

Question 2:
In single cycle MIPS, the jump instruction goes to a specified address in the program counter without returning any value. This is similar to a void function in C++ programming language. In addition, the jump instruction requires one multiplexer that connects to the PC counter. In comparison, the JAL (jump and link) instruction jumps to a specified address in the program counter and saves the return address in ra. This is similar to a return function in C++ programming language. JAL in single cycle mips is implemented with a JR mux as well as a link mux. This is because the JR function will jump to the register and the link function will link the register with the return address in ra.

## 5. Conclusions & Summary

This lab was the hardest for CPE 300 L because I am fairly new to single cycle MIPS. I encountered several issues with experiment one because I did not know how to properly create JR nor JAL. I got some assistance from the TA, and a few of my classmates on how to approach experiment one. I realized that the main issue occurred with my datapath as my $ra register was not properly initialized. At first I had it at 32'b01111, which would not be the proper constant value for $ra; therefore, my classmate pointed out to change it to 5'b11111 as this is how $ra is called. In addition, I was having some struggles with experiment two on how to handle the jr $ra. In order to fix this problem, I had to watch the MIPSII.mp4 video over and over to fully understand how jr $ra works. Last, experiment three I had some trouble with. I did not know how to properly rotate the LEDs, and I initially only utilized addi and sub instructions. I figured out that looping and branching would cause the LEDs to be stuck within a loop to properly rotate within a one-second interval. In addition, I had to reread what the experiment was asking for because I originally had the code increment and decrement by one rather than shifting left and shifting right. In the end, this was the hardest lab of the semester, but I am becoming more and more familiar with single cycle MIPS processors through practice.