ECE 469 Spring 2024 Lab 7: MIPS Multi-Cycle CPU

You can form a group of 1-3 people.

Grading components:

- **(40%) Demo - to TA during week 13 and 14 Fridays**:
  - Week 13: Complete Table 4 and 5.
  - Week 14: Demonstrate Part A.
- **(60%) Lab Report deadline is week 15 Mon end of the day on gradescope:**

  Submission components:

  1. A completed copy of Table 4.
  2. A complete copy of Table 5 (one for Part B, and one for Part C) indicating the expected outcome of running the test programs.
  3. SystemVerilog code of both control and datapath.
  4. Simulation waveforms of the processor showing *CLK*, *PC*, *Reset*, *state*, *instr*, and *ALUResult* in this order while running the test program. As always, output the values in hex or decimal (whichever is more relevant) and make sure they are clearly readable. Do the results match your expectations? Does the program indicate Simulation Succeeded?
  5. RTL view from Quartus' compilation result of your MIPS processor.
  6. Workload report: How many hours have you spent for this lab entirely? Which activity takes the most significant amount of time? This will not affect your grade (unless omitted).

Introduction

In this part, you will design and build your own **multicycle MIPS processor**. You will be much more on your own to complete this lab than you have been in the past, but you may reuse any of your hardware (SystemVerilog modules) from previous ones.

Your multicycle processor should match the design from the text, which is reprinted in Figure 1 for your convenience. It should handle the following instructions: `{add, sub, and, or, slt, lw, sw, beq, addi, j}` The multicycle processor is divided into three units: the `controller`, `datapath`, and `mem` (memory) units. Note that the `mem` unit contains the shared memory used to hold both data and instructions. Also note that the `controller` unit comprises both the `Main Decoder` that takes $OP_{5:0}$ as inputs and the `ALU Decoder` that takes as inputs $ALUOp_{1:0}$ and the $Funct_{5:0}$ code from the 6 least significant bits of the instruction. The `controller` unit also includes the gates needed to produce the write enable signal, `PCEn`, for the PC register.

Part A: Controller

1. Controller Design

Generating Control Signals

Before you begin developing the hardware for your MIPS multicycle processor, you'll need to determine the correct control signals for each state in the multicycle processor's state transition diagram. This state transition diagram is shown in Figure 7.42 in the book. Complete the output table of the Main Decoder in Table 4. Give the FSM control word in hexadecimal for each state. The first two rows are filled in as examples. Be careful with this step. It takes much longer to debug an erroneous circuit than to design it correctly the first time.

Overall Design

Now you will begin the hardware implementation of your multicycle processor. Read and understand the `mipsmulti.sv` code at the Appendix section, as it provides a framework for your design:

- The `mips` module instantiates both the `datapath` and control unit (called the `controller` module). The controller module in turn instantiates the main decoder module (`maindec`) and the ALU decoder module (`aludec`).
- You will be responsible of designing the controller and the datapath.
- The memory is essentially identical to the data memory from Lab 5, as provided for you.

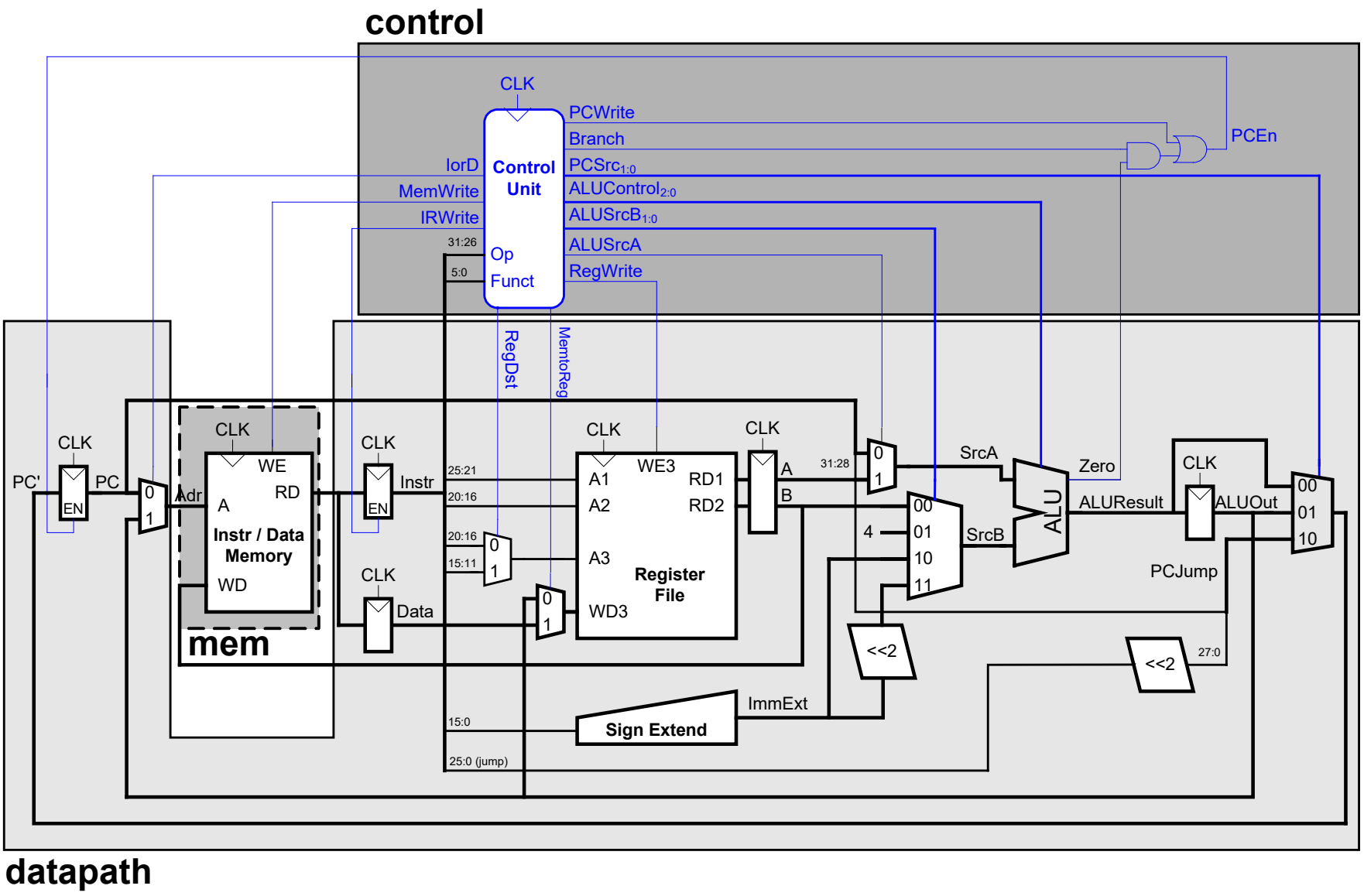


Figure 1. **Multicycle Processer**

Unit Overview

The three units have the following inputs and outputs. Although the signal names are in upper case here to match the diagram, remember to use lower case for all names in your SystemVerilog files.

Table 1. Controller

CLK		Input
Reset		Input
Op	[5:0]	Input
Funct	[5:0]	Input
Zero		Input
IorD		Output
MemWrite		Output
IRWrite		Output
RegDst		Output
MemtoReg		Output
RegWrite		Output
ALUSrcA		Output
ALUSrcB	[1:0]	Output
ALUControl	[2:0]	Output
PCSrc	[1:0]	Output
PCEn		Output

Note that *PCWrite* and *Branch* are internal signals within the controller.

Table 2. Datapath

CLK		Input
Reset		Input
PCEn		Input
IorD		Input
IRWrite		Input
RegDst		Input
MemtoReg		Input
RegWrite		Input
ALUSrcA		Input
ALUSrcB	[1:0]	Input
ALUControl	[2:0]	Input
PCSrc	[1:0]	Input
ReadData	[31:0]	Input
Op	[5:0]	Output
Funct	[5:0]	Output
Zero		Output
Adr	[5:0]	Output
WriteData	[31:0]	Output

Table 3. Memory (mem)

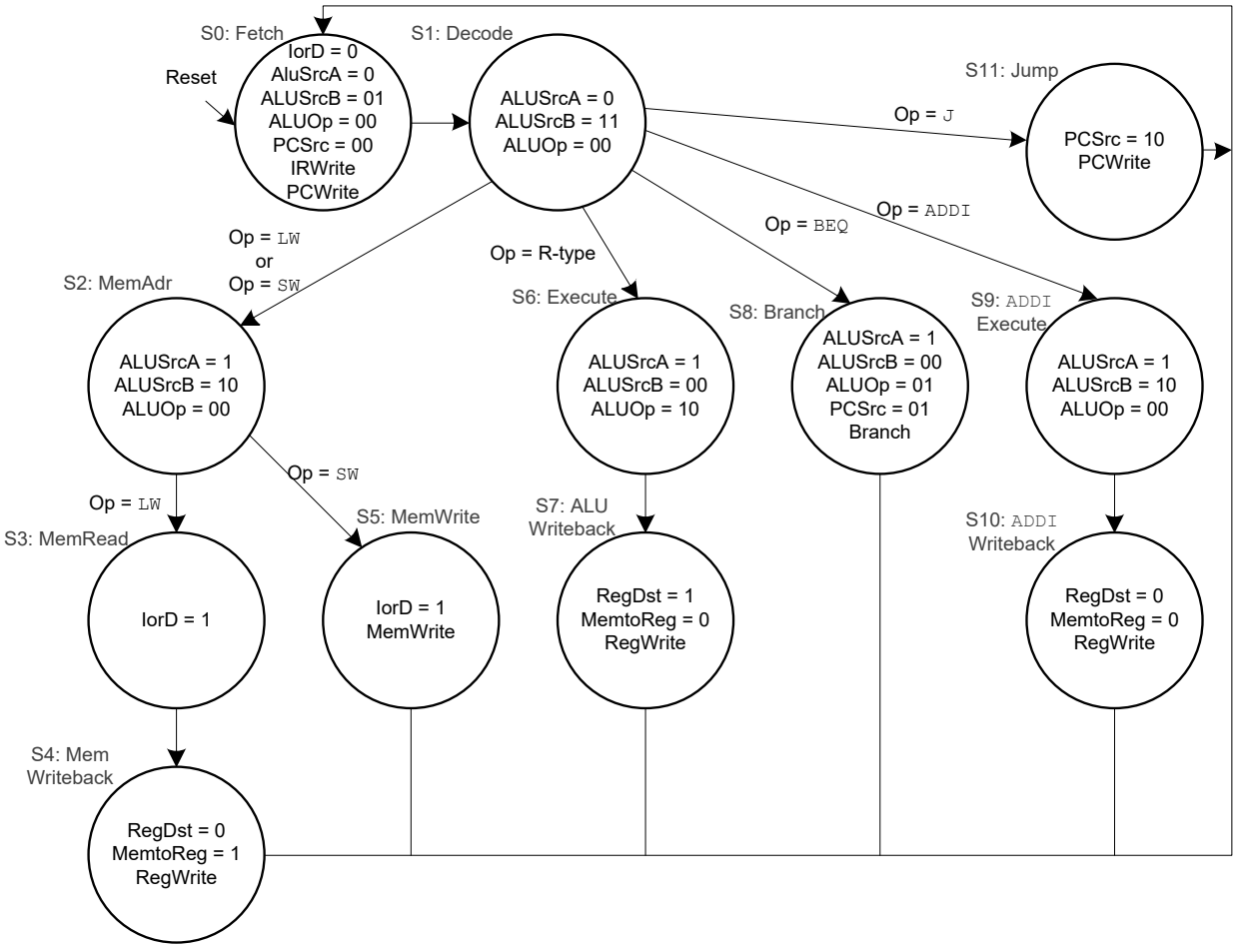
CLK		Input
Reset		Input
MemWrite		Input
Adr	[5:0]	Input
WriteData	[31:0]	Input
ReadData	[31:0]	Output

Control Unit Design

The control unit is the most complex part of the multicycle processor. It consists of two modules, the Main Decoder and the ALU Decoder. The Main Decoder, `maindec`, should take the Opcode input and produce the outputs described in Table 4. On reset, the control unit should start at State 0. The control unit should support the instructions listed in the introduction section. The state transition diagram is also given below.

Design your controller using an FSM for the Main Decoder and combinational logic for the ALU Decoder. Also include any additional logic needed to compute *PCEn* from the internal signals *PCWrite*, *Branch*, and *Zero*. The `controller`, `maindec`, and `aludec` headers are given showing the inputs and outputs for each module. A portion of the SystemVerilog code for the control unit has been given to you. Complete the SystemVerilog code to completely design the hardware of the controller and its submodules.

Design a `controllertest` testbench for the `controller` module. Test each of the instructions that the processor should support (`add`, `sub`, `and`, `or`, `slt`, `lw`, `sw`, `beq`, `addi`, and `j`). Be sure to test both taken and not-taken branches. Remember that the `controller` inputs are: `clk`, `Reset`, `OP`, `Funct`, and `Zero`. Your test bench should apply the inputs. Visually inspect the states and outputs to verify that they match your expectations from Table 4. Also verify that *PCEn* performs correctly. If you find any errors, debug your circuit and correct the errors. Save a copy of your waveforms showing the inputs, state, and control outputs, and *PCEn* at each state.



Demonstrate for Part A:

Prepare and show the following elements **in the given order**. Clearly label each part by number. Poorly organized submissions will lose points.

1. A completed Main Decoder output table (below).
2. Simulation waveforms of the controller module showing (in the given order): *CLK*, *Reset*, *OP*, *Funct*, *Zero*, the *state* (this is an internal registered signal), *ALUControl*, *PCEn*, and the entire control word (i.e. the 4-digit hex word you entered in the Table below) demonstrating each instruction (including taken and non-taken branches). Display all signals in hexadecimal. Does it match your expectations?
3. The SystemVerilog code file for your `controller`, `maindec`, and `aludec` modules.
4. Your `controllertest.sv` testbench.

State (Name)	PCWrite	MemWrite	IRWrite	RegWrite	ALUSrcA	Branch	IorD	MemtoReg	RegDst	ALUSrcB[1:0]	PCRsc[1:0]	ALUOp[1:0]	FSM Control Word
0 (Fetch)	1	0	1	0	0	0	0	0	0	01	00	00	0x5010
1 (Decode)	0	0	0	0	0	0	0	0	0	11	00	00	0x0030
2 (MemAdr)													
3 (MemRd)													
4 (MemWB)													
5 (MemWr)													
6 (RtypeEx)													
7 (RtypeWB)													
8 (BeqEx)													
9 (AddiEx)													
10 (AddiWB)													
11 (JEx)													

Table 4. Main Decoder Output

Part B: Datapath Design

Refer to Figure 1 for the hardware modules you need to set up your datapath. In this part, you will design the `datapath` and `mem` units and test your completed MIPS multicycle. Remember that you may reuse hardware from the previous labs (such as the ALU, multiplexers, registers, sign-extension hardware modules, register file, etc.) wherever possible.

All of your registers should take a *Reset* input to reset the initial value to a known state (0). The Instruction Register and PC also require enable inputs. Pay careful attention to bus connections; they are an easy place to make mistakes.

As in Lab 6, it is very helpful to first predict the results of a test program before running the program so that you know what to expect and can discover and track down discrepancies.

Table 5, which is partially completed, lists the expected instruction trace while running the test program. Complete the remainder of the table. Do this before you run simulations so you have a set of expectations to check your results against; otherwise, it is easy to fool yourself into believing that erroneous simulations are correct.

Simulate your processor using the same testbench from Lab 6 - below is Fig 7.60 in textbook as your test bench.

```
#          Assembly              Description              Address    Machine
main:      addi $2, $0, 5        # initialize $2 = 5       0          20020005
           addi $3, $0, 12       # initialize $3 = 12      4          2003000c
           addi $7, $3, -9       # initialize $7 = 3       8          2067fff7
           or   $4, $7, $2       # $4 = (3 OR 5) = 7       c          00e22025
           and  $5, $3, $4       # $5 = (12 AND 7) = 4     10         00642824
           add  $5, $5, $4       # $5 = 4 + 7 = 11         14         00a42820
           beq  $5, $7, end      # shouldn't be taken      18         10a7000a
           slt  $4, $3, $4       # $4 = 12 < 7 = 0         1c         0064202a
           beq  $4, $0, around   # should be taken         20         10800001
           addi $5, $0, 0        # shouldn't happen        24         20050000
around:    slt  $4, $7, $2       # $4 = 3 < 5 = 1          28         00e2202a
           add  $7, $4, $5       # $7 = 1 + 11 = 12        2c         00853820
           sub  $7, $7, $2       # $7 = 12 - 5 = 7         30         00e23822
           sw   $7, 68($3)       # [80] = 7                34         ac670044
           lw   $2, 80($0)       # $2 = [80] = 7           38         8c020050
           j    end              # should be taken         3c         08000011
           addi $2, $0, 1        # shouldn't happen        40         20020001
end:       sw   $2, 84($0)       # write mem[84] = 7       44         ac020054
```

Notice that the instruction (`instr`) is fetched during state 0 and therefore not updated until state 1 of each instruction. When the ALUResult will not be used (e.g. in the Decode state of a non-branch instruction, or the Writeback state of any instruction), you may indicate an 'x' for don't care rather than predicting the useless value that the processor will actually compute.

The Reset signal is set high at first. Display, at a minimum, the *PC*, *Instr*, FSM *state* (from within your `controller` module), *SrcA* and *SrcB* (from within your datapath), *ALUResult*, and *Zero*, and the control word. You will likely want to add other signals to help debug. Check that your results match the expectations from Table 5 in the next page. If there are any mismatches, debug your design and fix the errors.

Cycle	Reset	PC	Instr	(FSM) state	SrcA	SrcB	ALUResult	Zero	Control Word
1	1	00	0	0	00	04	04	0	5010
2	0	04	addi 20020005	1	04	x	x	0	0030
3	0	04	addi 20020005	9	00	05	05	0	0420
4	0	04	addi 20020005	10	x	x	x	0	0800
5	0	04	addi 20020005	0	04	04	08	0	5010
6	0	08	addi 2003000c	1	08	x	x	0	0030
7	0	08	addi 2003000c	9	00	0c	0c	0	0420
8	0	08	addi 2003000c	10	x	x	x	0	0800
9	0								
10	0								
11	0								
12	0								
13	0								
14	0	10	or 00e22025	1	10	x	x	0	0030
15	0	10	or 00e22025	6	03	05	07	0	0402
16	0	10	or 00e22025	7	x	x	x	0	0840
17	0	10	or 00e22025	0	10	04	14	0	5010
18	0	14	and 00642824	1	14	x	x	0	0030
19	0	14	and 00642824	6	0c	07	04	0	0402
20	0	14	and 00642824	7	x	x	x	0	0840
21	0	14	and 00642824	0	14	04	18	0	5010
22	0	18	add 00a42820	1	18	x	x	0	0030
23	0	18	add 00a42820	6	04	07	0b	0	0402
24	0	18	add 00a42820	7	x	x	x	0	0840
25	0	18	add 00a42820	0	18	04	1c	0	5010
26	0								
27	0								
28	0								
29	0								
30	0								
31	0								
32	0								
33	0								
34	0								
35	0								
36	0								
37	0								
38	0								
39	0								
40	0	30	add 00853820	1	30	x	x	0	0030
41	0	30	add 00853820	6	01	0b	0c	0	0402
42	0	30	add 00853820	7	x	x	x	0	0840
43	0	30	add 00853820	0	30	04	34	0	5010
44	0	34	sub 00e23822	1	34	x	x	0	0030
45	0	34	sub 00e23822	6	0c	05	07	0	0402
46	0	34	sub 00e23822	7	x	x	x	0	0840
47	0	34	sub 00e23822	0	34	04	38	0	5010
48	0	38	sw ac670044	1	38	x	x	0	0030
49	0	38	sw ac670044	2	0c	44	50	0	0420
50	0	38	sw ac670044	5	x	x	x	0	2100
51	0	38	sw ac670044	0	38	04	3c	0	5010
52	0	3c	lw 8c020050	1	3c	x	x	0	0030
53	0	3c	lw 8c020050	2	00	50	50	0	0420
54	0	3c	lw 8c020050	3	x	x	x	0	0100
55	0	3c	lw 8c020050	4	x	x	x	0	0880
56	0	3c	lw 8c020050	0	3c	04	40	0	5010
57	0								
58	0								
59	0								
60	0								
61	0								
62	0								

Table 5. Expected Instruction Trace

Debugging

Hopefully your lab will have at least one error so you will get to hone your debugging skills! Here are some hints:

- Be sure you thoroughly understand how the MIPS multi-cycle processor is supposed to work. This system is too complex to debug by trial and error. You should be able to predict what value every signal should be at every point in time while debugging.
- In general, trace problems by finding the first point in a simulation where a signal has an incorrect value. Don't worry about later problems because they could have been caused by the first error. Identify which circuit element is producing the bad output and add all its inputs to the simulation. Repeat until you have isolated the problem.

Part C: MIPS Plus

1. Modifying the MIPS multi-cycle processor

You now need to modify the MIPS multi-cycle processor by adding the instructions of {`ble`, `sltu`}. First, modify the MIPS processor datapath to show what changes are necessary. Next, modify the controller FSM, main decoder and ALU decoder as required. Finally, modify the SystemVerilog code as needed to include your modifications.

2. Testing your modified MIPS multi-cycle processor

Next, you'll need a test program to verify that your modified processor work. The program should check that your new instructions work properly and that the old ones didn't break. Use the example `test_9c.asm` shown here. At the end of running this program, you should see data written in at 7 locations (from M[0x50] to M[0x6C].

```
# test_9c.asm

      addi  $8, $0,  3
      addi $9, $0, 80
      addi $7, $0, -4
loop:
      sltu $10, $0, $8
      slt $11, $0, $8
      sub $10, $10, $7
      sub $11, $11, $7
      sw $10, 0($9)
      sw $11, 4($9)
      addi $8, $8, -3
      addi $9, $9, 8
      ble $7, $8, loop
      sw $8, 4($9)
```

Convert the program to machine language and put it in a file named `memfile2.dat`. Modify your memory to load this file. Modify the testbench to check for the appropriate address and data value indicating that the simulation succeeded. Run the program and check your results. Debug if necessary. When you are done, print out the waveforms as before and indicate the address and data value written by the `sw` instruction.

When you are finished – congratulations! You have built a microprocessor by yourself and have proven your mastery of microarchitecture, SystemVerilog, FSMs, and logic design!

Appendix: MIPS multi-cycle CPU SV Code Framework:

```
module mips(input  logic        clk, reset,
            output logic [31:0] adr, writedata,
            output logic        memwrite,
            input  logic [31:0] readdata);

  logic        zero, pcen, irwrite, regwrite,
               alusrca, iord, memtoreg, regdst;
  logic [1:0]  alusrcb, pcsrc;
  logic [2:0]  alucontrol;
  logic [5:0]  op, funct;

  controller c(clk, reset, op, funct, zero,
               pcen, memwrite, irwrite, regwrite,
               alusrca, iord, memtoreg, regdst,
               alusrcb, pcsrc, alucontrol);
  datapath dp(clk, reset,
              pcen, irwrite, regwrite,
              alusrca, iord, memtoreg, regdst,
              alusrcb, pcsrc, alucontrol,
              op, funct, zero,
              adr, writedata, readdata);
endmodule



module controller(input  logic       clk, reset,
                  input  logic [5:0] op, funct,
                  input  logic       zero,
                  output logic       pcen, memwrite, irwrite, regwrite,
                  output logic       alusrca, iord, memtoreg, regdst,
                  output logic [1:0] alusrcb, pcsrc,
                  output logic [2:0] alucontrol);

  logic [1:0] aluop;
  logic       branch, pcwrite;

  // Main Decoder and ALU Decoder subunits.
  maindec md(clk, reset, op,
             pcwrite, memwrite, irwrite, regwrite,
             alusrca, branch, iord, memtoreg, regdst,
             alusrcb, pcsrc, aluop);
  aludec  ad(funct, aluop, alucontrol);

  // ADD CODE HERE
  // Add combinational logic (i.e. an assign statement)
  // to produce the PCEn signal (pcen) from the branch,
  // zero, and pcwrite signals

endmodule
```

```
module maindec(input  logic       clk, reset,
               input  logic [5:0] op,
               output logic       pcwrite, memwrite, irwrite, regwrite,
               output logic       alusrca, branch, iord, memtoreg, regdst,
               output logic [1:0] alusrcb, pcsrc,
               output logic [1:0] aluop);

  parameter   FETCH   = 4'b0000;      // State 0
  parameter   DECODE  = 4'b0001;      // State 1
  parameter   MEMADR  = 4'b0010;      // State 2
  parameter   MEMRD   = 4'b0011;      // State 3
  parameter   MEMWB   = 4'b0100;      // State 4
  parameter   MEMWR   = 4'b0101;      // State 5
  parameter   RTYPEEX = 4'b0110;      // State 6
  parameter   RTYPEWB = 4'b0111;      // State 7
  parameter   BEQEX   = 4'b1000;      // State 8
  parameter   ADDIEX  = 4'b1001;      // State 9
  parameter   ADDIWB  = 4'b1010;      // state 10
  parameter   JEX     = 4'b1011;      // State 11

  parameter   LW      = 6'b100011;    // Opcode for lw
  parameter   SW      = 6'b101011;    // Opcode for sw
  parameter   RTYPE   = 6'b000000;    // Opcode for R-type
  parameter   BEQ     = 6'b000100;    // Opcode for beq
  parameter   ADDI    = 6'b001000;    // Opcode for addi
  parameter   J       = 6'b000010;    // Opcode for j

  logic [3:0]  state, nextstate;
  logic [14:0] controls;



// state register
  always_ff @(posedge clk or posedge reset)
    if(reset) state <= FETCH;
    else state <= nextstate;


  // ADD CODE HERE
  // Finish entering the next state logic below.  The first
  // two states, FETCH and DECODE, have been completed for you.



 // next state logic
  always_comb
    case(state)
      FETCH:   nextstate = DECODE;
      DECODE:  case(op)
                 LW:      nextstate = MEMADR;
                 SW:      nextstate = MEMADR;
                 RTYPE:   nextstate = RTYPEEX;
                 BEQ:     nextstate = BEQEX;
                 ADDI:    nextstate = ADDIEX;
                 J:       nextstate = JEX;
                 default: nextstate = 4'bx; // should never happen
               endcase
```

```
                // Add code here
      MEMADR:
      MEMRD:
      MEMWB:
      MEMWR:
      RTYPEEX:
      RTYPEWB:
      BEQEX:
      ADDIEX:
      ADDIWB:
      JEX:
      default: nextstate = 4'bx; // should never happen
    endcase

  // output logic
  assign {pcwrite, memwrite, irwrite, regwrite,
          alusrca, branch, iord, memtoreg, regdst,
          alusrcb, pcsrc, aluop} = controls;

  // ADD CODE HERE
  // Finish entering the output logic below.  The
  // output logic for the first two states, S0 and S1,
  // have been completed for you.

  always_comb
    case(state)
      FETCH:   controls = 15'h5010;
      DECODE:  controls = 15'h0030;

    // your code goes here


      default: controls = 15'hxxxx; // should never happen
    endcase
endmodule



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


  // ADD CODE HERE
  // Complete the design for the ALU Decoder.
  // Your design goes here.  Remember that this is a combinational
  // module.

  // Remember that you may also reuse any code from previous labs.

endmodule
```

```
// Complete the datapath module below.

// The datapath unit is a structural verilog module.  That is,
// it is composed of instances of its sub-modules.  For example,
// the instruction register is instantiated as a 32-bit flopenr.
// The other submodules are likewise instantiated.

module datapath(input  logic        clk, reset,
                input  logic        pcen, irwrite, regwrite,
                input  logic        alusrca, iord, memtoreg, regdst,
                input  logic [1:0]  alusrcb, pcsrc,
                input  logic [2:0]  alucontrol,
                output logic [5:0]  op, funct,
                output logic        zero,
                output logic [31:0] adr, writedata,
                input  logic [31:0] readdata);

  // Below are the internal signals of the datapath module.

  logic [4:0]  writereg;
  logic [31:0] pcnext, pc;
  logic [31:0] instr, data, srca, srcb;
  logic [31:0] a;
  logic [31:0] aluresult, aluout;
  logic [31:0] signimm;   // the sign-extended immediate
  logic [31:0] signimmsh; // the sign-extended immediate shifted left by 2
  logic [31:0] wd3, rd1, rd2;

  // op and funct fields to controller
  assign op = instr[31:26];
  assign funct = instr[5:0];

  // Your datapath hardware goes below.  Instantiate each of the submodules
  // that you need.  Remember that alu's, mux's and various other
  // versions of parameterizable modules are available in textbook 7.6

  // Here, parameterizable 3:1 and 4:1 muxes are provided below for your use.

  // Remember to give your instantiated modules applicable names
  // such as pcreg (PC register), wdmux (Write Data Mux), etc.
  // so it's easier to understand.

  // ADD CODE HERE

  // datapath

endmodule


module mux3 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2,
              input  logic [1:0]       s,
              output logic [WIDTH-1:0] y);

  assign #1 y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule
```

```
module mux4 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2, d3,
              input  logic [1:0]       s,
              output logic [WIDTH-1:0] y);

   always_comb
      case(s)
         2'b00: y = d0;
         2'b01: y = d1;
         2'b10: y = d2;
         2'b11: y = d3;
      endcase
endmodule


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

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

  // initialize memory with instructions
  initial
    begin
      $readmemh("memfile.dat",RAM);
        // "memfile.dat" contains your instructions in hex
        // you must create this file
    end

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we)
      RAM[a[31:2]] <= wd;
Endmodule


module top(input  logic        clk, reset,
           output logic [31:0] writedata, adr,
           output logic        memwrite);

  logic [31:0] readdata;

  // microprocessor (control & datapath)
  mips mips(clk, reset, adr, writedata, memwrite, readdata);

  // memory
  mem mem(clk, memwrite, adr, writedata, readdata);

endmodule
```