E85: Digital Electronics and Computer Engineering
Lab 11: Multicycle Processor

Objective

In this lab, you will build, simulate, and debug a multicycle RISC-V processor in SystemVerilog.

1. Multicycle RISC-V Processor

Figure 1 (near the end of this lab) shows the complete multicycle processor. Figure 2 shows the high-level hierarchy of the single-cycle processor including the connections between the controller, datapath, instruction memory, and data memory. Your multi-cycle processor has only a single unified memory and has slightly different control signals, so you will need to modify these connections. Sketch a diagram similar to Figure 2 showing your controller, datapath, and memory modules. Draw a box for the `riscv` module that should encompass the controller and datapath. Label the signals passing between blocks.

Write a hierarchical Verilog description of the processor. The processor should have the following module declaration. The memory signals are tapped out for testing purposes. Use your controller from Lab 10 and any general Verilog building blocks you need (e.g., muxes, flops, adders, ALU, register file, immediate extender, etc.) from the single-cycle processor. The single-cycle processor code (`RISCVsingle.sv`) is on the class web site and you may wish to cut and paste blocks from it.

```
module top(input  logic        clk, reset,
           output logic [31:0] WriteData, DataAdr,
           output logic        MemWrite);
```

2. Test Bench

The riscv_testbench.sv and test code (in assembly .asm and machine language .dat) are on the class web page. Study the test bench to understand how it determines if your tests succeeded or failed.

Your memory should read the test code from the memory file at startup with the line:

```
initial $readmemh("memfile.dat", RAM);
```

Before you begin simulation, predict what the processor should do while executing the first three instructions. Table 1 has been filled out for you for the first instruction.

Generate simulation waveforms at least for clk, reset, PC, Instr, state, SrcA, SrcB, ALUResult, Adr, WriteData, and MemWrite. Display the 32-bit signals in hexadecimal for ease of reading (select the signals and right click, then choose Radix). Compare against your expectations. You may wish to add other signals to help debug. Fix any problems you may find until your code executes the program as expected and the testbench reports success.

Refer to the previous lab for debugging hints. Fix all relevant warnings from Quartus and Questa before you debug further. It will save you much time to carefully predict what each of the signals in your waveforms should be doing on each cycle, and to systematically debug beginning with the first known discrepancy and working your way backward until you have good inputs and bad outputs and have isolated the bug.

Common bugs include:

- Copying the single-cycle processor top, riscv, or datapath interface with signals that don't match the multi-cycle processor. Be sure you have a clear idea what belongs in each module. You will likely save time if you sketch a picture similar to Figure 2 and identify what signals flow between the riscv and memory modules.
- Connecting signals in different orders in a module declaration vs. in the instantiation.
- Forgetting to declare internal signals, or giving them the wrong widths.
- Inconsistent capitalization or spelling.

If you've checked these and your processor still isn't working, try adding all the outputs of the controller to your sim and make sure none are floating or X. If you still haven't found the problem, refer to your predicted waveforms in Table 1 and check that the processor is doing the right thing on each step. If the first few instructions are correct, you may need to extend the table to predict what the rest of the program should be doing. (Once you've filled out the table for several instructions, you may get the hang of the pattern and only fill out entries that are interesting…)

What to Turn In

1. Please indicate how many hours you spent on this lab. This will be helpful for calibrating the workload for next time the course is taught. → 7hours
2. Diagram showing your memory, riscv, datapath, and controller block hierarchy and names of all signals between them.
3. Hierarchical SystemVerilog for your top-level processor module (and submodules) matching the declaration given above.
4. Table 1 showing key signals for at least the first three instructions.

5. Simulation waveforms (in the order listed above) at least for the specified signals. Does your system pass your testbench? Circle or highlight the waves showing that the correct value is written to the correct address, and make sure it is legible.

Please indicate any bugs you found in this lab manual, or any suggestions you would have to improve the lab.

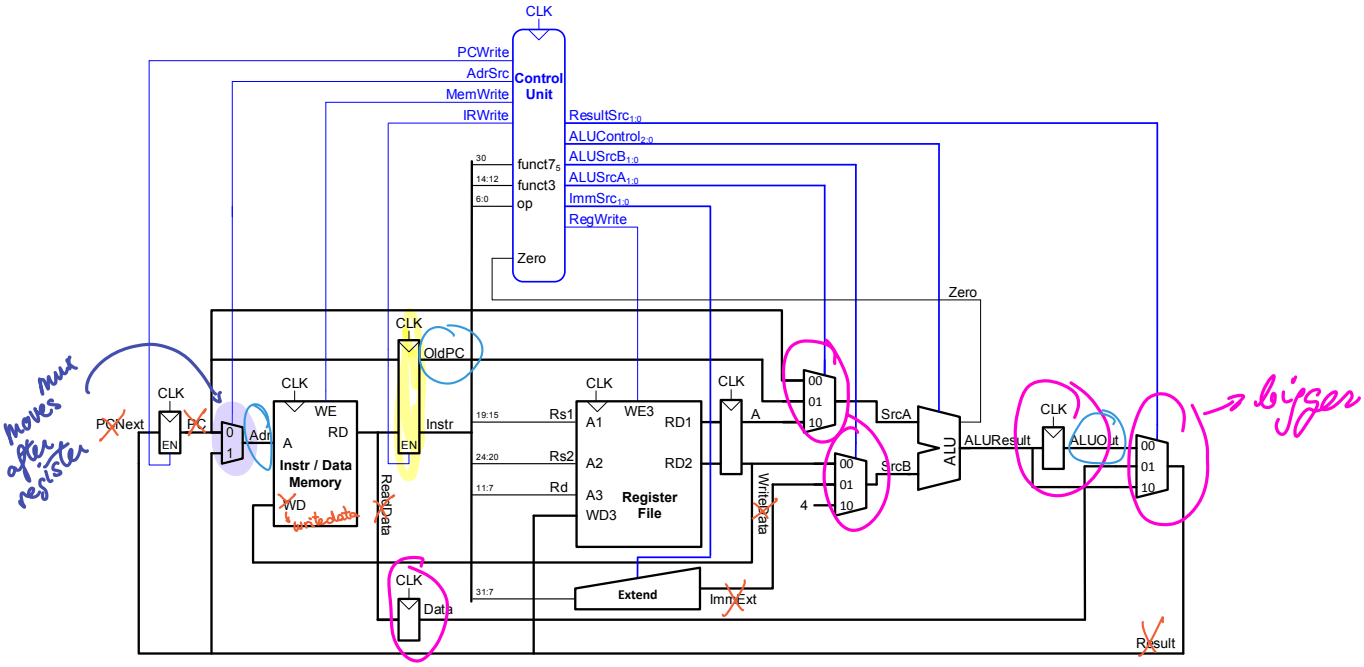
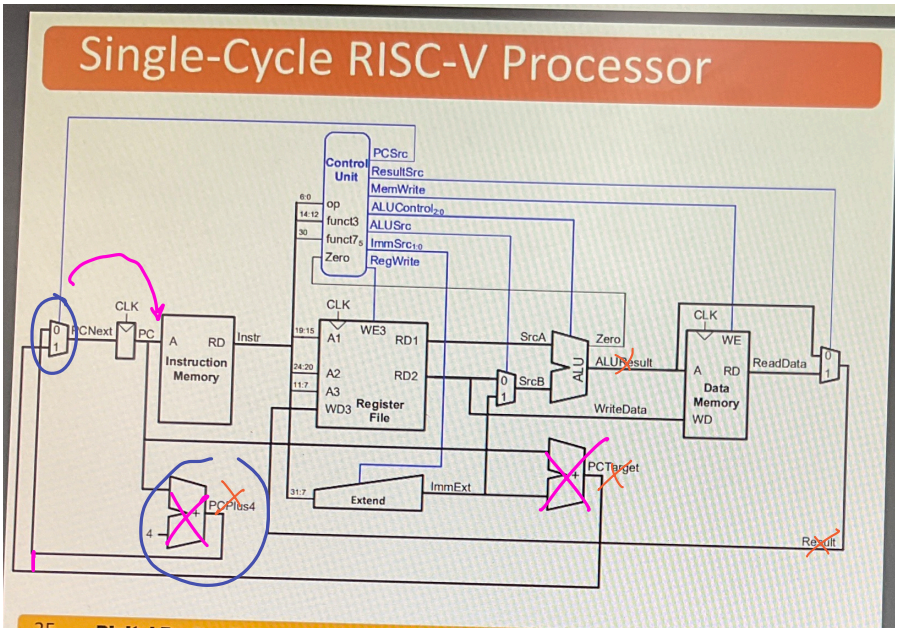


Figure 1: Complete multicycle processor



2) Diagram → The second one where the ✡ is

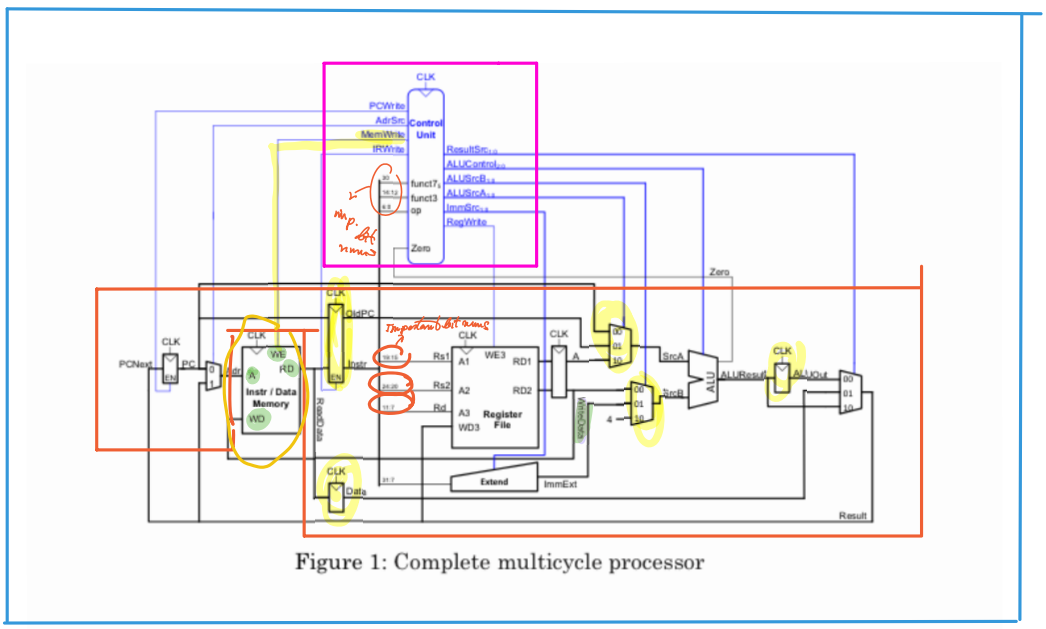
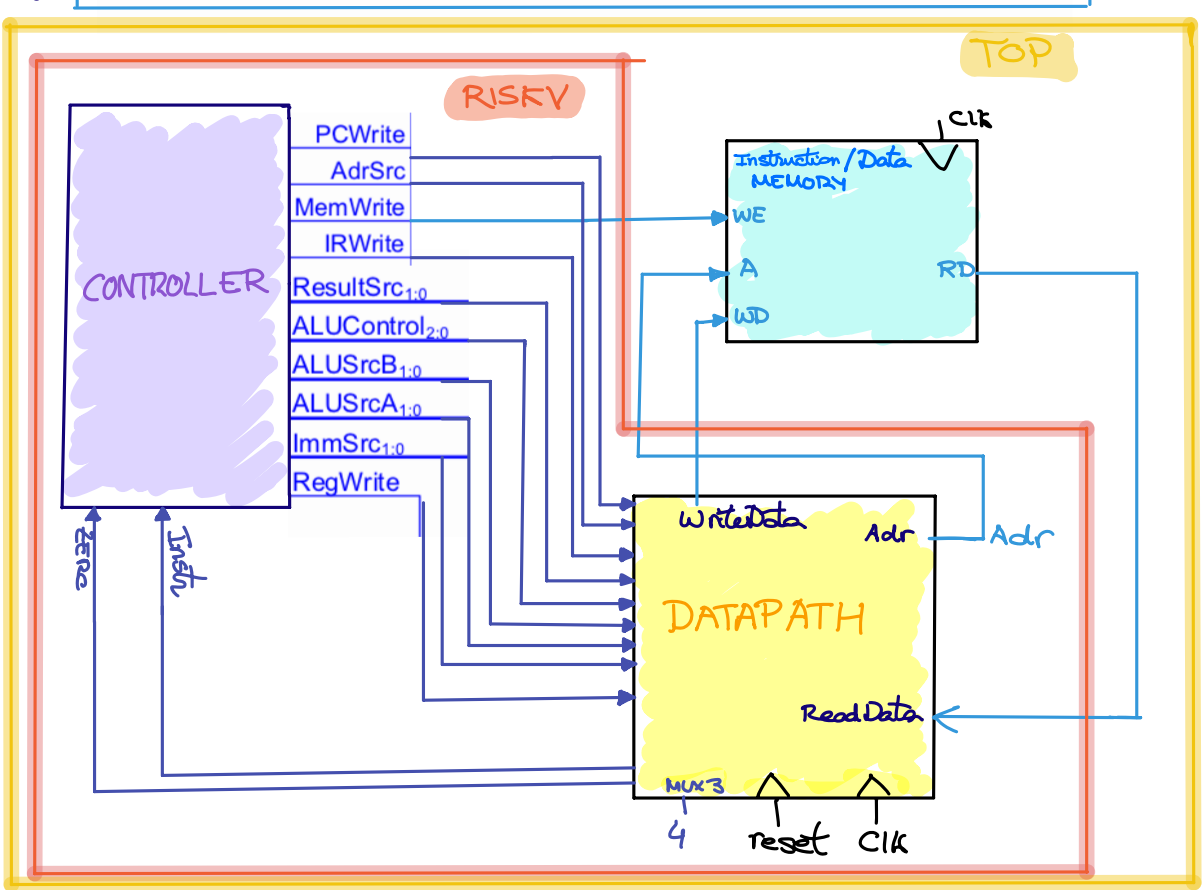


Figure 1: Complete multicycle processor

✡2



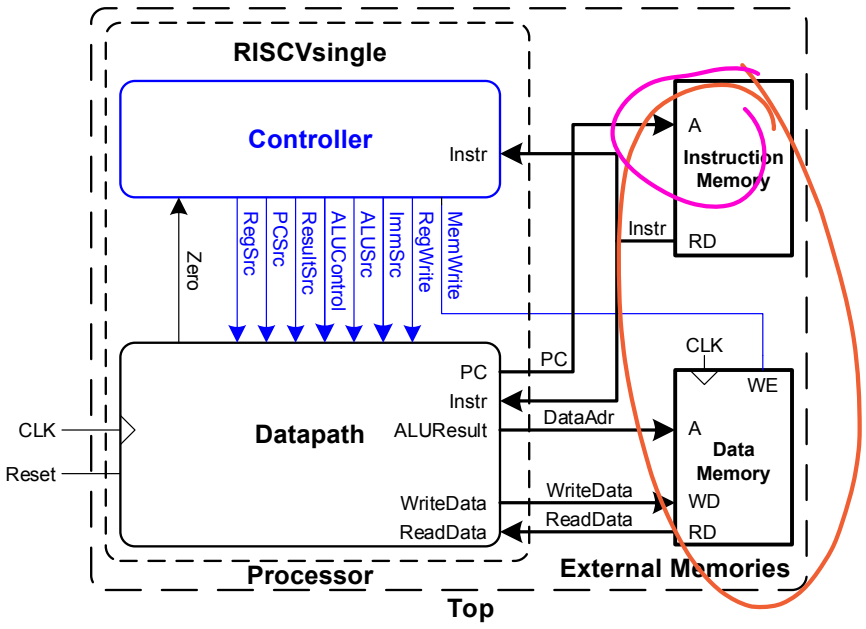
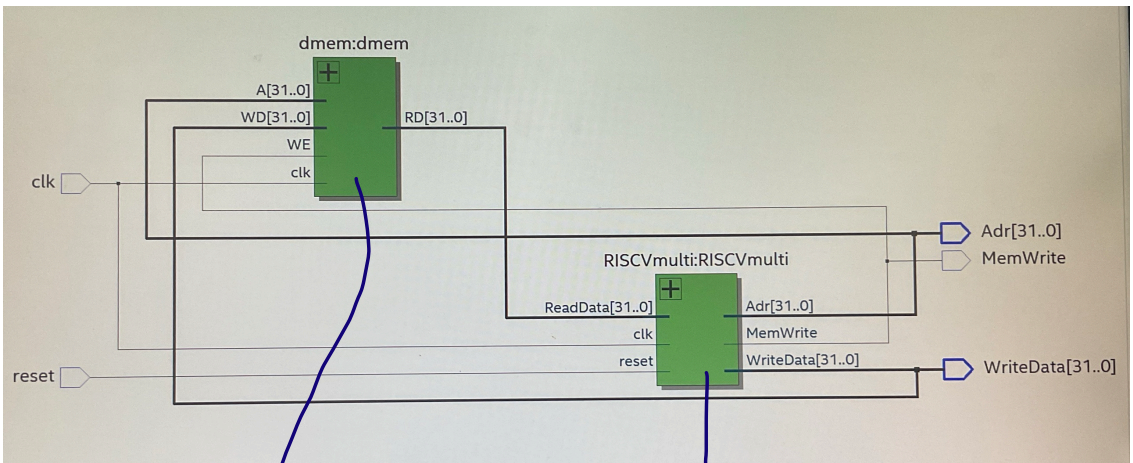
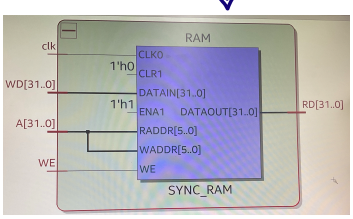
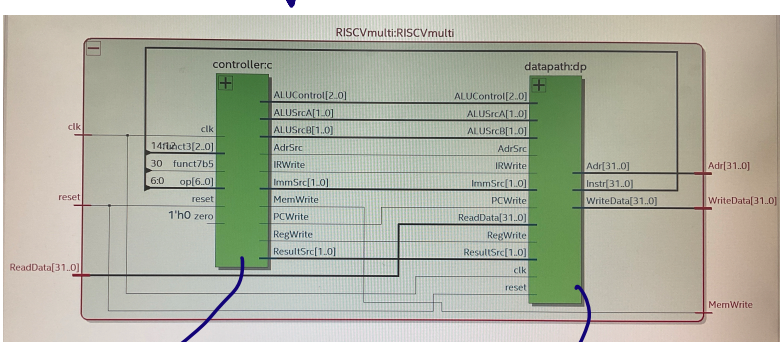


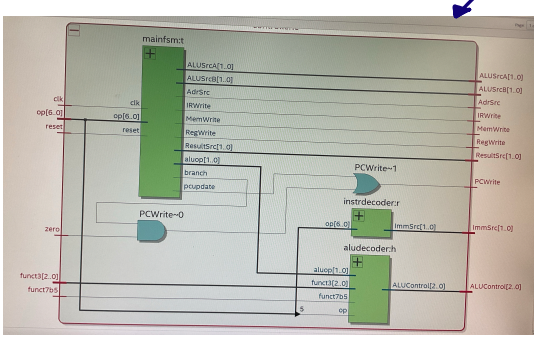
Figure 2: Single-cycle processor interfaced to external memories

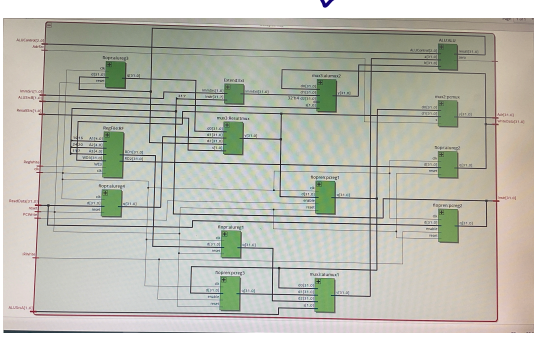
3. Hierarchical SystemVerilog for your top-level processor module (and submodules) matching the declaration given above.











```
//START OF MULTI-CYCLE PROCESSOR!!
//everything correct here!!

module top(input  logic        clk, reset,
           output logic [31:0] WriteData, Adr,
           output logic        MemWrite);

  logic [31:0] ReadData;

  // instantiate processor and memories
  RISCVmulti RISCVmulti(clk, reset, ReadData, Adr, MemWrite, WriteData);
  dmem dmem(clk, MemWrite, Adr, WriteData, ReadData);
endmodule



//MEMORY - CORRECT!!
module dmem(input  logic        clk, WE,
            input  logic [31:0] A, WD,
            output logic [31:0] RD);

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

  initial
      $readmemh("memfile.dat",RAM);

  assign RD = RAM[A[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (WE) RAM[A[31:2]] <= WD;

endmodule


//RISKV incudes datapath and contorller
module RISCVmulti( input  logic         clk, reset,
                   input  logic [31:0]  ReadData,
                   output logic [31:0]  Adr,
                   output logic         MemWrite,
                   output logic [31:0]  WriteData);

  logic        ALUSrc, RegWrite, Zero, AdrSrc, IRWrite, PCWrite;
  logic [1:0]  ResultSrc, ImmSrc, ALUSrcA, ALUSrcB;
  logic [31:0] Instr;
  logic [2:0]  ALUControl;


  controller c(clk, reset,  Instr[6:0], Instr[14:12], Instr[30], zero, ImmSrc, ALUSrcA,
               ALUSrcB, ResultSrc, AdrSrc, ALUControl, IRWrite, PCWrite,
               RegWrite, MemWrite);

  datapath dp(clk, reset, ResultSrc, PCWrite, AdrSrc, ALUSrcA, ALUSrcB, RegWrite,  IRWrite,
              ImmSrc, ALUControl, ReadData, Zero, Instr, WriteData, Adr);
endmodule

module datapath(input  logic        clk, reset,
                input  logic [1:0]  ResultSrc,
                input  logic        PCWrite, AdrSrc,
                input  logic [1:0]  ALUSrcA, ALUSrcB,
                input  logic        RegWrite, IRWrite,
                input  logic [1:0]  ImmSrc,
                input  logic [2:0]  ALUControl,
                input  logic [31:0] ReadData,
                output logic        Zero,
                output logic [31:0] Instr,
                output logic [31:0] WriteData, Adr);

  logic [31:0]  PCNext, PC, OldPC;
  logic [31:0]  ImmExt;
  logic [31:0]  Data;
  logic [31:0]  A, SrcA, SrcB;
  logic [31:0]  Result;
  logic [31:0]  ALUResult, ALUOut;
  logic [31:0]  RD1, RD2;


  // next PC logic
  assign PCNext = Result;
  flopren #(32)   pcreg1(clk, reset, PCWrite, PCNext, PC);
  mux2 #(32)      pcmux(PC, Result, AdrSrc, Adr);

  //data logic
  flopr #(32)     alureg4(clk, reset, ReadData, Data);

  //After the memory
  flopren #(32)   pcreg2(clk, reset, IRWrite, ReadData, Instr);
  flopren #(32)   pcreg3(clk, reset, IRWrite, PC, OldPC);

  // register file logic

  Extend         Ext(Instr[31:7], ImmSrc, ImmExt);
  flopr #(32)    alureg1(clk, reset, RD1, A);
  flopr #(32)    alureg2(clk, reset, RD2, WriteData);
```

```
    // ALU logic
    mux3  #(32)    alumux1(PC, OldPC, A, ALUSrcA, SrcA);
    mux3  #(32)    alumux2(WriteData, ImmExt,  32'd4, ALUSrcB, SrcB);
    ALU            ALU(SrcA, SrcB, ALUControl, ALUResult, Zero);
    flopr #(32)    alureg3(clk, reset, ALUResult, ALUOut);
    mux3  #(32)    Resultmux(ALUOut, Data, ALUResult, ResultSrc, Result);

  //moved down bc mau moved it down
    RegFile      RF(clk, RegWrite, Instr[19:15], Instr[24:20],
                    Instr[11:7], Result, RD1, RD2);

  endmodule




module Extend(input  logic [31:7] Instr,
              input  logic [1:0]  ImmSrc,
              output logic [31:0] ImmExt);

    always_comb
      case(ImmSrc)
                    // I-type
        2'b00:   ImmExt = {{20{Instr[31]}}, Instr[31:20]};
                    // S-type (Stores)
        2'b01:   ImmExt = {{20{Instr[31]}}, Instr[31:25], Instr[11:7]};
                    // B-type (Branches)
        2'b10:   ImmExt = {{20{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0};
                    // J-type (Jumps)
        2'b11:   ImmExt = {{12{Instr[31]}}, Instr[19:12], Instr[20], Instr[30:21], 1'b0};
        default: ImmExt = 32'bx; // undefined
      endcase
  endmodule


module RegFile(input  logic        clk,    //THE SAME NO NEED TO CHANGE
               input  logic        WE3,
               input  logic [ 4:0] A1, A2, A3,
               input  logic [31:0] WD3,
               output logic [31:0] RD1, RD2);

   logic [31:0] rf[31:0];

   // three ported register file
   // read two ports combinationally (A1/RD1, A2/RD2)
   // write third port on rising edge of clock (A3/WD3/WE3)
   // register 0 hardwired to 0
   always_ff @(posedge clk)
     if (WE3) rf[A3] <= WD3;

   assign RD1 = (A1 != 0) ? rf[A1] : 0;
   assign RD2 = (A2 != 0) ? rf[A2] : 0;
 endmodule



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

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


  //added flopr enable
  module flopren #(parameter WIDTH = 8)
               (input  logic             clk, reset, enable,
                input  logic [WIDTH-1:0] d,
                output logic [WIDTH-1:0] q);
    always_ff @(posedge clk, posedge reset)
      if (reset)       q <= 0;
      else if(enable)  q <= d;
      else             q <= q;   //!!!! no lo tiene
  endmodule

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

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




module  ALU(input  logic [31:0] a, b,
            input  logic [2:0]  ALUControl,
            output logic [31:0] result,
            output logic        zero);

   logic [31:0] condinvb, sum;
   logic        sub;

   assign sub = (ALUControl[1:0] == 2'b01);
   assign condinvb = sub ? ~b : b; // for subtraction or slt
   assign sum = a + condinvb + sub;


   always_comb
     case (ALUControl)
       3'b000: result = sum;          // addition
       3'b001: result = sum;          // subtraction
       3'b010: result = a & b;        // and
       3'b011: result = a | b;        // or
       3'b101: result = sum[31];      // slt
       default: result = 0;
   endcase

   assign zero = (result == 32'b0);
endmodule
```

```
//CONTROLLER

module controller(input logic clk, reset,
   // input  logic reset,
    input  logic [6:0] op,
    input  logic [2:0] funct3,
    input  logic funct7b5,
    input  logic zero,
    output logic [1:0] ImmSrc,
    output logic [1:0] ALUSrcA, ALUSrcB,
    output logic [1:0] ResultSrc,
    output logic AdrSrc,
    output logic [2:0] ALUControl,
    output logic IRWrite, PCWrite,
    output logic RegWrite, MemWrite);


    logic  [1:0] aluop;
    logic  pcupdate;
    logic  branch;
    assign PCWrite = (zero & branch)||pcupdate;

    aludecoder h(op[5], aluop, funct3, funct7b5, ALUControl);
    instrdecoder r(op,ImmSrc);
    mainfsm t(clk, reset, op, ALUSrcA, ALUSrcB, ResultSrc, AdrSrc, IRWrite,
    RegWrite, MemWrite, pcupdate, branch, aluop);

endmodule
```

```
module aludecoder(
      input  logic op,
      input  logic [1:0] aluop,
      input  logic [2:0] funct3,
      input  logic funct7b5,
      output logic [2:0] ALUControl);


        //next state logic
      always_comb // ALU DECODER LOGIC
         casez({aluop, funct3, op, funct7b5}) //op choose the 5th bit
            7'b0077777: ALUControl = 3'b000;
            7'b017????: ALUControl = 3'b001;
            7'b1000000: ALUControl = 3'b000;
            7'b1000001: ALUControl = 3'b000;
            7'b1000010: ALUControl = 3'b000;
            7'b1000011: ALUControl = 3'b001;
            7'b1001077: ALUControl = 3'b101;
            7'b1011077: ALUControl = 3'b011;
            7'b1011177: ALUControl = 3'b010;
            default:    ALUControl = 3'b000;
      endcase

endmodule



module instrdecoder(
      input  logic [6:0] op,
      output logic [1:0] ImmSrc);

      always_comb // instr DECODER LOGIC
        case(op)//op choose the 5th bit
            7'b0110011: ImmSrc = 2'b00;
            7'b0010011: ImmSrc = 2'b00;
            7'b0000011: ImmSrc = 2'b00;
            7'b0100011: ImmSrc = 2'b01;
            7'b1100011: ImmSrc = 2'b10;
            7'b1101111: ImmSrc = 2'b11;
            default:    ImmSrc = 2'b00;
      endcase

endmodule
```

```
module mainfsm(input logic clk,
    input  logic reset,
    input  logic [6:0] op,
    output logic [1:0] ALUSrcA, ALUSrcB,
    output logic [1:0] ResultSrc,
    output logic AdrSrc,
    output logic IRWrite,
    output logic RegWrite, MemWrite,
    output logic pcupdate,
    output logic branch,
    output logic [1:0] aluop);


typedef enum logic [4:0] {S0,S1, S2, S3, S4, S5, S6, S7,S8, S9, S10}
  statetype;
  statetype state, nextstate;

    //state register
    always_ff @(posedge clk, posedge reset)
      if(reset) state <= S0;
      else state <= nextstate;

    //next state logic
    always_comb
      case(state)
        S0: nextstate = S1;
        S1: if (op==7'b0000011 || op== 7'b0100011) nextstate = S2; //MemAdr
            else if (op==7'b0110011) nextstate = S6; //ExecuteR
            else if (op==7'b0010011) nextstate = S8; //ExecuteI
            else if (op==7'b1101111) nextstate = S9; //JAL
            else if (op==7'b1100011) nextstate = S10; //BEQ
            else nextstate = S0;

        S2: if    (op==7'b0000011) nextstate = S3; //MemRead
            else if (op==7'b0100011) nextstate = S5; //MemWrite
            else nextstate = S0;

        S3: nextstate = S4; //MemWB
        S4: nextstate = S0; //Fetch
        S5: nextstate = S0; //Fetch
        S6: nextstate = S7; //ALUWB
        S7: nextstate = S0; //Fetch
        S8: nextstate = S7; //ALUWB
        S9: nextstate = S7; //ALUWB
        S10:nextstate = S0;//Fetch
      endcase
```

```
    //next state logic
    always_comb begin
       case(state)

          S0: begin //Fetch
             AdrSrc   = 0;
             IRWrite  = 1;
             ALUSrcA  = 2'b00;
             ALUSrcB  = 2'b10;
             aluop    = 2'b00;
             ResultSrc= 2'b10;
             pcupdate = 1;
             RegWrite = 0;
             MemWrite = 0;
             branch   = 0;
          end


          S1: begin //Decode
             AdrSrc   = 0;
             IRWrite  = 0;
             ALUSrcA  = 2'b01;
             ALUSrcB  = 2'b01;
             aluop    = 2'b00;
             ResultSrc= 2'b00;
             pcupdate = 0;
             RegWrite = 0;
             MemWrite = 0;
             branch   = 0;
          end

          S2: begin //MemAdr
             AdrSrc   = 0;
             IRWrite  = 0;
             ALUSrcA  = 2'b10;
             ALUSrcB  = 2'b01;
             aluop    = 2'b00;
             ResultSrc= 2'b00;
             pcupdate = 0;
             RegWrite = 0;
             MemWrite = 0;
             branch   = 0;
          end

          S3: begin //MemRead
             AdrSrc   = 1;
             IRWrite  = 0;
             ALUSrcA  = 2'b00;
             ALUSrcB  = 2'b00;
             aluop    = 2'b00;
             ResultSrc= 2'b00;
             pcupdate = 0;
             RegWrite = 0;
             MemWrite = 0;
             branch   = 0;
          end
```

```
                S4: begin //MemWB
                    AdrSrc   = 0;
                    IRWrite  = 0;
                    ALUSrcA  = 2'b00;
                    ALUSrcB  = 2'b00;
                    aluop    = 2'b00;
                    ResultSrc= 2'b01;
                    pcupdate = 0;
                    RegWrite = 1;
                    MemWrite = 0;
                    branch   = 0;
                end

                S5: begin //MemWrite
                    AdrSrc   = 1;
                    IRWrite  = 0;
                    ALUSrcA  = 2'b00;
                    ALUSrcB  = 2'b00;
                    aluop    = 2'b00;
                    ResultSrc= 2'b00;
                    pcupdate = 0;
                    RegWrite = 0;
                    MemWrite = 1;
                    branch   = 0;
                end

                S6: begin //ExecuteR
                    AdrSrc   = 0;
                    IRWrite  = 0;
                    ALUSrcA  = 2'b10;
                    ALUSrcB  = 2'b00;
                    aluop    = 2'b10;
                    ResultSrc= 2'b00;
                    pcupdate = 0;
                    RegWrite = 0;
                    MemWrite = 0;
                    branch   = 0;
                end

                S7: begin //ALUWB
                    AdrSrc   = 0;
                    IRWrite  = 0;
                    ALUSrcA  = 2'b00;
                    ALUSrcB  = 2'b00;
                    aluop    = 2'b00;
                    ResultSrc= 2'b00;
                    pcupdate = 0;
                    RegWrite = 1;
                    MemWrite = 0;
                    branch   = 0;
                end

                S8: begin //ExecuteL
                    AdrSrc   = 0;
                    IRWrite  = 0;
                    ALUSrcA  = 2'b10;
                    ALUSrcB  = 2'b01;
                    aluop    = 2'b10;
                    ResultSrc= 2'b00;
                    pcupdate = 0;
                    RegWrite = 0;
                    MemWrite = 0;
                    branch   = 0;
                end
```

```
                S9: begin //jal
                    AdrSrc   = 0;
                    IRWrite  = 0;
                    ALUSrcA  = 2'b01;
                    ALUSrcB  = 2'b10;
                    aluop    = 2'b00;
                    ResultSrc= 2'b00;
                    pcupdate = 1;
                    RegWrite = 0;
                    MemWrite = 0;
                    branch   = 0;
                end

                S10: begin //BEQ
                    AdrSrc   = 0;
                    IRWrite  = 0;
                    ALUSrcA  = 2'b10;
                    ALUSrcB  = 2'b00;
                    aluop    = 2'b01;
                    ResultSrc= 2'b00;
                    pcupdate = 0;
                    RegWrite = 0;
                    MemWrite = 0;
                    branch   = 1;
                end

                default: begin
                    AdrSrc   = 0;
                    IRWrite  = 0;
                    ALUSrcA  = 2'b00;
                    ALUSrcB  = 2'b00;
                    aluop    = 2'b00;
                    ResultSrc= 2'b00;
                    pcupdate = 0;
                    RegWrite = 0;
                    MemWrite = 0;
                    branch   = 0;
                end

            endcase
        end

endmodule
```

4)

4. Table 1 showing key signals for at least the first three instructions.

Hexadecimal → 10 ⇒ 16
c ⇒ 12
14 ⇒ 20
↳ 4x1 + 16x1 = 20
e → 14

Table 1: Expected Operation (after two cycles of reset)

Step	PC	Instr	State	Result	Result Notes
3	00	00500113	S0: Fetch	4	PC+4
4	04	""	S1: Decode	X	OldPC+Immediate
5	04	""	S8: ExecuteI	X	ALUResult = x0 (0) + 5 = 5
6	04	""	S7: ALUWB	5	Result = ALUOUT
7	04	""	S0: Fetch	8	PC+4
8	08	00c00193	S1: Decode	X	OldPC+Immediate
9	08	" "	S8 : ExecuteI	X	ALUResult = x0 (0) +12 = 12
10	08	" "	S7 : ALUWB	12 = C	Result = ALUOUT
11	08	" "	S0 : Fetch	12 = C	PC+4
12	C	ff718393	S1: Decode	X	OldPC+Immediate
13	C	" "	S8 : ExecuteI	X	ALUResult = x3 (12) -9 = 3
14	C	" "	S7 : ALUWB	3	Result=ALUOUT
15	C	" "	S0 : Fetch	16 = 10	PC+4
16	10	0023e233	S1 : Decode	X	OldPC + Imm
17	10	" "	S6 : Execute R	X	ALUResult =
18	10	" "	S7 : ALUWB	7	Result = ALUOUT
19	10	" "	S0 : Fetch	20 = 14	PC+4
20	14	00412283	S1 : Decode	X	OldPc + Imm
21	14	" "	S6 : Execute R	X	ALUResult =
22	14	" "	S7 : ALUWB	4	Result = ALUOUT
23	14	" "	S0 : Fetch	24	PC+4
24	24	00428263	S1	X	
25			S6		
26			S7	11	
27			S0		
28					
29					
30					
31					
32					
33					
34					
35					
36					
37					
38					
39					
40					
41					
42					
43					

addi x2, x0, 5 → what value inside reg. x0

addi x3, x0, 12

addi x7, x3, -9

or x4, x7, x2 → R-type
x7 → 011
x2 → 101
} Or
001111 → 7

12 ← 16 ← 14 ← 10 11 0011

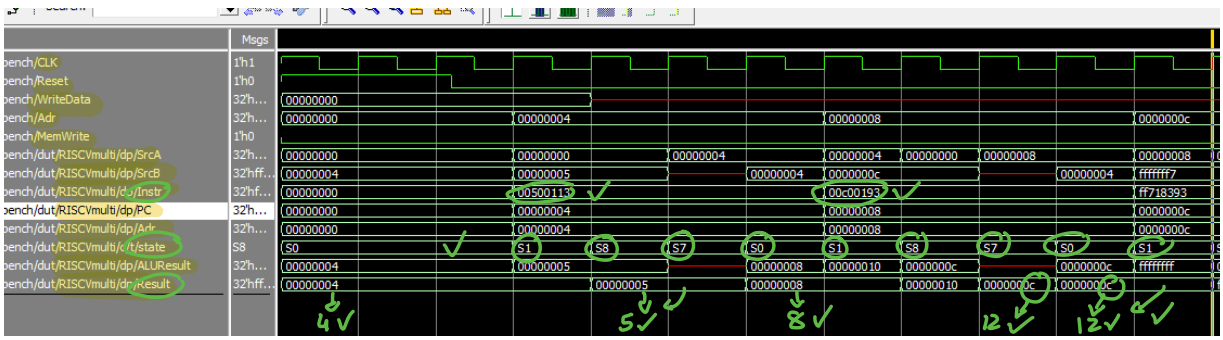
and x5, x3, x4 ↳ = 4

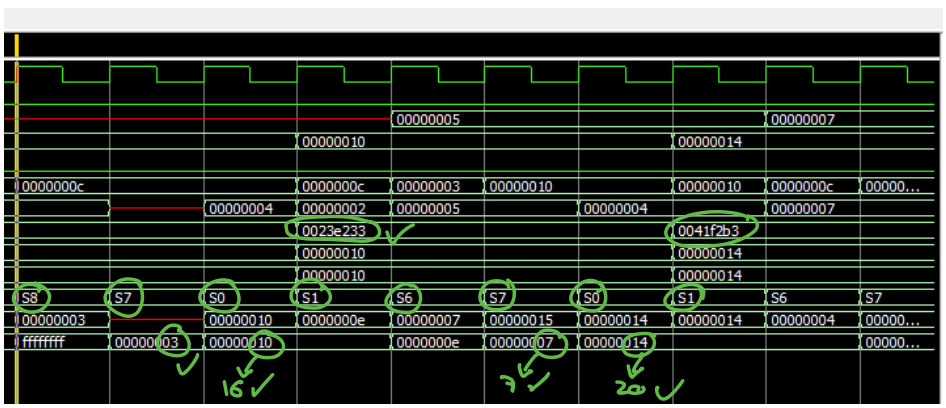
and x5, x5, x4 ↳ = 11

(Only fill in the notes column if it is helpful to you to understand what is happening and why.)

5. Simulation waveforms (in the order listed above) at least for the specified signals. Does your system pass your testbench? Circle or highlight the waves showing that the correct value is written to the correct address, and make sure it is legible.

My system does pass the testbench





Showed above that the 1st 4 instructions work ✓