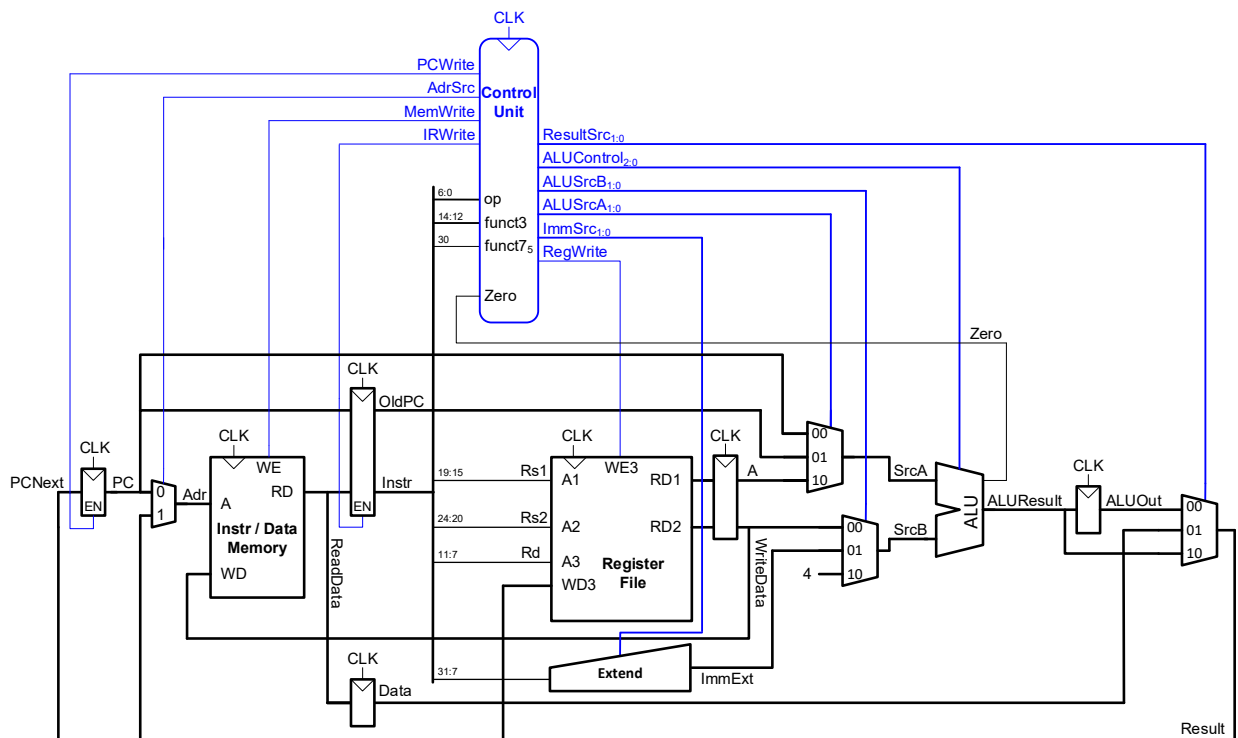


## Lab 15: Multicycle Processor SOLUTIONS

*Digital Design and Computer Architecture: RISC-V Edition (Harris & Harris, Elsevier © 2021)*

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.
2. Diagram showing your memory, riscv, datapath, and controller block hierarchy and names of all signals between them.

**Answer:**



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

**Answer:**

```

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

    logic [31:0] ReadData;

    // instantiate processor and memories
    riscvmulti rvmulti(clk, reset, MemWrite, DataAdr,
                       WriteData, ReadData);
    mem mem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule

```

```

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

    logic      RegWrite, jump;
    logic [1:0] ResultSrc, ImmSrc;
    logic [2:0] ALUControl;
    logic      PCWrite;
    logic      IRWrite;
    logic [1:0] ALUSrcA;
    logic [1:0] ALUSrcB;
    logic      AdrSrc;
    logic      Zero;
    logic [6:0] op;
    logic [2:0] funct3;
    logic      funct7b5;

    controller c(clk, reset, op, funct3, funct7b5, Zero,
                 ImmSrc, ALUSrcA, ALUSrcB,
                 ResultSrc, AdrSrc, ALUControl,
                 IRWrite, PCWrite, RegWrite, MemWrite);

    datapath dp(clk, reset,
                ImmSrc, ALUSrcA, ALUSrcB,
                ResultSrc, AdrSrc, IRWrite, PCWrite,
                RegWrite, MemWrite, ALUControl, op, funct3,
                funct7b5, Zero, Adr, ReadData, WriteData);
endmodule

module controller(input  logic      clk,
                  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      Branch, PCUpdate;

    // Main FSM
    mainfsm fsm(clk, reset, op,
                ALUSrcA, ALUSrcB, ResultSrc, AdrSrc,
                IRWrite, PCUpdate, RegWrite, MemWrite,
                ALUOp, Branch);

    // ALU Decoder
    aludec ad(op[5], funct3, funct7b5, ALUOp, ALUControl);

    // Instruction Decoder

```

```

    instrdec id(op, ImmSrc);

    // Branch logic
    assign PCWrite = (Branch & Zero) | PCUpdate;

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, PCUpdate,
               output logic      RegWrite, MemWrite,
               output logic [1:0] ALUOp,
               output logic      Branch);

    typedef enum logic [3:0] {FETCH, DECODE, MEMADR, MEMREAD, MEMWB,
                               MEMWRITE,
                               EXECUTER, EXECUTEI, ALUWB,
                               BEQ, JAL, UNKNOWN} statetype;

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

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

    // next state logic
    always_comb
        case(state)
            FETCH:                nextstate = DECODE;
            DECODE: casez(op)
                7'b0?00011:      nextstate = MEMADR;      // lw or sw
                7'b0110011:      nextstate = EXECUTER;    // R-type
                7'b0010011:      nextstate = EXECUTEI;    // addi
                7'b1100011:      nextstate = BEQ;         // beq
                7'b1101111:      nextstate = JAL;         // jal
                default:          nextstate = UNKNOWN;
            endcase
            MEMADR:
                if (op[5])        nextstate = MEMWRITE;  // sw
                else              nextstate = MEMREAD;    // lw
            MEMREAD:              nextstate = MEMWB;
            EXECUTER:             nextstate = ALUWB;
            EXECUTEI:             nextstate = ALUWB;
            JAL:                  nextstate = ALUWB;
            default:              nextstate = FETCH;
        endcase

    // state-dependent output logic
    always_comb
        case(state)
            FETCH:  controls = 15'b00_10_10_0_1100_00_0;

```

```

        DECODE:          controls = 15'b01_01_00_0_0000_00_0;
        MEMADR:          controls = 15'b10_01_00_0_0000_00_0;
        MEMREAD: controls = 15'b00_00_00_1_0000_00_0;
        MEMWRITE: controls = 15'b00_00_00_1_0001_00_0;
        MEMWB:           controls = 15'b00_00_01_0_0010_00_0;
        EXECUTER:        controls = 15'b10_00_00_0_0000_10_0;
        EXECUTEI: controls = 15'b10_01_00_0_0000_10_0;
        ALUWB:           controls = 15'b00_00_00_0_0010_00_0;
        BEQ:             controls = 15'b10_00_00_0_0000_01_1;
        JAL:             controls = 15'b01_10_00_0_0100_00_0;
        default:         controls = 15'bxx_xx_xx_x_xxxx_xx_x;
    endcase

    assign {ALUSrcA, ALUSrcB, ResultSrc, AdrSrc, IRWrite, PCUpdate,
            RegWrite, MemWrite, ALUOp, Branch} = controls;

endmodule

module aludec(input  logic      opb5,
              input  logic [2:0] funct3,
              input  logic      funct7b5,
              input  logic [1:0] ALUOp,
              output logic [2:0] ALUControl);

    logic RtypeSub;
    assign RtypeSub = funct7b5 & opb5; // TRUE for R-type subtract
    instruction

    always_comb
        case(ALUOp)
            2'b00:          ALUControl = 3'b000; // addition
            2'b01:          ALUControl = 3'b001; // subtraction
            default: case(funct3) // R-type or I-type ALU
                3'b000: if (RtypeSub)
                    ALUControl = 3'b001; // sub
                else
                    ALUControl = 3'b000; // add, addi
                3'b010: ALUControl = 3'b101; // slt, slti
                3'b110: ALUControl = 3'b011; // or, ori
                3'b111: ALUControl = 3'b010; // and, andi
                default: ALUControl = 3'bxxx; // ???
            endcase
        endcase
    endmodule

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

    always_comb
        case(op)
            7'b0110011: ImmSrc = 2'bxx; // R-type
            7'b0010011: ImmSrc = 2'b00; // I-type ALU
            7'b0000011: ImmSrc = 2'b00; // lw
            7'b0100011: ImmSrc = 2'b01; // sw
            7'b1100011: ImmSrc = 2'b10; // beq
            7'b1101111: ImmSrc = 2'b11; // jal
            default: ImmSrc = 2'bxx; // ???
        endcase
    endmodule

```

```

endmodule

module datapath(input  logic      clk, reset,
               input  logic [1:0] ImmSrc, ALUSrcA, ALUSrcB,
               input  logic [1:0] ResultSrc,
               input  logic      AdrSrc,
               input  logic      IRWrite, PCWrite,
               input  logic      RegWrite, MemWrite,
               input  logic [2:0] alucontrol,
               output logic [6:0] op,
               output logic [2:0] funct3,
               output logic      funct7b5,
               output logic      Zero,
               output logic [31:0] Adr,
               input  logic [31:0] ReadData,
               output logic [31:0] WriteData);

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

// next PC logic
flopnr #(32) pcreg(clk, reset, PCWrite, Result, PC);
flopnr #(32) oldpcreg(clk, reset, IRWrite, PC, OldPC);

// memory logic
mux2      #(32) adrmux(PC, Result, AdrSrc, Adr);
flopnr #(32) ir(clk, reset, IRWrite, ReadData, Instr);
flopnr #(32) datareg(clk, reset, ReadData, Data);

// register file logic
regfile      rf(clk, RegWrite, Instr[19:15], Instr[24:20],
               Instr[11:7], Result, RD1, RD2);
extend      ext(Instr[31:7], ImmSrc, immext);
flopnr #(32) srcareg(clk, reset, RD1, A);
flopnr #(32) wdreg(clk, reset, RD2, WriteData);

// ALU logic
mux3      #(32) srcamux(PC, OldPC, A, ALUSrcA, SrcA);
mux3      #(32) srcbmux(WriteData, immext, 32'd4, ALUSrcB, SrcB);
alu      alu(SrcA, SrcB, alucontrol, ALUResult, Zero);
flopnr #(32) aluoutreg(clk, reset, ALUResult, ALUOut);
mux3      #(32) resmux(ALUOut, Data, ALUResult, ResultSrc, Result);

// outputs to control unit
assign op      = Instr[6:0];
assign funct3   = Instr[14:12];
assign funct7b5 = Instr[30];

endmodule

module regfile(input  logic      clk,
               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 combinationaly (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 adder(input  [31:0] a, b,
             output [31:0] y);

    assign y = a + b;
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 (jal)
            2'b11: immext = {{12{instr[31]}}, instr[19:12], instr[20],
instr[30:21], 1'b0};
            default: immext = 32'bx; // undefined
        endcase
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

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

```

```

    always_ff @(posedge clk, posedge reset)
        if (reset)    q <= 0;
        else if (en) q <= d;
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 mux4 #(parameter WIDTH = 8)
    (input  logic [WIDTH-1:0] d0, d1, d2, d3,
     input  logic [1:0]      s,
     output logic [WIDTH-1:0] y);

    assign y = s[1] ? (s[0] ? d3 : d2) : (s[0] ? d1 : d0);
endmodule

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

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

    initial
        $readmemh("riscvtest.txt",RAM);

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

    always_ff @(posedge clk)
        if (we) RAM[a[31:2]] <= wd;
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        v;           // overflow
    logic        isAddSub;     // true when is add or subtract operation

    assign condinvb = alucontrol[0] ? ~b : b;
    assign sum = a + condinvb + alucontrol[0];
    assign isAddSub = ~alucontrol[2] & ~alucontrol[1] |
        ~alucontrol[1] & alucontrol[0];

```

```

always_comb
case (alucontrol)
  3'b000: result = sum;          // add
  3'b001: result = sum;          // subtract
  3'b010: result = a & b;        // and
  3'b011: result = a | b;        // or
  3'b100: result = a ^ b;        // xor
  3'b101: result = sum[31] ^ v;  // slt
  3'b110: result = a << b[4:0];  // sll
  3'b111: result = a >> b[4:0];  // srl
  default: result = 32'bx;
endcase

assign zero = (result == 32'b0);
assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) &
isAddSub;

endmodule

```

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

**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	x2 = 5
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	x3 = 12
11	08	""	S0: Fetch	0C	PC+4
12	0C	FF718393	S1: Decode	X	OldPC+Immediate
13	0C	""	S8: ExecuteI	X	ALUResult = x3 (12) - 9 = 3
14	0C	""	S7: ALUWB	-9	x7 = 3
15					
16					
17					
18					
19					
20					
21					
22					
23					
24					
25					
26					
27					
28					

29					
30					
31					
32					
33					
34					
35					
36					
37					
38					
39					
40					
41					
42					
43					

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

- Simulation waveforms (in the order listed in the lab) 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.

