```systemverilog
// RISCVmulti.sv
// risc-v multicycle processor
// Christian Johnson
// chrjohnson@hmc.edu
// 12/3/2024

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

    // Declaring internal logic for the top module
    logic [31:0]    readdata;

    // instantiate processor and memory
    RISCVmulti      RISCVmulti(clk, reset, memwrite, writedata, dataadr, readdata);
    unimem          unimem(clk, memwrite, dataadr, writedata, readdata);

endmodule

// unified memory module
//      combines both the dmem and imem module from the RISCVsingle processor
module unimem(input  logic          clk, WE,
              input  logic [31:0]   A, WD,
              output logic [31:0]   RD);

    // Declaring and initalizing RAM
    logic  [31:0] RAM[63:0];

    // Instruction memory logic
    initial
      $readmemh("memfile.dat",RAM);

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

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

// RISCVmulti module containing calls to the controller and datapath modules
module RISCVmulti(input         clk, reset,
                  output        memwrite,
                  output [31:0] writedata, dataadr,
                  input  [31:0] readdata);

    // Internal logic for RISCVmulti
    logic           zero, adrsrc, irwrite, pcwrite, regwrite;
    logic [1:0]     resultsrc, alusrcb, alusrca, immsrc;
    logic [2:0]     alucontrol;
    logic [31:0]    instr;

    // instantiate the controller and datapath
    multicycle_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, regwrite,
                adrsrc, pcwrite, irwrite,
                resultsrc, immsrc, alucontrol,
                alusrca, alusrcb,
                readdata, instr, zero, dataadr, writedata);
endmodule

// Datapath module
module datapath(input  logic        clk, reset,
                input  logic        regwrite,
                input  logic        adrsrc,
                input  logic        pcwrite,
                 input  logic        irwrite,
                 input  logic [1:0]  resultsrc, immsrc,
                 input  logic [2:0]  alucontrol,
                 input  logic [1:0]  alusrca, alusrcb,
                 input  logic [31:0] readdata,
       output logic [31:0]  instr,
                 output logic        zero,
                 output logic [31:0] adr, writedata);

    // Defining all internal logic for datapath
    logic [31:0]    pcnext, pc, oldpc;
    logic [31:0]    data;
    logic [31:0]    a, srca, srcb;
    logic [31:0]    immext;
    logic [31:0]    aluresult, aluout;
    logic [31:0]    result;
    logic [31:0]    regfileout1, regfileout2;

    // pc logic
    flopenr #(32) pcreg(clk, reset, pcwrite, pcnext, pc);

    // adr logic
    mux2 #(32)    adrmux(pc, result, adrsrc, adr);

    // oldpc/instr logic
    flopenr #(32) oldpcreg(clk, reset, irwrite, pc, oldpc);
    flopenr #(32) instrreg(clk, reset, irwrite, readdata, instr);

    // data logic
    flopr #(32)   datareg(clk, reset, readdata, data);

    // register file logic
    RegFile       rf(clk, regwrite, instr[19:15], instr[24:20],
                     instr[11:7], result, regfileout1, regfileout2);

    // extend logic
    Extend        ext(instr[31:7], immsrc, immext);

    // a/writedata logic
    flopr #(32)   areg(clk, reset, regfileout1, a);
    flopr #(32)   writedatareg(clk, reset, regfileout2, writedata);

    // srca logic
    mux3  #(32)   srcamux(pc, oldpc, a, alusrca, srca);

    // srcb logic
    mux3  #(32)   srcbmux(writedata, immext, 32'd4, alusrcb, srcb);

    // alu logic
    alu           alu(srca, srcb, alucontrol, aluresult, zero);

    // aluout logic
    flopr #(32)        aluoutreg(clk, reset, aluresult, aluout);

    // result logic
    mux3  #(32)   resultmux(aluout, data, aluresult, resultsrc, result);
    assign        pcnext = result;
endmodule

// Structural Verilog Code for the ALU controller (Adapted from lab10_CJ)
module multicycle_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);

    // Defining internal logic for the multicycle controller
    logic pcupdate, branch;
    logic [1:0] aluop;

    // Calling mainFSM module
    mainFSM fsm(clk, reset, op, pcupdate, branch, regwrite, memwrite, irwrite, resultsrc, 
alusrca, alusrcb, adrsrc, aluop);


    /*
    Calling alu_decoder (from lab 2 implementation)
    aluop[1]: a
    aluop[0]: b

    funct3[2]: c
    funct3[1]: d
    funct3[0]: e

    op[5]:    f
    funct7b5: g
    */
    alu_decoder aludec(aluop[1], aluop[0], funct3[2], funct3[1], funct3[0], op[5], 
funct7b5, alucontrol[2], alucontrol[1], alucontrol[0]);

    // Calling instr_decoder module
    instr_decoder id(op, immsrc);


    // output logic
    assign pcwrite = branch & zero | pcupdate;
endmodule

module mainFSM(input logic clk,
               input logic reset,
               input logic [6:0] op,
               output logic pcupdate,
               output logic branch,
               output logic regwrite, memwrite,
               output logic irwrite,
               output logic [1:0] resultsrc,
               output logic [1:0] alusrca, alusrcb,
               output logic adrsrc,
               output logic [1:0] aluop);

    // Internal logic for states
    typedef enum logic [3:0] {FETCH, DECODE, MEMADR, MEMREAD, MEMWB, MEMWRITE, EXECUTER, 
ALUWB, EXECUTEI, JAL, BEQ} statetype;
    statetype state, nextstate;
    logic [13:0] controls;

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


    // next state logic
    always_comb
        casez (state)
            FETCH:      nextstate = DECODE;

            DECODE:     casez(op)
                                7'b0?00011:        nextstate = MEMADR;
                                7'b0110011:        nextstate = EXECUTER;
                                7'b0010011:        nextstate = EXECUTEI;
                                7'b1101111:        nextstate = JAL;
```

```
                                 7'b1100011:         nextstate = BEQ;
                                 default:            nextstate = state;
                             endcase

            MEMADR:     casez(op)
                                 7'b0000011:         nextstate = MEMREAD;
                                 7'b0100011:         nextstate = MEMWRITE;
                                 default:            nextstate = state;
                             endcase

            MEMREAD:    nextstate = MEMWB;

            MEMWB:      nextstate = FETCH;

            MEMWRITE:   nextstate = FETCH;

            EXECUTER:   nextstate = ALUWB;

            ALUWB:      nextstate = FETCH;

            EXECUTEI:   nextstate = ALUWB;

            JAL:        nextstate = ALUWB;

            BEQ:        nextstate = FETCH;

            default:    nextstate = state;
    endcase



    // setting current state signals
    always_comb
        case (state)
                    //pcupdate__branch__regwrite__memwrite__irwrite__resultsrc__alusrcb__alusrca__adrsrc__aluop
            FETCH:      controls = 14'b1_0_0_0_1_10_10_00_0_00 ;

            DECODE:     controls = 14'b0_0_0_0_0_00_01_01_0_00 ;

            MEMADR:     controls = 14'b0_0_0_0_0_00_01_10_0_00 ;

            MEMREAD:    controls = 14'b0_0_0_0_0_00_00_00_1_00 ;

            MEMWB:      controls = 14'b0_0_1_0_0_01_00_00_0_00 ;

            MEMWRITE:   controls = 14'b0_0_0_1_0_00_00_00_1_00 ;

            EXECUTER:   controls = 14'b0_0_0_0_0_00_00_10_0_10 ;

            ALUWB:      controls = 14'b0_0_1_0_0_00_00_00_0_00 ;

            EXECUTEI:   controls = 14'b0_0_0_0_0_00_01_10_0_10 ;

            JAL:        controls = 14'b1_0_0_0_0_00_10_01_0_00 ;

            BEQ:        controls = 14'b0_1_0_0_0_00_00_10_0_01 ;

            default:    controls = 14'b0_0_0_0_0_00_00_00_0_00 ;
    endcase

        assign {pcupdate, branch, regwrite, memwrite, irwrite, resultsrc, alusrcb, alusrca, adrsrc, aluop} = controls;
    endmodule

    // Structural Verilog Code for the ALU Decoder adapted from lab 2
    module alu_decoder(input  logic a, b, c, d, e, f, g,
                       output logic y2, y1, y0);
    // Declaring the internal logic signals or local variables
    // which can only be used inside of this module
    logic n1, n2, n3, na, nb, nc, nd, ne;

    // Getting negations of each input
    not g1(na, a);
    not g2(nb, b);
    not g3(nc, c);
    not g4(nd, d);
    not g5(ne, e);

    // y2 output logic
    and a1(y2, a, nb, nc, d, ne);

    // y1 output logic
    and a2(y1, a, nb, c, d);

    // y0 output logic
    and a3(n1, na, b);
    and a4(n2, a, nb, nc, nd, ne, f, g);
    and a5(n3, a, nb, d, ne);
    or o1(y0, n1, n2, n3);
endmodule

// Structural Verilog Code for the instr decoder
module instr_decoder(input logic [6:0] op,
               output logic [1:0] immsrc);

    // Logic for choosing the immsrc
    always_comb
        casez (op)
            7'b0110011: immsrc <= 2'b00; // R-type data processing (dont care but setting
to 0's)
            7'b0010011: immsrc <= 2'b00; // I-type data processing
            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'b00;       // ???
    endcase
endmodule

// Extend module
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

// Register File module
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 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

// flopr module (resettable flip-flop)
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

// flopenr module (resetable flip-flop with enable)
/*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 flopenr #(parameter WIDTH = 8)
                (input logic              clk, reset,
                 input logic              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

// 2x1 Mux module
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

// 3x1 Mux module
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

// ALU module
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

```