design.sv

SV/Verilog Design

```
// Code your design here
// deep code risc 8bit  processor
`timescale 1ns / 1ps
module RISC_SPM (clk, rst);
  parameter word_size = 8;
  parameter Sel1_size = 3;
  parameter Sel2_size = 2;
  wire [Sel1_size-1: 0] Sel_Bus_1_Mux;
  wire [Sel2_size-1: 0] Sel_Bus_2_Mux;

  input clk, rst;

  // Data Nets
  wire zero;
  wire [word_size-1: 0] instruction, address, Bus_1, mem_word;

  // Control Nets
  wire Load_R0, Load_R1, Load_R2, Load_R3, Load_PC, Inc_PC, Load_IR;
  wire Load_Add_R, Load_Reg_Y, Load_Reg_Z;
  wire write;

  Processing_Unit M0_Processor (instruction, zero, address, Bus_1, mem_word, Load_R0, Load_R1,
    Load_R2, Load_R3, Load_PC, Inc_PC, Sel_Bus_1_Mux, Load_IR, Load_Add_R, Load_Reg_Y,
    Load_Reg_Z,  Sel_Bus_2_Mux, clk, rst);

  Control_Unit M1_Controller (Load_R0, Load_R1, Load_R2, Load_R3, Load_PC, Inc_PC,
    Sel_Bus_1_Mux, Sel_Bus_2_Mux , Load_IR, Load_Add_R, Load_Reg_Y, Load_Reg_Z,
    write, instruction, zero, clk, rst);

  Memory_Unit M2_SRAM (
    .data_out(mem_word),
    .data_in(Bus_1),
    .address(address),
    .clk(clk),
    .write(write) );
endmodule


module Processing_Unit (instruction, Zflag, address, Bus_1, mem_word, Load_R0, Load_R1, Load_R2,
  Sel_Bus_2_Mux, clk, rst);

  parameter word_size = 8;
  parameter op_size = 4;
  parameter Sel1_size = 3;
  parameter Sel2_size = 2;

  output [word_size-1: 0]   instruction, address, Bus_1;
  output Zflag;

  input [word_size-1: 0]    mem_word;
  input             Load_R0, Load_R1, Load_R2, Load_R3, Load_PC, Inc_PC;
  input [Sel1_size-1: 0]    Sel_Bus_1_Mux;
  input [Sel2_size-1: 0]    Sel_Bus_2_Mux;
  input             Load_IR, Load_Add_R, Load_Reg_Y, Load_Reg_Z;
  input             clk, rst;

  wire          Load_R0, Load_R1, Load_R2, Load_R3;
  wire [word_size-1: 0]     Bus_2;
  wire [word_size-1: 0]     R0_out, R1_out, R2_out, R3_out;
  wire [word_size-1: 0]     PC_count, Y_value, alu_out;
  wire          alu_zero_flag;
  wire [op_size-1 : 0]  opcode = instruction [word_size-1: word_size-op_size];

  Register_Unit         R0  (R0_out, Bus_2, Load_R0, clk, rst);
  Register_Unit         R1  (R1_out, Bus_2, Load_R1, clk, rst);
  Register_Unit         R2  (R2_out, Bus_2, Load_R2, clk, rst);
  Register_Unit         R3  (R3_out, Bus_2, Load_R3, clk, rst);
  Register_Unit         Reg_Y   (Y_value, Bus_2, Load_Reg_Y, clk, rst);
  D_flop            Reg_Z   (Zflag, alu_zero_flag, Load_Reg_Z, clk, rst);
  Address_Register  Add_R   (address, Bus_2, Load_Add_R, clk, rst);
  Instruction_Register  IR  (instruction, Bus_2, Load_IR, clk, rst);
  Program_Counter   PC  (PC_count, Bus_2, Load_PC, Inc_PC, clk, rst);
  Multiplexer_5ch       Mux_1   (Bus_1, R0_out, R1_out, R2_out, R3_out, PC_count, Sel_Bus_1_Mux);
  Multiplexer_3ch       Mux_2   (Bus_2, alu_out, Bus_1, mem_word, Sel_Bus_2_Mux);
  Alu_RISC      ALU (alu_zero_flag, alu_out, Y_value, Bus_1, opcode);
endmodule


module Clock_Unit (clock);
output clock;
 reg clock;
parameter delay = 0;
parameter half_cycle = 10;
initial begin
#delay clock = 0;
forever #half_cycle clock = ~clock;
end
endmodule

module Register_Unit (data_out, data_in, load, clk, rst);
  parameter         word_size = 8;
  output [word_size-1: 0]   data_out;
  input     [word_size-1: 0]    data_in;
  input             load;
  input             clk, rst;
  reg   [word_size-1: 0]    data_out;

  always @ (posedge clk or negedge rst)
    if (rst == 0) data_out <= 0; else if (load) data_out <= data_in;
endmodule

  output        data_out;
  input         data_in;
  input         load;
  input         clk, rst;
  reg       data_out;

  always @ (posedge clk or negedge rst)
    if (rst == 0) data_out <= 0; else if (load == 1)data_out <= data_in;
endmodule

 module Address_Register (data_out, data_in, load, clk, rst);
  parameter word_size = 8;
  output [word_size-1: 0]   data_out;
  input     [word_size-1: 0]    data_in;
  input             load, clk, rst;
  reg   [word_size-1: 0]    data_out;
  always @ (posedge clk or negedge rst)
    if (rst == 0) data_out <= 0; else if (load) data_out <= data_in;
endmodule

module Instruction_Register (data_out, data_in, load, clk, rst);
  parameter word_size = 8;
  output [word_size-1: 0]   data_out;
  input     [word_size-1: 0]    data_in;
  input             load;
  input             clk, rst;
  reg   [word_size-1: 0]    data_out;
  always @ (posedge clk or negedge rst)
    if (rst == 0) data_out <= 0; else if (load) data_out <= data_in;
endmodule

module Program_Counter (count, data_in, Load_PC, Inc_PC, clk, rst);
  parameter word_size = 8;
  input     [word_size-1: 0]    data_in;
  input             Load_PC, Inc_PC;
  input             clk, rst;
  reg   [word_size-1: 0]    count;
  always @ (posedge clk or negedge rst)
    if (rst == 0) count <= 0; else if (Load_PC) count <= data_in; else if  (Inc_PC) count <= count +1;
endmodule

module Multiplexer_5ch (mux_out, data_a, data_b, data_c, data_d, data_e, sel);
  parameter word_size = 8;
  output [word_size-1: 0]   mux_out;
  input     [word_size-1: 0]    data_a, data_b, data_c, data_d, data_e;
  input     [2: 0] sel;

  assign  mux_out = (sel == 0)  ? data_a: (sel == 1)
? data_b : (sel == 2)
? data_c: (sel == 3)
? data_d : (sel == 4)
? data_e : 'bx;
endmodule

module Multiplexer_3ch (mux_out, data_a, data_b, data_c, sel);
  parameter     word_size = 8;
  output        [word_size-1: 0]     mux_out;
  input         [word_size-1: 0]    data_a, data_b, data_c;
  input         [1: 0] sel;

  assign  mux_out = (sel == 0) ? data_a: (sel == 1) ? data_b : (sel == 2) ? data_c: 'bx;
endmodule



ADD         Adds the datapaths to form data_1 + data_2.
SUB         Subtracts the datapaths to form data_1 - data_2.
AND         Takes the bitwise-and of the datapaths, data_1 & data_2.
NOT         Takes the bitwise Boolean complement of data_1.
*/
// Note: the carries are ignored in this model.

module Alu_RISC (alu_zero_flag, alu_out, data_1, data_2, sel);
  parameter word_size = 8;
  parameter op_size = 4;
  // Opcodes
  parameter NOP     = 4'b0000;
  parameter ADD     = 4'b0001;
  parameter SUB     = 4'b0010;
  parameter AND     = 4'b0011;
  parameter NOT     = 4'b0100;
  parameter RD          = 4'b0101;
  parameter WR      = 4'b0110;
  parameter BR      = 4'b0111;
  parameter BRZ         = 4'b1000;

  output            alu_zero_flag;
  output [word_size-1: 0]   alu_out;
  input     [word_size-1: 0]    data_1, data_2;
  input     [op_size-1: 0]  sel;
  reg   [word_size-1: 0]    alu_out;

  assign  alu_zero_flag = ~|alu_out;
  always @ (sel or data_1 or data_2)
     case  (sel)
      NOP:  alu_out = 0;
      ADD:  alu_out = data_1 + data_2;  // Reg_Y + Bus_1
      SUB:  alu_out = data_2 - data_1;
      NOT:  alu_out = ~ data_2;  // Gets data from Bus_1
      default:  alu_out = 0;
    endcase
endmodule


module Control_Unit (
  Load_R0, Load_R1,
  Load_R2, Load_R3,
  Load_PC, Inc_PC,
  Sel_Bus_1_Mux, Sel_Bus_2_Mux,
  Load_IR, Load_Add_R, Load_Reg_Y, Load_Reg_Z,
  write, instruction, zero, clk, rst);

  parameter word_size = 8, op_size = 4, state_size = 4;
  parameter src_size = 2, dest_size = 2, Sel1_size = 3, Sel2_size = 2;
  // State Codes
  parameter S_idle = 0, S_fet1 = 1, S_fet2 = 2, S_dec = 3;
  parameter  S_ex1 = 4, S_rd1 = 5, S_rd2 = 6;
  parameter S_wr1 = 7, S_wr2 = 8, S_br1 = 9, S_br2 = 10, S_halt = 11;
  // Opcodes
  parameter NOP = 0, ADD = 1, SUB = 2, AND = 3, NOT = 4;
  parameter RD  = 5, WR =  6,  BR =  7, BRZ = 8;
  // Source and Destination Codes
  parameter R0 = 0, R1 = 1, R2 = 2, R3 = 3;

  output Load_R0, Load_R1, Load_R2, Load_R3;
  output Load_PC, Inc_PC;
  output [Sel1_size-1:0] Sel_Bus_1_Mux;
  output Load_IR, Load_Add_R;
  output Load_Reg_Y, Load_Reg_Z;
  output [Sel2_size-1: 0] Sel_Bus_2_Mux;
  input [word_size-1: 0] instruction;
  input zero;
  input clk, rst;

  reg [state_size-1: 0] state, next_state;
  reg Load_R0, Load_R1, Load_R2, Load_R3, Load_PC, Inc_PC;
  reg Load_IR, Load_Add_R, Load_Reg_Y;
  reg Sel_ALU, Sel_Bus_1, Sel_Mem;
  reg Sel_R0, Sel_R1, Sel_R2, Sel_R3, Sel_PC;
  reg Load_Reg_Z, write;
  reg err_flag;

  wire [op_size-1:0] opcode = instruction [word_size-1: word_size - op_size];
  wire [src_size-1: 0] src = instruction [src_size + dest_size -1: dest_size];
  wire [dest_size-1:0] dest = instruction [dest_size -1:0];

  // Mux selectors
  assign  Sel_Bus_1_Mux[Sel1_size-1:0] = Sel_R0 ? 0:
              Sel_R1 ? 1:
              Sel_R2 ? 2:
              Sel_R3 ? 3:
              Sel_PC ? 4: 3'bx;  // 3-bits, sized number

  assign  Sel_Bus_2_Mux[Sel2_size-1:0] = Sel_ALU ? 0:
              Sel_Bus_1 ? 1:
              Sel_Mem ? 2: 2'bx;

  always @ (posedge clk or negedge rst) begin: State_transitions
    if (rst == 0) state <= S_idle; else state <= next_state; end

/*  always @ (state or instruction or zero) begin:  Output_and_next_state

Note: The above event control expression leads to incorrect operation.  The state transition causes the activity to be evaluated once, then the
opcode had before the state change, which results in Sel_PC = 0 in state 3, which will cause a return to state 1 at the next clock.  Finally, opcode
is changed, but this does not trigger a re-evaluation because it is not in the event control expression.  So, the caution is to be sure to use opcode
in the event control expression. That way, the final execution of the behavior uses the value of opcode that results from the state change, and leads
to the correct value of Sel_PC.
*/

  always @ (state or opcode or zero) begin: Output_and_next_state
    Sel_R0 = 0;     Sel_R1 = 0;         Sel_R2 = 0;     Sel_R3 = 0;         Sel_PC = 0;
    Load_R0 = 0;    Load_R1 = 0;    Load_R2 = 0;    Load_R3 = 0;    Load_PC = 0;

    Load_IR = 0;    Load_Add_R = 0; Load_Reg_Y = 0; Load_Reg_Z = 0;
    Inc_PC = 0;
    Sel_Bus_1 = 0;
    Sel_ALU = 0;
    Sel_Mem = 0;
    write = 0;
    err_flag = 0;   // Used for de-bug in simulation
    next_state = state;

     case  (state)
           S_idle:
               next_state = S_fet1;

           S_fet1:   begin
                     next_state = S_fet2;
                   Sel_PC = 1;
                   Sel_Bus_1 = 1;
               Load_Add_R = 1;
                     end

           S_fet2:   begin
                      next_state = S_dec;
                   Load_IR = 1;
                   Inc_PC = 1;
                     end

        S_dec:        case  (opcode)
                        NOP: next_state = S_fet1;
                        ADD, SUB, AND: begin
                                          next_state = S_ex1;
                                          Sel_Bus_1 = 1;
                                          Load_Reg_Y = 1;
                                          case  (src)
                                                R0:         Sel_R0 = 1;
                                                R1:         Sel_R1 = 1;
                                                R2:         Sel_R2 = 1;
                                                R3:         Sel_R3 = 1;
                                                default :   err_flag = 1;
                                          endcase
                                        end // ADD, SUB, AND

                        NOT: begin
                               next_state = S_fet1;
                               Load_Reg_Z = 1;
                               Sel_Bus_1 = 1;
                               Sel_ALU = 1;
                               case  (src)
                                      R0:         Sel_R0 = 1;
                                      R1:         Sel_R1 = 1;
                                      R2:         Sel_R2 = 1;
                                      R3:         Sel_R3 = 1;
                                      default :   err_flag = 1;
                               endcase
                               case  (dest)
                                      R0:         Load_R0 = 1;
                                      R2:         Load_R2 = 1;
                                      R3:         Load_R3 = 1;
                                      default:    err_flag = 1;
                               endcase
                               end // NOT

                      RD: begin
                               next_state = S_rd1;
                               Sel_PC = 1; Sel_Bus_1 = 1; Load_Add_R = 1;
                          end // RD

                      WR: begin
                               next_state = S_wr1;
                               Sel_PC = 1; Sel_Bus_1 = 1; Load_Add_R = 1;
                          end  // WR

                      BR: begin
                              next_state = S_br1;
                              Sel_PC = 1; Sel_Bus_1 = 1; Load_Add_R = 1;
                          end  // BR

                      BRZ: if (zero == 1)
                                begin
                                    next_state = S_br1;
                                    Sel_PC = 1; Sel_Bus_1 = 1; Load_Add_R = 1;
                                end // BRZ
                              else
                                begin
                                     next_state = S_fet1;
                                     Inc_PC = 1;
                                end
                      default :
                      endcase  // (opcode)

              S_ex1:      begin
                                next_state = S_fet1;
                                Load_Reg_Z = 1;
                                Sel_ALU = 1;
                                    case  (dest)
                                       R0: begin Sel_R0 = 1; Load_R0 = 1; end
                                       R1: begin Sel_R1 = 1; Load_R1 = 1; end
                                       R2: begin Sel_R2 = 1; Load_R2 = 1; end
                                       R3: begin Sel_R3 = 1; Load_R3 = 1; end
                                       default : err_flag = 1;
                                    endcase
                             end

            S_rd1:        begin
                                next_state = S_rd2;
                                Sel_Mem = 1;
                                Load_Add_R = 1;
                                Inc_PC = 1;
                             end

            S_wr1:        begin
                                next_state = S_wr2;
                                Sel_Mem = 1;
                                Load_Add_R = 1;
                                Inc_PC = 1;
                             end

          S_rd2:      begin
                                next_state = S_fet1;
                                Sel_Mem = 1;
                                    case  (dest)
                                       R1:        Load_R1 = 1;
                                       R2:        Load_R2 = 1;
                                       R3:        Load_R3 = 1;
                                       default :  err_flag = 1;
                                    endcase
                             end

            S_wr2:        begin
                                next_state = S_fet1;
                                write = 1;
                                case  (src)
                                       R0:        Sel_R0 = 1;
                                       R1:        Sel_R1 = 1;
                                       R2:        Sel_R2 = 1;
                                       R3:        Sel_R3 = 1;
                                       default :  err_flag = 1;
                                endcase
                             end

              S_br1:      begin next_state = S_br2; Sel_Mem = 1; Load_Add_R = 1; end
              S_br2:      begin next_state = S_fet1; Sel_Mem = 1; Load_PC = 1; end
              S_halt:         next_state = S_halt;
              default:        next_state = S_idle;
      endcase
  end
endmodule


module Memory_Unit (data_out, data_in, address, clk, write);
  parameter word_size = 8;
  parameter memory_size = 256;

  input [word_size-1: 0] data_in;
  input [word_size-1: 0] address;
  input clk, write;
  reg [word_size-1: 0] memory [memory_size-1: 0];

  assign data_out = memory[address];

  always @ (posedge clk)
    if (write) memory[address] = data_in;
endmodule
```