```
// Written by David_Harris@hmc.edu

// Top level system including MIPS and memories


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

  // instantiate processor and memories
  mips mips (clk, reset, pc, instr, memwrite, dataadr, writedata, readdata);
  imem imem (pc[7:0], instr);
  dmem dmem (clk, memwrite, dataadr, writedata, readdata);

endmodule



// External data memory used by MIPS single-cycle processor

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

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

  assign rd = RAM[a[31:2]];    // word-aligned  read (for lw)
```

```
  always_ff @(posedge clk)
    if (we)
      RAM[a[31:2]] <= wd;      // word-aligned write (for sw)

endmodule



// External instruction memory used by MIPS single-cycle
// processor. It models instruction memory as a stored-program
// ROM, with address as input, and instruction as output


module imem ( input logic [7:0] addr, output logic [31:0] instr);

// imem is modeled as a lookup table, a stored-program byte-addressable ROM
          always_comb
            case (addr)                       // word-aligned fetch
//                  address           instruction
//                  -------           -----------
                    8'h00: instr = 32'h20020005;      // disassemble, by hand
                    8'h04: instr = 32'h2003000c;      // or with a program,
                    8'h08: instr = 32'h2067fff7;      // to find out what
                    8'h0c: instr = 32'h00e22025;      // this program does!
                    8'h10: instr = 32'h00642824;
                    8'h14: instr = 32'h00a42820;
                    8'h18: instr = 32'h10a7000a;
                    8'h1c: instr = 32'h0064202a;
                    8'h20: instr = 32'h10800001;
```

```
                8'h24: instr = 32'h20050000;
                8'h28: instr = 32'h00e2202a;
                8'h2c: instr = 32'h00853820;
                8'h30: instr = 32'h00e23822;
                8'h34: instr = 32'hac670044;
                8'h38: instr = 32'h8c020050;
                8'h3c: instr = 32'h08000011;
                8'h40: instr = 32'h20020001;
                8'h44: instr = 32'hac020054;
                8'h48: instr = 32'h08000012;      // j 48, so it will loop here
              default:  instr = {32{1'bx}};     // unknown address
            endcase
endmodule


// single-cycle MIPS processor, with controller and datapath

module mips (input  logic       clk, reset,
          output logic[31:0]  pc,
          input  logic[31:0]  instr,
          output logic       memwrite,
          output logic[31:0]  aluout, writedata,
          input  logic[31:0]  readdata);

  logic       memtoreg, pcsrc, zero, alusrc, regdst, regwrite, jump;
  logic [2:0]  alucontrol;

  controller c (instr[31:26], instr[5:0], zero, memtoreg, memwrite, pcsrc,
                  alusrc, regdst, regwrite, jump, alucontrol);
```

```
  datapath dp (clk, reset, memtoreg, pcsrc, alusrc, regdst, regwrite, jump,
                    alucontrol, zero, pc, instr, aluout, writedata, readdata);

endmodule




module controller(input  logic[5:0] op, funct,
                  input  logic     zero,
                  output logic     memtoreg, memwrite,
                  output logic     pcsrc, alusrc,
                  output logic     regdst, regwrite,
                  output logic     jump,
                  output logic[2:0] alucontrol);

   logic [1:0] aluop;
   logic       branch;

   maindec md (op, memtoreg, memwrite, branch, alusrc, regdst, regwrite,
                        jump, aluop); // Decoder for the multiplexer signals per instruction

   aludec  ad (funct, aluop, alucontrol); // Decoder for the ALU signals per instruction

   assign pcsrc = branch & zero; // Signal for the PC's branch multiplexer

endmodule
```

```
module maindec (input logic[5:0] op,
                output logic memtoreg, memwrite, branch,
                output logic alusrc, regdst, regwrite, jump,
                output logic[1:0] aluop );
  logic [8:0] controls;

  assign {regwrite, regdst, alusrc, branch, memwrite,
          memtoreg,  aluop, jump} = controls; // Assigning to all the output signals with one
                                              // variable to make assignments easy.
                                              // Note that aluop needs 2 bits. So in total 9 bits are needed.
// Connections for the regwrite, regdst, alusrc, branch, memwrite, memtoreg,  jump can be seen in the
// final datapath image of this lab

// However aluop is an internal wire to communicate with the aludec ALU decoder. So its use can be
// seen in the aludec module.

// Main decoder truth table can be seen in the table 7.3

  always_comb //This block shows the main actions of the controller according to the op instruction
    case(op)      // decide on the regwrite, regdst, alusrc, branch, memwrite, memtoreg,  aluop, jump
                  // in this order.
      6'b000000: controls <= 9'b110000100; // R-type
                                           // register write is needed – regwrite = 1
                                           // rd will be written – regdst = 1
                                           // alu op is 2b'10
```

```
    6'b100011: controls <= 9'b101001000; // LW
                                         // register write is needed – regwrite = 1
                                         // rt will be written – regdst = 0
                                         // Sign extended immediate will be used – alusrc = 1
                                         // value will be written from memory  - memtoreg = 1
                                         // alu op is 2b'00

    6'b101011: controls <= 9'b001010000; // SW
                                         // rt will be written – regdst = 0
                                         // Sign extended immediate will be used – alusrc = 1
                                         // value will be written to memory  - memwrite = 1
                                         // alu op is 2b'00

    6'b000100: controls <= 9'b000100010; // BEQ
                                         // if possible, branch will be done – branch = 1
                                         // alu op is 2b'01

    6'b001000: controls <= 9'b101000000; // ADDI
                                         // register write is needed – regwrite = 1
                                         // rt will be written – regdst = 0
                                         // Sign extended immediate will be used – alusrc = 1
                                         // alu op is 2b'00

    6'b000010: controls <= 9'b000000001; // J
                                         // jump will be done – jump = 1
    default:   controls <= 9'bxxxxxxxxx; // illegal op
  endcase
endmodule
```

```
module aludec (input    logic[5:0] funct,
               input    logic[1:0] aluop,
               output   logic[2:0] alucontrol);
  always_comb
// aludec uses funct and aluop to decide on the output bits
// aluop encoding can be seen in the table 7.1
// aludec truth table can be seen in the table 7.2
    case(aluop)
      2'b00: alucontrol  = 3'b010;  // add  (for lw/sw/addi)
      2'b01: alucontrol  = 3'b110;  // sub   (for beq)
      default: case(funct)          // R-TYPE instructions
          6'b100000: alucontrol  = 3'b010; // ADD
          6'b100010: alucontrol  = 3'b110; // SUB
          6'b100100: alucontrol  = 3'b000; // AND
          6'b100101: alucontrol  = 3'b001; // OR
          6'b101010: alucontrol  = 3'b111; // SLT
          default:   alucontrol  = 3'bxxx; // ???
        endcase
    endcase
endmodule



module datapath (input  logic clk, reset, memtoreg, pcsrc, alusrc, regdst,
                 input  logic regwrite, jump,
```

```
                input  logic[2:0]  alucontrol,
            output logic zero,
                output logic[31:0] pc,
              input  logic[31:0] instr,
            output logic[31:0] aluout, writedata,
              input  logic[31:0] readdata);

 logic [4:0]  writereg;
 logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
 logic [31:0] signimm, signimmsh, srca, srcb, result;

 // next PC logic
 flopr #(32) pcreg(clk, reset, pcnext, pc);
 adder       pcadd1(pc, 32'b100, pcplus4);
 sl2         immsh(signimm, signimmsh);
 adder       pcadd2(pcplus4, signimmsh, pcbranch);
 mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc,
               pcnextbr);
 mux2 #(32)  pcmux(pcnextbr, {pcplus4[31:28],
              instr[25:0], 2'b00}, jump, pcnext);

// register file logic
  regfile     rf (clk, regwrite, instr[25:21], instr[20:16], writereg,
             result, srca, writedata);

  mux2 #(5)    wrmux (instr[20:16], instr[15:11], regdst, writereg);
  mux2 #(32)  resmux (aluout, readdata, memtoreg, result);
  signext        se (instr[15:0], signimm);
```

```
  // ALU logic
   mux2 #(32)  srcbmux (writedata, signimm, alusrc, srcb);
   alu         alu (srca, srcb, alucontrol, aluout, zero);

endmodule




module regfile (input    logic clk, we3,
                input    logic[4:0]  ra1, ra2, wa3,
                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
  // write third port on rising edge of clock. Register0 hardwired to 0.

  always_ff@(posedge clk)
    if (we3)
        rf [wa3] <= wd3;

  assign rd1 = (ra1 != 0) ? rf [ra1] : 0;
  assign rd2 = (ra2 != 0) ? rf[ ra2] : 0;

endmodule
```

```
module alu(input  logic [31:0] a, b,
           input  logic [2:0]  alucont,
           output logic [31:0] result,
           output logic zero);
// ALU operations can be seen in the table 5.1
   always_comb
      case(alucont)
         3'b010: result = a + b;
         3'b110: result = a - b;
         3'b000: result = a & b;
         3'b001: result = a | b;
         3'b111: result = (a < b) ? 1 : 0;
         default: result = {32{1'bx}};
      endcase

   assign zero = (result == 0) ? 1'b1 : 1'b0;
endmodule
```

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

    assign y = a + b;
endmodule
```

```
module sl2 (input  logic[31:0] a,
            output logic[31:0] y);

    assign y = {a[29:0], 2'b00}; // shifts left by 2
endmodule




module signext (input  logic[15:0] a,
                output logic[31:0] y);

  assign y = {{16{a[15]}}, a};    // sign-extends 16-bit a
endmodule




// parameterized register
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



// paramaterized 2-to-1 MUX
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
```