```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 29.11.2024 01:46:40
// Design Name:
// Module Name: processor
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module processor(
input btnC, btnU, CLKK,
        output [6:0] seg, logic dp,
        output [3:0] an,
        output [15:0] led

);
wire clk, reset, memwrite, regwrite, lwstall, branchstall, branch, regdst;
    wire [4:0] WriteRegE, rsD, rtD;
    wire [31:0] writedata, dataaddr, readdata, pc;

    pulse_controller clkpulse (CLKK, btnC, 1'b0, clk);
    pulse_controller resetpulse (CLKK, btnU, 1'b0, reset);

    display_controller dc (CLKK, writedata[7:4], writedata[3:0], dataaddr[7:4], dataaddr[3:0],
seg, dp, an);

    assign led[15] = memwrite;
    assign led[14] = regwrite;
    assign led[12] = reset;
    assign led[12] = branch;
    assign led[11] = lwstall;
    assign led[10] = regdst;
    assign led[9] = branchstall;
    assign led[8] = clk;

endmodule
```

```
// Define pipes that exist in the PipelinedDatapath.
// The pipe between Writeback (W) and Fetch (F), as well as Fetch (F) and Decode (D) is given to you.
// However, you can change them, if you want.
// Create the rest of the pipes where inputs follow the naming conventions in the book.


module PipeFtoD(input logic[31:0] instr, PcPlus4F,
            input logic EN, clk,            // StallD will be connected as this EN
            output logic[31:0] instrD, PcPlus4D);

            always_ff @(posedge clk)
               if(EN)
                  begin
                  instrD<=instr;
                  PcPlus4D<=PcPlus4F;
                  end

endmodule

// Similarly, the pipe between Writeback (W) and Fetch (F) is given as follows.

module PipeWtoF(input logic[31:0] PC,
            input logic EN, clk,            // StallF will be connected as this EN
            output logic[31:0] PCF);

            always_ff @(posedge clk)
               if(EN)
                  begin
                  PCF<=PC;
                  end

endmodule

// ***************************************************************************
// Below, write the modules for the pipes PipeDtoE, PipeEtoM, PipeMtoW yourselves.
// Don't forget to connect Control signals in these pipes as well.
// ***************************************************************************


module PipeDtoE(
input logic reset,
input logic[31:0] instrD, PcPlus4D,
input logic RegWriteD, MemToRegD, MemWriteD, RegDstD,
input logic [3:0] ALUControlD,
input logic [4:0] rsD, rtD, rdD,
input logic EN, clk,
```

```
output logic[31:0] instrE, PcPlus4E,
output logic RegWriteE, MemToRegE, MemWriteE, RegDstE,
output logic [3:0] ALUControlE,
output logic [4:0] rsE, rtE, rdE
);
always_ff @(posedge clk) begin
if(reset) begin
instrE <= 0;
PcPlus4E <= 0;
RegWriteE <= 0;
MemToRegE <= 0;
MemWriteE <= 0;
ALUControlE<= 0;
RegDstE <= 0;
rsE <= 0;
rtE <= 0;
rdE <= 0;
end
else if (EN) begin
instrE <= instrD;
PcPlus4E <= PcPlus4D;
RegWriteE <= RegWriteD;
MemToRegE <= MemToRegD;
MemWriteE <= MemWriteD;
ALUControlE<= ALUControlD;
RegDstE <= RegDstD;
rsE <= rsD;
rtE <= rtD;
rdE <= rdD;
end
end
endmodule

module PipeEtoM(
input logic [4:0] WriteRegE, AluOutE,
input logic RegWriteE, MemToRegE, MemWriteE,WriteDataE,
input logic EN, clk,reset,
output logic RegWriteM, MemToRegM, MemWriteM,WriteDataM,
output logic [4:0] WriteRegM
);
always_ff @(posedge clk) begin
if(reset) begin
RegWriteM <= 0;
MemToRegM <= 0;
MemWriteM <= 0;
WriteRegM <= 0;
WriteDataM <= 0;
end
```

```
if (EN) begin
RegWriteM <= RegWriteE;
MemToRegM <= MemToRegE;
MemWriteM <= MemWriteE;
WriteRegM <= WriteRegE;
WriteDataM <= WriteDataE;
end
end
endmodule

module PipeMtoW(
input  logic clk, reset,
input  logic [31:0] ReadDataM, ALUOutM,
input  logic RegWriteM, MemtoRegM,
input  logic [4:0] WriteRegM,
output logic [31:0] ReadDataW, ALUOutW,
output logic RegWriteW, MemtoRegW,
output logic [4:0] WriteRegW
);

always_ff @(posedge clk or posedge reset)
begin
if (reset)
begin
ALUOutW   <= 32'b0;
ReadDataW <= 32'b0;
WriteRegW <= 5'b0;
MemtoRegW <= 0;
RegWriteW <= 0;
end
else
begin
ReadDataW <= ReadDataM;
WriteRegW <= WriteRegM;
MemtoRegW <= MemtoRegM;
RegWriteW <= RegWriteM;
ALUOutW   <= ALUOutM;
end
end

endmodule




// ********************************************************************************
// End of the individual pipe definitions.
// ********************************************************************************
```

```
// ********************************************************************************
// Below is the definition of the datapath.
// The signature of the module is given. The datapath will include (not limited to) the following items:
//  (1) Adder that adds 4 to PC
//  (2) Shifter that shifts SignImmE to left by 2
//  (3) Sign extender and Register file
//  (4) PipeFtoD
//  (5) PipeDtoE and ALU
//  (5) Adder for PCBranchM
//  (6) PipeEtoM and Data Memory
//  (7) PipeMtoW
//  (8) Many muxes
//  (9) Hazard unit
//  ...?
// ********************************************************************************

module datapath (
    input  logic clk, reset,
    input  logic [2:0] ALUControlD,
    input  logic BranchD,
    input  logic [4:0] rsD, rtD, rdD,
    input  logic [15:0] immD,
    output logic RegWriteE, MemToRegE, MemWriteE,
    output logic [31:0] ALUOutE, WriteDataE,
    output logic [4:0] WriteRegE,
    output logic [31:0] PCBranchE,
    output logic pcSrcE
);

    logic stallF, stallD, ForwardAD, ForwardBD, FlushE, ForwardAE, ForwardBE;
    logic PcSrcD, MemToRegW, RegWriteW;
    logic [31:0] PC, PCF, instrF, instrD, PcSrcA, PcSrcB, PcPlus4F, PcPlus4D;
    logic [31:0] PcBranchD, ALUOutW, ReadDataW, ResultW, RD1, RD2;
    logic [4:0] WriteRegW;


    mux2 #(32) result_mux(ReadDataW, ALUOutW, MemToRegW, ResultW);

    PipeWtoF pWtoF(PC, ~stallF, clk, PCF);

    assign PcPlus4F = PCF + 4;
    mux2 #(32) pc_mux(PcPlus4F, PcBranchD, PcSrcD, PC);

    imem im1(PCF[7:2], instrF);

    PipeFtoD pFtoD(instrF, PcPlus4F, ~stallD, clk, instrD, PcPlus4D);
```

```
regfile rf(clk, RegWriteW, instrD[25:21], instrD[20:16],
        WriteRegW, ResultW, RD1, RD2);

mux2 #(32) muxA(RD1, ResultW, ForwardAE, PcSrcA);
mux2 #(32) muxB(RD2, ResultW, ForwardBE, PcSrcB);
alu ALU(PcSrcA, PcSrcB, ALUControlD, ALUOutE);

assign PCBranchE = PcPlus4D + (immD << 2);

assign RegWriteE = RegWriteD;
assign MemToRegE = MemToRegD;
assign MemWriteE = MemWriteD;
assign WriteDataE = RD2;

PipeDtoE pDtoE(
   .reset(reset),
   .instrD(instrD),
   .PcPlus4D(PcPlus4D),
   .RegWriteD(RegWriteD),
   .MemToRegD(MemToRegD),
   .MemWriteD(MemWriteD),
   .RegDstD(RegDstD),
   .ALUControlD(ALUControlD),
   .rsD(rsD),
   .rtD(rtD),
   .rdD(rdD),
   .EN(~stallD),
   .clk(clk),
   .instrE(instrE),
   .PcPlus4E(PcPlus4E),
   .RegWriteE(RegWriteE),
   .MemToRegE(MemToRegE),
   .MemWriteE(MemWriteE),
   .RegDstE(RegDstE),
   .ALUControlE(ALUControlE),
   .rsE(rsE),
   .rtE(rtE),
   .rdE(rdE)
);

PipeEtoM pEtoM(
   .WriteRegE(WriteRegE),
   .AluOutE(ALUOutE),
   .RegWriteE(RegWriteE),
   .MemToRegE(MemToRegE),
   .MemWriteE(MemWriteE),
   .WriteDataE(WriteDataE),
```

```
        .EN(~stallE),
        .clk(clk),
        .reset(reset),
        .RegWriteM(RegWriteM),
        .MemToRegM(MemToRegM),
        .MemWriteM(MemWriteM),
        .WriteDataM(WriteDataM),
        .WriteRegM(WriteRegM)
    );

    dmem memory(
        .clk(clk),
        .we(MemWriteM),
        .a(ALUOutM),
        .wd(WriteDataM),
        .rd(ReadDataM)
    );

    PipeMtoW pMtoW(
        .clk(clk),
        .reset(reset),
        .ReadDataM(ReadDataM),
        .ALUOutM(ALUOutM),
        .RegWriteM(RegWriteM),
        .MemtoRegM(MemToRegM),
        .WriteRegM(WriteRegM),
        .ReadDataW(ResultW),
        .ALUOutW(ALUOutW),
        .RegWriteW(RegWriteW),
        .MemtoRegW(MemToRegW),
        .WriteRegW(WriteRegW)
    );



endmodule




// Hazard Unit with inputs and outputs named
// according to the convention that is followed on the book.

module HazardUnit(
input logic PcSrcD , JumpD,
input logic RegWriteW,
input logic [4:0] WriteRegW,
input logic RegWriteM, MemToRegM,
```

```
input logic [4:0] WriteRegM,
input logic RegWriteE, MemToRegE,
input logic [4:0] rsE, rtE,
input logic [4:0] rsD, rtD,
output logic [1:0] ForwardAE, ForwardBE,
output logic FlushE, StallD, StallF
);
always_comb begin
if (RegWriteM & (rsE != 0) && (WriteRegM == rsE))
ForwardAE = 2'b10;
else if (RegWriteW & (rsE != 0) & (WriteRegW == rsE))
ForwardAE = 2'b01;
else
ForwardAE = 2'b00;
if (RegWriteM & (rtE != 0) & (WriteRegM == rtE))
ForwardBE = 2'b10;
else if (RegWriteW & (rtE != 0) & (WriteRegW == rtE))
ForwardBE = 2'b01;
else
ForwardBE = 2'b00;
StallF = MemToRegE & ((rtE == rsD) | (rtE == rtD));
StallD = StallF;
FlushE = StallF | PcSrcD | JumpD;
end
endmodule



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

    logic       memtoreg, pcsrc, zero, alusrc, regdst, regwrite, jump;
    logic [2:0]  alucontrol;
    logic       branch;
    logic [1:0]  ForwardAE, ForwardBE;
    logic       FlushE, StallD, StallF;

    logic [31:0] PC, PCF, instrF, instrD, PcPlus4F, PcPlus4D, ALUOutE, WriteDataE, ResultW;
    logic [4:0]  WriteRegE, WriteRegM, WriteRegW;
    logic [31:0] ReadDataM, ALUOutM, RD1, RD2;
```

```
assign instrOut = instr;

HazardUnit hazard_unit(
    .PcSrcD(pcsrc),
    .JumpD(jump),
    .RegWriteW(regwrite),
    .WriteRegW(WriteRegW),
    .RegWriteM(regwrite),
    .MemToRegM(memtoreg),
    .WriteRegM(WriteRegM),
    .RegWriteE(regwrite),
    .MemToRegE(memtoreg),
    .rsE(instrD[25:21]),
    .rtE(instrD[20:16]),
    .rsD(instr[25:21]),
    .rtD(instr[20:16]),
    .ForwardAE(ForwardAE),
    .ForwardBE(ForwardBE),
    .FlushE(FlushE),
    .StallD(StallD),
    .StallF(StallF)
);

controller ctrl(
    .op(instr[31:26]),
    .funct(instr[5:0]),
    .memtoreg(memtoreg),
    .memwrite(memwrite),
    .alusrc(alusrc),
    .regdst(regdst),
    .regwrite(regwrite),
    .jump(jump),
    .alucontrol(alucontrol),
    .branch(branch)
);

datapath dp (
    .clk(clk),
    .reset(reset),
    .ALUControlD(alucontrol),
    .BranchD(branch),
    .rsD(instr[25:21]),
    .rtD(instr[20:16]),
    .rdD(instr[15:11]),
    .immD(instr[15:0]),
    .RegWriteE(regwrite),
    .MemToRegE(memtoreg),
    .MemWriteE(memwrite),
```

```
        .ALUOutE(ALUOutE),
        .WriteDataE(WriteDataE),
        .WriteRegE(WriteRegE),
        .PCBranchE(PC),
        .pcSrcE(pcsrc)
    );

    mux2 #(32) pc_mux(PcPlus4F, PC, pcsrc, PCF);
    assign pc = PCF;

    assign PcPlus4F = PCF + 4;

    mux2 #(32) result_mux(ReadDataM, ALUOutM, memtoreg, ResultW);



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
// Modify it to test your own programs.

module imem ( input logic [5:0] addr, output logic [31:0] instr);

// imem is modeled as a lookup table, a stored-program byte-addressable ROM
          always_comb
            case ({addr,2'b00})                     // word-aligned fetch
//
//        ***************************************************************************
//        Here, you can paste your own test cases that you prepared for the part 1-g.
//        Below is a program from the single-cycle lab.
//        ***************************************************************************
//
//                  address                   instruction
//                  -------             -----------
8'h00: instr = 32'h20020005;
8'h04: instr = 32'h2003fffb;
8'h08: instr = 32'h20040007;
8'h0c: instr = 32'h20050001;
8'h10: instr = 32'h00a43020;
8'h14: instr = 32'h00000062;
8'h18: instr = 32'h0043282a;
8'h1c: instr = 32'h20020007;
8'h20: instr = 32'h20430003;
8'h24: instr = 32'h8c640000;
```

```
8'h28: instr = 32'h20850001;
8'h2c: instr = 32'h20050005;
8'h30: instr = 32'h20040005;
8'h34: instr = 32'h1085fff2;
8'h38: instr = 32'h00000062;
8'h3c: instr = 32'h00000062;
              default:  instr = {32{1'bx}}; // unknown address
            endcase
endmodule


//        ***************************************************************************
//        Below are the modules that you shouldn't need to modify at all..
//        ***************************************************************************

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

  logic [1:0] aluop;

  maindec md (op, memtoreg, memwrite, branch, alusrc, regdst, regwrite,
       jump, aluop);

  aludec  ad (funct, aluop, alucontrol);

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
```

```
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;

  always_comb
    case(op)
      6'b000000: controls <= 9'b110000100; // R-type
      6'b100011: controls <= 9'b101001000; // LW
      6'b101011: controls <= 9'b001010000; // SW
      6'b000100: controls <= 9'b000100010; // BEQ
      6'b001000: controls <= 9'b101000000; // ADDI
      6'b000010: controls <= 9'b000000001; // J
      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
    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 regfile (input    logic clk, reset, 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 @(negedge clk)
            if (reset)
                    for (int i=0; i<32; i++) rf[i] = 32'b0;
     else 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);

    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

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 11/29/2024 01:36:00 PM
// Design Name:
// Module Name: my_cpu_tb
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module my_cpu_tb(

    );
```

```
    logic clk, reset;
    logic [31:0] writedata, dataaddr;
    logic        memwrite, regwrite;
    logic [31:0] pc, PC_prime;
    logic lwstall, branchstall, branch;
    logic [4:0] writereg, rsD, rtD;
    logic regdst;

    mips uut(clk, reset,writedata, dataaddr,memwrite, regwrite, pc, PC_prime,lwstall,
branchstall, branch,writereg, rsD, rtD, regdst);


    always begin
        clk = 0;    #5;
        clk = 1;    #5;
    end

    initial begin
        #50;
        reset = 1; #10;
        reset = 0;
    end

endmodule
```