CS224 Lab 5 Preliminary Report Uygar Aras
22103277
Data Hazards:

Type: Compute-use hazard Reason:

Data computed after the execute stage is not written until the write back step is complete. During the Decode stage, a succeeding instruction could read the incorrect data from the previous instruction's destination register. The source register for the following instruction has not yet been updated with the right data value.

Affected Stages:

Following the instruction is the decode stage, where the computation will provide the incorrect data value. As a result, the Execute and WriteBack stages will execute actions using incorrect data values.

When It Occurs:

When the R type instruction add is being executed, an example can be shown. When the source registers for the following instruction (rs) or the target register (rt) request access to the destination register (rd) of the preceding instruction, the destination register (rd) of the R-type instruction is not yet written.

Type: Load-use hazard Reason:

Until the Memory stage is complete, instructions that require reading from memory cannot read data values. Thus, a future instruction cannot access data that was loaded from memory during a prior instruction's execution during the Execute stage.

Affected Stages:

This two-cycle latency may have an impact on the Execute and Memory stages (in the event that memory writes).

When It Occurs:

A hazard occurs when a subsequent instruction attempts to access data in memory that was loaded by a previous instruction.

Type: Load-store hazard Reason:

When data wanted to store at a memory location just after immediately loaded from memory.

Affected Stages:

Memory stage of storing instruction will affected because wrong data will stored in memory location.

When It Occurs:
Consecutive lw and sw instructions are being used with same rt register. Control Hazards:
Type: Branch hazard
Reason:

Branch decision does not made by the time next instruction fetched from the instruction memory. The branch prediction will made at Memory stage which causes delay.

Affected Stages:

Due to delay at branch prediction unnecessary 3 instruction will fetched in the case of branch misprediction.

When It Occurs: When a branch decision is needed.

Solutions:

Solution for Compute Use hazard:

When the R type instruction add is being executed, an example can be shown. When the source registers for the following instruction (rs) or the target register (rt) request access to the destination register (rd) of the preceding instruction, the destination register (rd) of the R-type instruction is not yet written.

Solution for Load Use Hazard:

Forwarding does not solve this problem. A pause is a type of solution where the pipeline is held until data is available.

Solution for Load Store Hazard:

Stalling is an effective strategy to allow loading from memory done until subsequent instruction fetches data from same register at it’s Decode stage.

Solution for Branch Hazard:

Pipeline can stalled for 3 cycles or with additional hardware (equality comparators) branch decision can be made at an earlier stage. Flushing the fetched instructions also important to fix branch mispredictions.

Logic of Hazard Unit for Forwarding

if ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM) then ForwardAE = 10 else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW) then ForwardAE = 01

else ForwardAE = 00

Logic of Hazard Unit for Stalling & Flushing

lwstall = ((rsD = = rtE) OR (rtD = = rtE)) AND MemtoRegE StallF = StallD = FlushE = lwstall

No hazards

addi $t0,
addi $t1,
addi $t2,
or $t3, $t1, $t2 add $t4, $t0, $t1 sub $t5, $t2, $t0 sw $t0, 10($t4) lw $t5, 8($zero) beq $t1, $zero, 1 sub $t1, $t1, $t2 and $t4, $t2, $t3

Compute Use Hazard

addi $t0, $zero, 8 addi $t1, $zero, 9 sub $t2, $t1, $t0 **Load Use Hazard** addi $t0, $zero, 5 addi $t1, $zero, 10 addi $t2, $zero, 15 lw $t0, 10($t1) add $t3, $t0, $t2 sub $t4, $t2, $t0

Load Store hazard

addi $t0, $zero, 8 addi $t1, $zero, 7 addi $t2, $zero, 6

$zero, 1
$zero, 2
$zero, 3

lw $t0, 6($t1)

sw $t0, 7($t3)

Branch Hazard

addi $t0, $zero, 4 addi $t1, $zero, 4 addi $t2, $zero, 3 addi $t3, $zero, 5 beq $t0, $t1, Continue

addi $t5, $zero, 9

add $t0, $t1, $t2 or $t3, $t2, $t1 addi $t6, $zero, 7 Continue:
and $t0, $t0, $t1

System Verilog Code

timescale 1ns / 1ps

// Define pipes that exist in the PipelinedDatapath.
// The pipes between Writeback (W) and Fetch (F), as well as Decode (D) and Execute (E) are given to you.
// Create the rest of the pipes where inputs follow the naming conventions in the book.

```
module PipeFtoD(input logic[31:0] instrF, PcPlus4F,
        input logic EN, clear, clk, reset,
        output logic[31:0] instrD, PcPlus4D);

        always_ff @(posedge clk, posedge reset)
         if(reset)
            begin
            instrD <= 0;
            PcPlus4D <= 0;
            end
          else if(EN)
            begin
             if(clear)
              begin
                   instrD <= 0;
                   PcPlus4D <= 0;
              end
             else
              begin
                       instrD<=instrF;
                       PcPlus4D<=PcPlus4F;
              end
            end


endmodule
```

// The pipe between Writeback (W) and Fetch (F) is given as follows.

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

        always_ff @(posedge clk, posedge reset)
          if(reset)
            PCF <= 0;
          else if(EN)
            PCF <= PC;
endmodule
```

```
module PipeDtoE(input logic[31:0] RD1, RD2, SignImmD,
                input logic[4:0] RsD, RtD, RdD,
                input logic RegWriteD, MemtoRegD, MemWriteD, ALUSrcD,
                input logic [1:0] RegDstD,
                input logic[2:0] ALUControlD,
                input logic clear, clk, reset,EN,
                output logic[31:0] RsData, RtData, SignImmE,
                output logic[4:0] RsE, RtE, RdE,
                output logic RegWriteE, MemtoRegE, MemWriteE, ALUSrcE,
                output logic [1:0] RegDstE,
                output logic[2:0] ALUControlE);

        always_ff @(posedge clk, posedge reset)
          if(reset || clear)
                begin
                // Control signals
                RegWriteE <= 0;
                MemtoRegE <= 0;
                MemWriteE <= 0;
                ALUControlE <= 0;
                ALUSrcE <= 0;
                RegDstE <= 0;

                // Data
                RsData <= 0;
                RtData <= 0;
                RsE <= 0;
                RtE <= 0;
                RdE <= 0;
                SignImmE <= 0;

                end
            else if (EN)
                begin
                // Control signals
                RegWriteE <= RegWriteD;
                MemtoRegE <= MemtoRegD;
                MemWriteE <= MemWriteD;
                ALUControlE <= ALUControlD;
                ALUSrcE <= ALUSrcD;
                RegDstE <= RegDstD;

                // Data
                RsData <= RD1;
                RtData <= RD2;
                RsE <= RsD;
                RtE <= RtD;
                RdE <= RdD;
                SignImmE <= SignImmD;
                end

    endmodule

    module PipeEtoM(input logic clk, reset, EN, RegWriteE, MemtoRegE, MemWriteE,
                input logic [31:0] ALUOutE, WriteDataE,
                input logic [4:0] WriteRegE,
                output logic RegWriteM, MemtoRegM, MemWriteM, output logic [31:0] ALUOutM,
    WriteDataM, output logic [4:0] WriteRegM);

        always_ff @(posedge clk, posedge reset)
                    if(reset)
                        begin
                        // Control signals
                        RegWriteM <= 0;
                        MemtoRegM <= 0;
                        MemWriteM <= 0;

                        // Data
                        ALUOutM <= 0;
                        WriteDataM <= 0;
                        WriteRegM <= 0;
                        end

                     else if (EN)
                        begin
                        // Control signals
                        RegWriteM <= RegWriteE;
                        MemtoRegM <= MemtoRegE;
                        MemWriteM <= MemWriteE;

                        // Data
                        ALUOutM <= ALUOutE;
                        WriteDataM <= WriteDataE;
                        WriteRegM <= WriteRegE;
                        end

    endmodule

    module PipeMtoW (input logic clk, reset, EN, RegWriteM, MemtoRegM,
                 input logic [31:0] ReadDataM, ALUOutM,
                 input logic [4:0] WriteRegM,
                 output logic RegWriteW, MemtoRegW,
                 output logic [31:0] ReadDataW, ALUOutW,
                 output logic [4:0] WriteRegW);
        always_ff @(posedge clk, posedge reset)
            if(reset)
                begin
                // Control signals
                RegWriteW <= 0;
                MemtoRegW <= 0;

                // Data
                ALUOutW <= 0;
                ReadDataW <= 0;
                WriteRegW <= 0;
                end

             else if (EN)
                begin
                // Control signals
                RegWriteW <= RegWriteM;
                MemtoRegW <= MemtoRegM;

                // Data
                ALUOutW <= ALUOutM;
                ReadDataW <= ReadDataM;
                WriteRegW <= WriteRegM;
                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 SignImmD to left by 2
    //  (3) Sign extender and Register file
    //  (4) PipeFtoD
    //  (5) PipeDtoE and ALU
    //  (5) Adder for PcBranchD
    //  (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 RegWriteD, MemtoRegD, MemWriteD, ALUSrcD,
                input logic [1:0] RegDstD,
                input logic BranchD,
                input logic jal,
                 output logic [31:0] instrF,
                 output logic [31:0] instrD, PC, PCF,
                output logic PcSrcD,
                output logic [31:0] ALUOutE, WriteDataE,
                output logic [1:0] ForwardAE, ForwardBE,
                 output logic ForwardAD, ForwardBD,
                 output logic [31:0] jalAdressD, output logic StallJal, FlushE); // Add or remove
input-outputs if necessary

            // ********************************************************************
            // Here, define the wires that are needed inside this pipelined datapath module
            // ********************************************************************

            //* We have defined a few wires for you
    logic [31:0] PcSrcA, PcSrcB, PcBranchD, PcPlus4F;
            logic [31:0] SignImmD, ShiftedImmD;
            logic [31:0] ResultW;
            logic [4:0] WriteRegW;
            logic [31:0] RD1, RD2;
            logic [31:0] SrcBE;
            logic [31:0] ReadDataM;

            logic [31:0] ReadDataW;
            logic [31:0] ALUOutW;

            logic StallF;
            // logic FlushE;
            logic StallD;
    logic RegWriteW;
    logic [4:0] WriteRegE;
    logic RegWriteM;
    logic MemtoRegM;
    logic [4:0] WriteRegM;
    logic RegWriteE;
    logic MemtoRegE;
    logic MemWriteE;
    logic ALUSrcE;
    logic[1:0] RegDstE;
    logic MemWriteM;
    logic MemtoRegW;
    logic EqualD;
    logic [4:0] rsE;
    logic [4:0] rtE;
    logic [4:0] rdE;
    logic [4:0] rsD;
    logic [4:0] rtD;
    logic [4:0] rdD;

    logic [31:0] PcPlus4D;
    logic [31:0] RsData;
    logic [31:0] RtData;
    logic [31:0] SignImmE;
    logic [31:0] ALUOutM;
    logic [31:0] WriteDataM;
    logic [2:0] ALUControlE;
    logic [31:0] rtMuxOut;
    logic [31:0] rdMuxOut;
    logic [31:0] SrcAE;
    logic [31:0] newResult;
    logic zero;
    // logic StallJal;
        //* You should define others down below (you might want to rename some of the
wires above while implementing the pipeline)

        //* We have provided you with a single-cycle datapath
        //* You should convert it to a pipelined datapath, changing the connections between
modules as necessary

        // Replace with PipeWtoF
    // flopr #(32 pcreg(clk, reset, PC, PCF);
    PipeWtoF pipewf(PC, ~StallF, clk, reset, PCF);

        // Do some operations
    assign PcPlus4F = PCF + 4;
    assign PcSrcB = PcBranchD;
        assign PcSrcA = PcPlus4F;
        logic [31:0] PCtemp;
        mux2 #(32) pc_mux(PcSrcA, PcSrcB, PcSrcD, PCtemp);
        assign jalAdressD = {PcPlus4D[31:28], instrD[25:0], 2'b00};
    mux2 #(32) jal_mux(PCtemp, jalAdressD, jal, PC);
    imem im1(PCF[7:2], instrF);

        // Replace the code below with
        // assign instrD = instrF;
        PipeFtoD pipefd(instrF, PcPlus4F,
            ~StallD, (PcSrcD || jal), clk, reset,
            instrD, PcPlus4D);
```

```
   mux2 #(32) wrMux(ResultW, PcPlus4D, jal, newResult);
        // Decode stage
        regfile rf(clk, reset, RegWriteW, instrD[25:21], instrD[20:16], WriteRegW, newResult,
RD1, RD2);
        signext se(instrD[15:0], SignImmD);

        sl2 shiftimm(SignImmD, ShiftedImmD);
        adder branchadd(PcPlus4D, ShiftedImmD, PcBranchD);
        assign rsD = instrD[25:21];
   assign rtD = instrD[20:16];
   assign rdD = instrD[15:11];

   mux2 #(32) rdmux(RD1,ALUOutM,ForwardAD,rdMuxOut);
   mux2 #(32) rtmux(RD2,ALUOutM,ForwardBD,rtMuxOut);

   assign EqualD = (rdMuxOut == rtMuxOut) ? 1 : 0;

   assign PcSrcD = (EqualD && BranchD);

        // Instantiate PipeDtoE here
        PipeDtoE pipede(RD1, RD2, SignImmD,
                rsD, rtD, rdD,
                RegWriteD, MemtoRegD, MemWriteD, ALUSrcD, RegDstD,
                ALUControlD,
                FlushE, clk, reset, ~StallJal,
                RsData, RtData, SignImmE,
                rsE, rtE,rdE,
                RegWriteE, MemtoRegE, MemWriteE, ALUSrcE, RegDstE,
                ALUControlE);

        // Execute stage
   mux4 #(32) fourmux1(RtData, ResultW, ALUOutM, 32'bx, ForwardBE, WriteDataE);
        mux2 #(32) srcBMux(WriteDataE,SignImmE,ALUSrcE,SrcBE);

   mux4 #(32) fourmux2(RsData,ResultW,ALUOutM,32'bx,ForwardAE,SrcAE);
        //mux2 #(5) wrMux(rtE, rdE, RegDstE, WriteRegE);
        mux4 #(5) wr4Mux(rtE, rdE,5'b11111,5'bx, RegDstE, WriteRegE);

   alu alu(SrcAE, SrcBE, ALUControlE, ALUOutE, zero);

        // Replace the code below with PipeEtoM
        // assign WriteDataE = RD2;
   PipeEtoM pipeem(clk, reset, ~StallJal, RegWriteE, MemtoRegE, MemWriteE,
                ALUOutE, WriteDataE,
                WriteRegE,
                RegWriteM, MemtoRegM, MemWriteM,
                ALUOutM, WriteDataM,
                WriteRegM);
        // Memory stage
        dmem DM(clk, MemWriteM, ALUOutM, WriteDataM, ReadDataM);

        // Instantiate PipeMtoW
    PipeMtoW pipemw(clk, reset, ~StallJal, RegWriteM, MemtoRegM,
              ReadDataM, ALUOutM,
              WriteRegM,
              RegWriteW, MemtoRegW,
              ReadDataW, ALUOutW,
              WriteRegW);

        // Writeback stage
        mux2 #(32) wbmux(ALUOutW, ReadDataW, MemtoRegW, ResultW);

    HazardUnit hazardUnit(RegWriteW, BranchD,
                WriteRegW, WriteRegE,
                RegWriteM,MemtoRegM,
                WriteRegM,
                RegWriteE,MemtoRegE,
                rsE,rtE,
                rsD,rtD,
                jal,
                ForwardAE, ForwardBE,
                FlushE, StallD, StallF, ForwardAD, ForwardBD, StallJal);

endmodule

module HazardUnit( input logic RegWriteW, BranchD,
            input logic [4:0] WriteRegW, WriteRegE,
            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,
            input logic jalSignal,
            output logic [1:0] ForwardAE,ForwardBE,
            output logic FlushE,StallD,StallF,ForwardAD, ForwardBD, output logic StallJal
             ); // Add or remove input-outputs if necessary

        // ********************************************************************
        // Here, write equations for the Hazard Logic.
        // If you have troubles, please study pages ~420-430 in your book.
        // ********************************************************************
    logic lwstall;
    logic branchstall;

    always_comb
    begin
        if ((rsE != 0 ) && (rsE == WriteRegM) && RegWriteM)
            ForwardAE = 2'b10;
        else if ((rsE != 0 ) && (rsE == WriteRegW) && RegWriteW)
            ForwardAE = 2'b01;
        else
             ForwardAE = 2'b00;

        if ((rtE != 0 ) && (rtE == WriteRegM) && RegWriteM)
            ForwardBE = 2'b10;
        else if ((rtE != 0 ) && (rtE == WriteRegW) && RegWriteW)
            ForwardBE = 2'b01;
        else
             ForwardBE = 2'b00;

        ForwardAD = (rsD != 0) && (rsD == WriteRegM) && RegWriteM;
        ForwardBD = (rtD != 0) && (rtD == WriteRegM) && RegWriteM;

        StallJal = jalSignal;

        lwstall  = ((rsD == rtE) || (rtD == rtE)) && MemtoRegE;
        branchstall = (BranchD && RegWriteE && ( (WriteRegE == rsD) || (WriteRegE == rtD)) )
|| (BranchD && MemtoRegM && ( (WriteRegE == rsD) || (WriteRegE == rtD)));
        StallF = (lwstall || branchstall);
        StallD = (lwstall || branchstall);
        FlushE = (lwstall || branchstall);

     end
endmodule


// You can add some more logic variables for testing purposes
// but you cannot remove existing variables as we need you to output
// these values on the waveform for grading
module top_mips (input  logic        clk, reset,
             output  logic[31:0]  instrF,
             output logic[31:0] PC, PCF,
             output logic PcSrcD,
             output logic MemWriteD, MemtoRegD, ALUSrcD, BranchD, jal,
             output logic [1:0] RegDstD,
             output logic RegWriteD,
             output logic [2:0]  alucontrol,
             output logic [31:0] instrD,
             output logic [31:0] ALUOutE, WriteDataE,
             output logic [1:0] ForwardAE, ForwardBE,
             output logic ForwardAD, ForwardBD, output logic [31:0] jalAdressD, output logic
StallJal, FlushE);
```

```
    controller CU(instrD[31:26], instrD[5:0], MemtoRegD, MemWriteD, ALUSrcD, RegDstD,
RegWriteD, alucontrol, BranchD, jal);


    datapath DP(clk, reset, alucontrol, RegWriteD, MemtoRegD, MemWriteD, ALUSrcD,
RegDstD, BranchD, jal,
        instrF, instrD,
        PC, PCF, PcSrcD,
        ALUOutE, WriteDataE,
        ForwardAE, ForwardBE, ForwardAD, ForwardBD, jalAdressD, StallJal, FlushE); // Add
or remove input-outputs as necessary



endmodule


// External instruction memory used by MIPS
// 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-e.
//  An example test program is given below.
//          ****************************************************************************
//
//                  address                 instruction
//                  -------             -----------
//          8'h00: instr = 32'h23FF0000; // add
//              8'h04: instr = 32'h23FF0000; // add
//              8'h08: instr = 32'h0c000000; // Jal 0
//              8'h10: instr = 32'h21080005;
//      default:  instr = {32{1'bx}};       // unknown address
    // Test code for no hazards
      8'h00: instr = 32'h20080005;    // addi $t0, $zero, 5
      8'h04: instr = 32'h2009000c;    // addi $t1, $zero, 12
      8'h08: instr = 32'h200a0006;    // addi $t2, $zero, 6
      8'h0c: instr = 32'h210bfff7;    // addi $t3, $t0, -9
      8'h10: instr = 32'h01288025;    // or $s0, $t1, $t0
      8'h14: instr = 32'h012a8824;    // and $s1, $t1, $t2
```

```
        8'h18: instr = 32'h010b9020;    // add $s2, $t0, $t3
        8'h1c: instr = 32'h010a202a;    // slt $a0, $t0, $t2
        8'h20: instr = 32'h02112820;    // add $a1, $s0, $s1
        8'h24: instr = 32'h02493022;    // sub $a2, $s2, $t1
        8'h28: instr = 32'had320074;    // sw $s2, 0x74($t1)
        8'h2c: instr = 32'h8c020080;    // lw $v0, 0x80($zero)

//      // Test code for Compute-use hazards
        8'h30: instr = 32'h20080005;    // addi $t0, $zero, 5
        8'h34: instr = 32'h21090007;    // addi $t1, $t0, 7
        8'h38: instr = 32'h210A0002;    // addi $t2, $t0, 2
        8'h3c: instr = 32'h012A5025;    // or $t2, $t1, $t2
        8'h40: instr = 32'h01498024;    // and $s0, $t2, $t1
        8'h44: instr = 32'h01108820;    // add $s1, $t0, $s0
        8'h48: instr = 32'h0151902A;    // slt $s2, $t2, $s1
        8'h4c: instr = 32'h02318820;    // add $s1, $s1, $s1
        8'h50: instr = 32'h02329822;    // sub $s3, $s1, $s2
        8'h54: instr = 32'hAD330074;    // sw $s3, 0x74($t1)
        8'h58: instr = 32'h8C020080;    // lw $v0, 0x80($zero)


//        // Test code for  load-use hazard
        8'h5c: instr = 32'h20080005;    // addi $t0, $zero, 5
        8'h60: instr = 32'hac080060;    // sw $t0, 0x60($zero)
        8'h64: instr = 32'h8c090060;    // lw $t1, 0x60($zero)
        8'h68: instr = 32'h212a0004;    // addi $t2, $t1, 4
        8'h6c: instr = 32'h212b0003;    // addi $t3, $t1, 3
        8'h70: instr = 32'h8d6b0058;    // lw $t3, 0x58($t3)
        8'h74: instr = 32'h014b5022;    // sub $t2, $t2, $t3
        8'h78: instr = 32'hac0a0070;    // sw $t2, 0x70($zero)
        8'h7c: instr = 32'h8c080070;    // lw $t0, 0x70($zero)
        8'h80: instr = 32'h8d09006c;    // lw $t1, 0x6c($t0)
        8'h84: instr = 32'h01094820;    // add $t1, $t0, $t1


//        // Test code for branch hazard
        8'h88: instr = 32'h20080005;    // addi $t0, $zero, 5
        8'h8c: instr = 32'h20090003;    // addi $t1, $zero, 3
        8'h90: instr = 32'h11090002;    // beq $t0, $t1, 2
        8'h94: instr = 32'h01285020;    // add $t2, $t1, $t0
        8'h98: instr = 32'h01094022;    // sub $t0, $t0, $t1
        8'h9c: instr = 32'h2129FFFF;    // addi $t1, $t1, -1
        8'ha0: instr = 32'h11280002;    // beq $t1, $t0, 2
        8'ha4: instr = 32'hac0a0050;    // sw $t2, 0x50($zero)
        8'ha8: instr = 32'h01284025;    // or $t0, $t1, $t0
        8'hac: instr = 32'h0128482A;    // slt $t1, $t1, $t0
        8'hb0: instr = 32'h11200002;    // beq $t1, $zero, 2
        8'hb4: instr = 32'h8c0b0050;    // lw $t3, 0x50($zero)
        8'hb8: instr = 32'h01284024;    // and $t0, $t1, $t0
        8'hbc: instr = 32'h1108FFFF;    // beq $t0, $t0, -1

       default:  instr = {32{1'bx}};          // unknown address
            endcase
endmodule


//       ***************************************************************************
//       Below are the modules that you should modify to add more instructions to the CPU
//       ***************************************************************************

module controller(input  logic[5:0] op, funct,
                 output logic     memtoreg, memwrite,
                 output logic     alusrc,
                 output logic [1:0] regdst,
                 output logic regwrite,
                 output logic[2:0] alucontrol,
                 output logic branch, jal);

  logic [1:0] aluop;

  maindec md (op, memtoreg, memwrite, branch, jal, alusrc, regdst, regwrite, 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, jal,
                output logic alusrc,
                output logic [1:0] regdst,
                         output logic regwrite,
                         output logic[1:0] aluop );
  logic [9:0] controls;

   assign {regwrite, regdst, alusrc, branch, jal, memwrite,
              memtoreg,  aluop} = controls;

  always_comb
    case(op)
      6'b000000: controls <= 10'b1010000010; // R-type
      6'b100011: controls <= 10'b1001000100; // LW
      6'b101011: controls <= 10'b0001001000; // SW
      6'b000100: controls <= 10'b0000100001; // BEQ
      6'b001000: controls <= 10'b1001000000; // ADDI
      6'b000011: controls <= 10'b1100010000;  // jal
      default:   controls <= 10'bxxxxxxxxxx; // 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 falling 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

// paramaterized 4-to-1 MUX
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
```