Bilkent University
CS 224 - Section 1 - Preliminary Design Report - Lab 5

Mehmet Akif Şahin - 22203673

15 April 2024

Part B - Potential Hazards

1. Compute use - data hazard : can be solved using forwarding
2. Load use - data hazard : can be solved using forwarding and stalling depending on case
3. Branch - control hazard : can be solved using early branch detection with branch prediction and stalling depending on case

Part C - Hazand Unit Signals

$StallF = StallD = FlushE = lwstall \mid branchstall$

$lwstall = MemToRegE \;\&\; (\; rtE == rsD \mid rtE == rtD \;)$

$branchstall = branchD \;\&\; RegWriteE \;\&\; (\; WriteRegE == rsD \mid WriteRegE == rtD \;)$

$\mid$

$branchD \;\&\; MemToRegM \;\&\; (WriteRegM == rsD \mid WriteRegM == rtD)$

$ForwardAD = RegWriteM \;\&\; (\; rsD \mathrel{!}= 0 \;\&\; rsD == WriteRegM \;)$

$ForwardBD = RegWriteM \;\&\; (\; rtD \mathrel{!}= 0 \;\&\; rtD == WriteRegM \;)$

```
IF  $rsE != 0 \& rsE == WriteRegM \& RegWriteM$
THEN  $ForwardAE = 10$
ELSE IF  $rsE != 0 \& rsE == WriteRegW \& RegWriteW$
THEN  $ForwardAE = 01$
ELSE
THEN  $ForwardAE = 00$

IF  $rtE != 0 \& rtE == WriteRegM \& RegWriteM$
THEN  $ForwardBE = 10$
ELSE IF  $rtE != 0 \& rtE == WriteRegW \& RegWriteW$
THEN  $ForwardBE = 01$
ELSE
THEN  $ForwardBE = 00$
```

Part D - Code

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

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

endmodule

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) begin
                if (reset) begin
                    PCF <= 0;
                end else if (EN) begin
                    PCF <= PC;
                end
            end
endmodule

module PipeDtoE(input logic clk, clear, reset, // connect clear to FlushE
                input logic RegWriteD, MemtoRegD, MemWriteD,
                input logic [2:0] ALUControlD,
                input logic ALUSrcD, RegDstD,
                input logic [31:0] Read1D, Read2D,
                input logic [4:0] RsD, RtD, RdD,
                input logic [31:0] SignImmD,
                output logic RegWriteE, MemtoRegE, MemWriteE,
                output logic [2:0] ALUControlE,
                output logic ALUSrcE, RegDstE,
                output logic [31:0] Read1E, Read2E,
                output logic [4:0] RsE, RtE, RdE,
                output logic [31:0] SignImmE
                );

                always_ff @(posedge clk or posedge reset)
                begin
                    if ( reset )
                    begin
                        RegWriteE <= 0;
                        MemtoRegE <= 0;
                        MemWriteE <= 0;
                        ALUControlE <= 0;
                        ALUSrcE <= 0;
                        RegDstE <= 0;
                        Read1E <= 0;
                        Read2E <= 0;
                        RsE <= 0;
                        RtE <= 0;
                        RdE <= 0;
                        SignImmE <= 0;
                    end
                    else
                    begin
                        if ( clear )
                        begin
                            RegWriteE <= 0;
                            MemtoRegE <= 0;
                            MemWriteE <= 0;
                            ALUControlE <= 0;
                            ALUSrcE <= 0;
                            RegDstE <= 0;
                            Read1E <= 0;
                            Read2E <= 0;
                            RsE <= 0;
                            RtE <= 0;
                            RdE <= 0;
                            SignImmE <= 0;
                        end
                        else
                        begin
                            RegWriteE <= RegWriteD;
                            MemtoRegE <= MemtoRegD;
                            MemWriteE <= MemWriteD;
                            ALUControlE <= ALUControlD;
                            ALUSrcE <= ALUSrcD;
                            RegDstE <= RegDstD;
                            Read1E <= Read1D;
                            Read2E <= Read2D;
                            RsE <= RsD;
                            RtE <= RtD;
                            RdE <= RdD;
                            SignImmE <= SignImmD;
                        end
                    end
                end

endmodule

module PipeEtoM(input logic clk, reset,
                input logic 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 or posedge reset)
                begin
                    if ( reset )
                    begin
                        RegWriteM <= 0;
                        MemtoRegM <= 0;
                        MemWriteM <= 0;
                        ALUOutM <= 0;
                        WriteDataM <= 0;
                        WriteRegM <= 0;
                    end
                    else
                    begin
                        RegWriteM <= RegWriteE;
                        MemtoRegM <= MemtoRegE;
                        MemWriteM <= MemWriteE;
                        ALUOutM <= ALUOutE;
                        WriteDataM <= WriteDataE;
                        WriteRegM <= WriteRegE;
                    end
                end
endmodule

module PipeMtoW(input logic clk, reset,
                input logic 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 or posedge reset)
                begin
                    if (reset)
                    begin
                        RegWriteW <= 0;
                        MemtoRegW <= 0;
                        ReadDataW <= 0;
                        ALUOutW <= 0;
                        WriteRegW <= 0;
                    end
                    else
                    begin
                        RegWriteW <= RegWriteM;
                        MemtoRegW <= MemtoRegM;
                        ReadDataW <= ReadDataM;
                        ALUOutW <= ALUOutM;
                        WriteRegW <= WriteRegM;
                    end
                end

endmodule

module datapath (input  logic clk, reset,
                input logic RegWriteD, MemtoRegD, MemWriteD,
                input  logic[2:0]  ALUControlD,
                input logic ALUSrcD, RegDstD, BranchD, jump,
                input logic stallF, stallD, ForwardAD, ForwardBD, FlushE,
                input logic [1:0] ForwardAE, ForwardBE,

                output logic [4:0] RsD, RtD, RsE, RtE,
                output logic [4:0] WriteRegE, WriteRegM, WriteRegW,
                output logic [5:0] opcode, func,
                output logic RegWriteW, RegWriteM, RegWriteE, MemtoRegE, MemtoRegM,

                output logic MemWriteE,
                output logic[31:0] ALUOutE, WriteDataE, pc, PC_prime,
                output logic [4:0] writereg
                );

    logic EqualD, MemWriteM, ftodclear;
    logic PcSrcD, MemtoRegW;
    logic [31:0] PC, PCF, instrF, instrD, PcSrcA, PcSrcB, PcPlus4F, PcPlus4D, EqualD1, EqualD2;
    logic [31:0] PcBranchD, ALUOutW, ReadDataW, ResultW, RD1, RD2;
    logic [4:0] RdD;

    logic [31:0] PCbranch, SignImmD, SignImmShifted, SrcAE, SrcBE, SrcBEwImm, ALUOutM, WriteDataM, ReadDataM;

    logic [2:0] ALUControlE;
    logic ALUSrcE, RegDstE;
    logic [31:0] Read1E, Read2E;
    logic [4:0] RdE;
    logic [31:0] SignImmE;

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

    PipeWtoF pWtoF(PC, ~stallF, clk, reset, PCF);                          // Writeback stage pipe

    assign pc = PCF;
    assign PC_prime = PC;

    assign PcPlus4F = PCF + 4;                                      // Here PCF is from fetch stage
    mux2 #(32) pc_mux(PcPlus4F, PcBranchD, PcSrcD, PCbranch);             // Here PcBranchD is from decode stage
    mux2 #(32) jump_mux(PCbranch, { PcPlus4D[31:28], instrD[25:0], 2'b00}, jump, PC
       );
    // Note that normally whole PCF should be driven to
    // instruction memory. However for our instruction
    // memory this is not necessary
    imem im1(PCF[7:2], instrF);                                   // Instantiated
         instruction memory

    assign ftodclear = PcSrcD | jump;

    PipeFtoD pFtoD(instrF, PcPlus4F, ~stallD, clk, ftodclear, reset, instrD,
       PcPlus4D);    // Fetch stage pipe

    regfile rf(clk, RegWriteW, instrD[25:21], instrD[20:16],
                WriteRegW, ResultW, RD1, RD2);
                                // Add the rest.

    signext immsignext (instrD[15:0], SignImmD);
    sl2 shiftimm (SignImmD, SignImmShifted);
    adder branchadder (SignImmShifted, PcPlus4D, PcBranchD);

    mux2 #(32) RD1mux (RD1, ALUOutM, ForwardAD, EqualD1);
    mux2 #(32) RD2mux (RD2, ALUOutM, ForwardBD, EqualD2);
    assign EqualD = EqualD1 == EqualD2;
    assign PcSrcD = BranchD && EqualD;

    assign opcode = instrD[31:26];
    assign func = instrD[5:0];

    assign RsD = instrD[25:21];
    assign RtD = instrD[20:16];
    assign RdD = instrD[15:11];

    PipeDtoE pipedtoe (clk, FlushE, reset,
                RegWriteD, MemtoRegD, MemWriteD,
                ALUControlD,
                ALUSrcD, RegDstD,
                RD1, RD2,
                RsD, RtD, RdD,
                SignImmD,
                RegWriteE, MemtoRegE, MemWriteE,
                ALUControlE,
                ALUSrcE, RegDstE,
                Read1E, Read2E,
                RsE, RtE, RdE,
                SignImmE
                );

    assign writereg = RtE;

    mux2 #(5) writeregEmux (RtE, RdE, RegDstE, WriteRegE);

    mux4 #(32) SrcAEmux (Read1E, ResultW, ALUOutM, 0, ForwardAE, SrcAE);
    mux4 #(32) SrcBEmux (Read2E, ResultW, ALUOutM, 0, ForwardBE, SrcBE);

    mux2 #(32) immmux (SrcBE, SignImmE, ALUSrcE, SrcBEwImm);

    alu alu (SrcAE, SrcBEwImm,
               ALUControlE,
               ALUOutE);

    assign WriteDataE = SrcBE;

    PipeEtoM pipeetom (clk, reset,
                             RegWriteE, MemtoRegE, MemWriteE,
                             ALUOutE, WriteDataE,
                             WriteRegE,
                             RegWriteM, MemtoRegM, MemWriteM,
                             ALUOutM, WriteDataM,
                             WriteRegM
                             );

    dmem dmem (clk, MemWriteM,
              ALUOutM, WriteDataM,
              ReadDataM);

    PipeMtoW pipemtow (clk, reset,
                RegWriteM, MemtoRegM,
                ReadDataM, ALUOutM,
                WriteRegM,
                RegWriteW, MemtoRegW,
                ReadDataW, ALUOutW,
                WriteRegW
                );

endmodule

// paramaterized 2-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


module HazardUnit(
                input logic branchD,
                input logic [4:0] WriteRegW, WriteRegM, WriteRegE,
                input logic RegWriteW, RegWriteM, RegWriteE, MemtoRegE, MemtoRegM,
                input logic [4:0] rsE,rtE,
                input logic [4:0] rsD,rtD,
                output logic ForwardAD,ForwardBD,
                output logic [2:0] ForwardAE,ForwardBE,
                output logic FlushE,StallD,StallF, lwstall, branchstall

    );

    // logic lwstall, branchstall;

    always_comb begin
        lwstall = MemtoRegE & ( rtE == rsD | rtE == rtD );
        branchstall = (branchD & RegWriteE & ( WriteRegE == rsD | WriteRegE == rtD
            ))
                                  |
                      (branchD & MemtoRegM & ( WriteRegM == rsD | WriteRegM == rtD
                          ));
        StallF = lwstall | branchstall;
        StallD = lwstall | branchstall;
        FlushE = lwstall | branchstall;
        ForwardAD = RegWriteM & ( rsD != 0 & rsD == WriteRegM );
        ForwardBD = RegWriteM & ( rtD != 0 & rtD == WriteRegM );

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

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


module mips (input  logic        clk, reset,
             output logic [31:0] writedata, dataaddr,
             output logic        memwrite, regwrite,
             output logic [31:0] pc, PC_prime,
             output logic lwstall, branchstall, branch,
             output logic [4:0] writereg, rsD, rtD, regdst

             );

    logic        memtoreg, pcsrc, zero, alusrc, regWriteD, jump;
    logic [2:0]  alucontrol;
    logic [5:0]  op, funct;

    logic stallF, stallD, ForwardAD, ForwardBD, FlushE, RegWriteW, RegWriteM,
        MemtoRegE, MemtoRegM, MemWriteD;
    logic [1:0] ForwardAE, ForwardBE;

    logic [4:0] rsE, rtE, WriteRegE, WriteRegM, WriteRegW;

    datapath dp (clk, reset,
                regWriteD, memtoreg, MemWriteD,
                alucontrol,
                alusrc, regdst, branch, jump,
                stallF, stallD, ForwardAD, ForwardBD, FlushE,
                ForwardAE, ForwardBE,

                rsD, rtD, rsE, rtE,
                WriteRegE, WriteRegM, WriteRegW,
                op, funct,
                RegWriteW, RegWriteM, regwrite, MemtoRegE, MemtoRegM,

                memwrite,
                dataaddr, writedata,
                pc, PC_prime, writereg
                );

    controller cont (op, funct,

                  memtoreg, MemWriteD,
                  alusrc,
                  regdst, regWriteD,
                  jump,
                  alucontrol,
                  branch);

    HazardUnit hu (
                branch,
                WriteRegW, WriteRegM, WriteRegE,
                RegWriteW, RegWriteM, regwrite, MemtoRegE, MemtoRegM,
                rsE,rtE,
                rsD,rtD,
                ForwardAD,ForwardBD,
                ForwardAE,ForwardBE,
                FlushE,stallD,stallF,

                lwstall, branchstall
    );

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'h20080001;
        8'h04: instr = 32'h20090002;
        8'h08: instr = 32'h0109502a;
        8'h0c: instr = 32'h1140ffff;
        8'h10: instr = 32'h0128502a;
        8'h14: instr = 32'h11400001;
        8'h18: instr = 32'h200a0008;
        8'h1c: instr = 32'h08000000;
         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, 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 (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
```

my_cpu.sv

Part E - Hazard Tests

```
// no hazard
addi $s0, $zero, 1
addi $s1, $zero, 2
addi $s2, $zero, 3
addi $s3, $zero, 4
sw $s0, 0($zero)
and $t0, $s0, $s1
add $t1, $s1, $s2
slt $t2, $s2, $s3
sub $t3, $s3, $s0
lw $t4, 0($zero)

// compute use hazard
addi $t0, $zero, 9
sw $t0, 0($zero)
or $t1, $0, $t0
and $t2, $0, $t0
lw $t3, 0($t2)
addi $t4, $zero, $t3

// load use
addi $t0, $zero, 7
addi $t1, $zero 9
lw $t0, 0($zero)
lw $t1, 4($zero)
and $t2, $t1, $t0

// branch hazard
addi $t0, $zero, 1
addi $t1, $zero, 2
slt $t2, $t0, $t1
beq $t2, $zero, -1
slt $t2, $t1, $t0
beq $t2, $zero, 1
addi $t2, $zero, 8
j 0
```

test_codes.asm