Bilkent University

Computer Science Department

CS224 - Computer Organization

Preliminary Design Report

Lab #5

Section #4

Zeynep Doğa Dellal

22002572

22/11/2023

b)

Compute-use hazard:

It is a data type hazard. Execute, Memory or WriteBack stages are affected. The reason this hazard happens is because an instruction cannot use register because it is not written back by the previous instruction yet.

Load-use hazard

It is a data type hazard. Fetch, Decode, Execute and WriteBack stages are affected. It is caused when an instruction tries to read the value of a register when the value is not set yet by a load word (lw) instruction.

Load-store hazard

It is a data type hazard. Fetch, Execute, Decode and WriteBack are affected. Similar to load-use hazard, it happens when a value is stored by store word (sw) instruction that is not yet set by load word (lw) instruction.

Branch hazard

It is a control hazard type. Fetch and Decode stages are affected. The problem is decision is taken in the mem stage of the pipeline, and next instructions until mem stage will be processed.

c) Logic equation for forwarding

if((rsE!=0) AND (rsE == WriteRegM) AND RegWriteM) then

ForwardAE = 10

else if((rsE!=0)AND (rsE == WriteRegW) AND RegWriteM) then

ForwardAE =01

if((rtE!=0)AND (rtE == WriteRegM) AND RegWriteM) then

ForwardBE =10

else if((rtE!=0) AND (rtE == WriteRegW) && RegWriteM) then

ForwardBE = 01

else

ForwardBE = 00

end

Logic equation for forwarding in branch hazard

ForwardAD = (rsD != 0)AND (rsD == WriteRegM) AND RegWriteM

ForwardBD = (rtD != 0)AND (rtD == WriteRegM) AND RegWriteM

Logic equation for lw stall

lwStall = ((rsD == rtE) OR (rtD == rtE)AND MemtoRegE)

StallF = StallD = FlushE = lwStall

Logic Equation for branch stall

branchStall = (BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD))

OR (BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD))

StallF = StallD = FlushE = lwStall OR branchStall

d)

```
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
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[31:0] RD1, RD2, SignImmD,
                input logic[4:0] RsD, RtD, RdD,
                input logic RegWriteD, MemtoRegD, MemWriteD, ALUSrcD, RegDstD,
                input logic[2:0] ALUControlD,
                input logic clear, clk, reset,
                output logic[31:0] RsData, RtData, SignImmE,
                output logic[4:0] RsE, RtE, RdE,
                output logic RegWriteE, MemtoRegE, MemWriteE, ALUSrcE, 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
        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,
        input logic [4:0] WriteRegE,
        input logic [31:0]ALUOutE,
        input logic RegWriteE, MemtoRegE, MemWriteE,
        input logic [31:0] WriteDataE,
        output logic [31:0]ALUOutM,
        output logic [31:0]WriteDataM,
        output logic [4:0] WriteRegM,
        output logic RegWriteM, MemtoRegM, MemWriteM
        );
```

```
    always_ff @(posedge clk, posedge reset)
      if(reset)
          begin
          // Control signals indicated
          RegWriteM <= 0;
          MemWriteM <= 0;
          ALUOutM <= 0;
          MemtoRegM <= 0;
          WriteDataM <= 0;
          WriteRegM <= 0;
          // Data
          end
      else
          begin
          ALUOutM <= ALUOutE;
          WriteRegM <= WriteRegE;
          WriteDataM <= WriteDataE;
          MemtoRegM <= MemtoRegE;
          MemWriteM <= MemWriteE;
          RegWriteM <= RegWriteE;
          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, posedge reset) begin
```

```
  if (reset) begin
    // Reset all signals
    RegWriteW = 0;
    MemtoRegW = 0;
    ReadDataW = 0;
    ALUOutW = 0;
    WriteRegW = 0;
  end
else begin
    // Update control unit signals
    RegWriteW <= RegWriteM;
    MemtoRegW <= MemtoRegM;
    // Update memory stage values
    ReadDataW <= ReadDataM;
    ALUOutW <= ALUOutM;
    WriteRegW <= WriteRegM;
  end
endmodule
module datapath (input  logic clk, reset, RegWriteW,
                         input  logic[2:0]  ALUControlD,
                         input logic BranchD,
                         input logic [31:0] pcPlus4D,
                         input logic [31:0] ResultW,
                         input logic [4:0] rsD,rtD,rdD,
                         input logic [15:0] immD,                    // Add input-outputs if necessary
                         input logic [4:0] WriteRegW,
                         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 [31:0] PcSrcA, PcSrcB, ALUOutE;
 logic [31:0] PcBranchD, PcPlus4F, equll, equl2, adOut;
logic StallF, zero;
//* You should define others down below
   logic [4:0]WriteRegW;
   logic [31:0] PcPlus4D;
   logic [31:0]SignImmD, adderIn;
   logic [31:0]RD1, RD2, RD3, ResultW;
   logic StallD;
   logic equalD;
   logic ALURDSrcE;
   //Pipe 3
   logic[2:0] ALUControlE;
   logic[4:0] RsE, RtE, RdE, WriteRegE;
   logic RegWriteE;
   logic MemtoRegE;
   logic MemWriteE;
   logic ALUSrcE;
   logic RegDstE;
logic[31:0] RsData, RtData,RdData, SignImmE, ALUOutM;
//Pipe 4
logic [4:0] WriteRegM;
logic RegWriteM;
logic MemtoRegM;
logic MemWriteM;
logic[31:0] WriteDataM, ReadDataM;

 //Pipe5
logic MemtoRegW;
logic RegWriteW;
logic[31:0] ALUOutW,ReadDataW;
```

```
//hazardnuit
logic FlushE;
// Pipeline Stage 1: Fetch (F)
  PipeWtoF pipe1(PC, ~StallF, clk, reset, PCF);
  adder adder2(PCF, 32'd4, PcPlus4F);
  mux2 #(32) pc_mux(PcPlus4F, PcBranchD, PCSrcD, PC);

  // Pipeline Stage 2: Decode (D)
  PipeFtoD pipeNumber2(instrF, PcPlus4F, ~StallD, PCSrcD, clk, reset, instrD, PcPlus4D);
  regfile rf1(clk, reset, RegWriteD, instrD[25:21], instrD[20:16], WriteRegW, ResultW, RD1, RD2, RD3);
  signext signext1(instrD[15:0], SignImmD);
  sl2 sl1(SignImmD, adderIn);
  adder adder3(adderIn, PcPlus4D, PcBranchD);
  mux2 #(32) muxeqNumber1(RD1, ALUOutM, ForwardAD, equll );
  mux2 #(32) muxeqNumber2(RD2, ALUOutM, ForwardBD, equl2 );
  assign equalD = (equll == equl2) ? 1 : 0;
  assign PCSrcD = (equalD & BranchD);

  // Pipeline Stage 3: Execute (E)
  PipeDtoE pipeNumber3(RD1, RD2, RD3, SignImmD, instrD[25:21], instrD[20:16], instrD[15:11],
RegWriteD, MemtoRegD, MemWriteD, ALUSrcD, RegDstD, ALUControlD, clk, reset, RsData, RtData,
SignImmE, RsE, RtE, RdE, RegWriteE, MemtoRegE, MemWriteE, ALUSrcE, RegDstE, ALUControlE);
  mux2 #(5) muxDE1(RtE, RdE, RegDstE, WriteRegE );
  mux4 #(32) muxDE2(RsData, ResultW, ALUOutM, 0, ForwardAE, SrcAE );
  mux4 #(32) muxDE3(RtData, ResultW, ALUOutM, 0, ForwardBE, WriteDataE );
  mux2 #(32) muxDE4(WriteDataE, SignImmE, ALUSrcE, SrcBE );
  alu aluDE(SrcAE, SrcBE, ALUControlE, ALUOutE, zero);
  mux2 #(32) muxsracDE(ALUOutE, adOut, ALURDSrcE, ALUOutEMod );
  adder adder5(RdData, ALUOutE, adOut);

  // Pipeline Stage 4: Memory (M)
```

```
   PipeEtoM pipeNumber4(clk, reset, WriteRegE, ALUOutEMod, RegWriteE, MemtoRegE, MemWriteE, WriteDataE, ALUOutM, WriteDataM, WriteRegM, RegWriteM, MemtoRegM, MemWriteM);

   dmem dmem1(clk, MemWriteM, ALUOutM, WriteDataM, ReadDataM);


   // Pipeline Stage 5: Write Back (W)

   PipeMtoW pipeNumber5(clk, reset, RegWriteM, MemtoRegM, ALUOutM, WriteRegM, ReadDataM, RegWriteW, MemtoRegW, WriteRegW, ALUOutW, ReadDataW);

   mux2 #(32) muxNumber5(ALUOutW, ReadDataW, MemtoRegW, ResultW );

   // Hazard Unit

   HazardUnit unitofHazard(RegWriteW, BranchD, WriteRegW, WriteRegE, RegWriteM, MemtoRegM, WriteRegM, RegWriteE, MemtoRegE, RsE, RtE, instrD[25:21], instrD[20:16], ForwardAE, ForwardBE, FlushE, StallD, StallF, ForwardAD, ForwardBD);

endmodule


module HazardUnit( 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 [2:0] ForwardAE,ForwardBE,
                   output logic FlushE,StallD,StallF
   );
   always_comb
            begin
               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
    lwstall <= ((rsD== rtE) | (rtD== rtE)) & MemtoRegE;
    StallF <= lwstall;
    StallD <= lwstall;
    FlushE <= lwstall;
    ForwardAD <= ((rsD != 0) & (rsD == WriteRegM)) & (RegWriteM);
    ForwardBD <= ((rtD != 0) & (rtD == WriteRegM)) & (RegWriteM);
    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;
 assign instrOut = instr;
controller controllerTM(instrD[31:26], instrD[5:0],MemtoRegD, MemWriteD,ALUSrcD,RegDstD,
RegWriteD, alucontrol,BranchD, ALURDSrc);
datapath dpTM (clk, reset,alucontrol,RegWriteD, MemtoRegD, MemWriteD, ALUSrcD, RegDstD,
BranchD,ALURDSrc, instrF, instrD, PC, PCF, PcSrcD, ALUOutE, WriteDataE,ForwardAE,
ForwardBE,ForwardAD, ForwardBD, SrcAE,SrcBE );
imem imem1(PCF[5:0],instr);
endmodule
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;  // 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
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
module mux4 (
   input logic [3:0] data,
   input logic [1:0] sel,
   output logic out);
   assign out = (sel == 2'b00) ? data[0] :
```

```
            (sel == 2'b01) ? data[1] :
            (sel == 2'b10) ? data[2] :
            (sel == 2'b11) ? data[3] :
            1'bx; // Default case (undefined)
endmodule
```

e)

JR might cause branch hazard if the instruction in the pipeline immediately following the jr instruction is dependent on the result of the jr instruction.

When an instruction in the pipeline depends on the outcome of a preceding branch or jump instruction that has not yet updated the program counter (PC) and, therefore, the outcome of the branch or jump instruction is unknown, branch hazard occurs.

In the case of a jr instruction, the jump target address is stored in a register, typically $ra, and the instruction in the pipeline immediately following the jr instruction could be dependent on the value in the $ra register.

For example, consider the following code snippet:

jr $ra

add $t0, $ra, $zero

Assuming that the value in register $ra is not yet updated, the add instruction will use the value of $ra from the previous instruction, which could be incorrect if $ra was updated by the jr instruction. This would result in incorrect program behavior.

To avoid this hazard, the processor needs to ensure that the value in $ra is updated before the add instruction is executed. I mentioned how to fix this issue above.

I used the example jr code that was given in lab 4. I modified it in ways that it has different hazards.

jr hazard:

addi $t0, $zero, 1 # 0x20080001

addi $t1, $zero, 2 # 0x20090002

add $t2, $t0, $t1 # 0x01094820

jr $t3 # 0x013E0020

addi $t3, $zero, 3 # 0x200B0003

compute-use hazard:

add $s1, $zero, $zero # 0x00002820

addi $s0, $zero, 0x00AA # 0x201000AA

addi $s1, $s0, 0x0003 # 0x20110003

and $s2, $s0, $s1 # 0x02229024

load-use hazard:

addi $s0, $zero, 0x0010 # 0x20100010

lw $s0, 0($t0) # 0x8D100000

add $s2, $zero, $s0 # 0x00022020

load-store hazard:

addi $t0, $zero, 0x0010 # 0x20080010

addi $t1, $zero, 0x00CC # 0x200900CC

sw $t0, 0($t1) # 0xAD100000

lw $s0, 0($t1) # 0x8D100000

add $s0, $zero, $s0 # 0x00022020

branch hazard:

addi $s0, $zero, 0x0000 # 0x20100000

addi $s1, $zero, 0x0002 # 0x20110002

beq $s0, $zero, 0x0002 # 0x10210002

addi $s1, $s1, 0x0010 # Not computed due to branch

add $s1, $zero, $s1 # 0x00222020

no hazard:

addi $v0, $zero, 5 # 0x20020005

addi $v1, $zero, 12 # 0x2003000C

addi $a0, $zero, 10 # 0x2004000A

addi $a3, $zero, 20 # 0x20070014

sw $a3, 68($v1) # 0xAC630044

and $a1, $v0, $v1 # 0x00431024

or $a1, $a3, $a0 # 0x00E41025

addi $a3, $a3, 1 # 0x20070001

lw $a0, 0($v1) # 0x8C640000

slt $v0, $a1, $a1 # 0x00A3102A