CS 224 – Computer Organization

Bilkent University

CS

Preliminary Design Report

Lab 5

Section 1

Ahmet Faruk Ulutaş

21803717

Wednesday, 1 December, 13:30-17:20

B)

Data Hazards:

Type: Compute-use hazard

Why happens:

Data computed after the execute stage is not written until the write back stage is completed. During the Decode stage, a subsequent instruction may read incorrect data from the previous instruction's destination register. The correct data value has not yet been written to the next instruction's source register.

Affected Pipeline Stages:

The decode stage of the instruction, which comes after the computation stage, will return an incorrect data value. As a result, the Execute and WriteBack stages will carry out operations with incorrect data values.

When This Hazard Occurs:

When R type instructions are executed, an example can be given. The destination register (rd) of an R-type instruction is not yet written when the following instruction's source registers (rs) or (rt) request to access the destination register (rd) of a previous instruction that has not yet been written.

What's the Solution:

Instead of waiting for the WriteBack stage, you can do it now. Data can be forwarded to the Execute stage of the next instruction, allowing the next instruction to use the correct data value. Stalling can also be used as a solution.

Type: Load-use hazard

Why happens:

Instructions that require memory reading cannot read data values until the Memory stage is completed. As a result, subsequent instructions will be

unable to access data loaded from memory by previous instructions during the Execute stage.

Affected Pipeline Stages:
Possibly Execute and Memory (in case memory write) stages can be affected by this two cycle latency.

When This Hazard Occurs:
Hazard occurs when subsequent instruction tries to access data in memory which is loaded by previous instruction.

What's the Solution:
Forwarding does not solve this problem. Stalling is a one type of solution where pipeline hold until data is available.

Type: Load-store hazard

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

Affected Pipeline Stages:
Memory stage of storing instruction will affected because wrong data will stored in memory location.

When This Hazard Occurs:
Consecutive lw and sw instructions are being used with same rt register.

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

Control Hazards:

Type: Branch hazard

Why happens:

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 Pipeline Stages:

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

When This Hazard Occurs:

When a branch decision needed.

What's the Solution:

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.

C)

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

D)

// 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.

// 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 clr, clk, RegWriteD, MemtoRegD,
MemWriteD, AluSrcD, RegDstD, BranchD,
        input logic[2:0] AluControlD,
        input logic[4:0] RsD, RtD, RdD,
        input logic[31:0] RD1D, RD2D, SignImmD, PCPlus4D,
```

```
        output logic RegWriteE, MemtoRegE, MemWriteE, AluSrcE,
RegDstE, BranchE,
        output logic[2:0] AluControlE,
        output logic[4:0] RsE, RtE, RdE,
        output logic[31:0] RD1E, RD2E, SignImmE, PCPlus4E);

  always_ff @(posedge clk)
  begin
    if(clr)
    begin
      // RESET OR CLEAR SET 0
      RegWriteE <= 0;
      MemtoRegE <= 0;
      MemWriteE <= 0;
      AluSrcE <= 0;
      RegDstE <= 0;
      BranchE <= 0;
      AluControlE <= 0;
      RsE <= 0;
      RtE <= 0;
      RdE <= 0;
      RD1E <= 0;
      RD2E <= 0;
      SignImmE <= 0;
      PCPlus4E <= 0;
```

```
        end
        else
        begin
            // DECODE TO EXECUTE SET DATA
            RegWriteE <= RegWriteD;
            MemtoRegE <= MemtoRegD;
            MemWriteE <= MemWriteD;
            AluSrcE <= AluSrcD;
            RegDstE <= RegDstD;
            BranchE <= BranchD;
            AluControlE <= AluControlD;
            RsE <= RsD;
            RtE <= RtD;
            RdE <= RdD;
            RD1E <= RD1D;
            RD2E <= RD2D;
            SignImmE <= SignImmD;
            PCPlus4E <= PCPlus4D;
        end
    end
endmodule

module PipeEtoM(input logic clk, RegWriteE, MemtoRegE, MemWriteE,
BranchE, ZeroE,
                input logic[4:0] WriteRegE,
```

```
        input logic[31:0] ALUOutE,WriteDataE, PCBranchE,
        output logic RegWriteM, MemtoRegM, MemWriteM, BranchM,
ZeroM,
        output logic[4:0] WriteRegM,
        output logic[31:0] ALUOutM, WriteDataM, PCBranchM);

    always_ff @(posedge clk)
    begin
        // handle control signals
        RegWriteM <= RegWriteE;
        MemtoRegM <= MemtoRegE;
        MemWriteM <= MemWriteE;
        BranchM <= BranchE;
        ZeroM <= ZeroE;
        ALUOutM <= ALUOutE;
        WriteDataM <= WriteDataE;
        WriteRegM <= WriteRegE;
        PCBranchM <= PCBranchE;
    end
endmodule

module PipeMtoW(input logic clk, RegWriteM, MemtoRegM,
        input logic[4:0] WriteRegM,
        input logic[31:0] ReadDataM, ALUOutM,
        output logic RegWriteW, MemtoRegW,
```

```
            output logic[4:0] WriteRegW,
            output logic[31:0] ReadDataW, ALUOutW);

always_ff @(posedge clk)
begin
        RegWriteW <= RegWriteM;
        MemtoRegW <= MemtoRegM;
        WriteRegW <= WriteRegM;
        ReadDataW <= ReadDataM;
        ALUOutW <= ALUOutM;
    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,
        input logic [31:0] instr,
        input logic RegWriteD, MemtoRegD, MemWriteD,
        input logic [2:0] ALUControlD,
        input logic ALUSrcD, RegDstD, BranchD,
        output logic[31:0] instrD, PCF);
```

```
//
************************************************************
********

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

//
************************************************************
********


logic StallD, StallF, ForwardAD, ForwardBD, FlushE, ForwardAE,
ForwardBE;          // Wires for connecting Hazard Unit

// Add the rest of the wires whenever necessary.

logic [31:0] PCPlus4F, PCPlus4D, SignImmD, SignImmE, ResultW, RD1,
RD2, RD1E, RD2E, PCPlus4E, PCOut,

   SrcAE, SrcBE, ALUOut, PCBranchE, ALUOutE, WriteDataE, ALUOutM,
WriteDataM, PCBranchM, ReadDataM, ReadDataW, ALUOutW;

logic RegWriteW, RegWriteE, MemtoRegE, MemWriteE, ALUSrcE,
RegDstE, BranchE, ZeroE, RegWriteM, MemtoRegM, MemWriteM,
BranchM, ZeroM, MemtoRegW;

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

logic [2:0] ALUControlE;

logic [5:0] RsE, RtE, RdE;

logic [15:0] SignImmEShifted;


assign PCSrcM = (BranchM & ZeroM);

assign RsD = instrD[25:21];

    assign RtD = instrD[20:16];

    assign RdD = instrD[15:11];
```

```
//
**********************************************************************
********

// Instantiate the required modules below in the order of the datapath
flow.

//
**********************************************************************
********

adder adds4toPC(PCPlus4D, 4, PCPlus4F);

sl2 shiftSignImmE(SignImmE, SignImmEShifted);

signext signExtender(instr[15:0], SignImmD);

regfile rf (clk, RegWriteW, RsD, RtD, WriteRegW, ResultW, RD1, RD2);

PipeFtoD PipeFtoD(instr, PCPlus4F, ~StallD, clk, instrD, PCPlus4D);

PipeDtoE PipeDtoE(FlushE, clk, RegWriteD, MemtoRegD, MemWriteD,
ALUSrcD, RegDstD,

  BranchD, ALUControlD, RsD, RtD, RdD, RD1, RD2, SignImmD, PCPlus4D,
RegWriteE, MemtoRegE, MemWriteE,

  ALUSrcE, RegDstE, BranchE, ALUControlE, RsE, RtE, RdE, RD1E, RD2E,
SignImmE, PCPlus4E);

alu alu(SrcAE, SrcBE, AluControlE, ALUOut, ZeroE);

   adder adderPCBranchM(SignImmEShifted, PCPlus4E, PCBranchE);

   PipeEtoM PipeEtoM( clk, RegWriteE, MemtoRegE, MemWriteE,
BranchE, ZeroE, WriteRegE, ALUOutE,

      WriteDataE, PCBranchE, RegWriteM, MemtoRegM, MemWriteM,
BranchM, ZeroM, WriteRegM,

      ALUOutM, WriteDataM, PCBranchM);
```

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

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


    mux2 #(32) mux1(PCPlus4F, PCBranchM, PCSrcM, PCOut);

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

    mux3 #(32) mux3(RD1E, ResultW, ALUOutM, SrcAE);

    mux3 #(32) mux4(RD2E, ResultW, ALUOutM, mux3out2);

    mux2 #(32) mux5(mux3out2, SignImmE, ALUSrcE, SrcBE);

    mux2 #(32) mux6(ReadDataW, ALUOutW, MemtoRegW, ResultW);


    HazardUnit HazardUnit( RegWriteW, WriteRegW, RegWriteM,
MemtoRegM, WriteRegM,
        RegWriteE, MemtoRegE, RsE, RtE, RsD, RtD, ForwardAE, ForwardBE,
FlushE, StallD, StallF);


    PipeWtoF PipeWtoF( PCOut, ~StallF, clk, PCF); // mb STALLF normal ?
endmodule


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


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
    );
    logic lwstall;
    always_comb begin


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


    lwstall = ((rsD==rtE) | | (rtD==rtE)) & MemToRegE;
    StallF = lwstall;
    StallD = lwstall;
    FlushE = lwstall;
  end
endmodule


module mips (input  logic       clk,
             input  logic[31:0]  instr,
             output logic[31:0]  PCF);

  logic memtoreg, pcsrc, zero, alusrc, regdst, regwrite, jump, branch;
  logic [2:0]  alucontrol;
  logic [31:0] instrD;
```

```
//
************************************************************
********

// Below, instantiate a controller and a datapath with their new (if
modified) signatures

// and corresponding connections.

//
************************************************************
********

    controller controller( instrD[31:26], instrD[5:0], memtoreg, memwrite,
alusrc, regdst, regwrite, jump, alucontrol, branch);

    datapath datapath( clk, instr, regwrite, memtoreg, memwrite,
alucontrol, alusrc, regdst, branch, instrD, PCF);


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;         // 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 mux3 #(parameter WIDTH = 8)
           (input  logic[WIDTH-1:0] d0, d1, d2,
            input  logic[1:0] s,
            output logic[WIDTH-1:0] y);

  always_comb
      case(s)
      2'b00: y = d0;
      2'b01: y = d1;
      2'b10: y = d2;
      endcase

endmodule
```

E)

Compute-use hazard

addi $t0, $zero, 78’h00: 32’h2008007

addi $1, $t0, 8 8’h04: 32’h2109008

add $t2, $t1, $t0 8’h08: 32’h01285020

Load-use hazard

addi $t0, $zero, 7	8’h00: 32’h20080007
addi $t1, $zero, 7	8’h04: 32’h20090007
addi $a0, $zero, 7	8’h08: 32’h20040007
addi $a1, $zero, 7	8’h0c: 32’h20050007
sw $t0, 0($t1)	8’h10: 32’had280000
lw $t1, 1($t0)	8’h14: 32’h8d090001
add $t2, $t1, $a0 //hazard	8’h18: 32’h01245020
Sub $t2, $t1, $a1	8’h1c: 32’h01255022

Branch Hazard

addi $t1, $zero, 7	8’h00: 32’h20090007
beq $zero, $zero, 2	8’h04: 32’h10000002
addi $t1, $zero, 8	8’h08: 32’h20090008
addi $t1, $zero, 8	8’h0c: 32’h21290008
addi $t1, $zero, 0	8’h10: 32’h20090008
addi $a0, $zero, 0	8’h14: 32’h20040000
addi $a1, $t1, 0 //hazard	8’h18: 32’h20050000
sw $t1, 0($zero)	8’h1c: 32’hac090000