CS 224

Section.: 2

Spring 2019

Lab No.: 5

Zeynep Cankara/ 21703381

Question 1:

List of Possible Hazards that Can Occur in The Pipeline:

Data Hazards:

Type: `Compute-use hazard`

Why happens:

After the execute stage data computed is not written until the end of the write back stage. A subsequent instruction can read the wrong data from the previous instruction's destination register during Decode stage. The correct data value is not written in the source register of the next instruction yet.

Affected Pipeline Stages:

Decode stage of the instruction which follows the instruction where computation done will fetch wrong data value. Thus Execute and WriteBack stage will perform operations by using wrong data value.

When This Hazard Occurs:

An example can be given when R type instruction `add` being executed. R-type instruction's destination register `(rd)` not yet written when following instruction's source registers `(rs)` or `(rt)` request to access `(rd)` of previous instruction which is not written yet.

What's the Solution:

Instead of waiting for the WriteBack stage. Data can Forwarded to the next instruction's Execute stage for enabling the following instruction use correct data value. Stalling can be used as a solution as well.

Type: `Load-use hazard`

Why happens:

Instructions which require reading from memory cannot read data value until end of the Memory stage. Thus subsequent instruction cannot access the data at Execute stage which will be loaded from memory by previous instruction.

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.

Question 2:

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
```

Question 3:

```
`timescale 1ns / 1ps

// 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, reset,              // StallD will be connected as this EN
                output logic[31:0] instrD, PcPlus4D);


   always_ff @(posedge clk, posedge reset)begin
      if (reset)begin
         instrD <= 0;
         PcPlus4D <= 0;
      end
      else if(EN)
         begin
         instrD<=instr;
         PcPlus4D<=PcPlus4F;
         end
      else begin
         instrD <= instrD;
         PcPlus4D <= PcPlus4D;
      end
   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, reset,              // StallF will be connected as this EN
                output logic[31:0] PCF);
   always_ff @(posedge clk, posedge reset)begin
      if (reset)
         PCF <= 0;
      else if(EN)
         begin
         PCF<=PC;
         end
      else
         PCF <= PCF;
   end
endmodule

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

// Below, you are given the argument lists of the modules PipeDtoE, PipeEtoM, PipeMtoW.

// You should implement them by looking at pipelined processor image given to you.

// Don't forget that these codes are tested but you can always make changes if you want.

// Note that some output logics there for debugging purposes, in other words to trace their values in
simulation.

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

module PipeDtoE(input logic clr, clk, reset, RegWriteD, MemtoRegD, MemWriteD,
                input logic[2:0] AluControlD,
                input logic AluSrcD, RegDstD, BranchD,
                input logic[31:0] RD1D, RD2D,
                input logic[4:0] RsD, RtD, RdD,
```

```
    input logic[31:0] SignImmD,
    input logic[31:0] PCPlus4D,
        output logic RegWriteE, MemtoRegE, MemWriteE,
        output logic[2:0] AluControlE,
        output logic AluSrcE, RegDstE, BranchE,
        output logic[31:0] RD1E, RD2E,
        output logic[4:0] RsE, RtE, RdE,
        output logic[31:0] SignImmE,
        output logic[31:0] PCPlus4E);

always_ff @(posedge clk, posedge reset)begin
  // ****************** MY CODE STARTS ************************** /////
  if(reset)
  begin
    // handle control unit signals
    RegWriteE <= 0;
    MemToRegE <= 0;
    MemWriteE <= 0;
    AluControlE <= 0;
    AluSrcE <= 0;
    RegDstE <= 0;
    BranchE <= 0;

    // handle mux signals for forwarding
    RD1E <= 0;
    RD2E <= 0;

    // handle registers
```

```
    RsE <= 0;
    RtE <= 0;
    RdE <= 0;

    // sign extended immediate
    SignImmE <= 0;

    // propagate PC
    PCPlus4E <= 0;
end
else if(clr)
begin
    // Only clear signal which can change architectural state or update memory enough
    // Guranteed by clearing all signals just to be safe
    // handle control unit signals
    RegWriteE <= 0;
    MemToRegE <= 0;
    MemWriteE <= 0;
    AluControlE <= 0;
    AluSrcE <= 0;
    RegDstE <= 0;
    BranchE <= 0;
    // handle mux signals for forwarding
    RD1E <= 0;
    RD2E <= 0;
    // handle registers
    RsE <= 0;
    RtE <= 0;
```

```
    RdE <= 0;
    // sign extended immediate
    SignImmE <= 0;

    // propagate PC
    PCPlus4E <= 0;
end
else
begin
    // handle control unit signals
    RegWriteE <= RegWriteD;
    MemToRegE <= MemToRegD;
    MemWriteE <= MemWriteD;
    AluControlE <= AluControlD;
    AluSrcE <= AluSrcD;
    RegDstE <= RegDstD;
    BranchE <= BranchD;
    // handle mux signals for forwarding
    RD1E <= RD1D;
    RD2E <= RD2D;
    // handle registers
    RsE <= RsD;
    RtE <= RtD;
    RdE <= RtD;
    // sign extended immediate
    SignImmE <= SignImmD;
    // propagate PC
    PCPlus4E <= PCPlus4D;
```

```
    end
    // ****************** MY CODE ENDS  ************************ /////
  end
endmodule

module PipeEtoM(input logic clk, reset, RegWriteE, MemtoRegE, MemWriteE, BranchE, Zero,
            input logic[31:0] ALUOut,
            input logic [31:0] WriteDataE,
            input logic[4:0] WriteRegE,
            input logic[31:0] PCBranchE,
              output logic RegWriteM, MemtoRegM, MemWriteM, BranchM, ZeroM,
              output logic[31:0] ALUOutM,
              output logic [31:0] WriteDataM,
              output logic[4:0] WriteRegM,
              output logic[31:0] PCBranchM);

  always_ff @(posedge clk, posedge reset) begin
    // ****************** MY CODE STARTS ************************ /////
    if (reset)
    begin
      // reset the output signals
      // handle control signals
      RegWriteM <= 0;
      MemtoRegM <= 0;
      MemWriteM <= 0;

      // decision signals for branching
      BranchM <= 0;
```

```
    ZeroM <= 0;

    // execute stage ALU result
    ALUOutM <= 0;

    // memory signals
    WriteDataM <= 0;
    WriteRegM <= 0;
    PCBranchM <= 0;
end
else
begin
    // handle control signals
    RegWriteM <= RegWriteE;
    MemtoRegM <= MemtoRegE;
    MemWriteM <= MemWriteE;

    // decision signals for branching
    BranchM <= BranchE;
    ZeroM <= Zero;

    // execute stage ALU result
    ALUOutM <= ALUOut;

    // memory signals
    WriteDataM <= WriteDataE;
    WriteRegM <= WriteRegE;
    PCBranchM <= PCBranchE;
```

```
    end
    // ****************** MY CODE ENDS  ************************** /////
  end
endmodule

module PipeMtoW(input logic clk, reset, 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
      // ****************** MY CODE STARTS ************************** /////
              if(reset)
      begin
        // reset all signals
        // control unit signals
        RegWriteW <= 0;
        MemtoRegW <= 0;

        // memory stage values
        ReadDataW <= 0;
        ALUOutW <= 0;
        WriteRegW <= 0;
      end
      else
      begin
```

```
            // control unit signals
            RegWriteW <= RegWriteM;
            MemtoRegW <= MemtoRegM;

            // memory stage values
            ReadDataW <= ReadDataM;
            ALUOutW <= ALUOutM;
            WriteRegW <= WriteRegM;
        end
        // ****************** MY CODE ENDS  ************************ /////
    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 [done]
//  (2) Shifter that shifts SignImmE to left by 2 [done]
//  (3) Sign extender and Register file [done]
//  (4) PipeFtoD [done]
//  (5) PipeDtoE and ALU [done]
//  (5) Adder for PCBranchM [done]
//  (6) PipeEtoM and Data Memory [done]
```

```
// (7) PipeMtoW [done]
// (8) Many muxes [done]
// (9) Hazard unit
// ...?
// ***************************************************************************
module datapath (input  logic clk, reset,
                 input logic [31:0] PCF, instr,
                 input logic RegWriteD, MemtoRegD, MemWriteD,
                 input logic [2:0] ALUControlD,
                 input logic AluSrcD, RegDstD, BranchD,
                    output logic PCSrcM, StallD, StallF,
                    output logic[31:0] PCBranchM, PCPlus4F, instrD, ALUOut, ResultW,
WriteDataM);
        // *******************************************************************
        // Here, define the wires (logics) that are needed inside this pipelined datapath module
         // You are given the wires connecting the Hazard Unit.
         // Notice that StallD and StallF are given as output for debugging
        // *******************************************************************
        logic ForwardAD, ForwardBD,  FlushE;
        logic [1:0] ForwardAE, ForwardBE;
        // Add necessary wires (logics).


   // ------- NEW LOGICS -------//
   logic [31:0] ALUOutM, SrcAE, SrcBE, WriteDataENew, RD1D, RD2D, SignImmEShifted,
SignImmD, PCBranchE;
   logic [2:0] AluControlE;
   logic Zero, RegWriteE, MemtoRegE, MemWriteE;
   logic AluSrcE, RegDstE, BranchE;
   logic [4:0] WriteRegE, WriteRegM, WriteRegW;
```

```
logic RegWriteM, MemtoRegM, MemWriteM, BranchM, ZeroM;
logic [31:0] ReadDataM, ReadDataW, ALUOutW;
logic RegWriteW, MemtoRegW;
// ------- END NEW LOGICS ---//

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

// ============ Modified According to prompt ======== //
    assign PCSrcM = BranchM & ZeroM;
    assign RsD = instrD[25:21];
assign RtD = instrD[20:16];
assign RdD = instrD[15:11];
assign WriteDataE = WriteDataENew
// ============ Modified According to prompt ======== //

    // Below, PipeFtoD and regfile instantiations are given
// Add other instantiations
// BE CAREFUL ABOUT THE ORDER OF PARAMETERS!
    PipeFtoD ftd(instr, PCPlus4F, StallD, clk, reset, instrD, PCPlus4D);
    regfile rf (clk, RegWriteW, instrD[25:21], instrD[20:16],
    WriteRegW, ResultW, RD1D, RD2D);
// =============== MY CODE STARTS HERE ================= ///
// adder adds 4 to the PC
adder adderPC(PCF, 4, PCPlus4F);
// sign extended immediate
signext signExt(instrD[15:0], SignImmD);
```

```
// Decode to Execute pipe
PipeDtoE dte(clr, clk, reset, RegWriteD, MemtoRegD, MemWriteD,
             AluControlD,
             AluSrcD, RegDstD, BranchD,
             RD1D, RD2D,
             RsD, RtD, RdD,
             SignImmD,
             PCPlus4D,
             RegWriteE, MemtoRegE, MemWriteE,
             AluControlE,
             AluSrcE, RegDstE, BranchE,
             RD1E, RD2E,
             RsE, RtE, RdE,
             SignImmE,
             PCPlus4E);


// need 4 muxes at execute stage 2 2:1 and 2 3:1 mux
// 2:1 muxes one determines ALU input the other one WriteReg
mux2 #(5) mux2EWriteReg(RtE, RdE,
            RegDstE,
            WriteRegE);
mux2 #(32) mux2ESrcBE(WriteDataE, SignImmE,
          AluSrcE,
          SrcBE);
// 3:1 muxes needed to compute ALU source
mux4 #(32) mux3ESrcAE(RD1E, ResultW, ALUOutM, 0,
            ForwardAE,
            SrcAE);
```

```
mux4 #(32) mux3EWriteDataE(RD2E, ResultW, ALUOutM, 0,
        ForwardBE,
        WriteDataE);
// shifter for PC
sl2 shiftSignImmE(SignImmE,
        SignImmEShifted);
// adder for PCBranchM
adder adderPCBranchM(SignImmEShifted, PCPlus4E, PCBranchE);

// Execute Stage (ALU)
alu alu1(SrcAE, SrcBE,
        AluControlE,
        ALUOut,
        Zero, reset);
// execute to memory pipe
PipeEtoM etm( clk, reset, RegWriteE, MemtoRegE, MemWriteE, BranchE, Zero,
            ALUOut,
            WriteDataE,
            WriteRegE,
            PCBranchE,
            RegWriteM, MemtoRegM, MemWriteM, BranchM, ZeroM,
            ALUOutM,
            WriteDataM,
            WriteRegM,
            PCBranchM);
// data memory
dmem dataMemory( clk, MemWriteM,
        ALUOutM, WriteDataM,
```

```
            ReadDataM);
    // memory to write back pipe
    PipeMtoW mtw(clk, reset, RegWriteM, MemtoRegM,
                ReadDataM, ALUOutM,
                WriteRegM,
                RegWriteW, MemtoRegW,
                ReadDataW, ALUOutW,
                WriteRegW);
    // mux to choose ResultW
    mux2 #(32) mux2WResultW(ALUOutW, ReadDataW,
            MemtoRegW,
            ResultW);
    // ================ MY CODE ENDS HERE ==================== ///
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 [1:0] 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.
    // ********************************************************************

    // =============== MY CODE STARTS HERE ==================== ///
    // for stalling and flushing
    lwstall = ((rsD == rtE) | (rtD == rtE)) & MemtoRegE;
    StallF = StallD = FlushE = lwstall;

    // for forwarding
    if ((rsE != 0) & (rsE == WriteRegM) & RegWriteM)
    begin
     ForwardAE = 2'b10;
    end
    else if ((rsE != 0) & (rsE == WriteRegW) & RegWriteW)
    begin
       ForwardAE = 2'b01;
    end
    else
       ForwardAE = 2'00;
    // =============== MY CODE ENDS HERE ================ ///
  end
endmodule
```

```
module mips (input  logic        clk, reset,
             output logic[31:0]  PCF,
             input  logic[31:0]  instr,
             output logic[31:0]  aluout, resultW,
             output logic[31:0]  instrOut, WriteDataM,
             output logic StallD, StallF);

  // *************************************************************************
  // You can change the logics below but if you didn't change the signitures of
  // above modules you will need these.
  // *************************************************************************

  logic memtoreg, zero, alusrc, regdst, regwrite, jump, PCSrcM, branch, memwrite;
  logic [31:0] PCPlus4F, PCm, PCBranchM, instrD;
  logic [2:0] alucontrol;
  assign instrOut = instr;

        // ***********************************************************************
        // Below, instantiate a controller and a datapath with their new (if modified)
        // signatures and corresponding connections.
  // Also, you might want to instantiate PipeWtoF and pcsrcmux here.
  // Note that this is not the only solution.
  // You can do it in your way as long as it works.
        // ***********************************************************************
```

```
    // =============== MY CODE STARTS HERE ==================== ///

    controller c (instr[31:26], instr[5:0],
                  memtoreg, memwrite,
                  alusrc,
                  regdst, regwrite,
                  jump,
                  alucontrol,
                  branch);
    datapath dp (clk, reset,
                      PCF, instr,
                       RegWriteD, MemtoRegD, MemWriteD,
                       alucontrol,
                       alusrc, regdst, branch,
                       PCSrcM, StallD, StallF,
                       PCBranchM, PCPlus4F, instrD, aluout, resultW, WriteDataM);
    PipeWtoF wtf (PCm,
          1'b1, clk, reset,
          PCF);
    mux2 #(32) pcsrcmux(PCPlus4F, PCBranchM,
              PCSrcM,
              PCm);
    // =============== MY CODE ENDS HERE ====================== ///
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 should paste the test cases that are given to you in lab document.
//  You can write your own test cases and try it as well.
//      Below is the program from the single-cycle lab.
//      ***************************************************************************
```

```
//  ========= MY TEST WITH SUPPORTED INSTRUCTIONS ==============
```

```
//
//              address         instruction
//              -------         -----------
        8'h00: instr = 32'h20020005;     // addi $v0, $0, 0x0005
        8'h04: instr = 32'h2003000c;     // addi $v1, $0, 0x000c
        8'h08: instr = 32'h2067fff7;     // addi $a3, $v1, 0xfff7
        8'h0c: instr = 32'h00e22025;     // or $a0, $a3, $v0
        8'h10: instr = 32'h00642824;    // and $a1, $v1, $a0
        8'h14: instr = 32'h00a42820;    // add $a1, $a1, $a0
        8'h18: instr = 32'h10a7000a;    // beq $a1, $a1, $a0
        8'h1c: instr = 32'h0064202a;    // slt $a0, $v1, $a0
        8'h20: instr = 32'h10800001;    // beq $a0, $0, 0x0001
        8'h24: instr = 32'h20050000;    // addi $a1, $0, 0x0000
        8'h28: instr = 32'h00e2202a;    // slt $a0, $a3, $v0
        8'h2c: instr = 32'h00853820;    // add $a3, $a0, $a1
        8'h30: instr = 32'h00e23822;    // sub $a3, $a3, $v0
        8'h34: instr = 32'hac670044;    // sw $a3, 0x0044, $v1
        8'h38: instr = 32'h8c020050;    // lw $v0, 0x0050, $0
        8'h3c: instr = 32'h20040003;    // addi $a0, $0, 3
        8'h40: instr = 32'h20020001;    // addi $v0, $0, 0x0001
        8'h44: instr = 32'hac020054;    // sw $v0, 0x0054, $0
        default:  instr = {32{1'bx}};     // unknown address
        endcase
endmodule
```

Question 4:

- **Compute-use hazard**

HAZARD because $t0 does not written the value 5 yet

addi $t0, $zero, 5	8'h00: 32'h20080005;
addi $t1, $t0, 6	8'h04: 32'h21090006;
add $t2, $t1, $t0	8'h08: 32'h01285020;

$t1 does not written back from the previous step

- Load-use hazard

Data does not loaded from Memory to $t1 yet. Wrong data will fetched from $t1

addi $t0, $zero, 5	8'h00: 32'h20080005;
addi $t1, $zero, 6	8'h04: 32'h20090006;
addi $a0, $zero, 1	8'h08: 32'h20040001;
addi $a1, $zero, 2	8'h0c: 32'h20050002;
sw $t0, 0($t1)	8'h10: 32'had280000;
lw $t1, 1($t0)	8'h14: 32'h8d090001;
add $t2, $t1, $a0	8'h18: 32'h01245020;
sub $t2, $t1, $a1	8'h1c: 32'h01255022;

- **Branch hazard**

These instructions going to fetched as a hazard because of branch misprediction. The pipeline don't know branch decision untill the Memory stage

addi $t1, $zero, 2	8'h00: 32'h20090002;
beq $zero, $zero, 2	8'h04: 32'h10000002;
addi $t1, $zero, 5	8'h08: 32'h20090005;
addi $t1, $t1, 6	8'h0c: 32'h21290006;
addi $t1, $zero, 8	8'h10: 32'h20090008;
addi $a0, $zero, 0	8'h14: 32'h20040000;
addi $a1, $zero, 0	8'h18: 32'h20050000;
sw $t1, 0($zero)	8'h1c: 32'hac090000;