Bilkent University

Department:CS

CS224

Preliminary Design Report

Lab 5

Section 6

Fazlı Güdül

22002785

14/04/2024

b)Hazard Types and Their Properties

1) Compute - Use (Data Hazard)

Stages Affected: Compute - Use hazard occurs when a changed data in the previous instruction is used in the current instruction. Therefore, it leads to reading the register value incorrectly. At first it mostly affects the Decode stage, since the value of register is not correctly read. All the following stages (Execute, Memory, Writeback) are also affected since all operations are performed according to the wrong data read at the beginning.

Solution: Forwarding is employed to ensure correct data transfer between pipeline stages. By forwarding data from the Memory stage directly to the Execute stage, the hazard is addressed, especially when two consecutive instructions exhibit this hazard. Additionally, forwarding from the WriteBack stage to the Execute stage can resolve the hazard when an instruction intervenes between the two affected instructions. Stalling serves as an alternative solution, enabling the Decode stage to process after the preceding instruction writes into the register. Thus, a combination of forwarding and stalling effectively solves this hazard.

2) Load - Use (Data Hazard)

Stages Affected:This hazard arises when the data loaded into the register cannot be updated when the subsequent instruction begins execution. Because the lw (load word) instruction reads the data at the Memory stage's conclusion, it introduces a two-cycle delay before the received data can be utilized in the following instruction. This latency primarily impacts the Execute stage, as the data inputted into the ALU (Arithmetic Logic Unit) is incorrectly selected due to this delay. Moreover, depending on the type of instruction, the Memory or WriteBack stages may also be impacted.

Solution:When the data needed for the next instruction isn't loaded yet in the previous one, forwarding alone isn't sufficient. Because the data read from memory needs to be available earlier, stalling is necessary. This involves aligning the start of the Execute stage of the current instruction with the end of the Memory stage of the previous one. Additionally, to use this data before it's written into the register, forwarding from the WriteBack stage to the Execute stage is required. Moreover, in certain situations where there's an instruction in between, stalling alone can be enough since data is read before it's written. Hence, Forwarding and Stalling serve as solutions for this hazard, either separately or combined depending on the circumstances.

3) Load - Store (Data Hazard)

Stages Affected: This hazard occurs when the data loaded into the register is intended to be stored in memory in a subsequent instruction. This causes incorrect data being read in the Decode stage of the next instruction. Since the two-cycle latency in the lw instruction, false register value is read. So, incorrect data is passed into the subsequent stages, indirectly impacting all pipeline stages. Since the aim of the current instruction is to store, the Memory stage is mostly affected.

Solution: This hazard occurs since the register value is not uploaded yet at the lw instruction before it is read in the following sw instruction. Stalling can be used to solve. It would allow writing in the register before reading from it in the next instruction. By stalling the store instruction twice, problem would be solved. In addition, there might be an instruction in between lw and sw. In that condition, only one stall would be enough. So, the correct value would be stored in the subsequent sw instruction. So, stalling would be an suitable solution for the load store hazard.

4) J-type jump (Control Hazard)

Stages Affected: This hazard primarily affects the Fetch and Decode stages of the pipeline.The Fetch stage is affected since the correct instruction address for the next cycle might not be known due to the jump instruction, leading to fetching the wrong instructions. The Decode stage is also affected since it may decode incorrectly the instruction fetched from the wrong address.

Solution: To solve this hazard, Branch Prediction commonly used. anticipate the target address of the jump instruction, enabling the early retrieval of correct instructions. If the prediction aligns with the actual flow, the pipeline operates smoothly. However, if the prediction is wrong, flushing is necessary. Flushing involves discarding the mistakenly fetched instructions and fetching from the correct address instead. So, for this hazard, Branch Prediction is used and there is a false prediction exists, flushing will be used to ensure the pipeline continues with the correct instruction sequence.

5) Branch (Control Hazard)

Stages Affected: Since the decision for branch is done in the Decode stage of this pipeline, there occurs one cycle delay before determining for the next instruction to be executed. Therefore, it causes more instruction to be fetched. This mainly affects the Fetch stage of the pipeline.

Solution:Branch decisions are made in the Decode stage, prompting the next instruction to be fetched. If a branch occurs, the current instruction is flushed out. Additionally, if the branch instruction compares registers that might have been altered by previous instructions, forwarding is used to verify the comparison. Furthermore, if the comparison happens in the Execute stage, stalling is needed to update the register value. Thus, Flushing, Forwarding, and Stalling work together to overcome branch hazards.

c)Logic Equations for Outputs of Hazard Unit

a) Logic Equations for ForwardAE Signal

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

b) Logic Equations for ForwardBE Signal

if ((rtE != 0) AND (rtE == WriteRegM) AND RegWriteM) then ForwardBE = 10

else if ((rtE != 0) AND (rtE == WriteRegW) AND RegWriteW) then ForwardBE = 01

else
ForwardBE = 00

c) Logic Equations for ForwardAD Signal

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

d) Logic Equations for ForwardBD Signal

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

e) Logic Equations for StallF, StallD and FlushE Signals

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

// 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.

// However, you can change them, if you want.

// 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 clock, posedge reset) begin
      if (reset) begin
         instructionD <= 0;
         PCPlus4D <= 0;
      end
      else if(enable) begin
         instructionD <= instruction;
         PCPlus4D <= PCPlus4F;
      end
      else begin
         instructionD <= instructionD;
```

```
          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,                    // 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] PC,
                input logic EN_DtoE, clk_DtoE,
// StallE will be connected as this EN
            input logic [7:0] opcode_DtoE,         // Control signal
            output logic [31:0] PCE);

   always_ff @(posedge clk_DtoE)
      if (EN_DtoE)
         begin
            // Implement the functionality of PipeDtoE here
            // Example:
            PCE <= PC + 4; // Increment PC by 4 for next instruction
         end

endmodule

module PipeEtoM(input logic [31:0] PCE,
            input logic EN_EtoM, clk_EtoM,                // StallM will be connected as this
EN
            input logic [7:0] opcode_EtoM,         // Control signal
            output logic [31:0] PCM);

   always_ff @(posedge clk_EtoM)
      if (EN_EtoM)
         begin
            // Implement the functionality of PipeEtoM here
            // Example:
            PCM <= PCE + 4; // Increment PCE by 4 for next instruction
         end
```

```
endmodule

module PipeMtoW(input logic [31:0] PCM,
            input logic EN_MtoW, clk_MtoW,              // StallW will be connected as this
EN
            input logic [7:0] opcode_MtoW,              // Control signal
            output logic [31:0] PCW);


   always_ff @(posedge clk_MtoW)
      if (EN_MtoW)
         begin
            // Implement the functionality of PipeMtoW here
            // Example:
            PCW <= PCM + 4; // Increment PCM by 4 for next instruction
         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, reset,
    input  logic [2:0] ALUControlD,
    input  logic BranchD,
    input  logic [4:0] rsD, rtD, rdD,
    input  logic [15:0] immD,
    // Change input-outputs according to your datapath design
    output logic RegWriteE, MemToRegE, MemWriteE,
    output logic [31:0] ALUOutE, WriteDataE,
    output logic [4:0] WriteRegE,
    output logic [31:0] PCBranchE,
    output logic pcSrcE
);

    // Define the wires needed inside this pipelined datapath module
    logic stallF, stallD, ForwardAD, ForwardBD, FlushE, ForwardAE, ForwardBE;
    logic PcSrcD, MemToRegW, RegWriteW;
    logic [31:0] PC, PCF, instrF, instrD, PcSrcA, PcSrcB, PcPlus4F, PcPlus4D;
    logic [31:0] PcBranchD, ALUOutW, ReadDataW, ResultW, RD1, RD2;
    logic [4:0] WriteRegW;
    // Instantiate the required modules below in the order of the datapath flow


    // Connections for the writeback stage and the fetch stage are written for you.

    // You can change them if you want.


    // Writeback stage pipe

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


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


    // Fetch stage pipe

    assign PcPlus4F = PCF + 4; // Here PCF is from the fetch stage

    mux2 #(32) pc_mux(PcPlus4F, PcBranchD, PcSrcD, PC); // Here PcBranchD is from the
decode stage


    // Note that normally the whole PCF should be driven to

    // the instruction memory. However, for our instruction

    // memory, this is not necessary.

    imem im1(PCF[7:2], instrF); // Instantiated instruction memory


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


    regfile rf(clk, RegWriteW, instrD[25:21], instrD[20:16],

             WriteRegW, ResultW, RD1, RD2); // Register file


    // Adder that adds 4 to PC

    assign PcBranchD = instrD[15:0]; // Branch address from the decode stage

    assign PCBranchE = PcBranchD; // Output of PCBranchE
    // Shifter that shifts SignImmE to the left by 2

    assign ALUOutE = {immD, 2'b00}; // Output of ALUOutE

    assign WriteDataE = RD2; // Output of WriteDataE

    assign WriteRegE = instrD[20:16]; // Output of WriteRegE

    assign RegWriteE = RegWriteW; // Output of RegWriteE

    assign MemToRegE = MemToRegW; // Output of MemToRegE

    assign MemWriteE = MemWriteE; // Output of MemWriteE

    assign pcSrcE = PcSrcD; // Output of pcSrcE


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

);


    always_comb begin
```

```
        // Forwarding logic
        if (RegWriteM && (WriteRegM != 0) && (WriteRegM == rsE)) begin
          ForwardAE = 3'b010; // Forward data from MEM stage
        end else if (RegWriteM && (WriteRegM != 0) && (WriteRegM == rtE)) begin
          ForwardBE = 3'b010; // Forward data from MEM stage
        end else if (RegWriteW && (WriteRegW != 0) && (WriteRegW == rsE)) begin
          ForwardAE = 3'b001; // Forward data from WB stage
        end else if (RegWriteW && (WriteRegW != 0) && (WriteRegW == rtE)) begin
          ForwardBE = 3'b001; // Forward data from WB stage
        end else begin
          ForwardAE = 3'b000; // No forwarding from MEM or WB
          ForwardBE = 3'b000;
        end

        // Hazard detection
        if ((RegWriteE && (WriteRegE != 0) && ((WriteRegE == rsD) || (WriteRegE == rtD))) ||
          (MemToRegE && (WriteRegM != 0) && ((WriteRegM == rsD) || (WriteRegM ==
rtD)))) begin
          StallD = 1'b1; // Stall decode stage
          StallF = 1'b1; // Stall fetch stage
          FlushE = 1'b1; // Flush execute stage
        end else begin
          StallD = 1'b0; // No stall in decode stage
          StallF = 1'b0; // No stall in fetch stage
          FlushE = 1'b0; // No flush in execute stage
        end
      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;

   // Hazard Unit instantiation
   HazardUnit hazard_unit(
      .RegWriteW(RegWriteW),
      .WriteRegW(WriteRegW),
      .RegWriteM(RegWriteM),
      .MemToRegM(MemToRegM),
      .WriteRegM(WriteRegM),
      .RegWriteE(RegWriteE),
      .MemToRegE(MemToRegE),
      .WriteRegE(WriteRegE),
      .rsE(rsE),
      .rtE(rtE),
      .rsD(rsD),
      .rtD(rtD),
      .ForwardAE(ForwardAE),
      .ForwardBE(ForwardBE),
      .FlushE(FlushE),
      .StallD(StallD),
      .StallF(StallF)
   );

   // Datapath instantiation
   datapath dp_inst(
      .clk(clk),
      .reset(reset),
      .ALUControlD(alucontrol),
      .BranchD(jump),
      .rsD(rsD),
      .rtD(rtD),
      .rdD(0),
      .immD(0),
      .RegWriteE(regwrite),
      .MemToRegE(memtoreg),
      .MemWriteE(memwrite),
      .ALUOutE(aluout),
      .WriteDataE(resultW),
      .WriteRegE(0),
      .PCBranchE(pcsrc),
      .pcSrcE(pcsrc)
   );
```

```
    // Add connections as needed


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

1) No Hazards

Location (hex)	Machine Instruction (hex)	Assembly Language Equivalent
0	20020005	addi $v0, $zero, 0x0005
4	2003000C	addi $v1, $zero, 0x000C
8	2067FFF7	addi $a3, $v1, 0xFFF7
0C	0E+00	or $a0, $a3, $v0
10	642824	and $a1, $v1, $a0
14	00A42820	add $a1, $a1, $a0
18	10A7000A	beq $a1, $a3, 0x000A
1C	0064202A	slt $a0, $v1, $a0
20	20050000	addi $a1, $zero, 0x0000
24	00E2202A	slt $a0, $a3, $v0
28	853820	add $a3, $a0, $a1
2C	0E+00	sub $a3, $a3, $v0
30	AC670044	sw $a3, 0x0044($v1)
34	8C020050	lw $v0, 0x0050($zero)

2) Compute - Use Hazard

3) Load - Use Hazard

Location (hex)	Machine Instruction (hex)	Assembly Language Equivalent
0	20120005	addi $s2, $zero, 5
4	22500006	addi $s0, $s2, 6
8	2124022	sub $t0, $s0, $s2

Location (hex)	Machine Instruction (hex)	Assembly Language Equivalent
0	20100005	addi $s0, $zero, 5
4	20120006	addi $s2, $zero, 6
8	20110008	addi $s1, $zero, 8
0C	AC100044	sw $s0, 0x0044 ($zero)
10	8C120044	lw $s2, 0x0044 ($zero)
14	129820	add $s3, $zero, $s2

4) Load - Store Hazard

0	20100005	addi $s0, $zero, 5
4	20120006	addi $s2, $zero, 6
8	20110008	addi $s1, $zero, 8
0C	AC100044	sw $s0, 0x0044 ($zero)
10	8C120044	lw $s2, 0x0044 ($zero)
14	AC120020	sw $s2, 0x0020 ($zero)

5)J-Type Jump Hazard

Location (hex)	Machine Instruction (hex)	Assembly Language Equivalent
0	20100005	addi $s0, $zero, 5
4	20110008	addi $s1, $zero, 8
8	8000010	j jump_location
0C	2119020	add $s2, $s0, $s1
10		jump_location:
14	20110010	addi $s1, $zero, 10
18	AC120044	sw $s2, 0x0044($zero)
1C	8C100050	lw $s0, 0x0050($zero)

6) Branch Hazard

Location (hex)	Machine Instruction (hex)	Assembly Language Equivalent
0	20100005	addi $s0, $zero, 5
4	10000001	beq $zero, $zero, 1
8	20100008	addi $s0, $zero, 8
0C	2538820	add $s1, $s2, $s3
10	2939025	or $s2, $s4, $s3
14	22110004	addi $s1, $s0, 4