Bilkent University

Computer Science Department

CS224 – COMPUTER ORGANIZATION
Preliminary Design Report

Lab 5

Section 4

Seçkin Alp Kargı

22001942

24/11/2023

B) List of Hazards

Name of Hazard	Type of Hazard	Affected Stages	Solutions
compute-use	Data hazard	Instruction Decode Execute (Memory and WriteBack when wrong values come)	Forwarding Stalling
load-use	Data Hazard	Instruction Decode Execute Memory	Stalling
load-store	Data Hazard	Memory Execute	Stalling
J-type jump	Control Hazard	Instruction Fetch Instruction Decode	Inserting Nop Stalling
branch	Control Hazard	Instruction Fetch Instruction Decode	Stalling 3 cycles, Branch Prediction

Explanations, What, When, How:

Compute-use:

What is Compute-Use:

It is a particular form of data hazard that arises when a computation instruction relies on the outcome of a preceding instruction, and dependent instruction attempts to utilize the result before it has become accessible.

When it is Happening:

In R-type instructions for instance sub $t1, $t2, $t3, add $t2, $t4, $t1 in here before the changes done for $t1 in sub instruction, add instruction tries to use $t1.

How it is Happening:

For example, if we have two instructions that can cause compute use, first IF stage fetch than ID and RF starts at the same time second IF stage fetch. Then first execution starts for example sub instruction performs the subtraction, at the same time second IF and Rf stage happening. For add reads $t4, and $t1, in here we need $t1 is value stored in previous sub instruction. In execution 2 add instruction tries the add the values $t4 and $t1 but register 4 sub $t1, $t2, $t3, add $t2, $t4, $t1 is not updated yet.

Possible Solutions:

Forwarding: If the result of the sub is available in the execute stage, it can be forwarded directly to the decode stage, allowing the add to use the most recent value. It increases the performance.

Stalling: The pipeline can insert a stall or bubble; it will delay the execution of the add until the sub instruction's result is available in the write back stage. It decreases the performance.

Load-use:

What is Load-Use:

It is a particular form of data hazard that arises when one instruction to load data is reliant on the outcome of an earlier instruction to load data, and the dependent instruction attempts to utilize the loaded data before it is ready.

When it is Happening:

It happens after the using of lw (lw $s0, 0($s1)) following instruction tries to use source register (sub $s2, $s0, $s4).

How it is Happening:

For example, if we have two instructions that can cause load use, second instruction need to access data from $s0 that modifying in the first instruction for that first instruction needs to complete memory stage. But it did not so it happens in this situation. Second instruction unable to access.

Possible Solutions:

Stalling: The pipeline can insert a stall or bubble, The dependent instruction is not executed until the data from the load instruction is accessible, due to the stall. The pipeline may wait for the necessary data during the stall as no new instructions are retrieved or executed during that time. It decreases the performance.

Load-store:

What is Load-Store:

It is a particular form of data hazard that arises when an instruction to load data attempts to use stored data before it has been written to memory because it depends on the outcome of an earlier operation to store data.

When it is Happening:

It happens if we use sw and lw that has same rt immediately. -> sw $s0, 0($s1), lw $s0, 0($s2)

How it is Happening:

Writing the data to memory upon encountering a store instruction takes time. A dependent load instruction may try to use the data before it has been stored, leading to an inaccurate computation, if it is scheduled to run before the store instruction has finished storing the data.

Possible Solutions:

Stalling: The pipeline can insert a stall or bubble; it will delay the execution of the sw until the lw instruction’s result is available in the write back stage. It decreases the performance.

Branch:

What is Branch:

It is control hazard that arises when the next instructions stars before we finished the new branch address's location.

When it is Happening:

If we use branch instruction it happens.

How it is Happening:

For example, beq $s0,$s2, Next. Normally if we do not have branch, it will go simply next instruction but we have branch and we do not know where we go to before the second instruction's starting to fetch.

Possible Solutions:

Stalling: The pipeline can insert a stall or bubble; we can do this for 3 cycles.

Branch Predicting: We can try to predict the branch. We can do this by using three instructions. If we branch instruction exists other instruction will be cancelled. We can do branch decision in earlier.

j-type jump:

What is J-type Jump:

In MIPS assembly, the J-type jump instruction is an unconditional jump that moves execution to a different memory address, allowing the program to alter its control flow. It causes an instantaneous leap to the designated target address and has no conditions or dependencies.

When it is Happening:

When we use J type instruction for example J Next.

How it is Happening:

Next pipeline starts when j instruction is not finished

Possible Solutions:

Inserting Nop: It will fill the pipeline until we completed to jump instruction.

Stalling: The pipeline can insert a stall or bubble; it will delay the execution of the next instruction until the j instruction's result is available in the write back stage. It decreases the performance.

C) Logic Equations

Input Signals: BranchD, RsD, RtD, RsE, RtE, WriteRegE4:0, MemtoRegE, RegWriteE, WriteRegM4:0, MemtoRegM, RegWriteM, WriteRegW4:0, RegWriteW

BranchStall = (MemtoRegM AND BranchD AND (WriteRegM == rtD OR WriteRegM == rsD) OR (BranchD AND (WriteRegE == rsD OR WriteRegE = rtD) AND RegWriteE)

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

Stalling:

StallF: StallF = (branchstall OR lwstall)

StallD: StallD = (branchstall OR lwstall)

Flushing:

FlushE: FlushE = (branchstall OR lwstall)

Forwarding:

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

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

ForwardAE:

```
If ( RegWriteM AND (rsE != 0) AND ( rsE == WriteRegW))
    ForwardAE = 10
else if ( RegWriteW AND (rsE != 0) AND ( rsE == WriteRegW))
    ForwardAE = 01
Else
    ForwardAE = 00
```

ForwardBE:

```
If ( RegWriteM AND (rtE != 0) AND ( rtE == WriteRegW))
    ForwardBE = 10
else if ( RegWriteW AND (rtE != 0) AND ( rtE == WriteRegW))
    ForwardBE = 01
Else
    ForwardBE = 00
```

D) Fulfilled Pipelined MIPS Processor (Red Color for changes)

```
module PipeFtoD -> no change!

module PipeWtoF -> no change!

module PipeDtoE(input logic RegWriteD,MemtoRegD,MemWriteD,ALUSrcD,RegDstD,BranchD,clk,FlushE,reset
                input logic [2:0] ALUControlD,
                input logic [4:0] RsD, RtD, RdE,
                input logic [31:0] SignImmD, RD1, RD2,
                output logic RegWriteE, MemtoRegE, MemWriteE, ALUSrcE, RegDstE,
                output logic [2:0] ALUControlE,
                output logic [4:0] RsE, RtE, RdE,
                output logic [31:0] SignImmE, RD1out,RD2out );
endmodule
module PipeEtoM(input logic clk, MemWriteE, RegWriteE, MemtoRegE, reset,
                input logic [4:0] WriteDataE,
                input logic [31:0] ALUoute, WriteDataE,
                output logic RegWriteM, MemtoRegM, MemWriteM,
                output logic [4:0] WriteRegM
                output logic [31:0] AluOutM, WriteDataM );

endmodule


module PipeMtoW(input logic clk, RegWriteM, MemtoRegM, reset
                input logic [4:0] WriteRegM,
                input logic [31:0] RD, ALUOutM,
                output logic RegWriteW, MemtoRegW,
                output logic [4:0] WriteRegW,
                output logic [31:0] ReadDataW, ALUOutW );

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,
                        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] PC, PCF, PCBranchE,
                       output logic pcSrcE);
           logic stallF, stallD,  ForwardAD, ForwardBD,  FlushE, ForwardAE, ForwardBE;// Wires for connecting Hazard Unit
           logic [31:0] PcBranchD, PcPlus4F, PcSrcA, PcSrcB, ALUOutM, WriteDataM, ReadDataW, ALUOutW, PcPlus4D,
SignImmD, SignImmDShifted, ResultW, SignImmE, SrcAE, SrcBE, ReadData,RD1Out,RD2Out,mx2Out,mx3Out;
           logic [4:0] WriteRegW, WriteRegE, WriteRegM, rsE, rtE, rdE, rsD, rtD, rdD;
           logic[2:0] ALUControlE;
           logic MemWriteM, MemtoRegW,  ALUSrcE, RegDstE, RegWriteW;
           mux2 #32 mx1(PCPlus4F,PCBranchD,PCSrcD,PC');
           PipeWtoF pipe1(PC',StallF,clk,PCF);
           adder add1(PCF, 32'b04,PCPlus4F);
           PipeFtoD pipe2(RD,PCPlus4F,StallD,clk,InstrD,PCPlus4D);
           Imem imem1(PCF,RD);
           signext sign1(InstrD[15:0], SignImmD);
           sl2 sl21(SignImmD, SignShiftedOut);
           regfile     rf (clk, regwrite, instr[25:21], instr[20:16], writeregW,result, srca, writedata);
           adder add2(SignShifted, PCPlus4D, PCBranchD);
           mux2 #32 mx2(RD1,ALUOutM,mx2Out);
           mux2 #32 mx3(RD2,ALUOutM,mx3Out);
           equal equal1(mx2Out,mx3Out,EqualD);
           and2 and21(BranchD, EqualD, PCSrcD);
```

```
        PipeDtoE pipe3(RegWriteD,MemtoRegD,MemWriteD, ALUSrcD, RegDstD, BranchD, clk, FlushE, reset,
        ALUControlD, RsD, RtD, RdE, SignImmD, RD1, RD2, RegWriteE, MemtoRegE, MemWriteE, ALUSrcE, RegDstE,
        ALUControlE, RsE, RtE, RdE,  SignImmE, RD1out,RD2out);

        mux2 #4 mx4(RtE,RdE,WriteRegE);

        mux4 #32 mx5(RD1out,ResultW,ALUOutM,32'b0,SrcAE);

        mux4 #32 mx6(RD2out,ResultW,ALUOutM,32'b0,WriteDataE);

        mux2 #32 mx7(WriteDataE,SignImme,SrcBE);

        alu alu1(SrcAE, SrcBE,ALUControlE,ALUOute);

        PipeEtoM pipe4(clk, MemWriteE, RegWriteE, MemtoRegE, reset, WriteDataE, ALUoute, WriteDataE, RegWriteM,
        MemtoRegM, MemWriteM, WriteRegM,AluOutM, WriteDataM );

        PipeMtoW pipe5(clk, RegWriteM, MemtoRegM, reset, WriteRegM , ALUOutM, RegWriteW, MemtoRegW,
        WriteRegW, ReadDataW, ALUOutW);

        mux2 #32 mx7(ReadDataW, ALUOutW, MemtoRegW,ResultW);

        HazardUnite hazardunit1(RegWriteW, WriteRegW, RegWriteM,MemToRegM, WriteRegM, RegWriteE,
        MemToRegE,  rsE, rtE, rsD,rtD, ForwardAE,ForwardBE, ,StallD,StallF);

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

  );

        Logic branchStall,lwStall;

  always_comb begin

    BranchStall = (MemtoRegM & BranchD & (WriteRegM == rtD | WriteRegM == rsD) | (BranchD & (WriteRegE == rsD |
WriteRegE = rtD) & RegWriteE);

lwStall = MemtoRegE & ((rtD == rtE) | (rtE == rsD));

ForwardAD = RegWriteM & (rsD  != 0)  & (rsD == WriteRegM);

ForwardBD = RegWriteM & (rtD  != 0)  & (rtD == WriteRegM);

StallD <= lwstall | branchstall;
```

```
StallF <= lwstall | branchstall;

FlushE <= lwstall | branchstall;

If ( RegWriteM & (rsE != 0) & ( rsE == WriteRegW)) begin

              ForwardAE = 2b’10; end

       else if ( RegWriteW  & (rsE != 0) & ( rsE == WriteRegW)) begin

              ForwardAE =2b’01; end

       Else

              ForwardAE = 2b’00;

If ( RegWriteM & (rtE != 0) & ( rtE == WriteRegW)) begin

              ForwardBE = 2b’10; end

       else if ( RegWriteW & (rtE != 0) & ( rtE == WriteRegW)) begin

              ForwardBE = 2b’01; end

       Else

              ForwardBE = 2b’00;

  End

If ( RegWriteM & (rdE != 0) & ( rdE == WriteRegW)) begin

              ForwardCE = 2b’10; end

       else if ( RegWriteW & (rdE != 0) & ( rdE == WriteRegW)) begin

              ForwardCE = 2b’01; end

       Else

              ForwardCE = 2b’00;

  end

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 controller1(input  logic[5:0] op, funct, memtoreg, memwrite, alusrc, regdst, regwrite, jump, alucontrol,
branch);
datapath datapath1(clk, reset, RegWriteW, ALUControlD, BranchD, pcPlus4D, ResultW, rsD,rtD,rdD, immD,
WriteRegW,,RegWriteE, MemToRegE,MemWriteE,  ALUOutE, WriteDataE, WriteRegE, PC, PCF, PCBranchE, pcSrcE);
);
endmodule
module imem ( input logic [5:0] addr, output logic [31:0] instr);
                        8'h00: instr = 32'h20100019;    ->  addi $t4, $zero, 25
            8'h04: instr = 32'h 2015003C;    -> addi $t5, $zero, 60
            8'h08: instr = 32'h 018C8825;    -> or $s3, $t5, $
            8'h0c: instr = 32' 218FFFF8;    -> addi $t7, $t4, -24
            8'h10: instr = 32'h 01F78020;    -> add $s5, $t4, $t7
            8'h14: t4instr = 32'h 2016001E;    -> addi $t6, $zero, 30
            8'h18: instr = 32'h 01AE9024;    -> and $s4, $t5, $t6
            8'h1c: instr = 32'h 01D6202A;    -> slt $a0, $t4, $t6
            8'h20: instr = 32'h 01AD8822;    ->  sub $a2, $s4, $ t5
            8'h24: instr = 32'h 02948820;    ->  add $a1, $s3, $s4
            8'h28: instr = 32' AE8D0014;    -> sw $s4, 0x14($t5)
            8'h2c: instr = 32' 8C020020;    -> lw $v0, 0x20($zero)
                default:  instr = {32{1'bx}};      // unknown address
              endcase
endmodule


endmodule
```

New Modules:

```
module equal(
  input wire [3:0] a,
  input wire [3:0] b,
  output wire out
);
  assign out = (a == b);
endmodule
```

```
module and2(
  input wire a,
  input wire b,
  output wire out
);
  assign out = a & b;
endmodule
module Mux4(
  input wire [3:0] data,
  input wire [1:0] select,
  output wire out
);
  assign out = (select == 2'b00) ? data[0] :
               (select == 2'b01) ? data[1] :
               (select == 2'b10) ? data[2] :
               (select == 2'b11) ? data[3] :
               1'bz;
Endmodule
```

Other Modules Does Not Need Changes.

E) Test Programs

Test Code of No Hazard:

```
8'h00: instr = 32'h20100019;    ->  addi $t4, $zero, 25

8'h04: instr = 32'h 2015003C;    -> addi $t5, $zero, 60

8'h08: instr = 32'h 018C8825;    -> or $s3, $t5, $

8'h0c: instr = 32' 218FFFF8;    -> addi $t7, $t4, -24

8'h10: instr = 32'h 01F78020;    -> add $s5, $t4, $t7

8'h14: t4instr = 32'h 2016001E;    -> addi $t6, $zero, 30

8'h18: instr = 32'h 01AE9024;    -> and $s4, $t5, $t6

8'h1c: instr = 32'h 01D6202A;    -> slt $a0, $t4, $t6

8'h20: instr = 32'h 01AD8822;    ->  sub $a2, $s4, $ t5

8'h24: instr = 32'h 02948820;    ->  add $a1, $s3, $s4

8'h28: instr = 32' AE8D0014;    -> sw $s4, 0x14($t5)

8'h2c: instr = 32' 8C020020;    -> lw $v0, 0x20($zero)
```

Test Code of Compute-Use Hazard:

```
8'h00: instr = 32' 200E0004;    -> addi $t6, $zero, 4

8'h04: instr = 32'h 200D000E;    -> addi $t5, $zero, 14

8'h08: instr = 32'h 01AE8825;    -> or $t6, $t5, $t6

8'h0c: instr = 32'h 200C000A;    ->  addi $t4, $zero, 10

8'h10: instr =32'h 0231902A;    -> slt $s6, $t6, $s1

8'h14: instr = 32'h 02109020;    -> add $s5, $t4, $s0

8'h18: instr = 32'h 01CD8824;    -> and $s4, $t6, $t5

8'h1c: instr = 32'h 02AE8822;    ->  sub $s7, $s5, $s6

8'h20: instr = 32'h 02A5A020;    -> add $s5, $s5, $s5

8'h24: instr = 32'h AEAD0014;    ->  sw $s7, 0x14($t5)

8'h28: instr = 32' 8C020020;    -> lw $v0, 0x20($zero)
```

Test Code of Load-Use Hazard:

```
8'h00: instr = 32'h 200D0005;    ->  addi $t5, $zero, 5
8'h04: instr = 32'h AC0E0050;    -> sw $t6, 0x50($zero)
8'h08: instr = 32'h 8C0F0050;    -> lw $t7, 0x50($zero)
8'h0c: instr = 32' 21CF0004;    -> addi $t7, $t6, 4
8'h10: instr =32'h 97080048;    -> lw $t8, 0x48($t8)
8'h14: instr = 32'h 21AD0003;    -> addi $t8, $t5, 3
8'h18: instr = 32'h 01F82022;    -> sub $t7, $t7, $t8
8'h1c: instr = 32'h AC0F0085;    -> sw $t7, 0x85($zero)
8'h20: instr = 32'h 8C0D0085;    ->  lw $t5, 0x85($zero)
8'h24: instr = 32'h 8DCF007C;    ->  lw $t6, 0x7c($t5)
8'h28: instr = 32' 01AE4820;    -> add $t6, $t5, $t6
8'h2c: instr = 32' 200C0003;    -> add $t4, $zero, 3
```

Test Code of Branch Hazards:

```
8'h00: instr = 32'h 200C000A;    ->  addi $t4, $zero, 10
8'h04: instr = 32'h 200D0006;    -> addi $t5, $zero, 6
8'h08: instr = 32' 018C4820;    -> add $t6, $t5, $t4
8'h0c: instr = 32'h 198D0004;    -> beq $t4, $t5, 4
8'h10: instr = 32'h 018D4822;    -> sub $t4, $t4, $t5
8'h14: instr = 32'h 19AC0004;    -> beq $t5, $t4, 4
8'h18: instr = 32'h 25ADFFFE;    -> addi $t5, $t5, -2
8'h1c: instr = 32'h AC0E0030;    -> sw $t6, 0x30($zero)
8'h20: instr = 32'h 018C482A;    ->  slt $t5, $t5, $t4
8'h24: instr = 32'h 018C4825;    -> or $t4, $t5, $t4
8'h28: instr = 32' 15A00004;    -> beq $t5, $zero, 4
8'h2c: instr = 32' 018C4824;    -> and $t4, $t5, $t4
8'h30: instr = 32' 8C0F0030;    -> lw $t7, 0x30($zero)
8'h34: instr = 32' 198CFFFE;    -> beq $t4, $t4, -2
```