CS224

Lab 05

Section 003

Furkan Mert Aksakal

22003191

27.11.2024

b-)

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Affected Stages** | **Solution** |
| Data Hazard | Compute-Use | EX🡪EX, EX🡪MEM | Forwarding |
| Data Hazard | Load-Use | MEM🡪EX | Stall + Forwarding |
| Data Hazard | Load-Store | MEM🡪MEM | Forwarding |
| Control Hazard | Branch | ID | Flush |
| Control Hazard | J-Type Jump | ID | Flush |
| Data + Control Hazard | Control – Data Combination | ID | Forwarding + Possible Stall |
| Data Hazard | Multiple Compute-Use | EX🡪EX, MEM🡪EX | Forwarding with Priority |

Explanation:  
1) Occurs when an instruction needs a register value that's being computed by a previous ALU instruction

2) Occurs when an instruction needs a register value that's being loaded by a previous load instruction

3) Occurs when a store instruction needs data that's being loaded by a previous instruction

4) Occurs when a branch condition is evaluated

5) Occurs with unconditional jumps (j instruction)

6) Occurs when branch/jump needs a register value that's being computed

7) Occurs when an instruction has multiple RAW dependencies

c-)

1. **ForwardA**

ForwardA =

if (EX/MEM.RegWrite

&& (EX/MEM.RegisterRd ≠ 0)

&& (EX/MEM.RegisterRd == ID/EX.RegisterRs)) then "10"

else if (MEM/WB.RegWrite

&& (MEM/WB.RegisterRd ≠ 0)

&& (MEM/WB.RegisterRd == ID/EX.RegisterRs)

&& !(EX/MEM.RegWrite && (EX/MEM.RegisterRd ≠ 0)

&& (EX/MEM.RegisterRd == ID/EX.RegisterRs))) then "01"

else "00"

1. **ForwardB**

ForwardB =

if (EX/MEM.RegWrite

&& (EX/MEM.RegisterRd ≠ 0)

&& (EX/MEM.RegisterRd == ID/EX.RegisterRt)) then "10"

else if (MEM/WB.RegWrite

&& (MEM/WB.RegisterRd ≠ 0)

&& (MEM/WB.RegisterRd == ID/EX.RegisterRt)

&& !(EX/MEM.RegWrite && (EX/MEM.RegisterRd ≠ 0)

&& (EX/MEM.RegisterRd == ID/EX.RegisterRt))) then "01"

else "00"

1. **ForwardBranch**

ForwardBranchA =

if (ID/EX.RegWrite

&& (ID/EX.RegisterRd ≠ 0)

&& (ID/EX.RegisterRd == IF/ID.RegisterRs)) then "10"

else if (EX/MEM.RegWrite

&& (EX/MEM.RegisterRd ≠ 0)

&& (EX/MEM.RegisterRd == IF/ID.RegisterRs)) then "01"

else "00"

ForwardBranchB =

if (ID/EX.RegWrite

&& (ID/EX.RegisterRd ≠ 0)

&& (ID/EX.RegisterRd == IF/ID.RegisterRt)) then "10"

else if (EX/MEM.RegWrite

&& (EX/MEM.RegisterRd ≠ 0)

&& (EX/MEM.RegisterRd == IF/ID.RegisterRt)) then "01"

else "00"

1. **Stall**

LoadStall =

ID/EX.MemRead &&

((ID/EX.RegisterRd == IF/ID.RegisterRs) ||

(ID/EX.RegisterRd == IF/ID.RegisterRt))

BranchStall =

(IF/ID.Branch && ID/EX.RegWrite &&

((ID/EX.RegisterRd == IF/ID.RegisterRs) ||

(ID/EX.RegisterRd == IF/ID.RegisterRt))) ||

(IF/ID.Branch && EX/MEM.MemRead &&

((EX/MEM.RegisterRd == IF/ID.RegisterRs) ||

(EX/MEM.RegisterRd == IF/ID.RegisterRt)))

Stall = LoadStall || BranchStall

1. **Flush**

Flush =

(IF/ID.Branch && BranchTaken) || IF/ID.Jump ||

(IF/ID.Branch && BranchMispredicted)

1. **ForwardMemory**

ForwardMemory =

if (MEM/WB.RegWrite

&& (MEM/WB.RegisterRd ≠ 0)

&& (MEM/WB.RegisterRd == EX/MEM.RegisterRt)) then '1'

else '0'

d-)

1. **No Hazard Test**

main:

lw $1, 0($0)

add $2, $3, $4

sw $5, 4($0)

addi $6, $7, 5

Machine code (hex):

8C010000 # lw $1, 0($0)

00641020 # add $2, $3, $4

AC050004 # sw $5, 4($0)

20E60005 # addi $6, $7, 5

1. **Raw Data Hazard Test**

main:

add $1, $2, $3

sub $4, $1, $5

and $6, $1, $7

or $8, $4, $1

Machine code (hex):

00430820 # add $1, $2, $3

00252022 # sub $4, $1, $5

00273024 # and $6, $1, $7

00814025 # or $8, $4, $1

1. **Load-Use Hazard Test**

main:

lw $1, 0($2)

add $3, $1, $4

sub $5, $1, $6

sw $3, 4($2)

Machine code (hex):

8C410000 # lw $1, 0($2)

00241820 # add $3, $1, $4

00262822 # sub $5, $1, $6

AC430004 # sw $3, 4($2)

1. **Branch Hazard Test**

main:

add $1, $2, $3

beq $1, $4, skip

add $5, $6, $7

sub $8, $9, $10

skip:

lw $11, 0($0) # Branch target

Machine code (hex):

00430820 # add $1, $2, $3

10240002 # beq $1, $4, skip (offset 2)

00C72820 # add $5, $6, $7

012A4022 # sub $8, $9, $10

8C0B0000 # lw $11, 0($0)

e-)

1. **Hazard Unit**

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 ((RegWriteM && WriteRegM != 0) && (WriteRegM == rsE))

ForwardAE = 3'b010; // Forward from MEM stage

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

ForwardAE = 3'b001; // Forward from WB stage

else

ForwardAE = 3'b000; // No forwarding needed

if ((RegWriteM && WriteRegM != 0) && (WriteRegM == rtE))

ForwardBE = 3'b010;

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

ForwardBE = 3'b001;

else

ForwardBE = 3'b000;

StallF = ((MemToRegE) &&

((rtE == rsD) || (rtE == rtD)));

StallD = StallF;

FlushE = StallD;

end

endmodule

1. **PipeDtoE**

module PipeDtoE(

input logic clk,

input logic FlushE,

input logic RegWriteD, MemToRegD, MemWriteD,

input logic [2:0] ALUControlD,

input logic ALUSrcD, RegDstD,

input logic [31:0] RD1D, RD2D,

input logic [4:0] RsD, RtD, RdD,

input logic [31:0] SignImmD,

output logic RegWriteE, MemToRegE, MemWriteE,

output logic [2:0] ALUControlE,

output logic ALUSrcE, RegDstE,

output logic [31:0] RD1E, RD2E,

output logic [4:0] RsE, RtE, RdE,

output logic [31:0] SignImmE

);

always\_ff @(posedge clk) begin

if (FlushE) begin

RegWriteE <= 0;

MemToRegE <= 0;

MemWriteE <= 0;

ALUControlE <= 0;

ALUSrcE <= 0;

RegDstE <= 0;

RD1E <= 0;

RD2E <= 0;

RsE <= 0;

RtE <= 0;

RdE <= 0;

SignImmE <= 0;

end

else begin

RegWriteE <= RegWriteD;

MemToRegE <= MemToRegD;

MemWriteE <= MemWriteD;

ALUControlE <= ALUControlD;

ALUSrcE <= ALUSrcD;

RegDstE <= RegDstD;

// Transfer data

RD1E <= RD1D;

RD2E <= RD2D;

RsE <= RsD;

RtE <= RtD;

RdE <= RdD;

SignImmE <= SignImmD;

end

end

endmodule

1. **PipeEtoM**

module PipeEtoM(

input logic clk,

input logic RegWriteE, MemToRegE, MemWriteE,

input logic [31:0] ALUOutE, WriteDataE,

input logic [4:0] WriteRegE,

output logic RegWriteM, MemToRegM, MemWriteM,

output logic [31:0] ALUOutM, WriteDataM,

output logic [4:0] WriteRegM

);

always\_ff @(posedge clk) begin

RegWriteM <= RegWriteE;

MemToRegM <= MemToRegE;

MemWriteM <= MemWriteE;

ALUOutM <= ALUOutE;

WriteDataM <= WriteDataE;

WriteRegM <= WriteRegE;

end

endmodule