Bilkent University

Computer Engineering

CS-224 Section 3

Preliminary Design Report

Lab 5

Boran Yildirim

21401947

c)

“branch” control hazard

Pipelined processor does not know what instruction to fetch next, because the branch decision has not been made by the time the next instruction is fetched. The branch misprediction penalty can be reduced by deciding the branch earlier in the decode stage. If branch is mispredicted then 1 clock cycle instruction is flushed.

“Load-use” data hazard

Load word case you have to stall until the memory stage to have the value. Only way to **prevent** such behavior is to have smart compilers that will schedule those two instructions in a different way.

e)

module forward\_data\_a( input [PC\_BITS - 1 : 0] read\_data\_1\_e, input [PC\_BITS - 1 : 0] result\_w,

input [PC\_BITS - 1 : 0] alu\_out\_m, input wire forward\_a\_e,

output reg [PC\_BITS - 1 : 0] src\_a\_e);

module forward\_data\_b( input [PC\_BITS - 1 : 0] read\_data\_2\_e, input [PC\_BITS - 1 : 0] result\_w,

input [PC\_BITS - 1 : 0] alu\_out\_m, input wire forward\_b\_e,

output reg [PC\_BITS - 1 : 0] write\_data\_e );

module hazard\_unit( input [4 : 0] rs\_e, input [4 : 0] rt\_e, input rs\_d, input rt\_d, input mem\_to\_reg\_e,

input mem\_to\_reg\_m, input write\_reg\_e, input reg\_write\_e,

input write\_reg\_m, input reg\_write\_m, input write\_reg\_w, input reg\_write\_w,

input branch\_d,

output [1 : 0] forward\_a\_e, output [1 : 0] forward\_b\_e,

output forward\_a\_d, output forward\_b\_d,

output stall\_f, output stall\_d, output flush\_e);

module reg\_decode\_exec(input clk, input flush\_e,

input reg reg\_write\_d, input reg mem\_to\_reg\_d, input reg mem\_write\_d,

input reg alu\_control\_d, input reg alu\_src\_d, input reg reg\_dst\_d,

input logic [PC\_BITS - 1 : 0] read\_data\_1\_d,

input logic [PC\_BITS - 1 : 0] read\_data\_2\_d,

output logic [PC\_BITS - 1 : 0] read\_data\_1\_e,

output logic [PC\_BITS - 1 : 0] read\_data\_2\_e,

input [PC\_BITS - 1 : 0] pc\_plus\_4d,

output [PC\_BITS - 1 : 0] pc\_plus\_4e,

input [PC\_BITS - 1 : 0] sign\_imm\_d,

output [PC\_BITS - 1 : 0] sign\_imm\_e,

input rs\_d, input rt\_d, input rd\_d,

output rs\_e, output rt\_e, output rd\_e,

output reg reg\_write\_e,

output reg mem\_to\_reg\_e,

output reg mem\_write\_e,

output reg alu\_control\_e,

output reg alu\_src\_e,

output reg reg\_dst\_e );

f)

module forward\_data\_a( input [PC\_BITS - 1 : 0] read\_data\_1\_e, input [PC\_BITS - 1 : 0] result\_w,

input [PC\_BITS - 1 : 0] alu\_out\_m, input wire forward\_a\_e,

output reg [PC\_BITS - 1 : 0] src\_a\_e);

always\_comb begin

if (forward\_a\_e == 2'b00)

src\_a\_e = read\_data\_1\_e;

else if (forward\_a\_e == 2'b01)

src\_a\_e = result\_w;

else

src\_a\_e = alu\_out\_m;

end

endmodule

module forward\_data\_b( input [PC\_BITS - 1 : 0] read\_data\_2\_e, input [PC\_BITS - 1 : 0] result\_w,

input [PC\_BITS - 1 : 0] alu\_out\_m, input wire forward\_b\_e,

output reg [PC\_BITS - 1 : 0] write\_data\_e );

always\_comb begin

if (forward\_b\_e == 2'b00)

write\_data\_e = read\_data\_2\_e;

else if (forward\_b\_e == 2'b01)

write\_data\_e = result\_w;

else

write\_data\_e = alu\_out\_m;

end

endmodule

module hazard\_unit( input [4 : 0] rs\_e, input [4 : 0] rt\_e, input rs\_d, input rt\_d, input mem\_to\_reg\_e,

input mem\_to\_reg\_m, input write\_reg\_e, input reg\_write\_e,

input write\_reg\_m, input reg\_write\_m, input write\_reg\_w, input reg\_write\_w,

input branch\_d,

output [1 : 0] forward\_a\_e, output [1 : 0] forward\_b\_e,

output forward\_a\_d, output forward\_b\_d,

output stall\_f, output stall\_d, output flush\_e);

always\_comb begin

if ((rs\_e != 0) & (rs\_e == write\_reg\_m) & reg\_write\_m)

forward\_a\_e = 2'b10;

else if ((rs\_e != 0) && (rs\_e == write\_reg\_w) && reg\_wrote\_w)

forward\_a\_e = 2'b01;

else

forward\_a\_e = 2'b00;

end

always\_comb begin

if ((rt\_e != 0) && (rt\_e == write\_reg\_m) && reg\_write\_m)

forward\_b\_e = 2'b10;

else if ((rt\_e != 0) && (rt\_e == write\_reg\_w) && reg\_wrote\_w)

forward\_b\_e = 2'b01;

else

forward\_b\_e = 2'b00;

end

logic forward\_decode;

always\_comb begin

forward\_a\_d = (rs\_d != 0) & (rs\_d == write\_reg\_m) & reg\_write\_m;

forward\_b\_d = (rt\_d != 0) & (rt\_d == write\_reg\_m) & reg\_write\_m;

end

logic lwstall;

logic branch\_stall;

always\_comb begin

lwstall = ((rs\_d == rt\_e) | (rt\_d == rt\_e)) & mem\_to\_reg\_e;

branch\_stall = (branch\_d & reg\_write\_e & (write\_reg\_e == rs\_d | write\_reg\_e == rt\_d)) | (branch\_d & mem\_to\_reg\_m & (write\_reg\_m == rs\_d | write\_reg\_m == rt\_d);

stall\_f = lw\_stall | branch\_stall;

stall\_d = lw\_stall | branch\_stall;

flush\_e = lw\_stall | branch\_stall;

end

endmodule

module reg\_decode\_exec(input clk, input flush\_e,

input reg reg\_write\_d, input reg mem\_to\_reg\_d, input reg mem\_write\_d,

input reg alu\_control\_d, input reg alu\_src\_d, input reg reg\_dst\_d,

input logic [PC\_BITS - 1 : 0] read\_data\_1\_d,

input logic [PC\_BITS - 1 : 0] read\_data\_2\_d,

output logic [PC\_BITS - 1 : 0] read\_data\_1\_e,

output logic [PC\_BITS - 1 : 0] read\_data\_2\_e,

input [PC\_BITS - 1 : 0] pc\_plus\_4d,

output [PC\_BITS - 1 : 0] pc\_plus\_4e,

input [PC\_BITS - 1 : 0] sign\_imm\_d,

output [PC\_BITS - 1 : 0] sign\_imm\_e,

input rs\_d, input rt\_d, input rd\_d,

output rs\_e, output rt\_e, output rd\_e,

output reg reg\_write\_e,

output reg mem\_to\_reg\_e,

output reg mem\_write\_e,

output reg alu\_control\_e,

output reg alu\_src\_e,

output reg reg\_dst\_e );

always @(posedge clk, posedge clr) begin

if (flush\_e) begin

reg\_dst\_e <= 2'b00;

alu\_src\_e <= 1'b0;

mem\_to\_reg\_e <= 2'b00;

reg\_write\_e <= 1'b0;

mem\_write\_e <= 1'b0;

branch\_e <= 1'b0;

alu\_control\_e <= 2'b00;

jump\_e <= 1'b0;

rs\_e <= 5'b0;

rt\_e <= 5'b0;

rd\_e <= 5'b0;

pc\_plus\_4e <= 32'b0;

sign\_imm\_e <= 32'b0;

read\_data\_1\_e <= 0;

read\_data\_2\_e <= 0;

end

else begin

reg\_dst\_e <= reg\_dst\_d;

alu\_src\_e <= alu\_src\_d;

mem\_to\_reg\_e <= mem\_to\_reg\_d;

reg\_write\_e <= reg\_write\_d;

mem\_write\_e <= mem\_write\_d;

branch\_e <= branch\_d;

alu\_control\_e <= alu\_control\_d;

jump\_e <= jump\_d;

rs\_e <= rs\_d;

rt\_e <= rt\_d;

rd\_e <= rd\_d;

pc\_plus\_4e <= pc\_plus\_4d;

sign\_imm\_e <= sign\_imm\_d;

read\_data\_1\_e <= read\_data\_1\_d;

read\_data\_2\_e <= read\_data\_2\_d;

end

end

endmodule