**CS 224**

**Lab 4**

**Preliminary Design Report**

Group

Cüneyt EREM

21202398

b)datapath

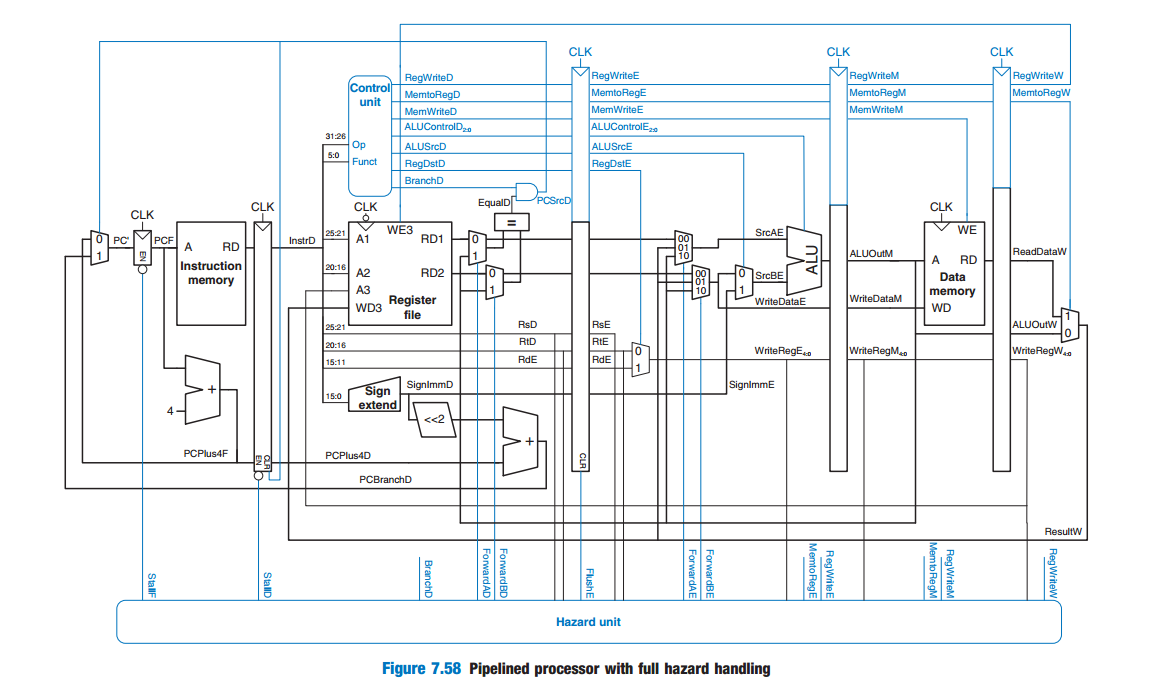
c)list of hazards(load-use, load-store)

d)logic equation of hazard unit

e-f)moduller (hazard unit dahil)

g)test programı hazard ve hazardsız olan

B)



C)

Data Hazard; pipeline stage

-Compute use(exe-mem)

-Compute store(exe-mem)

-load use(decode)

-load store(decode)

Control Hazard;

-branch handle(fetch-decode)

-jump handle(fetch-decode)

Solutions;

*Pipeline stall*, is a method for preventing data, structural, and branch hazards from occurring.

As instructions are fetched, and if determines whether a hazard could/will occur. then the control logic inserts [NOPs](https://en.wikipedia.org/wiki/NOP" \o "NOP) into the pipeline.

Thus, before the next instruction (which would cause the hazard) is executed, the previous one will have had sufficient time to complete and prevent the hazard. If the number of NOPs is equal to the number of stages in the pipeline, the processor has been cleared of all instructions and can proceed free from hazards. All forms of stalling introduce a delay before the processor can resume execution.

*Flushing the pipeline* occurs when a branch instruction jumps to a new memory location, invalidating all previous stages in the pipeline. These previous stages are cleared allowing the pipeline to continue at the new instruction indicated by the branch

So;

Control Hazard sol.;

* Insert nops in code at compile time
* Rearrange code at compile time

Data Hazard sol.:

* Stall the processor at run time
* Forward data at run time

D)

The forwarding logic for SrcB (ForwardBE) is identical except that it checks rt rather than rs.

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

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

StallF = StallD = FlushE = lwstall

The function of the Decode stage forwarding logic

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

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

Stalling logic:

*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)*

OR;

branchstall =

BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD) OR

BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD)

load or a branch hazard:

StallF = StallD = FlushE = lwstall OR branchstall

E-F)

module datapath(input clk, reset,

input pcsrc,

input alusrc, regdst,

input regwrite,

input [1:0] memtoreg, jump,

input [2:0] alucontrol,

output zero,

output [31:0] pc,

input [31:0] instr,

output [31:0] aluout, writedata,

input [31:0] readdata);

wire [4:0] writereg;

wire [31:0] pcnext, pcnextbr, pcplus4f, pcplus4d, pcplus4, pcbranch,readminus4;

wire [31:0] signimm, signimmsh;

wire [31:0] srca, srcb;

wire [31:0] result;

wire stallF, stallD, flushE, memwriteE, branchE, alucontrolE, alusrcE, regdstE, regwriteE, memtoregE;

wire [31:0] resultE, writedataE, instrE, signimmE, pcplus4E;

// next PC logic

flopenr #(32) pcreg(clk, reset, stallF, pcnext, pc);

flopenr2 #(32) pcreg(clk, reset, stallD, readdata, pcplus4f, instr, pcplus4d, pc);

flopenr3 #(32) pcreg(clk, reset, flushE, memwrite, branch, alucontrol, alusrc, regdst, regwrite, memtoreg,

result, writedata, instr[25:21], instr[21:16], instr[15:11], signimm, pcplus4d,

memwriteE, branchE, alucontrolE, alusrcE, regdstE, regwriteE, memtoregE

);

adder pcadd1(pc, 32'b100, pcplus4);

sl2 immsh(signimm, signimmsh);

adder pcadd2(pcplus4f, signimmsh, pcbranch);

adder rmin4( 32'hFFFFFFFC, aluout, readminus4); //for push

mux2 #(32) branchmux(pcplus4E, pcbranch, pcsrc, pcnextbr);

//mux3 #(32) jumpmux(pcnextbr, {pcplus4[31:28], instr[25:0], 2'b00}, readdata, jump, pcnext);

mux3 #(32) jumpmux(pcnextbr, {pcplus4[31:28], instr[25:0], 2'b00}, srca, jump, pcnext);

// register file logic

regfile rf(clk, regwrite, instr[25:21], instr[20:16], writereg, result, srca, writedata);

mux2 #(5) w\_addrmux(instrE[20:16], instrE[15:11], regdstE, writereg);

mux4 #(32) w\_datamux(aluout, readdata, pcplus4, readminus4, memtoreg, result);

//mux3 #(32) w\_datamux(aluout, readdata, pcplus4, memtoreg, result);

//mux4 #(32) w\_datamux(aluout, readdata, pcplus4, {instr[15:0], 16'h0000 }, memtoreg, result);

signext se(instr[15:0], signimm);

// ALU logic

mux2 #(32) srcbmux(writedataE, signimmE, alusrcE, srcb);

alu alu(srca, srcb, alucontrolE, aluout, zero);

endmodule

module Hazard(clk, reset, branch, memtoregE, regwriteE, memtoregM, regwriteM, regwriteW, stallF, stallD,

instr[25:21], instr[20:16], instr[15:11], flushE, instrE[25:21], instrE[20:16], ForwardAD, ForwardBD, ForwardAE,

ForwardBE, writeregE, writeregM, writeregW);

input clk, reset, branch, memtoregE, regwriteE, memtoregM, regwriteM, regwriteW

output reg stallF, stallD, instr[25:21], instr[20:16], instr[15:11], flushE, instrE[25:21], instrE[20:16],

ForwardAD, ForwardBD, ForwardAE, ForwardBE, writeregE, writeregM, writeregW;

/\*keep track of rw\*/

always @(negedge CLK) begin

if(Reset\_L == 0) begin

rw1 <= 0;

...

end

else begin

rw1 <= Bubble?0:rw;//insert bubble if needed

...

end

end

/\*FSM state register\*/

always @(negedge CLK) begin

if(Reset\_L == 0)

FSM\_state <= 0;

else

FSM\_state <= FSM\_nxt\_state;

end

/\*FSM next state and output logic\*/

always @(\*) begin //combinatory logic

case(FSM\_state)

NoHazard\_state: begin

if(Jump== 1'b1) begin //prioritize jump

//uncondition return to no hazard state

/\* Provde the value of FSM\_nxt\_state and outputs (PCWrite,IFWrite,Buble)\*/

end

else if(cmp1== 1'b1) begin //3-delay data hazard

//uncondition return to no hazard state

/\* Provde the value of FSM\_nxt\_state and outputs (PCWrite,IFWrite,Buble)\*/

end

...

/\* Complite the "NoHazard\_state" state as needed\*/

end

Bubble0\_state: begin

//uncondition return to no hazard state

/\* Provde the value of FSM\_nxt\_state and outputs (PCWrite,IFWrite,Buble)\*/

end

....

/\* Complite the states as needed\*/

default: begin

FSM\_nxt\_state <= #2 NoHazard\_state;

PCWrite <= #2 1'bx;

IFWrite <= #2 1'bx;

Bubble <= #2 1'bx;

end

endcase

end

endmodule

module regfile(input clk,

input we3,

input [4:0] ra1, ra2, wa3,

input [31:0] wd3,

output [31:0] rd1, rd2);

reg [31:0] rf[31:0];

// three-ported register file

// read two ports combinationally

// write third port on rising edge of clock

// register 0 hardwired to 0

always @(posedge clk)

if (we3) rf[wa3] <= wd3;

assign rd1 = (ra1 != 0) ? rf[ra1] : 0;

assign rd2 = (ra2 != 0) ? rf[ra2] : 0;

endmodule

module adder(input [31:0] a, b,

output [31:0] y);

assign y = a + b;

endmodule

module sl2(input [31:0] a,

output [31:0] y);

// shift left by 2

assign y = {a[29:0], 2'b00};

endmodule

module signext(input [15:0] a,

output [31:0] y);

assign y = {{16{a[15]}}, a};

endmodule

module flopr #(parameter WIDTH = 8)

(input clk, reset,

input [WIDTH-1:0] d,

output reg [WIDTH-1:0] q);

always @(posedge clk, posedge reset)

if (reset) q <= 0;

else q <= d;

endmodule

module flopenr #(parameter WIDTH = 8)

(input clk, reset,

input en,

input [WIDTH-1:0] d,

output reg [WIDTH-1:0] q);

always @(posedge clk, posedge reset)

if (reset) q <= 0;

else if (en) q <= d;

endmodule

module flopenr2 #(parameter WIDTH = 8)

(input clk, reset,

input en,

input [31:0] d1,

input [31:0] d2,

output reg [31:0] q1,

output reg [31:0] q2);

always @(posedge clk, posedge reset)

if (reset)

begin

q1 <= 0;

q2 <= 0;

end

else if (en)

begin

q1 <= d1;

q2 <= d2;

end

endmodule

module flopenr3 #(parameter WIDTH = 8)

(input clk, reset,

input en,

input d1, d2, d3, d4, d5, d6,

input [1:0] d7, // memtoreg, jump yok

input [31:0] d8, d9,

input [4:0] d10, d11, d12,

input [31:0] d13, d14,

output reg q1, q2, q3, q4, q5, q6,

output reg [1:0] q7,

output reg [31:0] q8, q9,

output reg [4:0] q10, q11, q12,

output reg [31:0] q13, q14);

always @(posedge clk, posedge reset)

if (reset)

begin

q1 <= 0;q2 <= 0;q3 <= 0;q4 <= 0;q5 <= 0;

q6 <= 0;q7 <= 0;q8 <= 0;q9 <= 0;q10 <= 0;

q11 <= 0;q12 <= 0;q13 <= 0;q14 <= 0;

end

else if (en)

begin

q1 <= d1;q2 <= d2;q3 <= d3;q4 <= d4;q5 <= d5;

q6 <= d6;q7 <= d7;q8 <= d8;q9 <= d9;q10 <= d10;

q11 <= d11;q12 <= d12;q13 <= d13;q14 <= d14;

end

endmodule

module mux2 #(parameter WIDTH = 8)

(input [WIDTH-1:0] d0, d1,

input s,

output [WIDTH-1:0] y);

assign y = s ? d1 : d0;

endmodule

module mux3 #(parameter WIDTH = 8)

(input [WIDTH-1:0] d0, d1, d2,

input [1:0] s,

output [WIDTH-1:0] y);

assign y = s[1] ? d2: (s[0] ? d1 : d0);

endmodule

module mux4 #(parameter WIDTH = 8)

(input [WIDTH-1:0] d0, d1, d2, d3,

input [1:0] s,

output [WIDTH-1:0] y);

assign y = s[1] ? (s[0] ? d3 : d2): (s[0] ? d1 : d0);

endmodule

G)

Without hazard

module imem ( input [5:0] addr,

output reg [31:0] instr);

always@(addr)

case ({addr,2'b00})

// address instruction

// ------- -----------

8'h00: instr = 32'h20020005; //addi $2,$0,5

8'h04: instr = 32'h2003000c; //addi $3,$0,12

8'h08: instr = 32'h2067fff7; //addi $7,$3,-9

8'h0c: instr = 32'h00e22025; //or $4,$7,$2

8'h10: instr = 32'h00642824; //and $5,$3,$4

8'h14: instr = 32'h00a42820; //add $5,$5,$4

8'h18: instr = 32'h10a7000a; //beq $5,$7, 0x10

8'h1c: instr = 32'h0064202a; //slt $4,$3,$4

8'h20: instr = 32'h10800001; //beq $5, $0, 0x28

8'h24: instr = 32'h20050000; //addi $5,$0,0

8'h28: instr = 32'h00e2202a; //slt $4,$7,$2

8'h2c: instr = 32'h00853820; //add $7,$4,$5

8'h30: instr = 32'h00e23822; //sub $7,$7,$2

8'h34: instr = 32'hac670044; //sw $7,68($3)

8'h38: instr = 32'h8c020050; //lw $2,80($0)

8'h3c: instr = 32'h08000011; //j 0x44

8'h40: instr = 32'h20020001; //addi $2,$0,1

8'h44: instr = 32'hac020054; //sw $2,84($0)

default: instr = {32{1'bx}}; // unknown instruction

endcase

endmodule

with hazard

module imem ( input [5:0] addr,

output reg [31:0] instr);

always@(addr)

case ({addr,2'b00})

// address instruction

// ------- -----------

8'h00: instr = 32'h20020005; //addi $2,$0,5

8'h04: instr = 32'h2003000c; //addi $3,$0,12

8'h08: instr = 32'h2067fff7; //addi $7,$3,-9

8'h0c: instr = 32'h00e22025; //or $4,$7,$2

8'h10: instr = 32'h00642824; //and $5,$3,$4

8'h14: instr = 32'h00a42820; //add $5,$5,$4

8'h18: instr = 32'h10a7000a; //beq $5,$7, 0x10

8'h1c: instr = 32'h04000000; //stall

8'h18: instr = 32'h04000000; //stall

8'h1c: instr = 32'h0064202a; //slt $4,$3,$4

8'h20: instr = 32'h10800001; //beq $5, $0, 0x28

8'h2c: instr = 32'h04000000; //stall

8'h20: instr = 32'h04000000; //stall

8'h24: instr = 32'h20050000; //addi $5,$0,0

8'h28: instr = 32'h00e2202a; //slt $4,$7,$2

8'h2c: instr = 32'h00853820; //add $7,$4,$5

8'h30: instr = 32'h00e23822; //sub $7,$7,$2

8'h34: instr = 32'hac670044; //sw $7,68($3)

8'h3c: instr = 32'h04000000; //nop

8'h34: instr = 32'h04000000; //nop

8'h38: instr = 32'h8c020050; //lw $2,80($0)

8'h3c: instr = 32'h08000011; //j 0x44

8'h40: instr = 32'h20020001; //addi $2,$0,1

8'h44: instr = 32'h04000000; //nop

8'h48: instr = 32'hac020054; //sw $2,84($0)

default: instr = {32{1'bx}}; // unknown instruction

endcase

endmodule