Bilkent University

Computer Engineering

CS 224 – Computer Organization

**Preliminary Design Report**

**Lab 4**

**Section 3**

**Gülsüm Güdükbay (21401148)**

**Date Of Lab: 22 March 2016**

**b) Datapath**

SignImm

CLK

A

RD

**Instruction**

**Memory**

+

4

A1

A3

WD3

RD2

RD1

WE3

A2

CLK

**Sign Extend**

**Register**

**File**

0

1

A

RD

**Data**

**Memory**

WD

WE

0

1

PCF

0

1

PC'

InstrD

25:21

20:16

15:0

5:0

SrcBE

20:16

15:11

<<2

+

WriteDataE

SrcAE

PCPlus4F

ResultW

31:26

RegDstD

MemWriteD

MemtoRegD

RegWriteD

ALUSrcD

BranchD

Op

Funct

**Control**

**Unit**

Zero

PCSrcE

CLK

ALUControlD

2:0

ALU

0

1

25:0

<<2

27:0

31:28

PCJump

JumpD

EN

**Hazard Unit**

CLK

EN

StallF

StallD

25:21

clr

CLK

rsD

rtD

rdD

FlushE

rsE

rtE

rdE

0

1

RegDstE

MemWrıiteE

MemtoRegE

RegWriteE

ALUSrcE

BranchE

ALUControlE

2:0

00

01

10

ForwardAE

ForwardBE

00

01

10

CLK

ALUOutM

PCBranchE

MemWriteM

MemtoRegM

RegWriteM

WriteRegE 4:0

WriteRegM 4:0

WriteDataM

MemToRegE

CLK

WriteRegW4:0

ALUOutW

RegWriteM

RegWriteW

PCPlus4D

ReadDataM

WriteRegM4:0

**c) List of All Hazards And Their Descriptions**

1. Compute – Use: This **data** hazard occurs when a calculation is not finished yet and it is used in the next instruction’s execute stage. If it is used in the next instruction’s execute stage, it has to be forwarded two times (from the memory stage) for two clock cycles to pass and come back to the register file. If it is used in the intruction one after the next, it has to be forwarded only once from the writeback stage to be moved into the register file. The amount of forwarding is selected via the 3 – to – 1 mux.
2. Load – Use: This **data** hazard occurs when a load is performed and that loaded register is used in the next instruction, since the data in the data memory isn’t read out yet. Since the data is not given yet, we have to wait for it. Since we cannot forward the data because it isn’t available yet, we have to “stall” meaning that the program counter is paused for one clock cycle (by disabling the PC register and Decode flip flop and flushing the Execute flip flop).
3. Load – Store: This is also a **data** hazard caused because when a data is loaded from memory and then stored in the next instuction. There are three types: if the ReadDataW (checking this since MemToRegW should be 1) is the data to be written (WriteDataM), it is only forwarded; else, if the signed immediate equals 0, forwarding is also enough, but if it’s not, a stall with the data forwarded from the Writeback stage should be performed to the Execute stage, and flush the Memory flip flops to prevent the wrong data to propagate.
4. J – Type Jump: This is a **control** hazard caused because of the fact that the jump decision is made during the Decode stage, and that’s why we have to flush the instruction that was executed during the decision making, since it should be because of its unnecessity and because of the fact that it shouldn’t propagate anymore to prevent data corruption.
5. Branch: This is a **control** hazard caused because of the fact that the instructions at the Execute and Decode stages shouldn’t be propagating. So, the Decode and the Execute flip flops should be flushed.

**d)Logic Equations**

//Compute - use hazard

forwardAE = ((rsE!=0)&&(rsE == writeRegM) && regWriteM)? 2'b10 : (((rsE != 0)&&(rsE == writeRegW) && regWriteW) ? 2'b01:2'b00);

forwardBE = ((rtE!=0)&&(rtE == writeRegM) && regWriteM)? 2'b10 : (((rtE != 0)&&(rtE == writeRegW) && regWriteW) ? 2'b01:2'b00);

//Load - use hazard

stallF = (((rsD == rtE) || (rtD == rtE)) && (memToRegE));

stallD = stallF;

assign flushE = stallF|branch;

//J - Type Jump Hazard and Branching Hazard

assign flushD = branch ;

**e) Changes in Verilog Modules**

The datapath is destroyed and new modules for each of the stages are created. The mips module will have the instance of fetch, decode, execute, memory and writeback stages. The top module will not change, however the mips module will have all the flip flops created for each of the signals to correctly propagate them in correct way. The controller’s and’ing for the branch decision is deleted and then it is taken into the execute stage. A hazard unit will be created to take care of the hazards during data propagation and due to controll signals.

**f) For each module in the part e) list, give the Verilog code.**

module HazardUnit( input [4:0] rsD, rtD, rsE, rtE, writeRegM, writeRegW,

input memToRegE, regWriteM, regWriteW, branch, jump,

output stallF, stallD, flushE, flushD,

output [1:0] forwardAE, forwardBE);

//Compute - use hazard

assign forwardAE = ((rsE!=0)&&(rsE == writeRegM) && regWriteM)? 2'b10 : (((rsE != 0)&&(rsE == writeRegW) && regWriteW) ? 2'b01:2'b00);

assign forwardBE = ((rtE!=0)&&(rtE == writeRegM) && regWriteM)? 2'b10 : (((rtE != 0)&&(rtE == writeRegW) && regWriteW) ? 2'b01:2'b00);

//Load - use hazard

assign stallF = (((rsD == rtE) || (rtD == rtE)) && (memToRegE));

assign stallD = stallF;

assign flushE = stallF|branch;

//J - Type Jump Hazard and Branching Hazard

assign flushD = branch ;//| jump;

endmodule

module controller(input [5:0] op, funct,

output memtoreg, memwrite,

output alusrc,

output regdst, regwrite,

output jump, branch,

output [2:0] alucontrol);

wire [1:0] aluop;

maindec md (op, regwrite, regdst, alusrc, branch,

memwrite, memtoreg, aluop, jump);

aludec ad (funct, aluop, alucontrol);

endmodule

module Fetch( input clk, reset, jumpD, PCSrcE, stallF,

input [31:0] jumpTAddrD, PCBranchE,

output [31:0] PCF, PCPlus4F);

not(enclock, stallF);

wire [31:0] pcNext, pcNextDoBranch;

flopenr #(32) pcregister(clk, reset, enclock, pcNext, PCF);

adder a(PCF, 32'd4, PCPlus4F);

mux2 #(32) branchMux(PCPlus4F, PCBranchE, PCSrcE, pcNextDoBranch);

mux2 #(32) jumpMux(pcNextDoBranch, jumpTAddrD, jumpD, pcNext);

endmodule

module Decode( input clk, RegWriteW,

input [4:0] writeRegW,

input [31:0] instructionD, PCPlus4D, resultW,

output [31:0] rd1D, rd2D, jtaD, signImm,

output [4:0] rsD, rtD, rdD);

assign rsD = instructionD[25:21];

assign rtD = instructionD[20:16];

assign rdD = instructionD[15:11];

regfile rf(clk, RegWriteW, instructionD[25:21],

instructionD[20:16], writeRegW,

resultW, rd1D, rd2D);

signext si(instructionD[15:0], signImm);

assign jtaD = {PCPlus4D[31:28], instructionD[25:0],2'b00};

endmodule

module Execute( input branchE, regDstE, ALUsrcE,

input [1:0] forwardAE, forwardBE,

input [2:0] ALUControlE,

input [4:0] rtE, rdE,

input [31:0] rd1E, rd2E, signImmE, PCPlus4E, ALUOutM, resultW,

output [31:0] ALUOutE, writeDataE, PCBranchE,

output [4:0] writeRegE,

output PCSrcE);

wire [31:0] outOffrwrdAMux, outputOfsrcBMux, shiftedImm;

mux4 #(32) forwrdAMux(rd1E, resultW, ALUOutM, forwardAE, outOffrwrdAMux);

mux4 #(32) forwrdBMux(rd2E, resultW, ALUOutM, forwardBE, writeDataE);

mux2 #(5) rtOrRd(rtE, rdE, regDstE, writeRegE);

mux2 #(32) srcBMux(writeDataE, signImmE, ALUsrcE, outputOfsrcBMux);

sl2 shiftSignExtImm(signImmE, shiftedImm);

adder adderForPCBranchE(shiftedImm, PCPlus4E, PCBranchE);

alu a(outOffrwrdAMux, outputOfsrcBMux, ALUControlE, ALUOutE,ALUZero);

and(PCSrcE, branchE, ALUZero);

endmodule

module WriteBack( input memToRegW,

input [31:0] readDataW, ALUOutW,

output [31:0] resultW);

mux2 #(32) res(ALUOutW, readDataW,memToRegW, resultW);

endmodule

module mips (input clk, reset,

output [31:0] PCF,

input [31:0] instrF,

output memWriteM,

output [31:0] ALUOutM, writeDataM,

input [31:0] readDataM);

wire [1:0] forwardAE, forwardBE;

wire [2:0] ALUControlD, ALUControlE;

wire [31:0] jumpTAddrD, PCBranchE,instrD, PCPlus4F, PCPlus4D,PCPlus4E, rd1D, rd1E, rd2D,rd2E, signImmD, signImmE, ALUOutE, ALUOutW, resultW, writeDataE, readDataW;

wire [4:0] writeRegW, writeRegM, writeRegE, rsD, rtD, rdD, rsE, rtE, rdE;

HazardUnit h(rsD, rtD, rsE, rtE, writeRegM, writeRegW, memToRegE, regWriteM, regWriteW, PCSrcE, jumpD, stallF, stallD, flushE,flushD, forwardAE, forwardBE);

controller c (instrD[31:26], instrD[5:0], memToRegD, memWriteD,ALUSrcD, regDstD, regWriteD, jumpD, branchD, ALUControlD);

////////////////FETCH/////////////////

or(clearD, flushD, reset);

not(decodeEn, stallD);

Fetch f( clk, reset, jumpD, PCSrcE, stallF, jumpTAddrD, PCBranchE, PCF, PCPlus4F);

flopenr #(32)ffpcPlus4(clk, clearD, decodeEn, PCPlus4F, PCPlus4D);

flopenr #(32) ffinstr (clk, clearD, decodeEn, instrF, instrD);

////////////////DECODE/////////////////

or(clearE, flushE, reset);

Decode d(clk, regWriteW, writeRegW, instrD, PCPlus4D, resultW, rd1D, rd2D, jumpTAddrD, signImmD, rsD, rtD, rdD);

flopr #(1) rwd(clk, clearE, regWriteD, regWriteE);

flopr #(1) mtrd(clk, clearE, memToRegD, memToRegE);

flopr #(1) mwd(clk, clearE, memWriteD, memWriteE);

flopr #(3) acd(clk, clearE, ALUControlD, ALUControlE);

flopr #(1) asd(clk, clearE, ALUSrcD, ALUSrcE);

flopr #(1) rdd(clk, clearE, regDstD, regDstE);

flopr #(1) bd(clk, clearE, branchD, branchE);

flopr #(32) read1(clk, clearE, rd1D, rd1E);

flopr #(32) read2(clk, clearE, rd2D, rd2E);

flopr #(5) rs(clk, clearE, rsD, rsE);

flopr #(5) rt(clk, clearE, rtD, rtE);

flopr #(5) rd(clk, clearE, rdD, rdE);

flopr #(32) imm(clk, clearE, signImmD, signImmE);

flopr #(32) pcp4(clk, clearE, PCPlus4D, PCPlus4E);

////////////////EXECUTE/////////////////

Execute e(branchE, regDstE, ALUSrcE, forwardAE, forwardBE, ALUControlE, rtE, rdE, rd1E, rd2E, signImmE, PCPlus4E, ALUOutM, resultW, ALUOutE, writeDataE, PCBranchE, writeRegE, PCSrcE);

flopr #(1) rwe(clk, reset, regWriteE, regWriteM);

flopr #(1) mtre(clk, reset, memToRegE, memToRegM);

flopr #(1) mwe(clk, reset, memWriteE, memWriteM);

flopr #(32) aluo(clk, reset, ALUOutE, ALUOutM);

flopr #(32) wde(clk, reset, writeDataE, writeDataM);

flopr #(5) wre(clk, reset, writeRegE, writeRegM);

////////////////MEMORY/////////////////

flopr #(1) rwm(clk, reset, regWriteM, regWriteW);

flopr #(1) mtrm(clk, reset, memToRegM, memToRegW);

flopr #(5) wrm(clk, reset, writeRegM, writeRegW);

flopr #(32) rdm(clk, reset, readDataM, readDataW);

flopr #(32) aluow(clk, reset, ALUOutM, ALUOutW);

////////////////WRITEBACK/////////////////

WriteBack w( memToRegW, readDataW, ALUOutW, resultW);

endmodule

**g) MIPS Assembly Code For Testing And Fixing Hazards**

addi $v0, $0, 5

addi $v1, $0, 12

nop

nop#compute - use

addi $a3, $v1, -9

nop

nop#compute - use

or $a0, $a3, $v0

nop

nop#compute - use

and $a1, $v1, $a0

nop

nop#compute - use

add $a1, $a1, $a0

nop

nop#compute - use

beq $a1, $a3, ten

slt $a0, $v1, $a0

nop

nop#compute - use

beq $a0, $0, one

nop

nop#forward

#stall

addi $a1, $0, 0

one: slt $a0, $a3, $v0

nop

nop#compute - use

add $a3, $a0, $a1

nop

nop#compute - use

sub $a3, $a3, $v0

nop

nop#compute - use

sw $a3, 68($v1)

lw $v0, 80($0)

j ten

nop #flush

addi $v0, $0, 1

nop

nop#compute - use

ten: sw $v0, 84($0)

loop: j loop