Part C)

There are two types of hazards exist in mips pipeline processor design:

**Data Hazards and Control Hazards**

**Data Hazard** occurs when next instruction tries to use an instruction that will be modified by previous its clock cycles cannot reach the modified version and this creates a data hazard.

3 types of Data Hazards occur: First, Write after reading (WAR) about anti-dependency. Second, Write after write(WAW) about output dependency. Third, Read after write(RAW) about true dependency

**The solution for the Data Hazards**

With Stall: Stall is related to the delay operation. This stops program until the wanted value is created or the instruction is handled. Namely, passes some stages until the value is available for the next instruction, these waiting stages are named as Bubble. These bubbles cause the wait signal until the wanted value is created and updated.

With Forwarding: “Forwarding is necessary when an instruction in the Execute stage has a source register matching the destination register of an instruction in the Memory or WriteBack stage”(Taken from the book) Namely, forwarding is a solution that connects the lacked values before their instruction completed to the necessary places. It connects the needed value with its place and this connection between the stages handles the problem.

**Control Hazards**

Instructions like Beq and Bne are related to the control hazards due to the problem of calculating PC next. For these instructions, their next can be either PC + 4 or PC next. Until the next PC state calculated. The pipeline will be computing the next instruction in the order but indeed instruction in the order is not calculated yet.

**Branch prediction:**

Branch prediction is a way for solving this hazard. In this method, we can compute the result of the branch after the register file. So we don’t have to wait until the execution stage. By comparing in the register file. We can recover ourselves from the flushing new instruction. Because comparison’s result can send us to the new address before we fetch new instruction to our processor.

**Stall:**

The stall is one way of solving this problem. As we said until we calculate the branch result, our processor will fetch the next instruction which is PC + 4. To solve this we can wait until the branch result is calculated. This can be done with the Stall method. But the timing can create a problem with stall solution method.

**Part D)**

//I have used same controller values and send them to the datapath I have made the most of the work in the datapath

module mips (input logic clk, reset,

output logic[31:0] pc,

input logic[31:0] instr,

output logic MemWriteD,

output logic[31:0] aluout, writedata,

input logic[31:0] readdata);

logic BranchD;

logic MemtoRegD;

logic pcsrc, zero, ALUSrcD;

logic RegDstD;

logic RegWriteD;

logic jump;

logic [2:0] ALUControlD;

controller c (instr[31:26], instr[5:0], zero, MemtoRegD, MemWriteD, pcsrc,

ALUSrcD, RegDstD, RegWriteD, jump, ALUControlD, BranchD);

datapath dp (clk, reset, BranchD , MemWriteD ,MemtoRegD, pcsrc, ALUSrcD, RegDstD, RegWriteD, jump,

ALUControlD, zero, pc, instr, aluout, writedata, readdata);

endmodule

**Part E)**

//This is datapath I have draw the Pipeline graph given in our lab files and created connections according to that, then I have send Control Unit values into datapath and I have send them into pipes and I also took Hazard value from Hazard Unit according to these values I have updated the control units according to given Hazard control values.

module datapath (input logic clk, reset, BranchD, MemWriteD ,MemtoRegD, pcsrc, ALUSrcD, RegDstD,

input logic RegWriteD, jump,

input logic[2:0] ALUControlD,

output logic ZeroE,

output logic[31:0] pc,

input logic[31:0] instr,

output logic[31:0] aluout, WriteDataE,

input logic[31:0] readdata);

logic [4:0] writereg;

logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;

logic [31:0] signimm, signimmsh, srca, srcb, result;

//New added

logic [4:0] rsD, rtD, rdD, rsE, rtE, rdE, WriteRegE;

logic [31:0] instrD, PcPlus4F, PcBranchM, PcPlus4D, ResultW, signimmD, AluOutW;

logic PcSrcM;

logic stallF;

logic stallD;

// ForwardAD, ForwardBD,

logic FlushE;

logic ege;

logic [1:0] ForwardAE;

logic [1:0] ForwardBE; // ForwardBE; // Wires for connecting Hazard Unit

logic RegWriteE, MemtoRegE, MemWriteE;

logic [2:0] ALUControlE;

logic ALUSrcE, RegDstE, BranchE;

logic [31:0] rd1E, rd2E, SignImmE, PcPlus4E, signExt, PcBranchE, hazardOut1, ALUOutM, SrcAE, SrcBE, ALUOutE, WriteDataM;

assign aluout = ALUOutE;

logic RegWriteM, MemtoRegM, MemWriteM, BranchM, ZeroM;

logic MemtoRegW;

logic [4:0] WriteRegM;

//logic transfers

logic RegWriteW, WriteRegW;

logic [31:0] rd1, rd2, ReadDataM, ReadDataW;

flopr #(32) pcreg(clk, reset, pcnext, pc);

adder pcadd1(pc, 32'b100, PcPlus4F);

mux2 #(32) dd(PcPlus4F, PcBranchM, PcSrcM, pcnext);

//This Pipe is pass fetch values to the Decode stage with the clock signal

PipeFtoD call1(instr, PcPlus4F,

stallD, clk, // StallD will be connected as this EN

instrD, PcPlus4D);

mux2 aa(PcPlus4F, ResultW, PcSrcM, pcnextbr);

assign rsD = instrD[25:21];

assign rtD = instrD[20:16];

assign rdD = instrD[15:11];

//Decoding part

signext se (instrD[15:0], signimmD);

regfile rf (clk, RegWriteW, instrD[25:21], instrD[20:16], WriteRegW, // Instantiated register file.

ResultW, rd1, rd2);

PipeDtoE call2(FlushE, clk, RegWriteD, MemtoRegD, MemWriteD, ALUControlD, ALUSrcD, RegDstD, BranchD,

rd1, rd2, rsD, rtD, rdD, signimmD, PcPlus4D,

RegWriteE, MemtoRegE, MemWriteE, ALUControlE, ALUSrcE, RegDstE, BranchE,

rd1E, rd2E, rsE, rtE, rdE, SignImmE, PcPlus4E);

//Execution

mux2 cc(rtE, rdE, RegDstE, WriteRegE);

sl2 ext(SignImmE, signExt);

adder adder2(PcPlus4E, signExt, PcBranchE); // pc + singExt = pcPlus4

mux3 #(32) first(rd1E, hazardOut1, ALUOutM, ForwardAE, SrcAE);

mux3 #(32) second(rd2E, hazardOut1, ALUOutM, ForwardBE, WriteDataE);

mux2 #(32) bb(WriteDataE, SignImmE, ALUSrcE, SrcBE);

alu as(SrcAE, SrcBE, ALUControlE , ALUOutE, ZeroE);

//Memory

PipeEtoM call3(clk, RegWriteE, MemtoRegE, MemWriteE, BranchE, ZeroE,

ALUOutE, WriteDataE, WriteRegE, PcBranchE,

RegWriteM, MemtoRegM, MemWriteM, BranchM, ZeroM,

ALUOutM, WriteDataM, WriteRegM, PcBranchM);

assign PcSrcM = BranchM & ZeroM;

//Mem M to WriteBack

PipeMtoW call4(clk, RegWriteM, MemtoRegM, ReadDataM, ALUOutM, WriteRegM,

RegWriteW, MemtoRegW, ReadDataW, AluOutW, WriteRegW);

mux2 mux1(AluOutW, ReadDataW, MemtoRegW ,ResultW);

HazardUnit callme(WriteRegE, BranchD ,RegWriteW,

WriteRegW,

RegWriteM,MemtoRegM,

WriteRegM,

RegWriteE, MemtoRegE,

rsE,rtE,

rsD,rtD,

ForwardAE,ForwardBE,

FlushE, stallD, stallF, ege);

endmodule

//This is hazard method which calculates the hazard outputs for me and control the pipe passes and stalls

module HazardUnit( input logic WriteRegE, BranchD ,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 [1:0] ForwardAE,ForwardBE,

output logic FlushE, StallD, StallF, input logic ege

);

logic lwstall, branchstall, branchstall1, branchstall12;

assign branchstall = BranchD && RegWriteE && ((WriteRegE == rsD) || (WriteRegE == rtD));

assign branchstall1 = BranchD && MemtoRegM && ((WriteRegM == rsD) || (WriteRegM == rtD));

assign branchstall12 = branchstall || branchstall1;

//assign StallD = lwstall ;

always\_comb

begin

if(ege)

begin

StallF = 1;

FlushE = 1;

StallD = 1;

end

else

begin

lwstall = ((rsD == rtE) || (rtD == rtE)) && MemToRegE;

StallD = lwstall;

StallF = lwstall;

FlushE = lwstall;

end

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

begin

ForwardAE = 2'b010;

end

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

begin

ForwardAE = 2'b001;

end

else

begin

ForwardAE = 2'b00;

end

end

always\_comb

begin

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

begin

ForwardBE = 2'b10;

end

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

begin

ForwardBE = 2'b01;

end

else

begin

ForwardBE = 2'b00;

end

end

endmodule

//This pipe passes Execution vaşues to the memory stage

module PipeEtoM(input logic clk, RegWriteE, MemtoRegE, MemWriteE, BranchE, ZeroE,

input logic [31:0] ALUOutE, WriteDataE,

input logic [5:0] WriteRegE,

input logic [31:0] PCBranchE,

output logic RegWriteM, MemtoRegM, MemWriteM, BranchM, ZeroM,

output logic [31:0] ALUOutM, WriteDataM,

output logic WriteRegM,

output logic [31:0] PCBranchM);

always\_ff @(posedge clk)

begin

RegWriteM<=RegWriteE;

MemtoRegM<=MemtoRegE;

MemWriteM<=MemWriteE;

BranchM<=BranchE;

ZeroM<=ZeroE;

ALUOutM<=ALUOutE;

WriteDataM<=WriteDataE;

WriteRegM<=WriteRegE;

PCBranchM<=PCBranchE;

end

endmodule

//This pipe passes memory value to the writeback stage

module PipeMtoW(input logic clk, RegWriteM, MemtoRegM,

input logic [31:0] ReadDataM, AluOutM,

input logic [5:0] WriteRegM,

output logic RegWriteW, MemtoRegW,

output logic [31:0]ReadDataW, AluOutW,

output logic [5:0] WriteRegW);

always\_ff @(posedge clk)

begin

RegWriteW<=RegWriteM;

MemtoRegW<=MemtoRegM;

ReadDataW<=ReadDataM;

AluOutW<=AluOutM;

WriteRegW<=WriteRegM;

end

endmodule

//This pipe passes decode values to the execution stage

module PipeDtoE(input logic FlushE, clk, RegWriteD, MemtoRegD, MemWriteD,

input logic [3:0] ALUControlD,

input logic ALUSrcD, RegDstD, BranchD,

input logic [31:0] rd1, rd2,

input logic [5:0] RsD, RtD, RdD,

input logic [31:0] SignImmD, PcPlus4D,

output logic RegWriteE, MemtoRegE, MemWriteE,

output logic [3:0] ALUControlE,

output logic ALUSrcE, RegDstE, BranchE,

output logic [31:0] rd1E, rd2E,

output logic [5:0] RsE, RtE, RdE,

output logic [31:0] SignImmE, PcPlus4E);

always\_ff @(posedge clk)

if(FlushE)

begin

RegWriteE<=RegWriteD;

MemtoRegE<=MemtoRegD;

MemWriteE<=MemWriteD;

ALUControlE<=ALUControlD;

ALUSrcE<=ALUSrcD;

RegDstE<=RegDstD;

BranchE<=BranchD;

rd1E<=rd1;

rd2E<=rd2;

RsE<=RsD;

RtE<=RtD;

RdE<=RdD;

SignImmE<=SignImmD;

PcPlus4E<=PcPlus4D;

end

endmodule

//This is 3 mux that I have write to make decisions

module mux3 #(parameter WIDTH = 8)

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

input logic [1:0] s,

output logic[WIDTH-1:0] y);

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

endmodule

Part G)

//code below is the test program without hazards

addi v0, zero, 5

addi v1, zero, 3

sw a1, 0(a2)

//its hex version

8'h00: instr = 32'h20020005;

8'h04: instr = 32'h20030003;

8'h08: instr = 32'hacc50000;

//code below

//this codes shows data hazard here writes into t0 and next instruction reads this register

RAW(Read After Write) hazard

add t0, t2, t3

and t3, t0, t1

or t5, t4, t0

sub t6,t0, t7

//its hex version

8'h00: instr = 32'h014b4020;

8'h04: instr = 32'h01095824;

8'h08: instr = 32'h01886825;

8'h0c: instr = 32'h010f7022;

//Code below is the RAW type when result stage is computed in the exeute stage of an intruction, solution with stall

lw $s0, 40($0)

and $t0, $s0, $s1

or $t1, $s4, $s0

sub $t2, $s0, $s5

//its hex version

8'h00: instr = 32'h8c100028;

8'h04: instr = 32'h02114024;

8'h08: instr = 32'h02904825;

8'h0c: instr = 32'h02155022;

// code below is the Branch prediction test program

beq $t1, $t2, 40

and $t0, $s0, $s1

or $t1, $s4, $s0

sub $t2, $s0, $s5

//its hex version

8'h00: instr = 32'h112a0027;

8'h04: instr = 32'h02114024;

8'h08: instr = 32'h02904825;

8'h0c: instr = 32'h02155022;

//compute use example program

addi $t0, $t9, 0x0016

bne $t0, $t2, 0x0004

and $t1, $a0, $s1

//its hex version

8'h00: instr = 32'h23280016;

8'h04: instr = 32'h150A0004;

8'h08: instr = 32'h00914824;

//load use program example

sw $t2, 0x0038($zero)

lw $t3, 0x0038($zero)

//its hex version

8'h00: instr = 32'hac0a0038;

8'h04: instr = 32'h8c0b0038;