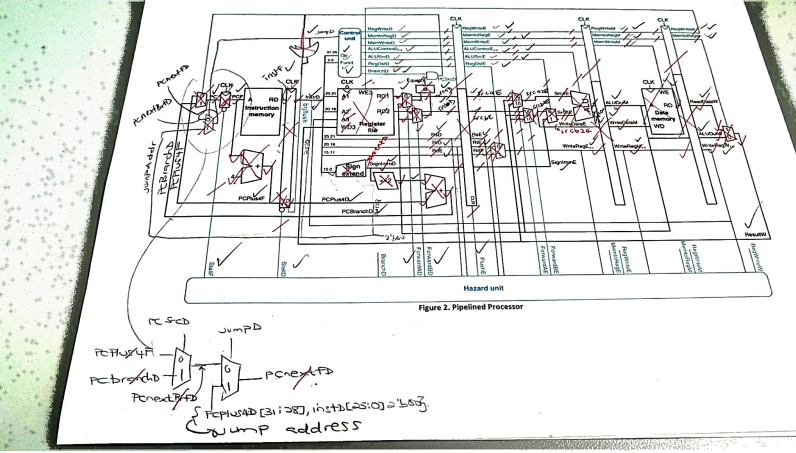
**Department of Electrical and Computer Engineering**

Author: Zainab Hussein

Title: Pipeline Processor

Date: 5-5-2017

1. Time spent: 20 hours (lab periods included)
2. Jump addition



1. Attached in last page
2. Datapath code

module pipeline\_mips(

input clk, reset,

input [31:0] instrF,

input [31:0] readdataM,

output [31:0] pcF,

output memwriteM,

output [31:0] aluoutM, writedataM);

logic [5:0] opD, functD;

logic regdstE, alusrcE,pcsrcD,

memtoregE, memtoregM, memtoregW, regwriteE, regwriteM, regwriteW;

logic [5:0] alucontrolE;

logic flushE, equalD, branchD, jumpD, zeroextendD;

controller c(clk, reset, opD, functD, flushE, equalD,

memtoregE, memtoregM, memtoregW, memwriteM, pcsrcD, branchD,

alusrcE, regdstE, regwriteE, regwriteM, regwriteW, jumpD,

zeroextendD, alucontrolE);

datapath dp(clk, reset, memtoregE, memtoregM, memtoregW, pcsrcD, branchD,

alusrcE, regdstE, regwriteE, regwriteM, regwriteW, jumpD,

zeroextendD, alucontrolE, readdataM, instrF,

equalD, pcF,

aluoutM, writedataM,

opD, functD, flushE);

endmodule

//module for the control unit of the pipeline processor

module controller( input logic clk, reset,

input logic [5:0] opD, functD,

input logic flushE, equalD,

output logic memtoregE, memtoregM, memtoregW, memwriteM,

output logic pcsrcD, branchD, alusrcE,

output logic regdstE, regwriteE, regwriteM, regwriteW,

output logic jumpD,

output logic zeroextendD,

output logic [5:0] alucontrolE);

logic [3:0] aluopD;

logic memwriteD, alusrcD,regdstD, regwriteD, memwriteE;

logic [5:0] alucontrolD;

// main decoder

maindec md( opD, memtoregD, memwriteD, branchD,alusrcD, regdstD,

regwriteD, jumpD,zeroextendD, aluopD);

// alu decoder

aludec ad(functD, aluopD, alucontrolD);

// check for branch

assign pcsrcD = branchD & equalD;

// register between Decode and Execute stages

floprc #(11) regE(clk, reset, flushE,

{memtoregD, memwriteD, alusrcD, regdstD, regwriteD, alucontrolD},

{memtoregE, memwriteE, alusrcE, regdstE, regwriteE, alucontrolE});

// register between Execute and Memory stages

flopr #(3) regM(clk, reset,

{memtoregE, memwriteE, regwriteE},

{memtoregM, memwriteM, regwriteM});

// register between Memory and Writeback stages

flopr #(2) regW(clk, reset,

{memtoregM, regwriteM},{memtoregW, regwriteW});

endmodule

//module for the DataPath of the processor

module datapath(input logic clk, reset,

input logic memtoregE, memtoregM, memtoregW,

input logic pcsrcD, branchD,

input logic alusrcE, regdstE,

input logic regwriteE, regwriteM, regwriteW,

input logic jumpD,

input logic zeroextendD,

input logic [5:0] alucontrolE,

input logic [31:0] readdataM,

input logic [31:0] instrF,

output logic equalD,

output logic [31:0] pcF,

output logic [31:0] aluoutM, writedataM,

output logic [5:0] opD, functD,

output logic flushE);

logic forwardaD, forwardbD;

logic [1:0] forwardaE, forwardbE;

logic stallF,stallD;

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

logic [4:0] writeregE, writeregM, writeregW;

logic brjflush;

logic [31:0] pcnextFD, pcnextbrFD, pcplus4F, pcbranchD;

logic [31:0] signimmD, signimmE, signimmshD;

logic [31:0] srcaD, srca2D, srcaE, srca2E;

logic [31:0] srcbD, srcb2D, srcbE, srcb2E, srcb3E;

logic [31:0] pcplus4D, instrD;

logic [31:0] aluoutE, aluoutW;

logic aluoverflowE;

logic [31:0] readdataW, resultW;

// Hazard Unit

hazard haz( rsD, rtD, rsE, rtE, writeregE, writeregM, writeregW,

regwriteE, regwriteM, regwriteW,

memtoregE, memtoregM, branchD,

forwardaD, forwardbD, forwardaE, forwardbE,

stallF, stallD, flushE);

// next PC logic

mux2 #(32) pcbrmux(pcplus4F, pcbranchD, pcsrcD, pcnextbrFD);

mux2 #(32) pcmux(pcnextbrFD,{pcplus4D[31:28], instrD[25:0], 2'b00},

jumpD, pcnextFD);

// register file -used in decode and writeback

regfile rf(clk, regwriteW, rsD, rtD, writeregW,

resultW, srcaD, srcbD);

//Fetch stage logic

flopenr #(32) pcreg(clk, reset, ~stallF, pcnextFD, pcF);

// add 4 to PC

adder pcadd1(pcF, 32'b100, pcplus4F);

// Decode stage

// Decode stage register (upper part)

flopenrc #(32) r2D(clk, reset, ~stallD, brjflush, instrF, instrD);

// Decode stage register (lower part)

flopenr #(32) r1D(clk, reset, ~stallD, pcplus4F, pcplus4D);

// sign extend immediate value

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

// shift left immediate value by 2

sl2 immsh(signimmD, signimmshD);

// add 2 to PC sign extended and shifted immediate value -branch

adder pcadd2(pcplus4D, signimmshD, pcbranchD);

// forwarding mux's

mux2 #(32) forwardadmux(srcaD, aluoutM, forwardaD, srca2D);

mux2 #(32) forwardbdmux(srcbD, aluoutM, forwardbD, srcb2D);

// branch prediction

eqcmp comp(srca2D, srcb2D, equalD);

// get op code (6 bit)

assign opD = instrD[31:26];

// get function code (6 bit)

assign functD = instrD[5:0];

// get source register (5 bit)

assign rsD = instrD[25:21];

// get target register (5 bit)

assign rtD = instrD[20:16];

// get destination register (5 bit)

assign rdD = instrD[15:11];

// flush D latch if branch or jump

assign brjflush = pcsrcD | jumpD;

//-- Execute stage ---

// latch E

floprc #(32) r1E(clk, reset, flushE, srcaD, srcaE);

floprc #(32) r2E(clk, reset, flushE, srcbD, srcbE);

floprc #(32) r3E(clk, reset, flushE, signimmD, signimmE);

floprc #(5) r4E(clk, reset, flushE, rsD, rsE);

floprc #(5) r5E(clk, reset, flushE, rtD, rtE);

floprc #(5) r6E(clk, reset, flushE, rdD, rdE);

// forwarding mux's

mux3 #(32) forwardaemux(srcaE, resultW, aluoutM, forwardaE, srca2E);

mux3 #(32) forwardbemux(srcbE, resultW, aluoutM, forwardbE, srcb2E);

// ALU B source selector

mux2 #(32) srcbmux(srcb2E, signimmE, alusrcE, srcb3E);

// ALU

Alu alu(srca2E, srcb3E, alucontrolE, aluoutE, aluoverflowE);

// Write register (rt or rd)

mux2 #(5) wrmux(rtE, rdE, regdstE, writeregE);

//-- Memory stage ---

// latch M

flopr #(32) r1M(clk, reset, srcb2E, writedataM);

flopr #(32) r2M(clk, reset, aluoutE, aluoutM);

flopr #(5) r3M(clk, reset, writeregE, writeregM);

//-- Writeback stage ---

// latch W

flopr #(32) r1W(clk, reset, aluoutM, aluoutW);

flopr #(32) r2W(clk, reset, readdataM, readdataW);

flopr #(5) r3W(clk, reset, writeregM, writeregW);

// result selector (from ALU or Memory)

mux2 #(32) resmux(aluoutW, readdataW, memtoregW, resultW);

endmodule

// adder

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

output logic [31:0] y);

assign y = a + b;

endmodule

// branch prediction comparator

module eqcmp( input logic [31:0] a, b,

output logic eq);

always\_comb

if( a==b) eq = 1;

else eq = 0;

endmodule

// shift left 2 bits

module sl2(input logic [31:0] a,

output logic [31:0] y);

// shift left 2 bits

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

endmodule

// extend sign of immediate value to 32 bits

module signext(input logic [15:0] a,

input logic zero,

output logic [31:0] y);

assign y = zero ? {16'b0, a} : {{16{a[15]}}, a};

endmodule

//latch

module flopr #(parameter WIDTH = 8)(

input logic clk, reset,

input logic [WIDTH-1:0] d,

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

always @(posedge clk, posedge reset)

if (reset) q <= 0;

else q <= d;

endmodule

// latch with clear

module floprc #(parameter WIDTH = 8)(

input logic clk, reset, clear,

input logic [WIDTH-1:0] d,

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

always @(posedge clk, posedge reset)

if (reset) q <= 0;

else if (clear) q <= 0;

else q <= d;

endmodule

// latch with enable

module flopenr #(parameter WIDTH = 8)(

input logic clk, reset,

input logic en,

input logic [WIDTH-1:0] d,

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

always @(posedge clk, posedge reset)

if (reset) q <= 0;

else if (en) q <= d;

endmodule

// latch with enable and clear

module flopenrc #(parameter WIDTH = 8)(

input logic clk, reset,

input logic en, clear,

input logic [WIDTH-1:0] d,

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

always @(posedge clk, posedge reset)

if (reset) q <= 0;

else if (clear) q <= 0;

else if (en) q <= d;

endmodule

// 2 input multiplexer

module mux2 #(parameter WIDTH = 8)(

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

input logic s,

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

assign y = s ? d1 : d0;

endmodule

// 3 input multiplexer

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] ? d2 : (s[0] ? d1 : d0);

endmodule

// 4 input multiplexer

module mux4 #(parameter WIDTH = 8)(

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

input logic [1:0] s,

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

assign y = s[2] ? d3 : (s[1] ? d2 : (s[0] ? d1 : d0));

endmodule

//main decoder

module maindec(input logic [5:0] op,

output logic memtoreg, memwrite,

output logic branch, alusrc,

output logic regdst, regwrite,

output logic jump,

output logic zeroextend,

output logic [3:0] aluop);

logic [11:0] controls;

assign {regwrite, regdst, alusrc, branch, memwrite, memtoreg,

jump, zeroextend,aluop} = controls;

always @(\*)

case(op)

6'b000000: controls = 12'b11000000\_1111; //Rtype

6'b000010: controls = 12'b00000010\_0000; //J

6'b000100: controls = 12'b00010000\_0001; //BEQ

6'b001000: controls = 12'b10100000\_0000; // ADDI

6'b001001: controls = 12'b10100000\_0000; // ADDIU

6'b001010: controls = 12'b10100000\_0010; // SLTI

6'b001100: controls = 12'b10100001\_0100; // ANDI

6'b001101: controls = 12'b10100001\_0101; // ORI

6'b001110: controls = 12'b10100001\_0110; // XORI

6'b001111: controls = 12'b10100001\_0111; // LUI

6'b100011: controls = 12'b10100100\_0000; //LW

6'b101011: controls = 12'b00101000\_0000; //SW

default: controls = 12'bxxxxxxxxxxxx; // unknown instruction

endcase

endmodule

module aludec(input logic [5:0] funct,

input logic [3:0] aluop,

output logic [5:0] alucontrol);

always @(\*)

case(aluop)

4'b0000: alucontrol = 6'b0\_00010; // ADDI

4'b0001: alucontrol = 6'b1\_00010; // SUBI

4'b0010: alucontrol = 6'b1\_00011; // SLTI

4'b0100: alucontrol = 6'b0\_00000; // ANDI

4'b0101: alucontrol = 6'b0\_00001; // ORI

4'b0110: alucontrol = 6'b0\_00100; // XORI

4'b0111: alucontrol = 6'b0\_00110; // LUI

4'b1111: case(funct) // RTYPE

6'b100000: alucontrol = 6'b0\_00010; // ADD

6'b100001: alucontrol = 6'b0\_00010; // ADDU

6'b100010: alucontrol = 6'b1\_00010; // SUB

6'b100011: alucontrol = 6'b1\_00010; // SUBU

6'b100100: alucontrol = 6'b0\_00000; // AND

6'b100101: alucontrol = 6'b0\_00001; // OR

6'b100110: alucontrol = 6'b0\_00100; // XOR

6'b100111: alucontrol = 6'b0\_00101; // NOR

6'b101010: alucontrol = 6'b1\_00011; // SLT

default: begin

alucontrol = 6'bxxxxxx; // ??? unknown function

end

endcase

endcase

endmodule

module regfile(input logic clk,

input logic we3,

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

input logic [31:0] wd3,

output logic [31:0] rd1, rd2);

logic [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\_ff @(negedge clk)

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

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

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

endmodule

// Hazard Unit

module hazard( input logic [4:0] rsD, rtD, rsE, rtE,

input logic [4:0] writeregE, writeregM, writeregW,

input logic regwriteE, regwriteM, regwriteW,

input logic memtoregE, memtoregM, branchD,

output logic forwardaD, forwardbD,

output logic [1:0] forwardaE, forwardbE,

output logic stallF, stallD, flushE);

logic lwstallD, branchstallD;

// forwarding sources to D stage (branch equality)

assign forwardaD = ((rsD != 0) & (rsD == writeregM) & regwriteM);

assign forwardbD = ((rtD != 0) & (rtD == writeregM) & regwriteM);

// forwarding sources to E stage (ALU)

always @(\*)

begin

if( (rsE != 0) & (rsE == writeregM) & regwriteM ) forwardaE = 2'b10;

else if( (rsE != 0) & (rsE == writeregW) & regwriteW ) forwardaE = 2'b01;

else forwardaE = 2'b00;

end

always @(\*)

begin

if( (rtE != 0) & (rtE == writeregM) & regwriteM ) forwardbE = 2'b10;

else if( (rtE != 0) & (rsE == writeregW) & regwriteW ) forwardbE = 2'b01;

else forwardbE = 2'b00;

end

// stalls

assign lwstallD = ( (rsD == rtE) | (rtD == rtE) ) & memtoregE;

assign branchstallD = branchD &

(regwriteE & (writeregE == rsD | writeregE == rtD) |

memtoregM & (writeregM == rsD | writeregM == rtD));

assign stallD = lwstallD | branchstallD;

assign stallF = stallD; // stalling D stalls all previous stages

assign flushE = stallD; // stalling D flushes next stage

endmodule

module top(input logic clk, reset,

output logic [31:0] writedataM, dataadr,

output logic memwriteM);

logic [31:0] readdataM, pcF, instrF;

// microprocessor (control & datapath)

pipeline\_mips mips( clk, reset, instrF,readdataM, pcF, memwriteM, dataadr, writedataM );

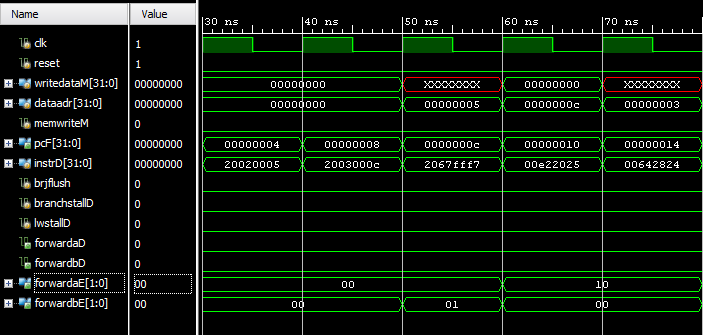
// memory

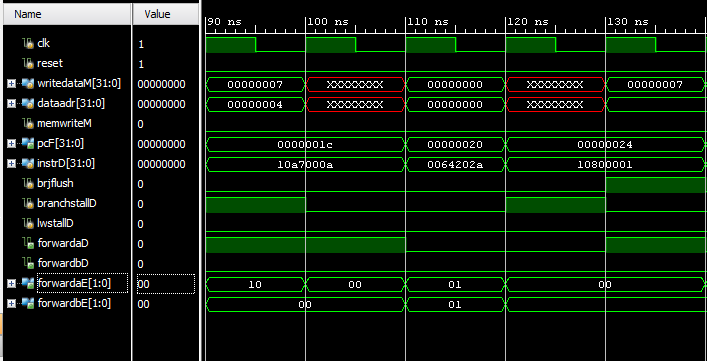
imem imem(pcF[7:2], instrF);

dmem dmem(clk, memwriteM, dataadr, writedataM, readdataM);

endmodule

1. Simulation waveforms

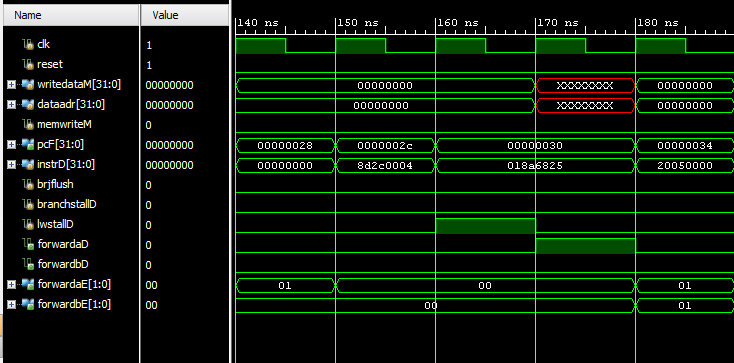




1. #cycles to complete program is 24 in testbench above, as predicted in the prelab. # instructions is 16, therefore, CPI = 24/16 = 1.5
2. Steady state CPI of program in case of a continuous loop (jump back in main) after sw. Sw does is not dependent on other registers, thus the decode to writeback stages can happen in situ with other programs, making the total number of instructions executed in 24 cycles be 20. Thus, CPI = 24/20 = 1.2
3. Assembly code and machine code added to test the forwarding and stalls. The previously untested case was a lw instruction.

|  |  |
| --- | --- |
| Assembly | Machine code |
| lw t4, 0x04(t1) | 8D2C0004 |
| or t5, t4, t2 | 018A6825 |

1. Simulation waveforms of tested forward and stalls



Stalling shown in pcF = 0x30 after the lw instruction in pcF = 0x2C. The processor stalls one stage for the $t4 from lw memory stage to be available for the or instruction. Lw stall asserted, the forwading done.

1. Pipeline is functional for all the test cases tested here.