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

// Module Name : clk\_gen

// Description : Generate 4 second and 5KHz clock cycle from

// the 50MHz clock on the Nexsys2 board

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

module clk\_gen(

input clk50MHz, reset,

output reg clksec );

reg clk\_5KHz;

integer count, count1;

always@(posedge clk50MHz) begin

if(reset) begin

count = 0;

count1 = 0;

clksec = 0;

clk\_5KHz =0;

end else begin

if (count == 500000000) begin

// Just toggle after certain number of seconds

clksec = ~clksec;

count = 0;

end

if (count1 == 20000) begin

clk\_5KHz = ~clk\_5KHz;

count1 = 0;

end

count = count + 1;

count1 = count1 + 1;

end

end

endmodule

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

// Source Code for a Single-cycle MIPS Processor (supports partial instruction)

// Developed by D. Hung, D. Herda and G. Gerken,

// based on the following source code provided by

// David\_Harris@hmc.edu (9 November 2005):

// mipstop.v

// mipsmem.v

// mips.v

// mipsparts.v

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

// Main Decoder

module maindec(

input [ 5:0] op,

output memtoreg, memwrite, branch, alusrc, regdst, regwrite, jump,

output [ 1:0] aluop,

output jalSel);

reg [ 9:0] controls;

initial

begin

controls = 0;

end

assign {jalSel, regwrite, regdst, alusrc, branch, memwrite, memtoreg, jump, aluop} = controls;

always @(\*)

case(op)

6'b000000: controls <= 10'b0110000010; //Rtype + jr

6'b100011: controls <= 10'b0101001000; //LW|regwrite|alursrc|memtoreg

6'b101011: controls <= 10'b0001010000; //SW-

6'b000100: controls <= 10'b0000100001; //BEQ

6'b001000: controls <= 10'b0101000000; //ADDI

6'b000010: controls <= 10'b0000000100; //J

6'b000011: controls <= 10'b1100000100; //JAL

default: controls <= 10'b0000000000; //???

endcase

endmodule

// ALU Decoder

module aludec(

input [5:0] funct,

input [1:0] aluop,

output reg [2:0] alucontrol,

output reg jumpreg,

output reg [1:0] regWriteSel,

output reg writeHiLoreg

);

always @(\*)

case(aluop)

2'b00: alucontrol <= 3'b010; // add

2'b01: alucontrol <= 3'b110; // sub

default: case(funct) // RTYPE

6'b100000: begin

alucontrol <= 3'b010; // ADD

jumpreg <= 1'b0;

regWriteSel <= 2'b00;

writeHiLoreg <= 1'b0;

end

6'b100010: begin

alucontrol <= 3'b110; // SUB

jumpreg <= 1'b0;

regWriteSel <= 2'b00;

writeHiLoreg <= 1'b0;

end

6'b100100: begin

alucontrol <= 3'b000; // AND

jumpreg <= 1'b0;

regWriteSel <= 2'b00;

writeHiLoreg <= 1'b0;

end

6'b100101: begin

alucontrol <= 3'b001; // OR

jumpreg <= 1'b0;

regWriteSel <= 2'b00;

writeHiLoreg <= 1'b0;

end

6'b101010: begin

alucontrol <= 3'b111; //SLT

jumpreg <= 1'b0;

regWriteSel <= 2'b00;

writeHiLoreg <= 1'b0;

end

6'b001000: begin

jumpreg <= 1'b1; // JR

alucontrol <= 3'bxxx; // ???

regWriteSel <= 2'b00;

writeHiLoreg <= 1'b0;

end

6'b011000: begin

jumpreg <= 1'b0; // MULT

alucontrol <= 3'b011;

regWriteSel <= 2'b00;

writeHiLoreg <= 1'b1;

end

6'b010000: begin

jumpreg <= 1'b0; // MFHI

alucontrol <= 3'bxxx; // ???

regWriteSel <= 2'b01;

writeHiLoreg <= 1'b0;

end

6'b010010: begin

jumpreg <= 1'b0; // MFLO

alucontrol <= 3'bxxx; // ???

regWriteSel <= 2'b10;

writeHiLoreg <= 1'b0;

end

default: begin

alucontrol <= 3'b000; // ???

jumpreg <= 1'b0;

regWriteSel <= 2'b00;

writeHiLoreg <= 2'b00;

end

endcase

endcase

endmodule

// ALU

module alu(

input [31:0] a, b,

input [ 2:0] alucont,

output reg [31:0] result,

output reg [31:0] resultExt,

output zero );

wire [31:0] sum, slt;

// wire [31:0] b2;

//assign b2 = alucont[2] ? ~b:b;

//assign sum = a + b2 + alucont[2];

assign sum = a + b;

//assign slt = sum[31];

assign slt = (a < b ? 32'h0001 : 32'h0000);

initial

begin

result <= 32'b0;

resultExt <= 32'b0;

end

always@(\*)

case(alucont[2:0]) // open remaining: 011, 100, 101, 110(sub)

3'b000: begin

result <= a & b;

resultExt <= 32'b0;

end

3'b001: begin

result <= a | b;

resultExt <= 32'b0;

end

3'b010: begin

result <= sum;

resultExt <= 32'b0;

end

3'b110: begin

result <= a-b;

resultExt <= 32'b0;

end

3'b111: begin

result <= slt;

resultExt <= 32'b0;

end

3'b011: {resultExt, result} <= a \* b; // unsure if this works

endcase

assign zero = (result == 32'b0);

endmodule

module smallReg(

input clk, writeEnable,

input [31:0] reg\_input,

output [31:0] reg\_out

);

reg [31:0] reg\_val;

initial

begin

reg\_val = 32'b0;

end

always @ (posedge clk)

begin

if( writeEnable == 1'b1 )

begin

reg\_val <= reg\_input;

end

end

assign reg\_out = (reg\_val != 0) ? reg\_val : 0;

endmodule

// Pipilened registers

module pipelineReg\_IF\_ID(

input clk,

// Datapath Inputs

input [31:0] instr\_IF,

input [31:0] pcplus4\_IF,

// Datapath Outputs

output [31:0] instr\_ID,

output [31:0] pcplus4\_ID

);

reg [31:0] instr, pc;

initial

begin

instr = 32'b0;

pc = 32'b0;

end

always @ (posedge clk)

begin

instr <= instr\_IF;

pc <= pcplus4\_IF;

end

assign instr\_ID = (instr != 0) ? instr : 0;

assign pcplus4\_ID = (pc != 0) ? pc : 0;

endmodule

module pipelineReg\_ID\_EX(

input clk,

// Control Inputs

input jalSel\_ID,

regwrite\_ID,

regdst\_ID,

alusrc\_ID,

memwrite\_ID,

memtoreg\_ID,

input [2:0] alucontrol\_ID,

input [1:0] regWriteSel\_ID,

input writeHiLoreg\_ID,

// Datapath Inputs

input [31:0] srca\_ID,

input [31:0] writedata\_ID,

input [4:0] instr\_20\_16\_ID,

input [4:0] instr\_15\_11\_ID,

input [31:0] signimm\_ID,

input [31:0] pcplus4\_ID,

// Control outputs

output jalSel\_EX,

regwrite\_EX,

regdst\_EX,

alusrc\_EX,

memwrite\_EX,

memtoreg\_EX,

output [2:0] alucontrol\_EX,

output [1:0] regWriteSel\_EX,

output writeHiLoreg\_EX,

// Datapath outputs

output [31:0] srca\_EX,

output [31:0] writedata\_EX,

output [4:0] instr\_20\_16\_EX,

output [4:0] instr\_15\_11\_EX,

output [31:0] signimm\_EX,

output [31:0] pcplus4\_EX

);

//Control regs

reg jalSel, regwrite, regdst, alusrc, memwrite, memtoreg;

reg [2:0] alucontrol, regWriteSel;

reg writeHiLoreg;

// Datapath regs

reg [31:0] srca, writedata;

reg [4:0] instr\_20\_16, instr\_15\_11;

reg [31:0] signimm, pcplus4;

initial

begin

jalSel = 0;

regwrite = 0;

regdst = 0;

alusrc = 0;

memwrite = 0;

memtoreg = 0;

alucontrol = 0;

regWriteSel = 0;

writeHiLoreg = 0;

srca = 0;

writedata = 0;

instr\_20\_16 = 0;

instr\_15\_11 = 0;

signimm = 0;

pcplus4 = 0;

end

always @ (posedge clk)

begin

jalSel <= jalSel\_ID;

regwrite <= regwrite\_ID;

regdst <= regdst\_ID;

alusrc <= alusrc\_ID;

memwrite <= memwrite\_ID;

memtoreg <= memtoreg\_ID;

alucontrol <= alucontrol\_ID;

regWriteSel <= regWriteSel\_ID;

writeHiLoreg <= writeHiLoreg\_ID;

srca <= srca\_ID;

writedata <= writedata\_ID;

instr\_20\_16 <= instr\_20\_16\_ID;

instr\_15\_11 <= instr\_15\_11\_ID;

signimm <= signimm\_ID;

pcplus4 <= pcplus4\_ID;

end

assign jalSel\_EX = (jalSel != 0) ? jalSel : 0;

assign regwrite\_EX = (regwrite != 0) ? regwrite : 0;

assign regdst\_EX = (regdst != 0) ? regdst : 0;

assign alusrc\_EX = (alusrc != 0) ? alusrc : 0;

assign memwrite\_EX = (memwrite != 0) ? memwrite : 0;

assign memtoreg\_EX = (memtoreg != 0) ? memtoreg : 0;

assign alucontrol\_EX = (alucontrol != 0) ? alucontrol : 0;

assign regWriteSel\_EX = (regWriteSel != 0) ? regWriteSel : 0;

assign writeHiLoreg\_EX = (writeHiLoreg != 0) ? writeHiLoreg : 0;

assign srca\_EX = (srca != 0) ? srca : 0;

assign writedata\_EX = (writedata != 0) ? writedata : 0;

assign instr\_20\_16\_EX = (instr\_20\_16 != 0) ? instr\_20\_16 : 0;

assign instr\_15\_11\_EX = (instr\_15\_11 != 0) ? instr\_15\_11 : 0;

assign signimm\_EX = (signimm != 0) ? signimm : 0;

assign pcplus4\_EX = (pcplus4 != 0) ? pcplus4 : 0;

endmodule

module pipelineReg\_EX\_MEM(

input clk,

// Control Inputs

input jalSel\_EX,

regwrite\_EX,

memwrite\_EX,

memtoreg\_EX,

writeHiLoreg\_EX,

input [1:0] regWriteSel\_EX,

// DataPath Inputs

input [31:0] aluout\_EX,

input [31:0] aluoutExt\_EX,

input [31:0] writedata\_EX,

input [4:0] writereg\_EX,

input [31:0] pcplus4\_EX,

// Control Ouputs

output jalSel\_MEM,

regwrite\_MEM,

memwrite\_MEM,

memtoreg\_MEM,

writeHiLoreg\_MEM,

output [1:0] regWriteSel\_MEM,

// DataPath Outputs

output [31:0] aluout\_MEM,

output [31:0] aluoutExt\_MEM,

output [31:0] writedata\_MEM,

output [4:0] writereg\_MEM,

output [31:0] pcplus4\_MEM

);

reg jalSel, regwrite, memwrite, memtoreg, writeHiLoreg;

reg [1:0] regWriteSel;

reg [31:0] aluout, aluoutExt, writedata;

reg [4:0] writereg;

reg [31:0] pcplus4;

initial

begin

jalSel = 0;

regwrite = 0;

memwrite = 0;

memtoreg = 0;

writeHiLoreg = 0;

regWriteSel = 0;

aluout = 0;

aluoutExt = 0;

writedata = 0;

writereg = 0;

pcplus4 = 0;

end

always @ (posedge clk)

begin

jalSel <= jalSel\_EX;

regwrite <= regwrite\_EX;

memwrite <= memwrite\_EX;

memtoreg <= memtoreg\_EX;

writeHiLoreg <= writeHiLoreg\_EX;

regWriteSel <= regWriteSel\_EX;

aluout <= aluout\_EX;

aluoutExt <= aluoutExt\_EX;

writedata <= writedata\_EX;

writereg <= writereg\_EX;

pcplus4 <= pcplus4\_EX;

end

assign jalSel\_MEM = (jalSel != 0) ? jalSel : 0;

assign regwrite\_MEM = (regwrite != 0) ? regwrite : 0;

assign memwrite\_MEM = (memwrite != 0) ? memwrite : 0;

assign memtoreg\_MEM = (memtoreg != 0) ? memtoreg : 0;

assign writeHiLoreg\_MEM = (writeHiLoreg != 0) ? writeHiLoreg : 0;

assign regWriteSel\_MEM = (regWriteSel != 0) ? regWriteSel : 0;

assign aluout\_MEM = (aluout != 0) ? aluout : 0;

assign aluoutExt\_MEM = (aluoutExt != 0) ? aluoutExt : 0;

assign writedata\_MEM = (writedata != 0) ? writedata : 0;

assign writereg\_MEM = (writereg != 0) ? writereg : 0;

assign pcplus4\_MEM = (pcplus4 != 0) ? pcplus4 : 0;

endmodule

module pipelineReg\_MEM\_WB(

input clk,

// Control Inputs

input jalSel\_MEM,

regwrite\_MEM,

memtoreg\_MEM,

input [1:0] regWriteSel\_MEM,

// Datapath Inputs

input [31:0] readdata\_MEM,

input [31:0] aluout\_MEM,

input [31:0] readhi\_MEM,

input [31:0] readlo\_MEM,

input [4:0] writereg\_MEM,

input [31:0] pcplus4\_MEM,

// Control Outputs

output jalSel\_WB,

regwrite\_WB,

memtoreg\_WB,

output [1:0] regWriteSel\_WB,

// Datapath Outputs

output [31:0] readdata\_WB,

output [31:0] aluout\_WB,

output [31:0] readhi\_WB,

output [31:0] readlo\_WB,

output [4:0] writereg\_WB,

output [31:0] pcplus4\_WB

);

reg jalSel, regwrite, memtoreg;

reg [1:0] regWriteSel;

reg [31:0] readdata, aluout, readhi, readlo;

reg [4:0] writereg;

reg [31:0] pcplus4;

initial

begin

jalSel = 0;

regwrite = 0;

memtoreg = 0;

regWriteSel = 0;

readdata = 0;

aluout = 0;

readhi = 0;

readlo = 0;

writereg = 0;

pcplus4 = 0;

end

always @ (posedge clk)

begin

jalSel <= jalSel\_MEM;

regwrite <= regwrite\_MEM;

memtoreg <= memtoreg\_MEM;

regWriteSel <= regWriteSel\_MEM;

readdata <= readdata\_MEM;

aluout <= aluout\_MEM;

readhi <= readhi\_MEM;

readlo <= readlo\_MEM;

writereg <= writereg\_MEM;

pcplus4 <= pcplus4\_MEM;

end

assign jalSel\_WB = (jalSel != 0) ? jalSel : 0;

assign regwrite\_WB = (regwrite != 0) ? regwrite : 0;

assign memtoreg\_WB = (memtoreg != 0) ? memtoreg : 0;

assign regWriteSel\_WB = (regWriteSel != 0) ? regWriteSel : 0;

assign readdata\_WB = (readdata != 0) ? readdata : 0;

assign aluout\_WB = (aluout != 0) ? aluout : 0;

assign readhi\_WB = (readhi != 0) ? readhi : 0;

assign readlo\_WB = (readlo != 0) ? readlo : 0;

assign writereg\_WB = (writereg != 0) ? writereg : 0;

assign pcplus4\_WB = (pcplus4 != 0) ? pcplus4 : 0;

endmodule

// Adder

module adder(

input [31:0] a, b,

output [31:0] y );

assign y = a + b;

endmodule

// Two-bit left shifter

module sl2(

input [31:0] a,

output [31:0] y );

// shift left by 2

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

endmodule

// Sign Extension Unit

module signext(

input [15:0] a,

output [31:0] y );

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

endmodule

// Parameterized Register

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

// commented out since flopenr is not used

//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

// Parameterized 2-to-1 MUX

module mux2 #(parameter WIDTH = 8) (

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

input s,

output [WIDTH-1:0] y );

assign y = s ? d1 : d0;

endmodule

// Parameterized 4-to-1 MUX

module mux4 #(parameter WIDTH = 8) (

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

input [1:0] s,

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

always @(\*)

begin

case(s)

2'b00: y = d0;

2'b01: y = d1;

2'b10: y = d2;

2'b11: y = d3;

endcase

end

endmodule

// register file with one write port and three read ports

// the 3rd read port is for prototyping dianosis

module regfile(

input clk,

input we3,

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

input [31:0] wd3,

output [31:0] rd1, rd2,

input [ 4:0] ra4,

output [31:0] rd4);

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

integer n;

//initialize registers to all 0s

initial

begin

for (n=0; n<32; n=n+1)

begin

rf[n] = 32'h00;

end

rf[n-3] = 32'h100;

end

//write first order, include logic to handle special case of $0

always @(posedge clk)

if (we3)

if (wa3 == 5'b00000);

else if (~ wa3[4])

rf[{1'b0,wa3[3:0]}] <= wd3;

else

rf[{1'b1,wa3[3:0]}] <= wd3;

// this leads to 72 warnings

//rf[wa3] <= wd3;

// this leads to 8 warnings

//if (~ wa3[4])

// rf[{0,wa3[3:0]}] <= wd3;

//else

// rf[{1,wa3[3:0]}] <= wd3;

assign rd1 = (ra1 != 0) ? rf[ra1[4:0]] : 0;

assign rd2 = (ra2 != 0) ? rf[ra2[4:0]] : 0;

assign rd4 = (ra4 != 0) ? rf[ra4[4:0]] : 0;

endmodule

// Control Unit

module controller(

input [5:0] op, funct,

input zero,

output memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, jump,

output [2:0] alucontrol,

output jalSel, jumpreg,

output [1:0] regWriteSel,

output writeHiLoreg );

wire [1:0] aluop;

wire branch;

maindec md(op, memtoreg, memwrite, branch, alusrc, regdst, regwrite, jump, aluop, jalSel); // editting

aludec ad(funct, aluop, alucontrol, jumpreg, regWriteSel, writeHiLoreg); // editting

assign pcsrc = branch & zero;

endmodule

// Data Path (excluding the instruction and data memories)

module datapath(

input clk, reset, memtoreg, pcsrc, alusrc, regdst, regwrite, memwrite, jump, jumpreg, jalSel,

input [1:0] regWriteSel,

input writeHiLoreg,

input [2:0] alucontrol,

output zero,

output [31:0] pc,

input [31:0] instr,

input [31:0] readdata,

input [ 4:0] dispSel,

output [31:0] dispDat,

output [31:0] instr\_ID,

output [31:0] aluout\_MEM,

output [31:0] writedata\_MEM,

output memwrite\_MEM );

wire [4:0] writereg;

wire [4:0] dstjalout;

wire [31:0] pcnext, pcnextbr, pcnextjr, pcplus4, pcbranch, signimm,

signimmsh, srca, srcb, result, regwritemuxout, jaloutmux, aluoutExt;

// Pipeline signals

wire [31:0] aluout, writedata;

// Pipeline\_IF\_ID

wire [31:0] pcplus4\_ID;

// Pipeline\_ID\_EX

wire [4:0] instr\_20\_16\_EX, instr\_15\_11\_EX;

wire [31:0] signimm\_EX;

// Pipeline\_EX\_MEM

wire jalSel\_EX, regwrite\_EX, memwrite\_EX, memtoreg\_EX, writeHiLoreg\_EX, regdst\_EX, alusrc\_EX;

wire [2:0] alucontrol\_EX;

wire [1:0] regWriteSel\_EX;

wire [31:0] srca\_EX, aluout\_EX, aluoutExt\_MEM, writedata\_EX;

wire [31:0] pcplus4\_EX;

// Pipeline\_MEM\_WB

wire jalSel\_MEM, regwrite\_MEM, memtoreg\_MEM, writeHiLoreg\_MEM;

wire [1:0] regWriteSel\_MEM;

wire [4:0] writereg\_MEM;

wire [31:0] pcplus4\_MEM;

wire [31:0] readhi\_MEM, readlo\_MEM;

wire jalSel\_WB, regwrite\_WB, memtoreg\_WB;

wire [1:0] regWriteSel\_WB;

wire [31:0] readdata\_WB, aluout\_WB, readhi\_WB, readlo\_WB;

wire [4:0] writereg\_WB;

wire [31:0] pcplus4\_WB;

// next PC logic

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

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

sl2 immsh(signimm, signimmsh);

adder pcadd2(pcplus4\_ID, signimmsh, pcbranch);

mux2 #(32) pcbrmux(pcnextjr, pcbranch, pcsrc, pcnextbr);

mux2 #(32) pcmux(pcnextbr, {pcplus4[31:28], instr\_ID[25:0], 2'b00}, jump, pcnext);

// editting

mux4 #(32) regwritemux(result, readhi\_WB, readlo\_WB, 0, regWriteSel\_WB, regwritemuxout);

mux2 #(32) jalmux(regwritemuxout, pcplus4\_WB, jalSel\_WB, jaloutmux); // editting

mux2 #(32) pcjrmux(pcplus4, srca, jumpreg, pcnextjr);

mux2 #(5) dstjalmux(writereg\_WB, 5'b11111, jalSel\_WB, dstjalout);

// hi, lo regs

smallReg hi(clk, writeHiLoreg\_MEM, aluoutExt\_MEM, readhi\_MEM); // done

smallReg lo(clk, writeHiLoreg\_MEM, aluout\_MEM, readlo\_MEM); // done

// end editting

// register file logic

regfile rf(clk, regwrite\_WB, instr\_ID[25:21], instr\_ID[20:16], dstjalout, jaloutmux, srca, writedata, dispSel, dispDat); // editting

mux2 #(5) wrmux(instr\_20\_16\_EX, instr\_15\_11\_EX, regdst\_EX, writereg);

mux2 #(32) resmux(aluout\_WB, readdata\_WB, memtoreg\_WB, result);

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

// ALU logic

mux2 #(32) srcbmux(writedata\_EX, signimm\_EX, alusrc\_EX, srcb);

alu alu(srca\_EX, srcb, alucontrol\_EX, aluout, aluoutExt, zero);

pipelineReg\_IF\_ID pipeline\_IF\_ID( clk, instr, pcplus4, instr\_ID, pcplus4\_ID );

pipelineReg\_ID\_EX pipeline\_ID\_EX( clk, jalSel, regwrite, regdst, alusrc, memwrite, memtoreg, alucontrol, regWriteSel, writeHiLoreg,

srca, writedata, instr\_ID[20:16], instr\_ID[15:11], signimm, pcplus4\_ID,

jalSel\_EX, regwrite\_EX, regdst\_EX, alusrc\_EX, memwrite\_EX, memtoreg\_EX, alucontrol\_EX, regWriteSel\_EX, writeHiLoreg\_EX,

srca\_EX, writedata\_EX, instr\_20\_16\_EX, instr\_15\_11\_EX, signimm\_EX, pcplus4\_EX);

pipelineReg\_EX\_MEM pipeline\_EX\_MEM(clk, jalSel\_EX, regwrite\_EX, memwrite\_EX, memtoreg\_EX, writeHiLoreg\_EX, regWriteSel\_EX,

aluout, aluoutExt, writedata\_EX, writereg, pcplus4\_EX,

jalSel\_MEM, regwrite\_MEM, memwrite\_MEM, memtoreg\_MEM, writeHiLoreg\_MEM, regWriteSel\_MEM,

aluout\_MEM, aluoutExt\_MEM, writedata\_MEM, writereg\_MEM, pcplus4\_MEM);

// Not sure whats up with the HI LO regs inputs. Both of them are getting aluout\_MEM? Thats how i have it currently, fix!

pipelineReg\_MEM\_WB pipeline\_MEM\_WB( clk, jalSel\_MEM, regwrite\_MEM, memtoreg\_MEM, regWriteSel\_MEM,

readdata, aluout\_MEM, readhi\_MEM, readlo\_MEM, writereg\_MEM, pcplus4\_MEM,

jalSel\_WB, regwrite\_WB, memtoreg\_WB, regWriteSel\_WB, readdata\_WB, aluout\_WB, readhi\_WB, readlo\_WB, writereg\_WB, pcplus4\_WB

);

// Stuff needs fixing: LO Register, output that comes from ALU and has to go through pipeline

// Also have not done anything to the MUX-es on the left-most side of the vizio diagram

endmodule

// The MIPS (excluding the instruction and data memories)

module mips(

input clk, reset,

output [31:0] pc,

input [31:0] instr,

output memwrite\_MEM,

output [31:0] aluout\_MEM, writedata\_MEM,

input [31:0] readdata,

input [ 4:0] dispSel,

output [31:0] dispDat );

// deleted wire "branch" - not used

wire memtoreg, pcsrc, zero, alusrc, regdst, regwrite, memwrite, jump, jumpreg, jalSel; // editting

wire [2:0] alucontrol;

wire [1:0] regWriteSel;

wire writeHiLoreg;

wire [31:0] instr\_ID;

//wire [31:0] aluout\_MEM;

//wire [31:0] writedata\_MEM;

controller c(instr\_ID[31:26], instr\_ID[5:0], zero,

memtoreg, memwrite, pcsrc,

alusrc, regdst, regwrite, jump,

alucontrol, jalSel, jumpreg, regWriteSel, writeHiLoreg);

datapath dp(clk, reset, memtoreg, pcsrc,

alusrc, regdst, regwrite, memwrite, jump, jumpreg, jalSel, regWriteSel, writeHiLoreg,

alucontrol, zero, pc, instr,

readdata, dispSel, dispDat,

instr\_ID, aluout\_MEM, writedata\_MEM, memwrite\_MEM);

endmodule

// Instruction Memory

module imem (

input [ 6:0] a,

output [31:0] dOut );

reg [31:0] rom[0:127];

//initialize rom from memfile\_s.dat

initial

begin

$readmemh("C:/Users/Greg/Documents/CMPE140/final lab/memfile\_s.dat", rom);

end

//simple rom

assign dOut = rom[a];

endmodule

// Data Memory

module dmem (

input clk,

input we,

input [31:0] addr,

input [31:0] dIn,

output [31:0] dOut );

reg [31:0] ram[511:0];

integer n;

//initialize ram to all FFs

initial

for (n=0; n<512; n=n+1)

ram[n] = 8'hFF;

assign dOut = ram[addr[31:2]];

always @(posedge clk)

if (we)

ram[addr[31:2]] = dIn;

endmodule

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

// mipstest.v

// David\_Harris@hmc.edu 23 October 2005

// Testbench for MIPS processor

// Used for simulation

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

module tb( );

// output herp;

reg clk;

reg reset;

// wire [31:0] writedata, dataadr;

wire [31:0] pc, instr, dataadr, writedata, readdata, dispDat;

wire clksec;

wire memwrite;

wire [ 7:0] switches;

// instantiate device to be tested

//top dut(clk, reset, writedata, dataadr, memwrite);

mips mips (clk, reset, pc, instr,

memwrite, dataadr, writedata, readdata, switches[4:0], dispDat);

imem imem (pc[8:2], instr);

dmem dmem (clk, memwrite, dataadr, writedata, readdata);

// initialize test

initial

begin

reset <= 1; # 22; reset <= 0;

end

// generate clock to sequence tests

always

begin

clk <= 1; # 5; clk <= 0; # 5;

end

endmodule

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// CMPE 140, CMPE Department, san Jose State University

// Authors: Donald Hung and Hoan Nguyen

//

// Create Date: 08:36:48 02/25/2010

// Design Name:

// Module Name: mips\_top

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments: Used for prototype

//

//////////////////////////////////////////////////////////////////////////////////

// MIPS Top-level Module (including the memories, clock,

// and the display module for prototyping)

module mips\_top(

input clk, reset,

output memwrite,

output [ 3:0] top\_an,

output [ 7:0] top\_sseg,

input [ 7:0] switches,

output sinkBit );

wire [31:0] pc, instr, dataadr, writedata, readdata, dispDat;

wire clksec;

reg [ 7:0] reg\_hex1, reg\_hex0;

// Clock (1 second) to slow down the running of the instructions

clk\_gen top\_clk(.clk50MHz(clk), .reset(reset), .clksec(clksec));

// Instantiate processor and memories

mips mips (clksec, reset, pc, instr,

memwrite, dataadr, writedata, readdata, switches[4:0], dispDat);

imem imem (pc[7:2], instr);

dmem dmem (clk, memwrite, dataadr, writedata, readdata);

// Instantiate 7-seg LED display module

disp\_hex\_mux disp\_unit (.clk(clk), .reset(1'b0),

.hex3(reg\_hex1[7:4]), .hex2(reg\_hex1[3:0]),

.hex1(reg\_hex0[7:4]), .hex0(reg\_hex0[3:0]),

.dp\_in(4'b1111), .an(top\_an), .sseg(top\_sseg));

// contents displayed on the 7 segment LEDs depending on DIP switches 7:5

// 7:5 = 000: display PC & LSB of register selected by DIP switches 4:0

// 7:5 = 001: display PC & LSB of instr

// 7:5 = 010: display PC & LSB of dataadr

// 7:5 = 011: display PC & LSB of writedata

// 7:5 = 100: display PC & instr byte 0

// 7:5 = 101: display PC & instr byte 1

// 7:5 = 110: display PC & instr byte 2

// 7:5 = 111: display PC & instr byte 3

always @ (posedge clk) begin

reg\_hex1 = pc[7:0];

case ({switches[7],switches[6], switches[5]})

3'b000: begin

reg\_hex0 = dispDat[ 7:0];

end

3'b001: begin

reg\_hex0 = instr[ 7:0];

end

3'b010: begin

reg\_hex0 = dataadr[ 7:0];

end

3'b011: begin

reg\_hex0 = writedata[ 7:0];

end

3'b100: begin

reg\_hex0 = instr[ 7:0];

end

3'b101: begin

reg\_hex0 = instr[ 15:8];

end

3'b110: begin

reg\_hex0 = instr[ 23:16];

end

3'b111: begin

reg\_hex0 = instr[ 31:24];

end

endcase

end

//sink unused bit(s) to knock down the number of warning messages

assign sinkBit = (pc > 0) ^ (instr > 0) ^ (dataadr > 0) ^ (writedata > 0) ^

(readdata > 0) ^ (dispDat > 0);

Endmodule

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

// Module Name : disp\_hex\_mux

// Description : Display the four Hex inputs on Nexys2's 7-seg LEDs

// Authors : D. Herda, D. Hung, G. Gergen

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

module disp\_hex\_mux

(

input wire clk, reset,

input wire [3:0] hex3, hex2, hex1, hex0, // hex digits

input wire [3:0] dp\_in, // 4 decimal points

output reg [3:0] an, // enable 1-out-of-4 asserted low

output reg [7:0] sseg // led segments

);

// constant declaration

// refreshing rate around 800 Hz (50 MHz/2^16)

localparam N = 18;

// internal signal declaration

reg [N-1:0] q\_reg;

wire [N-1:0] q\_next;

reg [3:0] hex\_in;

reg dp;

// N-bit counter register

always @(posedge clk, posedge reset)

if (reset)

q\_reg <= 0;

else

q\_reg <= q\_next;

// next-state logic

assign q\_next = q\_reg + 1;

// 2 MSBs of counter to control 4-to-1 multiplexing

// and to generate active-low enable signal. This

// will put the input on the vaious LEDs

always @\*

case (q\_reg[N-1:N-2])

2'b00:

begin

an = 4'b1110;

hex\_in = hex0;

dp = dp\_in[0];

end

2'b01:

begin

an = 4'b1101;

hex\_in = hex1;

dp = dp\_in[1];

end

2'b10:

begin

an = 4'b1011;

hex\_in = hex2;

dp = dp\_in[2];

end

default:

begin

an = 4'b0111;

hex\_in = hex3;

dp = dp\_in[3];

end

endcase

// hex to seven-segment led display

always @\*

begin

case(hex\_in)

4'h0: sseg[6:0] = 7'b0000001;

4'h1: sseg[6:0] = 7'b1001111;

4'h2: sseg[6:0] = 7'b0010010;

4'h3: sseg[6:0] = 7'b0000110;

4'h4: sseg[6:0] = 7'b1001100;

4'h5: sseg[6:0] = 7'b0100100;

4'h6: sseg[6:0] = 7'b0100000;

4'h7: sseg[6:0] = 7'b0001111;

4'h8: sseg[6:0] = 7'b0000000;

4'h9: sseg[6:0] = 7'b0000100;

4'ha: sseg[6:0] = 7'b0001000;

4'hb: sseg[6:0] = 7'b1100000;

4'hc: sseg[6:0] = 7'b0110001;

4'hd: sseg[6:0] = 7'b1000010;

4'he: sseg[6:0] = 7'b0110000;

default: sseg[6:0] = 7'b0111000; //4'hf

endcase

sseg[7] = dp;

end

endmodule

# Connected to Nexys onBoard USB controller

#========================================================

# clock and reset

#========================================================

NET CLK LOC = B8; # ON BOARD CLOCK WITH 50 MHz

#NET reset LOC = R17; # RESET PIN

#========================================================

# buttons & switches

#========================================================

# 4 push buttons

NET "reset" LOC = "B18";

#NET "start" LOC = "D18";

#NET "btn<2>" LOC = "E18";

#NET "btn<3>" LOC = "H13"; #btn<3> also used as reset

#========================================================

# 4-digit time-multiplexed 7-segment LED display

#========================================================

# digit enable

NET "top\_an<0>" LOC = "F17";

NET "top\_an<1>" LOC = "H17";

NET "top\_an<2>" LOC = "C18";

NET "top\_an<3>" LOC = "F15";

# 7-segment led segments

NET "top\_sseg<7>" LOC = "C17"; # dicimal point

NET "top\_sseg<6>" LOC = "L18"; # segment a

NET "top\_sseg<5>" LOC = "F18"; # segment b

NET "top\_sseg<4>" LOC = "D17"; # segment c

NET "top\_sseg<3>" LOC = "D16"; # segment d

NET "top\_sseg<2>" LOC = "G14"; # segment e

NET "top\_sseg<1>" LOC = "J17"; # segment f

NET "top\_sseg<0>" LOC = "H14"; # segment g

# 8 slide switches

NET "switches<0>" LOC = "G18";

NET "switches<1>" LOC = "H18";

NET "switches<2>" LOC = "K18";

NET "switches<3>" LOC = "K17";

NET "switches<4>" LOC = "L14";

NET "switches<5>" LOC = "L13";

NET "switches<6>" LOC = "N17";

NET "switches<7>" LOC = "R17";

#========================================================

# 8 discrete led

#========================================================

#

NET "memwrite" LOC = "J14";

#NET "load" LOC = "J15";

#NET "CS<2>" LOC = "K15";

#NET "CS<3>" LOC = "K14";

#NET "CS<4>" LOC = "E17";

#NET "CS<5>" LOC = "P15";

#NET "CS<6>" LOC = "F4";

NET "sinkBit" LOC = "R4";

#========================================================

# Timing constraint of S3 50-MHz onboard oscillator

# name of the clock signal is clk

#========================================================

#NET "clk" TNM\_NET = "clk";

#TIMESPEC "TS\_clk" = PERIOD "clk" 40 ns HIGH 50 %;