Rayan Ghaffar

Final Project Bonus Report

May 1, 2024

Description:

In this extra credit assignment, I further developed on the labs and project I had already completed. In my work on creating a CPU, in this iteration I implemented support for branching instructions like jal and handling of jumping hazards. This allows us to preview code to be ran by utilizing prediction. This is built on top of the preexisting work and forwarding I enabled in the work for the final project.

Waveforms:

A screenshot of a computer

Description automatically generated

A green lines on a black background

Description automatically generated

A computer screen shot of a keyboard

Description automatically generated

Schematic:

A green drawing of a rectangular object

Description automatically generatedA green lines on a white background

Description automatically generated

A green lines and squares

Description automatically generated with medium confidence

I/O Planning:  
A screenshot of a game

Description automatically generated

Floor planning:

A screen shot of a computer

Description automatically generated

`timescale 1ns / 1ps

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

// Company:

// Engineer:

//

// Create Date: 04/29/2024 05:40:07 PM

// Design Name:

// Module Name: CPU

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

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

module ProgramCounter(

input [31:0] nextPc,

input wpcir,

input clock,

output reg [31:0] pc

);

initial begin

pc = 32'd0; // initialize pc to 0

end

always @ (posedge clock) begin

if(wpcir) begin

pc = nextPc;

end

end

endmodule

module InstructionMemory(

input [31:0] pc,

output reg [31:0] instOut

);

//reg [31:0] memory [63:0];

reg [31:0] rom [0:63];

initial begin

rom[6'h00] = 32'h3c010000; // (00) main: lui $1, 0

rom[6'h01] = 32'h34240050; // (04) ori $4, $1, 80

rom[6'h02] = 32'h0c00001b; // (08) call: jal sum

rom[6'h03] = 32'h20050004; // (0c) dslot1: addi $5, $0, 4

rom[6'h04] = 32'hac820000; // (10) return: sw $2, 0($4)

rom[6'h05] = 32'h8c890000; // (14) lw $9, 0($4)

rom[6'h06] = 32'h01244022; // (18) sub $8, $9, $4

rom[6'h07] = 32'h20050003; // (1c) addi $5, $0, 3

rom[6'h08] = 32'h20a5ffff; // (20) loop2: addi $5, $5, -1

rom[6'h09] = 32'h34a8ffff; // (24) ori $8, $5, 0xffff

rom[6'h0a] = 32'h39085555; // (28) xori $8, $8, 0x5555

rom[6'h0b] = 32'h2009ffff; // (2c) addi $9, $0, -1

rom[6'h0c] = 32'h312affff; // (30) andi $10,$9,0xffff

rom[6'h0d] = 32'h01493025; // (34) or $6, $10, $9

rom[6'h0e] = 32'h01494026; // (38) xor $8, $10, $9

rom[6'h0f] = 32'h01463824; // (3c) and $7, $10, $6

rom[6'h10] = 32'h10a00003; // (40) beq $5, $0, shift

rom[6'h11] = 32'h00000000; // (44) dslot2: nop

rom[6'h12] = 32'h08000008; // (48) j loop2

rom[6'h13] = 32'h00000000; // (4c) dslot3: nop

rom[6'h14] = 32'h2005ffff; // (50) shift: addi $5, $0, -1

rom[6'h15] = 32'h000543c0; // (54) sll $8, $5, 15

rom[6'h16] = 32'h00084400; // (58) sll $8, $8, 16

rom[6'h17] = 32'h00084403; // (5c) sra $8, $8, 16

rom[6'h18] = 32'h000843c2; // (60) srl $8, $8, 15

rom[6'h19] = 32'h08000019; // (64) finish: j finish

rom[6'h1a] = 32'h00000000; // (68) dslot4: nop

rom[6'h1b] = 32'h00004020; // (6c) sum: add $8, $0, $0

rom[6'h1c] = 32'h8c890000; // (70) loop: lw $9, 0($4)

rom[6'h1d] = 32'h01094020; // (74) stall: add $8, $8, $9

rom[6'h1e] = 32'h20a5ffff; // (78) addi $5, $5, -1

rom[6'h1f] = 32'h14a0fffc; // (7c) bne $5, $0, loop

rom[6'h20] = 32'h20840004; // (80) dslot5: addi $4, $4, 4

rom[6'h21] = 32'h03e00008; // (84) jr $31

rom[6'h22] = 32'h00081000; // (88) dslot6: sll $2, $8, 0

end

always @(\*) begin

//instOut = memory[pc[7:2]];

instOut=rom[pc[7:2]];

end

endmodule

module PcAdder(

input [31:0] pc,

output reg [31:0] nextPc

);

reg [31:0] value = 32'h00000004; // create hard wired constant that holds the value of 4

always @(\*) begin

nextPc = pc + value; // increment pc by 4

end

endmodule

module IFIDPipeline(

input wpcir,

input [31:0] nextPc,

output reg [31:0] dpc4,

input [31:0] instOut,

input clock,

output reg [31:0] dinstOut

);

always @(posedge clock) begin

if (wpcir) begin

dinstOut = instOut;

dpc4 = nextPc;

end

end

endmodule

module npcModule(

input [1:0] pcsrc,

input [31:0] pc4,

input [31:0] bpc,

input [31:0] dqa,

input [31:0] jpc,

output reg [31:0] npc

);

always @(\*) begin

if(pcsrc==0) begin

npc = pc4;

end

if(pcsrc==1) begin

npc = bpc;

end

if(pcsrc==2) begin

npc = dqa;

end

if(pcsrc==3) begin

npc = jpc;

end

end

endmodule

module ControlUnit(

input [5:0] op, // dinstOut[31:26]

input [5:0] func, // dinstOut [5:0]

output reg wreg,

output reg m2reg,

output reg wmem,

output reg [3:0] aluc,

output reg aluimm,

output reg regrt,

// forwarding inputs

input [4:0] rs,

input [4:0] rt,

input [4:0] mdestReg,

input mm2reg,

input mwreg,

input [4:0] edestReg,

input em2reg,

input ewreg,

// forwarding outputs

output reg [1:0] fwda,

output reg [1:0] fwdb,

// extra Credit

output reg [1:0] pcsrc,

input rsrtequ,

output reg jal,

output reg shift,

output reg sext,

output reg wpcir

);

reg stall = 1'b0;

reg i\_rs = 1'b0;

reg i\_rt = 1'b0;

initial begin

wpcir = 1'b1;

pcsrc = 2'b00;

end

always @(\*) begin

// forwarding

fwda = 2'b00; // default: no hazards

if (ewreg & (edestReg != 0) & (edestReg == rs) & !em2reg) begin

fwda = 2'b01; // select exe\_alu

end else begin

if (mwreg & (mdestReg != 0) & (mdestReg == rs) & !mm2reg) begin

fwda = 2'b10; // select mem\_alu

end else begin

if (mwreg & (mdestReg != 0) & (mdestReg == rs) & mm2reg) begin

fwda = 2'b11; // select mem\_lw

end

end

end

// forward control signal for alu input b

fwdb = 2'b00; // default: no hazards

if (ewreg & (edestReg != 0) & (edestReg == rt) & !em2reg) begin

fwdb = 2'b01; // select exe\_alu

end else begin

if (mwreg & (mdestReg != 0) & (mdestReg == rt) & !mm2reg) begin

fwdb = 2'b10; // select mem\_alu

end else begin

if (mwreg & (mdestReg != 0) & (mdestReg == rt) & mm2reg) begin

fwdb = 2'b11; // select mem\_lw

end

end

end

// Branching

wpcir = 1'b1;

shift = 0;

if(op == 6'b000000) begin

i\_rs = 1'b1;

i\_rt = 1'b1;

end

if(op == 6'b100011) begin

i\_rs = 1'b1;

i\_rt = 1'b0;

end

stall = ewreg & em2reg & (edestReg!=0) & (i\_rs & (edestReg == rs) | i\_rt & (edestReg == rt));

if(stall) begin

wpcir = 1'b0;

wreg = 1'b0;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0000;

end

else begin

case (op)

6'b000000: begin //r-type

jal = 0;

pcsrc = 2'b00;

case (func)

6'b100000: begin //add

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0010;

aluimm = 1'b0;

regrt = 1'b0;

end

6'b100010: begin //sub

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0110;

aluimm = 1'b0;

regrt = 1'b0;

end

6'b100100:begin //and

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0000;

aluimm = 1'b0;

regrt = 1'b0;

end

6'b100101:begin //or

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0001;

aluimm = 1'b0;

regrt = 1'b0;

end

6'b100110:begin //xor

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0011;

aluimm = 1'b0;

regrt = 1'b0;

end

6'b000000: begin //sll

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b1010;

shift = 1'b1;

aluimm = 1'b0;

regrt = 1'b0;

end

6'b000010: begin //srl

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b1100;

shift = 1'b1;

aluimm = 1'b0;

regrt = 1'b0;

end

6'b000011: begin //sra

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b1011;

shift = 1'b1;

aluimm = 1'b0;

regrt = 1'b0;

end

6'b001000: begin //jr

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

pcsrc = 2'b10;

aluimm = 1'b0;

regrt = 1'b0;

end

endcase

end

6'b100011: begin //lw

wreg = 1;

regrt = 1;

m2reg = 1;

wmem = 0;

aluc = 4'b0010;

aluimm = 1;

sext = 1;

jal = 0;

pcsrc = 2'b00;

end

6'b101011: begin //sw

wreg = 0;

regrt = 1;

m2reg = 0;

wmem = 1;

aluc = 4'b0010;

aluimm = 1;

sext = 1;

jal = 0;

pcsrc = 2'b00;

end

6'b001000: begin //addi

wreg = 1;

regrt = 1;

m2reg = 0;

wmem = 0;

aluc = 4'b0010;

aluimm = 1;

sext = 1;

jal = 0;

pcsrc = 2'b00;

end

6'b001100: begin //andi

wreg = 1;

regrt = 1;

wmem = 0;

m2reg = 0;

aluc = 4'b0000;

aluimm = 1;

sext = 0;

jal = 0;

pcsrc = 2'b00;

end

6'b001101: begin //ori

wreg = 1;

regrt = 1;

wmem = 0;

m2reg = 0;

aluc = 4'b0001;

aluimm = 1;

sext = 0;

jal = 0;

pcsrc = 2'b00;

end

6'b001110: begin//xori

wreg = 1;

regrt = 1;

wmem = 0;

m2reg = 0;

aluc = 4'b0011;

aluimm = 1;

sext = 0;

jal = 0;

pcsrc = 2'b00;

end

6'b000100: begin//beq

wreg = 0;

m2reg = 0;

wmem = 0;

aluc = 4'b0110;

aluimm = 0;

jal = 0;

pcsrc = rsrtequ ? 2'b01 : 2'b00;

end

6'b000101: begin//bne

wreg = 0;

m2reg = 0;

wmem = 0;

aluc = 4'b0110;

aluimm = 0;

jal = 0;

pcsrc = rsrtequ ? 2'b00 : 2'b01;

end

6'b001111: begin //lui

pcsrc = 2'b00;

wreg = 1;

m2reg = 0;

wmem = 0;

aluc = 4'b0100;

aluimm = 1;

sext = 0;

jal = 0;

end

6'b000010: begin//j

wreg = 0;

m2reg = 0;

wmem = 0;

aluimm = 0;

aluc = 4'b0000;

pcsrc = 2'b11;

jal = 0;

end

6'b000011: begin//jal

m2reg = 0;

wmem = 0;

aluimm = 0;

aluc = 4'b0000;

pcsrc = 2'b11;

wreg = 1;

jal = 1;

end

endcase

end

end

endmodule

// FWDA multiplexer

module fwdaMux(

input [31:0] qa,

input [31:0] r,

input [31:0] mr,

input [31:0] mdo,

input [1:0] fwda,

output reg [31:0] fwda\_out

);

always @(\*) begin

case (fwda)

2'b00:

begin

fwda\_out = qa;

end

2'b01:

begin

fwda\_out = r;

end

2'b10:

begin

fwda\_out = mr;

end

2'b11:

begin

fwda\_out = mdo;

end

endcase

end

endmodule

// FWDB multiplexer

module fwdbMux(

input [31:0] qb,

input [31:0] r,

input [31:0] mr,

input [31:0] mdo,

input [1:0] fwdb,

output reg [31:0] fwdb\_out

);

always @(\*) begin

case (fwdb)

2'b00:

begin

fwdb\_out = qb;

end

2'b01:

begin

fwdb\_out = r;

end

2'b10:

begin

fwdb\_out = mr;

end

2'b11:

begin

fwdb\_out = mdo;

end

endcase

end

endmodule

module RegrtMux (

input [4:0] rt,

input [4:0] rd,

input regrt,

output reg [4:0] destReg

);

always @(\*) begin

case (regrt)

1'b0: destReg = rd;

1'b1: destReg = rt;

endcase

end

endmodule

module RegFile (

input [4:0] rs,

input [4:0] rt,

input [4:0] wdestReg,

input [31:0] wbData,

input wwreg,

input clk,

output reg [31:0] qa,

output reg [31:0] qb

);

reg [31:0] registers [31:0];

integer r;

initial begin

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

begin

registers[r] = 0; // initialize all registers to hold 0

end

end

always @(\*) begin

qa = registers[rs]; // load value stored in rs into qa

qb = registers[rt]; // load value stored in rt into qb

end

always @(negedge clk) begin

if (wwreg == 1) begin

registers[wdestReg] = wbData; // Write back to a register

end

end

endmodule

module ImmExt(

input [15:0] imm,

output reg [31:0] imm32,

input sext

);

always@(\*)begin

if (sext) begin

imm32={{16{imm[15]}}, imm[15:0]};

end

else begin

imm32={16'b0, imm};

end

end

endmodule

module IDEXEPipeline(

input wreg,

input m2reg,

input wmem,

input [3:0] aluc,

input aluimm,

input [4:0] destReg,

input [31:0] qa,

input [31:0] qb,

input [31:0] imm32,

input clock,

output reg ewreg,

output reg em2reg,

output reg ewmem,

output reg [3:0] ealuc,

output reg ealuimm,

output reg [4:0] edestReg,

output reg [31:0] eqa,

output reg [31:0] eqb,

output reg [31:0] eimm32,

// branching inputs

input [31:0] dpc4,

input jal,

input shift,

output reg ejal,

output reg eshift,

output reg [31:0] epc4

);

always @(posedge clock) begin

ewreg = wreg;

em2reg = m2reg;

ewmem = wmem;

ealuc = aluc;

ealuimm = aluimm;

edestReg = destReg;

eqa = qa;

eqb = qb;

eimm32 = imm32;

epc4 = dpc4;

ejal = jal;

eshift = shift;

end

endmodule

module ALU\_Mux (

input [31:0] eqb,

input [31:0] eimm32,

input ealuimm,

output reg [31:0] b

);

always @ (\*) begin

case (ealuimm)

1'b0: b = eqb; // setting b to register value

1'b1: b = eimm32; // setting b to immediate value

endcase

end

endmodule

module ALU (

input [31:0] eqa,

input [31:0] b,

input [3:0] ealuc,

output reg [31:0] r

);

always @ (\*) begin

case (ealuc)

4'b0010: r = b + eqa; // ADD

4'b0110: r = eqa - b; // SUB

4'b0000: r = eqa & b; // AND

4'b0001: r = eqa | b; // OR

4'b0011: r = eqa ^ b; // XOR

4'b1010: r = b << eqa[4:0]; // SLL

4'b1100: r = b >> eqa[4:0]; // SRL

4'b1011: r = $signed(b) >>> eqa[4:0]; // SRA

4'b0100: r = b >> 16; //{b[15:0], 16'b0};

endcase

end

endmodule

module EXEMEM\_Pipeline(

input ewreg,

input em2reg,

input ewmem,

input [31:0] ealu,

input [4:0] edestReg,

input [31:0] r,

input [31:0] eqb,

input clock,

output reg mwreg,

output reg mm2reg,

output reg mwmem,

output reg [4:0] mdestReg,

output reg [31:0] mr,

output reg [31:0] mqb

);

always @ (posedge clock) begin

mwreg = ewreg;

mm2reg = em2reg;

mwmem = ewmem;

mdestReg = edestReg;

mr = ealu; //r;

mqb = eqb;

end

endmodule

module Data\_Memory(

input [31:0] mr,

input [31:0] mqb,

input mwmem,

input clock,

output reg [31:0] mdo

);

// reg [31:0] memory [63:0];

// initial begin

// memory[0] = {32'hA00000AA};

// memory[1] = {32'h10000011};

// memory[2] = {32'h20000022};

// memory[3] = {32'h30000033};

// memory[4] = {32'h40000044};

// memory[5] = {32'h50000055};

// memory[6] = {32'h60000066};

// memory[7] = {32'h70000077};

// memory[8] = {32'h80000088};

// memory[9] = {32'h90000099};

// end

reg [31:0] ram [0:63];

integer i;

initial begin

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

ram[i] = 0;

// ram[word\_addr] = data // (byte\_addr) item in data array

ram[5'h14] = 32'h000000a3; // (50) data[0] 0 + a3 = a3

ram[5'h15] = 32'h00000027; // (54) data[1] a3 + 27 = ca

ram[5'h16] = 32'h00000079; // (58) data[2] ca + 79 = 143

ram[5'h17] = 32'h00000115; // (5c) data[3] 143 + 115 = 258

// ram[5'h18] should be 0x00000258, the sum stored by sw instruction

end

always @ (\*) begin

mdo = ram[mr >> 2]; // reading from memory

end

always @ (negedge clock) begin

case (mwmem)

0'b1: ram[mr >> 2] = mqb; // writing to memory

endcase

end

endmodule

module MEMWB\_Pipeline(

input mwreg,

input mm2reg,

input [4:0] mdestReg,

input [31:0] mr,

input [31:0] mdo,

input clock,

output reg wwreg,

output reg wm2reg,

output reg [4:0] wdestReg,

output reg [31:0] wr,

output reg [31:0] wdo

);

always @ (posedge clock) begin

wwreg = mwreg;

wm2reg = mm2reg;

wdestReg = mdestReg;

wr = mr;

wdo = mdo;

end

endmodule

module WriteBackMux(

input [31:0] wr,

input [31:0] wdo,

input wm2reg,

output reg [31:0] wbData

);

always @(\*) begin

case (wm2reg)

1'b0: wbData = wr; // set write back data to wr

1'b1: wbData = wdo; // set write back data to wdo

endcase

end

endmodule

// Branching Multiplexers

module Lextender(

input [25:0] addr,

input [31:0] dpc4,

output reg [31:0] jpc

);

always @(\*) begin

jpc = {dpc4[31:28], addr[25:0], 2'b00};

end

endmodule

module Rextender(

input [15:0] imm,

output reg [31:0]dimm

);

always@(\*) begin

dimm = {{16{imm[15]}}, imm};

dimm=dimm<<2;

end

endmodule

module bpcadd(

input [31:0] dpc4,

input [31:0] dimm,

output reg [31:0] bpc

);

always @(\*)

begin

bpc = dpc4 + dimm;

end

endmodule

module equal(

input [31:0] dqa,

input [31:0] dqb,

output reg rsrtequ

);

always @(\*)

begin

if(dqa == dqb) begin

rsrtequ = 1;

end

else begin

rsrtequ = 0;

end

end

endmodule

module exeadd(

input [31:0] epc4,

output reg [31:0] epc8

);

always @(\*)begin

epc8 = epc4 + 32'd4;

end

endmodule

module Alumux1(

input [31:0] eqa,

input [31:0] eimm32,

input eshift,

output reg [31:0] a

);

always @(\*)begin

if(eshift==1) begin

a = {27'b0, eimm32[10:6]};

end

else begin

a = eqa;

end

end

endmodule

module jalMux(

input [31:0] epc8,

input [31:0] r,

input ejal,

output reg [31:0] ealu

);

always @(\*)begin

if(ejal==1) begin

ealu = epc8;

end

else begin

ealu = r;

end

end

endmodule

module jalf(

input [4:0] edestReg,

input ejal,

output reg [4:0] edest2

);

always @(\*)begin

if(ejal==1) begin

edest2 = 5'd31;

end

else begin

edest2 = edestReg;

end

end

endmodule

module Datapath(

input clock,

output wire [31:0]pc,

output wire [31:0]dinstOut,

output wire [31:0]ealu,

output wire [31:0]mr,

output wire [31:0]wbData

);

wire [31:0] npc;

wire wreg;

wire wmem;

wire m2reg;

wire [3:0] aluc;

wire aluimm;

wire regrt;

wire [4:0] destReg;

wire [31:0] qa;

wire [31:0] qb;

wire [31:0] fwda\_out;

wire [31:0] fwdb\_out;

wire [31:0] imm32;

wire [31:0] b;

wire [31:0] a;

wire [1:0] fwda;

wire [1:0] fwdb;

wire [31:0] epc4;

wire [31:0] epc8;

wire [4:0] shamt;

wire [5:0] funct;

wire [15:0] imm;

wire [25:0] addr;

wire [31:0] dimm;

wire [31:0] instOut;

wire [31:0] r;

wire ewreg;

wire em2reg;

wire ewmem;

wire [3:0] ealuc;

wire ealuimm;

wire [4:0] edestReg;

wire mwreg;

wire mm2reg;

wire mwmem;

wire [4:0] mdestReg;

wire [31:0] mqb;

wire [31:0] mdo;

wire wwreg;

wire wm2reg;

wire [31:0] eqa;

wire [31:0] eqb;

wire [31:0] eimm32;

wire [4:0] wdestReg;

wire [31:0] wr;

wire [31:0] wdo;

wire shift;

wire rsrtequ;

wire sext;

wire ejal;

wire jal;

wire eshift;

wire [4:0] edest2;

wire [31:0] nextPc;

wire [31:0] instOut1;

wire [5:0] op;

wire [4:0] rs;

wire [4:0] rt;

wire [4:0] rd;

wire [1:0] pcsrc;

wire wpcir;

wire [31:0] bpc;

wire [31:0] jpc;

wire [31:0] pc4;

wire [31:0] dpc4;

//Lab3

ProgramCounter ProgramCounterDP(.nextPc(npc), .clock(clock), .pc(pc), .wpcir(wpcir));

InstructionMemory InstructionMemoryDP(.pc(pc), .instOut(instOut));

PcAdder PcAdderDP(.pc(pc), .nextPc(pc4));

IFIDPipeline IFIDPipelineDP(.instOut(instOut), .clock(clock), .dinstOut(dinstOut), .wpcir(wpcir), .nextPc(pc4), .dpc4(dpc4));

ControlUnit ControlUnitDP(.rs(rs),.rt(rt),.mdestReg(mdestReg),.mm2reg(mm2reg),.mwreg(mwreg),.edestReg(edestReg),.em2reg(em2reg),.ewreg(ewreg),.wpcir(wpcir), .fwda(fwda),.fwdb(fwdb),.op(op), .func(funct), .wreg(wreg), .m2reg(m2reg), .wmem(wmem),

.aluimm(aluimm),.regrt(regrt),.aluc(aluc),.pcsrc(pcsrc), .rsrtequ(rsrtequ),.sext(sext),.shift(shift),.jal(jal));

RegrtMux RegrtMuxDP(.rd(rd), .rt(rt), .regrt(regrt), .destReg(destReg));

RegFile RegFileDP(.rs(rs), .rt(rt), .qa(qa), .qb(qb), .wwreg(wwreg), .wdestReg(wdestReg), .wbData(wbData), .clk(clock));

ImmExt ImmExtDP(.imm(imm), .imm32(imm32), .sext(sext));

// Forward Multiplexers

fwdbMux fwdbMuxDP(.qb(qb), .r(r), .mr(mr), .mdo(mdo), .fwdb(fwdb), .fwdb\_out(fwdb\_out));

fwdaMux fwdaMuxDP(.qa(qa), .r(r), .mr(mr), .mdo(mdo), .fwda(fwda), .fwda\_out(fwda\_out));

IDEXEPipeline IDEXEPipeline\_DP(.clock(clock), .wreg(wreg), .m2reg(m2reg), .wmem(wmem),.aluimm(aluimm),

.aluc(aluc),.destReg(destReg),.qa(fwda\_out),.qb(fwdb\_out),.imm32(imm32),.ewreg(ewreg),.em2reg(em2reg),.ewmem(ewmem),

.ealuimm(ealuimm),.ealuc(ealuc),.edestReg(edestReg),.eqa(eqa),.eqb(eqb),.eimm32(eimm32),.dpc4(dpc4),.epc4(epc4),

.jal(jal), .ejal(ejal),.shift(shift), .eshift(eshift));

// Lab4

ALU\_Mux ALU\_Mux\_DP(.eqb(eqb), .eimm32(eimm32), .ealuimm(ealuimm), .b(b));

ALU ALU\_DP(.eqa(a), .b(b), .ealuc(ealuc), .r(r));

EXEMEM\_Pipeline EXEMEM\_Pipeline\_DP(.ewreg(ewreg), .em2reg(em2reg), .ewmem(ewmem), .edestReg(edest2), .r(r), .eqb(eqb), .clock(clock), .mwreg(mwreg), .mm2reg(mm2reg), .mwmem(mwmem), .mdestReg(mdestReg), .mr(mr), .mqb(mqb), .ealu(ealu));

Data\_Memory Data\_Memory\_DP(.mr(mr), .mqb(mqb), .mwmem(mwmem), .clock(clock), .mdo(mdo));

MEMWB\_Pipeline MEMWB\_Pipeline\_DP(.mwreg(mwreg), .mm2reg(mm2reg), .mdestReg(mdestReg), .mr(mr), .mdo(mdo), .clock(clock), .wwreg(wwreg), .wm2reg(wm2reg), .wdestReg(wdestReg), .wr(wr), .wdo(wdo));

//Lab 5

WriteBackMux WriteBackMux\_DP(.wr(wr), .wdo(wdo), .wm2reg(wm2reg), .wbData(wbData));

//Bonus

npcModule npcModuleDP(.pcsrc(pcsrc), .pc4(pc4), .bpc(bpc),.dqa(fwda\_out),.jpc(jpc),.npc(npc));

Lextender lextenderDP(.addr(addr), .dpc4(dpc4),.jpc(jpc));

Rextender rextenderDP(.imm(imm), .dimm(dimm));

bpcadd BPCaddDP(.dpc4(dpc4),.dimm(dimm),.bpc(bpc));

equal equalDP(.dqa(fwda\_out),.dqb(fwdb\_out),.rsrtequ(rsrtequ));

exeadd exeaddDP(.epc4(epc4),.epc8(epc8));

Alumux1 alumux1DP(.eqa(eqa), .eimm32(eimm32),.eshift(eshift), .a(a));

jalf jalfDP(.edestReg(edestReg),.ejal(ejal), .edest2(edest2));

jalMux jalMuxDP(.epc8(epc8),.r(r), .ejal(ejal),.ealu(ealu));

assign op = dinstOut[31:26];

assign rs = dinstOut [25:21];

assign rt = dinstOut [20:16];

assign rd = dinstOut [15:11];

assign shamt = dinstOut [10:6];

assign funct = dinstOut [5:0];

assign imm = dinstOut [15:0];

assign addr = dinstOut [25:0];

endmodule

`timescale 1ns / 1ps

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

// Company:

// Engineer:

//

// Create Date: 04/29/2024 05:40:58 PM

// Design Name:

// Module Name: testbench

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

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

module testbench();

reg clk;

wire [31:0]PC;

wire [31:0]inst;

wire [31:0]ealu;

wire [31:0] malu;

wire [31:0]wdi;

Datapath dataPath(

.clock(clk),

.pc(PC),

.dinstOut(inst),

.ealu(ealu),

.mr(malu),

.wbData(wdi)

);

initial begin

clk = 0; //initialize clock to 0

end

always begin

#10;

clk = ~clk; //set clock

end

endmodule