Title: Final lab

Class: CMPEN 331

Name: Zhengbang Fan

Abstract:

Complete implementation of 5 stage MIPS CPU Pipeline

This is a verilog implementation of a 5 stage MIPS Pipelined CPU. The five stages are fetch, decode, execute, memory, and write back. This implementation can handle most MIPS instructions with an exception of jump and branch instructions.

Introduction:

图示, 示意图

描述已自动生成

The connection is shown as the diagram above.

If stage: instruction memory, program counter adder

Id stage: control unit, forwardMUX A&B, immediate extender, regrtMux

Exe stage: alu mux, alu

Mem stage: data memory

Write back stage: regfile, WB Mux

With this architecture the cpu will be control hazard free.

a. `timescale 1ns / 1ps

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

// Company:

// Engineer: Zhengbang Fan

//

// Create Date: 11/03/2022 03:50:43 PM

// Design Name:

// Module Name: Datapath

// Project Name:

// Target Devices:

// Tool Versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

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

module ProgramCounter(next\_pc, clk, wpcir, pc);

input [31:0]next\_pc;

input clk;

input wpcir;

output pc;

reg[31:0] pc;

initial begin

pc = 100;

end

always @(posedge clk) begin

if(1) begin

pc = next\_pc;

end

end

endmodule

module InstructionMemory(pc, instOut);

input [31:0]pc;

output [31:0]instOut;

reg[31:0] instOut;

//32x64 memory

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

initial begin

//assign the instruction values in memory here (words 25, 26, 27, 28,29)

memory[25] = {6'b000000,5'd1,5'd2,5'd3,5'd0,6'b100000}; //add $3,$1,$2 rd rs rt

memory[26] = {6'b000000,5'd9,5'd3,5'd4,5'd0,6'b100010}; //sub $4,$9,$3

memory[27] = {6'b000000,5'd3,5'd9,5'd5,5'd0,6'b100101}; //or $5,$3,$9

memory[28] = {6'b000000,5'd3,5'd9,5'd6,5'd0,6'b100110}; //xor $6,$3,$9

memory[29] = {6'b000000,5'd3,5'd9,5'd7,5'd0,6'b100100}; //and $7,$3,$9

end

always @(\*) begin

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

end

endmodule

module PCAdder(pc, nextPc);

input [31:0]pc;

output [31:0]nextPc;

reg[31:0] nextPc;

always @(\*) begin

nextPc = pc + 4;

end

endmodule

module IFIDPipelineRegister(instOut, clk, wpcir, dinstOut);

input[31:0] instOut;

input clk;

input wpcir;

output[31:0] dinstOut;

reg[31:0] dinstOut;

always @(posedge clk) begin

if(1) begin

dinstOut = instOut;

end

end

endmodule

module ControlUnit(op, func, wreg, m2reg, wmem, aluc, aluimm, regrt, rs, rt, ewreg, em2reg, ern, mwreg, mm2reg, mrn, fwda, fwdb, wpcir);

input[5:0] op;

input[5:0] func;

output wreg;

output m2reg;

output wmem;

output[3:0] aluc;

output aluimm;

output regrt;

input[4:0] rs, rt;

input ewreg;

input em2reg;

input[31:0] ern;

input mwreg;

input mm2reg;

input[31:0] mrn;

output fwda;

output fwdb;

output wpcir;

reg wreg;

reg m2reg;

reg wmem;

reg[3:0] aluc;

reg aluimm;

reg regrt;

reg[31:0] fwda;

reg[31:0] fwdb;

reg wpcir;

always @(\*) begin

//do something

fwda = 2'b00;

fwdb = 2'b00;

if(ewreg == 1'b1 & ern == rs) begin

fwda = 2'b01;

end

if(ewreg == 1'b1 & ern == rt) begin

fwdb = 2'b01;

end

if(mwreg == 1'b1 & mrn == rs & ern != rs) begin

fwda = 2'b10;

end

if(mwreg == 1'b1 & mrn == rt & ern != rt) begin

fwdb = 2'b10;

end

wpcir = 1'b1;

case(op)

6'b000000:

begin

//r type instruction

case(func)

6'b100000:

begin

//add instruction

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0010;

aluimm = 1'b0;

regrt = 1'b1;

end

6'b100010:

begin

//subtract instruction

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0110;

aluimm = 1'b0;

regrt = 1'b1;

end

6'b100100:

begin

//and instruction

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0000;

aluimm = 1'b0;

regrt = 1'b1;

end

6'b100101:

begin

//or instruction

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0001;

aluimm = 1'b0;

regrt = 1'b1;

end

6'b100110:

begin

//xor instruction

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b1111;

aluimm = 1'b0;

regrt = 1'b1;

end

endcase

end

6'b100011:

begin

//load word instruction

wreg = 1'b1;

m2reg = 1'b1;

wmem = 1'b0;

aluc = 4'b0000;

aluimm = 1'b1;

regrt = 1'b0;

end

6'b101011:

begin

//store word

wreg = 1'b0;

m2reg = 1'b0;

wmem = 1'b1;

aluc = 4'b0000;

aluimm = 1'b1;

regrt = 1'b0;

end

endcase

end

endmodule

module RegRTMux(rd, rt, regrt, destReg);

input[4:0] rd, rt;

input regrt;

output[4:0] destReg;

reg[4:0] destReg;

always @(\*) begin

case(regrt)

1'b0:

begin

destReg = rd;

end

1'b1:

begin

destReg = rt;

end

endcase

end

endmodule

module FwdAMux(qa, r, mr, do, select, fwdA);

input[31:0] qa;

input[31:0] r;

input[31:0] mr;

input[31:0] do;

input[1:0] select;

output fwdA;

reg[31:0] fwdA;

always @(\*) begin

if(select == 2'b00) begin

fwdA = qa;

end

else if(select == 2'b01) begin

fwdA = r;

end

else if(select == 2'b10) begin

fwdA = mr;

end

else if(select == 2'b11) begin

fwdA = do;

end

end

endmodule

module FwdBMux(qb, r, mr, do, select, fwdB);

input[31:0] qb;

input[31:0] r;

input[31:0] mr;

input[31:0] do;

input[1:0] select;

output fwdB;

reg[31:0] fwdB;

always @(\*) begin

if(select == 2'b00) begin

fwdB = qb;

end

else if(select == 2'b01) begin

fwdB = r;

end

else if(select == 2'b10) begin

fwdB = mr;

end

else if(select == 2'b11) begin

fwdB = do;

end

end

endmodule

module RegisterFile(rs, rt, wdestReg, wbData, wwreg, clk, qa, qb);

input[4:0] rs;

input[4:0] rt;

input[4:0] wdestReg;

input[31:0] wbData;

input wwreg;

input clk;

output[31:0] qa;

output[31:0] qb;

reg[31:0] qa;

reg[31:0] qb;

//make a 32x32 register array

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

//initialize all registers to 0

integer r;

initial begin

for (r = 0; r <= 31; r = r + 1) begin

registers[r] = 0;

end

registers[0] = 32'h00000000;

registers[1] = 32'ha00000aa;

registers[2] = 32'h10000011;

registers[3] = 32'h20000022;

registers[4] = 32'h30000033;

registers[5] = 32'h40000044;

registers[6] = 32'h50000055;

registers[7] = 32'h60000066;

registers[8] = 32'h70000077;

registers[9] = 32'h80000088;

registers[10] = 32'h90000099;

end

always @(\*) begin

qa = registers[rs];

qb = registers[rt];

end

always @(negedge clk) begin

if (wwreg == 1) begin

registers[wdestReg] = wbData;

end

end

endmodule

module ImmediateExtender(imm, imm32);

input[15:0] imm;

output[31:0] imm32;

reg[31:0] imm32;

always @(\*) begin

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

end

endmodule

module IDEXEPipelineRegister(wreg, m2reg, wmem, aluc, aluimm, destReg, qa, qb, imm32, clk,

ewreg, em2reg, ewmem, ealuc, ealuimm, edestReg, eqa, eqb, eimm32);

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 clk;

output ewreg;

output em2reg;

output ewmem;

output[3:0] ealuc;

output ealuimm;

output[4:0] edestReg;

output[31:0] eqa;

output[31:0] eqb;

output[31:0] eimm32;

reg ewreg;

reg em2reg;

reg ewmem;

reg[3:0] ealuc;

reg ealuimm;

reg[4:0] edestReg;

reg[31:0] eqa;

reg[31:0] eqb;

reg[31:0] eimm32;

always @(posedge clk) begin

ewreg = wreg;

em2reg = m2reg;

ewmem = wmem;

ealuc = aluc;

ealuimm = aluimm;

edestReg = destReg;

eqa = qa;

eqb = qb;

eimm32 = imm32;

end

endmodule

module ALUMultiplexer(ealuimm, eqb, eimm32, b);

input ealuimm;

input[31:0] eqb;

input[31:0] eimm32;

output[31:0] b;

reg[31:0] b;

always @(\*) begin

case(ealuimm)

1'b0:

begin

b = eqb;

end

1'b1:

begin

b = eimm32;

end

endcase

end

endmodule

module ALU(eqa, b, ealuc, r);

input[3:0] ealuc;

input[31:0] eqa;

input[31:0] b;

output[31:0] r;

reg[31:0] r;

always @(\*)begin

case(ealuc)

4'b0000:

begin

//AND operation

r = eqa & b;

end

4'b0001:

begin

//OR operatiion

r = eqa | b;

end

4'b0010:

begin

//add operation

r = eqa + b;

end

4'b0110:

begin

//subtract operation

r = eqa - b;

end

4'b1100:

begin

//NOR operation

r = ~(eqa | b);

end

4'b1111:

begin

//XOR operation

r = eqa ^ b;

end

endcase

end

endmodule

module EXEMEMPipelineRegister(clk, ewreg, em2reg, ewmem, edestReg, r, eqb, mwreg, mm2reg,

mwmem, mdestReg, mr, mqb);

input clk;

input ewreg;

input em2reg;

input ewmem;

input[4:0] edestReg;

input[31:0] r;

input[31:0] eqb;

output mwreg;

output mm2reg;

output mwmem;

output[4:0] mdestReg;

output[31:0] mr;

output[31:0] mqb;

reg mwreg;

reg mm2reg;

reg mwmem;

reg[4:0] mdestReg;

reg[31:0] mr;

reg[31:0] mqb;

always @(posedge clk) begin

mwreg = ewreg;

mm2reg = em2reg;

mwmem = ewmem;

mdestReg = edestReg;

mr = r;

mqb = eqb;

end

endmodule

module DataMemory(clk, mr, mqb, mwmem, mdo);

input clk;

input[31:0] mr;

input[31:0] mqb;

input mwmem;

output[31:0] mdo;

reg[31:0] mdo;

//make the data memory array

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

initial begin

//assign the values in data memory here (words 0-9)

dataMemory[0] = 32'ha00000aa;

dataMemory[1] = 32'h10000011;

dataMemory[2] = 32'h20000022;

dataMemory[3] = 32'h30000033;

dataMemory[4] = 32'h40000044;

dataMemory[5] = 32'h50000055;

dataMemory[6] = 32'h60000066;

dataMemory[7] = 32'h70000077;

dataMemory[8] = 32'h80000088;

dataMemory[9] = 32'h90000099;

end

always @(\*)begin

//reading

mdo = dataMemory[mr];

end

always @(negedge clk)begin

//writing

if(mwmem == 1) begin

dataMemory[mr] = mqb;

end

end

endmodule

module MEMWBPipelineRegister(clk, mwreg, mm2reg, mdestReg, mr, mdo, wwreg, wm2reg, wdestReg, wr, wdo);

input clk;

input mwreg;

input mm2reg;

input[4:0] mdestReg;

input[31:0] mr;

input[31:0] mdo;

output wwreg;

output wm2reg;

output[4:0] wdestReg;

output[31:0] wr;

output[31:0] wdo;

reg wwreg;

reg wm2reg;

reg[4:0] wdestReg;

reg[31:0] wr;

reg[31:0] wdo;

always @(posedge clk) begin

wwreg = mwreg;

wm2reg = mm2reg;

wdestReg = mdestReg;

wr = mr;

wdo = mdo;

end

endmodule

module WBMultiplexer(wr, wdo, wm2reg, wbData);

input[31:0] wr;

input[31:0] wdo;

input wm2reg;

output reg[31:0] wbData;

always @(\*) begin

if(wm2reg == 0) begin

wbData = wr;

end

else begin

wbData = wdo;

end

end

endmodule

module Datapath(clk, pc,qa,qb, dinstOut, ewreg, em2reg , ewmem, ealuc, ealuimm, edestReg, eqa, eqb, eimm32,

b, r, mwreg, mm2reg, mwmem, mdestReg, mr, mqb, mdo, wwreg, wm2reg, wdestReg, wr, wdo, wbData);

input clk;

output wire[31:0] pc;

wire[31:0] next\_pc;

wire[31:0] instOut;

output wire[31:0] dinstOut;

wire wreg;

wire m2reg;

wire wmem;

wire[3:0] aluc;

wire aluimm;

wire regrt;

wire[4:0] destReg;

output wire[31:0] qa;

output wire[31:0] qb;

wire[31:0] imm32;

//outputs for id/exe pipeline register

output wire ewreg;

output wire em2reg;

output wire ewmem;

output wire[3:0] ealuc;

output wire ealuimm;

output wire[4:0] edestReg;

output wire[31:0] eqa;

output wire[31:0] eqb;

output wire[31:0] eimm32;

//outputs for execute stage

output wire[31:0] b;

output wire[31:0] r;

//outputs for exe/mem

output wire mwreg;

output wire mm2reg;

output wire mwmem;

output wire[4:0] mdestReg;

output wire[31:0] mr;

output wire[31:0] mqb;

//outputs for mem stage

output wire[31:0] mdo;

output wire wwreg;

output wire wm2reg;

output wire[4:0] wdestReg;

output wire[31:0] wr;

output wire[31:0] wdo;

//output for wb stage

output wire[31:0] wbData;

//wires added for final project

wire wpcir;

wire[1:0] fwda;

wire[1:0] fwdb;

wire[31:0] so;

wire[31:0] muxaOut;

wire[31:0] muxbOut;

PCAdder pcAdder(pc, next\_pc);

ProgramCounter programCounter(next\_pc, clk, wpcir, pc);

InstructionMemory instructionMemory(pc, instOut);

IFIDPipelineRegister ifIdPipelineRegister(instOut, clk, wpcir, dinstOut);

ControlUnit controlUnit(dinstOut[31:26], dinstOut[5:0], wreg, m2reg, wmem, aluc, aluimm, regrt,

dinstOut[25:21], dinstOut[20:16], ewreg, em2reg, edestReg, mwreg, mm2reg, mdestReg, fwda, fwdb);

RegRTMux regRTMux(dinstOut[20:16], dinstOut[15:11], regrt, destReg);

RegisterFile registerFile(dinstOut[25:21], dinstOut[20:16], wdestReg, wbData, wwreg, clk, qa, qb);

FwdAMux fwdaMux(qa, r, mr, so, fwda, muxaOut);

FwdBMux fwdbMux(qb, r, mr, so, fwdb, muxbOut);

ImmediateExtender immediateExtender(dinstOut[15:0],imm32);

IDEXEPipelineRegister idExePipelineRegister(wreg, m2reg, wmem, aluc, aluimm, destReg, muxaOut, muxbOut, imm32, clk,

ewreg, em2reg, ewmem, ealuc, ealuimm, edestReg, eqa, eqb, eimm32);

ALUMultiplexer aluMultiplexer(ealuimm, eqb, eimm32, b);

ALU alu(eqa, b, ealuc, r);

EXEMEMPipelineRegister exeMemPipelineRegister(clk, ewreg, em2reg, ewmem, edestReg, r, eqb, mwreg, mm2reg,

mwmem, mdestReg, mr, mqb);

DataMemory dataMemory(clk, mr, mqb, mwmem, mdo);

MEMWBPipelineRegister memWbPipelineRegister(clk, mwreg, mm2reg, mdestReg, mr, mdo, wwreg, wm2reg,

wdestReg, wr, wdo);

WBMultiplexer wbMultiplexer(wr, wdo, wm2reg, wbData);

endmodule

b. `timescale 1ns / 1ps

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

// Company:

// Engineer: Zhengbang Fan

//

// Create Date: 11/03/2022 03:50:43 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] dinstOut;

wire[31:0] qa;

wire[31:0] qb;

wire ewreg;

wire em2reg;

wire ewmem;

wire[3:0] ealuc;

wire ealuimm;

wire[4:0] edestReg;

wire[31:0] eqa;

wire[31:0] eqb;

wire[31:0] eimm32;

wire[31:0] b;

wire[31:0] r;

wire mwreg;

wire mm2reg;

wire mwmem;

wire[4:0] mdestReg;

wire[31:0] mr;

wire[31:0] mqb;

wire[31:0] mdo;

wire wwreg;

wire wm2reg;

wire[4:0] wdestReg;

wire[31:0] wr;

wire[31:0] wdo;

wire[31:0] wbData;

Datapath datapath(clk, pc, dinstOut,qa,qb, ewreg, em2reg , ewmem, ealuc, ealuimm, edestReg, eqa, eqb, eimm32,

b, r, mwreg, mm2reg, mwmem, mdestReg, mr, mqb, mdo, wwreg, wm2reg, wdestReg, wr, wdo, wbData);

initial begin

clk <= 1'b0;

end

always begin

#5;

clk = ~clk;

end

endmodule

c. 日程表

描述已自动生成

d. 图形用户界面

描述已自动生成

e. QR 代码

描述已自动生成

f. 电脑萤幕画面

描述已自动生成