Project Report

CMPEN-331 Fall 2022

Section 001

Dohyoung Ko

Penn State ID: 927713012

Abstract

In this project, I implemented the design of the 5 stages pipelined CPU based on MIPS instruction set architecture, using the Xilinx design package for FPGAs. In addition to previous labs, I added another feature of the processor where it detects data hazards and resolves them by pipeline stalls and full forwarding. It also supports all R-type and I-type instructions except for shift, branch, and jr.

Introduction

The processor uses pipelining technique to execute a set of instructions much faster. Our processor, with 5-stage pipelined CPU, can execute at most 5 instructions at a time. This is done by saving temporary results in each pipeline stage into corresponding pipeline registers and use those stored data when the instruction moves to the next stage. Our processor has mainly 5 stages: Instruction Fetch (IF), Instruction Decode (ID), Execution (EXE), Memory Access (MEM), and Write Back (WB). The program counter register (pc) serves as the first pipeline register, and other 4 registers are named IF/ID, ID/EXE, EXE/MEM, and MEM/WB as shown in *Figure 1*.

For example, suppose that we have a sequence of instructions A – B – C – D, to be executed sequentially, and instruction A first goes through the IF stage. As soon as A finishes IF stage, the output from ID stage is stored in IF/ID register and the next instruction B enters the IF stage. Since we lose information from resetting/reusing the component, pipeline registers are required to hold information produced in previous cycle and feed it to the next stage. Similarly, if A is in MEM stage, B is in EXE, C is in ID, and D is in IF stage respectively. An instruction exits the CPU Datapath as it finishes the WB stage. This way, the CPU can execute 5 instructions at a single clock cycle. Under ideal conditions and with a large number of instructions, our CPU is approximately 5 times faster than a single cycled CPU. This is an advantage by parallelism.

Diagram, schematic

Description automatically generated

Figure : 5-stages pipelined CPU

In addition to pipelining, the designed processor also supports hazard detection, pipeline stalls, and full forwarding. When there are dependencies between several instructions where next instruction uses data before it is ready from previous instruction, there exists a data hazard. In this case, CPU needs to wait for previous instruction to complete its data read/write procedure by using result as soon as it is computed or stalling the pipeline. In our implementation, data hazards are resolved by additional input/output signals from the control unit and two multiplexors (fqaMux, fqbMux), as shown in *Figure 2* schematics.

Diagram, schematic

Description automatically generated

Figure Enhanced CPU with Forwarding and Hazard Detection

There are 3 types of data hazards.

**I. EX hazard**

EX hazard happens when the destination register of the previous instruction is equal to either rs or rt of current instruction in ID stage. In this case, we forward from the output of ALU to the input of ALU to resolve this hazard, setting fwda and fwdb signal to 1.

**II. MEM hazard**

MEM hazard happens when the destination register of the second previous instruction is equal to either rs or rt of current instruction in ID stage. In this case, we forward from the MEM/WB register to the input of ALU to resolve this hazard, setting fwda and fwdb signal to 2.

**III. lw-use hazard**

lw-use hazard happens when the previous instruction is load word(lw) and its destination register (rt) is equal to either rs or rt of current instruction in ID stage. In this case, pipeline stall is necessary. Hence, we stall 1 time and forward from the output of memory to the input of ALU to resolve this hazard, setting fwda or fwdb signal to 3.

Given a stall signal from the control unit, pipeline stall is implemented by prohibiting the updates of the program counter and the content of IF/ID pipeline register and cancelling the first instruction. This stall signal becomes true only if either one of source registers, rs or rt, of an instruction right after the load word instruction is equivalent to the destination register rt of lw instruction.

If the processor needs to stall, the control unit sets the write enable signal (wpcir) to 0 to prevent the updates of pc and IF/ID Reg. Also, it sets all wreg, m2reg, wmem signals to 0 to prevent the instruction from updating the states of the registers and memory.

This forwarding architecture along with hazard detection is beneficial in terms of performance because the CPU does not have to wait until the data is finally ready at WB stage. Instead, it can use data in subsequent instructions as soon as it is available. This way, a CPU have a smaller number of stalls and finish the same number of instructions in fewer number of clock cycles even if the program has a lot of dependencies in between instructions.

The implementation of this processor is verified by the following set of MIPS instructions.

1 lw $9, 4($1)

2 add $8, $8, $9

3 sub $4, $9, $8

4 or $5, $8, $9

5 xor $6, $8, $9

6 and $7, $8, $9

**Verilog Design Code**

`timescale 1ns / 1ps

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

// Company: Penn State College of Engineering

// Engineer: Dohyoung Ko

// Create Date: 12/17/2022 05:26:09 AM

// Design Name: Final Project

// Module Name: datapath

// Project Name: CMPEN-331 FA22

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

module progCounter(

input [31:0] nextPc, // pre-determined next program counter

input clk, // clock signal

input wpcir, // program counter write enable

output reg [31:0] pc // updated program counter

);

initial begin

pc <= 100; // initial program counter = 100

end

always@(posedge clk) begin // sequential element

if (wpcir) begin // no stall

pc <= nextPc;

end

else begin // stall

pc <= pc;

end

end

endmodule

module instMemory(

input [31:0] pc, // address of fetching instruction

output reg [31:0] instOut // fetched 32-bit instruction

);

// internal components

reg [31:0] memory [0:63]; // 64 slots of 32-bit wide instruction memory

initial begin // initialize instruction memory with test instructions

memory[25] = { // address 100: lw $9, 4($1)

6'b100011, // opcode = lw

5'b00001, // rs = $1

5'b01001, // rt = $9

16'b0000000000000100 // offset = 0

};

memory[26] = { // address 104: add $8, $8, $9

6'b000000, // opcode = R-type

5'b01000, // rs = $8

5'b01001, // rt = $9

5'b01000, // rd = $8

5'b00000, // shamt = 0

6'b100000 // func = add

};

memory[27] = { // address 108: sub $4, $9, $8

6'b000000, // opcode = R-type

5'b01001, // rs = $9

5'b01000, // rt = $8

5'b00100, // rd = $4

5'b00000, // shamt = 0

6'b100010 // func = sub

};

memory[28] = { // address 112: or $5, $8, $9

6'b000000, // opcode = R-type

5'b01000, // rs = $8

5'b01001, // rt = $9

5'b00101, // rd = $5

5'b00000, // shamt = 0

6'b100101 // func = or

};

memory[29] = { // address 116: xor $6, $8, $9

6'b000000, // opcode = R-type

5'b01000, // rs = $8

5'b01001, // rt = $9

5'b00110, // rd = $6

5'b00000, // shamt = 0

6'b100110 // func = xor

};

memory[30] = { // address 120: and $7, $8, $9

6'b000000, // opcode = R-type

5'b01000, // rs = $8

5'b01001, // rt = $9

5'b00111, // rd = $7

5'b00000, // shamt = 0

6'b100100 // func = and

};

end

always@(\*) begin // asynchronous read

instOut = memory[pc[31:2]]; // pc[31:0] is the starting address of target instruction

end

endmodule

module pcAdder(

input [31:0] pc,

output reg [31:0] nextPc

);

always@(\*) begin // next instruction = program counter + 4

nextPc = pc + 4;

end

endmodule

module IFIDreg(

input [31:0] instOut,

input clk,

input wpcir, // program counter write enable

output reg [31:0] dinstOut

);

always@(posedge clk) begin

if (wpcir) begin // no pipeline stall

dinstOut <= instOut; // fetch next instruction

end

else begin // pipeline stall

dinstOut <= dinstOut; // fetch same instruction

end

end

endmodule

module controlUnit(

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

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

input [4:0] rs, // inst[25:21] : used to determine hazard

input [4:0] rt, // inst[20:16] : used to determine hazard

input [4:0] ern, // EXE/MEM destReg

input em2reg, // EXE/MEM load from memory to register

input ewreg, // EXE/MEM write to register

input [4:0] mrn, // MEM/WB destReg

input mm2reg, // MEM/WB load from memory to register

input mwreg, // MEM/WB write to register

output reg wreg, // write to register

output reg m2reg, // load from memory to register

output reg wmem, // write to memory

output reg [3:0] aluc, // ALU control signal

output reg aluimm, // ALU immediate selector signal

output reg regrt, // register rt (result)

output reg wpcir, // Write enable for the PC and IF/ID reg

output reg [1:0] fwdb, // register B forwarding signal

output reg [1:0] fwda // register A forwarding signal

);

reg i\_rt; // instruction uses rt

reg i\_rs; // instruction uses rs

reg stall; // pipeline stall signal

initial begin

wpcir = 1;

end

always@(\*) begin // hazard detection unit

i\_rt = ((rt == ern) | (rt == mrn));

i\_rs = ((rs == ern) | (rs == mrn));

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

if (stall) begin

wpcir = 0;

end else begin

wpcir = 1;

end

end

always@(\*) begin // forwarding unit

if (stall) begin

fwda = 2'b00;

fwdb = 2'b00;

end

else begin

// determine forwarding signal for ALU input A

if ((ewreg) && (ern != 0) && (ern == rs)) begin

// EXE hazard: forward from ALU

fwda = 2'b01;

end else if ((mwreg) && (mm2reg) && (mrn != 0) && (mrn == rs)) begin

// lw-use hazard: forward from memory output after 1 stall

fwda = 2'b11;

end else if ((mwreg) && (mrn != 0) && !(ewreg && (ern != 0) && (ern == rs)) && (mrn == rs)) begin

// MEM hazard: forward from exe/mem register

fwda = 2'b10;

end else begin

// no forwarding needed

fwda = 2'b00;

end

// determine forwarding signal for ALU input B

if ((ewreg) && (ern != 0) && (ern == rt)) begin

// EXE hazard: forward from ALU

fwdb = 2'b01;

end else if ((mwreg) && (mm2reg) && (mrn != 0) && (mrn == rt)) begin

// lw-use hazard: forward from memory output after 1 stall

fwdb = 2'b11;

end else if ((mwreg) && (mrn != 0) && !(ewreg && (ern != 0) && (ern == rt)) && (mrn == rt)) begin

// MEM hazard: foward from exe/mem register

fwdb = 2'b10;

end else begin

// no forwarding needed

fwdb = 2'b00;

end

end

end

always@(\*) begin // determine control signal

if (stall) begin // cancel current instruction (NOP)

aluc = 4'b0000; // don't care

wreg = 0; // not writing to register

m2reg = 0; // not loading from memory

wmem = 0; // not writing to memory

aluimm = 0; // don't care

end

else case (op) // no stall: regular control signal

6'b000000: // R-type instruction

begin

case (func) // based on function code in R-type instructions

6'b100000: // ADD

begin // signals for ADD instruction

aluc = 4'b0010; // ALU = add(2)

wreg = 1; // write to register

m2reg = 0; // not loading from memory to register

wmem = 0; // not writing to memory

aluimm = 0; // R-type: not immediate

regrt = 0; // write to register rd

end

6'b100010: // SUB

begin // signals for SUB instruction

aluc = 4'b0110; // ALU = subtract(6)

wreg = 1; // write to register

m2reg = 0; // not loading from memory to register

wmem = 0; // not writing to memory

aluimm = 0; // R-type: not immediate

regrt = 0; // write to register rd

end

6'b100100: // AND

begin // signals for AND instruction

aluc = 4'b0000; // ALU = and(0)

wreg = 1; // write to register

m2reg = 0; // not loading from memory to register

wmem = 0; // not writing to memory

aluimm = 0; // R-type: not immediate

regrt = 0; // write to register rd

end

6'b100101: // OR

begin // signals for OR instruction

aluc = 4'b0001; // ALU = or(1)

wreg = 1; // write to register

m2reg = 0; // not loading from memory to register

wmem = 0; // not writing to memory

aluimm = 0; // R-type: not immediate

regrt = 0; // write to register rd

end

6'b101010: // Set on Less Than

begin // signals for set-on-less-than instruction

aluc = 4'b0111; // ALU = set-on-less-than(7)

wreg = 1; // write to register

m2reg = 0; // not loading from memory to register

wmem = 0; // not writing to memory

aluimm = 0; // R-type: not immediate

regrt = 0; // write to register rd

end

6'b100110: // XOR

begin

aluc = 4'b0011; // ALU = xor(3)

wreg = 1; // write to register

m2reg = 0; // not loading from memory to register

wmem = 0; // not writing to memory

aluimm = 0; // R-type: not immediate

regrt = 0; // write to register rd

end

endcase

end

6'b100011: // Load Word

begin // signals for LW instruction

aluc = 4'b0010; // ALU = add(2)

wreg = 1; // write to register

m2reg = 1; // load from memory to register

wmem = 0; // not writing to memory

aluimm = 1; // use sign-extended constant (offset)

regrt = 1; // write to register rt

end

6'b101011: // Store Word

begin

aluc = 4'b0010; // ALU = add(2)

wreg = 0; // not writing to register

m2reg = 0; // not loading from memory to register

wmem = 1; // write to memory

aluimm = 1; // use sign-extended constant (offset)

regrt = 0; // don't care

end

6'b001000: // Add Immediate

begin

aluc = 4'b0010; // ALU = add(2)

wreg = 1; // write to register

m2reg = 0; // not loading from memory to register

wmem = 0; // not writing to memory

aluimm = 1; // use sign-extended constant

regrt = 1; // write to register rd

end

6'b001100: // AND Immediate

begin

aluc = 4'b0000; // ALU = and(0)

wreg = 1; // write to register

m2reg = 0; // not loading from memory to register

wmem = 0; // not writing to memory

aluimm = 1; // use sign-extended constant

regrt = 1; // write to register rd

end

6'b001101: // OR Immediate

begin

aluc = 4'b0001; // ALU = or(1)

wreg = 1; // write to register

m2reg = 0; // not loading from memory to register

wmem = 0; // not writing to memory

aluimm = 1; // use sign-extended constant

regrt = 1; // write to register rd

end

6'b001110: // XOR Immediate

begin

aluc = 4'b0011; // ALU = xor(3)

wreg = 1; // write to register

m2reg = 0; // not loading from memory to register

wmem = 0; // not writing to memory

aluimm = 1; // use sign-extended constant

regrt = 1; // write to register rd

end

endcase

end

endmodule

module regRt(

input [4:0] rt, // dinstOut[20:16]

input [4:0] rd, // dinstOut[15:11]

input regrt,

output reg [4:0] destReg

);

always@(\*) begin

if (regrt == 0) begin

destReg = rd;

end

else begin

destReg = rt;

end

end

endmodule

module fqaMux(

input [1:0] fwda, // selector bit

input [31:0] qa, // qa from register file

input [31:0] r, // forward from ALU output

input [31:0] mr, // forward from EXE/MEM

input [31:0] mdo, // forward from Memory output

output reg [31:0] da // output da

);

always@(\*) begin

if (fwda == 2'b00) begin

da = qa; // forward from register file

end

else if (fwda == 2'b01) begin

da = r; // forward from ALU output

end

else if (fwda == 2'b10) begin

da = mr; // forward from EXE/MEM register

end

else if (fwda == 2'b11) begin

da = mdo; // forward from Memory output

end

end

endmodule

module fqbMux(

input [1:0] fwdb, // selector bit

input [31:0] qb, // qb from register file

input [31:0] r, // forward from ALU output

input [31:0] mr, // forward from EXE/MEM

input [31:0] mdo, // forward from Memory output

output reg [31:0] db // output db

);

always@(\*) begin

if (fwdb == 2'b00) begin

db = qb; // return from register file

end

if (fwdb == 2'b01) begin

db = r; // forward from ALU output

end

if (fwdb == 2'b10) begin

db = mr; // forward from EXE/MEM

end

if (fwdb == 2'b11) begin

db = mdo; // forward from Memory output

end

end

endmodule

module regFile(

input [4:0] rs, // dinstOut[25:21]

input [4:0] rt, // dinstOut[20:16]

input [4:0] wdestReg, // destination register to which the data is written

input [31:0] wbData, // data writing to the register

input wwreg, // control signal: enable write to register

input clk, // clock signal

output reg [31:0] qa, // var for $rs content

output reg [31:0] qb // var for $rt content

);

// internal components

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

integer i; // variable for loop

initial begin

// zero register

registers[0] = 32'h00000000;

// initialize first 11 registers

registers[1] = 32'h00000000;

registers[2] = 32'ha00000aa;

registers[3] = 32'h10000011;

registers[4] = 32'h20000022;

registers[5] = 32'h30000033;

registers[6] = 32'h40000044;

registers[7] = 32'h50000055;

registers[8] = 32'h60000066;

registers[9] = 32'h70000077;

registers[10] = 32'h80000088;

registers[11] = 32'h90000099;

// initialize rest of the registers to 0

for (i=11; i<31; i=i+1) begin

registers[i] <= 32'h0000;

end

end

always@(\*) begin

qa = registers[rs]; // read from register $rs

qb = registers[rt]; // read from register $rt

end

always@(negedge clk) begin // synchronous write to register

if (wwreg == 1)

registers[wdestReg] <= wbData;

end

endmodule

module immExtender( // sign extend 16-bit to 32-bit

input [15:0] imm, // original imm

output reg [31:0] imm32 // sign extended imm

);

always@(\*) begin

imm32={{imm[15]\*16},imm}; // concatenate first 16 MSBs with sign bit

end

endmodule

module IDEXEreg( // ID/EXE pipeline register

input clk,

// end of ID stage

input wreg,

input m2reg,

input wmem,

input [3:0] aluc,

input aluimm,

input [4:0] destReg,

input [31:0] da,

input [31:0] db,

input [31:0] imm32,

// beginning of EXE stage

output reg ewreg,

output reg em2reg,

output reg ewmem,

output reg [3:0] ealuc,

output reg ealuimm,

output reg [4:0] edestReg, // ID/EXE.RegisterRd

output reg [31:0] eqa,

output reg [31:0] eqb,

output reg [31:0] eimm32

);

always@(posedge clk) begin // assign ID -> EXE

ewreg <= wreg;

em2reg <= m2reg;

ewmem <= wmem;

ealuc <= aluc;

ealuimm <= aluimm;

edestReg <= destReg;

eqa <= da;

eqb <= db;

eimm32 <= imm32;

end

endmodule

module AluMux( // select ALU input B

input [31:0] eqb, // register output

input [31:0] eimm32, // sign extension

input ealuimm, // selector bit

output reg [31:0] b // selected output

);

always@(\*) begin

if (ealuimm == 0) begin // select register value

b = eqb;

end

else begin // select sign extended value

b = eimm32;

end

end

endmodule

module Alu( // combinational element

input [31:0] eqa, // 1st operand

input [31:0] b, // 2nd operand

input [3:0] ealuc, // alu control: which operation to execute

output reg [31:0] r // arithmetic output

);

always@(\*) begin

case (ealuc) // arithmetic based on alu control signal

4'b0000: // AND

begin

r = eqa & b;

end

4'b0001: // OR

begin

r = eqa | b;

end

4'b0010: // add

begin

r = eqa + b;

end

4'b0110: // subtract

begin

r = eqa - b;

end

4'b0111: // set on less than

begin

if (eqa < b) begin // less than

r = 32'h00000001;

end

else begin // otherwise

r = 32'h00000000;

end

end

4'b1100: // NOR

begin

r = ~(eqa | b);

end

4'b0011: // XOR

begin

r = eqa ^ b;

end

endcase

end

endmodule

module EXEMEMreg( // sequential element

input ewreg,

input em2reg,

input ewmem,

input [4:0] edestReg,

input [31:0] r,

input [31:0] eqb,

input clk,

output reg mwreg, // EXE/MEM.RegWrite

output reg mm2reg,

output reg mwmem,

output reg [4:0] mdestReg, // EXE/MEM.RegisterRd

output reg [31:0] mr,

output reg [31:0] mqb

);

always@(posedge clk) begin // assign EXE -> MEM

mwreg <= ewreg;

mm2reg <= em2reg;

mwmem <= ewmem;

mdestReg <= edestReg;

mr <= r;

mqb <= eqb;

end

endmodule

module dataMem( // both asynchronous & synchronous action

input [31:0] mr, // memory address

input [31:0] mqb, // data to write in memory

input mwmem, // write to memory signal

input clk, // clk signal for write op

output reg [31:0] mdo // memory output

);

reg [31:0] memory[0:63]; // 64 slots of 32 bits wide memory line

initial begin

// initialize first 10 words of data memory

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

always@(\*) begin // asynchronous read from memory

mdo = memory[mr[31:2]];

end

always@(negedge clk) begin // synchronous write to memory

if (mwmem == 1) begin // memory write enabled

memory[mr[31:2]] <= mqb;

end

end

endmodule

module MEMWBreg(

input mwreg,

input mm2reg,

input [4:0] mdestReg,

input [31:0] mr,

input [31:0] mdo,

input clk,

output reg wwreg, // MEM/WB.RegWrite

output reg wm2reg,

output reg [4:0] wdestReg, // MEM/WB.RegisterRd

output reg [31:0] wr,

output reg [31:0] wdo

);

always@(posedge clk) begin // assign MEM -> WB

wwreg <= mwreg;

wm2reg <= mm2reg;

wdestReg <= mdestReg;

wr <= mr;

wdo <= mdo;

end

endmodule

module wbMux( // select write back data

input [31:0] wr,

input [31:0] wdo,

input wm2reg,

output reg [31:0] wbData

);

always@(\*) begin

if (wm2reg == 0) begin // set to ALU calculation

wbData = wr;

end

else begin // set to data retrieved

wbData = wdo;

end

end

endmodule

module datapath(

input clk,

output wire [31:0] pc,

output wire [31:0] dinstOut,

output wire [31:0] r,

output wire [31:0] mr,

output wire [31:0] wbData

);

// internal components

wire [31:0] nextpc;

wire [31:0] instOut;

wire ewreg;

wire em2reg;

wire ewmem;

wire [3:0] ealuc;

wire ealuimm;

wire wreg;

wire m2reg;

wire wmem;

wire [3:0] aluc;

wire aluimm;

wire regrt;

wire [4:0] destReg;

wire [31:0] eqb;

wire [31:0] qa;

wire [31:0] qb;

wire [31:0] imm32;

// wire [31:0] b;

wire [31:0] mdo;

wire wpcir;

wire [1:0] fwda;

wire [1:0] fwdb;

wire [31:0] da; // forwarded reg a

wire [31:0] db; // forwarded reg b

wire [31:0] eimm32;

wire mwreg;

wire mm2reg;

wire mwmem;

wire [4:0] mdestReg;

wire [4:0] edestReg;

wire wwreg;

wire wm2reg;

wire [4:0] wdestReg;

wire [31:0] wdo;

wire [31:0] mqb;

wire [31:0] wr;

wire [31:0] eqa;

wire [31:0] b;

controlUnit ControlUnit ( // connect Control Unit

.op(dinstOut[31:26]),

.func(dinstOut[5:0]),

.rs(dinstOut[25:21]),

.rt(dinstOut[20:16]),

.wreg(wreg),

.m2reg(m2reg),

.wmem(wmem),

.aluc(aluc),

.aluimm(aluimm),

.regrt(regrt),

.fwda(fwda),

.fwdb(fwdb),

.mrn(mdestReg),

.mm2reg(mm2reg),

.mwreg(mwreg),

.ern(edestReg),

.em2reg(em2reg),

.ewreg(ewreg),

.wpcir(wpcir)

);

IDEXEreg IDEXEReg ( // connect ID/EXE register

.wreg(wreg),

.m2reg(m2reg),

.wmem(wmem) ,

.aluc(aluc),

.aluimm(aluimm),

.destReg(destReg),

.da(da),

.db(db),

.imm32(imm32),

.clk(clk),

.ewreg(ewreg),

.em2reg(em2reg),

.ewmem(ewmem),

.ealuc(ealuc),

.ealuimm(ealuimm),

.edestReg(edestReg),

.eqa(eqa),

.eqb(eqb),

.eimm32(eimm32)

);

immExtender ImmExtender ( // connect sign extension

.imm(dinstOut[15:0]),

.imm32(imm32[31:0])

);

pcAdder PcAdder ( // connect program counter adder

.pc(pc),

.nextPc(nextpc)

);

instMemory InstMemory ( // connect instruction memory

.pc(pc),

.instOut(instOut)

);

IFIDreg IFIDReg ( // connect IF/ID register

.instOut(instOut),

.clk(clk),

.dinstOut(dinstOut),

.wpcir(wpcir)

);

progCounter ProgCounter ( // connect program counter

.clk(clk),

.nextPc(nextpc),

.pc(pc),

.wpcir(wpcir)

);

regFile RegFile ( // connect regFile

.rs(dinstOut[25:21]),

.rt(dinstOut[20:16]),

.wdestReg(wdestReg),

.wbData(wbData),

.wwreg(wwreg),

.clk(clk),

.qa(qa),

.qb(qb)

);

fqaMux FqaMux(

.fwda(fwda),

.qa(qa),

.r(r),

.mr(mr),

.mdo(mdo),

.da(da)

);

fqbMux FqbMux(

.fwdb(fwdb),

.qb(qb),

.r(r),

.mr(mr),

.mdo(mdo),

.db(db)

);

regRt RegRt( // connect regRt

.rt(dinstOut[20:16]),

.rd(dinstOut[15:11]),

.regrt(regrt),

.destReg(destReg)

);

AluMux ALUMux( // connect ALU multiplexer

.eqb(eqb),

.eimm32(eimm32),

.ealuimm(ealuimm),

.b(b)

);

Alu ALU( // connect ALU

.eqa(eqa),

.b(b),

.ealuc(ealuc),

.r(r)

);

EXEMEMreg EXEMEMReg( // connect EXE/MEM Register

.ewreg(ewreg),

.em2reg(em2reg),

.ewmem(ewmem),

.edestReg(edestReg),

.r(r),

.eqb(eqb),

.clk(clk),

.mwreg(mwreg),

.mm2reg(mm2reg),

.mwmem(mwmem),

.mdestReg(mdestReg),

.mr(mr),

.mqb(mqb)

);

MEMWBreg MEMWBReg( // connect MEM/WB Register

.mwreg(mwreg),

.mm2reg(mm2reg),

.mdestReg(mdestReg),

.mr(mr),

.mdo(mdo),

.clk(clk),

.wwreg(wwreg),

.wm2reg(wm2reg),

.wdestReg(wdestReg),

.wr(wr),

.wdo(wdo)

);

dataMem DataMem( // connect Data Memory

.mr(mr),

.mqb(mqb),

.mwmem(mwmem),

.clk(clk),

.mdo(mdo)

);

wbMux WbMux( // connect WB Data Multiplexer

.wr(wr),

.wdo(wdo),

.wm2reg(wm2reg),

.wbData(wbData)

);

endmodule

**Verilog® Testbench Design Code**

`timescale 1ns / 1ps

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

// Company: Penn State College of Engineering

// Engineer: Dohyoung Ko

// Create Date: 12/17/2022 05:26:34 AM

// Design Name: Final Project

// Module Name: testbench

// Project Name: CMPEN-331 FA22

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

module testbench();

reg clk\_tb; // clock signal

wire [31:0] pc\_tb; // IF stage

wire [31:0] inst\_tb; // IF/ID stage

wire [31:0] ealu\_tb; // ID/EXE stage

wire [31:0] malu\_tb; // EXE/MEM stage

wire [31:0] wdi\_tb; // MEM/WB stage

datapath datapath\_tb (

.clk(clk\_tb),

.pc(pc\_tb),

.dinstOut(inst\_tb),

.r(ealu\_tb),

.mr(malu\_tb),

.wbData(wdi\_tb)

);

initial begin

clk\_tb = 0; // initial clock is 0

end

always begin

#5;

clk\_tb = ~clk\_tb; // alternate clock in every 10ns

end

endmodule

**Waveforms**

**Graphical user interface

Description automatically generated**

**Design Schematics**

Diagram

Description automatically generated

**Snapshot of the I/O planning**

A screenshot of a computer

Description automatically generated with medium confidence

**Snapshot of the Floor planning**

A screenshot of a computer

Description automatically generated with medium confidence