**FINAL PROJECT**

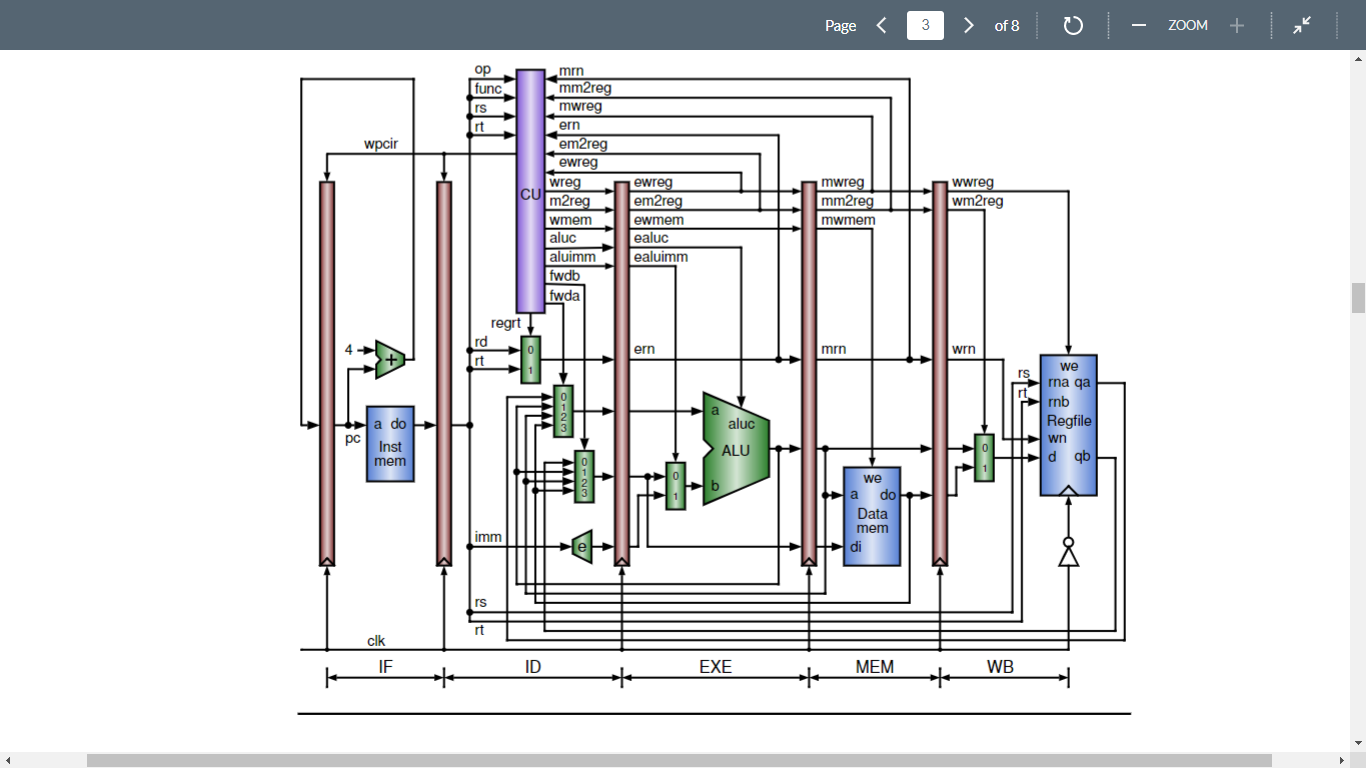
**Name:** Harshit Jain

**Student ID:** 942698258

**Course:** CMPEN 331

**ABSTRACT:**

The final project is the combination of Lab 3, 4, and 5. The final project involved the implementation which ensures the data hazards are avoided given instructions that can induce them. The processor works by having stages, which is the instruction fetch (IF), instruction decode (ID), execution (EXE), memory access (MEM), and write back (WB), as shown on Figure below. Between each stage are registers that take in the outputs of the stage before it and delivers the values for the inputs for the stage after it. It is storing the previous stage’s output values so it can feed it to the next stage. This is beneficial since it is needed in order to have an implementation that uses several clock cycles per instruction. As one instruction is in the IF stage, another instruction is in the ID stage and another in the EXE stage. This allows five instructions to be running at the same time, which allows our program to run faster, taking advantage of parallelism. This is possible due to the values being stored in the registers so they can be used by the next stages.



**Figure 1. Mechanism of internal forwarding and pipeline stall**

**VERILOG CODE:**

`timescale 1ns / 1ps

module datapath (clk, pc, nextPc, dinstOut, ewreg, em2reg, ewmem, ealuc, ealuimm, edestReg, eqa, eqb, eimm32, mwreg, mm2reg, mwmem,

mdestReg, mr, mqb, wwreg, wm2reg, wdestReg, wr, wdo);

input clk;

//outputs of Program Counter

output [31:0] pc;

output [31:0] nextPc;

//outputs of IF/ID Pipeline Register

output [31:0] dinstOut;

//outputs of ID/EXE Pipeline Register

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;

//outputs of EXE/MEM Pipeline Register

output mwreg;

output mm2reg;

output mwmem;

output [4:0] mdestReg;

output [31:0] mr;

output [31:0] mqb;

//outputs of MEM/WB Pipeline Register

output wwreg;

output wm2reg;

output [4:0] wdestReg;

output [31:0] wr;

output [31:0] wdo;

wire [31:0] instOut;

wire regrt;

wire wreg;

wire m2reg;

wire wmem;

wire [3:0] aluc;

wire aluimm;

wire [4:0] rd;

wire [4:0] rt;

wire [4:0] destReg;

wire [31:0] qa;

wire [31:0] qb;

wire [31:0] imm32;

wire [31:0] mdo;

wire [31:0] b;

wire [31:0] r;

wire [1:0] fwda;

wire [1:0] fwdb;

wire [31:0] fqa;

wire [31:0] fqb;

wire [31:0] wbData;

program\_Counter program\_Counter (clk, nextPc, pc);

pcAdder pcAdder (pc, nextPc);

Instruction\_memory Instruction\_memory (pc, instOut);

ifid\_pipeline\_register ifid\_pipeline\_register (clk, instOut, dinstOut);

control\_unit control\_unit (dinstOut[31:26], dinstOut[5:0], dinstOut[25:21], rt, mdestReg, mm2reg, mwreg, edestReg, em2reg, ewreg, regrt, wreg, m2reg, wmem, aluimm, aluc, fwda, fwdb);

regrt\_multiplexer regrt\_multiplexer (dinstOut[15:11], dinstOut[20:16], regrt, destReg);

register\_file register\_file (clk, dinstOut[25:21], dinstOut[20:16], wwreg, wbData, wdestReg, qa, qb);

immediate\_extender immediate\_extender (dinstOut[15:0], imm32);

idexe\_pipeline\_register idexe\_pipeline\_register (clk, wreg, m2reg, wmem, aluc, aluimm, destReg, fqa, fqb, imm32, ewreg, em2reg, ewmem, ealuc, ealuimm, edestReg, eqa, eqb, eimm32);

AluMux AluMux (eqb, eimm32, ealuimm, b);

Alu Alu (eqa, b, ealuc, r);

exemem\_pipeline\_register exemem\_pipeline\_register (clk, ewreg, em2reg, ewmem, edestReg, r, eqb, mwreg, mm2reg, mwmem, mdestReg, mr, mqb);

data\_memory data\_memory (clk ,mr, mqb, mwmem, mdo);

memwb\_pipeline\_register memwb\_pipeline\_register (clk, mwreg, mm2reg, mdestReg, mr, mdo, wwreg, wm2reg, wdestReg, wr, wdo);

WbMux WbMux (wr, wdo, wm2reg, wbData);

mux4 mux4 (qa, r, mr, mdo, fwda, fqa);

mux5 mux5 (qb, r, mr, mdo, fwdb, fqb);

endmodule

//PROGRAM COUNTER (PC)

module program\_Counter(clk, nextPc, pc);

input clk;

input [31:0] nextPc;

output reg [31:0] pc;

always @ (posedge clk)

begin

pc <= nextPc;

end

endmodule

//PC ADDER

module pcAdder(pc, nextPc);

input [31:0] pc;

output reg [31:0] nextPc;

initial

begin

nextPc = 32'd100;

end

always @ (\*)

begin

nextPc [31:0] <= pc [31:0] + 32'd4;

end

endmodule

//INSTRUCTION MEMORY

module Instruction\_memory(pc, instOut);

input [31:0] pc;

output reg [31:0] instOut;

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

initial begin

memory[25] <= 32'b10001100001000100000000000000000; //LW

memory[26] <= 32'b10001100001000110000000000000100; //LW

memory[27] <= 32'b10001100001001000000000000001000; //LW

memory[28] <= 32'b10001100001001010000000000001100; //LW

memory[29] <= 32'b00000000010010100011000000100000; //ADD

end

always @ (\*) begin

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

end

endmodule

//IFID PIPELINE REGISTER

module ifid\_pipeline\_register (clk, instOut, dinstOut);

input clk;

input [31:0] instOut;

output reg [31:0] dinstOut;

always @(posedge clk) begin

dinstOut <= instOut;

end

endmodule

//CONTROL UNIT

module control\_unit (op, func, rs, rt, mdestReg, mm2reg, mwreg, edestReg, em2reg, ewreg, regrt, wreg, m2reg, wmem, aluimm, aluc, fwda, fwdb);

input [5:0] op;

input [5:0] func;

input [4:0] rs;

input [4:0] rt;

input [4:0] mdestReg;

input mm2reg, mwreg, edestReg, em2reg, ewreg;

output reg regrt, wreg, m2reg, wmem, aluimm;

output reg [3:0] aluc;

output reg [31:0] fwda;

output reg [31:0] fwdb;

always @(\*) begin

fwda = 2'b00;

fwdb = 2'b00;

if (edestReg == rs) begin

fwda = 2'b01;

end

if (edestReg == rt) begin

fwdb = 2'b01;

end

if (mdestReg == rs) begin

if (mm2reg == 1) begin

fwda = 2'b11;

end

else begin

fwda = 2'b10;

end

end

if (mdestReg == rt) begin

if (mm2reg == 1) begin

fwdb = 2'b11;

end

else begin

fwdb = 2'b10;

end

end

end

always @(\*) begin

if (op == 6'b000000) begin //R-Types

if (func == 6'b100000) begin //ADD instruction

regrt <= 1'b0;

wreg <= 1'b1;

m2reg <= 1'b0;

wmem <= 1'b0;

aluimm <= 1'b0;

aluc <= 4'b0010;

end

else if (func == 6'b100010) begin //SUB instruction

regrt <= 1'b0;

wreg <= 1'b1;

m2reg <= 1'b0;

wmem <= 1'b0;

aluimm <= 1'b0;

aluc <= 4'b0110;

end

else if (func == 6'b100100) begin //AND instruction

regrt <= 1'b0;

wreg <= 1'b1;

m2reg <= 1'b0;

wmem <= 1'b0;

aluimm <= 1'b0;

aluc <= 4'b0000;

end

else if (func == 6'b100101) begin //OR instruction

regrt <= 1'b0;

wreg <= 1'b1;

m2reg <= 1'b0;

wmem <= 1'b0;

aluimm <= 1'b0;

aluc <= 4'b0001;

end

else if (func == 6'b100110) begin //XOR instruction

regrt <= 1'b0;

wreg <= 1'b1;

m2reg <= 1'b0;

wmem <= 1'b0;

aluimm <= 1'b0;

aluc <= 4'b0011;

end

end

else if (op == 6'b100011) begin //LW

regrt <= 1'b1;

wreg <= 1'b1;

m2reg <= 1'b1;

wmem <= 1'b0;

aluimm <= 1'b1;

aluc <= 4'b0010;

end

end

endmodule

//MULTIPLER 1 (REGRT MULTIPLEXER)

module regrt\_multiplexer (rd, rt, regrt, destReg);

input [4:0] rd;

input [4:0] rt;

input regrt;

output reg [4:0] destReg;

always@(\*) begin

if (regrt == 1'b1) begin

destReg <= rt;

end

else begin

destReg <= rd;

end

end

endmodule

//REGISTER FILE

module register\_file (clk, rs, rt, wwreg, wbData, wdestReg, qa, qb);

input clk;

input [4:0] rs;

input [4:0] rt;

input wwreg; `timescale 1ns / 1ps

module datapath (clk, pc, nextPc, dinstOut, ewreg, em2reg, ewmem, ealuc, ealuimm, edestReg, eqa, eqb, eimm32, mwreg, mm2reg, mwmem,

mdestReg, mr, mqb, wwreg, wm2reg, wdestReg, wr, wdo);

input clk;

//outputs of Program Counter

output [31:0] pc;

output [31:0] nextPc;

//outputs of IF/ID Pipeline Register

output [31:0] dinstOut;

//outputs of ID/EXE Pipeline Register

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;

//outputs of EXE/MEM Pipeline Register

output mwreg;

output mm2reg;

output mwmem;

output [4:0] mdestReg;

output [31:0] mr;

output [31:0] mqb;

//outputs of MEM/WB Pipeline Register

output wwreg;

output wm2reg;

output [4:0] wdestReg;

output [31:0] wr;

output [31:0] wdo;

wire [31:0] instOut;

wire regrt;

wire wreg;

wire m2reg;

wire wmem;

wire [3:0] aluc;

wire aluimm;

wire [4:0] rd;

wire [4:0] rt;

wire [4:0] destReg;

wire [31:0] qa;

wire [31:0] qb;

wire [31:0] imm32;

wire [31:0] mdo;

wire [31:0] b;

wire [31:0] r;

wire [1:0] fwda;

wire [1:0] fwdb;

wire [31:0] fqa;

wire [31:0] fqb;

wire [31:0] wbData;

program\_Counter program\_Counter (clk, nextPc, pc);

pcAdder pcAdder (pc, nextPc);

Instruction\_memory Instruction\_memory (pc, instOut);

ifid\_pipeline\_register ifid\_pipeline\_register (clk, instOut, dinstOut);

control\_unit control\_unit (dinstOut[31:26], dinstOut[5:0], dinstOut[25:21], rt, mdestReg, mm2reg, mwreg, edestReg, em2reg, ewreg, regrt, wreg, m2reg, wmem, aluimm, aluc, fwda, fwdb);

regrt\_multiplexer regrt\_multiplexer (dinstOut[15:11], dinstOut[20:16], regrt, destReg);

register\_file register\_file (clk, dinstOut[25:21], dinstOut[20:16], wwreg, wbData, wdestReg, qa, qb);

immediate\_extender immediate\_extender (dinstOut[15:0], imm32);

idexe\_pipeline\_register idexe\_pipeline\_register (clk, wreg, m2reg, wmem, aluc, aluimm, destReg, fqa, fqb, imm32, ewreg, em2reg, ewmem, ealuc, ealuimm, edestReg, eqa, eqb, eimm32);

AluMux AluMux (eqb, eimm32, ealuimm, b);

Alu Alu (eqa, b, ealuc, r);

exemem\_pipeline\_register exemem\_pipeline\_register (clk, ewreg, em2reg, ewmem, edestReg, r, eqb, mwreg, mm2reg, mwmem, mdestReg, mr, mqb);

data\_memory data\_memory (clk ,mr, mqb, mwmem, mdo);

memwb\_pipeline\_register memwb\_pipeline\_register (clk, mwreg, mm2reg, mdestReg, mr, mdo, wwreg, wm2reg, wdestReg, wr, wdo);

WbMux WbMux (wr, wdo, wm2reg, wbData);

mux4 mux4 (qa, r, mr, mdo, fwda, fqa);

mux5 mux5 (qb, r, mr, mdo, fwdb, fqb);

endmodule

//PROGRAM COUNTER (PC)

module program\_Counter(clk, nextPc, pc);

input clk;

input [31:0] nextPc;

output reg [31:0] pc;

always @ (posedge clk)

begin

pc <= nextPc;

end

endmodule

//PC ADDER

module pcAdder(pc, nextPc);

input [31:0] pc;

output reg [31:0] nextPc;

initial

begin

nextPc = 32'd100;

end

always @ (\*)

begin

nextPc [31:0] <= pc [31:0] + 32'd4;

end

endmodule

//INSTRUCTION MEMORY

module Instruction\_memory(pc, instOut);

input [31:0] pc;

output reg [31:0] instOut;

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

initial begin

memory[25] <= 32'b10001100001000100000000000000000; //LW

memory[26] <= 32'b10001100001000110000000000000100; //LW

memory[27] <= 32'b10001100001001000000000000001000; //LW

memory[28] <= 32'b10001100001001010000000000001100; //LW

memory[29] <= 32'b00000000010010100011000000100000; //ADD

end

always @ (\*) begin

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

end

endmodule

//IFID PIPELINE REGISTER

module ifid\_pipeline\_register (clk, instOut, dinstOut);

input clk;

input [31:0] instOut;

output reg [31:0] dinstOut;

always @(posedge clk) begin

dinstOut <= instOut;

end

endmodule

//CONTROL UNIT

module control\_unit (op, func, rs, rt, mdestReg, mm2reg, mwreg, edestReg, em2reg, ewreg, regrt, wreg, m2reg, wmem, aluimm, aluc, fwda, fwdb);

input [5:0] op;

input [5:0] func;

input [4:0] rs;

input [4:0] rt;

input [4:0] mdestReg;

input mm2reg, mwreg, edestReg, em2reg, ewreg;

output reg regrt, wreg, m2reg, wmem, aluimm;

output reg [3:0] aluc;

output reg [31:0] fwda;

output reg [31:0] fwdb;

always @(\*) begin

fwda = 2'b00;

fwdb = 2'b00;

if (edestReg == rs) begin

fwda = 2'b01;

end

if (edestReg == rt) begin

fwdb = 2'b01;

end

if (mdestReg == rs) begin

if (mm2reg == 1) begin

fwda = 2'b11;

end

else begin

fwda = 2'b10;

end

end

if (mdestReg == rt) begin

if (mm2reg == 1) begin

fwdb = 2'b11;

end

else begin

fwdb = 2'b10;

end

end

end

always @(\*) begin

if (op == 6'b000000) begin //R-Types

if (func == 6'b100000) begin //ADD instruction

regrt <= 1'b0;

wreg <= 1'b1;

m2reg <= 1'b0;

wmem <= 1'b0;

aluimm <= 1'b0;

aluc <= 4'b0010;

end

else if (func == 6'b100010) begin //SUB instruction

regrt <= 1'b0;

wreg <= 1'b1;

m2reg <= 1'b0;

wmem <= 1'b0;

aluimm <= 1'b0;

aluc <= 4'b0110;

end

else if (func == 6'b100100) begin //AND instruction

regrt <= 1'b0;

wreg <= 1'b1;

m2reg <= 1'b0;

wmem <= 1'b0;

aluimm <= 1'b0;

aluc <= 4'b0000;

end

else if (func == 6'b100101) begin //OR instruction

regrt <= 1'b0;

wreg <= 1'b1;

m2reg <= 1'b0;

wmem <= 1'b0;

aluimm <= 1'b0;

aluc <= 4'b0001;

end

else if (func == 6'b100110) begin //XOR instruction

regrt <= 1'b0;

wreg <= 1'b1;

m2reg <= 1'b0;

wmem <= 1'b0;

aluimm <= 1'b0;

aluc <= 4'b0011;

end

end

else if (op == 6'b100011) begin //LW

regrt <= 1'b1;

wreg <= 1'b1;

m2reg <= 1'b1;

wmem <= 1'b0;

aluimm <= 1'b1;

aluc <= 4'b0010;

end

end

endmodule

//MULTIPLER 1 (REGRT MULTIPLEXER)

module regrt\_multiplexer (rd, rt, regrt, destReg);

input [4:0] rd;

input [4:0] rt;

input regrt;

output reg [4:0] destReg;

always@(\*) begin

if (regrt == 1'b1) begin

destReg <= rt;

end

else begin

destReg <= rd;

end

end

endmodule

//REGISTER FILE

module register\_file (clk, rs, rt, wwreg, wbData, wdestReg, qa, qb);

input clk;

input [4:0] rs;

input [4:0] rt;

input wwreg;

input [31:0] wbData;

input [4:0] wdestReg;

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

output reg [31:0] qa;

output reg [31:0] qb;

integer i;

initial begin

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

begin

registers[i] <= 32'b0;

end

end

always @(\*) begin

qa = registers[rs];

qb = registers[rt];

if (wwreg == 1) begin

registers[wdestReg] = wbData;

end

end

endmodule

//SIGN EXTEND

module immediate\_extender (imm, imm32);

input [15:0] imm;

output reg [31:0] imm32;

always @(\*) begin

imm32[31:0] = {{16{imm[15]}}, imm[15:0]};

end

endmodule

//IDEXE PIPELINE REGISTER

module idexe\_pipeline\_register (clk, wreg, m2reg, wmem, aluc, aluimm, destReg,

fqa, fqb, imm32, ewreg, em2reg, ewmem, ealuc, ealuimm, edestReg, eqa, eqb, eimm32);

input clk;

input wreg, m2reg, wmem;

input [3:0] aluc;

input aluimm;

input [4:0] destReg;

input [31:0] fqa;

input [31:0] fqb;

input [31:0] imm32;

output reg ewreg, em2reg, 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;

always @(posedge clk) begin

ewreg <= wreg;

em2reg <= m2reg;

ewmem <= wmem;

ealuc <= aluc;

ealuimm <= aluimm;

edestReg <= destReg;

eqa <= fqa;

eqb <= fqb;

eimm32 <= imm32;

end

endmodule

//LAB 4

//MULTIPLEXER 2 (ALU MULTIPLEXER)

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

input [31:0] eqb;

input [31:0] eimm32;

input ealuimm;

output reg [31:0] b;

always@(\*) begin

if (ealuimm == 1) begin

b <= eimm32;

end

else if (ealuimm == 0) begin

b <= eqb;

end

end

endmodule

//ALU

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

input [31:0] eqa;

input [31:0] b;

input [3:0] ealuc;

output reg [31:0] r;

always@(\*) begin

if (ealuc == 0) begin

r <= eqa & b;

end

else if (ealuc == 4'b0001) begin

r <= eqa | b;

end

else if (ealuc == 4'b0010) begin

r <= eqa + b;

end

else if (ealuc == 4'b0011) begin

r <= eqa ^ b;

end

else if (ealuc == 4'b0110) begin

r <= eqa - b;

end

end

endmodule

//EXEMEM PIPELINE REGISTER

module exemem\_pipeline\_register (clk, ewreg, em2reg, ewmem, edestReg, r, eqb,

mwreg, mm2reg, mwmem, mdestReg, mr, mqb);

input clk;

input ewreg, em2reg, ewmem;

input [4:0] edestReg;

input [31:0] r;

input [31:0] eqb;

output reg mwreg, mm2reg, mwmem;

output reg [4:0] mdestReg;

output reg [31:0] mr;

output reg [31:0] mqb;

always @(posedge clk) begin

mwreg <= ewreg;

mm2reg <= em2reg;

mwmem <= ewmem;

mdestReg <= edestReg;

mr <= r;

mqb <= eqb;

end

endmodule

//DATA MEMORY

module data\_memory (clk, mr, mqb, mwmem, mdo);

input clk;

input [31:0] mr;

input [31:0] mqb;

input mwmem;

output reg [31:0] mdo;

reg [31:0] memory [128:0];

initial begin

memory[0] = 32'hA00000AA;

memory[4] = 32'h10000011;

memory[8] = 32'h20000022;

memory[12] = 32'h30000033;

memory[16] = 32'h40000044;

memory[20] = 32'h50000055;

memory[24] = 32'h60000066;

memory[28] = 32'h70000077;

memory[32] = 32'h80000088;

memory[36] = 32'h90000099;

end

always @(\*) begin

if (mwmem == 1'b0) begin

mdo <= memory[mr];

end

end

//always @(negedge clk) begin

// if (mwmem == 1'b1) begin

// memory[mr] <= mqb;

// end

// end

endmodule

//MEMWB PIPELINE REGISTER

module memwb\_pipeline\_register (clk, mwreg, mm2reg, mdestReg, mr, mdo, wwreg, wm2reg, wdestReg, wr, wdo);

input clk;

input mwreg, mm2reg;

input [4:0] mdestReg;

input [31:0] mr;

input [31:0] mdo;

output reg wwreg, wm2reg;

output reg [4:0] wdestReg;

output reg [31:0] wr;

output reg [31:0] wdo;

always @(posedge clk) begin

wwreg <= mwreg;

wm2reg <= mm2reg;

wdestReg <= mdestReg;

wr <= mr;

wdo <= mdo;

end

endmodule

//LAB 5

// MULTIPLEXER 3 (WRITEBACK MULTIPLEXER)

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

input [31:0] wr;

input [31:0] wdo;

input wm2reg;

output reg [31:0] wbData;

always@(\*) begin

if (wm2reg == 1) begin

wbData <= wdo;

end

else if (wm2reg == 0) begin

wbData <= wr;

end

end

endmodule

// Final Project Implementation

//MULTIPLEXER 4

module mux4 (qa, r, mr, mdo, fwda, fqa);

input [31:0] qa;

input [31:0] r;

input [31:0] mr;

input [31:0] mdo;

input [1:0] fwda;

output reg [31:0] fqa;

always@(\*) begin

if (fwda == 2'b00) begin

fqa <= qa;

end

else if (fwda == 2'b01) begin

fqa <= r;

end

else if (fwda == 2'b10) begin

fqa <= mr;

end

else if (fwda == 2'b11) begin

fqa <= mdo;

end

end

endmodule

//MULTIPLEXER 5

module mux5 (qb, r, mr, mdo, fwdb, fqb);

input [31:0] qb;

input [31:0] r;

input [31:0] mr;

input [31:0] mdo;

input [1:0] fwdb;

output reg [31:0] fqb;

always@(\*) begin

if (fwdb == 2'b00) begin

fqb <= qb;

end

else if (fwdb == 2'b01) begin

fqb <= r;

end

else if (fwdb == 2'b10) begin

fqb <= mr;

end

else if (fwdb == 2'b11) begin

fqb <= mdo;

end

end

endmodule

**TESTBENCH:**

`timescale 1ns / 1ps

module testbench();

reg clk;

wire [31:0] pc\_tb;

wire [31:0] nextPc\_tb;

wire [31:0] dinstOut\_tb;

wire ewreg\_tb;

wire em2reg\_tb;

wire ewmem\_tb;

wire [3:0] ealuc\_tb;

wire ealuimm\_tb;

wire [4:0] edestReg\_tb;

wire [31:0] eqa\_tb;

wire [31:0] eqb\_tb;

wire [31:0] eimm32\_tb;

wire mwreg\_tb;

wire mm2reg\_tb;

wire mwmem\_tb;

wire [4:0] mdestReg\_tb;

wire [31:0] mr\_tb;

wire [31:0] mqb\_tb;

wire wwreg\_tb;

wire wm2reg\_tb;

wire [4:0] wdestReg\_tb;

wire [31:0] wr\_tb;

wire [31:0] wdo\_tb;

initial begin

clk = 0;

end

datapath datapath\_tb (clk, pc\_tb, nextPc\_tb, dinstOut\_tb, ewreg\_tb, em2reg\_tb, ewmem\_tb, ealuc\_tb, ealuimm\_tb, edestReg\_tb, eqa\_tb, eqb\_tb, eimm32\_tb, mwreg\_tb, mm2reg\_tb, mwmem\_tb, mdestReg\_tb, mr\_tb, mqb\_tb, wwreg\_tb, wm2reg\_tb, wdestReg\_tb, wr\_tb, wdo\_tb);

always begin

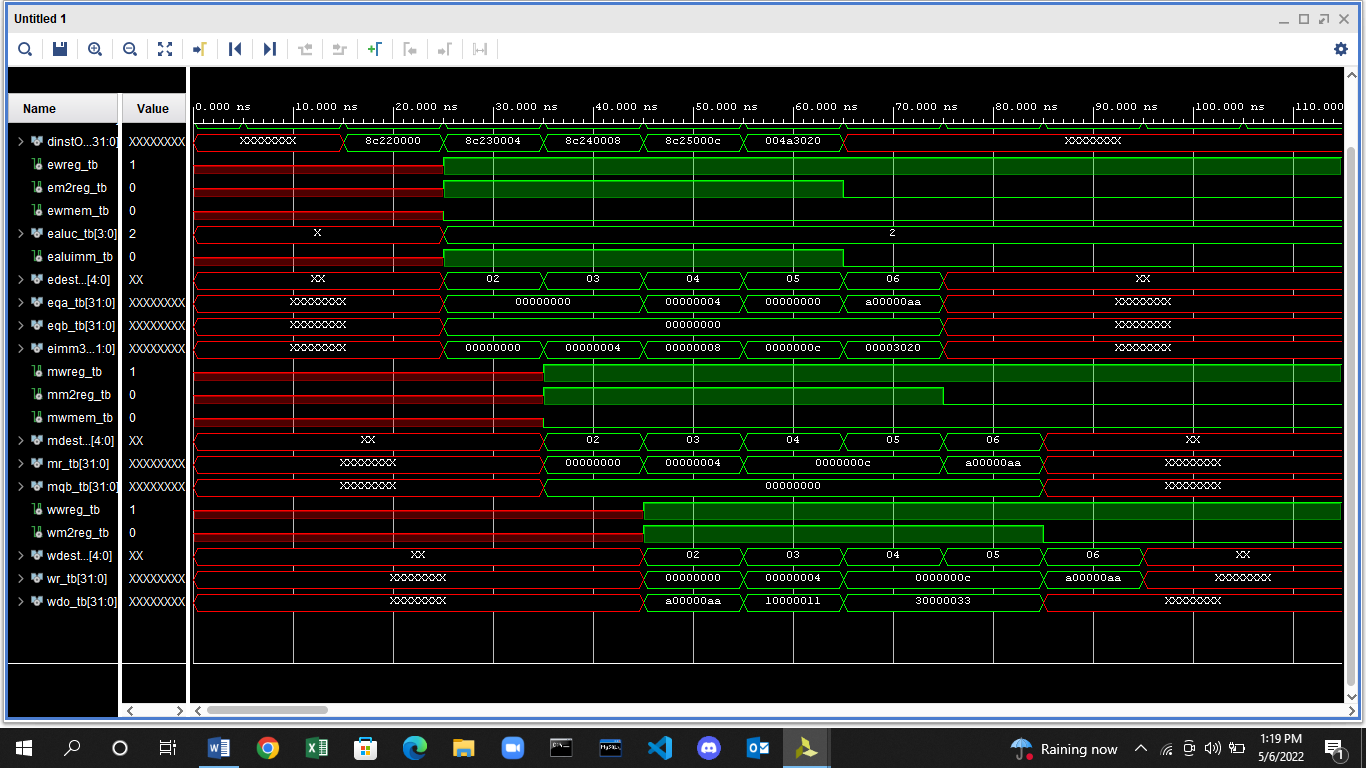
#5

clk = ~clk;

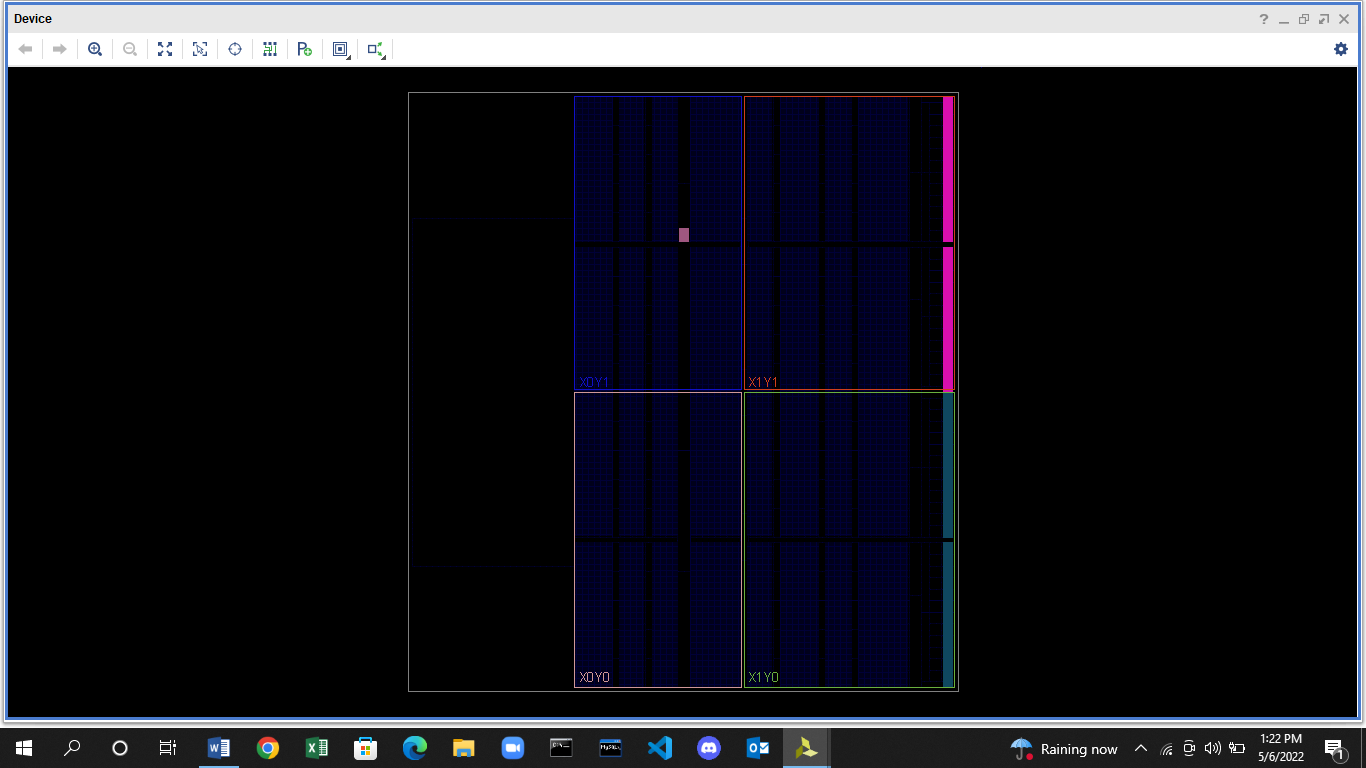
end

endmodule

**WAVESFORM:**



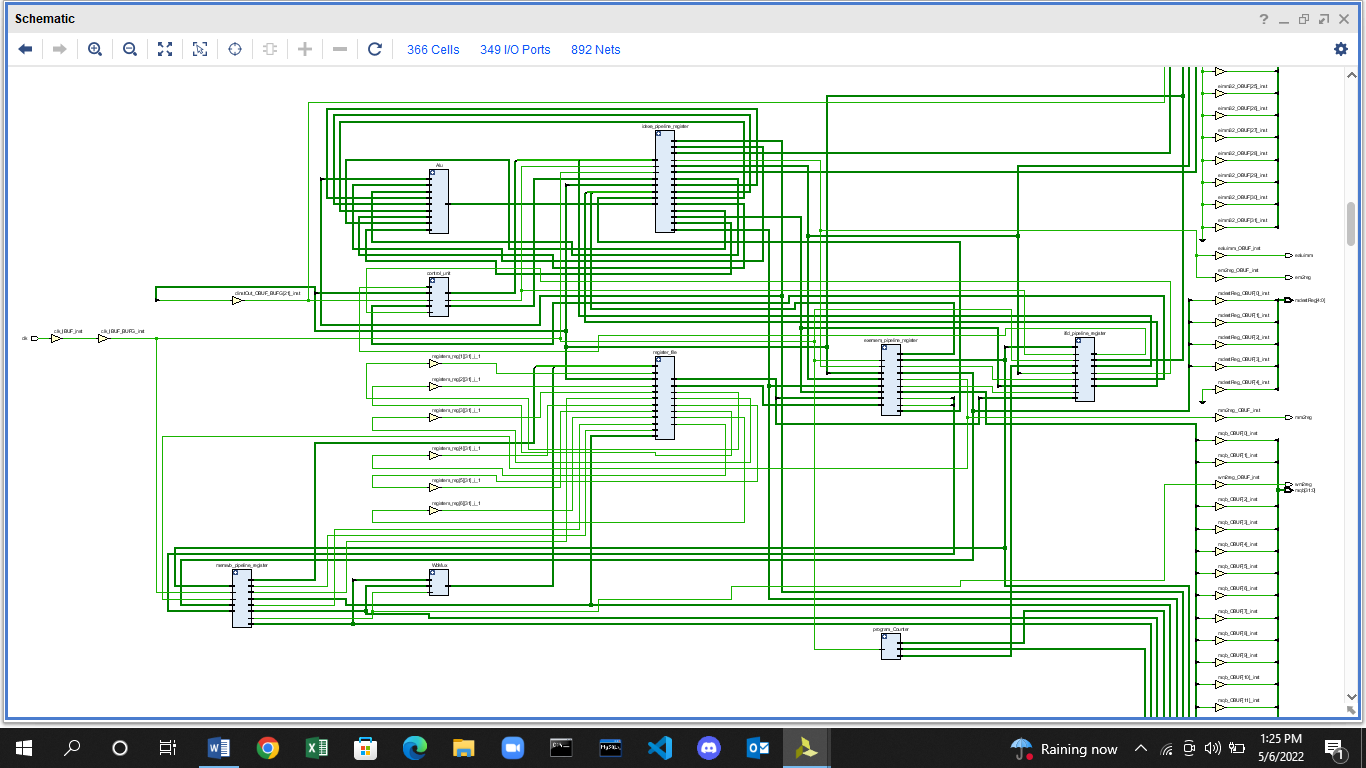
**I/O PLANNING:**



**FLOOR PLANNING:**



**DESIGN SCHEMATICS:**



**DESIGN SCHEMATICS (RTL ANALYSIS):**

