**Lab 4 – CMPEN 331**

**Tanner Kleintop**

**Section 002**

**Dr. Almekkawy**

**Verilog Code:**

**Testbench:**

`timescale 1ns / 1ps

module Datapath(

// Lab 3 - IDEXEpipeline

input clk, // Clock signal

output wire [31:0] pc, // Program Counter

output wire [31:0] dinstOut, // Data from the Instruction Memory

output wire ewreg, // Control signal for write enable in the EX stage

output wire em2reg, // Control signal for write enable in the Memory stage

output wire ewmem, // Control signal for write enable in the MEM stage

output wire [3:0] ealuc, // ALU control signal in the EX stage

output wire ealuimm, // ALU immediate in the EX stage

output wire [4:0] edestReg, // Destination register address in the EX stage

output wire [31:0] eqa, // Data from source register A in the EX stage

output wire [31:0] eqb, // Data from source register B in the EX stage

output wire [31:0] eimm32, // 32-bit immediate in the EX stage

// Lab 4 - EXEMEM

output wire mwreg, // Control signal for write enable in the MEM stage

output wire mm2reg, // Control signal for write enable in the M2 stage (MEM-WB)

output wire mwmem, // Control signal for memory write enable in the MEM stage

output wire [4:0] mdestReg, // Destination register address in the MEM stage

output wire [31:0] mr, // Data read from memory in the MEM stage

output wire [31:0] mqb, // Data from source register B in the MEM stage

// Lab 4 - MEMWB

output wire wwreg, // Control signal for write enable in the WB stage

output wire wm2reg, // Control signal for write enable in the M2 stage (WB)

output wire [4:0] wdestReg, // Destination register address in the WB stage

output wire [31:0] wr, // Result to be written to the register file in the WB stage

output wire [31:0] wdo // Data read from memory or ALU result to be written to the register file

);

// Lab 3 wires

wire wreg; // Control signal for write enable

wire m2reg; // Control signal for write enable (Memory stage)

wire wmem; // Control signal for memory write enable

wire aluimm; // ALU immediate value

wire regrt; // Control signal for selecting a register

wire [4:0] destReg; // Destination register address

wire [31:0] qa; // Data from source register A

wire [31:0] qb; // Data from source register B

wire [31:0] imm32; // 32-bit immediate value

wire [31:0] nextPc; // Next program counter value

wire [31:0] instOut; // Data from the Instruction Memory

wire [3:0] aluc; // ALU control signal

wire [15:0] imm; // Immediate value

wire [4:0] rs; // Source register address

wire [4:0] rt; // Source register address

wire [4:0] rd; // Destination register address

wire [5:0] op; // Operation code

wire [5:0] func; // Function code

// Lab 4 wires

wire [31:0] b; // Data input to the ALU

wire [31:0] r; // Result from the ALU

wire [31:0] mdo; // Data read from memory or result from ALU

// Instantiate various components and connect them

ProgramCounter ProgramCounter\_db(.clk(clk), .nextPc(nextPc), .pc(pc));

pcAdder pcAdder\_db(.pc(pc), .nextPc(nextPc));

InstructionMemory InstructionMemory\_db(.pc(pc), .instOut(instOut));

IFIDpipelineReg IFIDpipelineReg\_db(.clk(clk), .instOut(instOut), .dinstOut(dinstOut));

ControlUnit controlUnit\_db(.op(op), .func(func), .wreg(wreg), .m2reg(m2reg), .wmem(wmem), .aluc(aluc), .aluimm(aluimm), .regrt(regrt));

RegrtMultiplexer RegrtMultiplexer\_db(.rt(rt), .rd(rd), .regrt(regrt), .destReg(destReg));

RegisterFile RegisterFile\_db(.rs(rs), .rt(rt), .qa(qa), .qb(qb));

ImmediateExtender ImmediateExtender\_db(.imm(imm), .imm32(imm32));

IDEXEpipeline IDEXEpipeline\_db(

.wreg(wreg),

.m2reg(m2reg),

.wmem(wmem),

.aluc(aluc),

.aluimm(aluimm),

.destReg(destReg),

.qa(qa),

.qb(qb),

.imm32(imm32),

.clk(clk),

.ewreg(ewreg),

.em2reg(em2reg),

.ewmem(ewmem),

.ealuc(ealuc),

.ealuimm(ealuimm),

.edestReg(edestReg),

.eqa(eqa),

.eqb(eqb),

.eimm32(eimm32)

);

ALU ALU\_db(.eqa(eqa), .b(b), .ealuc(ealuc), .r(r));

ALUMux ALUMux\_db(.eqb(eqb), .eimm32(eimm32), .ealuimm(ealuimm), .b(b));

EXEMEMpipeline EXEMempipeline\_db(

.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)

);

DataMemory DataMemory\_db(.mr(mr), .mqb(mqb), .mwmem(mwmem), .clk(clk), .mdo(mdo));

MEMWBpipeline MEMWBpipeline\_db(

.mwreg(mwreg),

.mm2reg(mm2reg),

.mdestReg(mdestReg),

.mr(mr),

.mdo(mdo),

.clk(clk),

.wwreg(wwreg),

.wm2reg(wm2reg),

.wdestReg(wdestReg),

.wr(wr),

.wdo(wdo)

);

// Assign some control signals and data values

assign op = dinstOut[31:26];

assign func = dinstOut[5:0];

assign rs = dinstOut[25:21];

assign rt = dinstOut[20:16];

assign rd = dinstOut[15:11];

assign imm = dinstOut[15:0];

endmodule

module ProgramCounter(

input clk, // Clock input necessary as PC only updates on the positive edge of the clock.

input [31:0] nextPc, // Input from the PC adder looped back to update the next PC.

output reg [31:0] pc // Output of the PC module.

);

initial

begin

pc = 32'd100; // Initializing the PC value to start at 100 in decimal.

end

always @(posedge clk)

begin

pc = nextPc; // Update PC to be nextPc only on the positive edge of the clock.

end

endmodule // End of the module

module pcAdder( //creation of the module used for the PC adder module in the cpu.

input [31:0] pc, //input of pc set to be 32 bits wide.

output reg [31:0] nextPc //output register of next pc that is also 32 bits wide.

);

always @(\*) begin //always block that changes ony any signal used to continually update nextPc.

nextPc <= pc + 32'b00000000000000000000000000000100; //setting nextPc equal to ithe input of pc plus a unsigned binary 32 bit 4.

end //end always block

endmodule //end of this module

module InstructionMemory( //instruction memory module within the cpu.

input [31:0] pc, //input of the instruction memory module.

output reg [31:0] instOut //instruction output of the memory module

);

reg [31:0] memory [0:63]; //32x64 array used to store instructions to memory.

initial begin

//assign the instruction values in memory here (words 25 and 26)

//lw $v0, 00($at)

memory[25] = {6'b100011, 5'b00001, 5'b00010, 5'b00000, 5'b00000, 6'b000000};

//lw $v1, 04($at)

memory[26] = {6'b100011, 5'b00001, 5'b00011, 5'b00000, 5'b00000, 6'b000100};

//lw $a0, 08($at)

memory[27] = {6'b100011, 5'b00001, 5'b00100, 5'b00000, 5'b00000, 6'b001000};

//lw $a1, 12($at)

memory[28] = {6'b100011, 5'b00001, 5'b00101, 5'b00000, 5'b00000, 6'b001100};

end

always @ (\*) //always block to update instruction out setting to the memory array of pc bits 7 to 2.

begin

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

end

endmodule //end of module

module IFIDpipelineReg( //IFID pipeline

input clk, //clock input needed as dinstOut only updates on the positive edge of clock.

input [31:0] instOut, //input

output reg [31:0] dinstOut //output

);

always @ (posedge clk) //always block that will only update dinstOut on the positive edge of the clock. dinstOut is to the instOut input of this module.

begin

dinstOut <= instOut;

end

endmodule //end module

module ControlUnit( //control unit module of the cpu

//inputs

input [5:0] op,

input [5:0] func,

//outputs

output reg wreg,

output reg m2reg,

output reg wmem,

output reg [3:0] aluc,

output reg aluimm,

output reg regrt

);

always @ (\*) begin //always block that will continually update

case (op) //case statement of the op code portion of dinstOut which is connected in the datapath module.

6'b000000: // R-type instructions

begin

case (func) //case statement to check which operation is performed.

6'b100000: // ADD instruction

begin

// Set control signals for ADD instruction

wreg = 1'b1; // Write to the register file

m2reg = 1'b0; // Do not write to memory

wmem = 1'b0; // Do not write to memory

aluc = 4'b0010; // ALU operation for addition

aluimm = 1'b0; // ALU source from registers

regrt = 1'b0; // Destination register address

end

/\* 6'b100010: // SUB instruction

begin

// Set control signals for ADD instruction

wreg = 1'b1; // Write to the register file

m2reg = 1'b0; // Do not write to memory

wmem = 1'b0; // Do not write to memory

aluc = 4'b0110; // ALU operation for addition

aluimm = 1'b0; // ALU source from registers

regrt = 1'b1; // Destination register address

end \*/

/\* 6'b100100: // AND instruction

begin

// Set control signals for ADD instruction

wreg = 1'b1; // Write to the register file

m2reg = 1'b0; // Do not write to memory

wmem = 1'b0; // Do not write to memory

aluc = 4'b0000; // ALU operation for addition

aluimm = 1'b0; // ALU source from registers

regrt = 1'b1; // Destination register address

end \*/

/\* 6'b100101: // OR instruction

begin

// Set control signals for ADD instruction

wreg = 1'b1; // Write to the register file

m2reg = 1'b0; // Do not write to memory

wmem = 1'b0; // Do not write to memory

aluc = 4'b0001; // ALU operation for addition

aluimm = 1'b0; // ALU source from registers

regrt = 1'b1; // Destination register address

end \*/

/\* 6'b100110: // XOR instruction

begin

// Set control signals for ADD instruction

wreg = 1'b1; // Write to the register file

m2reg = 1'b1; // Do not write to memory

wmem = 1'b1; // Do not write to memory

aluc = 4'b1111; // ALU operation for addition

aluimm = 1'b0; // ALU source from registers

regrt = 1'b1; // Destination register address

end \*/

default: // Default behavior for unspecified func values

begin

// Set default control signals here

// You can assign default values or behavior

// for cases where func is unspecified

// For example, you can set all signals to 0.

wreg = 1'b0;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0000;

aluimm = 1'b0;

regrt = 1'b0;

end

endcase

end

6'b100011: // LW instruction

begin

// Set control signals for LW instruction

wreg = 1'b1; // Write to the register file

m2reg = 1'b1; // Write to memory

wmem = 1'b0; // Do not write to memory

aluc = 4'b0010; // ALU operation for addition

aluimm = 1'b1; // ALU source from registers

regrt = 1'b1; // Destination register address

end

// 6'b101011: //SW instruction

// begin

// wreg = 1'b0;

// m2reg = 1'b0;

// wmem = 1'b1;

// aluc = 4'b0000;

// aluimm = 1'b1;

// regrt = 1'b0;

// end

default: // Default behavior for unspecified op values

begin

// Set default control signals here for unspecified op values

// You can assign default values or behavior for cases where op is unspecified.

// For example, you can set all signals to 0.

wreg = 1'b0;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0000;

aluimm = 1'b0;

regrt = 1'b0;

end

endcase

end

endmodule

module RegrtMultiplexer(

// Inputs

input [4:0] rt, // Input register value rt

input [4:0] rd, // Input register value rd

input regrt, // Control signal to select the output (0 for rd, 1 for rt)

// Output

output reg [4:0] destReg // Output register value (selected based on the control signal)

);

always @(\*)

begin

if (regrt == 0)

destReg = rd; // If regrt is 0, select rd as the output.

else

destReg = rt; // If regrt is 1, select rt as the output.

end

endmodule // End of the module

module RegisterFile(

// Inputs

input [4:0] rs, // Input for the source register (rs)

input [4:0] rt, // Input for the target register (rt)

// Outputs

output reg [31:0] qa, // Output for the value stored in the source register

output reg [31:0] qb // Output for the value stored in the target register

);

reg [31:0] register [0:31]; // 32x32 array for registers (register file)

// Initialize all registers to 0

integer r;

initial begin

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

register[r] = 0; // Initialize each register to 0.

end

end

always @ (\*) // Always block to update qa and qb based on the input rs and rt values.

begin

qa = register[rs]; // Output qa is the value stored in the source register (rs).

qb = register[rt]; // Output qb is the value stored in the target register (rt).

end

endmodule // End of the module

module ImmediateExtender( //immediate extender module.

input [15:0] imm,

output reg [31:0] imm32

);

always @ (\*) //always block to update the value of imm32.

begin

imm32 = {{16{imm[15]}}, imm}; //sets imm32 to be equal to imm. the last bit is concatinated to the other 16 bits based on if the sign bit is a zero or one.

end

endmodule

module IDEXEpipeline(

// Inputs

input wreg, // Control signal for writing to the register file

input m2reg, // Control signal for writing to the register file (M2 stage)

input wmem, // Control signal for writing to memory

input [3:0] aluc, // ALU control signal

input aluimm, // ALU immediate value

input [4:0] destReg, // Destination register address

input [31:0] qa, // Value from source register A

input [31:0] qb, // Value from source register B

input [31:0] imm32, // 32-bit immediate value

input clk, // Clock signal

// Outputs

output reg ewreg, // Output for write enable signal

output reg em2reg, // Output for write enable signal (M2 stage)

output reg ewmem, // Output for memory write enable signal

output reg [3:0] ealuc, // Output for ALU control signal

output reg ealuimm, // Output for ALU immediate value

output reg [4:0] edestReg, // Output for destination register address

output reg [31:0] eqa, // Output for source register A value

output reg [31:0] eqb, // Output for source register B value

output reg [31:0] eimm32 // Output for 32-bit immediate value

);

// On the positive edge of the clock, update the output signals with the input values.

always @ (posedge clk)

begin

ewreg = wreg;

em2reg = m2reg;

ewmem = wmem;

ealuc = aluc;

ealuimm = aluimm;

edestReg = destReg;

eqa = qa;

eqb = qb;

eimm32 = imm32;

end

endmodule // End of the module

module ALU(

input [31:0] eqa, // Input A for the ALU

input [31:0] b, // Input B for the ALU

input [3:0] ealuc, // ALU control signal

output reg [31:0] r // Output of the ALU

);

// ALU operation codes

// 0000 - AND

// 0001 - OR

// 0010 - ADD

// 0110 - SUBTRACT

// 0111 - SET LESS THAN

// 1100 - NOR

// 1111 - XOR

always @ (\*)

begin

case(ealuc)

4'b0000: // AND operation

begin

r = eqa & b;

end

4'b0001: // OR operation

begin

r = eqa | b;

end

4'b0010: // ADD operation

begin

r = eqa + b;

end

4'b0110: // SUBTRACT operation

begin

r = eqa - b;

end

4'b1100: // NOR operation

begin

r = ~(eqa | b);

end

4'b1111: // XOR operation

begin

r = eqa ^ b;

end

endcase

end

endmodule // End of the module

module ALUMux(

input [31:0] eqb, // Input B data from the ALU

input [31:0] eimm32, // Immediate value from the pipeline

input ealuimm, // Mux control signal

output reg [31:0] b // Output data selected by the Mux

);

always @(\*) begin

case(ealuimm)

1'b0: // Select eqb as the output

begin

b <= eqb;

end

1'b1: // Select eimm32 as the output

begin

b <= eimm32;

end

endcase

end

endmodule // End of the module

module EXEMEMpipeline(

input ewreg, // Control signal for writing to the register file

input em2reg, // Control signal for writing to the register file (M2 stage)

input ewmem, // Control signal for writing to memory

input [4:0] edestReg, // Destination register address

input [31:0] r, // Result from the ALU

input [31:0] eqb, // Value from source register B

input clk, // Clock signal

output reg mwreg, // Output for write enable signal

output reg mm2reg, // Output for write enable signal (M2 stage)

output reg mwmem, // Output for memory write enable signal

output reg [4:0] mdestReg, // Output for destination register address

output reg [31:0] mr, // Output for result from the ALU

output reg [31:0] mqb // Output for value from source register B

);

always @ (posedge clk)

begin

mwreg = ewreg;

mm2reg = em2reg;

mwmem = ewmem;

mdestReg = edestReg;

mr = r;

mqb = eqb;

end

endmodule // End of the module

module DataMemory(

input [31:0] mr, // Memory read address

input [31:0] mqb, // Data to be written to memory

input mwmem, // Memory write control signal

input clk, // Clock signal

output reg [31:0] mdo // Data read from memory

);

// Define a data memory array with 64 words

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

// Initialize data memory with some values (words 0-9)

initial begin

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: Set mdo to the value at the memory read address (bits 7:2 of mr)

mdo = dataMemory[mr[7:2]];

end

always @(negedge clk) begin

// Writing: If mwmem is 1, write the value in mqb to the memory at the read address

if (mwmem == 1) begin

dataMemory[mr[7:2]] <= mqb;

end

end

endmodule // End of the module

module MEMWBpipeline(

input mwreg, // Control signal for writing to the register file

input mm2reg, // Control signal for writing to the register file (Memory Stage)

input [4:0] mdestReg, // Destination register address

input [31:0] mr, // Result from the data memory

input [31:0] mdo, // Data read from the data memory

input clk, // Clock signal

output reg wwreg, // Output for write enable signal

output reg wm2reg, // Output for write enable signal (Memory stage)

output reg [4:0] wdestReg, // Output for destination register address

output reg [31:0] wr, // Output for result from the data memory

output reg [31:0] wdo // Output for data read from the data memory

);

always @ (posedge clk)

begin

wwreg = mwreg;

wm2reg = mm2reg;

wdestReg = mdestReg;

wr = mr;

wdo = mdo;

end

endmodule // End of the module

**Testbench:**

`timescale 1ns / 1ps

module TestBench;

// Inputs

reg clk;

// Add other input signals here

// Outputs

wire [31:0] pc;

wire [31:0] dinstOut;

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;

//lab 4 - EXEMEM

wire mwreg;

wire mm2reg;

wire mwmem;

wire [4:0] mdestReg;

wire [31:0] mr;

wire [31:0] mqb;

//lab 4 - MEMWB

wire wwreg;

wire wm2reg;

wire [4:0] wdestReg;

wire [31:0] wr;

wire [31:0] wdo;

initial begin

clk <= 1'b0;

end

// Instantiate the Datapath module

Datapath datapath(

.clk(clk),

.pc(pc),

.dinstOut(dinstOut),

.ewreg(ewreg),

.em2reg(em2reg),

.ewmem(ewmem),

.ealuc(ealuc),

.ealuimm(ealuimm),

.edestReg(edestReg),

.eqa(eqa),

.eqb(eqb),

.eimm32(eimm32),

.mwreg(mwreg),

.mm2reg(mm2reg),

.mwmem(mwmem),

.mdestReg(mdestReg),

.mr(mr),

.mqb(mqb),

.wwreg(wwreg),

.wm2reg(wm2reg),

.wdestReg(wdestReg),

.wr(wr),

.wdo(wdo)

);

// Clock generation

always begin

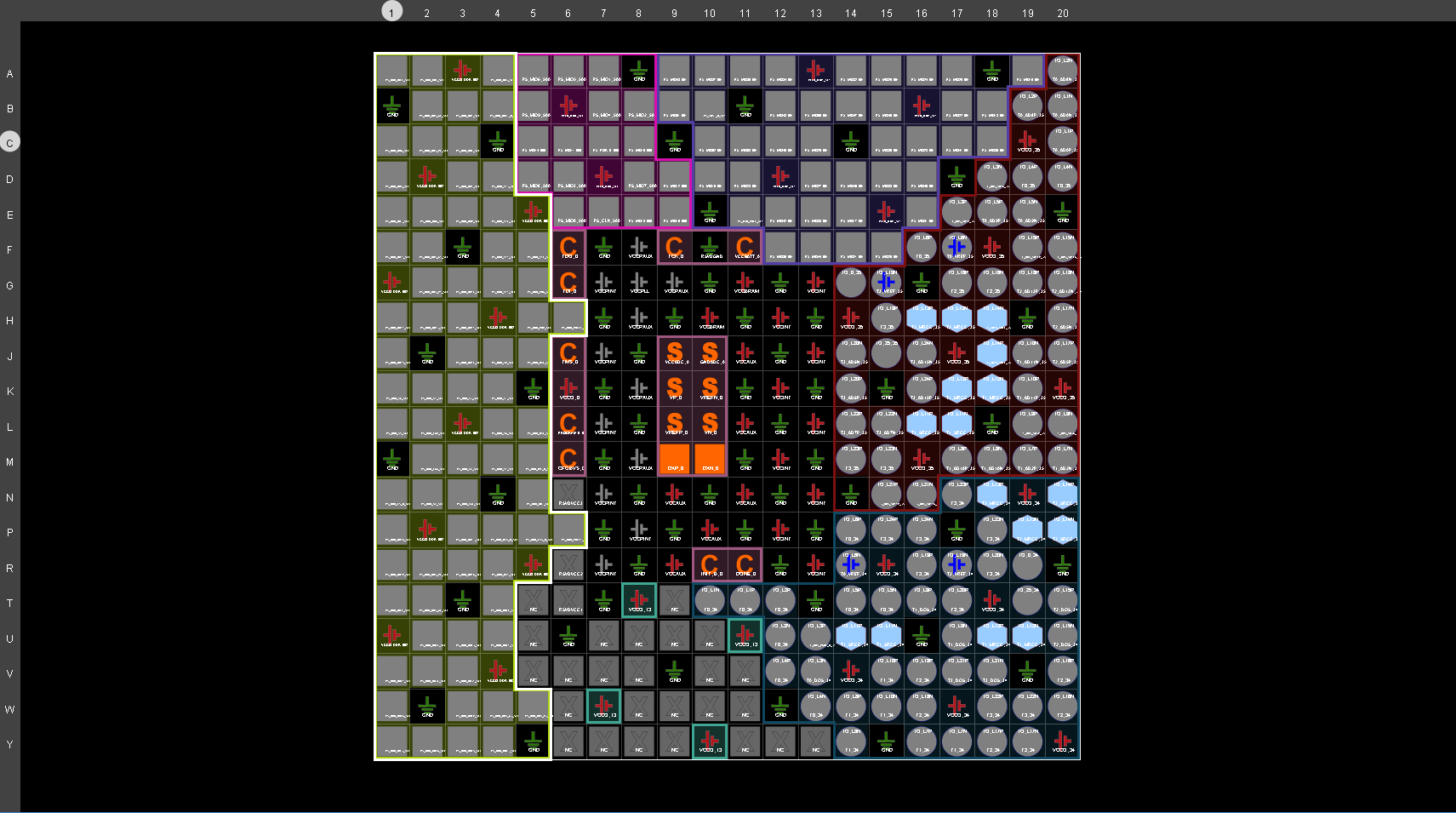
#10;

clk = ~clk;

end

endmodule

**I/O**

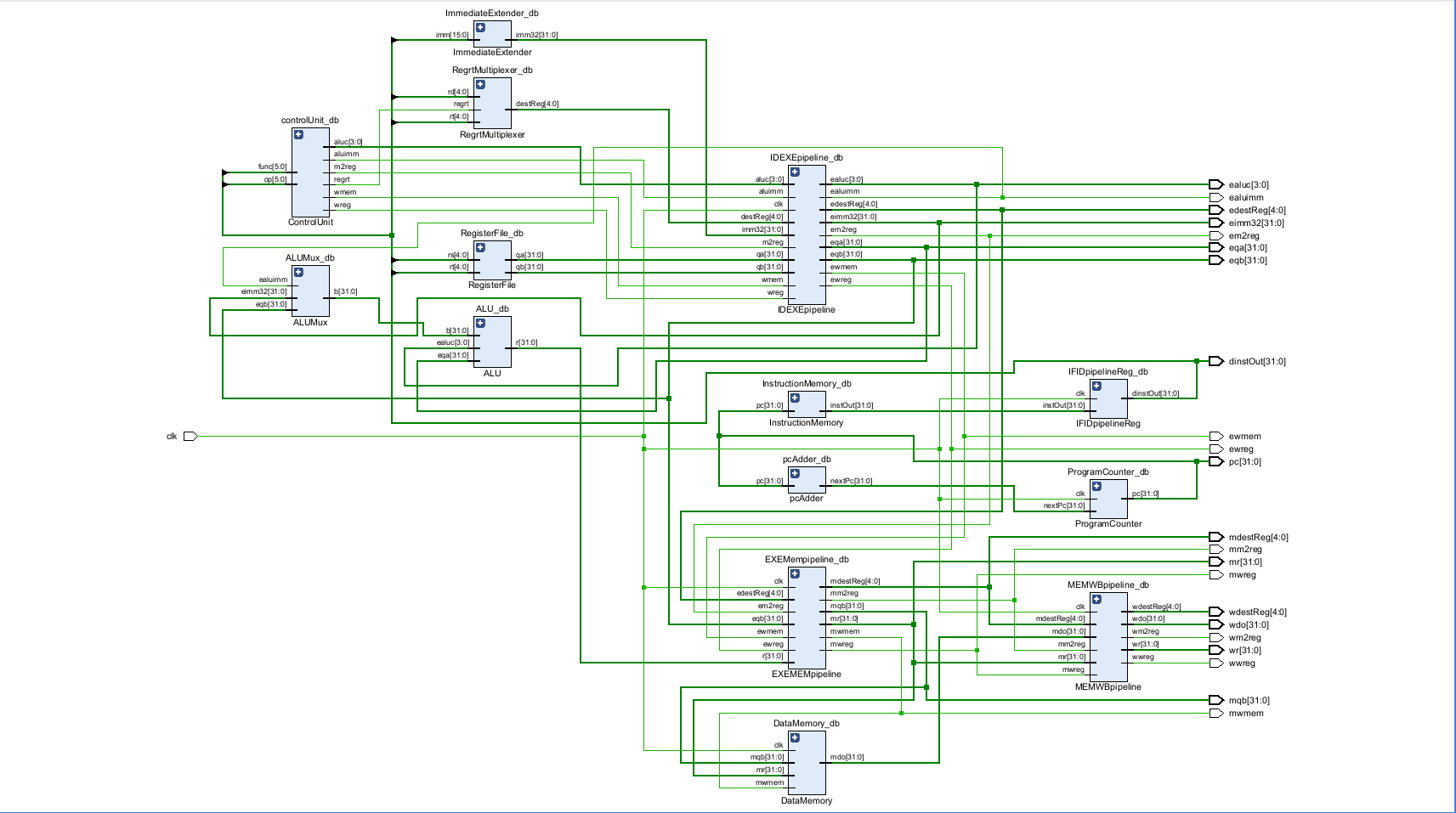
****

**Floorplanning**

**A screen shot of a computer

Description automatically generated**

**Schematic**

**RTL:  
**

Screenshot of the rtl schematic that properly displays each module, outputs and flow of logic.

**Post-Synthesis:**

**A diagram of a computer network

Description automatically generated**

Screenshot of the central part of the synthesis schematic.

**A green pole with a handle

Description automatically generated**

Above is a diagram of the entire synthesis schematic.

**Waveform:**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

Note that the first waveform diagram is the same as the second one, it is just the full waveform for the duration of the test bench. In this waveform output, the 4 load word instructions are loaded into the dinstOut shown in the image. Then, edestReg is “telling” the cpu where the data will be stored in the register. After it passes through to the memory stage of the cpu which continues to keep those destination registers through the memory stage. Wdo is the given values stored into data memory that will be used in future labs to perform writing back, those will be the values used to write back to the register. Here, this waveform displays proper pipelining and handling of each logical value.