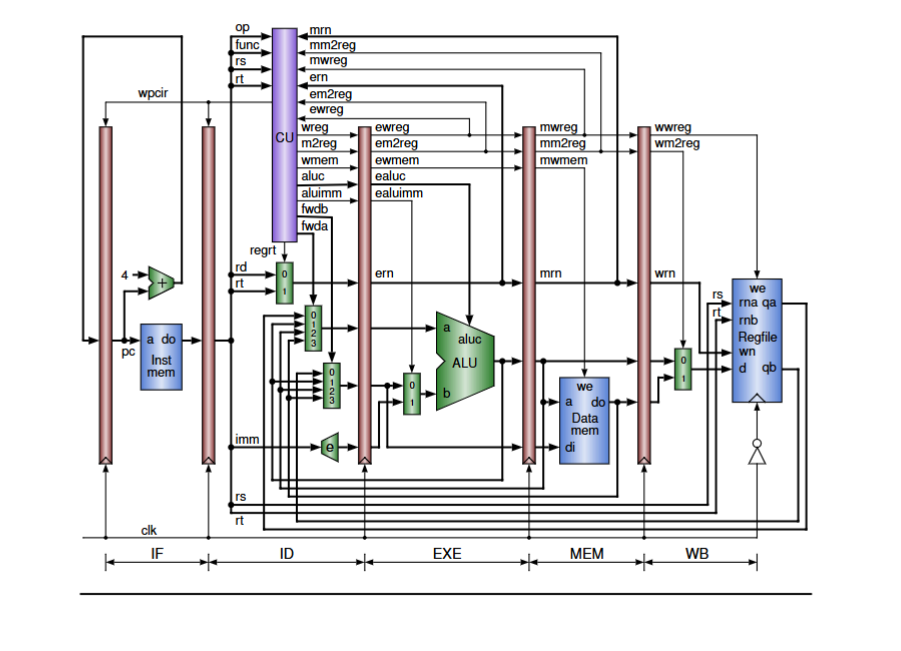
CMPEN 331 PROJECT REPORT

NAME: SHREYA MANKARE

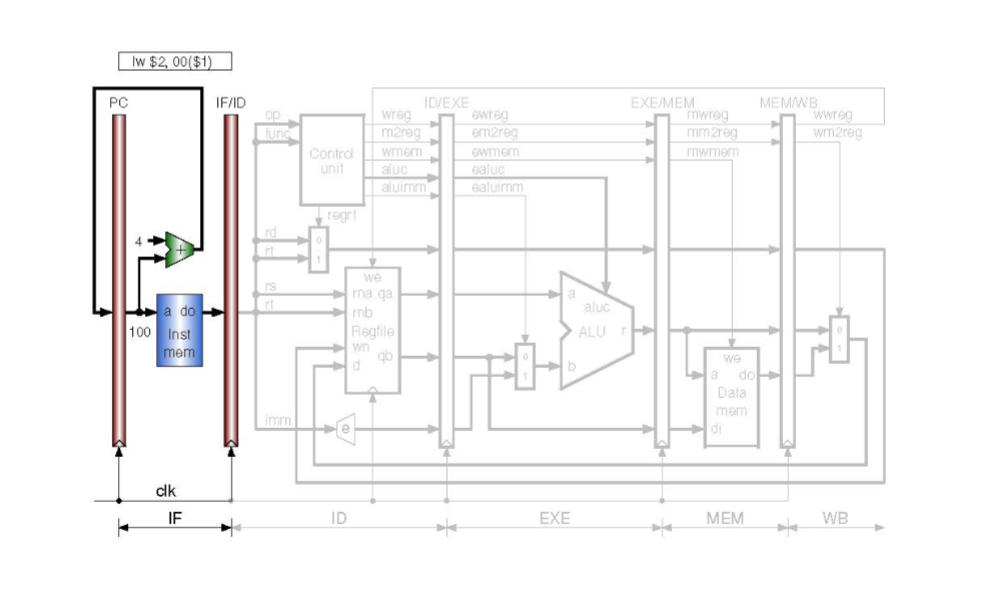
SECTION: 2

**ABSTRACT**

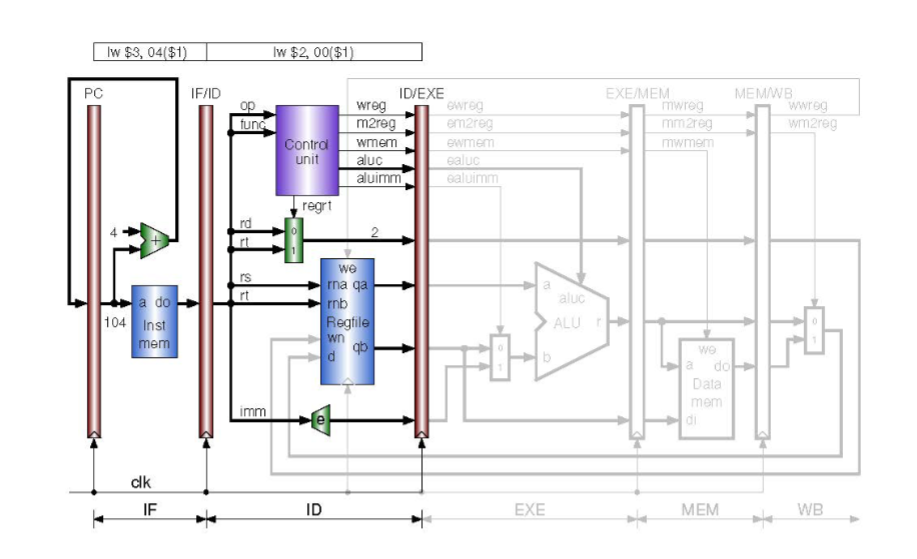
This project will focus on expanding on what was done in Lab 5. Firstly, the Register File will be put in the WB (Write Back) stage so that the data can be read in the ID stage right after it is written in the WB stage. However, certain data hazards may occur while doing so, which is why we may need to perform forwarding. In order to do so, we have introduced the ForwardA and ForwardB multiplexers. To execute the new instructions, changes were made in the Instruction Memory too. The Register File was modified as well by initializing the first 11 words of the register with certain HEX values.



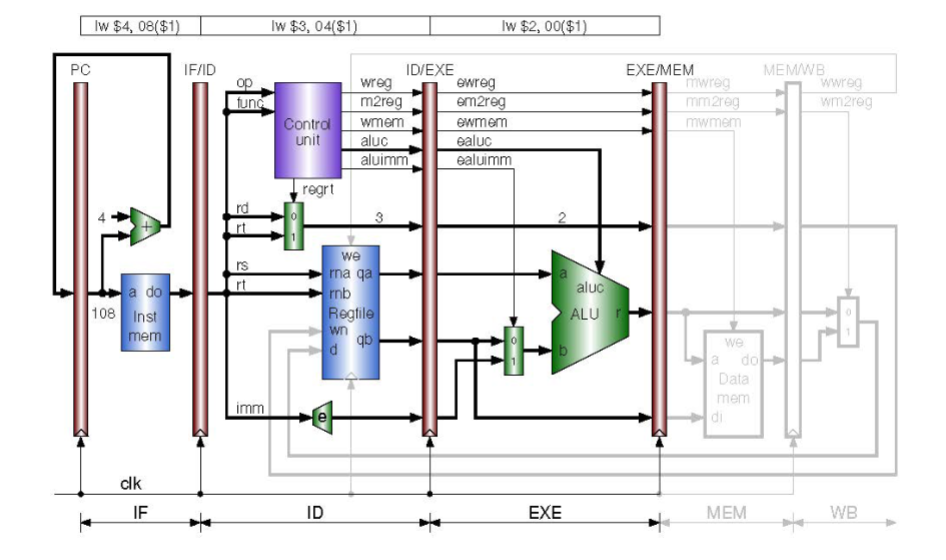
**INTRODUCTION**

****

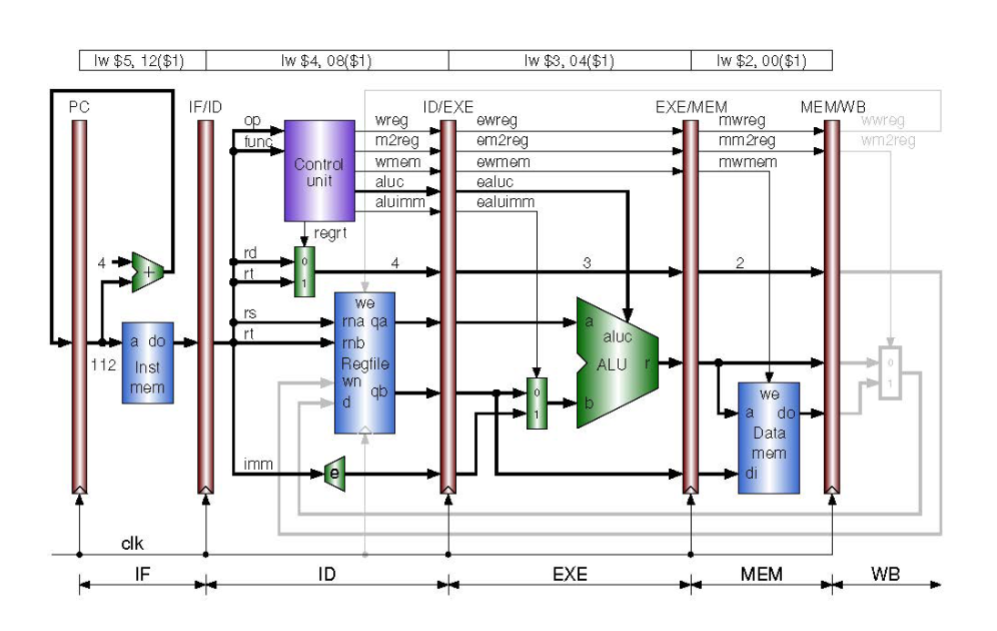
In the IF stage, the instructions are fetched. The IF stage contains the instruction memory and an adder between the two pipeline registers, where the left most pipeline register is PC. After the first cycle, the instruction fetched from the instruction memory is written into the IF/ID pipeline register. The output of the adder, that is PC = PC + 4, is written into PC.



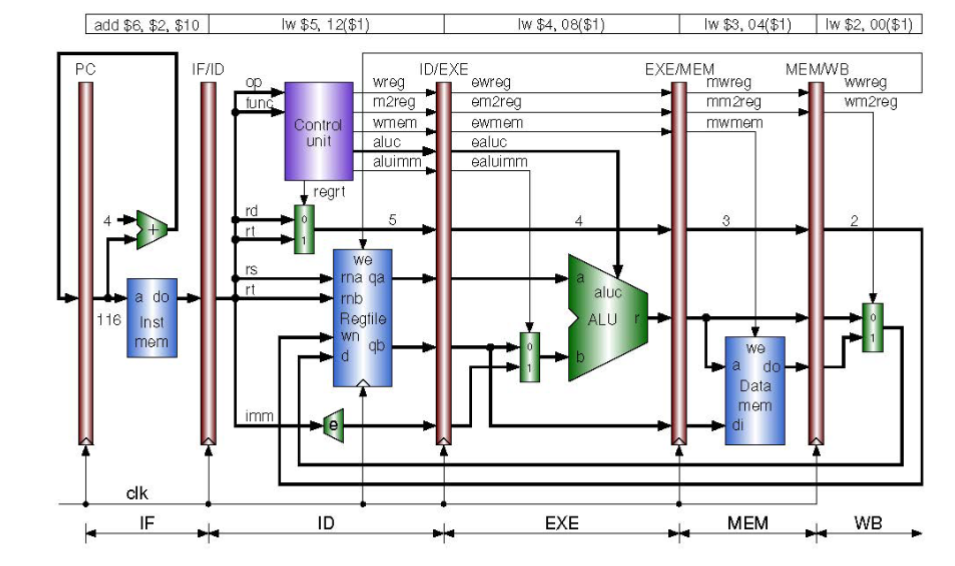
The ID stage consists of the Control Unit, register multiplexer, Register file and the immediate extender. In the second cycle, the first instruction enters the ID stage. While the first instruction is being decoded in the ID stage, the second instruction is fetched in the IF stage. Two operands are read from the Register file based on rs and rt. The immediate extender sign-extends the immediate value to 32 bits, depending on the instruction type. The register multiplexer chooses the destination register. The rest of the signals are written into the ID/EXE pipeline register.



In the third cycle, the first instruction enters the EXE stage. The ALU multiplexer selects the immediate. The ALU then performs the addition. During this cycle, the third instruction is fetched in the IF stage and the second instruction is decoded in the ID stage. At the end of the cycle, all the four pipeline registers are updated.



In the fourth cycle, the first instruction enters the MEM stage. In this stage, data is read from the data memory if required. During this cycle, the fourth instruction is fetched in the IF stage, the third instruction is decoded in the ID stage and the second instruction enters the EXE stage. The five pipeline registers are updated at the end of this cycle.



In the fifth cycle, the first instruction enters the WB stage. The multiplexer selects the data memory which is written to the register file at the end of the cycle. During this cycle, the fifth instruction is fetched in the IF stage, the fourth instruction is decoded in the ID stage, the third instruction is in the EXE stage and the second instruction is in the MEM stage. All six pipeline registers, the sixth one being the destination register, are updated at the end of the cycle. The first instruction is then committed.

Advantages of pipelining on computer architecture:  
1. Since pipelining allows more work to be completed in a shorter time, it ensured that the throughput is increased.

2. Due to increased throughput, the overall performance of the processor is enhanced.

3. Waiting time is reduced since the instructions are moving through the pipelines continuously.

4. Processors can work at higher clock frequencies since work is done faster.

**VERILOG DESIGN CODE:**

**Module 1: PC**

module PC(input clock, input [31:0] nextPC, output reg [31:0] pc);

initial begin

pc = 32'd100;

end

always @ (posedge clock)

begin

pc <= nextPC;

end

endmodule

**Module 2: PC Adder**

module PC\_Adder(input [31:0] pc, output reg [31:0] nextPC);

always @ (\*)

begin

nextPC = pc + 32'h00000004;

end

endmodule

**Module 3: Instruction Memory**

module Inst\_mem(input [31:0] pc, output reg [31:0] instOut);

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

initial begin

memory[25] = 32'b00000000001000100001100000100000;

memory[26] = 32'b00000001001000110010000000100010;

memory[27] = 32'b00000000011010010010100000100101;

memory[28] = 32'b00000000011010010011000000100110;

memory[29] = 32'b00000000011010010011100000100100;

end

always @ (\*)

begin

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

end

endmodule

**Module 4: IFID Pipeline Register**

module IFID\_Pipeline(input [31:0] instOut, input clock, output reg [31:0] dinstOut);

always @ (posedge clock)

begin

dinstOut = instOut;

end

endmodule

**Module 5: Control Unit**

module Control\_Unit(

input [5:0] op,

input [5:0] func,

input [4:0] rs, rt, ern, mrn,

input mm2reg, mwreg, em2reg, ewreg,

output reg [1:0] fwda, fwdb,

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

output reg [3:0] aluc

);

always @(\*) begin

case (op)

6'b000000: //R-Type

begin

case (func)

6'b100000: //ADD

begin

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0010;

aluimm = 1'b0;

regrt = 1'b0;

end

6'b100010: //SUB

begin

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0110;

aluimm = 1'b0;

regrt = 1'b0;

end

6'b100101: //OR

begin

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0001;

aluimm = 1'b0;

regrt = 1'b0;

end

6'b100110: //XOR

begin

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0011;

aluimm = 1'b0;

regrt = 1'b0;

end

6'b100100: //AND

begin

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

aluc = 4'b0000;

aluimm = 1'b0;

regrt = 1'b0;

end

endcase

end

6'b100011: // lw

begin

wreg = 1'b1;

m2reg = 1'b1;

wmem = 1'b0;

aluc = 4'b0010;

aluimm = 1'b1;

regrt = 1'b1;

end

endcase

if ((ewreg != 1'b0) && (ern != 5'b0) && (ern == rs)) begin

fwda = 2'b01;

end

else if ((mwreg != 1'b0) && (mm2reg == 1'b0) && (mrn != 5'b0) && (mrn == rs)) begin

fwda = 2'b10;

end

else if ((mwreg != 1'b1) && (mm2reg == 1'b1) && (mrn != 5'b0) && (mrn == rs)) begin

fwda = 2'b11;

end

else begin

fwda = 2'b00;

end

if ((ewreg != 1'b0) && (ern != 5'b0) && (ern == rt)) begin

fwdb = 2'b01;

end

else if ((mwreg == 1'b1) && (mm2reg == 1'b0) && (mrn != 5'b0) && (mrn == rt)) begin

fwdb = 2'b10;

end

else if ((mwreg != 1'b1) && (mm2reg == 1'b1) && (mrn != 5'b0) && (mrn == rt)) begin

fwdb = 2'b11;

end

else begin

fwdb = 2'b00;

end

end

endmodule

**Module 6: Register Multiplexer**

module Regrt\_Multiplexer(input [4:0] rt, input [4:0] rd, input regrt, output reg [4:0] destReg);

always @ (\*)

begin

case (regrt)

0:

begin

destReg = rd;

end

1:

begin

destReg = rt;

end

endcase

end

endmodule

**Module 7: Register File**

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 [0:31];

integer i;

initial begin

registers[0] = 32'b00000000000000000000000000000000;

registers[1] = 32'b10100000000000000000000010101010;

registers[2] = 32'b00010000000000000000000000010001;

registers[3] = 32'b00100000000000000000000000100010;

registers[4] = 32'b00110000000000000000000000110011;

registers[5] = 32'b01000000000000000000000001000100;

registers[6] = 32'b01010000000000000000000001010101;

registers[7] = 32'b01100000000000000000000001100110;

registers[8] = 32'b01110000000000000000000001110111;

registers[9] = 32'b10000000000000000000000010001000;

registers[10] = 32'b10010000000000000000000010011001;

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 8: ForwardA Multiplexer**

module FwdA\_Mux(

input [1:0] fwda, input [31:0] qa, input [31:0] r, input [31:0] mr, input [31:0] do, output reg [31:0] da);

always @(\*) begin

case (fwda)

2'b00: begin da = qa; end

2'b01: begin da = r; end

2'b10: begin da = mr; end

2'b11: begin da = do; end

endcase

end

endmodule

**Module 9: ForwardB Multiplexer**

module FwdB\_Mux(input [1:0] fwdb, input [31:0] qb, input [31:0] r, input [31:0] mr, input [31:0] do, output reg [31:0] db);

always @(\*) begin

case (fwdb)

2'b00: begin db = qb; end

2'b01: begin db = r; end

2'b10: begin db = mr; end

2'b11: begin db = do; end

endcase

end

endmodule

**Module 10: Immediate Extender**

module Immediate\_Extender(input [15:0] imm, output reg [31:0] imm32);

always @ (\*)

begin

if (imm[15] == 0) begin

imm32 = {16'h0000, imm};

end

else

begin

imm32 = {16'hffff, imm};

end

end

endmodule

**Module 9: IDEXE Pipeline Register**

module IDEXE\_Pipeline\_Reg(

input wreg, m2reg, wmem, aluimm, clock,

input [3:0] aluc,

input [4:0] destReg,

input [31:0] qa,

input [31:0] qb,

input [31:0] imm32,

output reg ewreg, em2reg, ewmem, ealuimm,

output reg [3:0] ealuc,

output reg [4:0] edestReg,

output reg [31:0] eqa,

output reg [31:0] eqb,

output reg [31:0] eimm32

);

always @ (posedge clock)

begin

ewreg <= wreg;

em2reg <= m2reg;

ewmem <= wmem;

ealuc <= aluc;

ealuimm <= aluimm;

edestReg <= destReg;

eqa <= qa;

eqb <= qb;

eimm32 <= imm32;

end

endmodule

**Module 11: ALU Multiplexer**

module Alu\_Mux(input [31:0] eqb, input [31:0] eimm32, input ealuimm, output reg [31:0] b);

always @ (\*)

begin

case (ealuimm)

0:

begin

b = eqb;

end

1:

begin

b = eimm32;

end

endcase

end

endmodule

**Module 12: ALU**

module Alu(input [31:0] eqa, input [31:0] b, input [3:0] ealuc, output reg [31:0] r);

always @ (\*)

begin

case (ealuc)

4'b0000: begin r = eqa & b; end

4'b0001: begin r = eqa | b; end

4'b0010: begin r = eqa + b; end

4'b0110: begin r = eqa - b; end

4'b0111: begin r = eqa < b ? 1 : 0; end

4'b1100: begin r = ~(eqa | b); end

endcase

end

endmodule

**Module 13: EXEMEM Pipeline Register**

module EXEMEM\_Pipeline\_Register(

input ewreg, em2reg, ewmem,

input [4:0] edestReg,

input [31:0] r,

input [31:0] eqb,

input clock,

output reg mwreg, mm2reg, 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 <= r;

mqb <= eqb;

end

endmodule

**Module 14: Data Memory**

module Data\_Memory(

input [31:0] mr,

input [31:0] mqb,

input mwmem, clk,

output reg [31:0] mdo);

reg [31:0] data\_memory [0:63];

initial begin

data\_memory[0] = 32'b10100000000000000000000010101010;

data\_memory[1] = 32'b00010000000000000000000000010001;

data\_memory[2] = 32'b00100000000000000000000000100010;

data\_memory[3] = 32'b00110000000000000000000000110011;

data\_memory[4] = 32'b01000000000000000000000001000100;

data\_memory[5] = 32'b01010000000000000000000001010101;

data\_memory[6] = 32'b01100000000000000000000001100110;

data\_memory[7] = 32'b01110000000000000000000001110111;

data\_memory[8] = 32'b10000000000000000000000010001000;

data\_memory[9] = 32'b10010000000000000000000010011001;

end

always @ (\*)

begin

mdo = data\_memory[mr[7:2]];

end

always @ (negedge clk)

begin

if (mwmem == 1) begin

data\_memory[mr[7:2]] <= mqb;

end

end

endmodule

**Module 15: MEMWB Pipeline Register**

module MEMWB\_Pipeline\_Register(

input mwreg, mm2reg,

input [4:0] mdestReg,

input [31:0] mr,

input [31:0] mdo,

input clock,

output reg wwreg, 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 16: Writeback Multiplexer**

module WbMux(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 17: Datapath**

module Datapath(

input clock,

output wire [31:0] pc,

output wire [31:0] dinstOut,

output wire ewreg, em2reg, ewmem, ealuimm,

output wire [3:0] ealuc,

output wire [4:0] ern,

output wire [31:0] eqa,

output wire [31:0] eqb,

output wire [31:0] eimm32,

output wire mwreg, mm2reg, mwmem,

output wire [4:0] mrn,

output wire [31:0] mr,

output wire [31:0] mqb,

output wire wwreg, wm2reg,

output wire [4:0] wrn,

output wire [31:0] wr,

output wire [31:0] wdo,

output wire [31:0] wbData,

output wire [1:0] fwda,

output wire [1:0] fwdb,

output wire [31:0] qa\_wire,

output wire [31:0] qb\_wire,

output wire [31:0] fwva,

output wire [31:0] fwvb

);

wire [31:0] pc\_address;

wire [31:0] instmem\_out;

wire [5:0] op\_wire;

wire [5:0] func\_wire;

wire regrt\_wire;

wire [4:0] rs\_wire;

wire [4:0] rt\_wire;

wire [4:0] rd\_wire;

wire [15:0] imm\_wire;

wire [31:0] imm32\_wire;

wire wreg\_wire, m2reg\_wire, wmem\_wire, aluimm\_wire;

wire [3:0] aluc\_wire;

wire [4:0] destReg;

wire [31:0] b;

wire [31:0] r;

wire [31:0] mdo;

PC PC(clock, pc\_address, pc);

PC\_Adder PC\_Adder(pc, pc\_address);

Inst\_mem Inst\_mem(pc, instmem\_out);

IFID\_Pipeline IFID\_Pipeline(instmem\_out, clock, dinstOut);

assign op\_wire = dinstOut[31:26];

assign rs\_wire = dinstOut[25:21];

assign rt\_wire = dinstOut[20:16];

assign rd\_wire = dinstOut[15:11];

assign func\_wire = dinstOut[5:0];

assign imm\_wire = dinstOut[15:0];

Control\_Unit Control\_Unit(op\_wire, func\_wire, rs\_wire, rt\_wire, ern, mrn, mm2reg, mwreg, em2reg, ewreg, fwda, fwdb, wreg\_wire, m2reg\_wire, wmem\_wire, aluimm\_wire, regrt\_wire, aluc\_wire);

Regrt\_Multiplexer Regrt\_Multiplexer(rt\_wire, rd\_wire, regrt\_wire, destReg);

Regfile Regfile(rs\_wire, rt\_wire, wrn, wbData, wwreg, clock, qa\_wire, qb\_wire);

FwdA\_Mux FwdA\_Mux(fwda, qa\_wire, r, mr, mdo, fwva);

FwdB\_Mux FwdB\_Mux(fwdb, qb\_wire, r, mr, mdo, fwvb);

Immediate\_Extender Immediate\_Extender(imm\_wire, imm32\_wire);

IDEXE\_Pipeline\_Reg IDEXE\_Pipeline\_Reg(wreg\_wire, m2reg\_wire, wmem\_wire, aluimm\_wire, clock, aluc\_wire, destReg, fwva, fwvb, imm32\_wire, ewreg, em2reg, ewmem, ealuimm, ealuc, ern, eqa, eqb, eimm32);

Alu\_Mux Alu\_Mux(eqb, eimm32, ealuimm, b);

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

EXEMEM\_Pipeline\_Register EXEMEM\_Pipeline\_Register(ewreg, em2reg, ewmem, ern, r, eqb, clock, mwreg, mm2reg, mwmem, mrn, mr, mqb);

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

MEMWB\_Pipeline\_Register MEMWB\_Pipeline\_Register(mwreg, mm2reg, mrn, mr, mdo, clock, wwreg, wm2reg, wrn, wr, wdo);

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

endmodule

**Testbench Code:**

module testbench();

reg clk;

wire [31:0] pc;

wire [31:0] dinstOut;

wire [31:0] eqa;

wire [31:0] eqb;

wire [31:0] eimm32;

wire [31:0] mr;

wire [31:0] mqb;

wire [31:0] wr;

wire [31:0] wdo;

wire [31:0] wbData;

wire ewreg, em2reg, ewmem;

wire ealuimm;

wire mwreg, mm2reg, mwmem;

wire wwreg, wm2reg;

wire [3:0] ealuc;

wire [4:0] edestReg;

wire [4:0] mdestReg;

wire [4:0] wdestReg;

wire [1:0] fwda;

wire [1:0] fwdb;

wire [31:0] qa;

wire [31:0] qb;

wire [31:0] fwva;

wire [31:0] fwvb;

initial begin

clk = 0;

end

Datapath Datapath(clk, pc, dinstOut, ewreg, em2reg, ewmem, ealuimm, ealuc, edestReg, eqa, eqb, eimm32, mwreg, mm2reg, mwmem, mdestReg, mr, mqb, wwreg, wm2reg, wdestReg, wr, wdo, wbData, fwda, fwdb, qa, qb, fwva, fwvb);

always begin

#1;

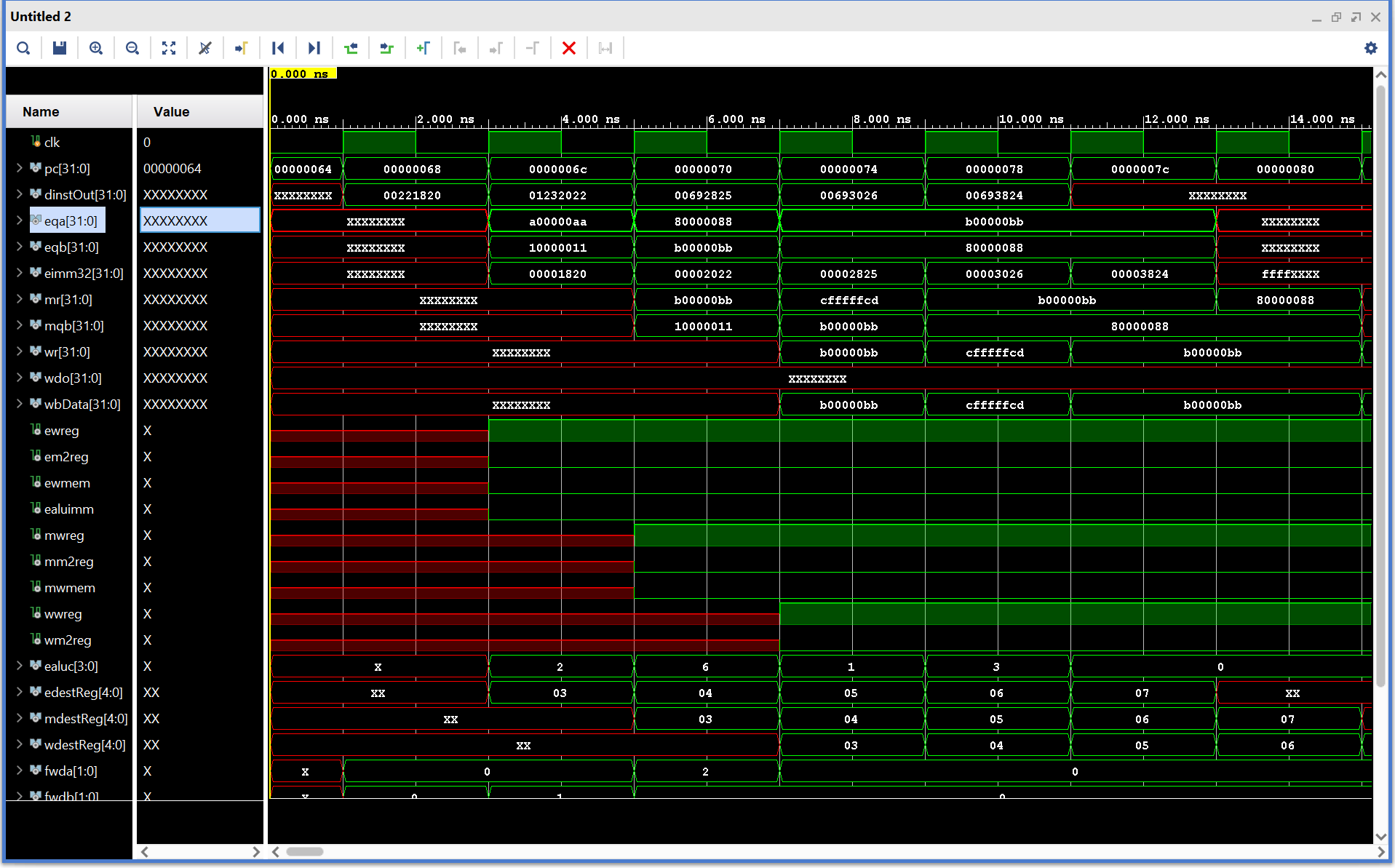
clk = ~clk;

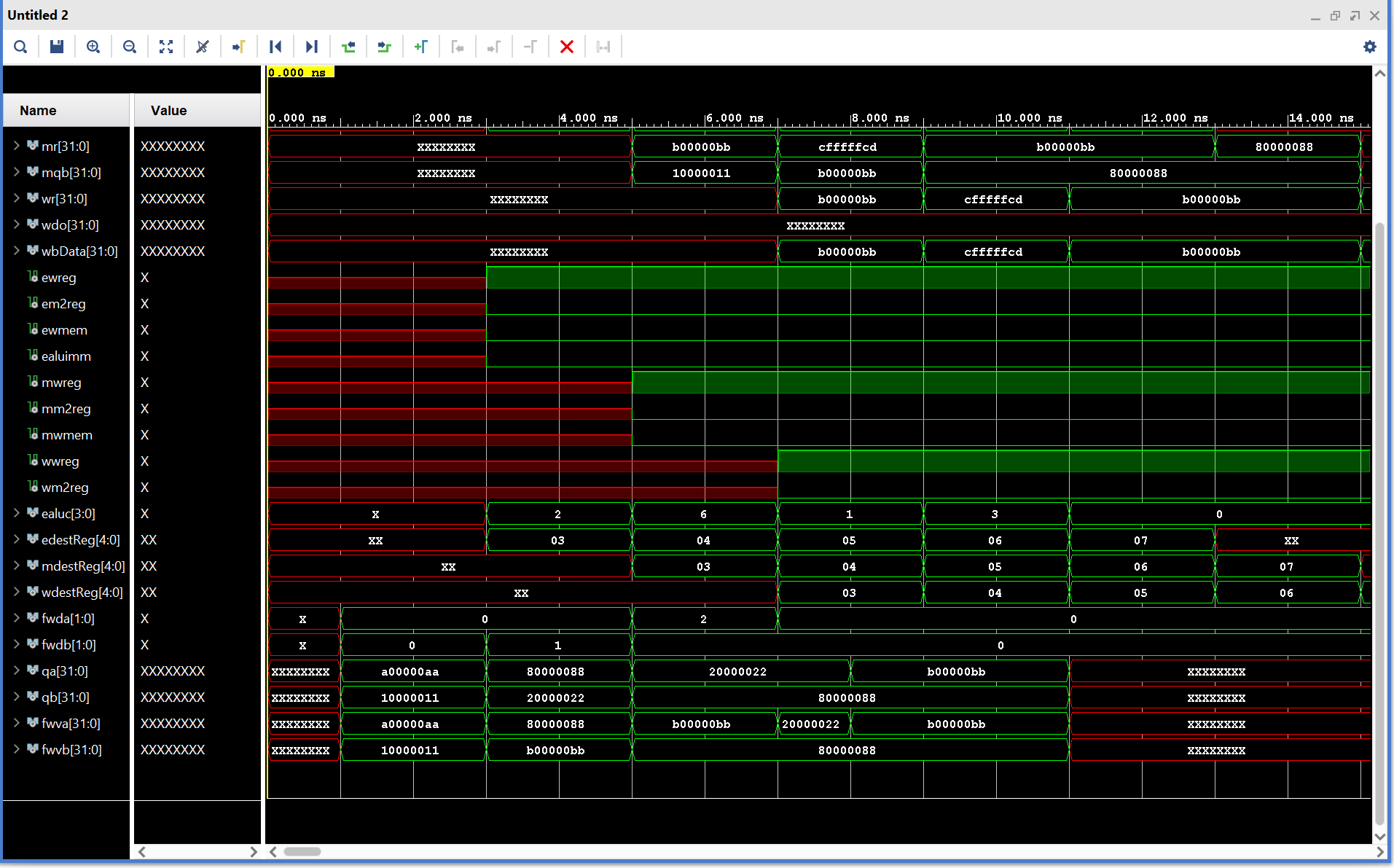
end

endmodule

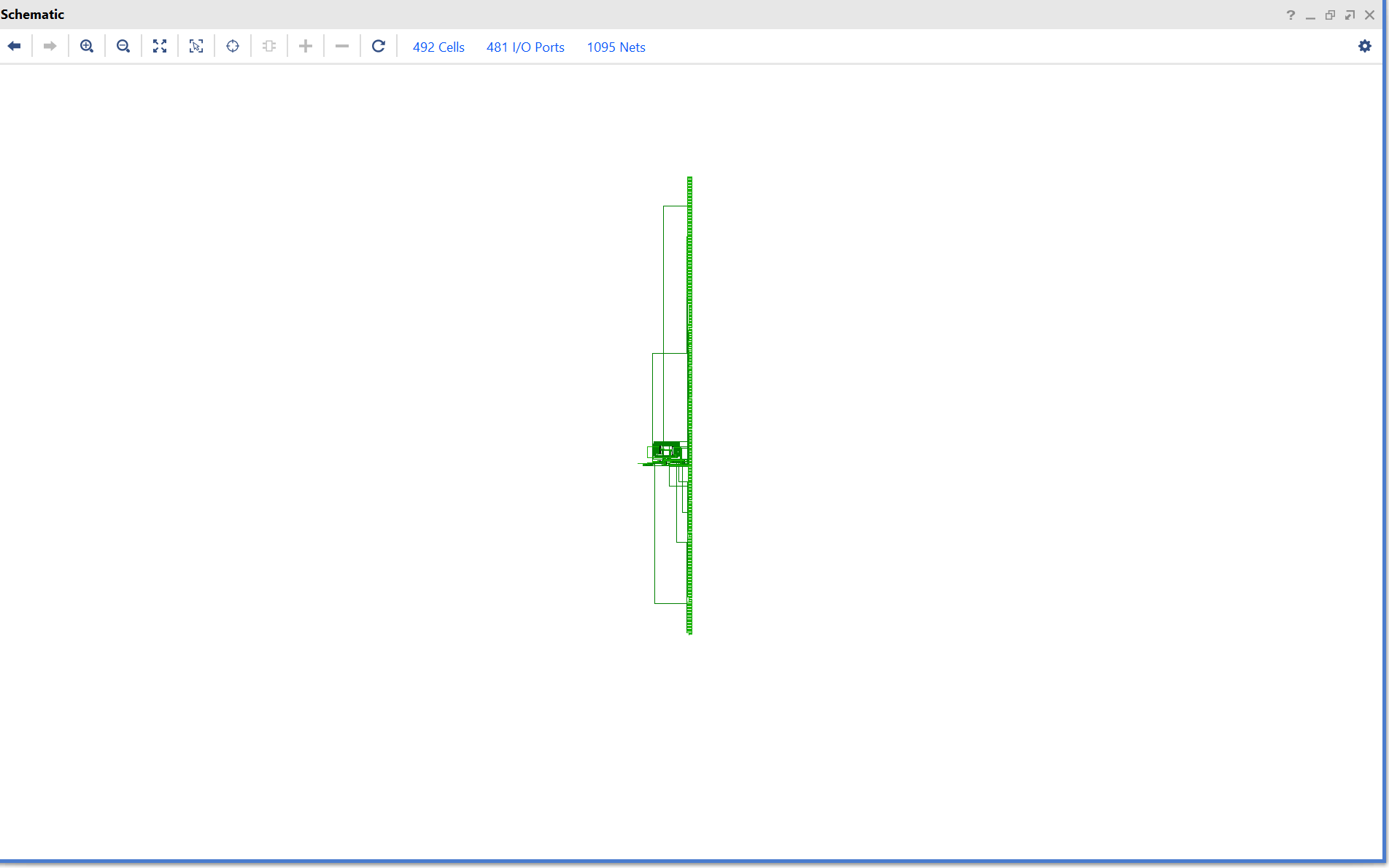
----------------------------------------------------------------------------------------------

Screenshot of the waveform:

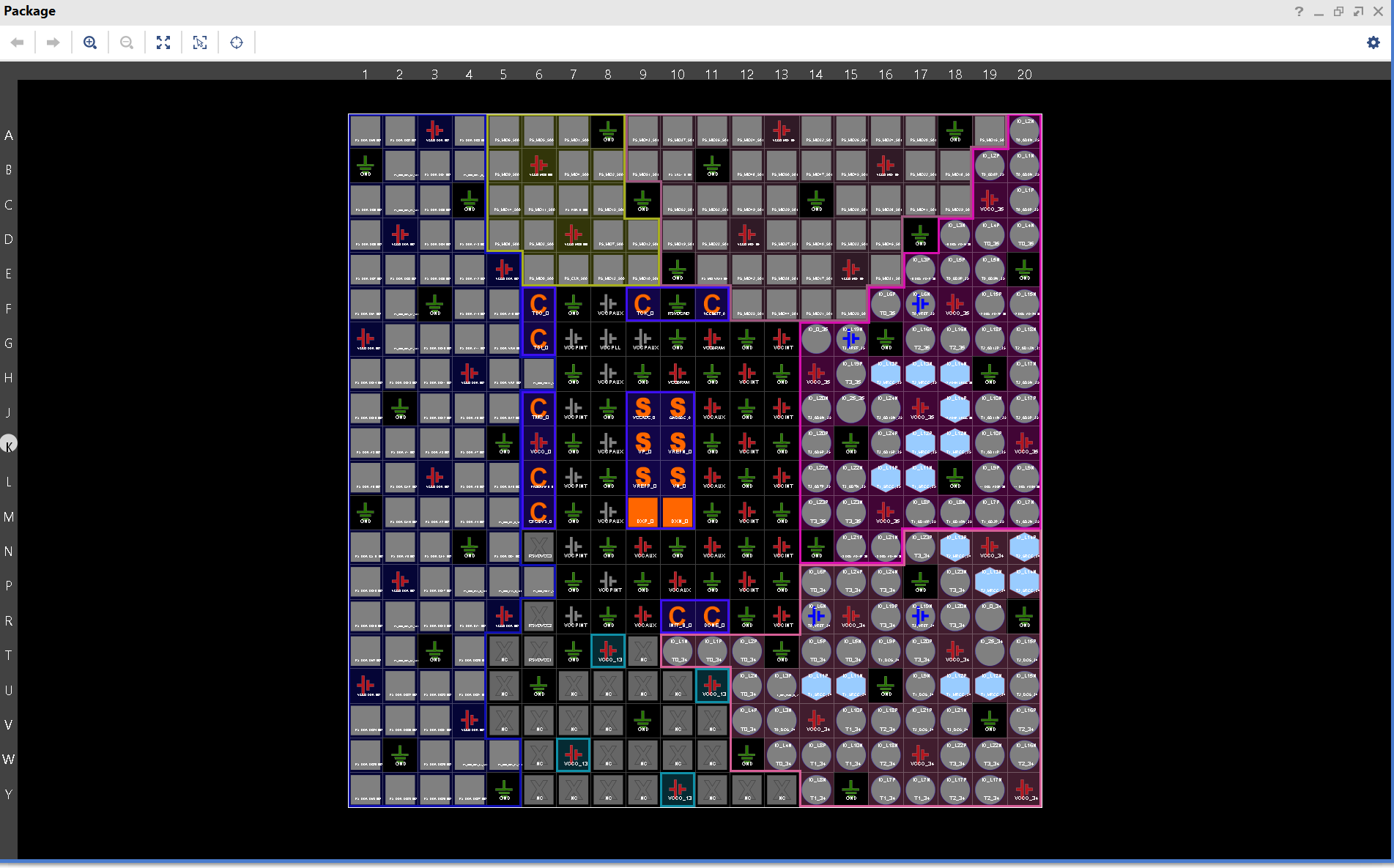




Screenshot of the design schematic:



Screenshot of the I/O planning:



Screenshot of the floor planning:

