# Final Project

## CMPEN 331

### Sparsh Jha

#### Device: xc7z030fbg676-1

#### Date: 16th December 2022

## Abstract

In this project, we will be simulating and implementing a MIPS based CPU using Verilog in Xilinix Vivado. We use the 5-Staged processor pipelining alongside ALU-ALU and MEM-ALU Forwarding to integrate add, sub, AND, OR, and XOR instructions.

## Introduction

In this project we will divide the implementation into 5 blocks, that are able to function independently, when given inputs. They are, in order of their implementation: InstructionFetch, InstructionDecoding, EXEcution, MEMory, WriteBack. Stages.

### Instruction Fetch Stage:

This stage consists of 3 components and 1 forwarding register. The PC register, the PC adder, and the Instruction Memory. The PC register stores the current position of the pointer of the program and outputs the current position. The PC adder increments the PC register by 4. The Instruction Memory takes the output from the PC register. When the signal of the PC register changes, it retrieves the instruction in the current memory pointer as its output. This output is then sent to the input of the IF/ID Forwarding register.

### Instruction Decoding stage:

This stage consists of 6 different components. There are 3 Multiplexers, a Control Unit, the Register Memory and an immediate sign extender. The Control Unit takes input from the IF/ID forwarding register as well as inputs from other forwarding registers. Based on the input instruction, it decides what actions happen in the later stages of the CPU Cycle. In this stage, it gives output for the signals to the ID/EXE registers and the selector outputs for the multiplexers.

The first multiplexer is a 2:1 mux where it decides the destination Register for the instructions. It takes the selector input from the Control Unit and changes the output accordingly.

The next two multiplexers decide on the two inputs to the forwarding register to the inputs of the ALU. These are present as a means to change the inputs of the ALU when there is a forwarding possible. The control Unit detects whether there is forwarding and the multiplexer changes accordingly. This allows us to avoid stalls, and further optimize the pipelining.

The register memory takes the input instruction from the IF/ID forwarding register and processes it to return the values contained in the register positions stored in the instruction. Additionally, It has a functionality of updating the values stored in the registers, with the control signal being output from the Control Unit. The writing back values are provided by wires from the very last stage of the cycle. The values get updated only in the 2nd half of the cycle, so they are available to other stages by positive clock edge.

### Execution Stage:

In this stage, there are 2 components. The ALU and one multiplexer.

The multiplexer takes the input from the forwarding register and choses between the sign extended immediate value or the value from the register memory. The selector input is dependent on the outputs from the Control Unit from the previous stage through the forwarding register.

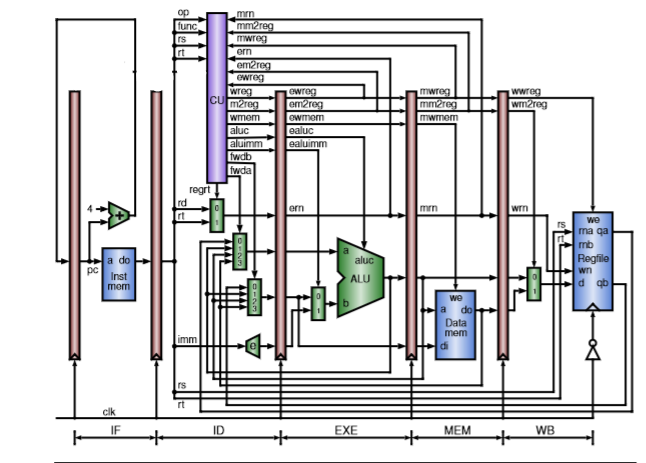
The ALU is an Arithmetic Logic Unit. It takes 3 inputs and has 1 output. It takes 2 values and completes an operation defined by the outputs from the forwarding register provided by the Control Unit.

### Memory Stage:

This stage only consists of 1 component, The Data Memory. The DM stores the values or data provided by the ALU and the register Memory. The DM only stores the value from the inputs when the control signal is 1. This signal is provided by the Control Unit and is dependent on the instruction. The Output is then sent to the next forwarding register, MEM/WB register.

### Write Back Stage:

This stage consists of only 1 Multiplexer. This mux decides whether the value to be written back into the register Memory is the result from the ALU operation, or the data retrieved from the Data Memory. The destination register has been continuously forwarded from ID stage, and it is now input to the Register Memory as the location where the output from the Mux is going to be stored.



# Verilog Code:

`timescale 1ns / 1ps

module PC (

input [31:0] nextPC,

input clk,

output reg [31:0] PC

);

always @(posedge clk) begin

PC = nextPC;

end

endmodule

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

initial

nextPC = 100;

always @(\*) begin

nextPC <= PC+4;

end

endmodule

module insMem(input [31:0] PC, output reg [31:0] instOut);

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

initial begin

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};

memory[25] = {6'b000000, 5'b00001, 5'b00010, 5'b00011, 5'b00000, 6'b100000}; // $3 = $1+$2

memory[26] = {6'b000000, 5'b01001, 5'b00011, 5'b00100, 5'b00000, 6'b100010}; // $4 = $9 - $3

memory[27] = {6'b000000, 5'b00011, 5'b01001, 5'b00101, 5'b00000, 6'b100101}; // $5 = $3 OR $9

memory[28] = {6'b000000, 5'b00011, 5'b01001, 5'b00110, 5'b00000, 6'b100110}; // $6 = $3 XOR $9

memory[29] = {6'b000000, 5'b00011, 5'b01001, 5'b00111, 5'b00000, 6'b100100}; // $7 = $3 AND $9

end

always @(\*) begin

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

end

endmodule

module ifidPipelineReg(input [31:0] instOut, input clk, output reg [31:0] dinstOut);

always @(posedge clk) begin

dinstOut = instOut;

end

endmodule

module controlUnit(

input [5:0] op,

input [5:0] func,

input [4:0] rs,

input [4:0] rt,

input [4:0] mdestreg,

input mm2reg,

input mwreg,

input [4:0] edestreg,

input em2reg,

input ewreg,

output reg [1:0] fwda,

output reg [1:0] fwdb,

output reg wreg,

output reg m2reg,

output reg wmem,

output reg [3:0] ALUc,

output reg ALUimm,

output reg regrt

);

initial begin

fwda =0;

fwdb = 0;

end

always @(\*) begin

if((rt == edestreg) && (ewreg == 1))

fwdb <= 1;

else if ((rt ==mdestreg) && (mwreg == 1) && (mm2reg == 0))

fwdb <= 2;

else if ((rt == mdestreg) && (mwreg == 1) && (mm2reg == 1))

fwdb <= 3;

else

fwdb <= 0;

if((rs == edestreg) && (ewreg == 1))

fwda <= 1;

else if((rs == mdestreg) && (mm2reg == 0)&& (mwreg == 1))

fwda <= 2;

else if((rs == mdestreg) && (mm2reg == 1)&& (mwreg == 1))

fwda <= 3;

else

fwda <= 0;

case(op)

6'b000000: //r-types

begin

case (func)

6'b100000: //Add instruction

begin

wreg <= 1;

m2reg <= 0;

wmem <= 0;

ALUc <= 2;

ALUimm <= 0;

regrt = 0;

end

6'b100010: //Sub inst

begin

wreg <= 1;

m2reg <=0;

wmem <=0;

ALUc <= 6;

ALUimm <=0;

regrt = 0;

end

6'b100100: //and instructions

begin

ALUc <= 0;

wreg <= 1;

m2reg <=0;

wmem <=0;

ALUimm <=0;

regrt =0;

end

6'b100101: //OR

begin

ALUc <=1;

wreg <=1;

m2reg <=0;

wmem <=0;

ALUimm <=0;

regrt =0;

end

6'b100110: //XOR

begin

ALUc <= 3;

wreg <=1;

m2reg <=0;

wmem <=0;

ALUimm <=0;

regrt =0;

end

endcase

end

6'b100011: //lw instruct

begin

ALUc <= 2;

regrt = 1;

wreg <= 0;

m2reg <= 1;

wmem <= 0;

ALUimm <= 1;

end

endcase

end

endmodule

module mux(

input [4:0] rd,

input [4:0] rt,

input regrt,

output reg [4:0] destreg

);

always @(\*) begin

if (regrt == 0)

destreg = rd;

else

destreg = rt;

end

endmodule

module registerFile(

input [4:0] rs,

input [4:0] rt,

input clk,

input [4:0] wdestreg,

input [31:0] wbData,

input wwreg,

output reg [31:0] qa,

output reg [31:0] qb

);

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

initial begin

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];

if(wwreg == 1) begin

registers[wdestreg] <= wbData;

end

end

endmodule

module mux4(

input [1:0] fwd,

input [31:0] a,

input [31:0] b,

input [31:0] c,

input [31:0] d,

output reg [31:0] e

);

always @(\*) begin

if (fwd == 0)

e = a;

else if(fwd == 1)

e = b;

else if(fwd == 2)

e = c;

else if(fwd ==3)

e = d;

end

endmodule

module immExtender (

input [15:0] imm,

output reg [31:0] imm32

);

always @(\*)

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

endmodule

module idexePipelinereg(

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 reg ewreg,

output reg em2reg,

output reg 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 = qa;

eqb= qb;

eimm32 = imm32;

end

endmodule

module aluMux(

input [31:0] eqb,

input [31:0] eimm32,

input ealuimm,

output reg [31:0] b

);

always @\* begin

if (ealuimm == 0)

b = eqb;

else

b = eimm32;

end

endmodule

module ALU(

input [31:0] eqa,

input [31:0] b,

input [3:0] eALUc,

output reg [31:0] r

);

always @\* begin

if(eALUc == 2) //add

r <= eqa + b;

else if(eALUc == 6) //sub

r <= eqa - b;

else if (eALUc == 0) //and

r <= eqa & b;

else if (eALUc == 1) //or

r <= eqa | b;

else if (eALUc == 3) //xor

r <= (~eqa & b) |(eqa & ~b);

end

endmodule

module exeMemPipelineReg(

input ewreg,

input em2reg,

input ewmem,

input [4:0] edestReg,

input [31:0] r,

input [31:0] eqb,

input clk,

output reg mwreg,

output reg mm2reg,

output reg 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

module dataMemory(

input [31:0] mr,

input [31:0] mqb,

input mwmem,

input clk,

output reg [31:0] mdo

);

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

initial begin

memoryArray[0] = {32'hA00000AA};

memoryArray[4] = {32'h10000011};

memoryArray[8] = {32'h20000022};

memoryArray[12] = {32'h30000033};

memoryArray[16] = {32'h40000044};

memoryArray[20] = {32'h50000055};

memoryArray[24] = {32'h60000066};

memoryArray[28] = {32'h70000077};

memoryArray[32] = {32'h80000088};

memoryArray[36] = {32'h90000099};

end

always @\*

mdo <= memoryArray[mr];

always @(negedge clk) begin

if (mwmem == 1) memoryArray[mr] = mqb;

end

endmodule

module memWBPipelineReg(

input mwreg,

input mm2reg,

input [4:0] mdestReg,

input [31:0] mr,

input [31:0] mdo,

input clk,

output reg wwreg,

output reg 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

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

wbData = wdo;

end

endmodule

module Datapath(input clk,

output wire [31:0] pc,

output wire [31:0] dinstOut,

output wire [4:0] mdestReg,

output wire [31:0] mqb,

output wire wwreg,

output wire wm2reg,

output wire [4:0] wdestReg,

output wire [31:0] wr,

output wire [31:0] wdo,

output wire [31:0] qa,

output wire [31:0] qb

);

wire [31:0] newPC;

wire [31:0] instOut;

wire mwreg;

wire mm2reg;

wire mwmem;

wire [31:0] mdo;

wire [31:0] mr;

wire regrt;

wire wreg;

wire m2reg;

wire wmem;

wire [3:0] ALUc;

wire ALUimm;

wire [4:0] destreg;

wire [31:0] immExt32;

wire [31:0] r;

wire [31:0] ALU\_muxOut;

wire [31:0] wbData;

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 wpcir;

PC PCreg(pc, clk, newPC);

pcAdder PCAdder(newPC, pc);

insMem MemoryReg(newPC, instOut);

ifidPipelineReg if\_id(instOut, clk, dinstOut);

wire [1:0] fwda;

wire [1:0] fwdb;

controlUnit CTRLunit(dinstOut[31:26], dinstOut[5:0],dinstOut[25:21], dinstOut[20:16], mdestReg, mm2reg, mwreg, edestReg, em2reg, ewreg, fwda, fwdb, wreg,m2reg, wmem, ALUc, ALUimm, regrt);

mux regMux(dinstOut[15:11],dinstOut[20:16], regrt, destreg);

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

wire [31:0] muxOutA;

wire [31:0] muxOutB;

mux4 mux\_A(fwda,qa,r, mr,mdo, muxOutA);

mux4 mux\_B(fwdb,qb,r, mr,mdo, muxOutB);

immExtender Extend32( dinstOut[15:0], immExt32);

idexePipelinereg id\_exe(wreg, m2reg, wmem, ALUc, ALUimm, destreg, muxOutA, muxOutB, immExt32, clk, ewreg, em2reg, ewmem, eALUc, eALUimm, edestReg, eqa, eqb, eimm32);

aluMux ALU\_mux(eqb, eimm32,eALUimm, ALU\_muxOut);

ALU alu(eqa, ALU\_muxOut, eALUc, r);

exeMemPipelineReg exe\_mem(ewreg, em2reg, ewmem, edestReg, r, eqb, clk, mwreg, mm2reg, mwmem, mdestReg, mr, mqb);

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

memWBPipelineReg memWB(mwreg, mm2reg,mdestReg, mr, mdo, clk, wwreg, wm2reg,wdestReg, wr, wdo);

WBMux writeBack(wr, wdo, wm2reg, wbData);

endmodule

TestBench Code: (Please use the provided testbench configuration for testing in the folder)

reg clk;

wire [31:0] pc;

wire [31:0] dinstOut;

wire [4:0] mdestReg;

wire [31:0] mqb;

wire wwreg;

wire wm2reg;

wire [4:0] wdestReg;

wire [31:0] wr;

wire [31:0] wdo;

wire [31:0] qa;

wire [31:0] qb;

Datapath datapath1(clk, pc, dinstOut, mdestReg, mqb, wwreg, wm2reg, wdestReg, wr, wdo, qa,qb ); initial begin

clk = 0;

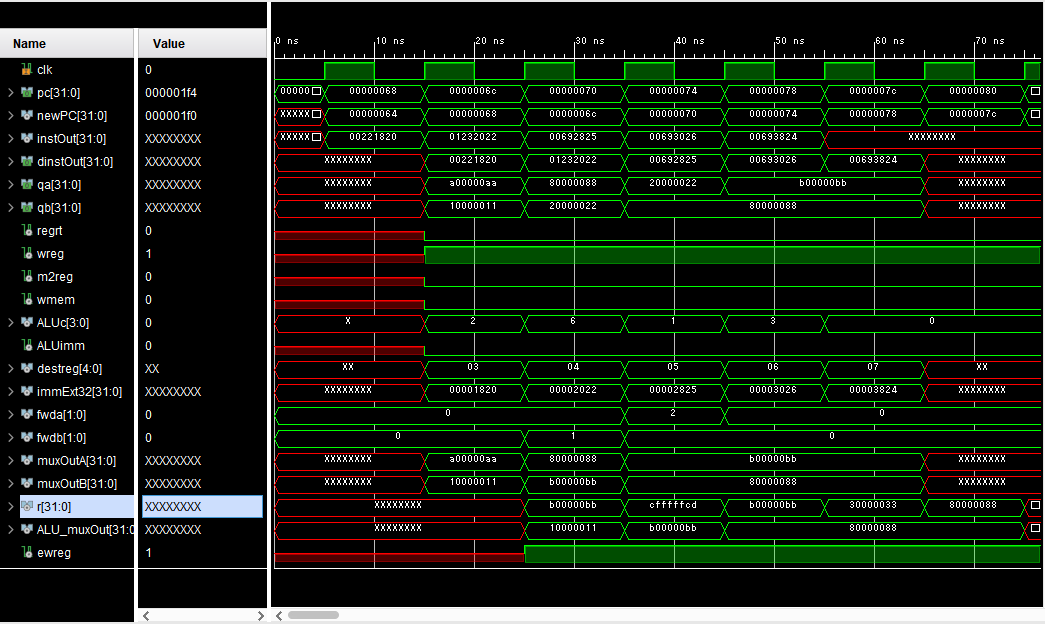
end

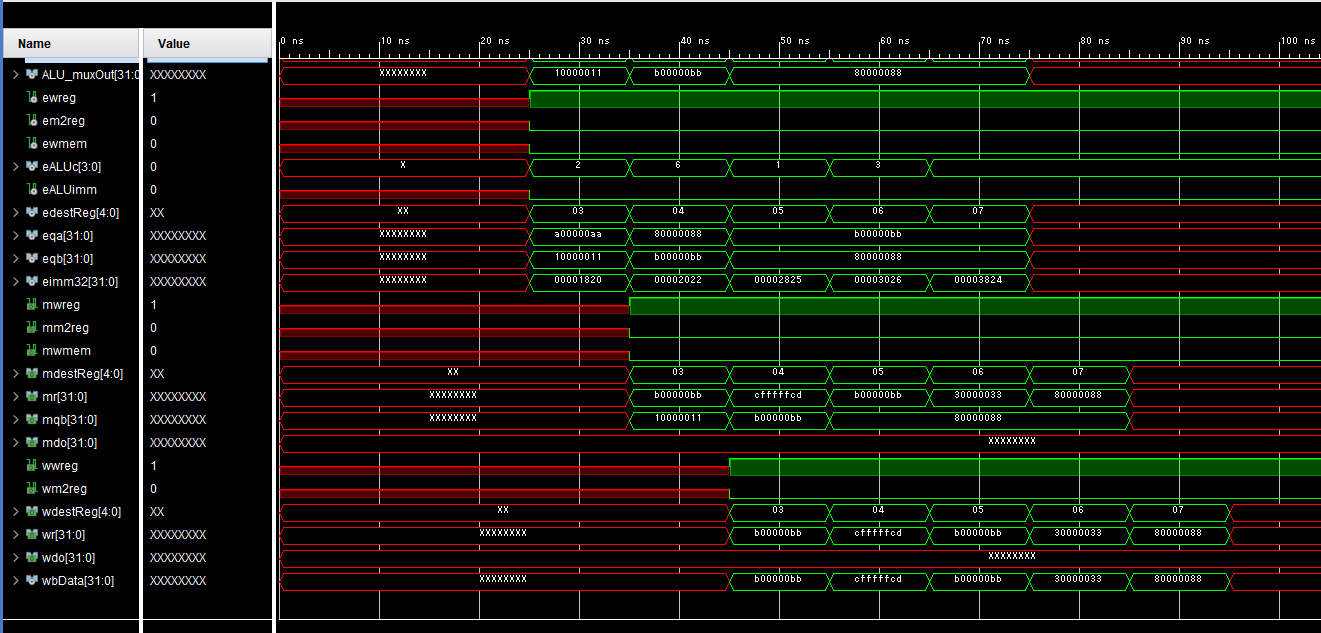
always

#5 clk = ~clk;

endmodule

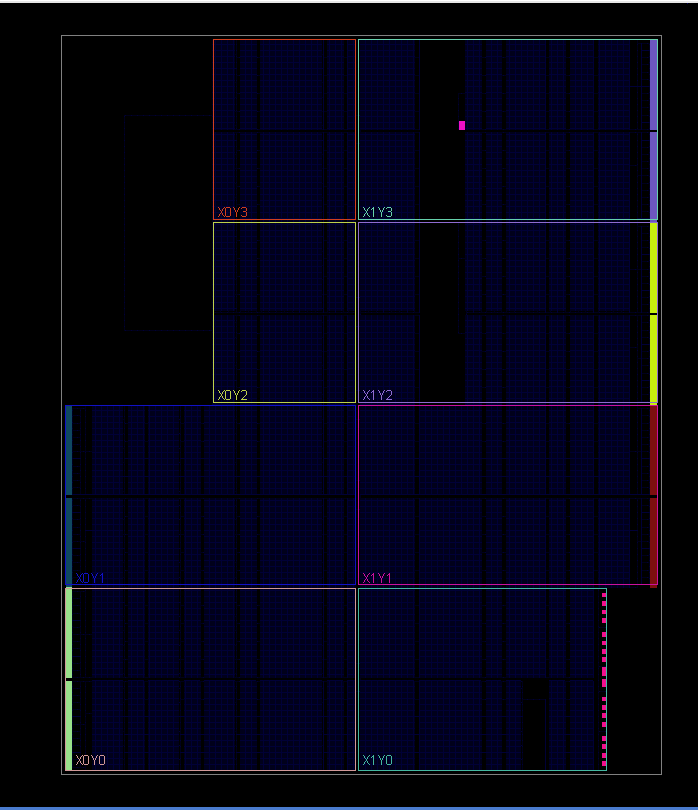
Simulation Screenshot



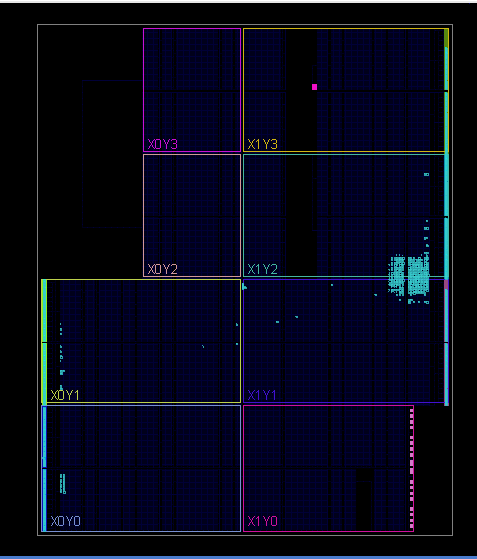


Floor Planning:

Synthesis:

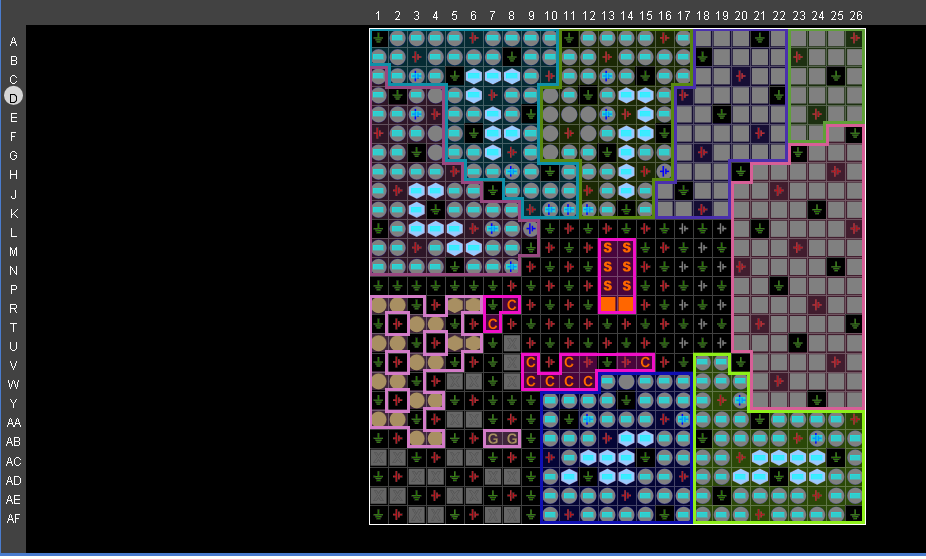


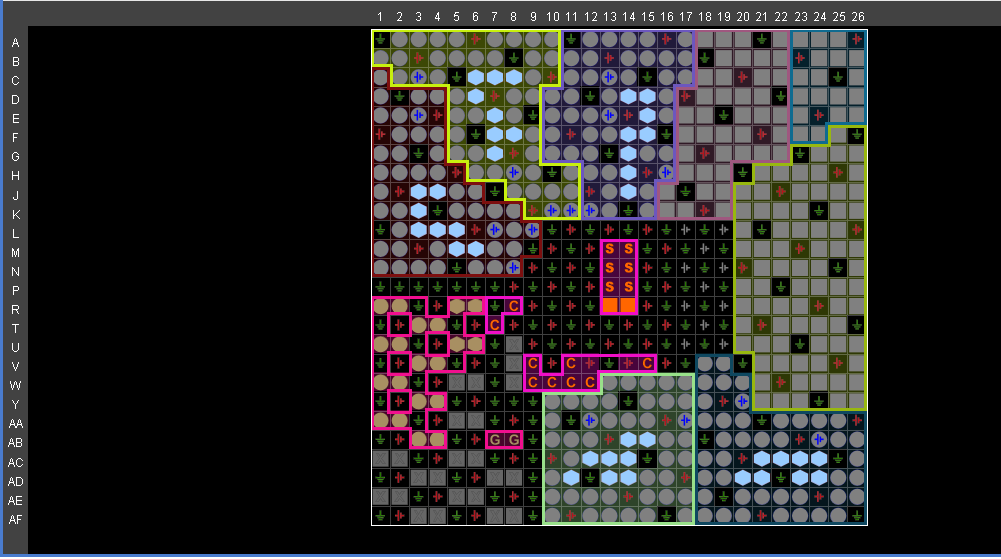
Implementation:



I/O Planning

Implementation:



Synthesis:

Schematic

