VLSI - FINAL PROJECT REPORT

MANAN CHICHRA - 2022102058

1) Verilog Component

I) ADDER- SUBTRACTOR SUB CIRCUIT

adder_sub.v

```
module Adder_Sub(a,b,cin,M,sum,cout);
input [3:0]a,b;
input cin,M;
output wire [3:0]sum;
output cout;
FullAdd_SUB F1(a[0],b[0],cin,M,sum[0],cout1);
FullAdd_SUB F2(a[1],b[1],cout1,M,sum[1],cout2);
FullAdd_SUB F3(a[2],b[2],cout2,M,sum[2],cout3);
FullAdd_SUB F4(a[3],b[3],cout3,M,sum[3],cout);
endmodule

module FullAdd_SUB(a,b,cin,M,sum,cout);
input a,b,cin,M;
output wire sum,cout;
wire s_M;
wire s1,c1,c2,c3;
xor(b_M,M,b);
xor(s1,a,b_M);
xor(sum,s1,cin);
and(c1,a,b_M);
and(c2,s1,cin);
//and(c3,a,cin);
or(cout,c1,c2);
endmodule
```

In this Adder_Sub module, I have used another module which adds 3 bits – a,b and cin.
I have used this submodule four times inside my main module.

The subtraction of unsigned binary numbers can be done most conveniently by

means of complements

• Remember that the subtraction A - B can be done by taking the 2's complement of

B and adding it to A

• The 2's complement can be obtained by taking the 1's complement and adding 1

to the least significant pair of bits

• The 1's complement can be implemented with inverters, and a 1 can be added to

the sum through the input carry

• The circuit for subtracting A - B consists of an adder with inverters placed between

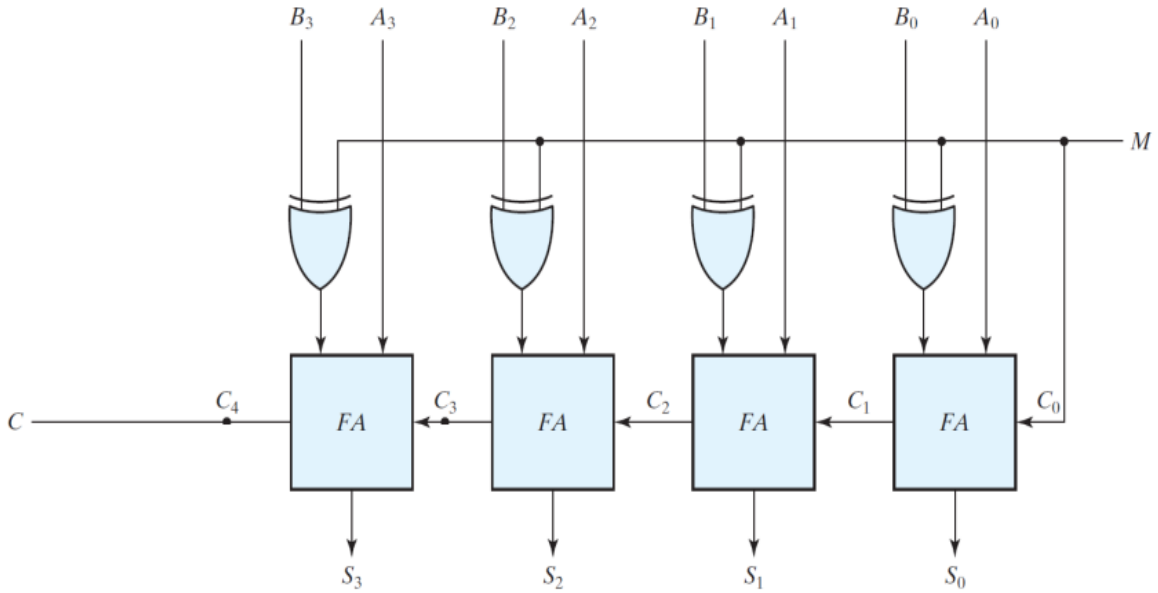
each data input B and the corresponding input of the full adder

• The input carry C0 must be equal to 1 when subtraction is performed

• The operation thus performed becomes A, plus the 1's complement of B, plus 1.

This is equal to A plus the 2's complement of B

Circuit Reference



M = 1 for subtraction

M = 0 for additional

• The mode input M controls the operation

• When M = 0, the circuit is an adder, and when M = 1, the

circuit becomes a subtractor

• When M = 0, the full adders receive the value of B, the

input carry is 0, and the circuit performs A + B

• When M = 1, the full adders receive B' and C0 = 1

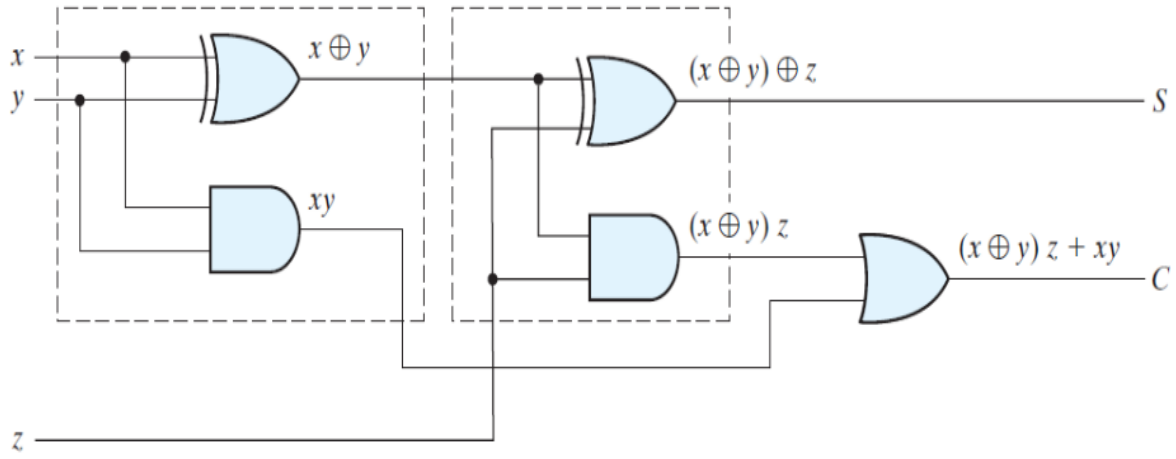
• Thus, the B inputs are all complemented and a 1 is added

through the input carry

• The circuit performs the operation A plus the 2's

complement of B

FA block:



II) COMPARATOR SUB-CIRCUIT

comparator.v

```
module bit_comparator (
  input [3:0] A,
  input [3:0] B,
  output equal,
  output A_greater,
  output B_greater
);
wire w0,w1,w2,w3,s0,s1,s2,s3,s4,s5,s6,s7;
xnor(w0,A[0],B[0]);
xnor(w1,A[1],B[1]);
xnor(w2,A[2],B[2]);
xnor(w3,A[3],B[3]);

wire [3:0] A_;
wire [3:0] B_;
not ((A_[0]),(A[0]));
not ((A_[1]),(A[1]));
not ((A_[2]),(A[2]));
not ((A_[3]),(A[3]));

not ((B_[0]),(B[0]));
not ((B_[1]),(B[1]));
not ((B_[2]),(B[2]));
not ((B_[3]),(B[3]));



and(equal,w0,w1,w2,w3);

and (s0,A[3],B_[3]);
and (s1,w3,A[2],B_[2]);
and (s2,w2,w3,A[1],B_[1]);
and (s3,w1,w2,w3,A[0],B_[0]);
or (A_greater,s0,s1,s2,s3);

and (s4,A_[3],B[3]);
and (s5,w3,A_[2],B[2]);
and (s6,w2,w3,A_[1],B[1]);
and (s7,w1,w2,w3,A_[0],B[0]);

or (B_greater,s4,s5,s6,s7);

endmodule

```

- The comparison of two numbers is an operation that determines whether

one number is greater than, less than, or equal to the other number

- A magnitude comparator is a combinational circuit that compares two

numbers A and B and determines their relative magnitudes

• The outcome of the comparison is specified by three binary variables that indicate whether A > B, A = B, or A < B

• On the one hand, the circuit for comparing two n -bit numbers have 22n entries in the truth table and becomes too cumbersome, even with n = 3

• On the other hand, as one may suspect, a comparator circuit possesses a certain amount of regularity.

• Consider two numbers, A and B, with four digits each A3A2A1A0 and B3B2B1B0

• The two numbers are equal if all pairs of significant digits are equal: A3 = B3, A2 = B2, A1 = B1, and A0 = B0

• To check bit-wise equality, we can use the XNOR gate

$xi = Ai\ Bi + A'i\ Bi'$ for i = 0, 1, 2, 3

• For equality to exist, all xi variables must be equal to 1, a condition that dictates an AND operation of all variables:

$(A = B) = x3\ x2\ x1\ x0$

• To determine whether A is greater or less than B, we inspect the relative magnitudes of pairs of significant digits, starting from the most significant position

• If the two digits of a pair are equal, we compare the next lower significant pair of digits

• The comparison continues until a pair of unequal digits is reached

• If the corresponding digit of A is 1 and that of B is 0, we conclude that A > B. If the corresponding digit of A is 0 and that of B is 1, we have A < B

• The comparison can be expressed logically by the two Boolean functions:

$A > B = A3\ B3' + x3\ A2\ B2' + x3\ x2\ A1\ B1' + x3\ x2\ x1\ A0\ B0'$

$(A < B) = A'3\ B3 + x3\ A'2\ B2 + x3\ x2\ A1'\ B1 + x3\ x2\ x1\ A'0\ B0$

III) AND BLOCK

and.v

```
1   module And (
2     input [3:0] A,
3     input [3:0] B,
4     output [3:0] out
5   );
6
7   and (out[0],A[0],B[0]);
8   and (out[1],A[1],B[1]);
9   and (out[2],A[2],B[2]);
10  and (out[3],A[3],B[3]);
11
12
13  endmodule
```

IV) ALU

I have also created enabled subcircuits, which send inputs to their corresponding operation block, if the enable is ‘High’.

In the ALU testbench, I have created instances of the enable modules, which are responsible for giving the input to the operation modules.

tb_alu.v

```
1   `include "enable_add.v"
2   `include "enable_and.v"
3   `include "enable_comp.v"
4   module tb_alu;
5
6   reg [3:0]A;
7   reg [3:0]B;
8   reg cin, M,e;
9   wire [3:0] sum;
10  wire equal, A_greater, B_greater;
11  wire [3:0] out;
12  wire cout;
13  integer index;
14
15  enable_add F(.A(A),.B(B),.M(M),.cin(cin),.cout(cout),.sum(sum),.e(e));
16  enable_comp dut1(
17      .A(A),
18      .B(B),
19      .equal(equal),
20      .A_greater(A_greater),
21      .B_greater(B_greater),
22      .e(e)
23    );
24
25  enable_and dut(
26      .A(A),
27      .B(B),
28      .e(e),
29      .out(out)
30    );
31
32  initial begin
```

2) NgSpice Component

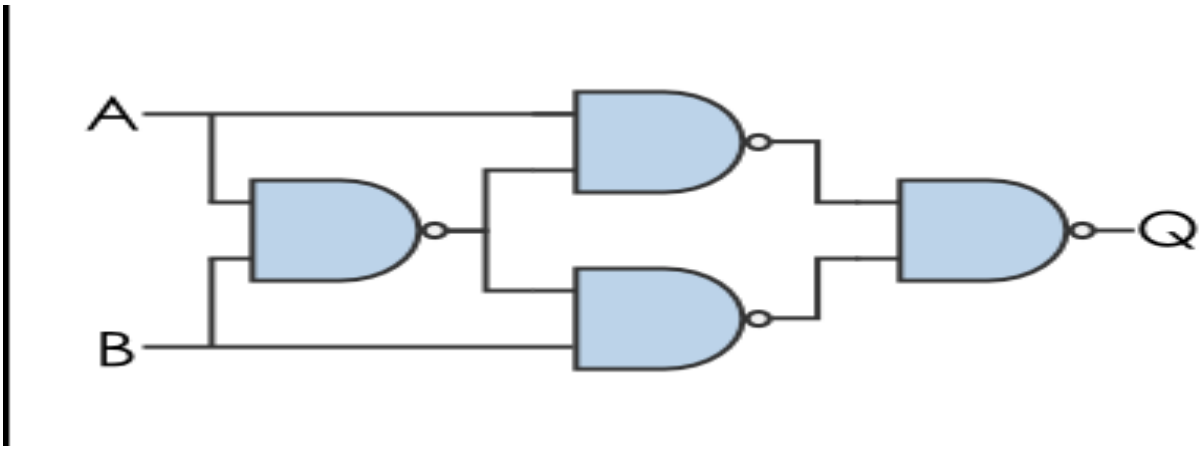
NAND subcircuit used:

NAND.sub

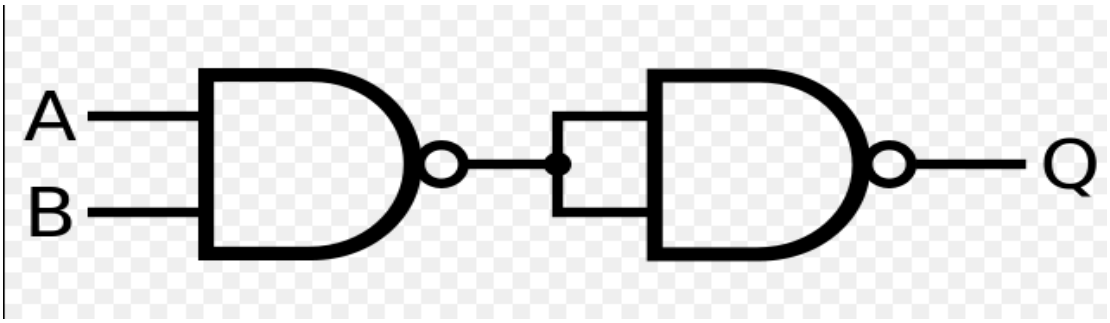
```
.subckt NAND node_out node_a node_b vdd gnd

    Mn1 node_out node_a node_y gnd CMOSN W = {wn1} L = {ln1}
    + AS = {5*wn1*LAMBDA} PS = {10*LAMBDA + 2*wn1} AD = {5*wn1*LAMBDA} PD = {10*LAMBDA + 2*wn1}

    Mn2 node_y node_b gnd gnd CMOSN W = {wn2} L = {ln2}
    + AS = {5*wn2*LAMBDA} PS = {10*LAMBDA + 2*wn2} AD = {5*wn2*LAMBDA} PD = {10*LAMBDA + 2*wn2}


    Mp1 node_out node_a vdd vdd CMOSP W = {wp1} L = {lp1}
    + AS = {5*wp1*LAMBDA} PS = {10*LAMBDA + 2*wp1} AD = {5*wp1*LAMBDA} PD = {10*LAMBDA + 2*wp1}

    Mp2 node_out node_b vdd vdd CMOSP W = {wp2} L = {lp2}
    + AS = {5*wp2*LAMBDA} PS = {10*LAMBDA + 2*wp2} AD = {5*wp2*LAMBDA} PD = {10*LAMBDA + 2*wp2}

.ends NAND
```

I have used this subcircuit to make, AND XOR, XNOR & OR gates.

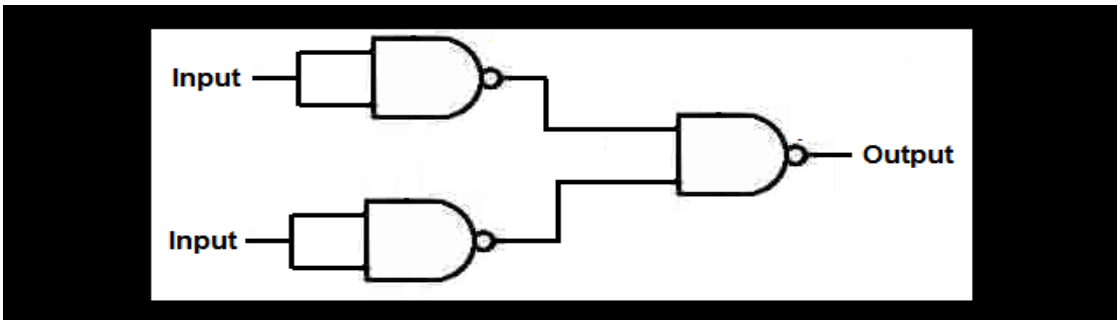
XOR uses 4 NAND gates.



AND using two NAND gates



OR using two NAND gates



XNOR using XOR & NAND

Working

The ALU circuit uses full_adder_sub, and_clock , comparator and enable sub-circuit.

Each of the sub-circuit has been implemented according to the logic discussed previously under Verilog.

ALU ckt NgSpice file

ALU.ckt

```
.include TSMC_180nm.txt
.include NAND.sub
.include OR.sub
.include AND.sub
.include XOR.sub
.include XNOR.sub
.include half_adder.sub
.include full_adder_sub.sub
.include and_block.sub

.include enable_and_block.sub
.include comp.sub
.include enable_full_adder_sub.sub
.include enable_comp.sub

.param SUPPLY = 1.8
.param LAMBDA = 0.18u

.param wn1 = {10*LAMBDA}
.param wn2 = {10*LAMBDA}
.param ln1 = {2*LAMBDA}
.param ln2 = {2*LAMBDA}

.param wp1 = wn1
.param wp2 = wn1
.param lp1 = {LAMBDA}
.param lp2 = {LAMBDA}

.global gnd

Vdd node_x gnd 'SUPPLY'

V_in_a0 a0 gnd PULSE(0 0 0ns 100ps 100ps 20ns 40ns)
V_in_a1 a1 gnd PULSE(0 0 0ns 100ps 100ps 50ns 70ns)
V_in_a2 a2 gnd PULSE(1.8 1.8 0ns 100ps 100ps 30ns 40ns)
V_in_a3 a3 gnd PULSE(1.8 1.8 0ns 100ps 100ps 50ns 100ns)

V_in_b0 b0 gnd PULSE(1.8 1.8 0ns 100ps 100ps 20ns 40ns)
V_in_b1 b1 gnd PULSE(1.8 1.8 0ns 100ps 100ps 50ns 70ns)
V_in_b2 b2 gnd PULSE(1.8 1.8 0ns 100ps 100ps 30ns 40ns)
V_in_b3 b3 gnd PULSE(0 0 0ns 100ps 100ps 25ns 50ns)
V_cin cin gnd PULSE(0 0 0ns 100ps 100ps 50ns 70ns)

V_M1 M1 gnd PULSE(0 0 0ns 100ps 100ps 50ns 70ns)
V_M2 M2 gnd PULSE(1.8 1.8 0ns 100ps 100ps 50ns 70ns)
V_e1 e1 gnd PULSE(1.8 1.8 0ns 100ps 100ps 50ns 70ns)
V_e2 e2 gnd PULSE(1.8 1.8 0ns 100ps 100ps 50ns 70ns)
V_e3 e3 gnd PULSE(1.8 1.8 0ns 100ps 100ps 50ns 70ns)
V_e4 e4 gnd PULSE(1.8 1.8 0ns 100ps 100ps 50ns 70ns)

X1 A B A_B a0 a1 a2 a3 b0 b1 b2 b3 e1 node_x gnd enable_comp
X2 s0 s1 s2 s3 cout M1 a0 a1 a2 a3 b0 b1 b2 b3 cin e2 node_x gnd enable_full_adder_s
X3 s0_ s1_ s2_ s3_ cout_ M2 a0 a1 a2 a3 b0 b1 b2 b3 cin e3 node_x gnd enable_full_ad
X4 c0 c1 c2 c3 a0 a1 a2 a3 b0 b1 b2 b3 e4 node_x gnd enable_and_block

.tran 1n 800n

.control
run


plot V(a0) v(a1)+2 v(a2) +4 v(a3)+6
plot v(b0) v(b1)+2 v(b2) +4 v(b3)+6
plot v(s0) v(s1)+2 v(s2) +4 v(s3)+6 v(cout) +8
plot v(s0_) v(s1_)+2 v(s2_) +4 v(s3_)+6 v(cout_) +8
plot v(c0) v(c1)+2 v(c2) +4 v(c3)+6
plot v(A) v(B)+2 v(A_B) +4


.end
.endc
```

3) Magic Component

All the layouts are built using SCN6M_DEEP.09 Cmos technology.

Basic Structure:

- PMOS transistor is formed by placing p-diffusion in an n-well.
- NMOS transistor is formed by placing n-diffusion in a p-well.

Material Usage:

- The gate of both PMOS and NMOS transistors is made using Polysilicon material.
- Polysilicon is connected to metal using polycontact.

Terminology and Connectivity:

- Terminals for the drain and source of PMOS and NMOS are established by connecting metal to the p-diffusion and n-diffusion, respectively.

pdicontact and ndcontact are used to connect metal to p-diffusion and n-diffusion, respectively.

Metal Connectivity:

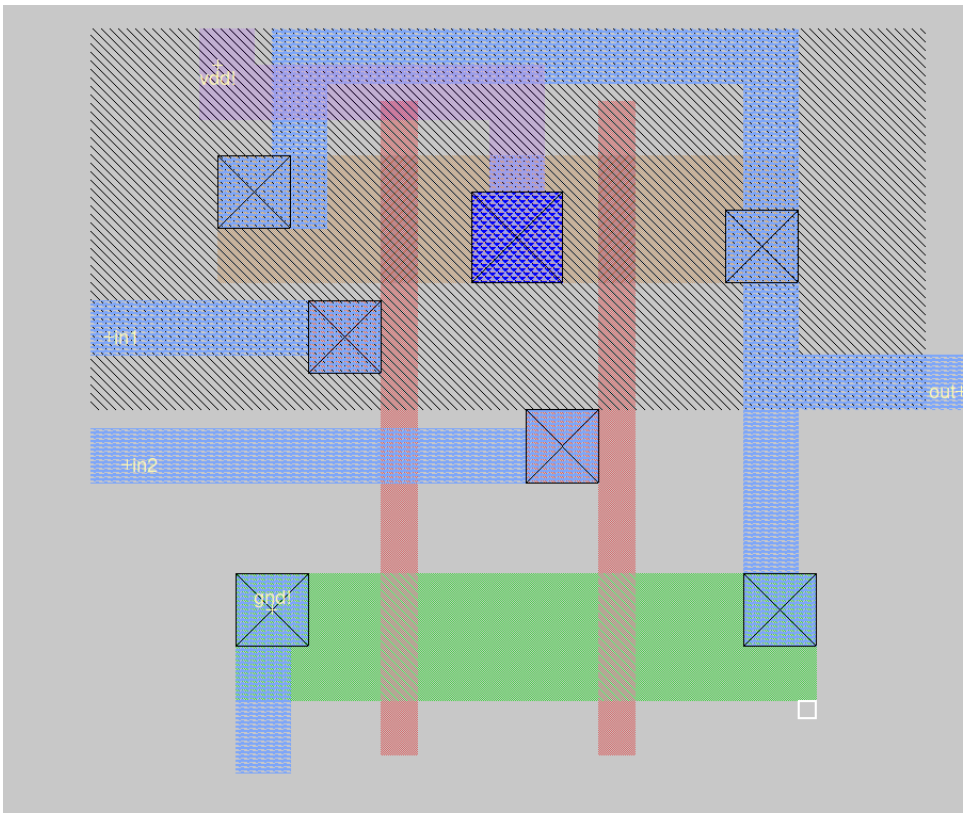
- Two different metals can overlap, but if they need to be electrically connected, a contact must be placed between them.
- Example: M3contact is used when Metal 3 and Metal 2 need to be connected in the overlapping area.

Specific Constraint:

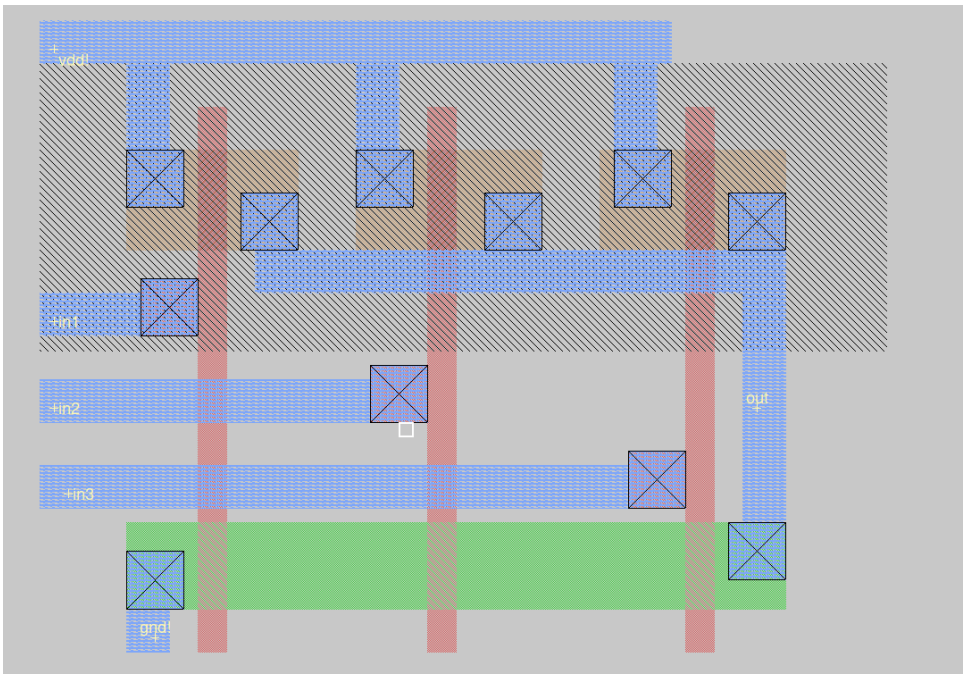
- Only Metal 1 can be connected directly to polysilicon.

I have made the following gates, which have been used in various sub-circuit layouts.

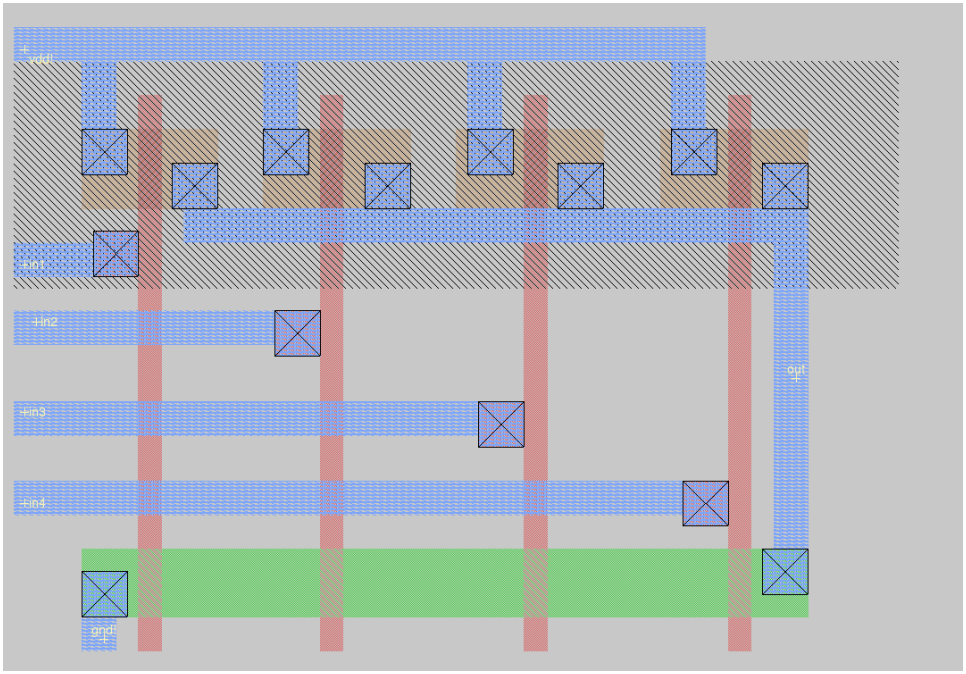
1) 2 INPUT – NAND



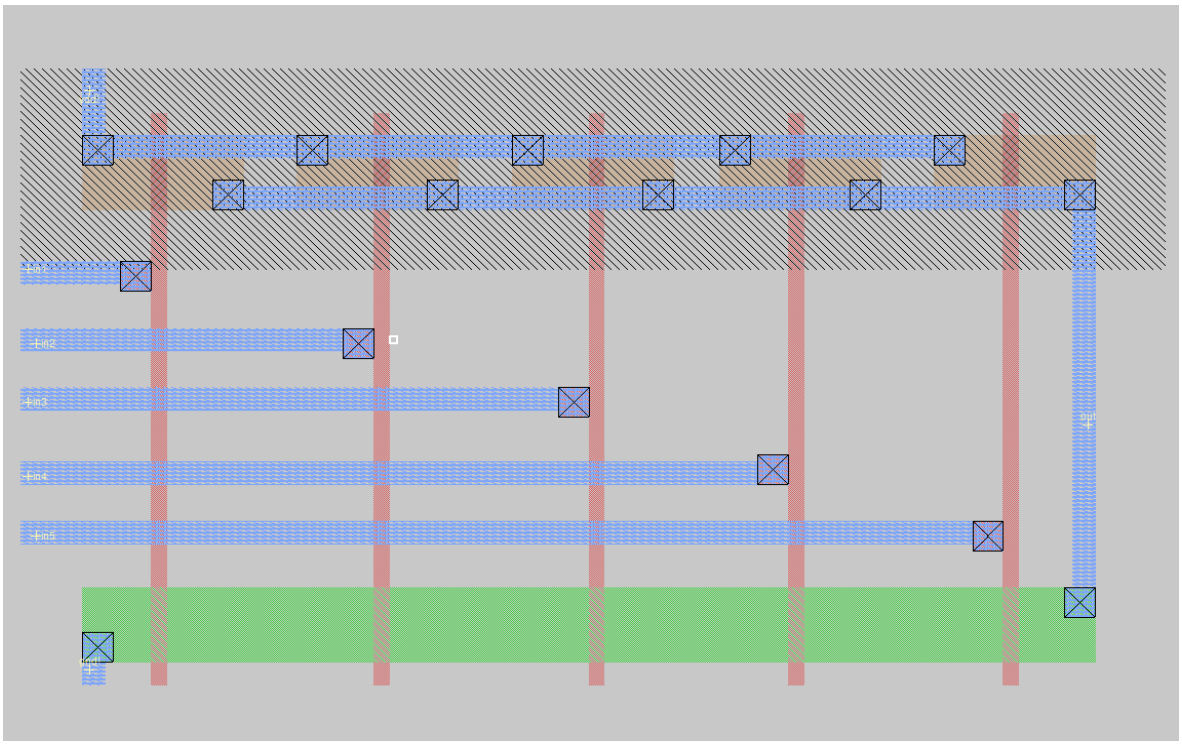
2) 3 INPUT – NAND



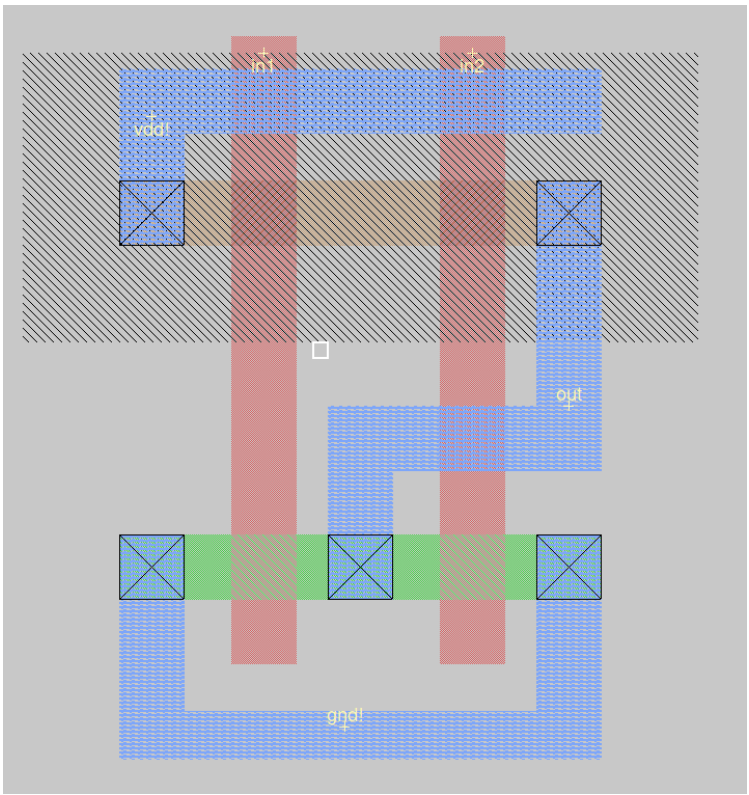
3) 4 INPUT NAND



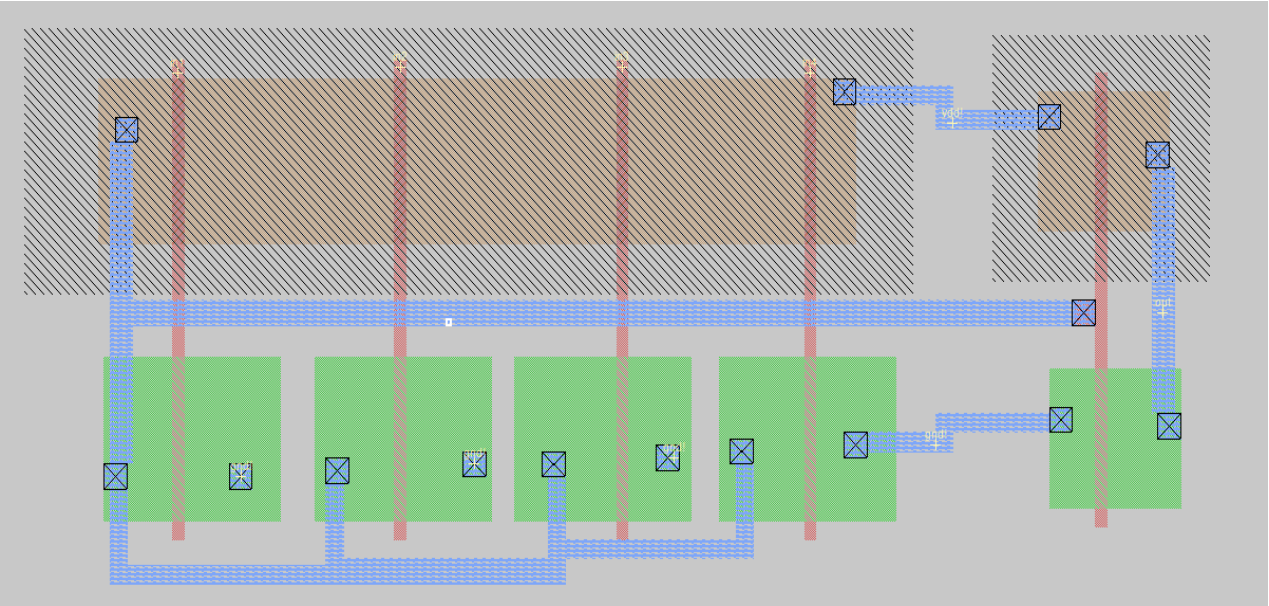
4) 5 INPUT NAND



5) 2 INPUT NOR

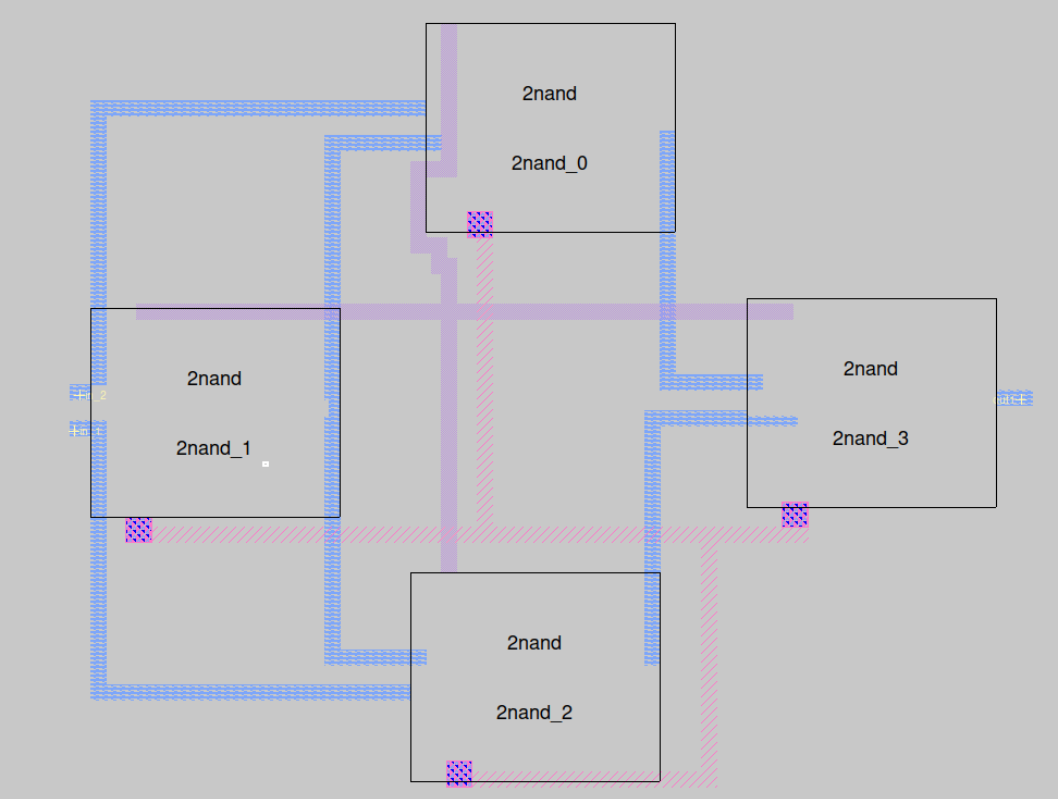


6) 4 INPUT OR

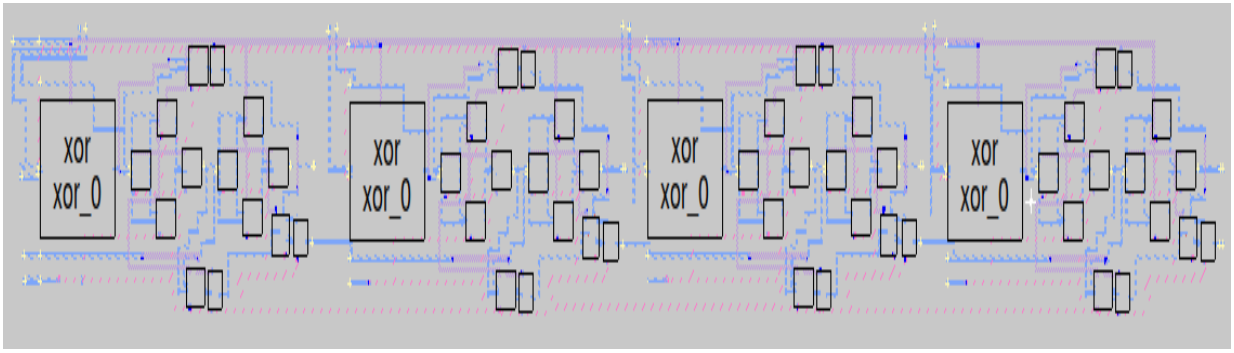


7) 2 INPUT XOR

**Implemented using 4 two input NAND gates

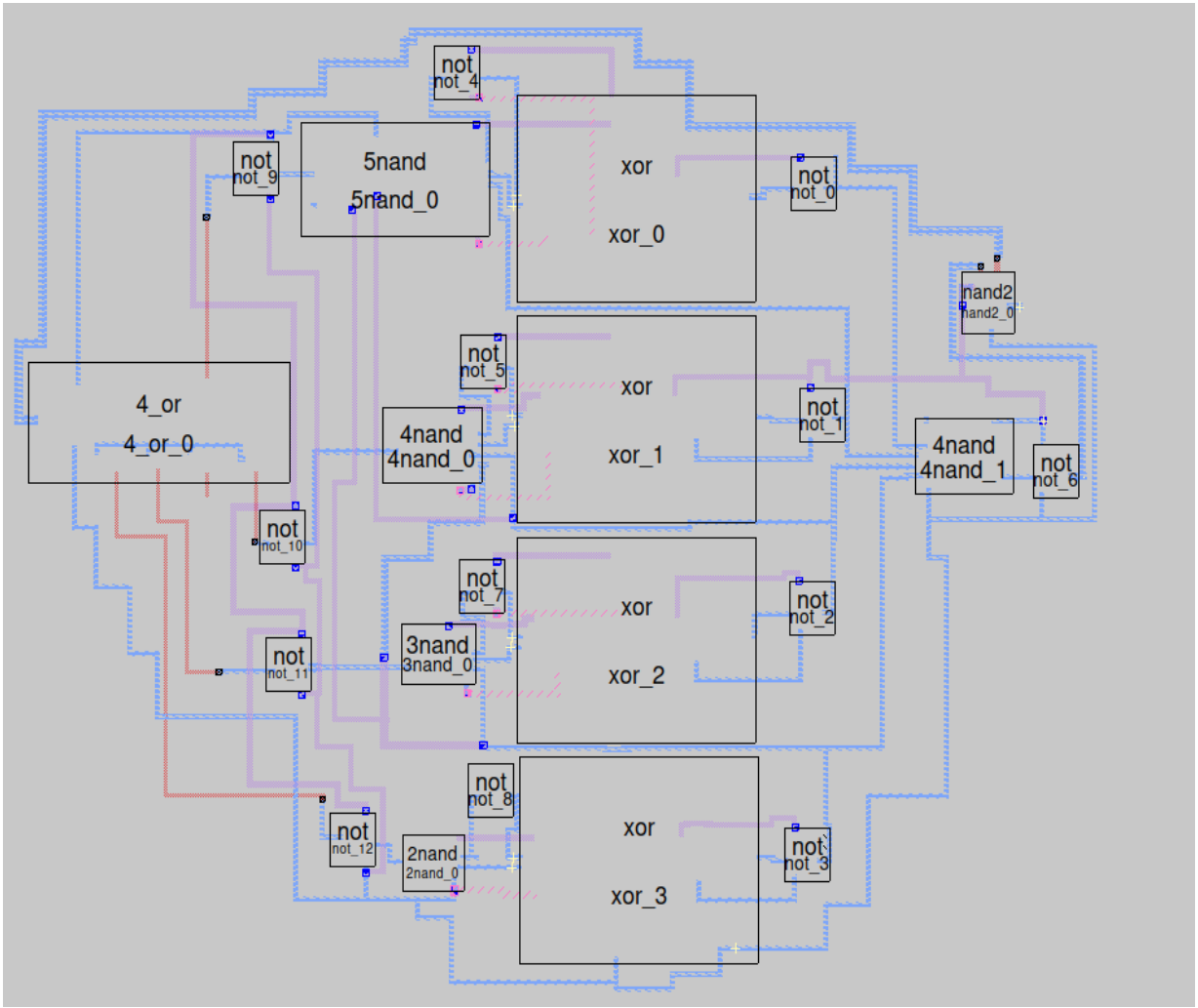


8) FULL ADDER (series of 4 (3-bit FA's)



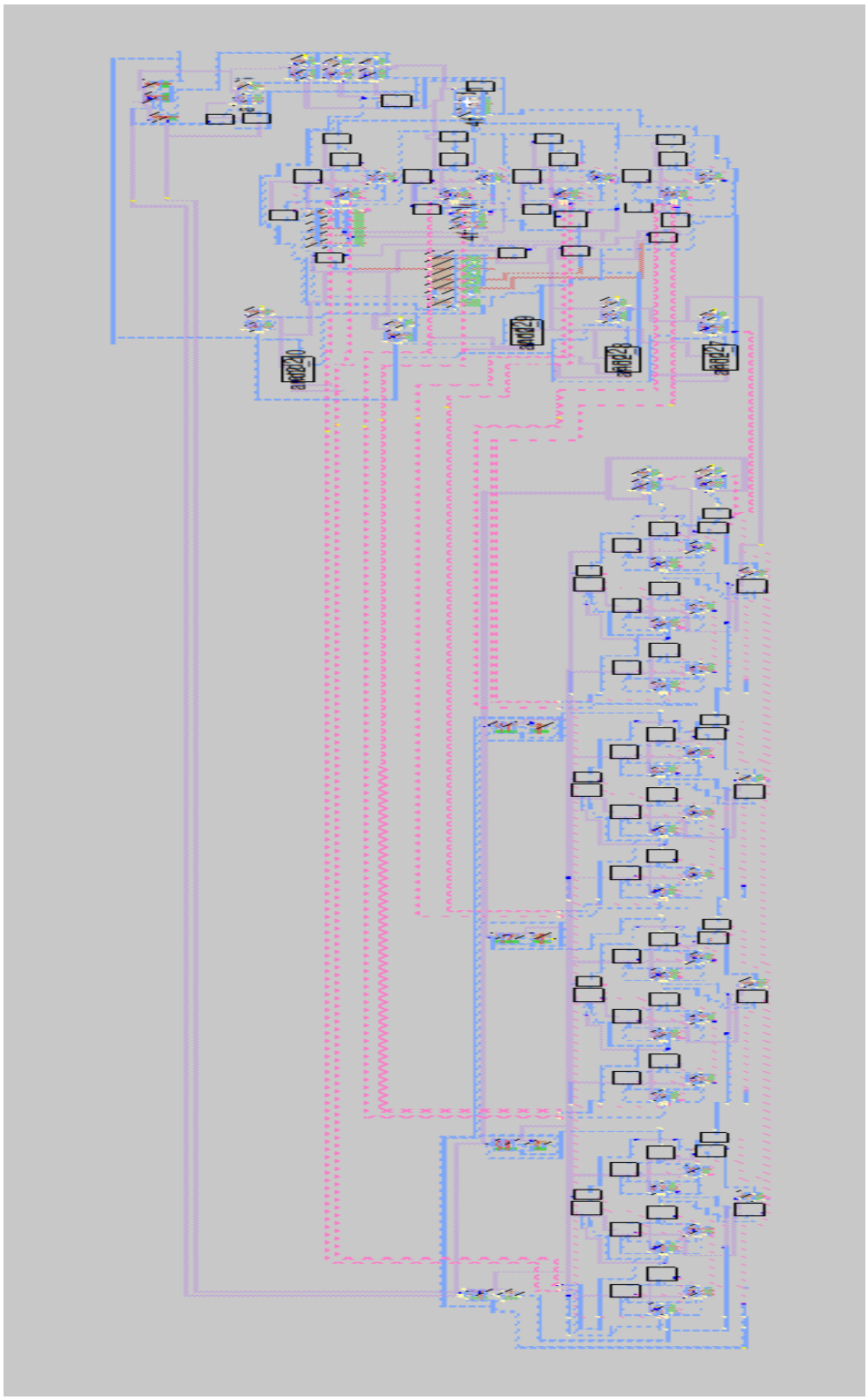
9) COMPARATOR

**Implemented using 5,4,3,2 input NAND gates. Also used XOR and 4 input OR gates.



10) ALU

This includes the 2:4 decoder, AND block, Comparator and Adder/Subtractor (rotated image)



*DELAY ANANLYSIS IS PRESENTED IN A SEPARATE REPORT.

(DELAY ANALYSIS HAS BEEN DONE USING THE EXTRACTED SPICE FILES)