# Final Project Report: Convolutional Computation Unit Chip Design

ECE6024

Group 8

Yihui Wang

Prof. M. E. Zaghloul

# Contents

# Introduction

Convolution operation is being widely used in engineering, such as signal processing, image processing and audio process, and depend on the kernel that the convolution uses, it can extract, sharpen, or smoothen the signal, for example, use a kernel of $[5\ 5\ 5]$ to a noised sin wave signal, the signal could be smoothened, in other word the signal is being denoised:

$$y[n] = f[n] * g[n] = \sum_{m=0}^{M-1} f[n-m]g[m]$$

*Figure 1 Convolution Function*

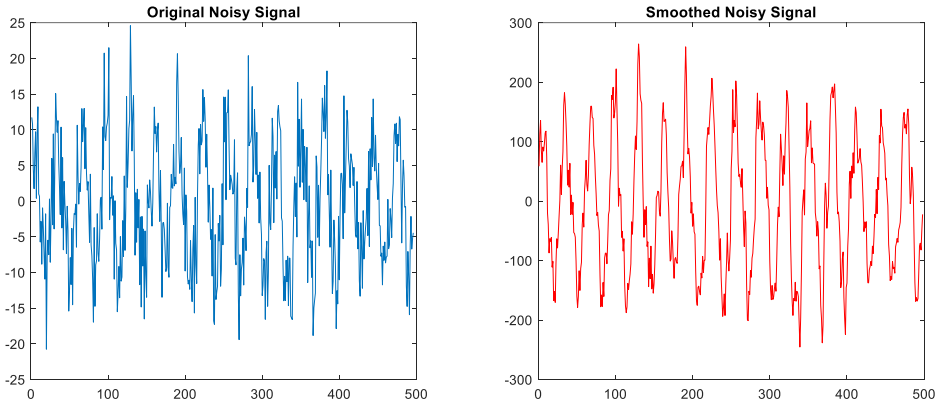


*Figure 2 Kernel of [5 5 5]*

For my project, I designed a chip that could do 1 dimensional convolution operation with 4-bit unsigned integer input data and kernel of 3 element. The schematic of chip as below:

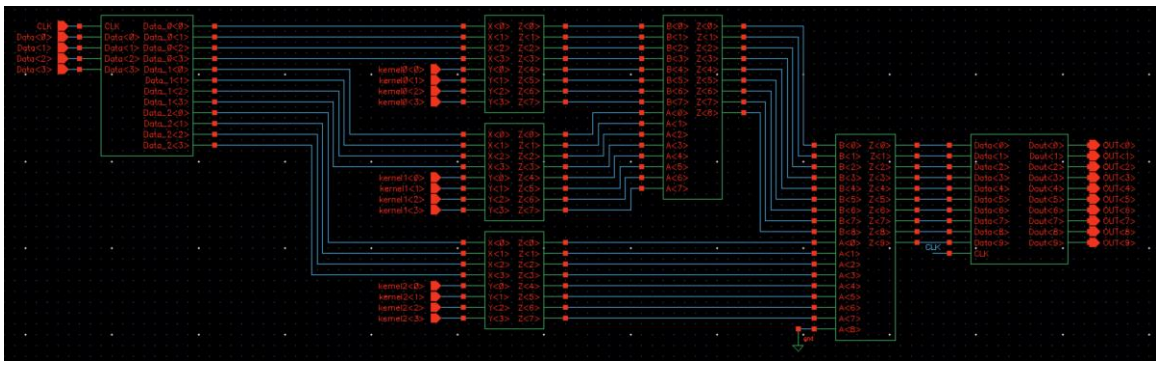

*Figure 3 Schematic of Chip*

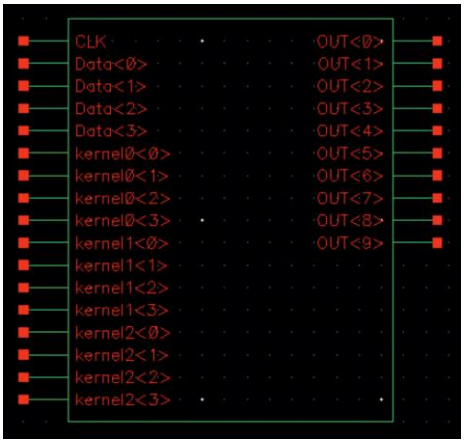


*Figure 4 Symbol of Convolutional Unit*

The computation logic and data flow are as follows:

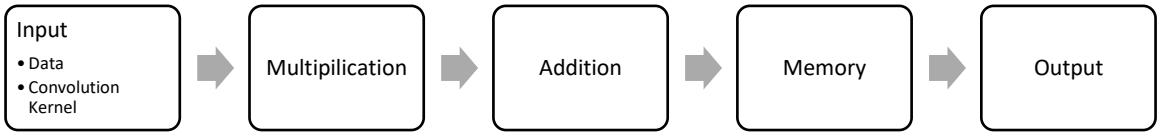


*Figure 5 Data Flow of One Cycle*

The element of schematic design is as follows:

| Cell | Number |
|---|---|
| 4-bit Serial-in Parallel-out Flip-flop | 1 |
| 4-bit multiplier | 3 |
| 8-bit adder | 1 |
| 9-bit adder | 1 |
| 10-bit Flip-flop | 1 |

*Figure 6 Composition of Design*

The final design uses 2,382 transistors.

# Basic Gate Design

## Inverter

Inverter is one of the basic logic gates that means ‘not’ logic in the digital circuit, following is the schematic, symbol, layout, and simulation result of inverter:

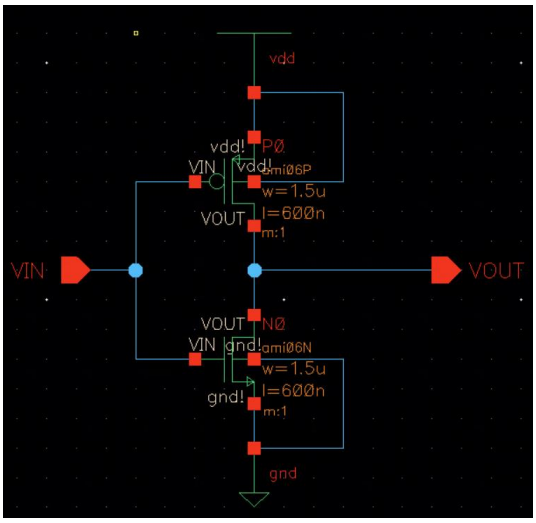


*Figure 7 Schematic of Inverter*

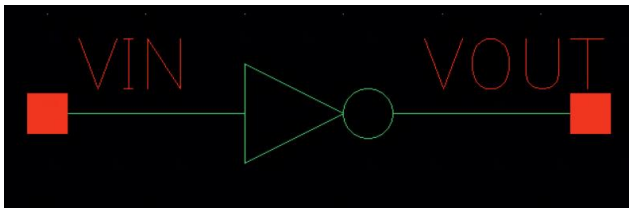


*Figure 8 Symbol of Inverter*

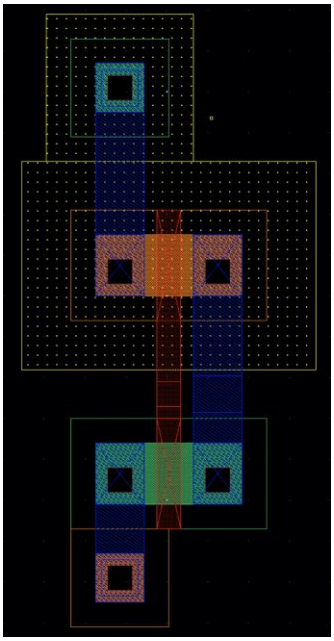

*Figure 9 Layout of Inverter*


*Figure 10 Extracted View of Inverter*

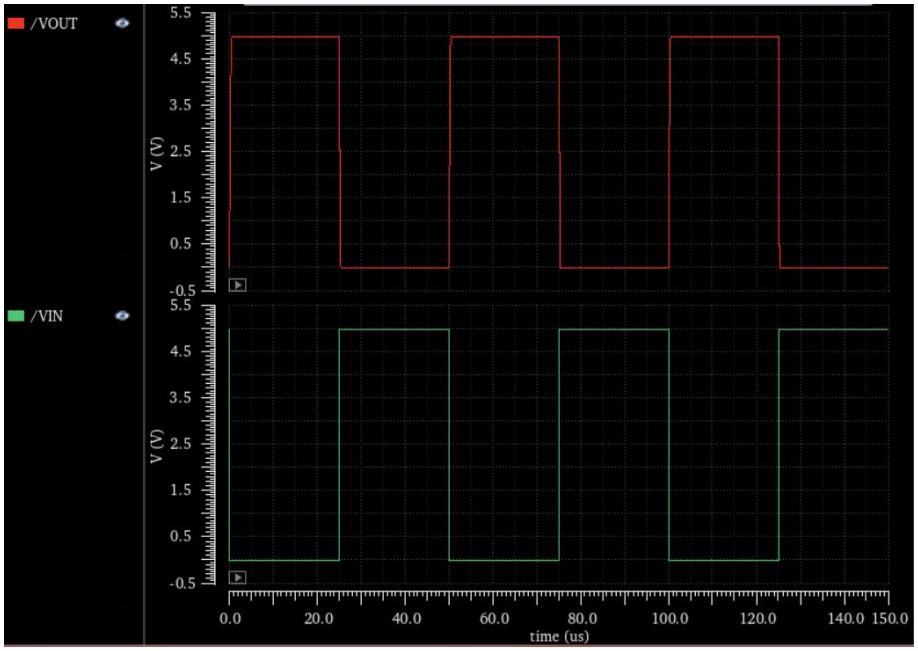


*Figure 11 Simulation Result of Inverter*

## NAND Gate

Following is the schematic, symbol, layout, and simulation result of NAND gate:

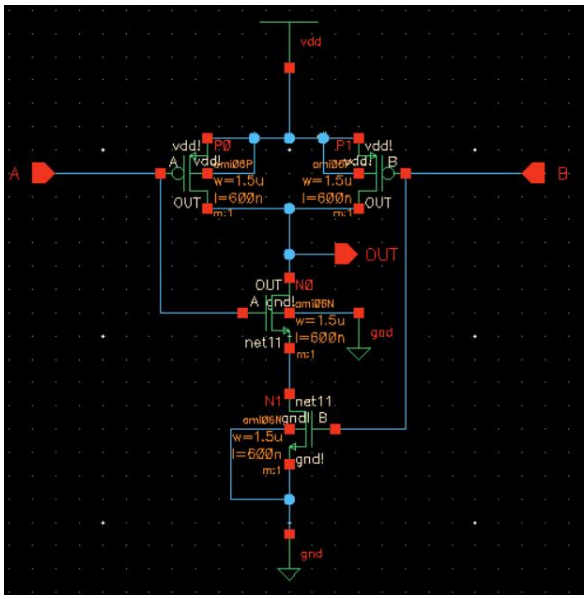


*Figure 12 Schematic of NAND Gate*

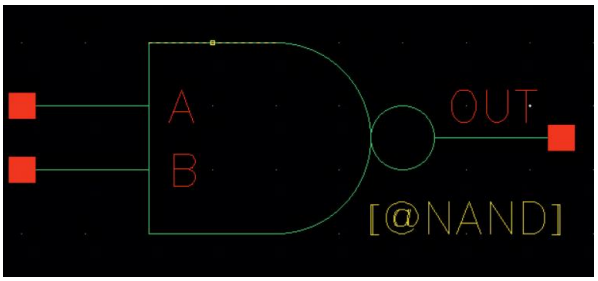


*Figure 13 Symbol of NAND Gate*

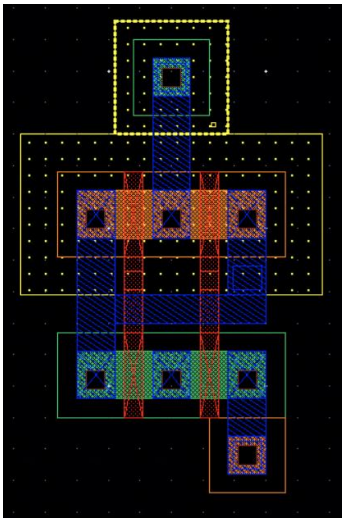

*Figure 14 Layout of NAND Gate*

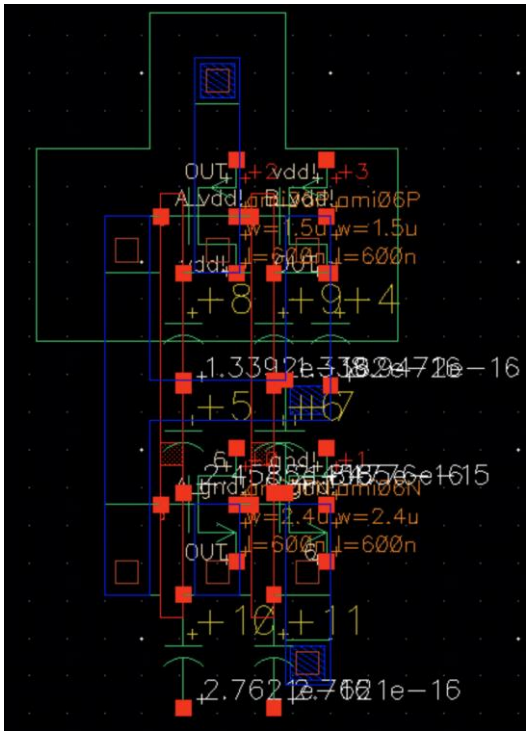


*Figure 15 Extracted View of NAND*

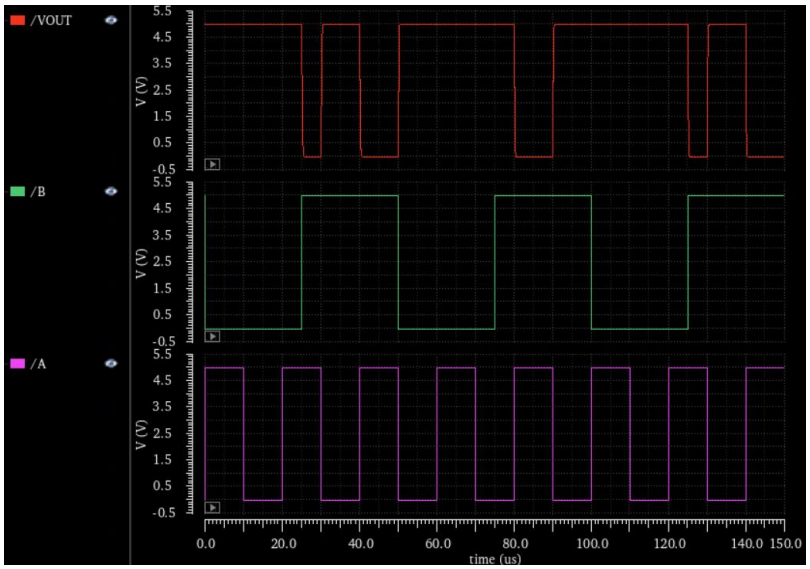


*Figure 16 Simulation Result of NAND Gate*

## AND Gate

AND gate is the combination of NAND and inverter, following is the schematic, symbol, layout, and simulation result of AND gate:

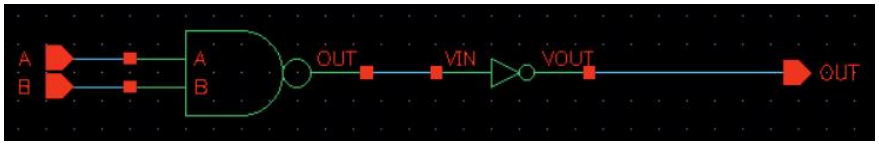


*Figure 17 Schematic of AND Gate*

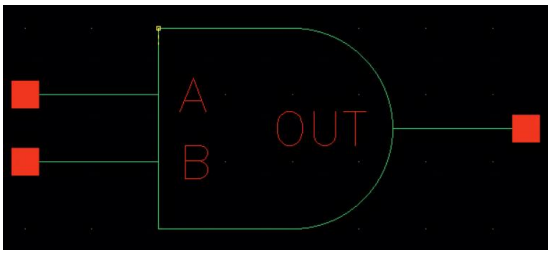


*Figure 18 Symbol of AND Gate*

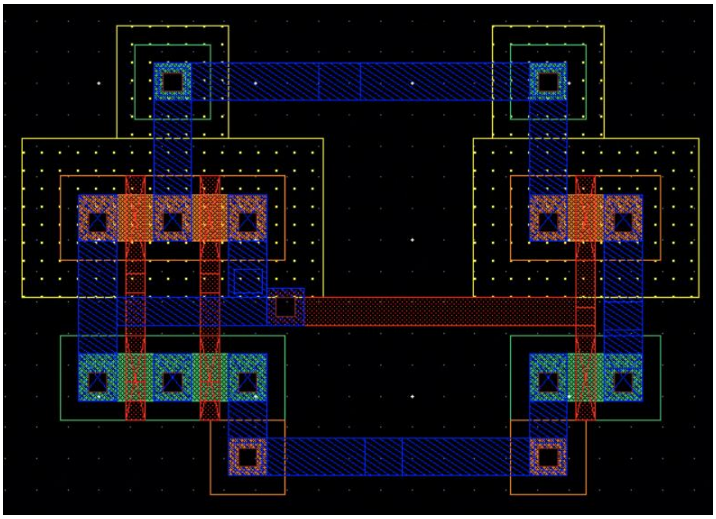

*Figure 19 Layout of AND Gate*

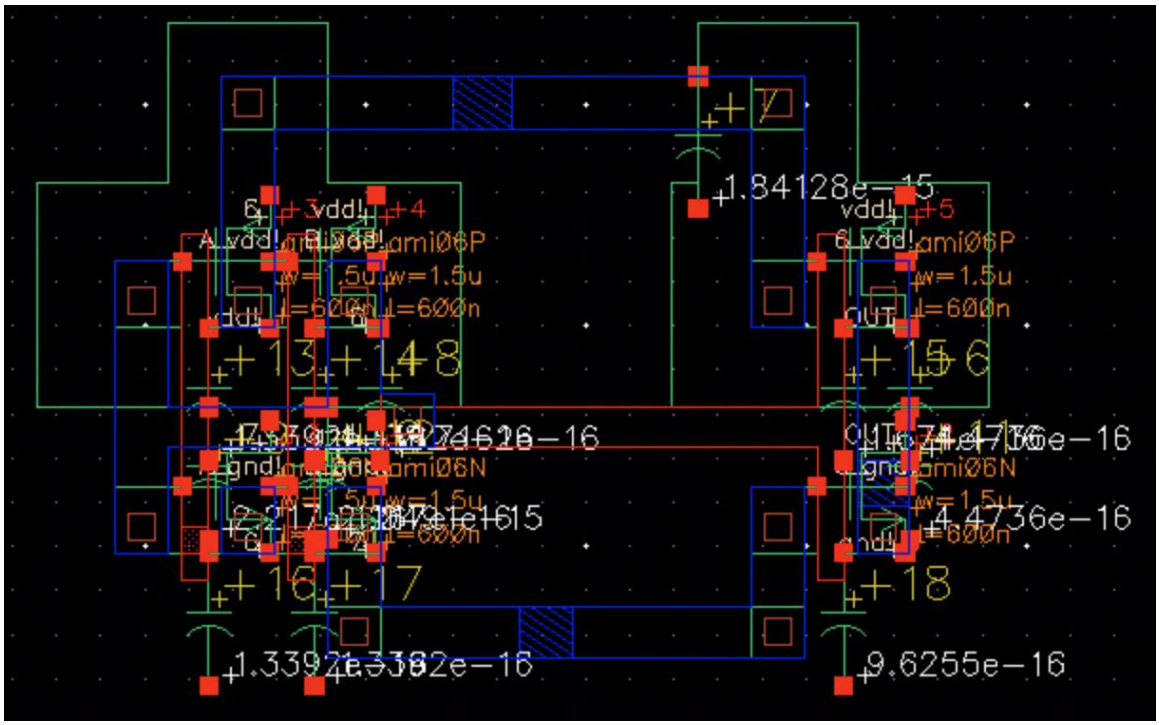


*Figure 20 Extracted View of AND*

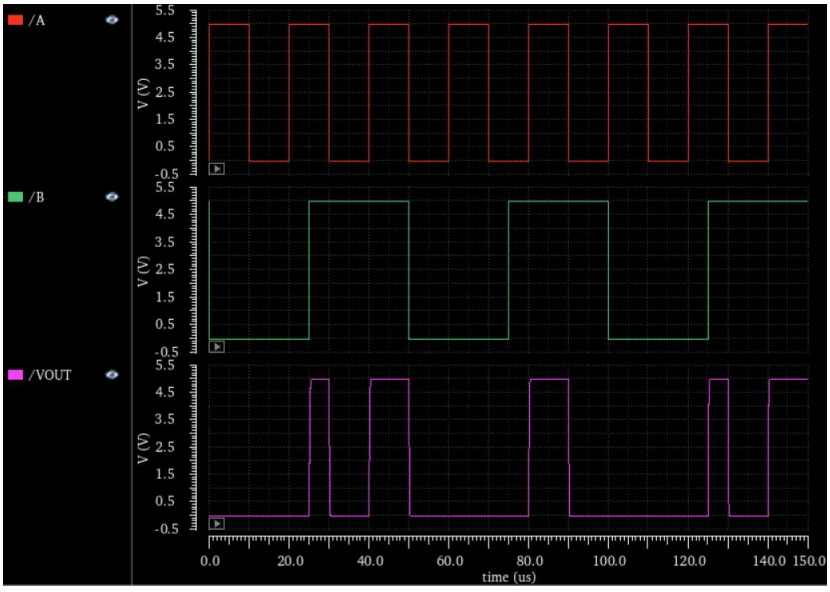


*Figure 21 Simulation Result of AND Gate*

## NOR Gate

Following is the schematic, symbol, layout, and simulation result of NOR gate:

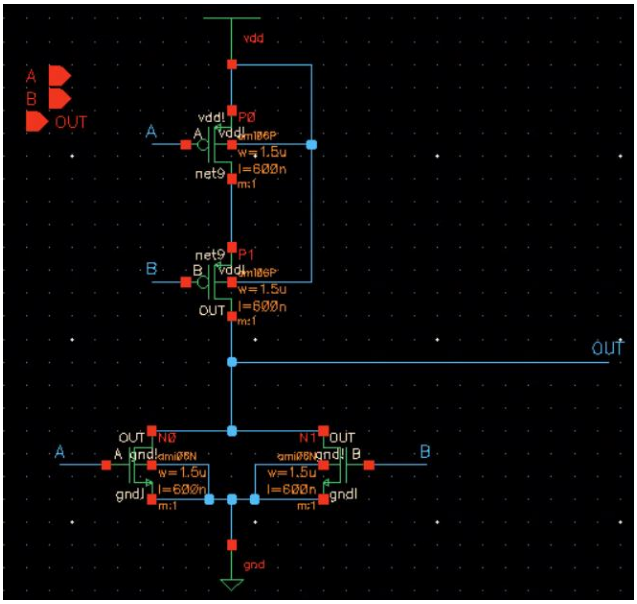


*Figure 22 Schematic of NOR Gate*

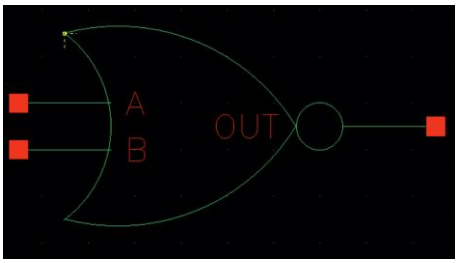


*Figure 23 Symbol of NOR Gate*

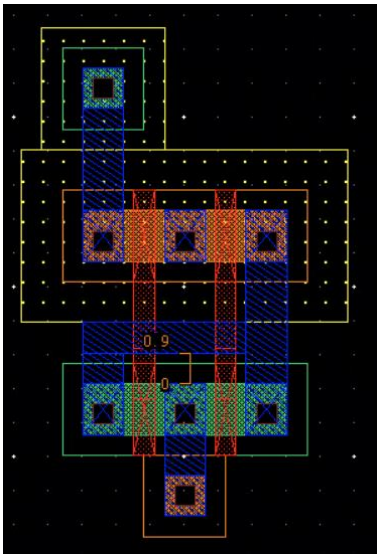

*Figure 24 Layout of NOR Gate*

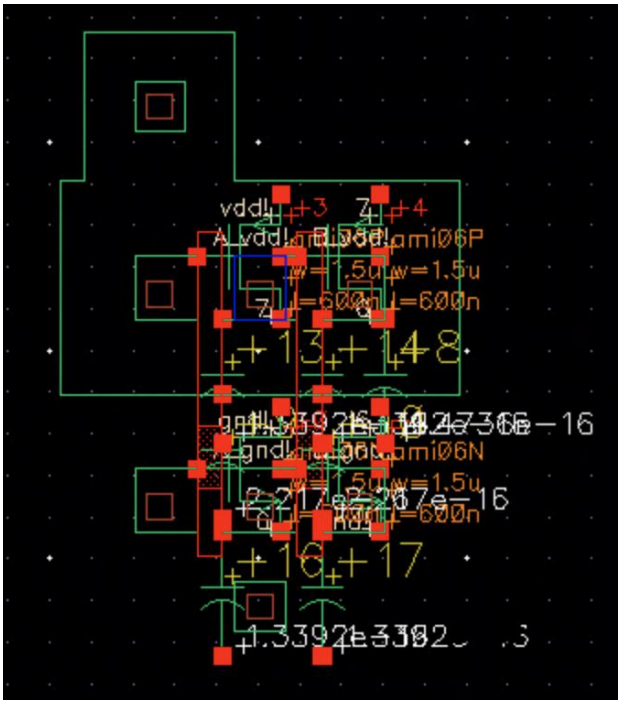


*Figure 25 Extracted View of NOR Gate*

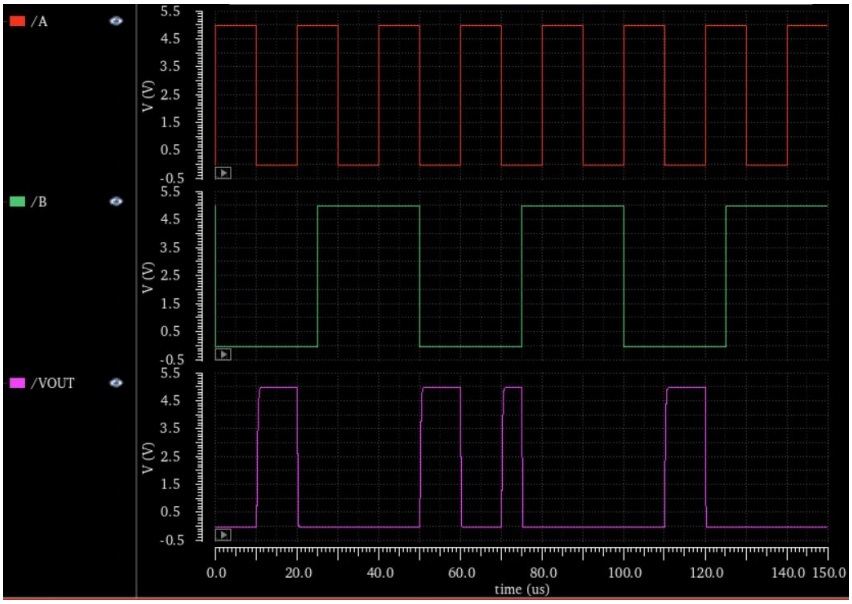


*Figure 26 Simulation Result of NOR Gate*

## OR Gate

OR Gate is the combination of NOR gate and inverter, following is the schematic, symbol, layout, and simulation result of OR gate:

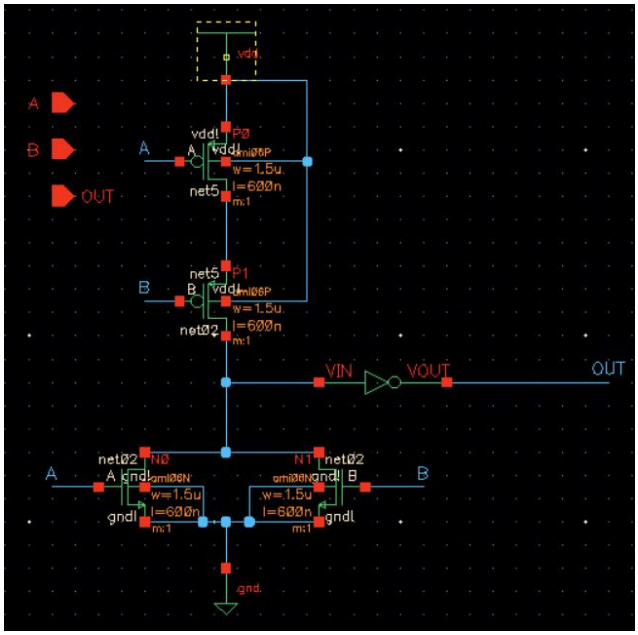


*Figure 27 Schematic of OR Gate*

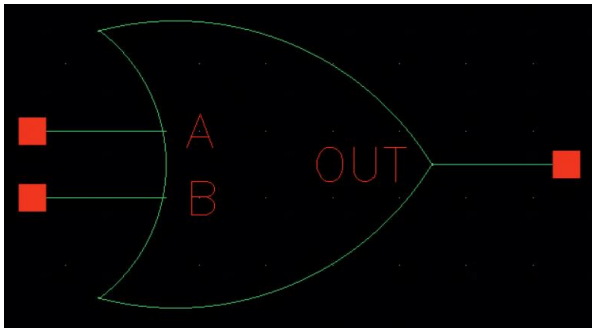


*Figure 28 Symbol of OR Gate*

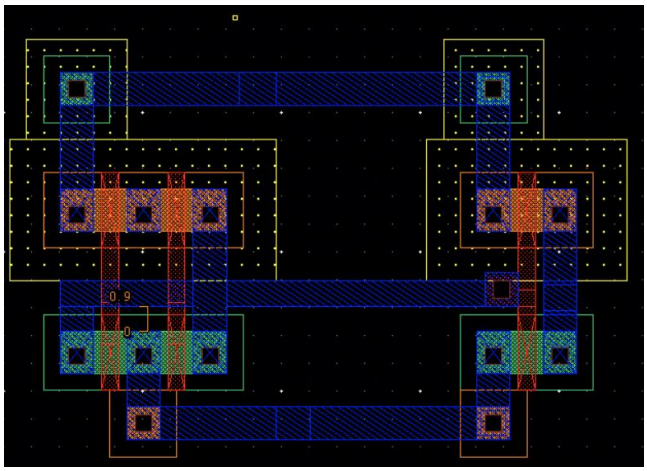

*Figure 29 Layout of OR Gate*

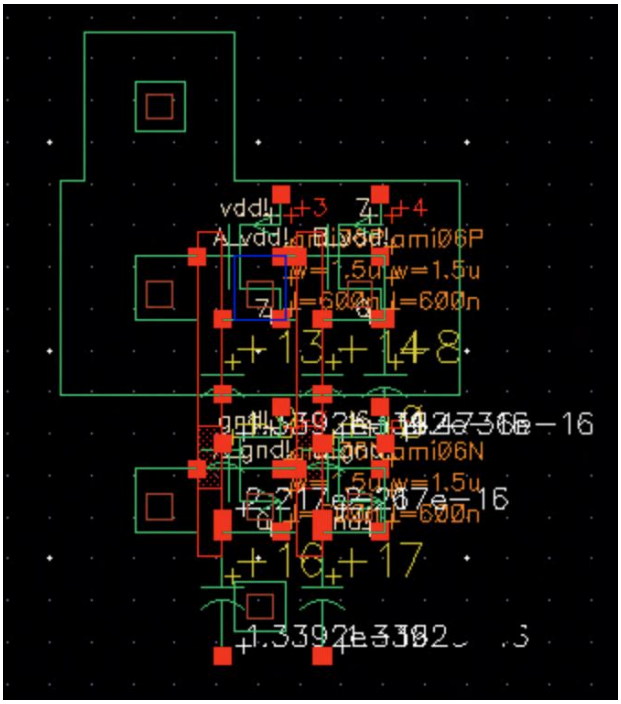


*Figure 30 Extracted View of OR Gate*

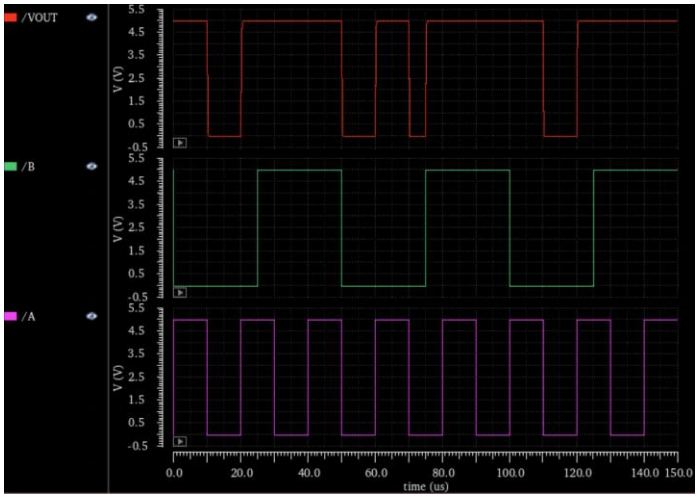


*Figure 31 Simulation Result of OR Gate*

## Transmission Gate

A transmission gate (TG) is an analog gate similar to a relay that can conduct in both directions or block by a control signal with almost any voltage potential, following is the schematic, symbol, layout, and simulation result of transmission gate:

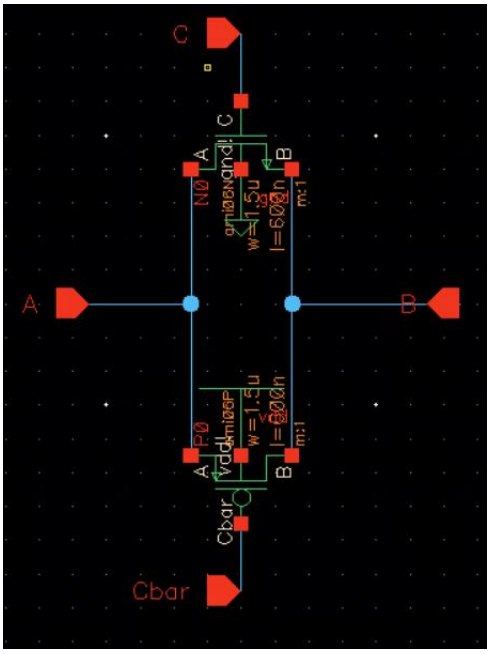


*Figure 32 Schematic of Transmission Gate*

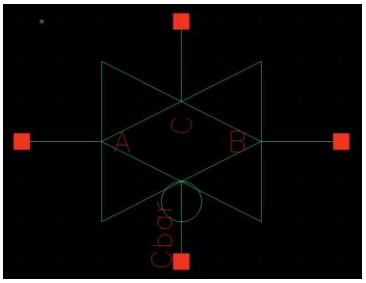


*Figure 33 Symbol of Transmission Gate*

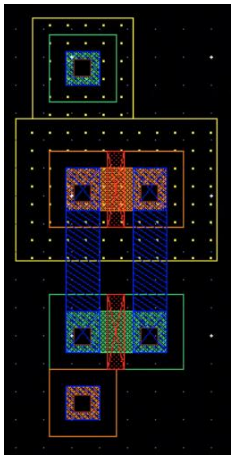

*Figure 34 Layout of Transmission Gate*

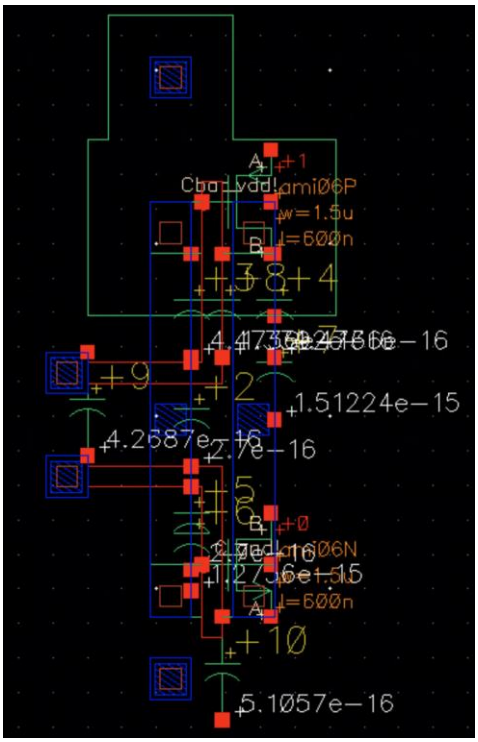


*Figure 35 Extracted View of Transmission Gate*

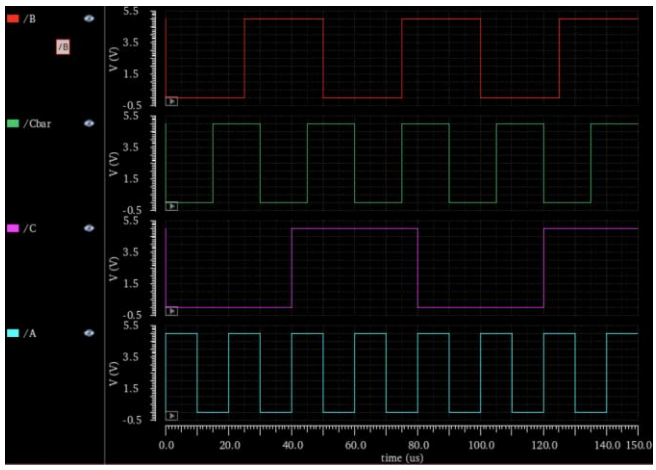


*Figure 36 Simulation Result of Transmission Gate*

## XOR Gate

There are two ways to make a XOR gate, which is using transmission gate and basic logic gate. In this project, the XOR gate is made of transmission gate. Following is the schematic, symbol, layout, and simulation result of XOR gate:

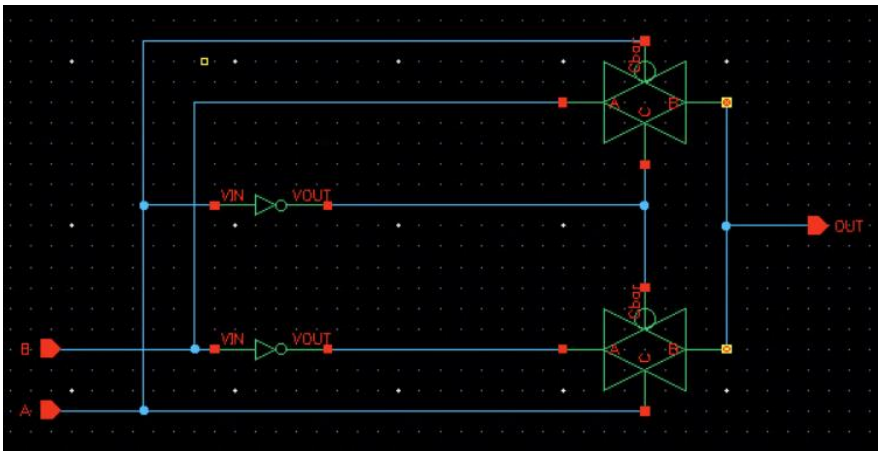


*Figure 37 Schematic of TG-XOR Gate*

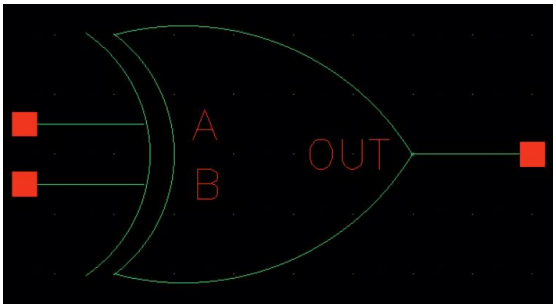


*Figure 38 Symbol of XOR Gate*

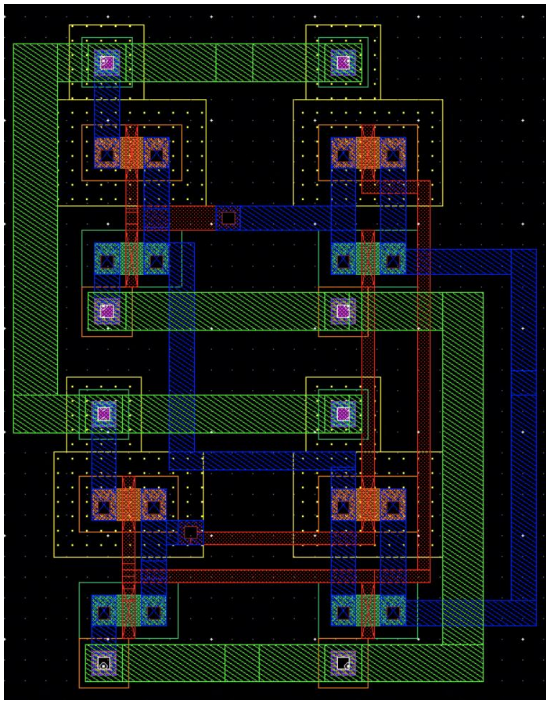

*Figure 39 Layout of XOR Gate*

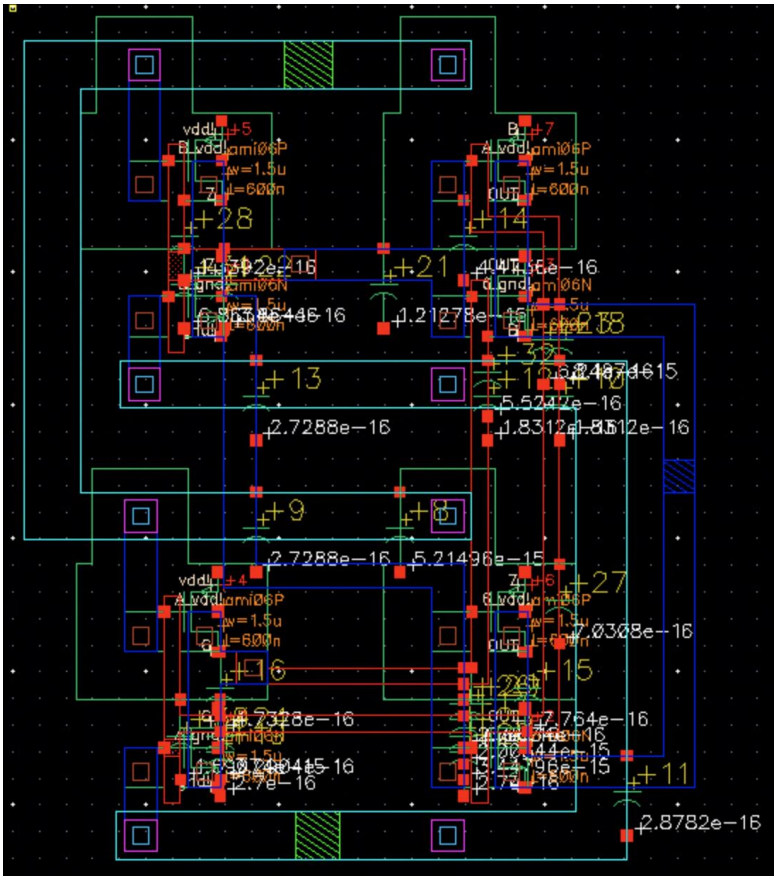


*Figure 40 Extracted View of XOR*

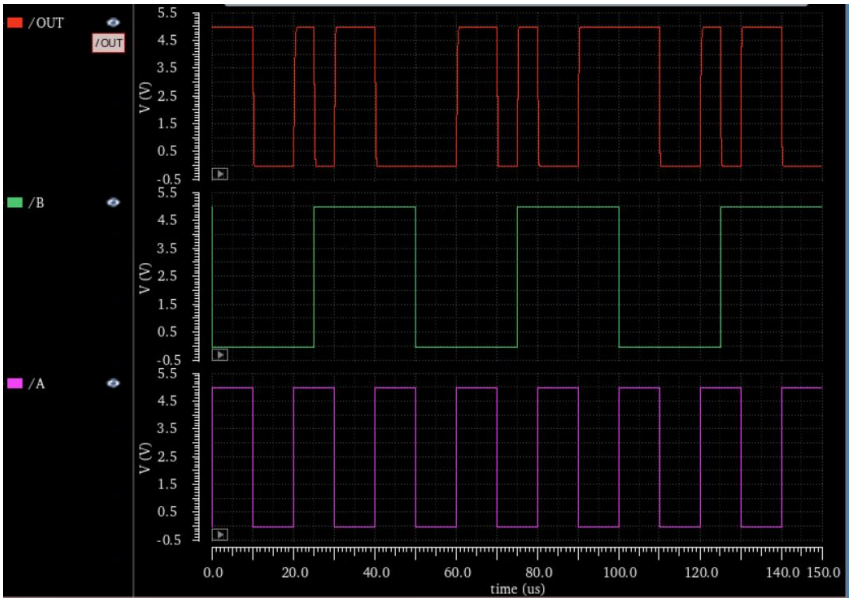


*Figure 41 Simulation Result of XOR Gate*

## XNOR Gate

XNOR gate is the combination of XOR Gate and inverter, following is the schematic, symbol, layout, and simulation result of XNOR gate:

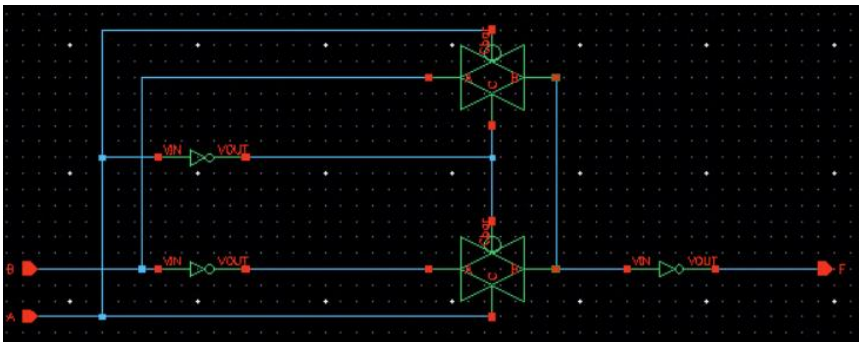


*Figure 42 Schematic of XNOR Gate*

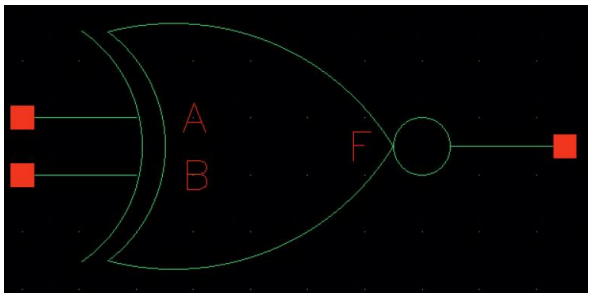


*Figure 43 Symbol of XNOR Gate*

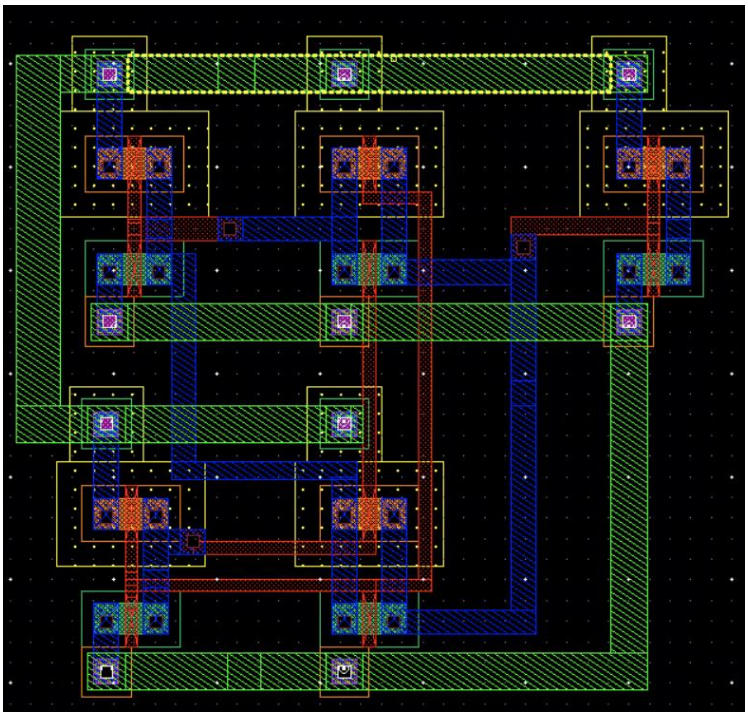

*Figure 44 Layout of XNOR Gate*

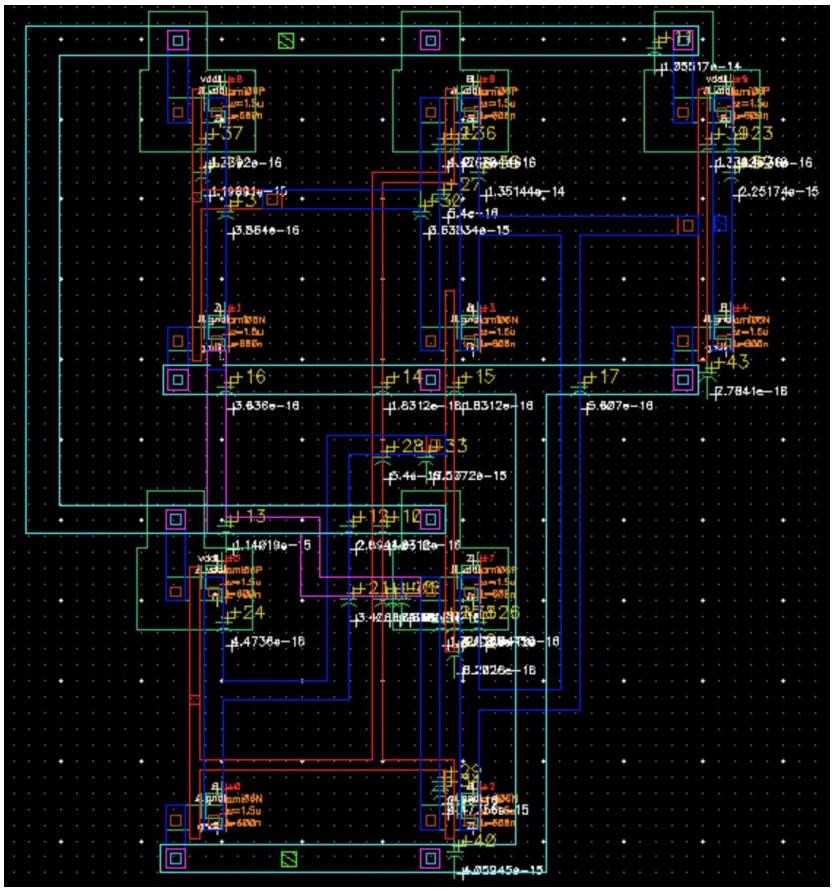

*Figure 45 Extracted View of XNOR*

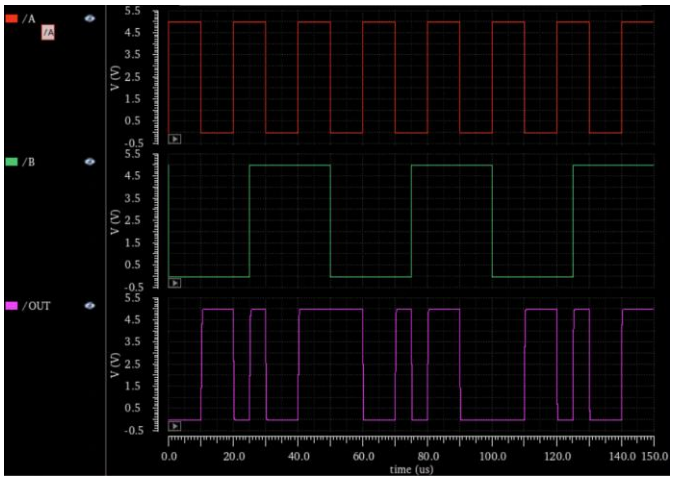


*Figure 46 Simulation Result of XNOR Gate*

# Combined Logic Device

## Half Adder

Half adder is an adder without carry in, and it consists of XOR gate and AND gate, following is the schematic, symbol, layout, and simulation result of Half Adder:f

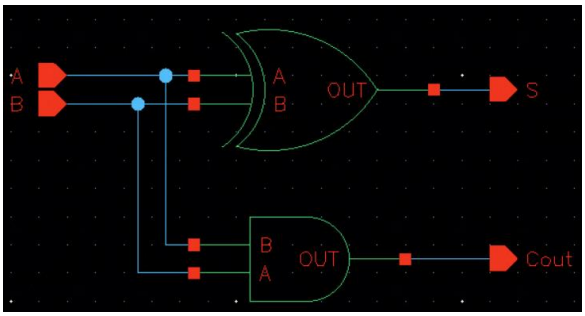


*Figure 47 Schematic of Half Adder*

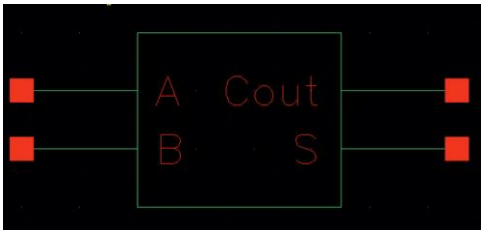


*Figure 48 Symbol of Half Adder*

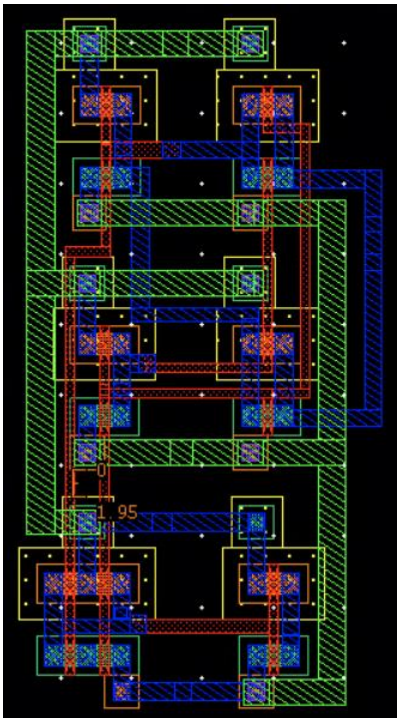


*Figure 49 Layout of Half Adder*

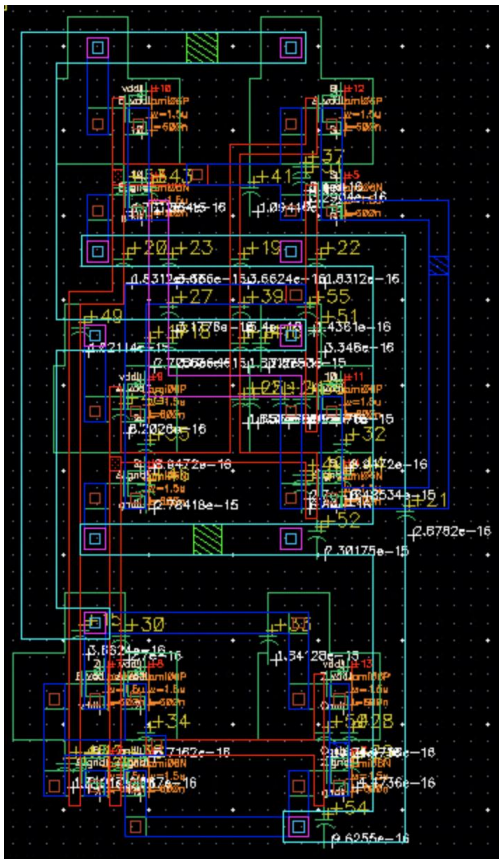

*Figure 50 Extracted View of Half Adder*

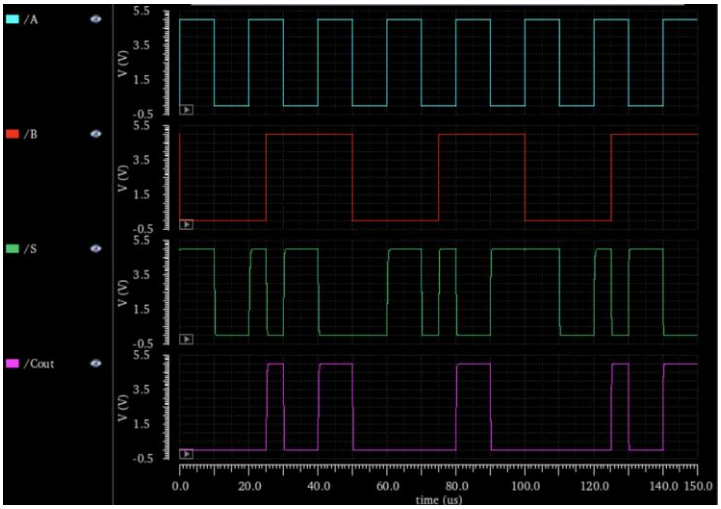


*Figure 51 Simulation Result of Half Adder*

## Full Adder

Full adder is capable of calculate carry in, and is made of XOR, OR and AND gate, following is the schematic, symbol, layout, and simulation result of Full Adder:

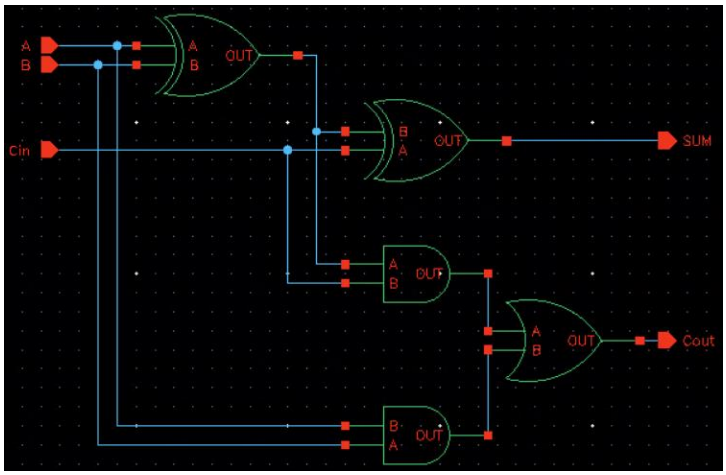


*Figure 52 Schematic of Full Adder*

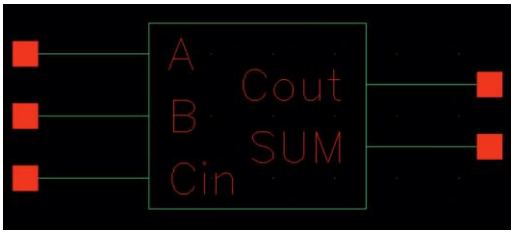


*Figure 53 Symbol of Full Adder*

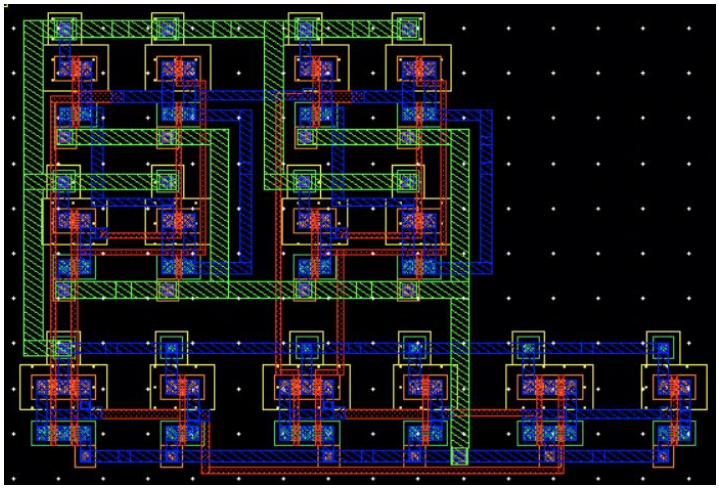

*Figure 54 Layout of Full Adder*

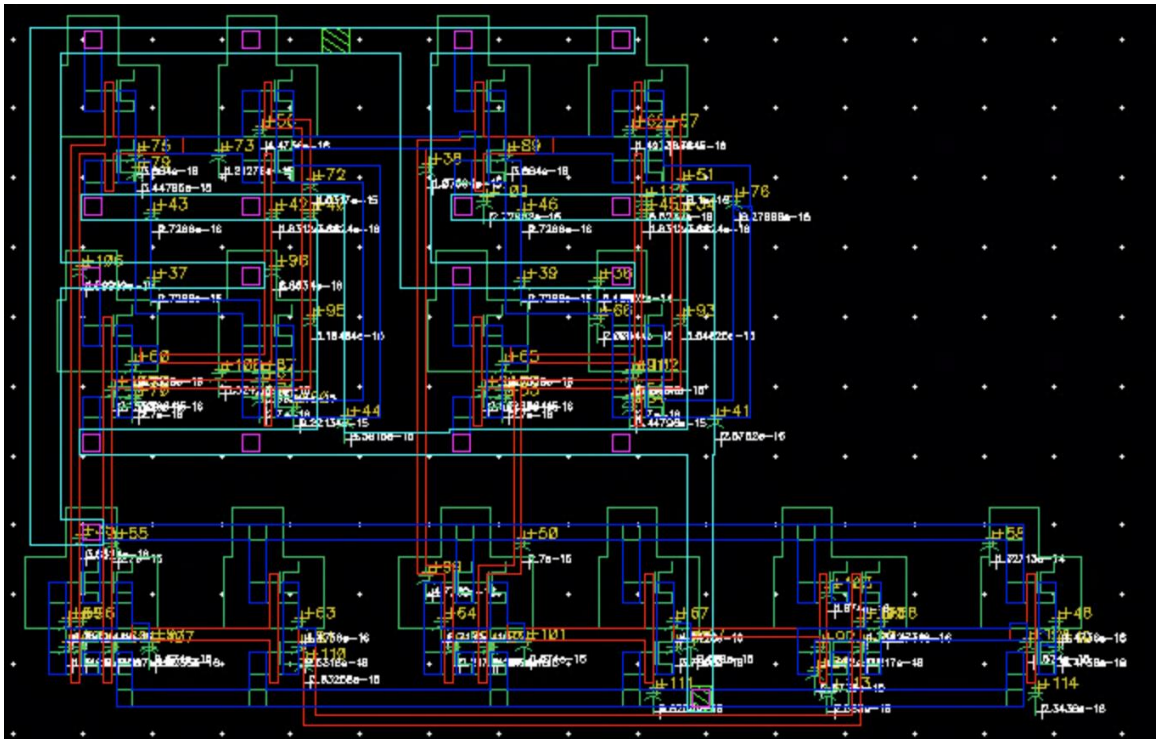

*Figure 55 Extracted View of Full Adder*

For my project, an 8-bit full adder and 9-bit full adder are used, which are all combination of half adder and full adder, the schematic and layout are as follows:

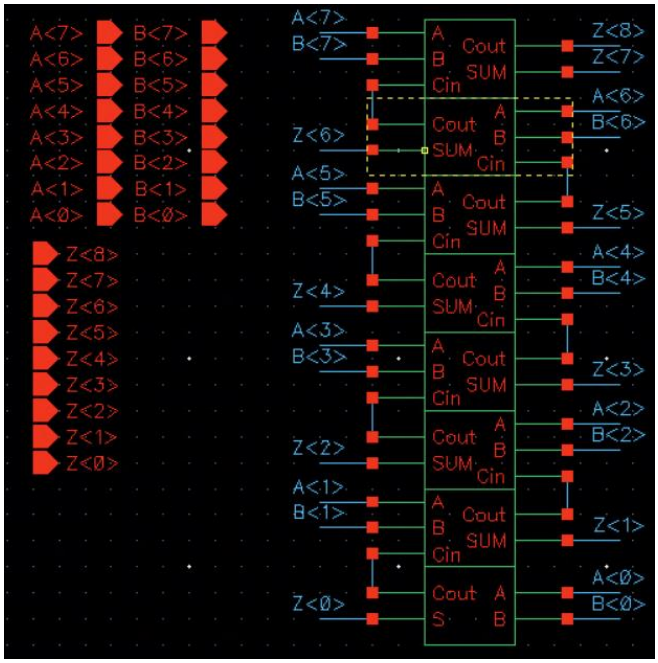


*Figure 56 8-bit Full Adder*

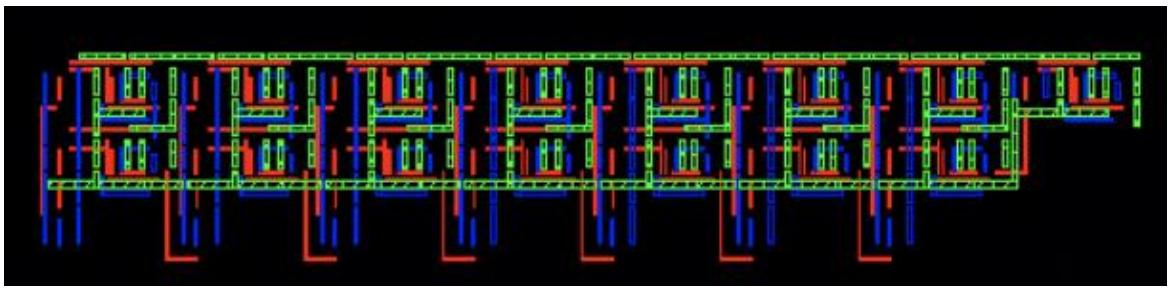

*Figure 57 Layout of 8-bit Full Adder*

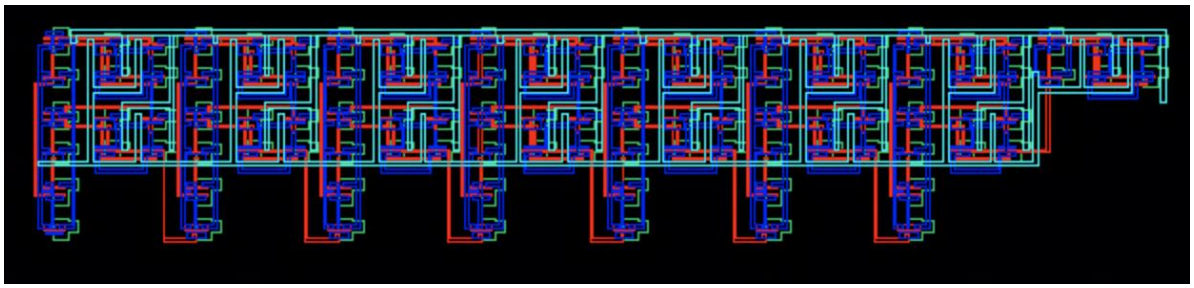

*Figure 58 Extracted View of 8-bit Full Adder*

The schematic and layout of 9-bit Full Adder just has 1 more full adder than 8-bit Full Adder, so it is not being display here. For the simulation, a Verilog program is used, and because the simulation process and result of 8-bit and 9-bit full adder is the same, only 8-bit full adder's information is displayed here:

```
//Verilog HDL for "Lab", "verilog_tb" "functional"
`timescale 10us/1us
module verilog_tb ( test_data_A, test_data_B, results );

    input [8:0] results;
    output [7:0] test_data_A;
    output [7:0] test_data_B;
    reg [7:0] test_data_A;
    reg [7:0] test_data_B;
    integer i;
    integer j;
    initial begin

        test_data_A = 0;
        test_data_B = 0;
        for (i=0;i<=15; i=i+1) begin
            test_data_B = 0;
            #1 test_data_A = test_data_A + 1;
            for (j=0;j<=5; j=j+1) begin
            #1 test_data_B = test_data_B + 1;
            #0.1 $display ("A = %d, B = %d, Result = %d", test_data_A, test_data_B, results);
            end
        end

    end

endmodule
```

*Figure 59 Verilog Code of 8-bit Full Adder Simulation*

```
    tran: time = 25.93 us    (2.59 %), step = 2.439 us     (244 m%)
A =   1, B =   2, Result =   3
A =   1, B =   3, Result =   4
A =   1, B =   4, Result =   5
A =   1, B =   5, Result =   6
    tran: time = 75 us        (7.5 %), step = 3.666 us     (367 m%)
A =   1, B =   6, Result =   7
A =   2, B =   1, Result =   3
A =   2, B =   2, Result =   4
A =   2, B =   3, Result =   5
    tran: time = 125.9 us    (12.6 %), step = 3.377 us     (338 m%)
A =   2, B =   4, Result =   6
A =   2, B =   5, Result =   7
A =   2, B =   6, Result =   8
A =   3, B =   1, Result =   4
    tran: time = 176.6 us    (17.7 %), step = 1.634 us     (163 m%)
A =   3, B =   2, Result =   5
A =   3, B =   3, Result =   6
A =   3, B =   4, Result =   7
A =   3, B =   5, Result =   8
    tran: time = 226.8 us    (22.7 %), step = 3.845 us     (384 m%)
A =   3, B =   6, Result =   9
A =   4, B =   1, Result =   5
A =   4, B =   2, Result =   6
A =   4, B =   3, Result =   7
    tran: time = 277.9 us    (27.8 %), step = 3.379 us     (338 m%)
A =   4, B =   4, Result =   8
A =   4, B =   5, Result =   9
A =   4, B =   6, Result =  10
A =   5, B =   1, Result =   6
    tran: time = 325.2 us    (32.5 %), step = 500 ns        (50 m%)
A =   5, B =   2, Result =   7
A =   5, B =   3, Result =   8
A =   5, B =   4, Result =   9
A =   5, B =   5, Result =  10
    tran: time = 376.1 us    (37.6 %), step = 3.5 us       (350 m%)
A =   5, B =   6, Result =  11
A =   6, B =   1, Result =   7
A =   6, B =   2, Result =   8
A =   6, B =   3, Result =   9
    tran: time = 427.1 us    (42.7 %), step = 2.179 us     (218 m%)
A =   6, B =   4, Result =  10
A =   6, B =   5, Result =  11
A =   6, B =   6, Result =  12
    tran: time = 475.5 us    (47.6 %), step = 3.4 us       (340 m%)
A =   7, B =   1, Result =   8
A =   7, B =   2, Result =   9
A =   7, B =   3, Result =  10
A =   7, B =   4, Result =  11
A =   7, B =   5, Result =  12
```

*Figure 60 Simulation Result of 8-bit Full Adder*

## Multiplier

A 4-bit unsigned integer multiplier is used in my project for the multiplication computation of convolution, which is consist of AND, half adder and full adder, following is the schematic, symbol, layout, and simulation result of Multiplier:

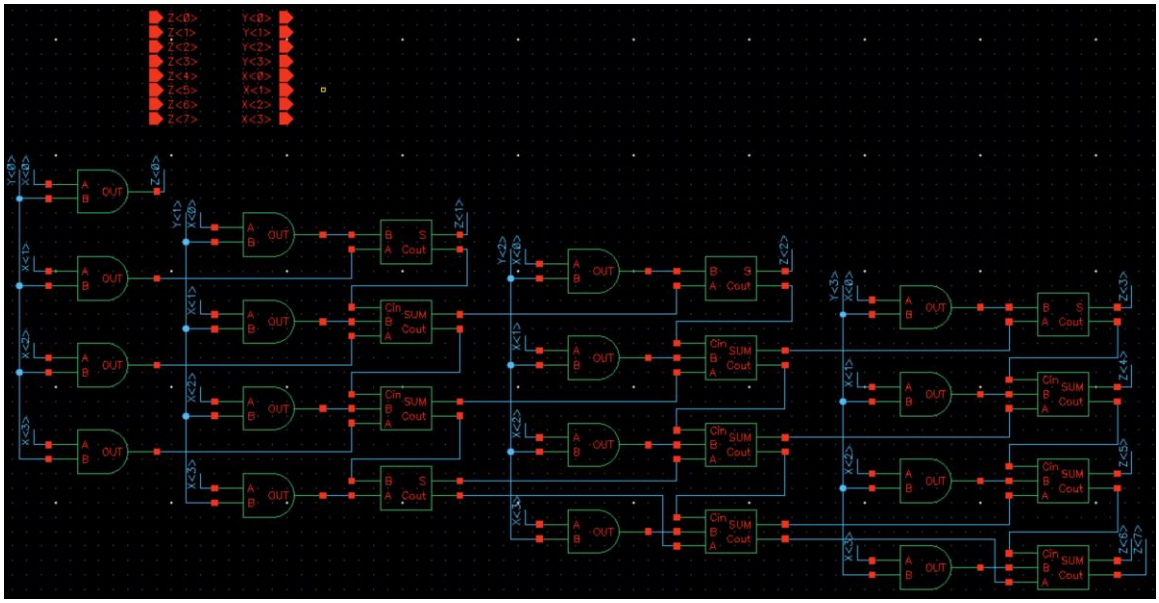

*Figure 61 Schematic of Multiplier*

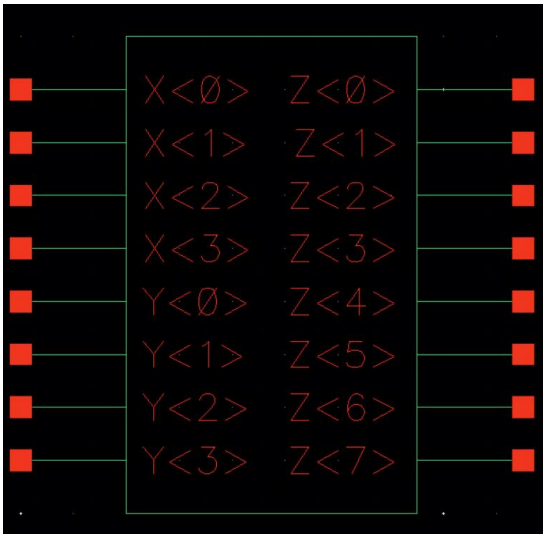


*Figure 62 Symbol of Multiplier, X<0:3> and Y<0:3> is multiplicator, Z<0:7> is result*

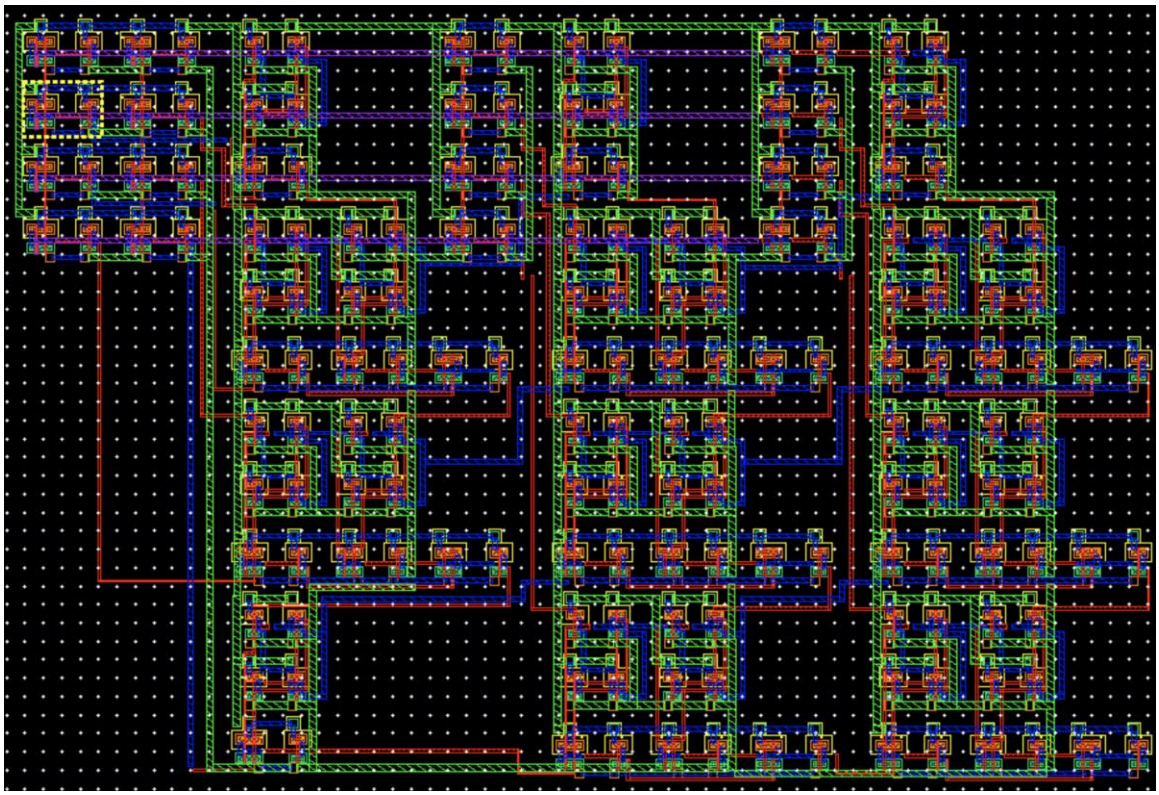

*Figure 63 Layout of 4-bit Multiplier*

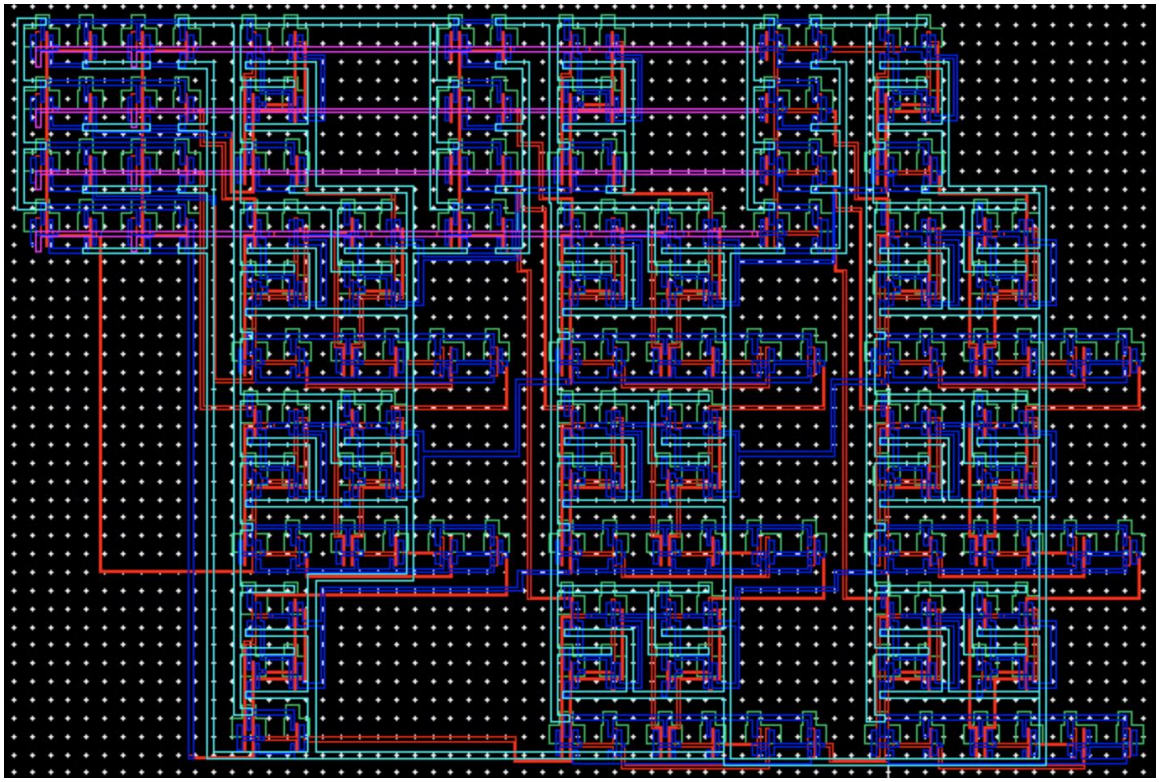

*Figure 64 Extracted View of 4-bit Multiplier*

A Verilog program is used for simulation:

```
//Verilog HDL for "Lab", "verilog_tb" "functional"
`timescale 10us/1us
module verilog_tb ( test_data_A, test_data_B, results );

    input [7:0] results;
    output [3:0] test_data_A;
    output [3:0] test_data_B;
    reg [3:0] test_data_A;
    reg [3:0] test_data_B;
    integer i;
    integer j;
    initial begin

        test_data_A = 4'b0000;
        test_data_B = 4'b0000;
        for (i=0;i<=15; i=i+1) begin
            test_data_B = 4'b0000;
            #1 test_data_A = test_data_A + 4'b0001;
            for (j=0;j<=5; j=j+1) begin
            #1 test_data_B = test_data_B + 4'b0001;
            #0.1 $display ("A = %d, B = %d, Result = %d", test_data_A, test_data_B, results);
            end
        end

    end

endmodule
```

*Figure 65 Verilog Code for multiplier simulation*

```
A =  1, B =  2, Result =   2
A =  1, B =  3, Result =   3
A =  1, B =  4, Result =   4
A =  1, B =  5, Result =   5
    tran: time = 75 us        (7.5 %), step = 1.198 us     (120 m%)
A =  1, B =  6, Result =   6
A =  2, B =  1, Result =   2
A =  2, B =  2, Result =   4
A =  2, B =  3, Result =   6
    tran: time = 125.2 us    (12.5 %), step = 2.457 us     (246 m%)
A =  2, B =  4, Result =   8
A =  2, B =  5, Result =  10
A =  2, B =  6, Result =  12
A =  3, B =  1, Result =   3
    tran: time = 176.3 us    (17.6 %), step = 1.526 us     (153 m%)
A =  3, B =  2, Result =   6
A =  3, B =  3, Result =   9
A =  3, B =  4, Result =  12
A =  3, B =  5, Result =  15
    tran: time = 225.4 us    (22.5 %), step = 3.374 us     (337 m%)
A =  3, B =  6, Result =  18
A =  4, B =  1, Result =   4
A =  4, B =  2, Result =   8
A =  4, B =  3, Result =  12
    tran: time = 275.2 us    (27.5 %), step = 1.838 us     (184 m%)
A =  4, B =  4, Result =  16
A =  4, B =  5, Result =  20
A =  4, B =  6, Result =  24
A =  5, B =  1, Result =   5
    tran: time = 325 us      (32.5 %), step = 301.2 ns    (30.1 m%)
A =  5, B =  2, Result =  10
A =  5, B =  3, Result =  15
A =  5, B =  4, Result =  20
A =  5, B =  5, Result =  25
    tran: time = 375.2 us    (37.5 %), step = 2.337 us     (234 m%)
A =  5, B =  6, Result =  30
A =  6, B =  1, Result =   6
A =  6, B =  2, Result =  12
A =  6, B =  3, Result =  18
    tran: time = 425.2 us    (42.5 %), step = 1.126 us     (113 m%)
A =  6, B =  4, Result =  24
A =  6, B =  5, Result =  30
A =  6, B =  6, Result =  36
    tran: time = 475.9 us    (47.6 %), step = 3.337 us     (334 m%)
A =  7, B =  1, Result =   7
A =  7, B =  2, Result =  14
A =  7, B =  3, Result =  21
A =  7, B =  4, Result =  28
A =  7, B =  5, Result =  35
```

*Figure 66 Simulation Result of Multiplier*

## S-R Flip-flop

Memory is essential for logic circuits design, for my project, S-R Flip-flop is used as lowest memory unit to ensure the overall computation logic. following is the schematic, symbol, layout, and simulation result of S-R Flip-flop:

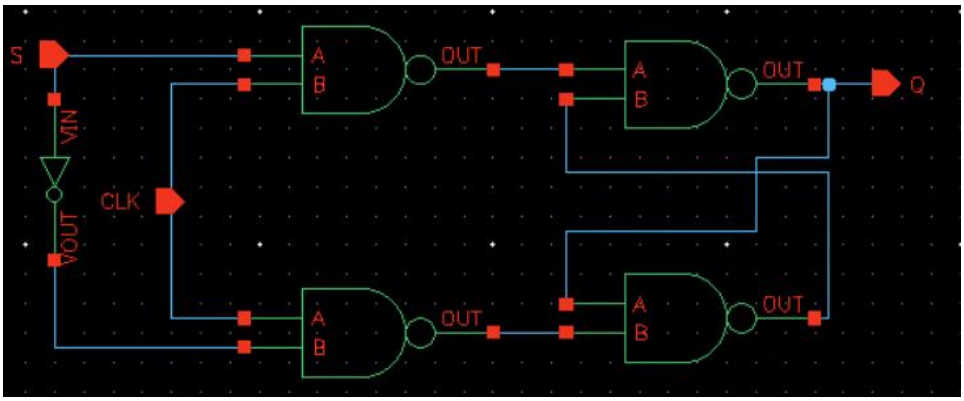


*Figure 67 Schematic of S-R Flip-flop*

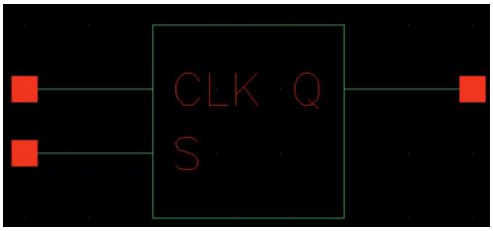


*Figure 68 Symbol of S-R Flip-flop*

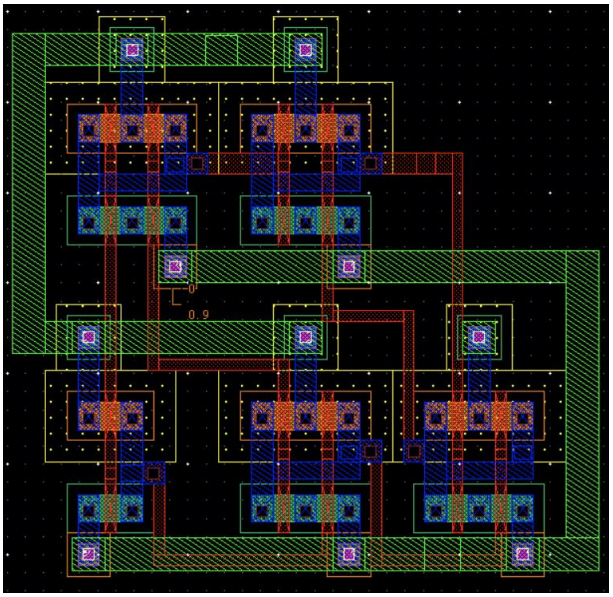

*Figure 69 Layout of S-R Flip-flop*

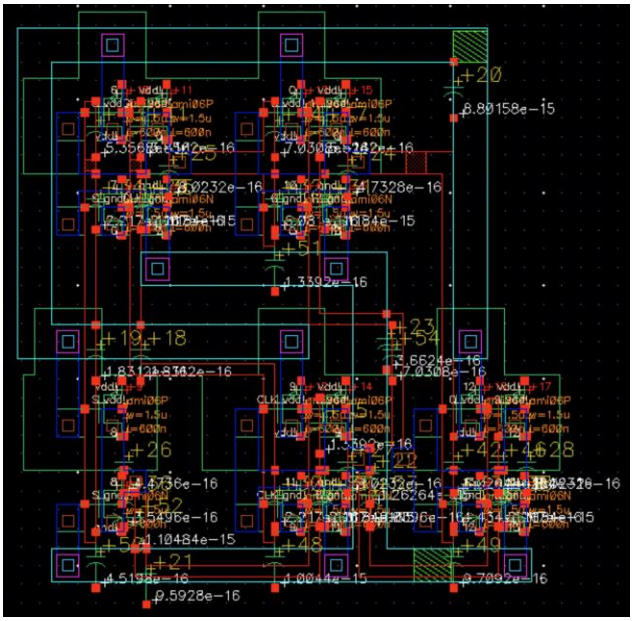

*Figure 70 Extracted View of SR Flip-flop*

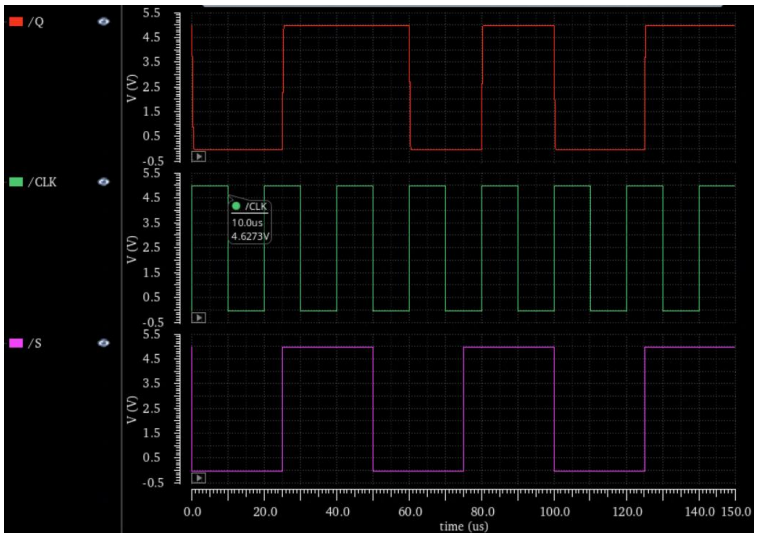


*Figure 71 Simulation Result of S-R Flip-flop*

## Serial-in Parallel-out Flip-flop

12 bits of data excluding convolutional kernel is needed for doing one cycle of convolution operation, however if we input 12 bits of data in parallel, there will not enough pins on the pad frame, so we need a shift register that can parallelize the input data, in other word, a Serial-in Parallel-out Flip-flop, combining the S-R Flip-flop could achieve that. Following is the schematic, symbol, layout, and simulation result of 1-bit-in 3-bit-out Flip-flop:

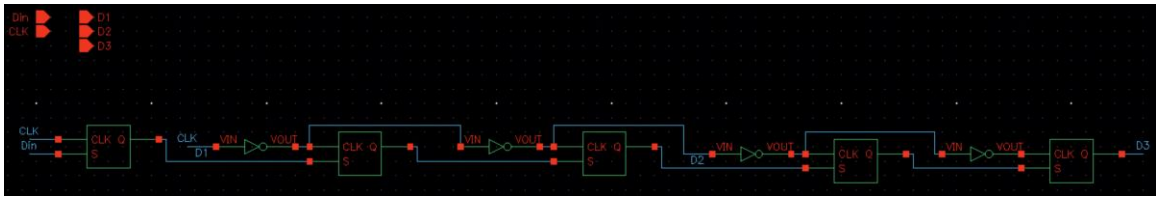


*Figure 72 Schematic of 1-bit-in 3-bit-out Flip-flop*

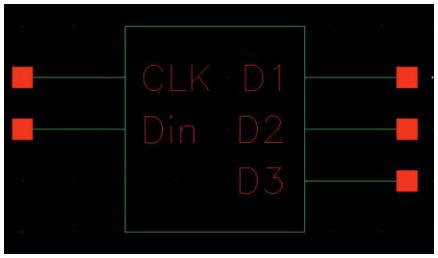


*Figure 73 Symbol of 1-bit-in 3-bit-out Flip-flop*

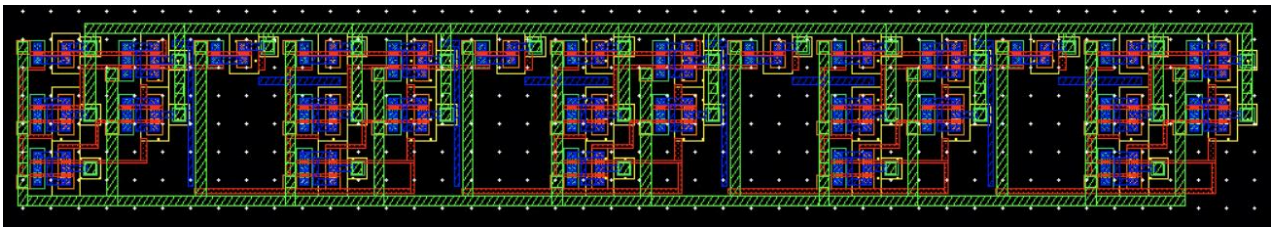

*Figure 74 Layout of 1-bit-in 3-bit-out Flip-flop*

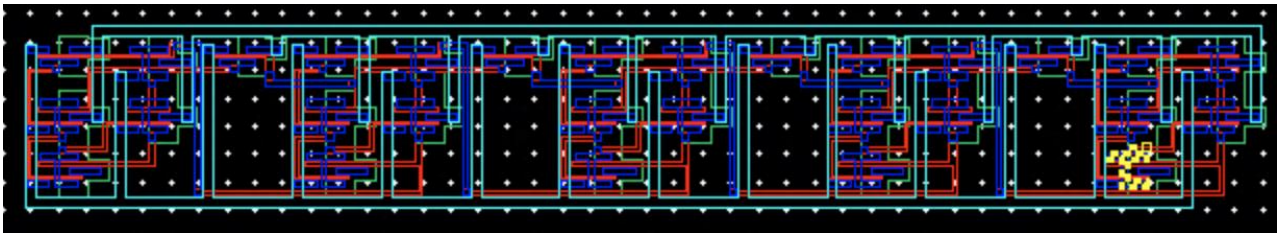

*Figure 75 Extracted View of 1-bit-in 3-bit-out Flip-flop*

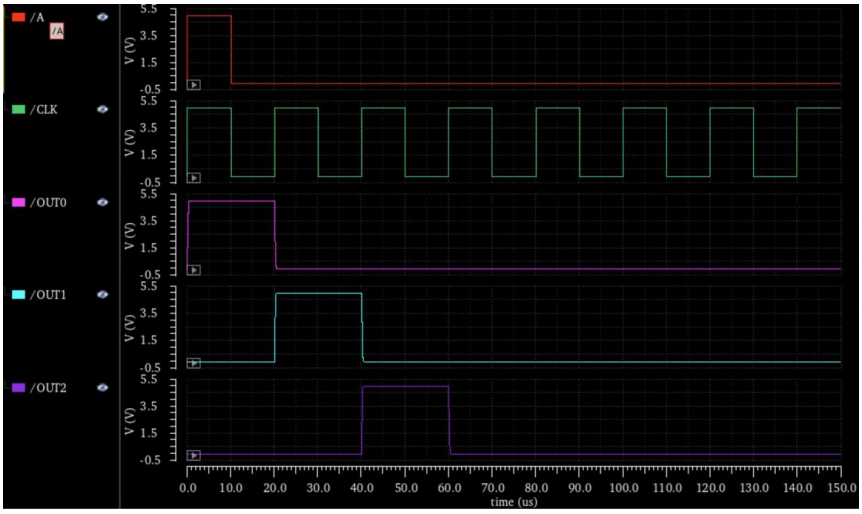


*Figure 76 Simulation Result of 1-bit-in 3-bit-out Flip-flop*

From the simulation result we could see that the input bit will be “pushed” to next register after a clock cycle, then next input bit will come in, after at least 3 clock cycles, the 3 bits of output will be the parallelized 3 bit of input data. Since my project need 4-bit input data, so we can combine 4 of 1-bit-in 3-bit-out Flip-flop:

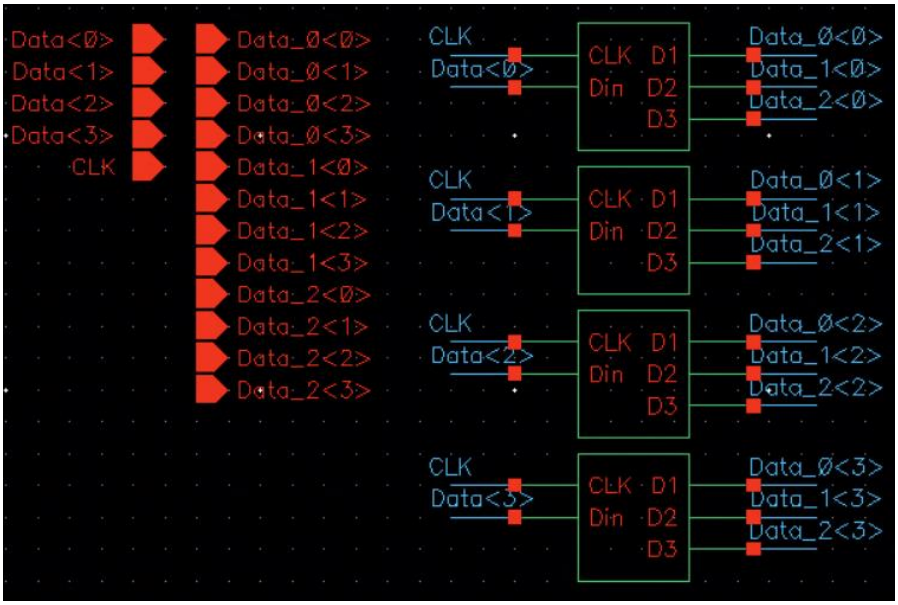


*Figure 77 Schematic of 4bits 1-bit-in 3-bit-out Flip-flop*

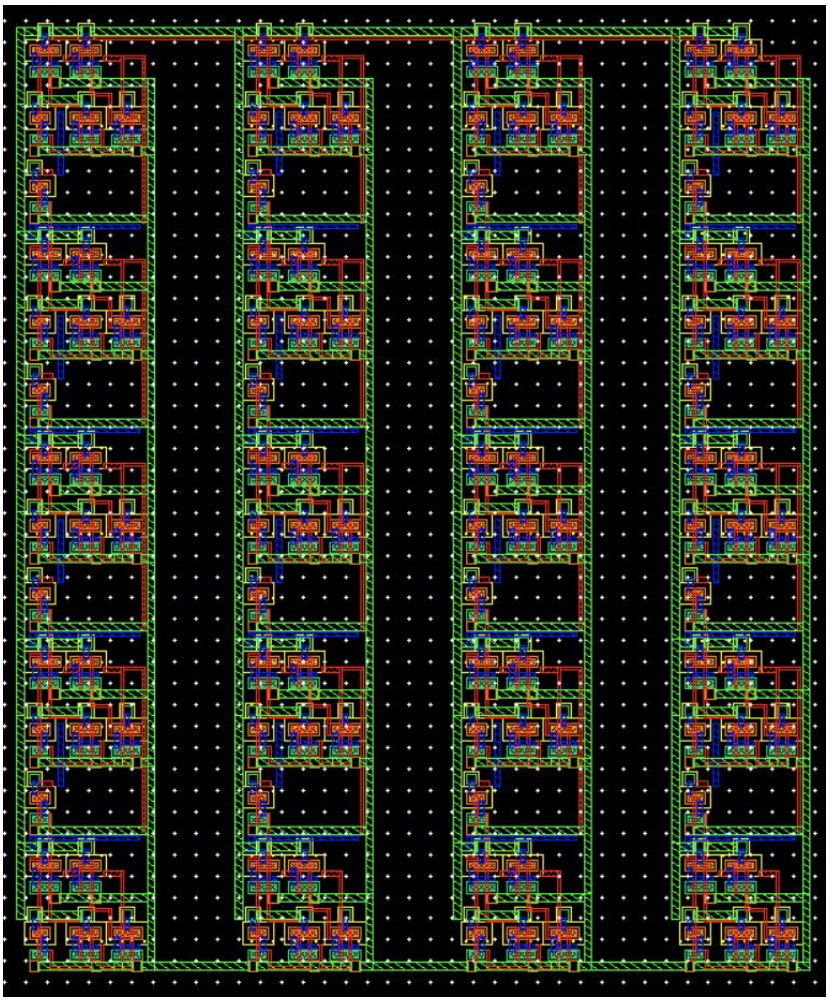

*Figure 78 Layout of 4bits 1-bit-in 3-bit-out Flip-flop*

## Pad Frame Layout and Pin Description

Whole design is being placed on a nearly 900um*900um pad frame, all data input/output are connected to physical pins for fabrication:

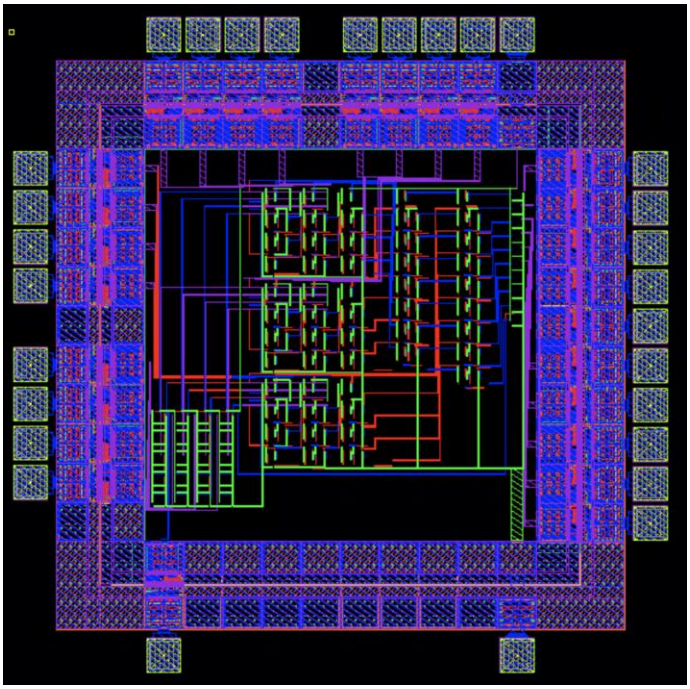

*Figure 79 Pad Frame of Chip Design*

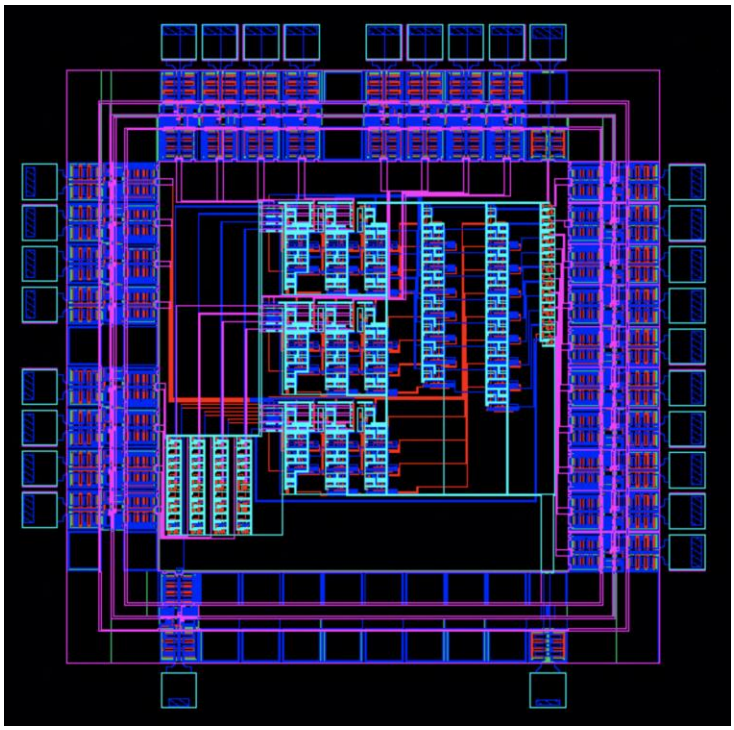

*Figure 80 Extracted View of Pad Frame*

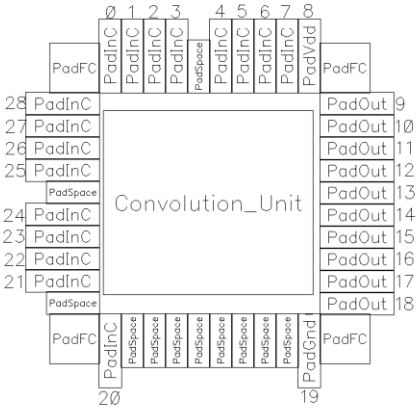


*Figure 81 Pinout*

*Table 1 Pin Description*

| Pin Number | Pin Name | Function |
|---|---|---|
| 0 | Kernel0<0> | 1st bit of kernel0 |
| 1 | Kernel0<1> | 2nd bit of kernel0 |
| 2 | Kernel0<2> | 3rd bit of kernel0 |
| 3 | Kernel0<3> | 4th bit of kernel0 |
| 4 | Kernel1<0> | 1st bit of kernel1 |
| 5 | Kernel1<1> | 2nd bit of kernel1 |
| 6 | Kernel1<2> | 3rd bit of kernel1 |
| 7 | Kernel1<3> | 4th bit of kernel1 |
| 8 | vdd! | VDD |
| 9 | OUT<0> | 1st bit of output |
| 10 | OUT<1> | 2nd bit of output |
| 11 | OUT<2> | 3rd bit of output |
| 12 | OUT<3> | 4th bit of output |
| 13 | OUT<4> | 5th bit of output |
| 14 | OUT<5> | 6th bit of output |
| 15 | OUT<6> | 7th bit of output |
| 16 | OUT<7> | 8th bit of output |
| 17 | OUT<8> | 9th bit of output |
| 18 | OUT<9> | 10th bit of output |
| 19 | gnd! | GND |
| 20 | CLK | Clock signal |
| 21 | Data<3> | 4th bit of input data |
| 22 | Data<2> | 3rd bit of input data |
| 23 | Data<1> | 2nd bit of input data |
| 24 | Data<0> | 1st bit of input data |
| 25 | Kernel2<0> | 1st bit of kernel2 |
| 26 | Kernel2<1> | 2nd bit of kernel2 |
| 27 | Kernel2<2> | 3rd bit of kernel2 |
| 28 | Kernel2<3> | 4th bit of kernel2 |

## Simulation of Completed Convolution Chip

Since the desired design is convolution operation, the logic cannot be testified with simple transient simulation, a Verilog program is coded for the test:

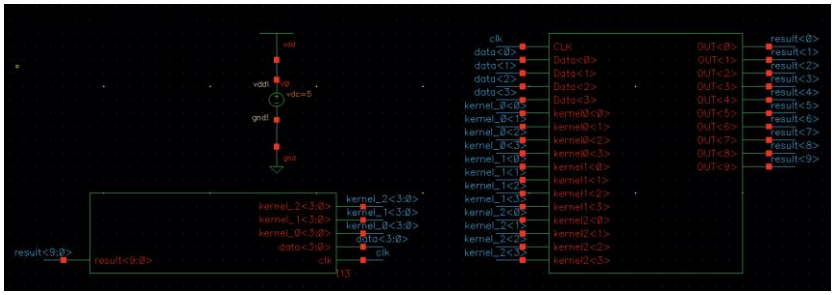

*Figure 82 Verilog Testbench of Convolution*

```
//Verilog HDL for "Lab", "verilog_tb_driver" "functional"
`timescale 10us/1us

module verilog_tb_driver ( clk, kernel_0, kernel_1, kernel_2, data, result );
    input [9:0] result;
    output [3:0] data;
    output [3:0] kernel_0;
    output [3:0] kernel_1;
    output [3:0] kernel_2;
    output clk;

    reg [3:0] kernel_0 = 1;
    reg [3:0] kernel_1 = 3;
    reg [3:0] kernel_2 = 1;
    reg [3:0] data = 0;
    reg [3:0] data_3 = 0;
    reg [3:0] data_1 = 0;
    reg [3:0] data_2 = 0;
    reg clk = 0;

    always #1 clk = ~clk;
    always @(negedge clk) begin
        #0.1;
        data = $random();
        data_3 = data_2;
        data_2 = data_1;
        data_1 = data;
    end
    always @(posedge clk) begin
        #0.1 $display ("d0 = %d, d1 = %d, d2 = %d, result = %d, clk = %d", data_1, data_2, data_3, result, clk);
    end

endmodule
```

*Figure 83 Verilog Code for Simulation*

The program input data every negative edge of clock signal, and the unit will do computation during the low-level voltage of clock signal, then the output memory will store the result at next high-level voltage of clock signal and keep it for an entire cycle, the simulation result as follows:

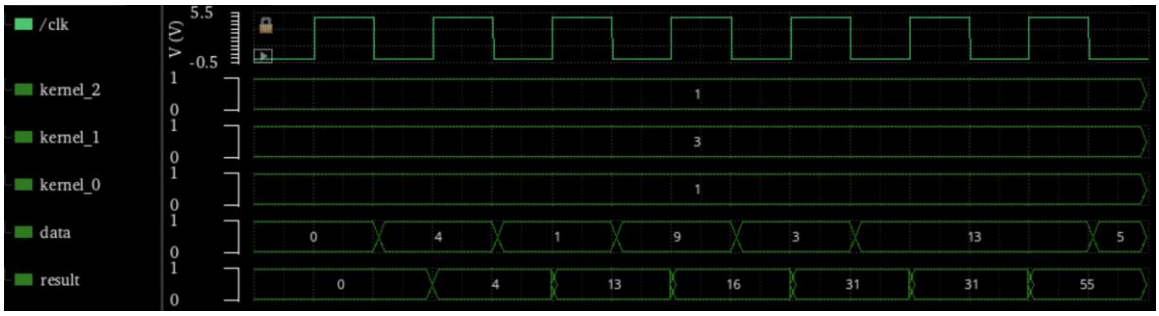


*Figure 84 The Simulation Result of Convolution, run with 100KHz Clock Rate*

Since the unit has complex device, it is hard to test the max delay directly, by multiple times simulation, the average max delay is 0.0130204us, which is nearly 130ns, that makes the maximum clock rate of the system to nearly 7.7MHz, comparing to standard microphone sample rate 48 KHz, our device could do real time voice signal processing; for accelerometer and gyroscope, the sampling rate is normally around 6.6KHz to 4KHz, which means my device could also do real time kinetic sensing data processing.

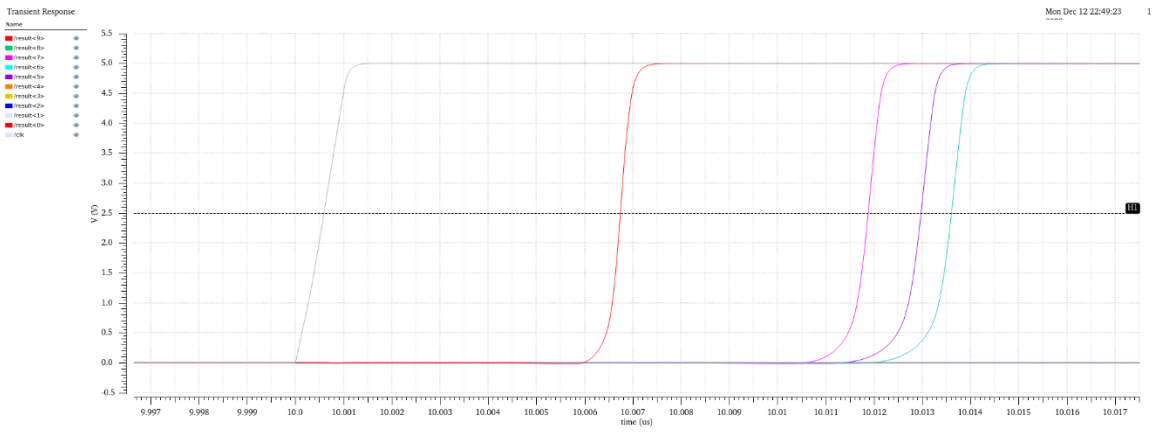


*Figure 85 Transient Simulation*

# Conclusion

Overall speaking, I designed a ALU that could do 1-dimensional convolution with 4-bit unsigned integer, which could be used for voice signal, kinetic signal, and other sensor signal processing in real time due to the high speed to 7.7MHz clock rate it supports.

Yet, there is several improvements I could do if there is more time:

a) Tune the CMOS space to get better VTC respond.
b) Reroute the layout to get lower parasitic capacitance and faster speed.
c) Add subtractor and divider into ALU to support more kernels.