DIGITAL CIRCUITS

ECE 212s

FACULTY OF ENGINEERING
AIN SHAMS UNIVERSITY

PROJECT REPORT

TEAM 35

Name	ID
Ammar Ahmed Wahidi	2200360
Ali Hany Zainhom	2201335
Hamza Basem Mohamed Ahmed	2200496
Omar Mohamed Hassan	2200294
Ammar Mohamed Abdelghany	2200707
Amr Ahmed Wahidi	2200429
Omar Ahmed Abd El-Kader	2200706
Youssef Emad Eldeen	2200536
Mohamed Elsayed Moustafa Hamouda	2200095
Anas Ayman Mohammed	2200325

SUBMITTED TO: PROF. SAMEH A. IBRAHIM

DATE OF SUBMISSION: 23TH MAY. 2025

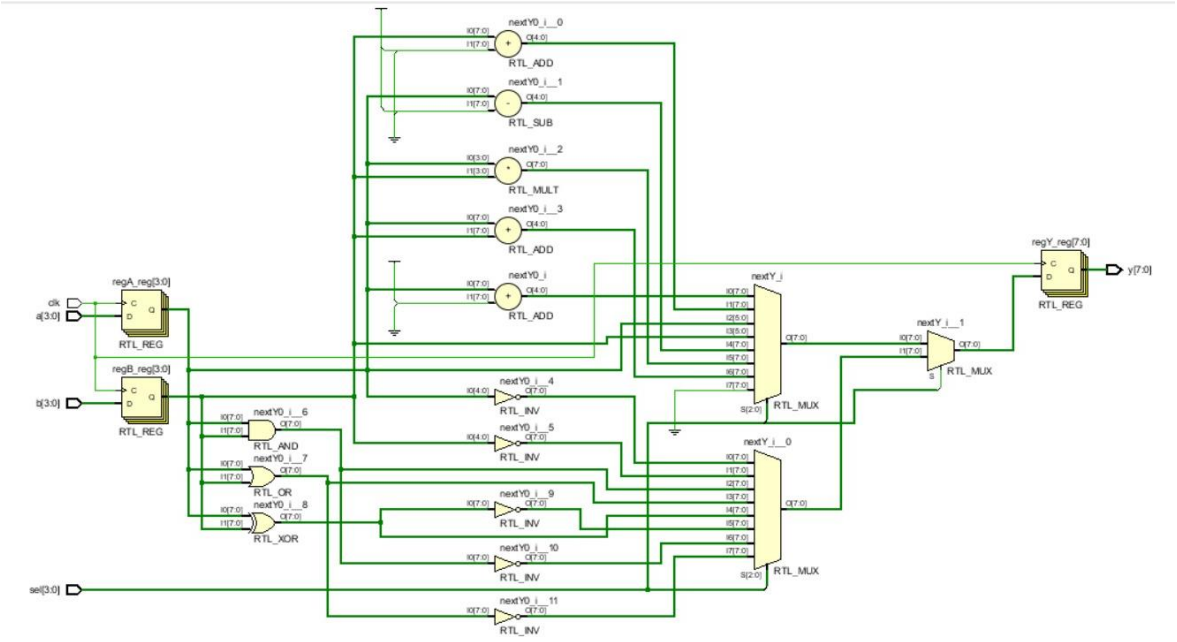
Bonus Contributed
FBGA
Fastest design (max frequency)

Verilog code

Code

```
module ALU (
    input signed [3:0] a,b,sel,
    input clk,
    output signed [7:0] y
);

    reg signed [3:0] regA;
    reg signed [3:0] regB;
    reg signed [7:0] regY;
    reg signed [7:0] nextY;

    always @(posedge clk)
    begin
        regY<=nextY;

    end
     always @(negedge clk)
    begin
        regA<=a;
        regB<=b;

    end

    //MUX
    always @(*) begin
      case (sel[3])
        0: begin
            case (sel[2:0])
                3'b000: nextY = regA + 1; // Increment A
                3'b001: nextY = regB + 1; // Increment B
                3'b010: nextY = regA; // Transfer A
                3'b011: nextY = regB; // Transfer B
                3'b100: nextY = regA - 1; // Decrement A
                3'b101: nextY = regA * regB; // Multiply A and B
                3'b110: nextY = regA + regB; // Add A and B
                3'b111: nextY = regA - regB;
                default: nextY = 8'b00000000;
            endcase
        end
        1:  begin
          case (sel[2:0])
            3'b000: nextY = ~regA;       // Complement a (1's complement)
            3'b001: nextY = ~regB;       // Complement b (1's complement)
            3'b010: nextY = regA & regB; // AND
            3'b011: nextY = regA | regB; // OR
            3'b100: nextY = regA ^ regB; // XOR
            3'b101: nextY = ~(regA ^ regB); // XNOR
            3'b110: nextY = ~(regA & regB); // NAND
            3'b111: nextY = ~(regA | regB); // NOR
            default: nextY = 8'b00000000;
          endcase
        end
        default: nextY = 8'b00000000;
      endcase
    end

    assign y=regY;

endmodule
```

schematics



Schematic of Verilog

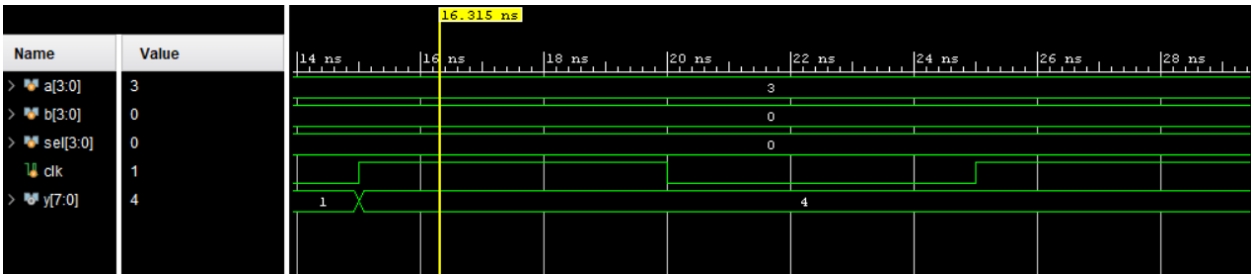
ALU Testbench Verilog Code

```
`timescale 1ns / 1ps

module ALU_tb;

    reg signed [3:0] a, b;
    reg [3:0] sel;
    reg clk;
    wire signed [7:0] y;

    // Instantiate ALU
    ALU uut (
        .a(a),
        .b(b),
        .sel(sel),
        .clk(clk),
        .y(y)
    );

    // Generate clock: 10ns period
    always #5 clk = ~clk;



    initial begin
        // Initialize
        clk = 0;
        a = 0;
        b = 0;
        sel = 4'b0000;
        #10;

        // Test Arithmetic Operations (sel[3] = 0)
        // Test Increment A: A = 3 -> Y = 4
        a = 4'd3; b = 4'd0;
        sel = 4'b0000;
        #20;  // Wait 2 clock cycles

        // Test Increment B: B = 5 -> Y = 6
        a = 4'd0; b = 4'd5;
        sel = 4'b0001;
        #20;

        // Test Transfer A: A = -2 -> Y = -2
        a = -4'sd2; b = 4'd0;
        sel = 4'b0010;
        #20;

        // Test Transfer B: B = -4 -> Y = -4
        a = 4'd0; b = -4'sd4;
        sel = 4'b0011;
        #20;

        // Test Decrement A: A = 7 -> Y = 6
        a = 4'd7; b = 4'd0;
        sel = 4'b0100;
        #20;
```

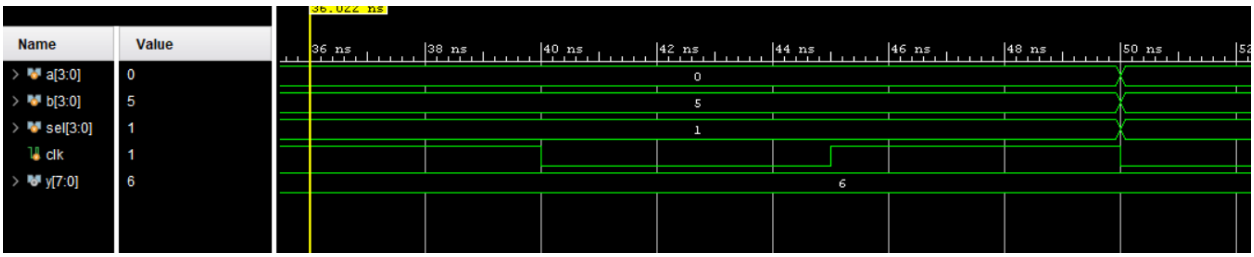
```
        // Test Multiply A * B: A = 3, B = -2 -> Y = -6
        a = 4'd3; b = -4'sd2;
        sel = 4'b0101;
        #20;

        // Test Add A + B: A = 3, B = 2 -> Y = 5
        a = 4'd3; b = 4'd2;
        sel = 4'b0110;
        #20;

        sel=4'b0111;
        #20;

        // Test Logical Operations (sel[3] = 1)
        // Test Complement A: A = 5 -> Y = -6 (1's complement)
        a = 4'd5; b = 4'd0;
        sel = 4'b1000;
        #20;

        // Test Complement B: B = 3 -> Y = -4 (1's complement)
        a = 4'd0; b = 4'd3;
        sel = 4'b1001;
        #20;

        // Test AND: A = 5, B = 3 -> Y = 1
        a = 4'd5; b = 4'd3;
        sel = 4'b1010;
        #20;

        // Test OR: A = 5, B = 3 -> Y = 7
        a = 4'd5; b = 4'd3;
        sel = 4'b1011;
        #20;

        // Test XOR: A = 5, B = 3 -> Y = 6
        a = 4'd5; b = 4'd3;
        sel = 4'b1100;
        #20;

        // Test XNOR: A = 5, B = 3 -> Y = -7 (1's complement of XOR)
        a = 4'd5; b = 4'd3;
        sel = 4'b1101;
        #20;

        // Test NAND: A = 5, B = 3 -> Y = -2 (1's complement of AND)
        a = 4'd5; b = 4'd3;
        sel = 4'b1110;
        #20;

        // Test NOR: A = 5, B = 3 -> Y = -8 (1's complement of OR)
        a = 4'd5; b = 4'd3;
        sel = 4'b1111;
        #20;

        // Stop simulation
        #10 $finish;
    end

endmodule
```

Arithmetic unit

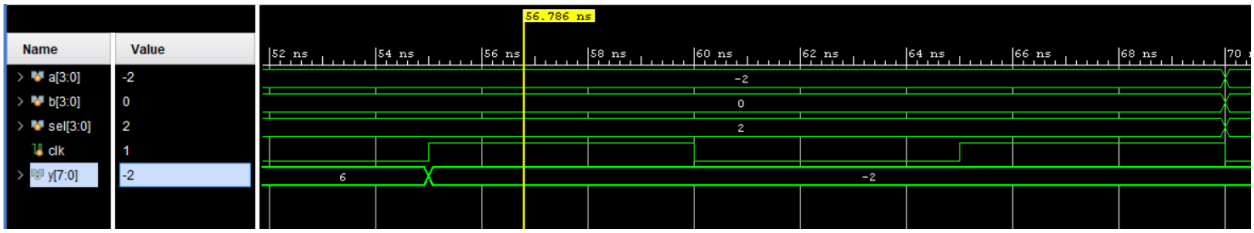
Increment a



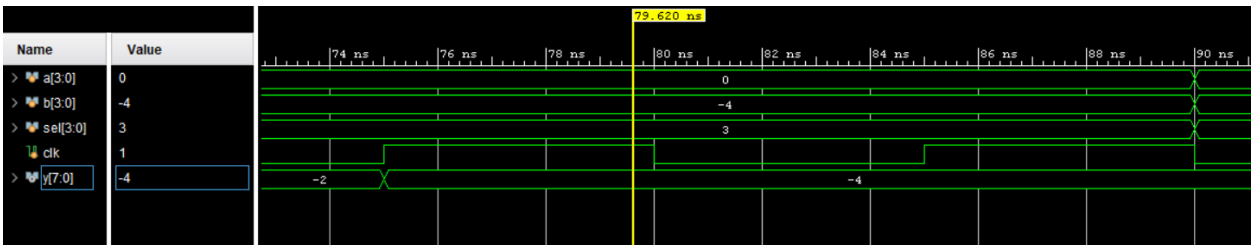
Increment b



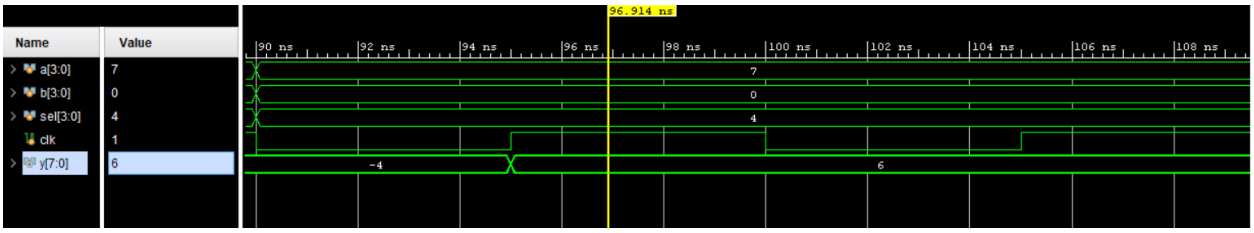
Transfer a



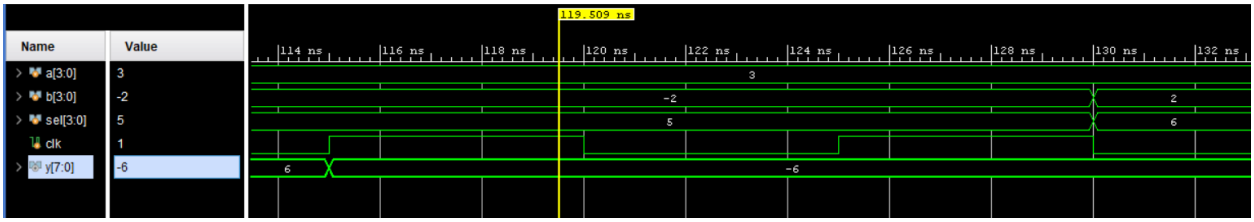
Transfer b



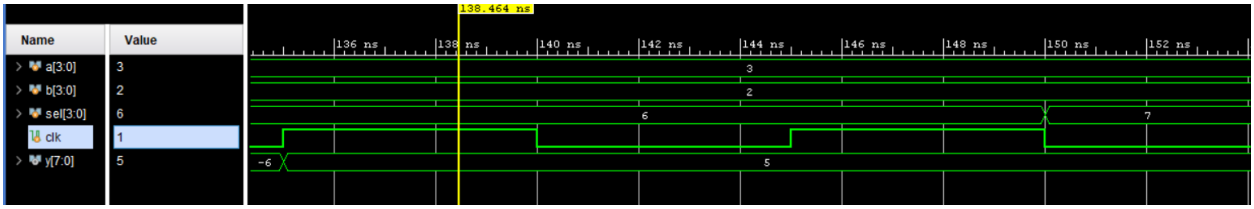
Decrement a



Multiply a and b



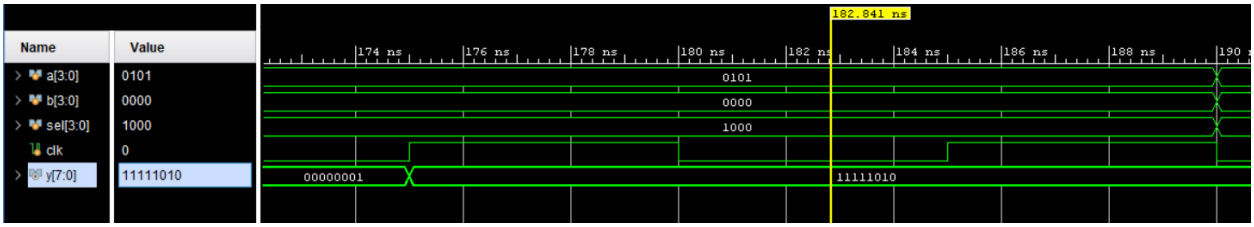
Add a and b



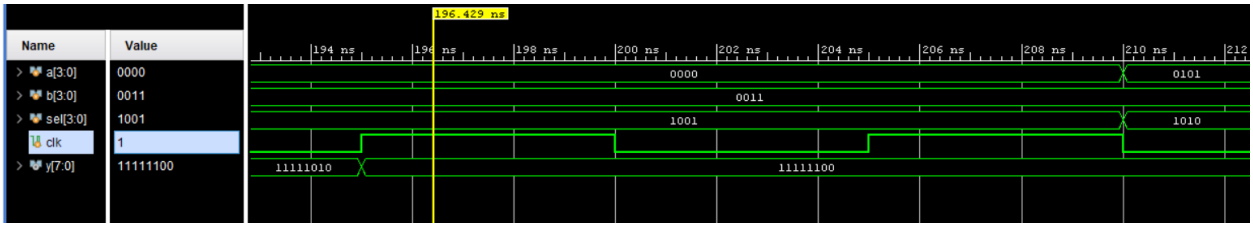
Subtract a and b

Logical unit

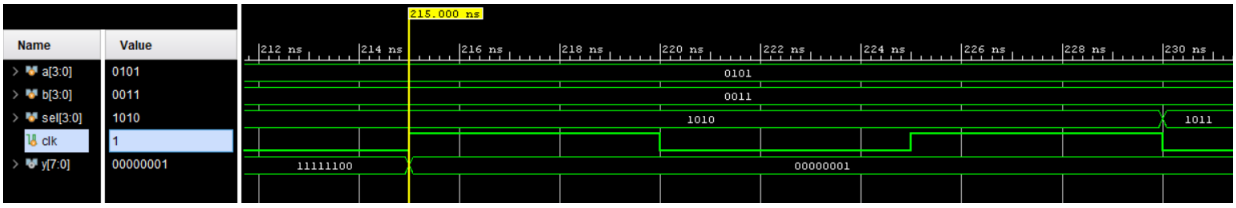
Complement a



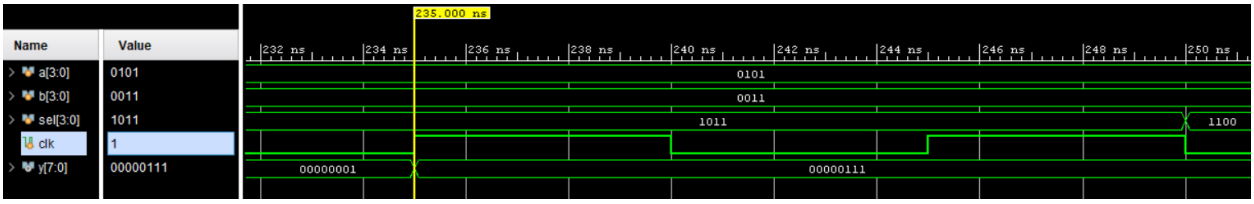
Complement b



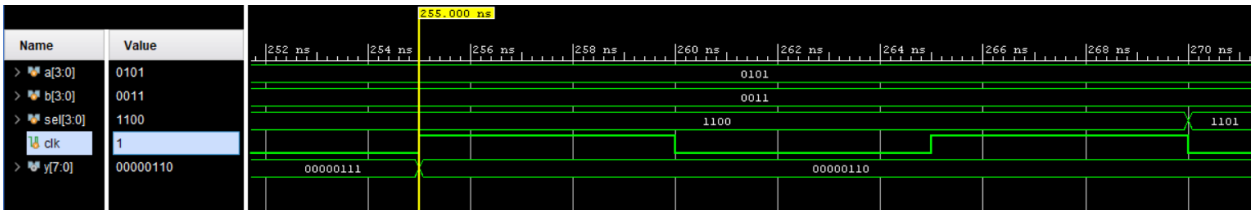
AND



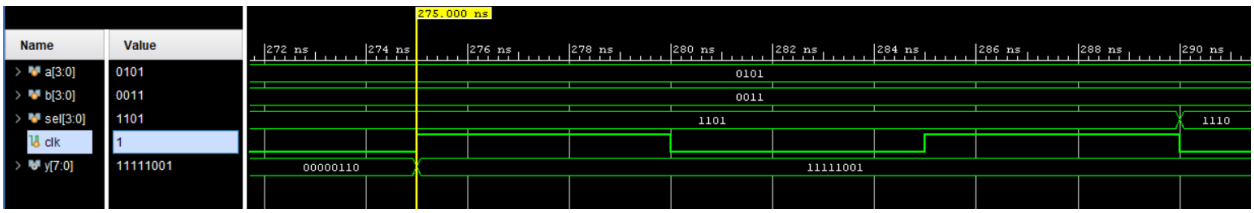
OR



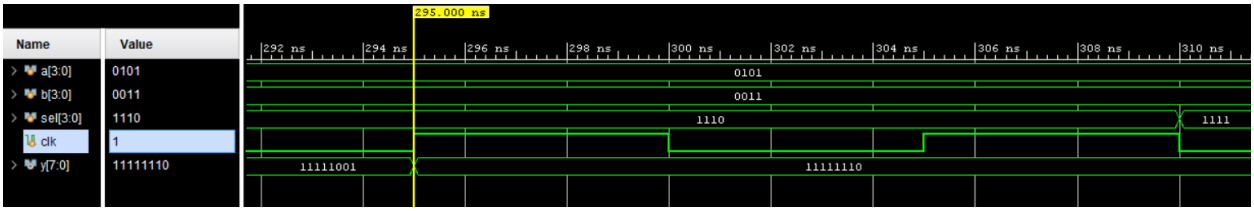
XOR



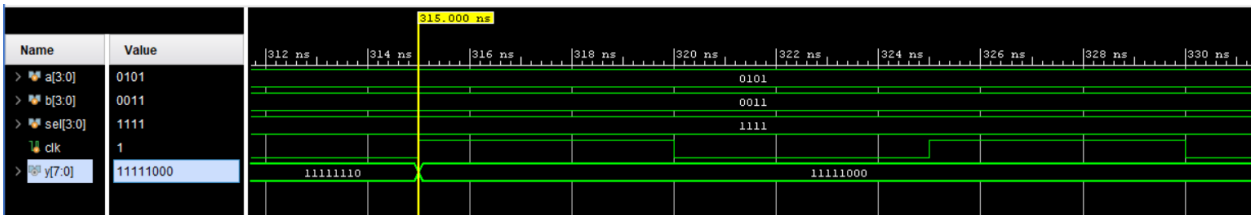
XNOR



NAND

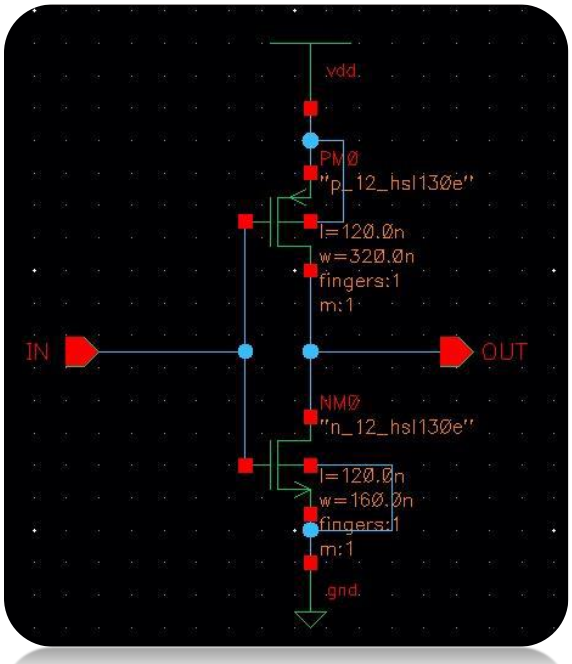


NOR

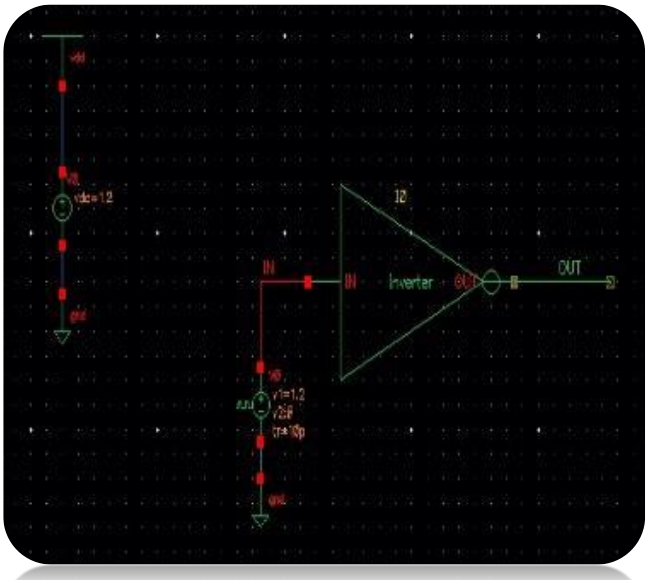


Transistor-level/Block-diagram schematics

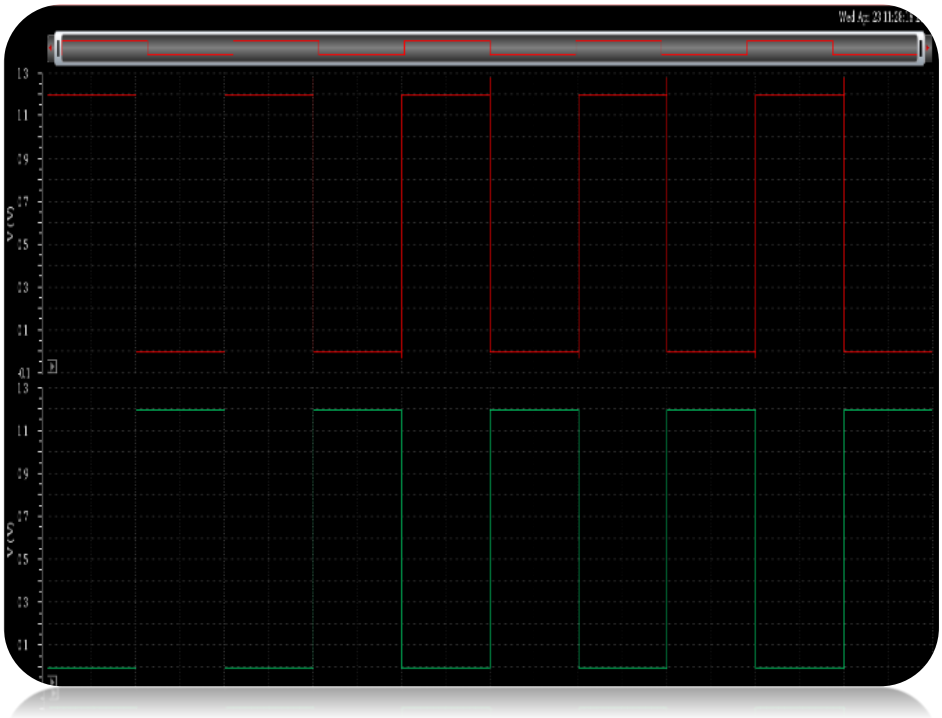
Inverter gate



Schematic

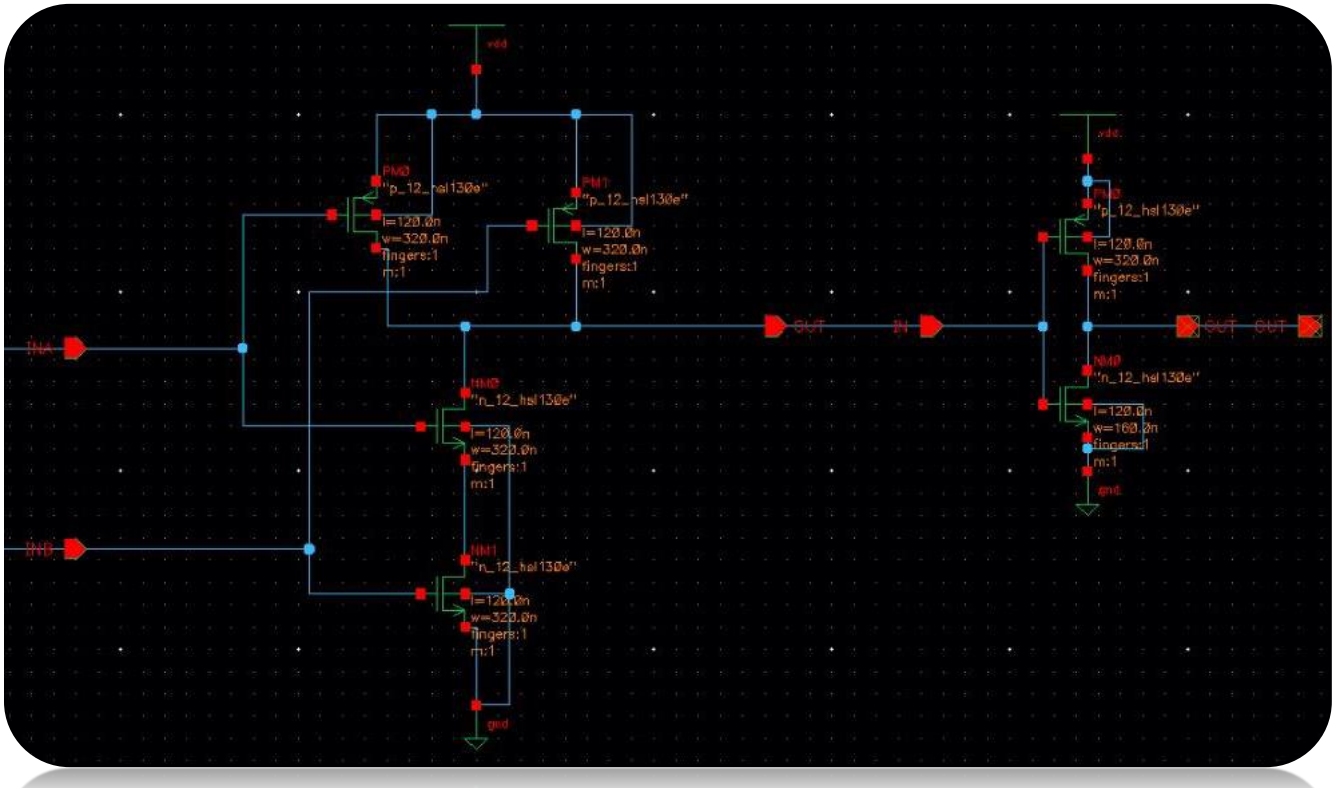


Symbol

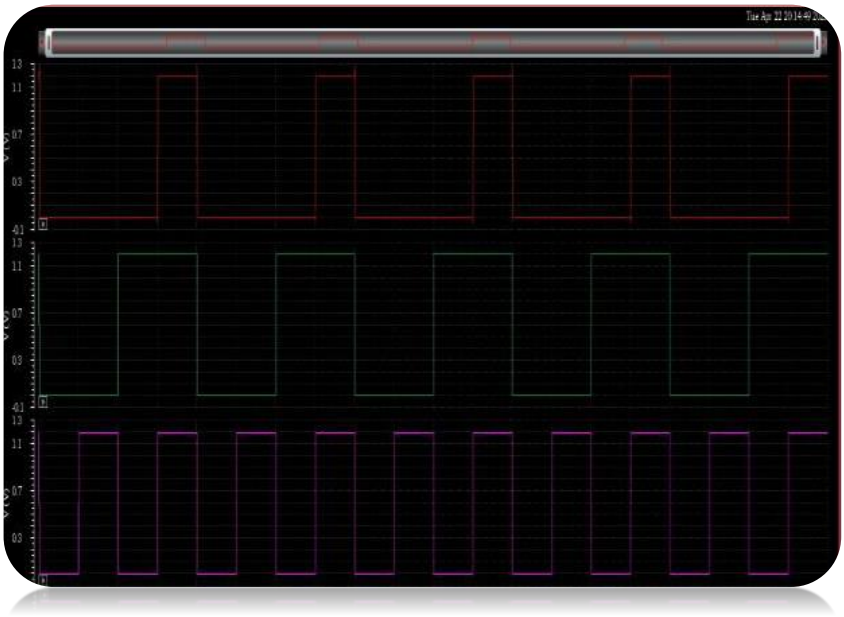


Testbench

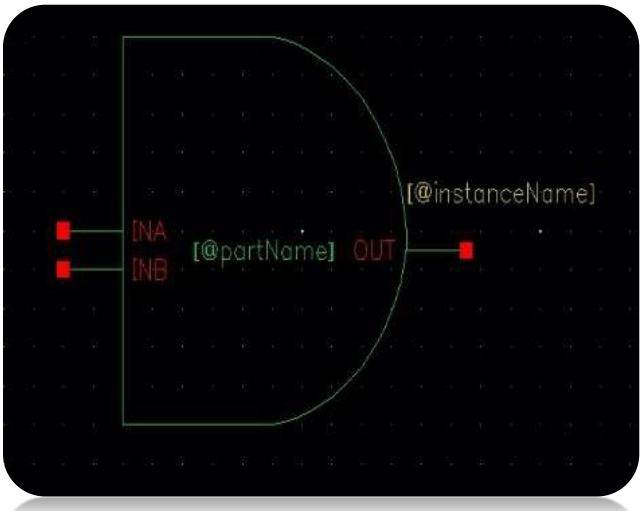
AND gate



Schematic

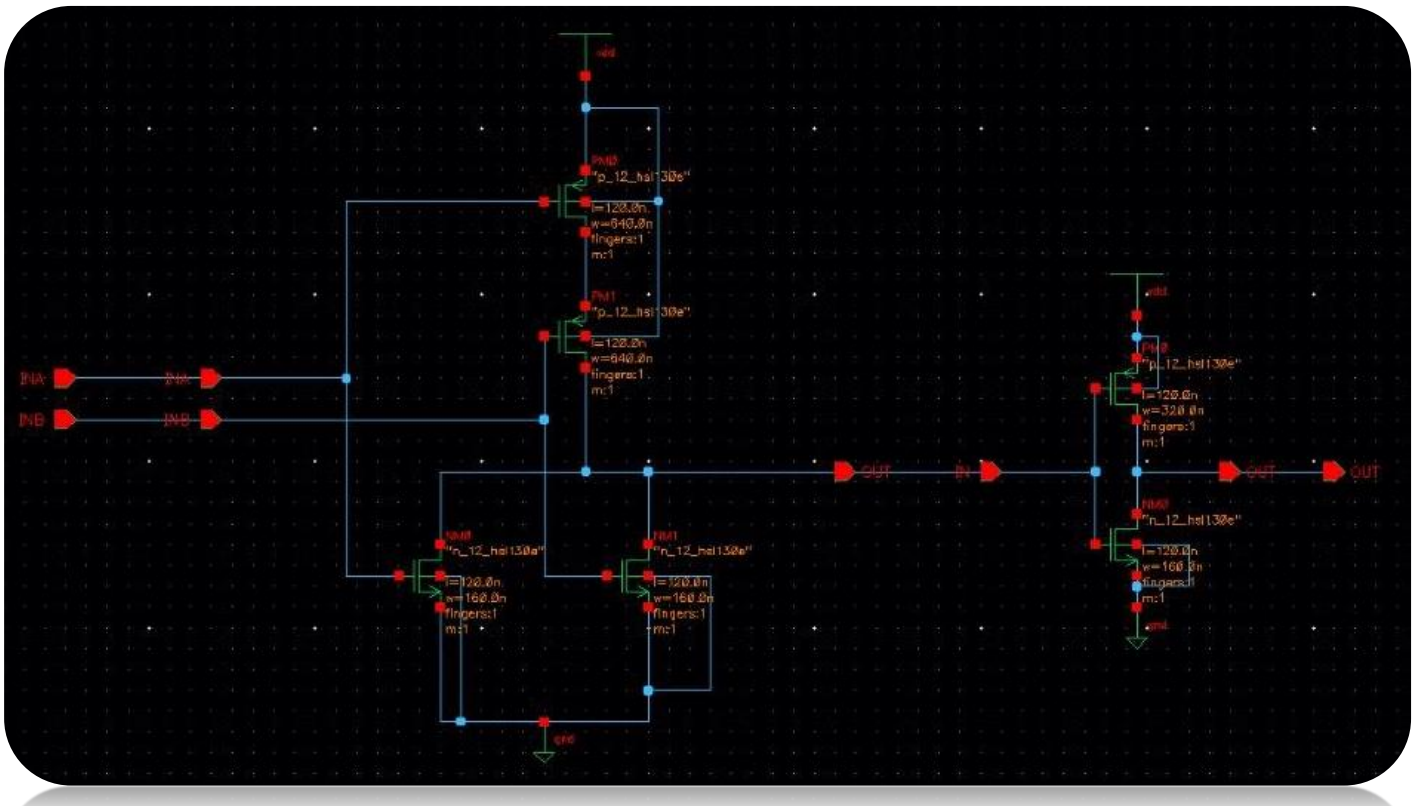


Testbench

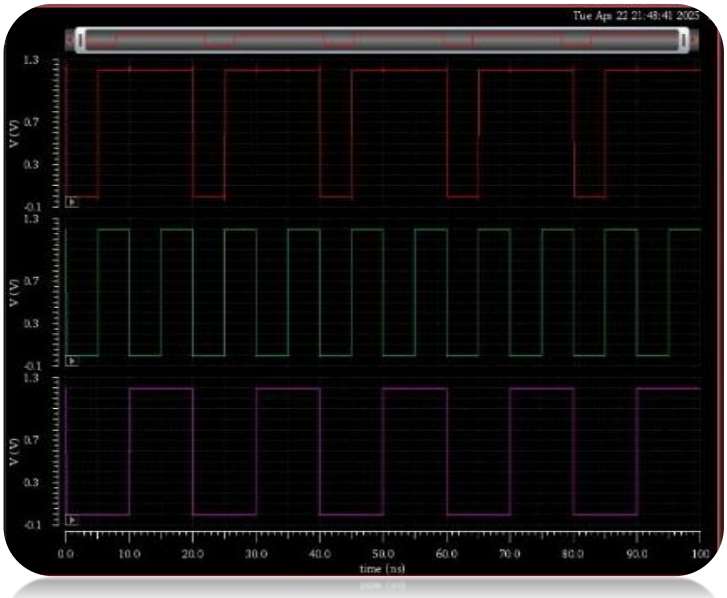


Symbol

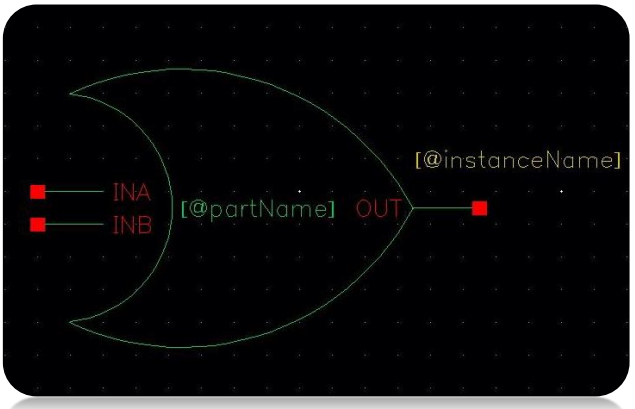
OR gate



Schematic

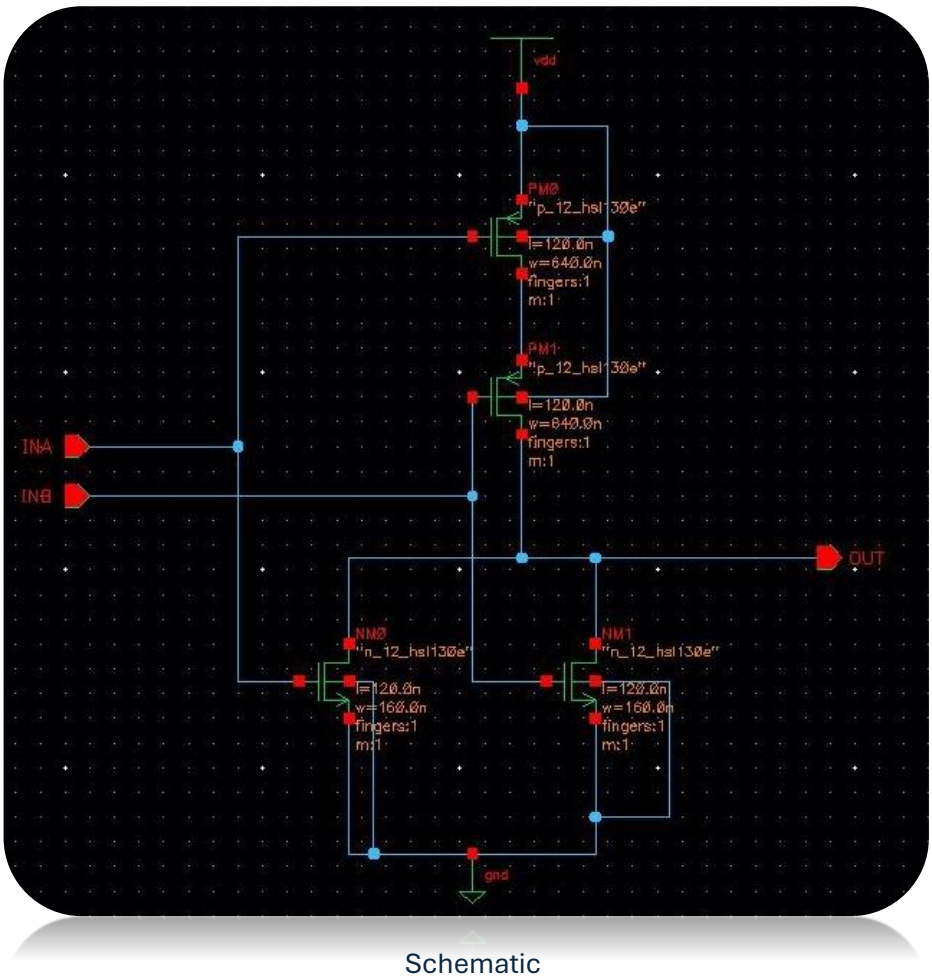


Testbench

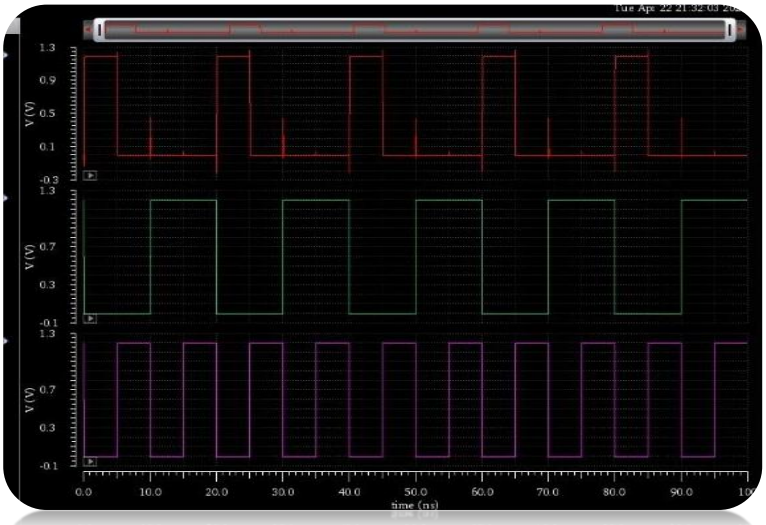


Symbol

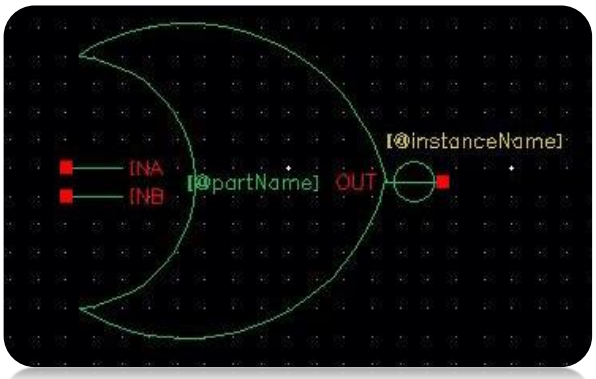
NOR gate



Schematic

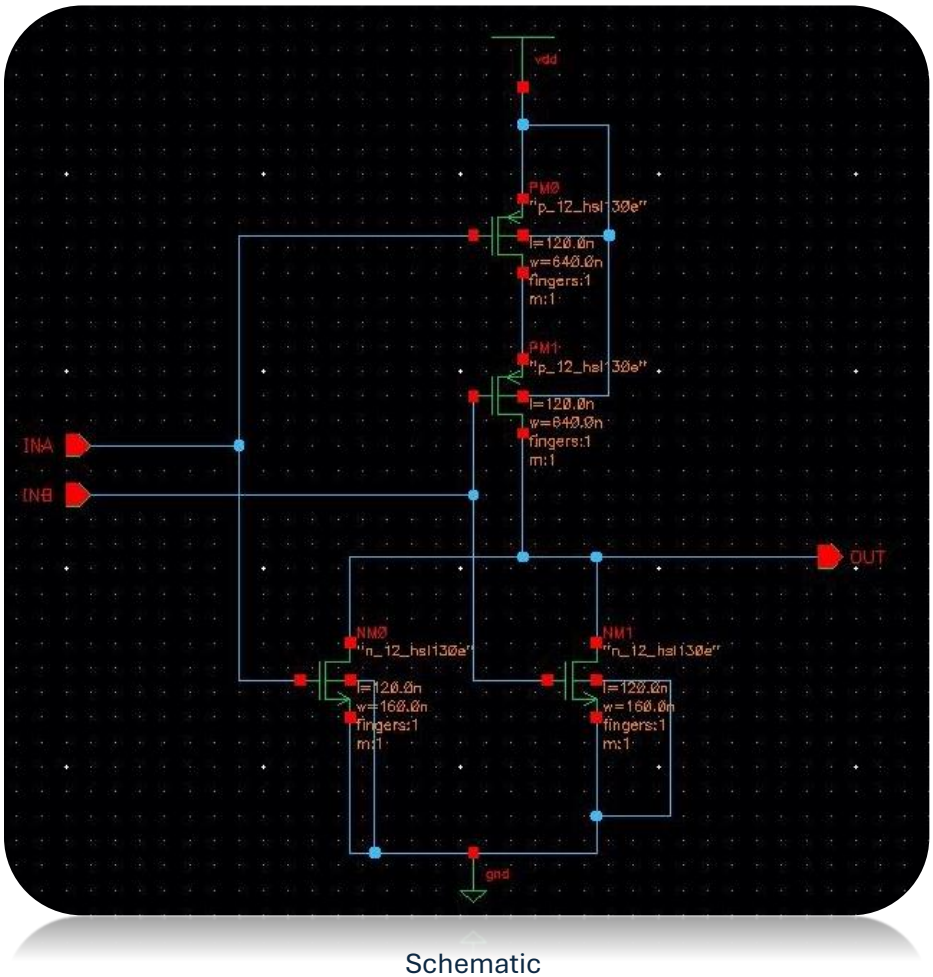


Testbench

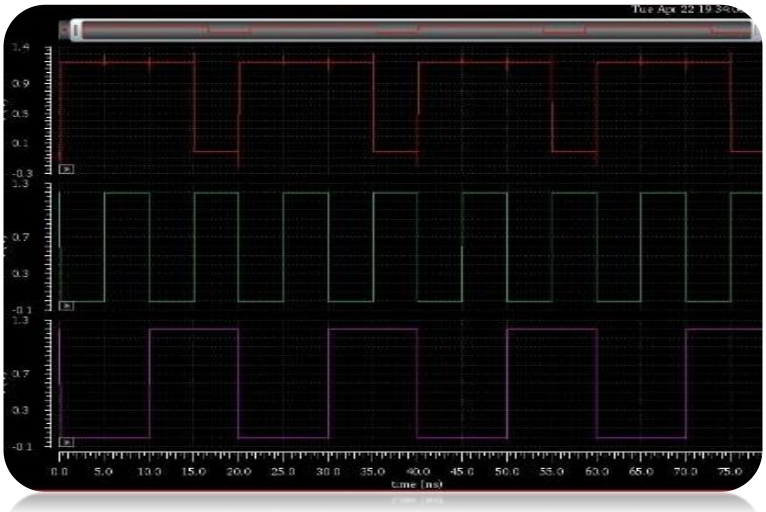


Symbol

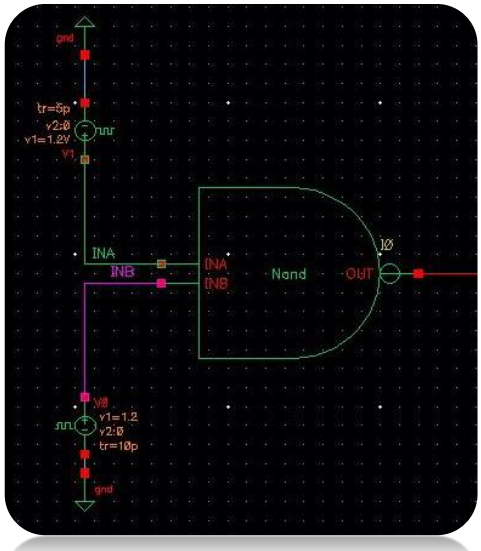
NAND gate



Schematic

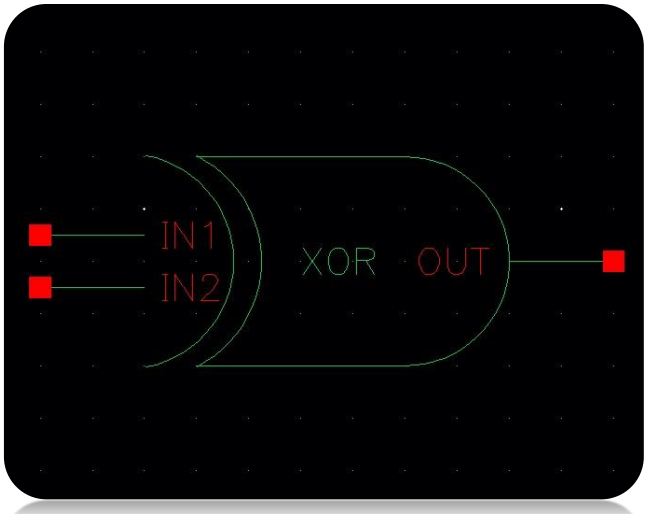


Testbench

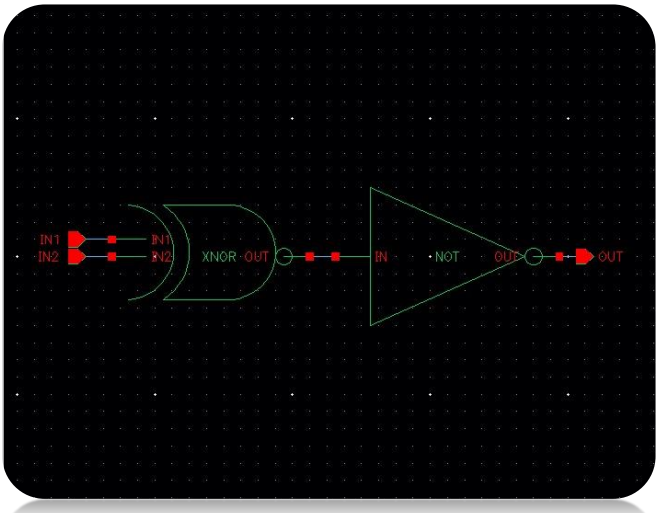


Symbol

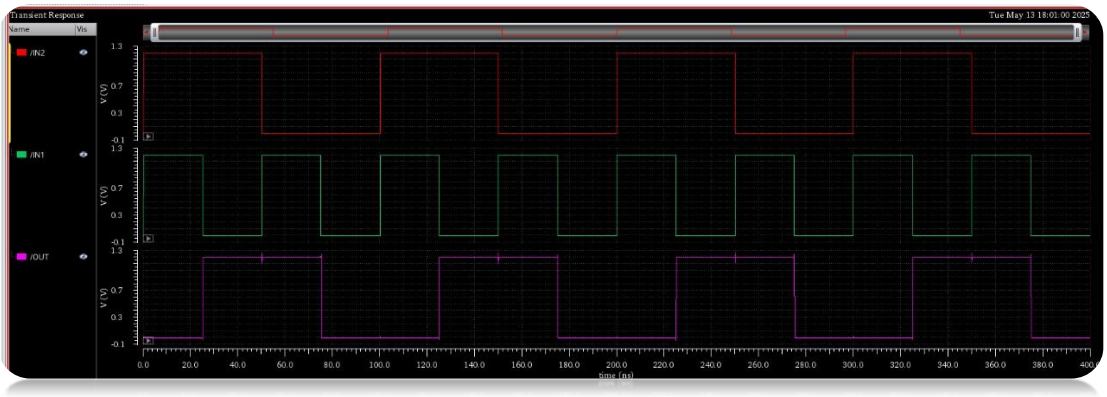
XOR gate



Symbol

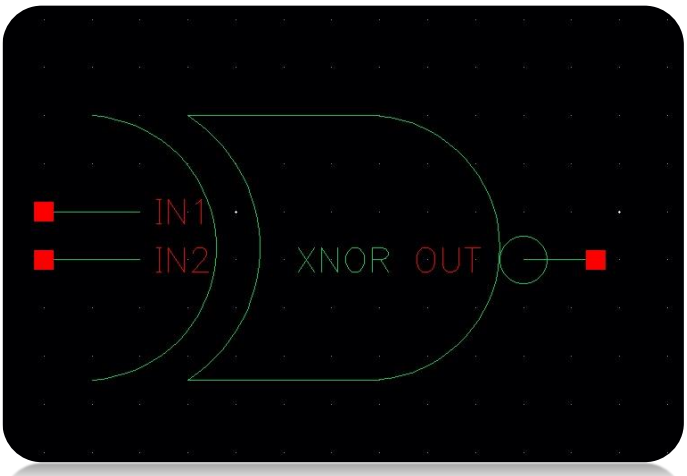


Schematic

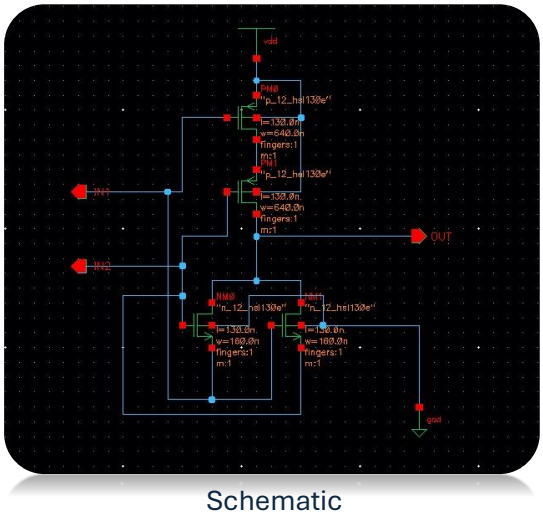


Testbench

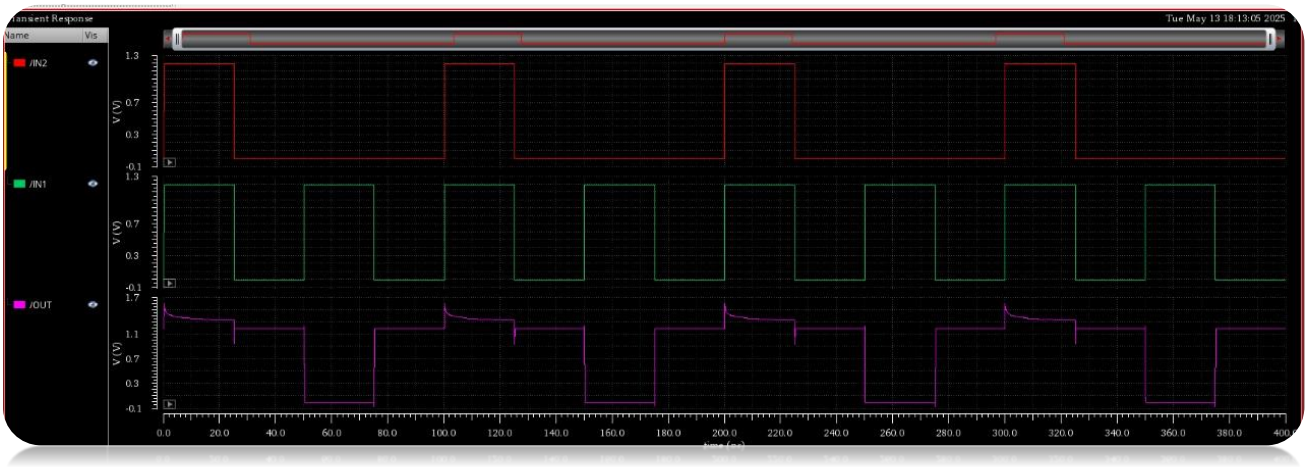
XNOR gate



Symbol



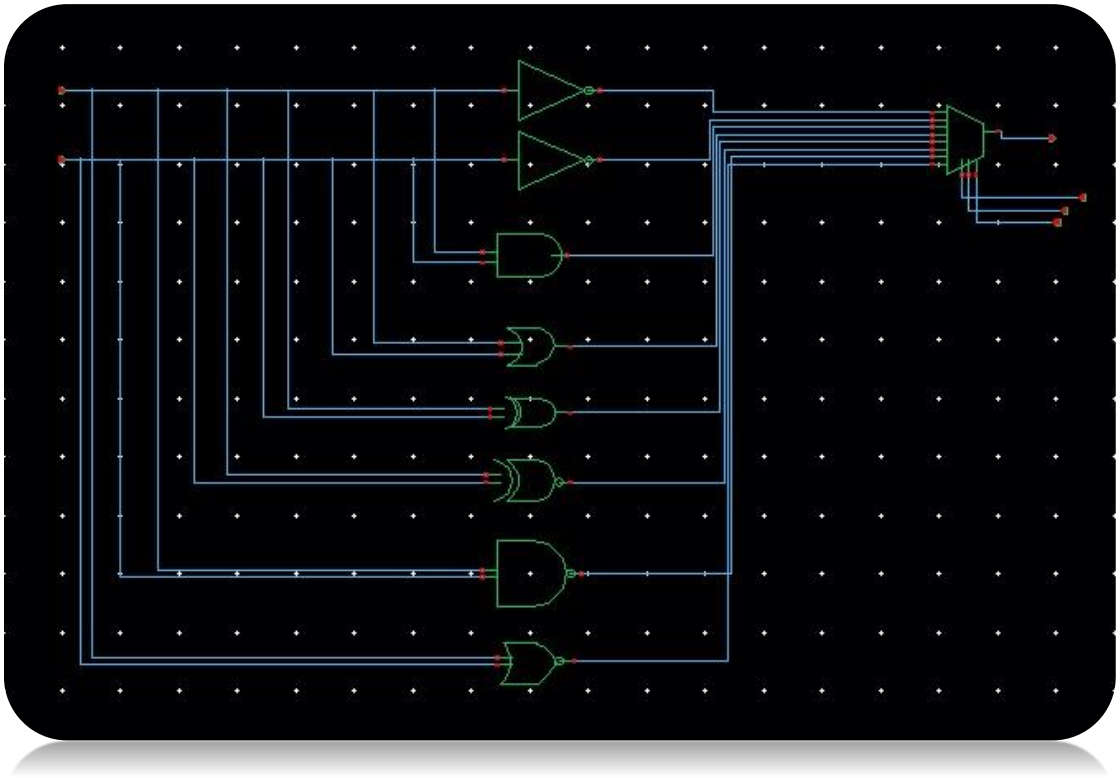
Schematic



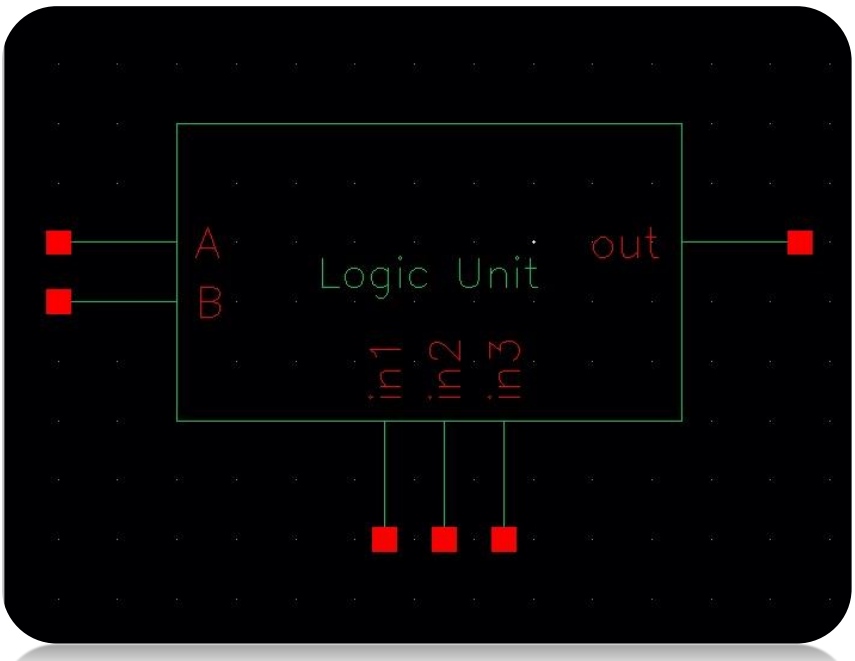
Testbench

Logical unit

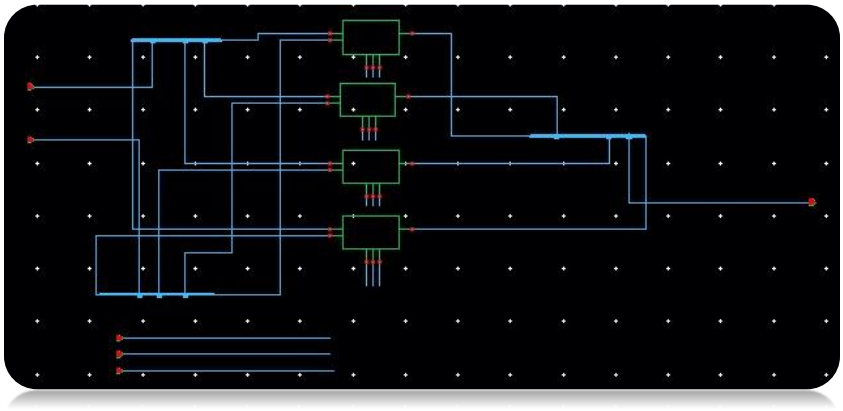
Constructing 1 bit logic unit:



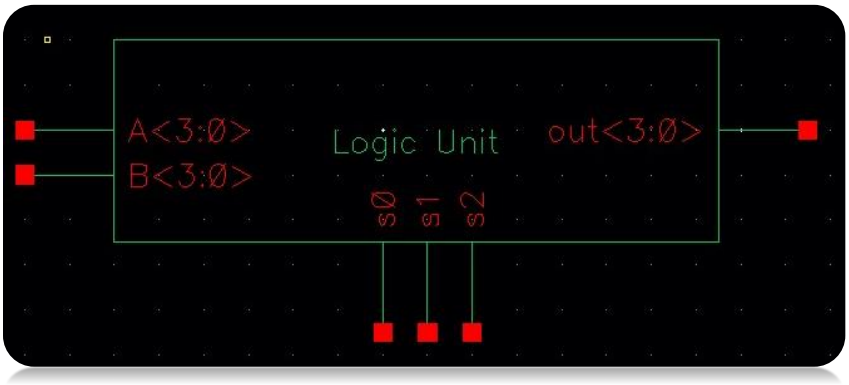
1 bit logic Symbol:



Constructing four bit logic unit from the one bit:

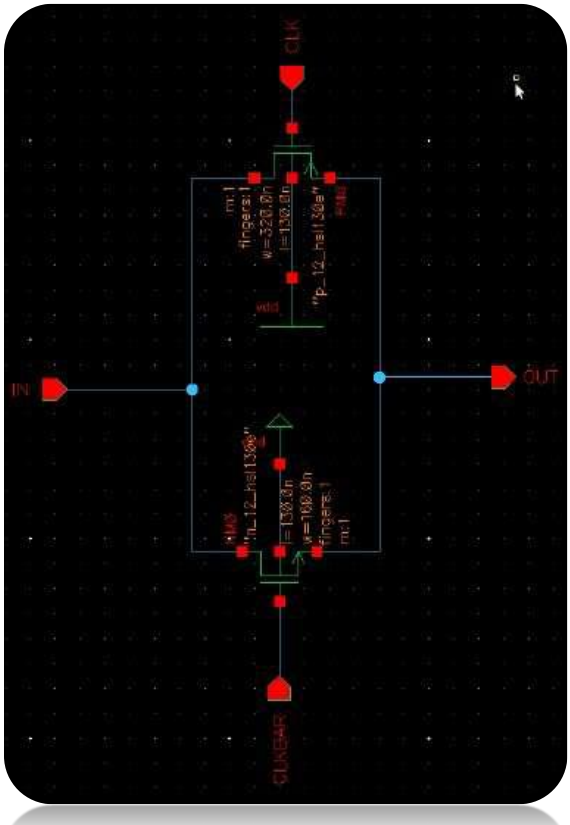


Four bit logic unit symbol:

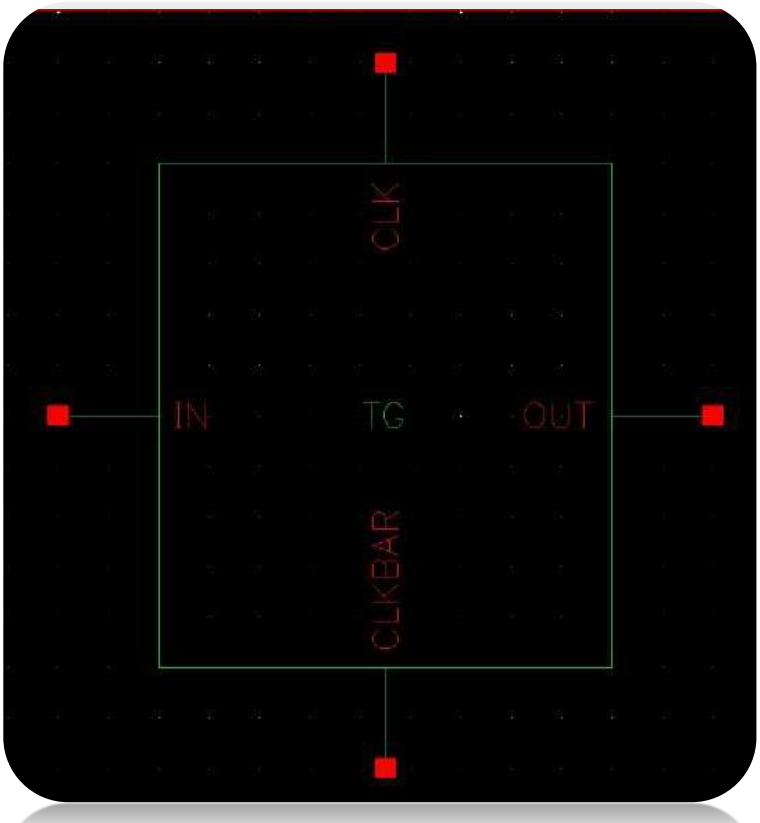


Output Waveform:

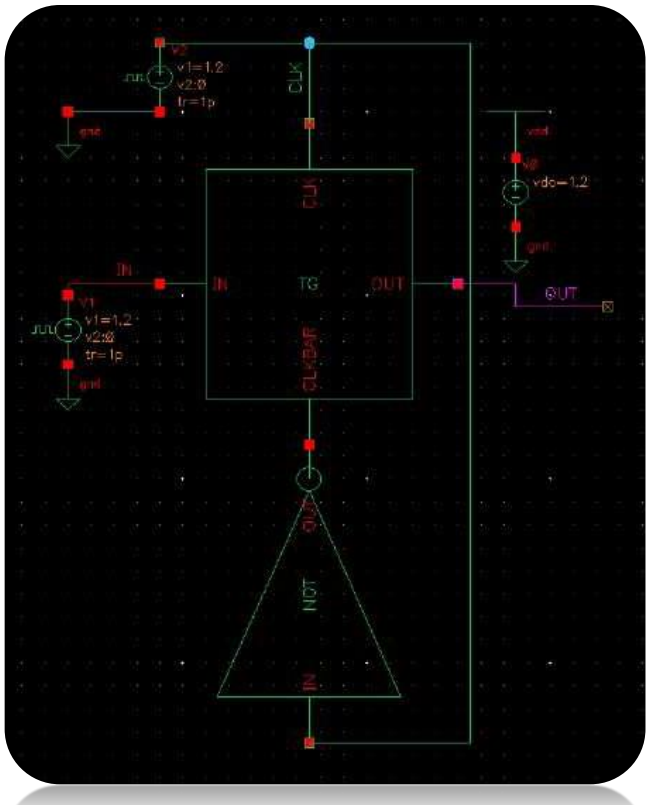
TG



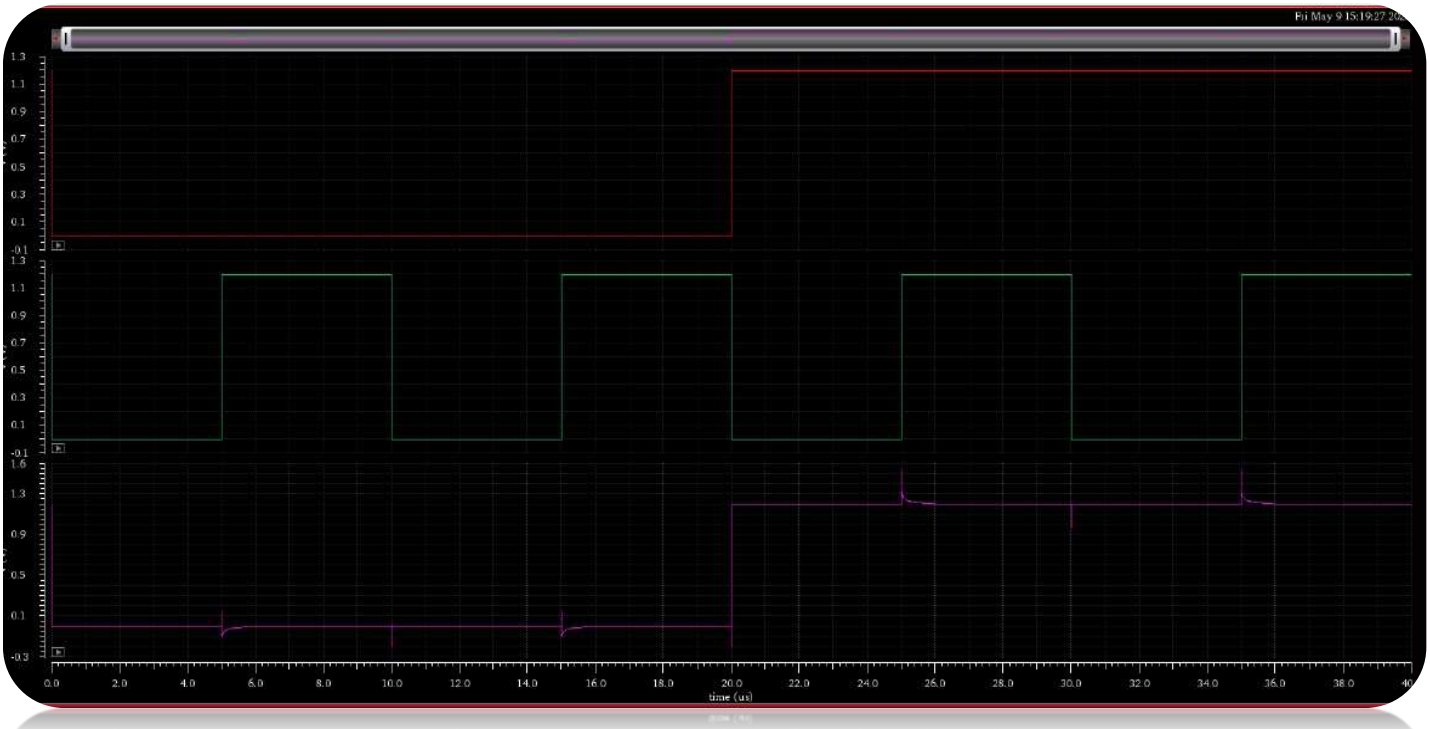
Schematic



Sybmol

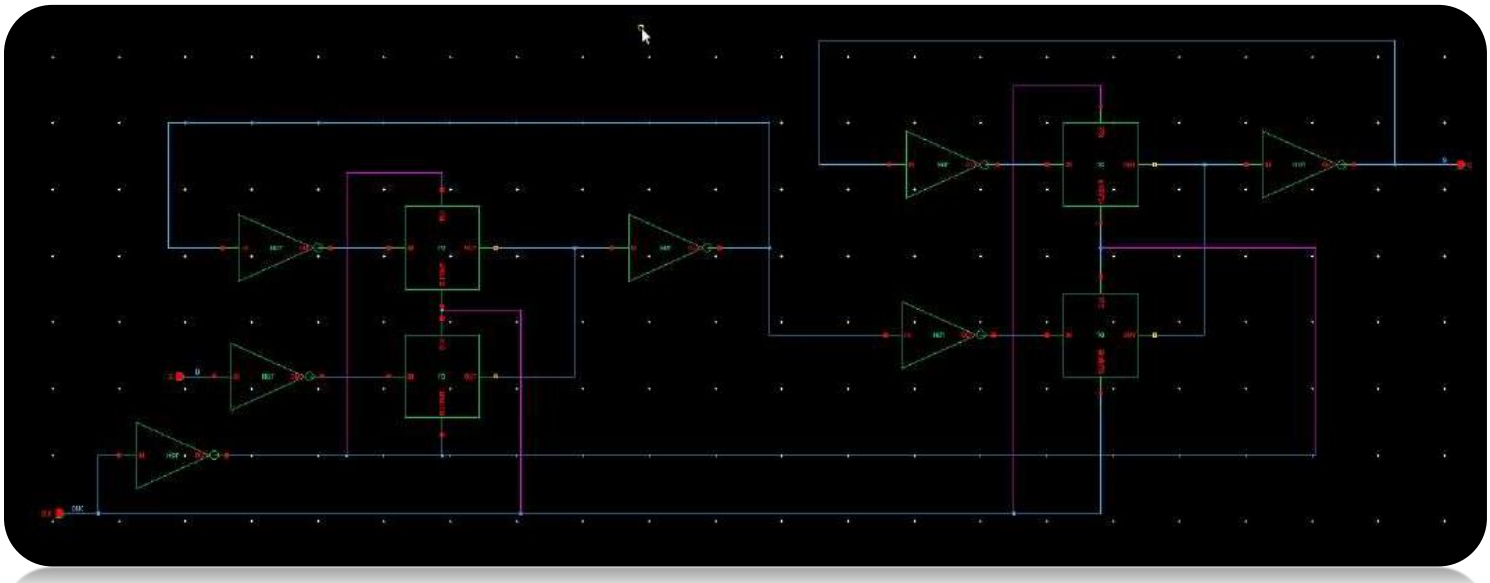


Testbench

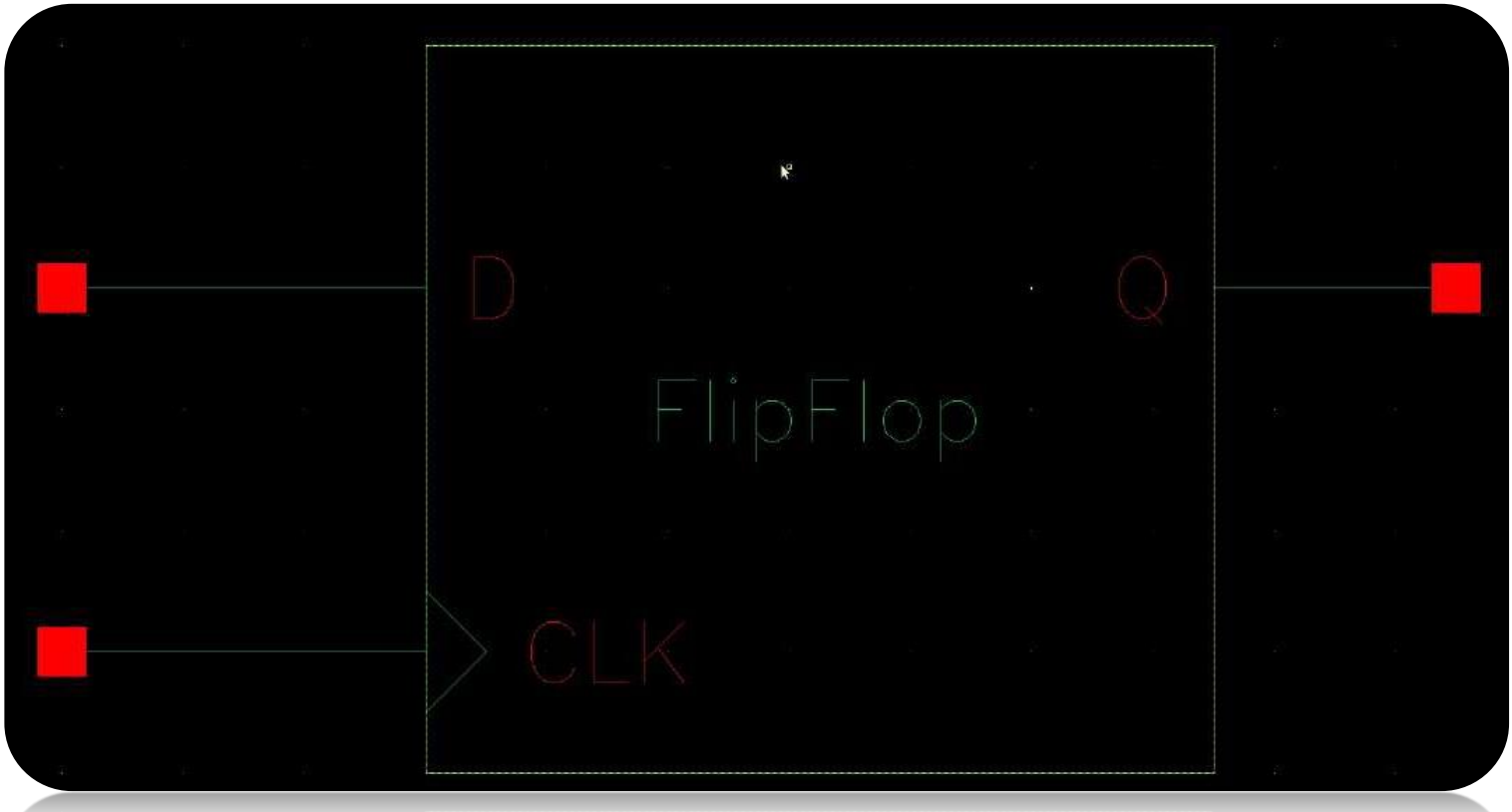


Wavefrom

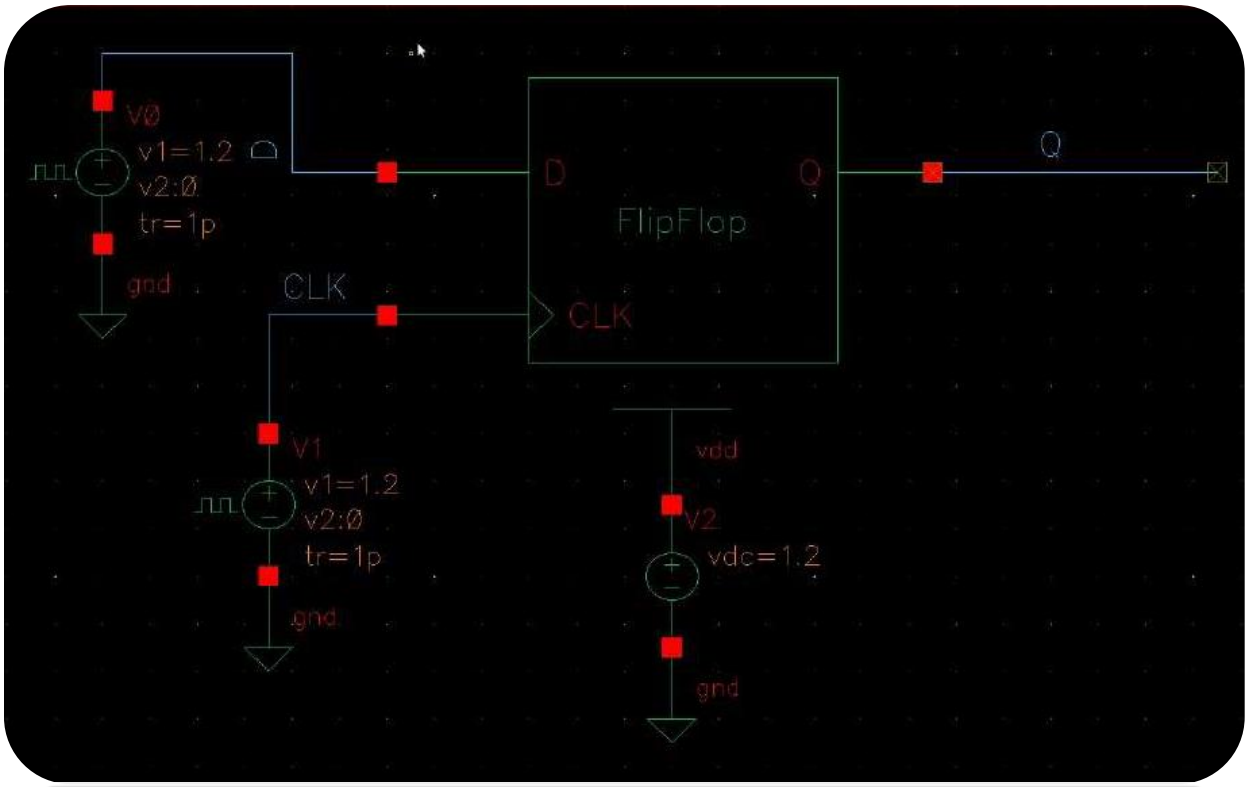
Filb-Flop



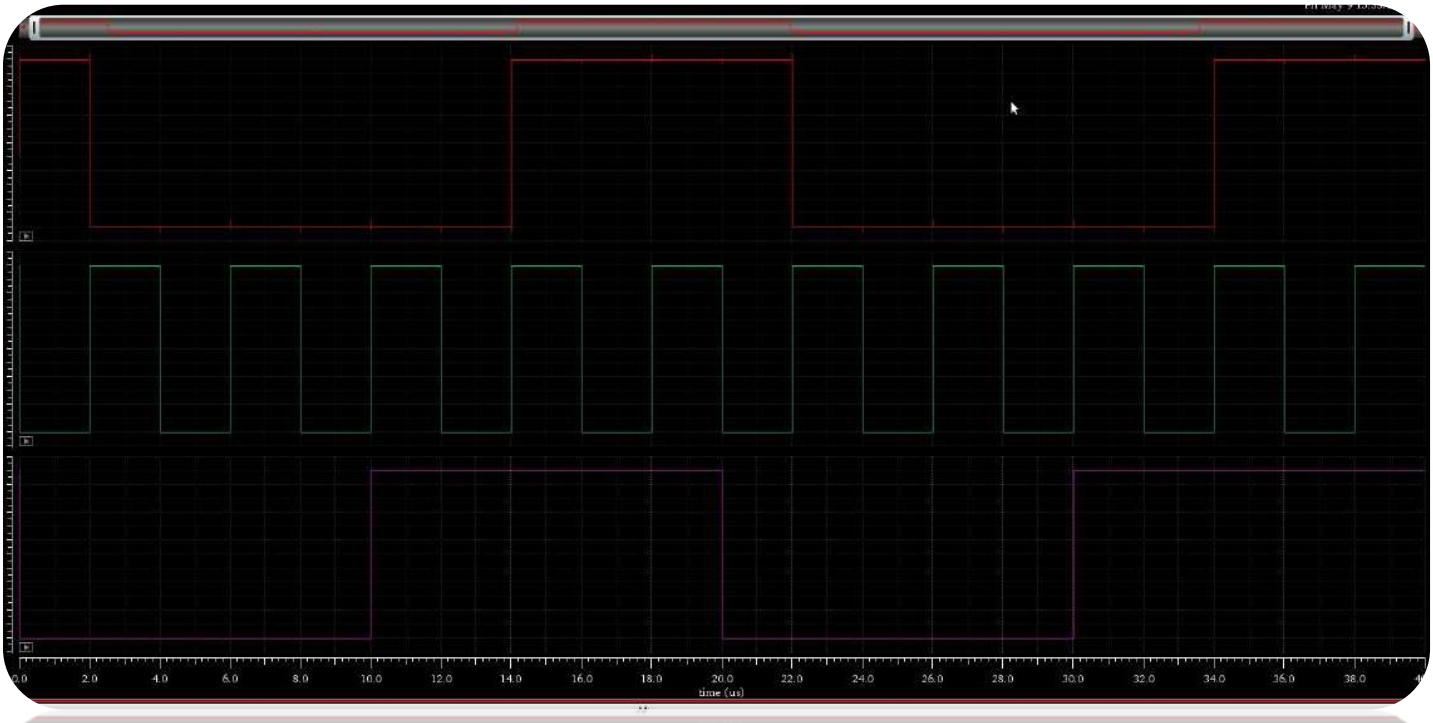
Schematic



Sybmol



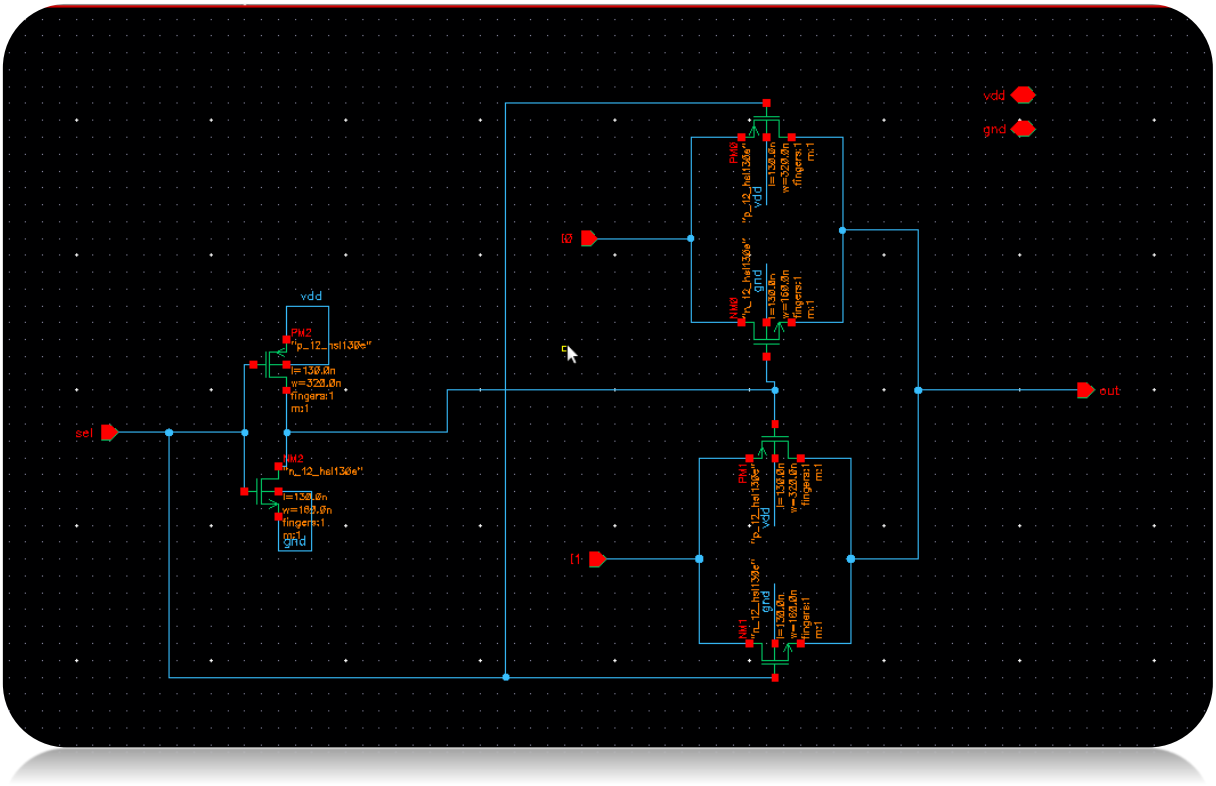
Testbench



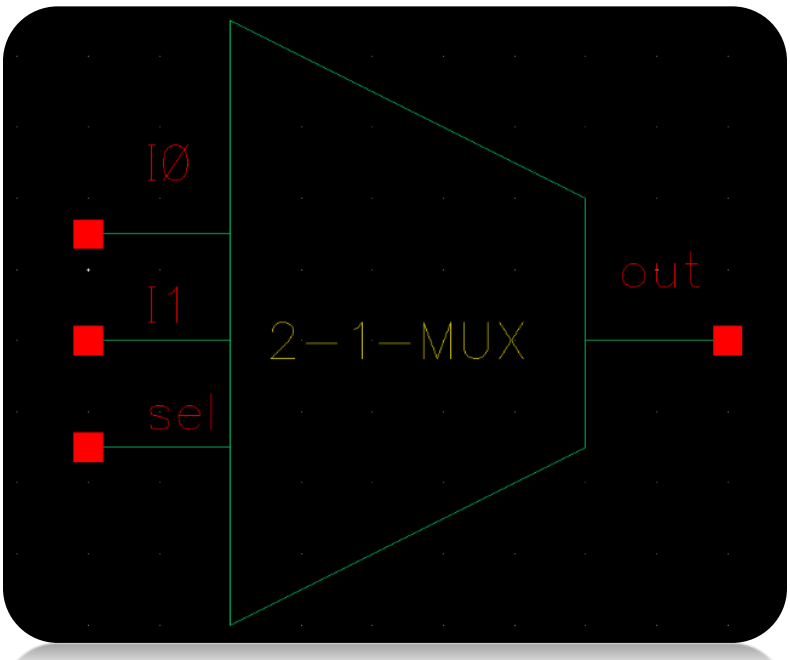
Wavefrom

MUX 2x1

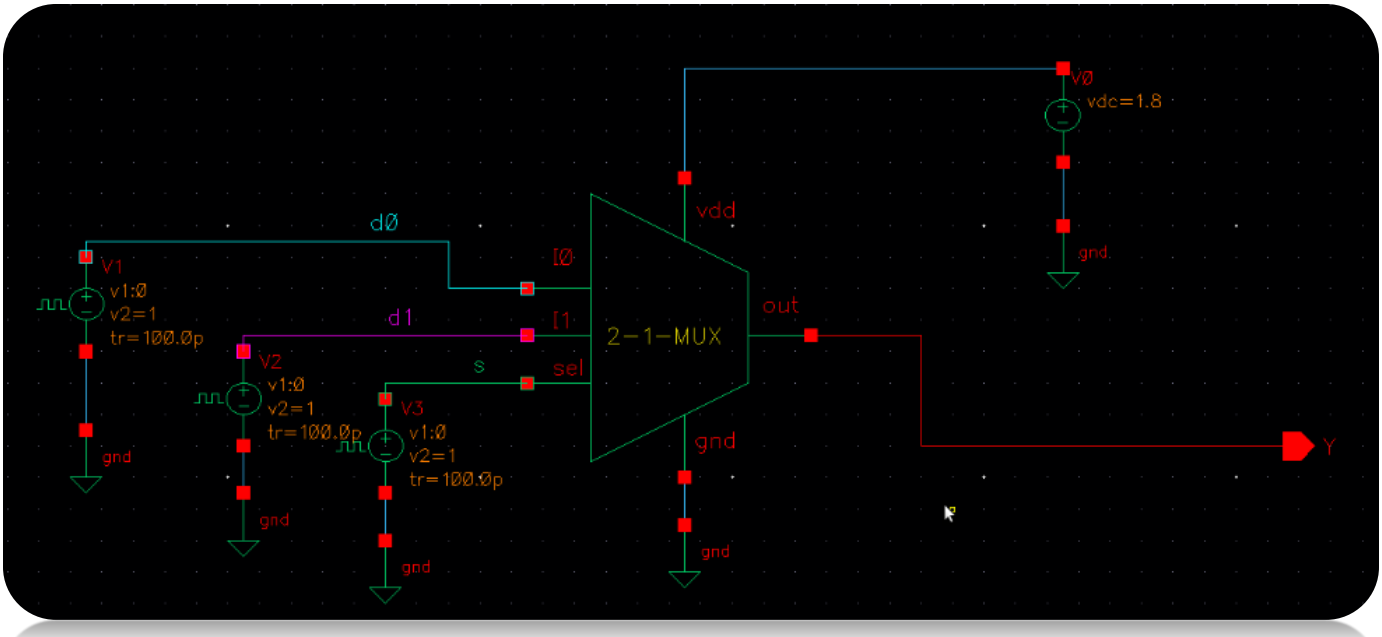
Schematic



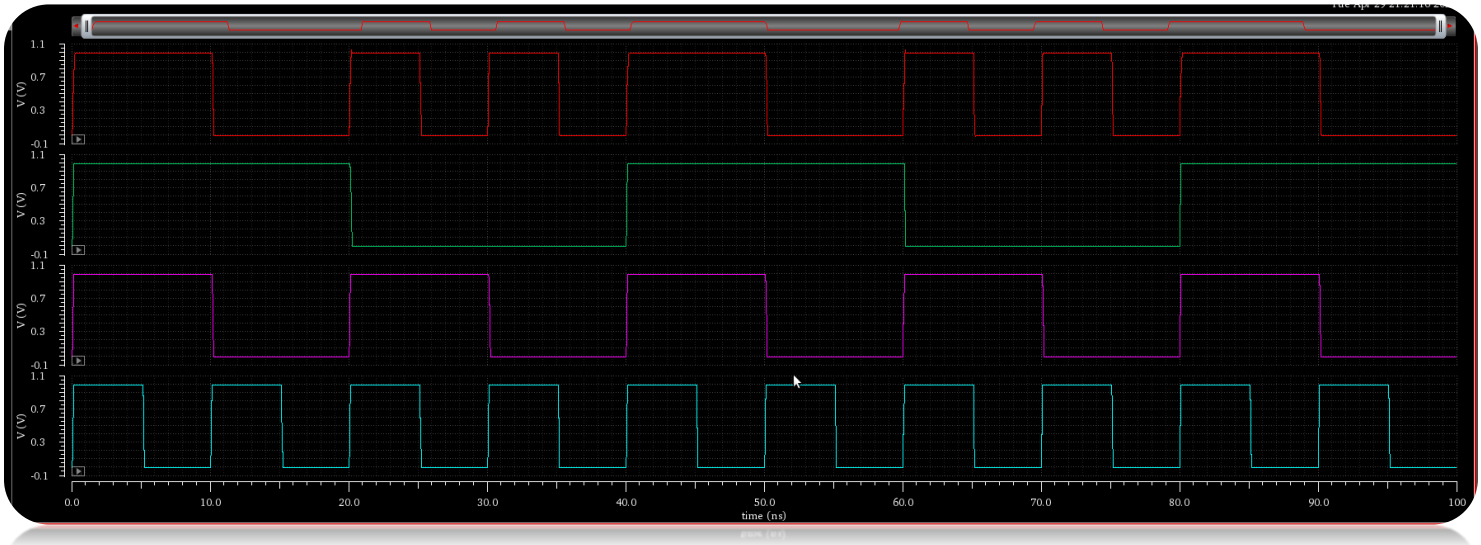
Symbol



Testbench Cell

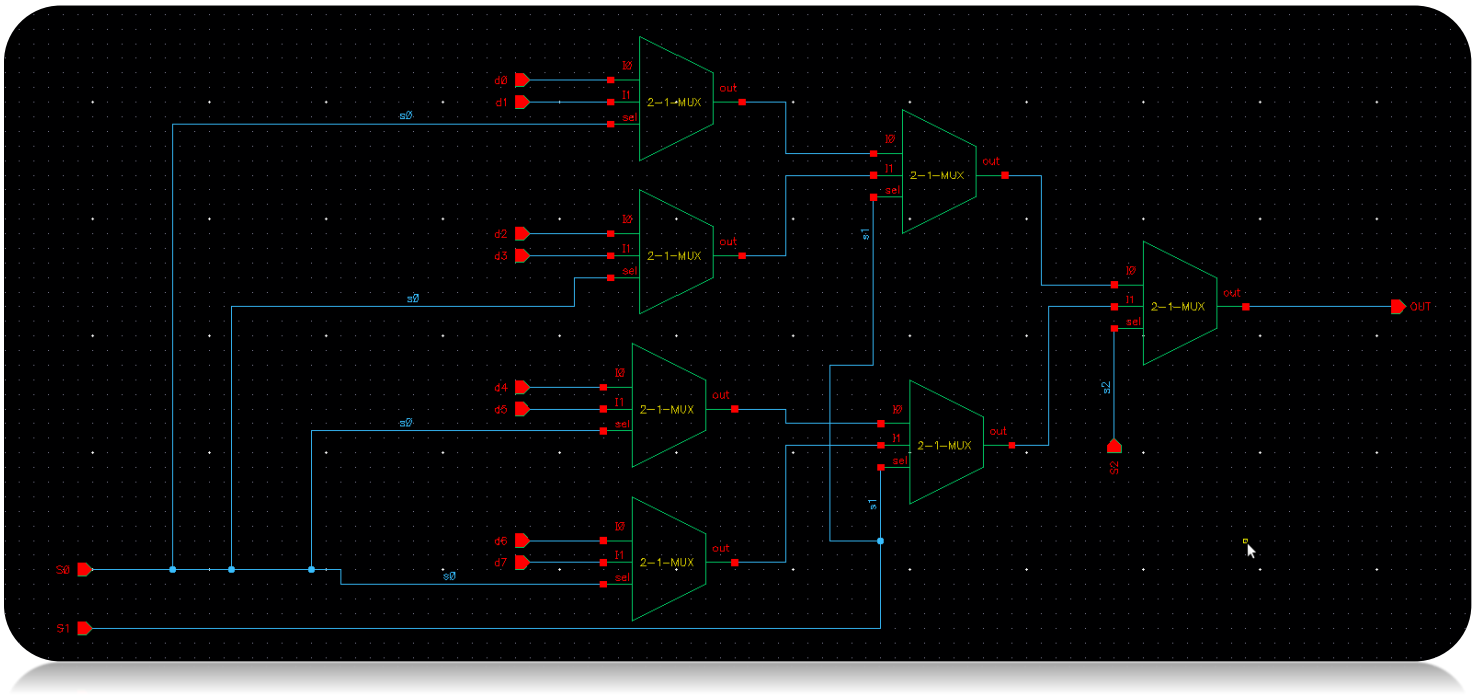


Output Waveform

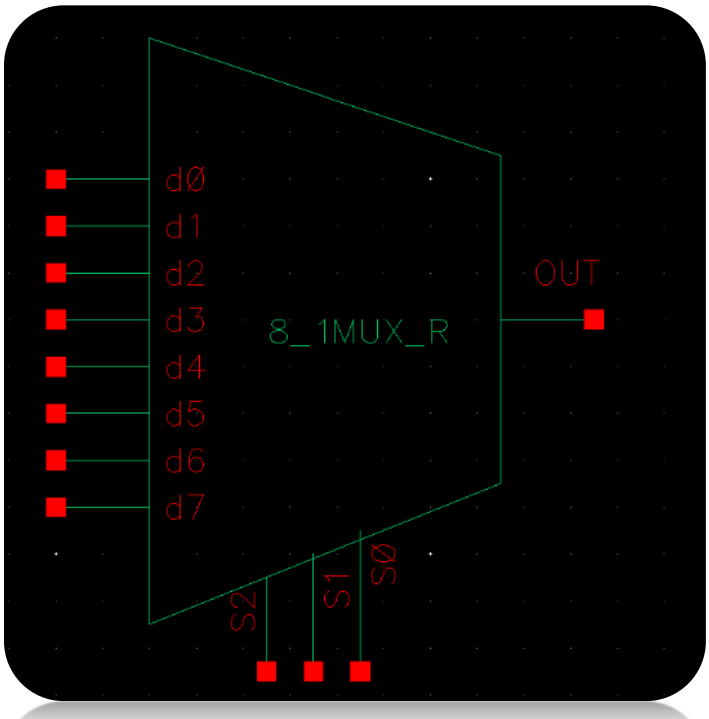


MUX 8x1

Schematic of 8-1 mux

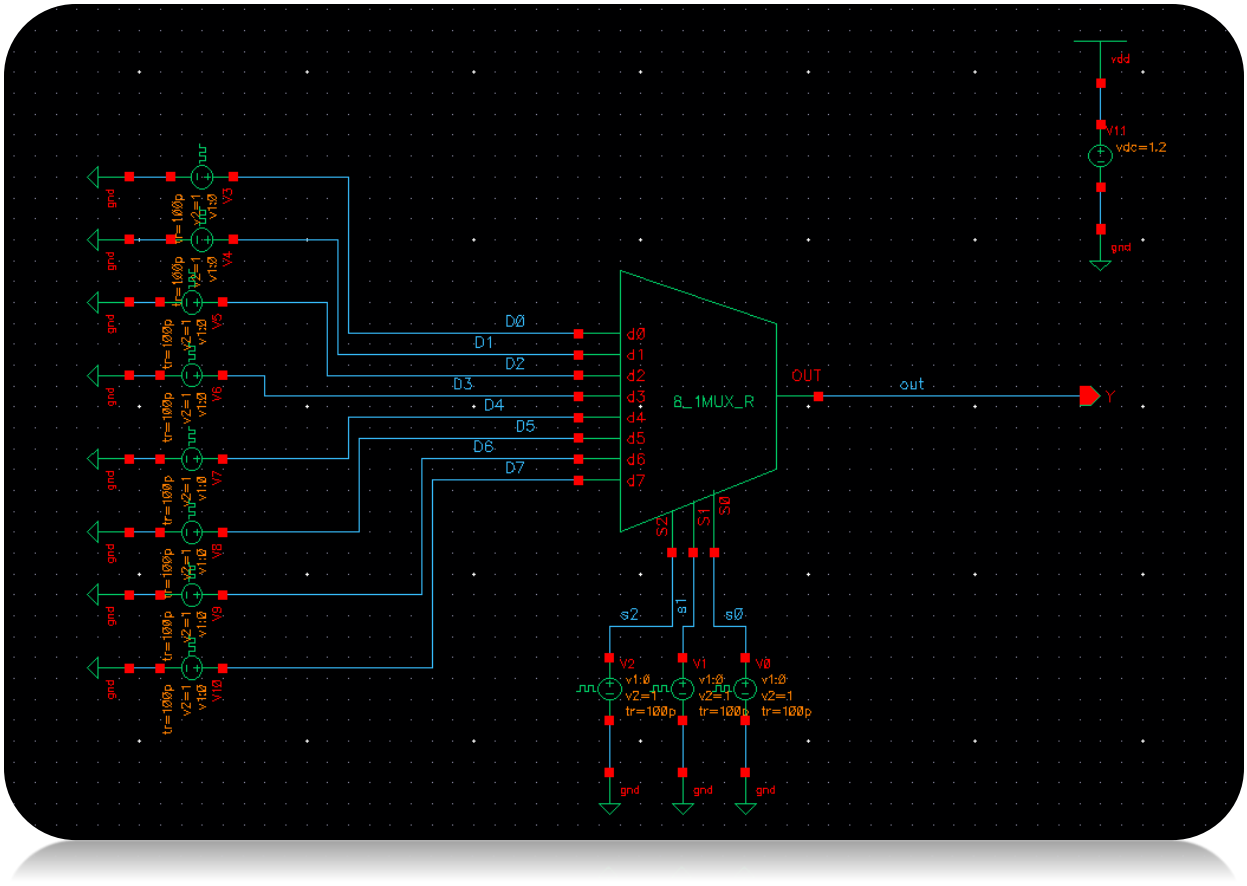


Symbol

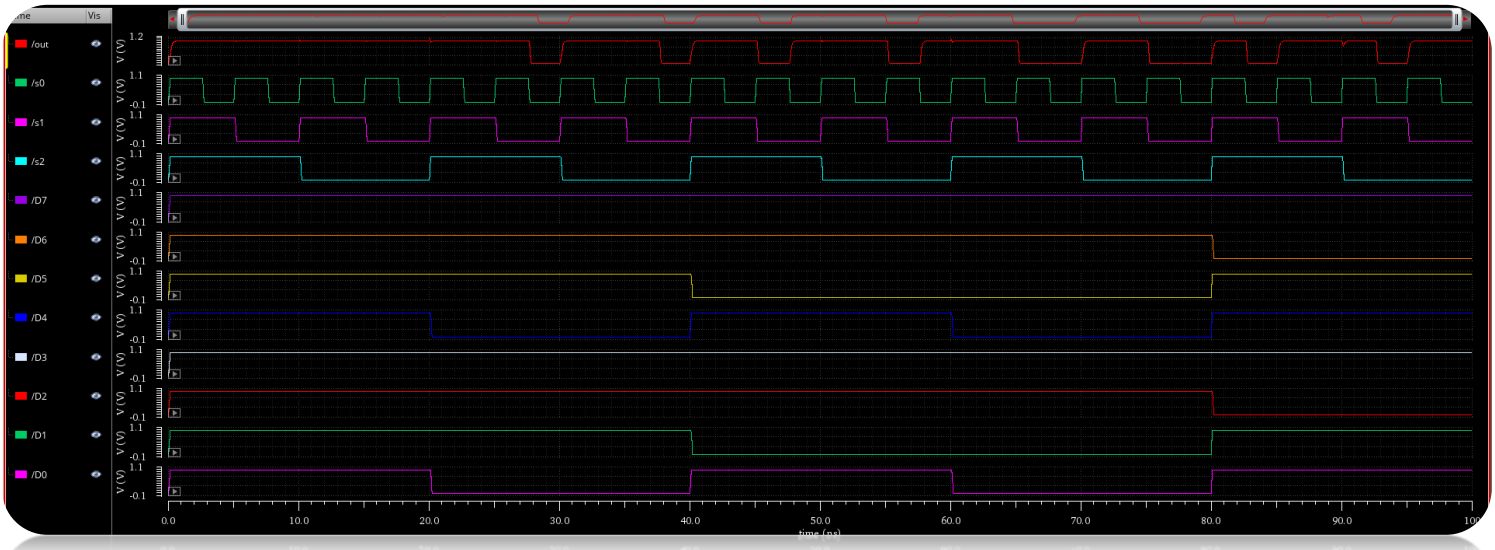


Note: The R in the symbol name refers to real because we construct a 8-1 mux using 4_1 muxs but it doesn't work smoothly as expected.

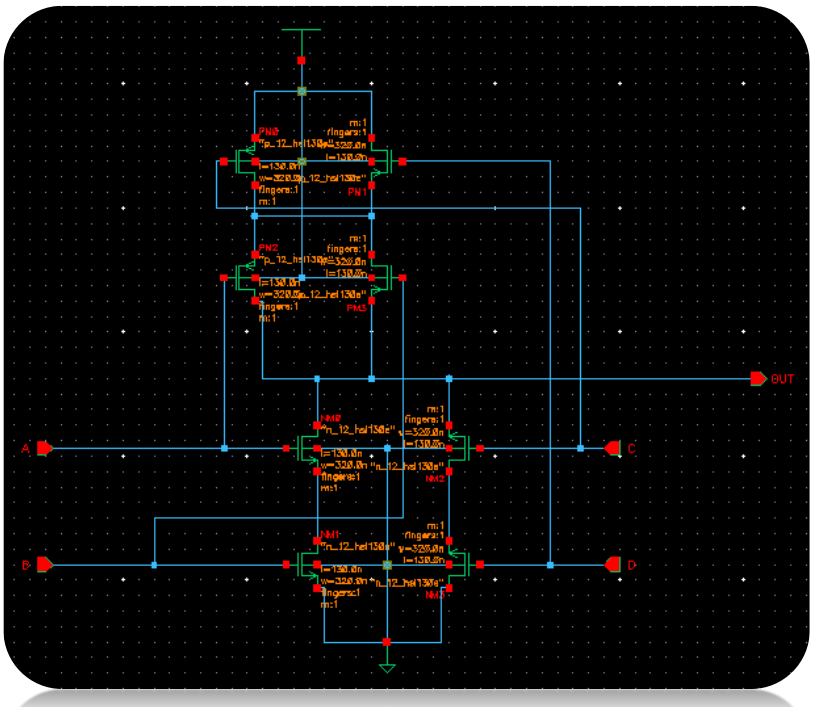
Testing of 8-1 Multiplexer



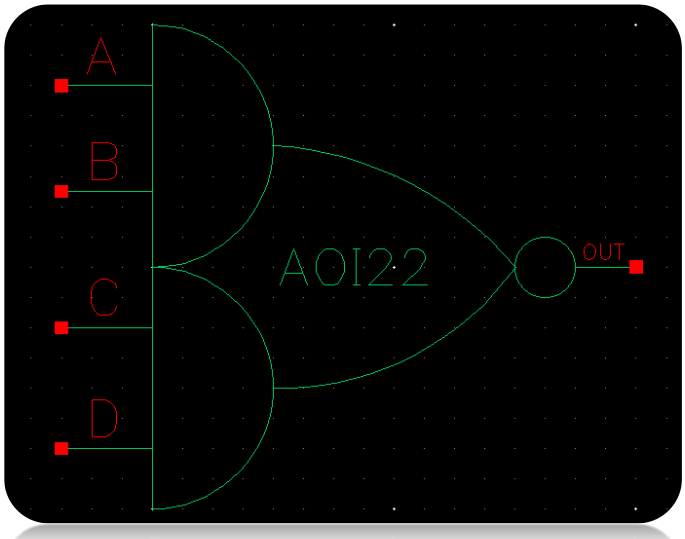
Output Waveform



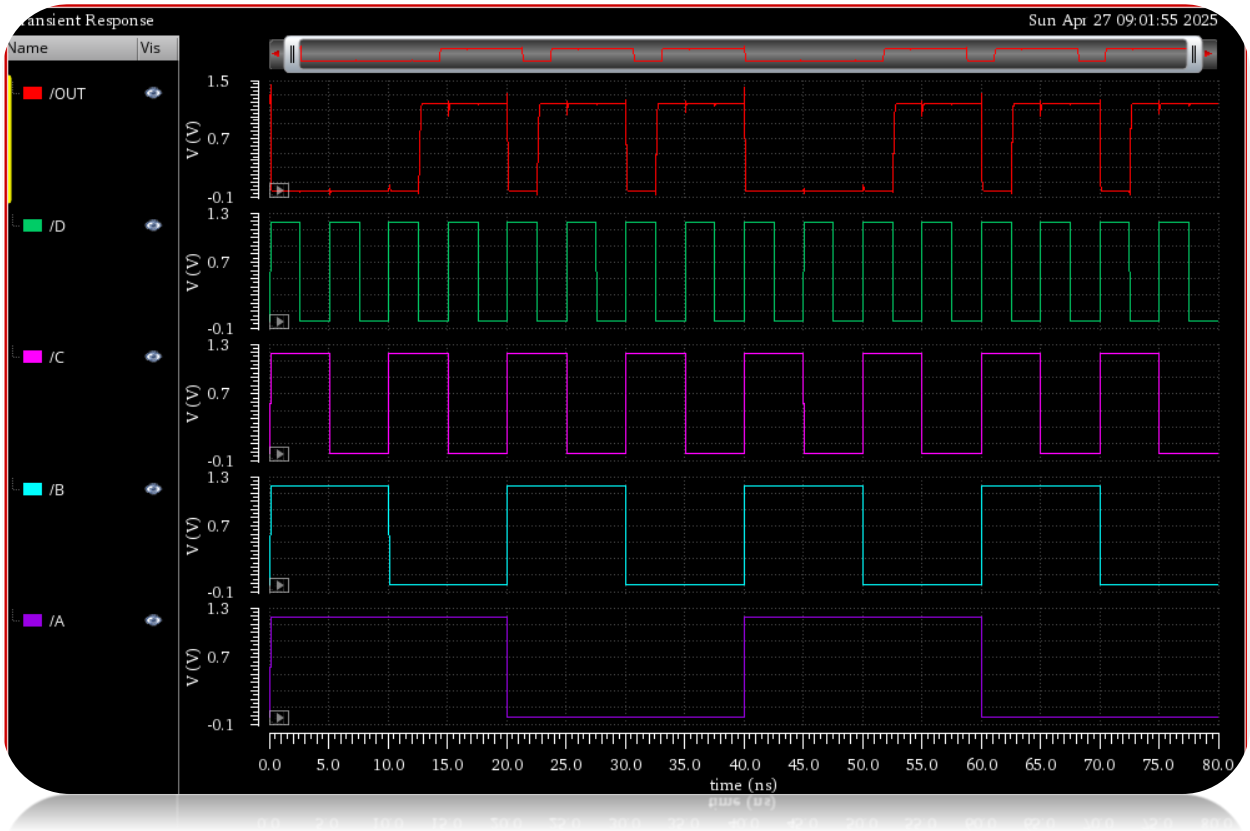
AOI



Schematic

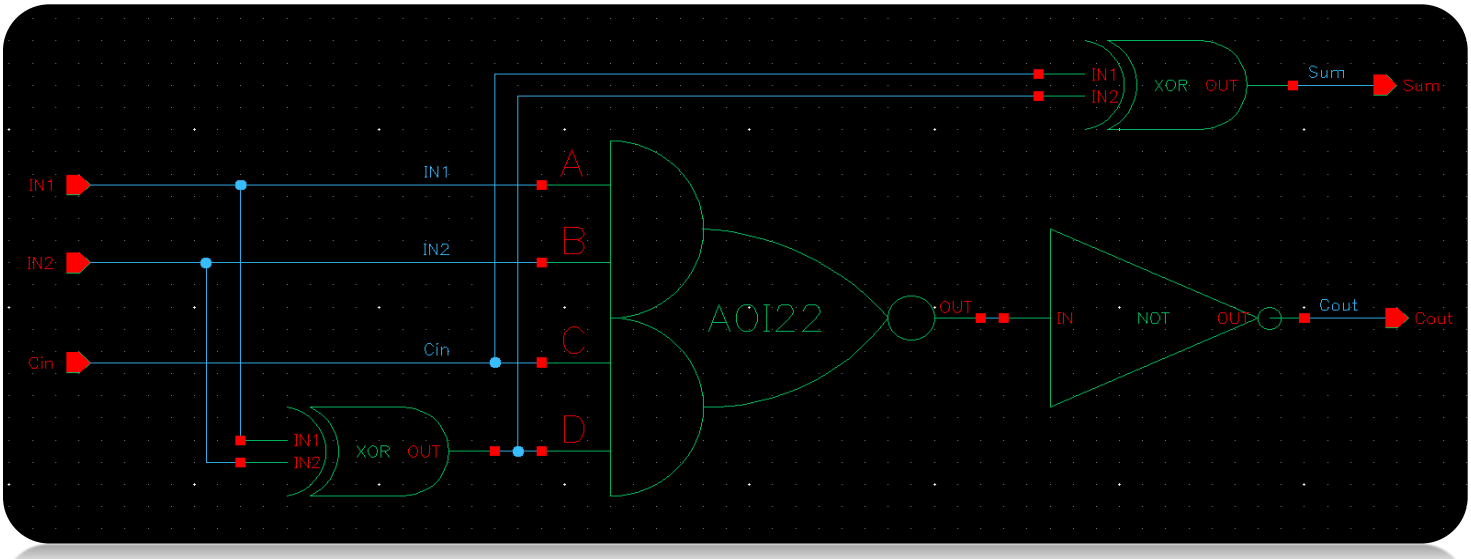


Symbol



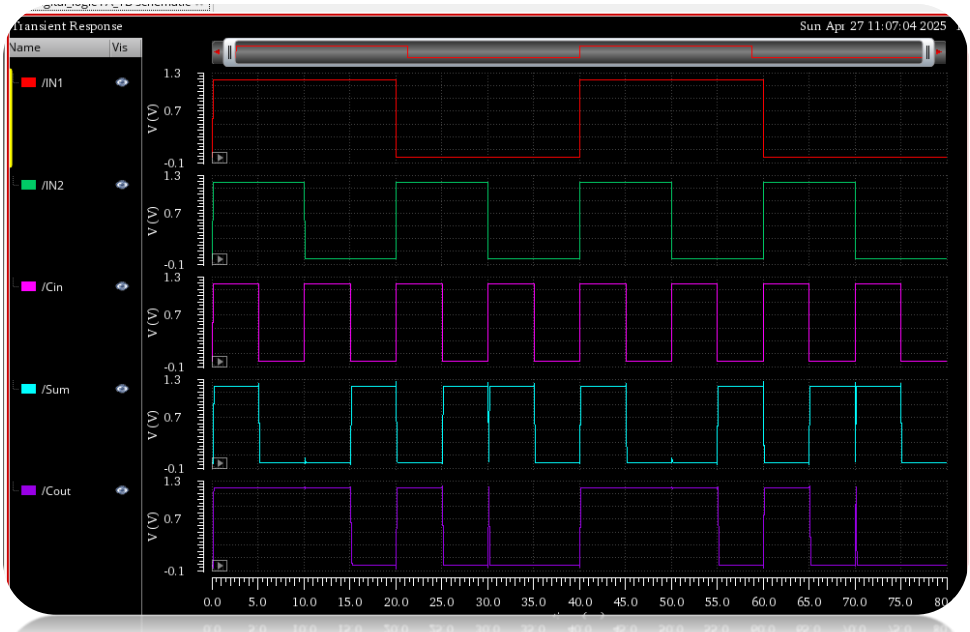
Testbench

Full Adder (1 bit)



Schematic

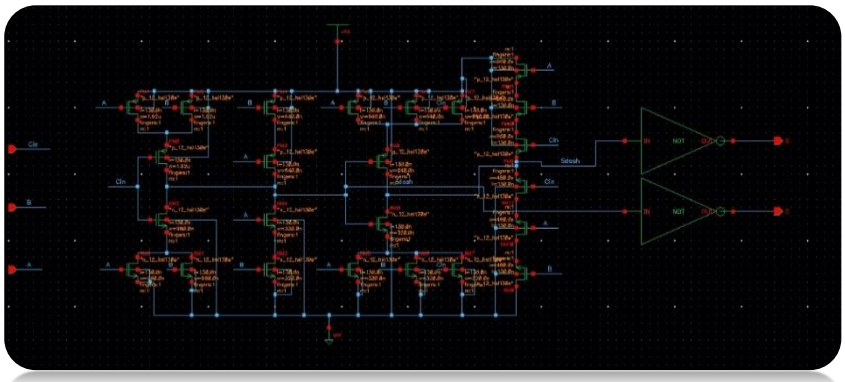
Symbol



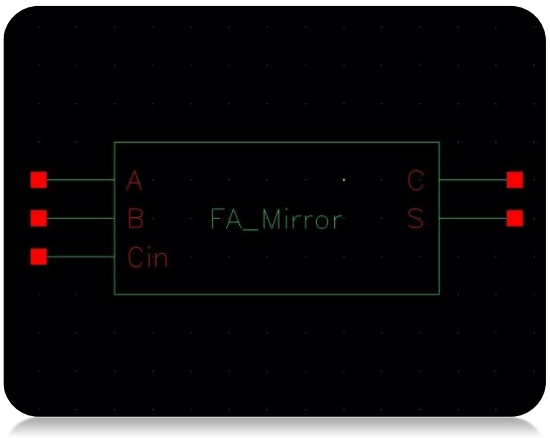
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Testbench

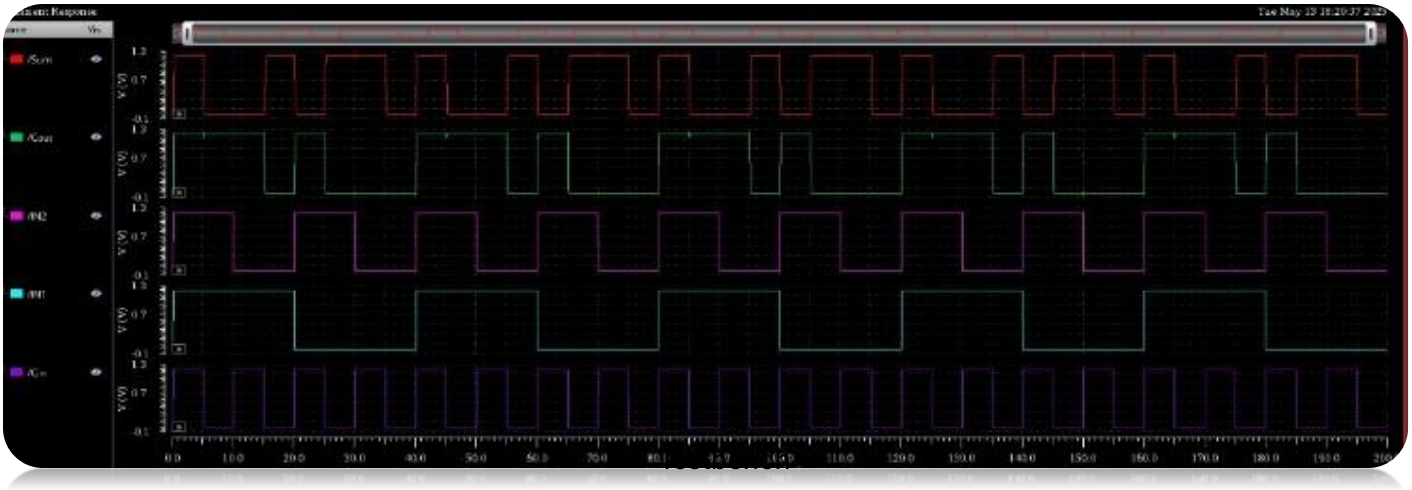
Full Adder Mirror (1 bit)



Schematic

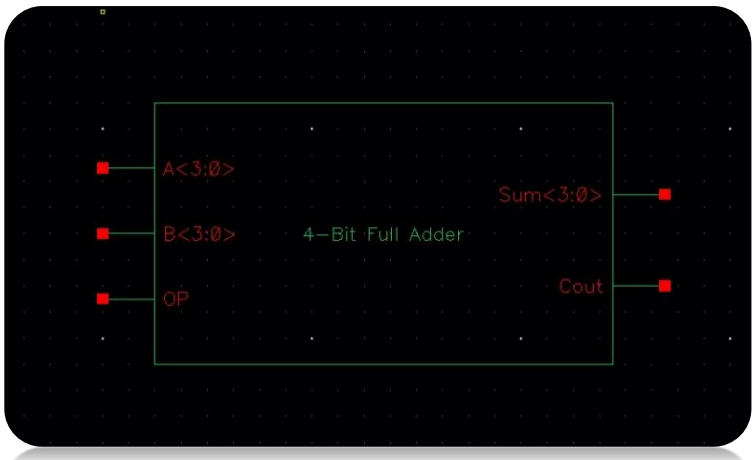


Symbol

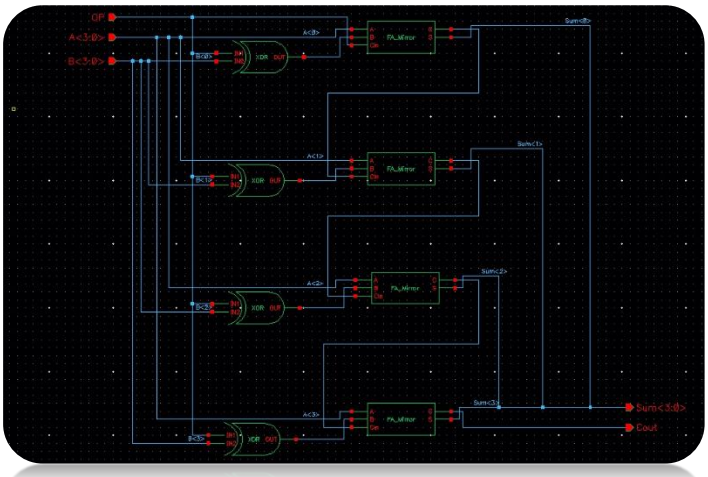


Testbench

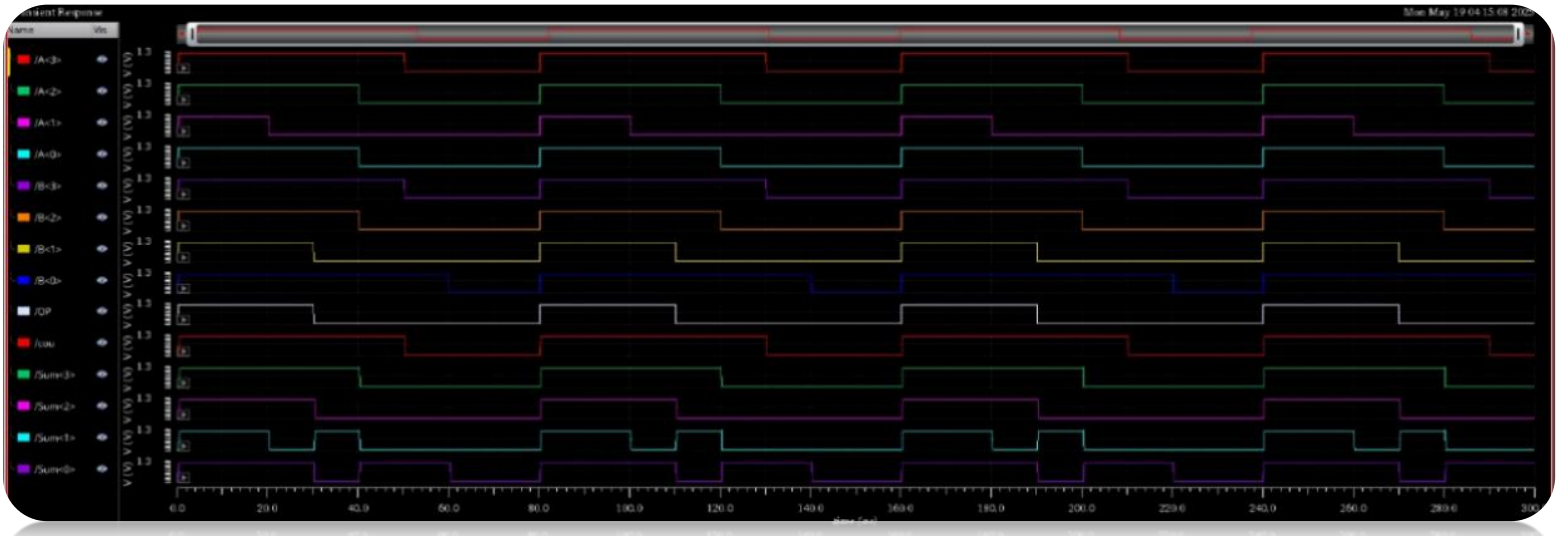
Full Adder Mirror (4Bit):



Symbol

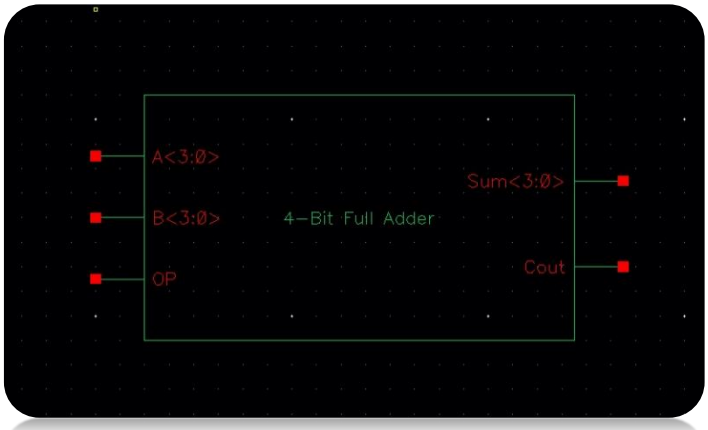


Schematic

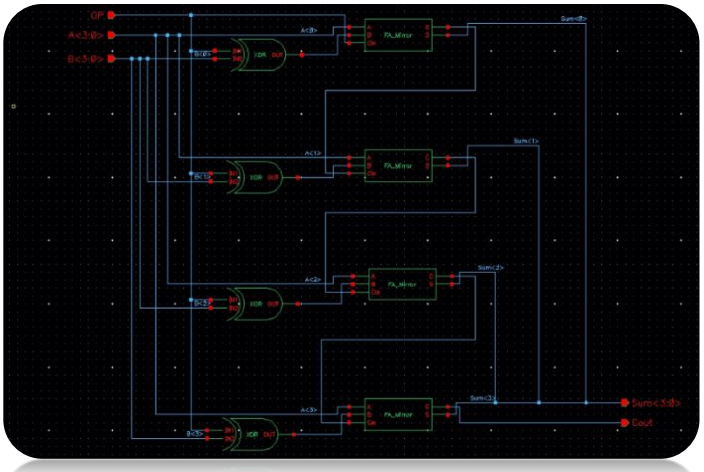


Testbench

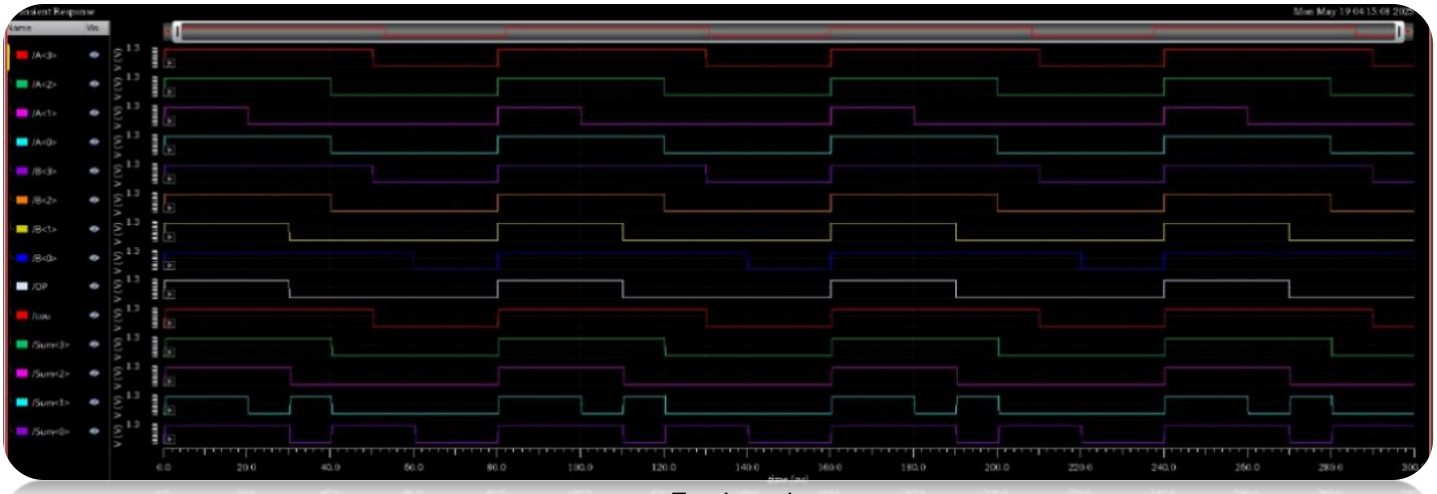
FA Mirror vs FA Basic logic



Symbol



Schematic



Testbench

Full Adder: Mirror Adder vs. Basic Logic

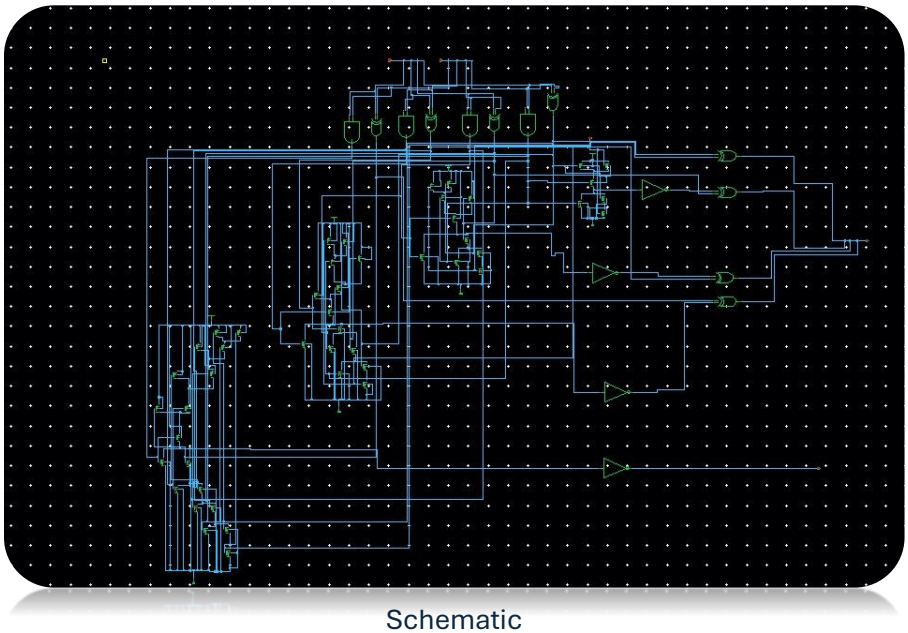
In the design of our Arithmetic and Logical Unit (ALU), we implemented and tested two types of full adders at the circuit level: the **Basic Logic Full Adder** and the **Mirror Adder**. The basic full adder is built using simple combinations of logic gates (AND, OR, XOR) directly derived from the Boolean expressions of sum and carry. While this approach is straightforward, it suffers from higher propagation delay due to longer logic paths and unbalanced signal flow.

To improve performance, we implemented the **Mirror Adder**, which is a more optimized CMOS design. It uses a symmetrical structure that ensures better signal balance and reduces the number of logic levels between input and output. We tested both designs by calculating and comparing their delay times. The **Mirror Adder showed significantly lower delay**, which translates to faster operation and better overall performance in the ALU. For this reason, we chose to integrate the Mirror Adder into our final ALU design.

In addition to the Basic Logic and Mirror full adders, we also implemented a **4-bit Carry Look-Ahead Adder (CLA)** at the circuit level. The CLA is designed to reduce the carry propagation delay by generating carry signals in parallel, rather than waiting for them to ripple through each stage. This architecture is known to provide significant speed improvements, especially for adders with large bit-widths such as 32 or 64 bits.

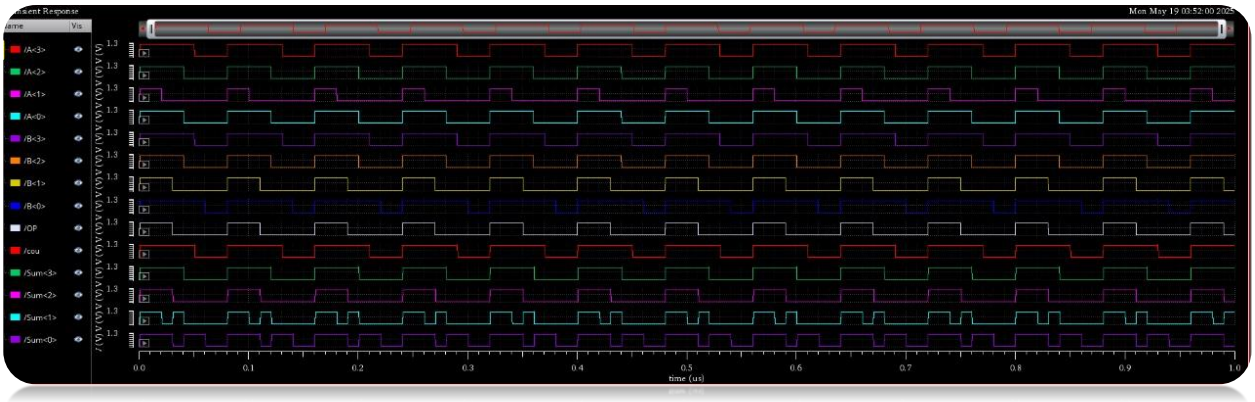
However, through our practical testing and delay time analysis, we observed that in the **4-bit configuration**, the CLA did **not outperform** the Basic Logic or Mirror adders. The added complexity of the carry generation logic introduces overhead that outweighs the benefits in such a small bit-width. Therefore, since our ALU is designed for 4-bit operations, we **chose not to use the CLA** in the final unit, as the **Mirror Adder provided better performance** in this specific case.

Carry Look Ahead 4 bit



Schematic

Symbol



Why CLA Not suitable in the design

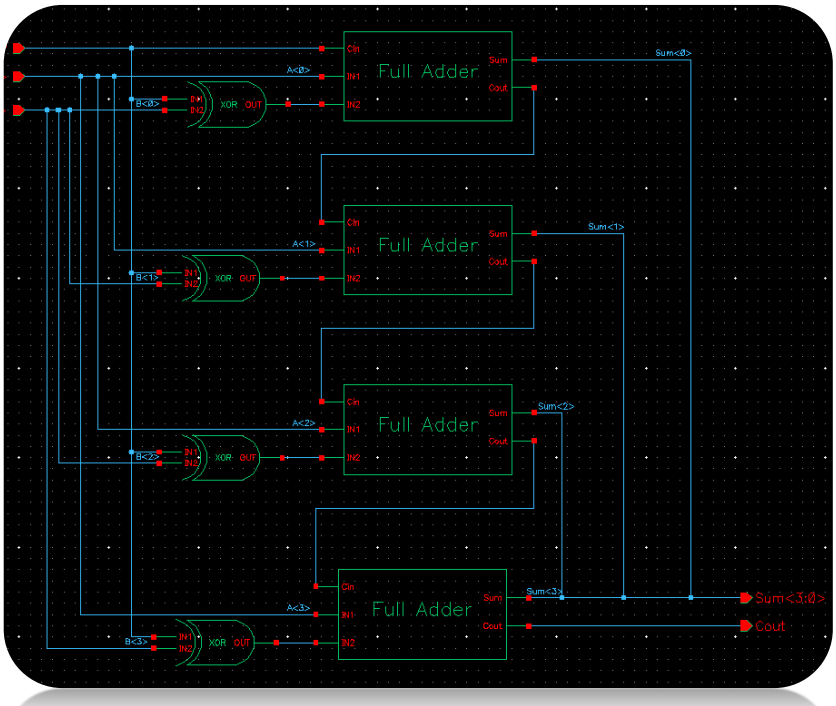
Although the **Carry Look-Ahead Adder (CLA)** is known for its ability to perform high-speed addition by reducing carry propagation delay, it was not suitable for our ALU design. Our unit is specifically designed for **4-bit operations**, where the carry delay is already minimal due to the small number of stages.

In practice, we implemented and tested a 4-bit CLA at the circuit level to compare its performance with the Basic and Mirror adders. Our **simulation results showed that the CLA was actually slower** than both, mainly due to the **increased complexity and gate overhead** required to compute the generate and propagate signals. These extra logic levels introduce delay that cancels out the advantage of parallel carry generation in such a small bit-width.

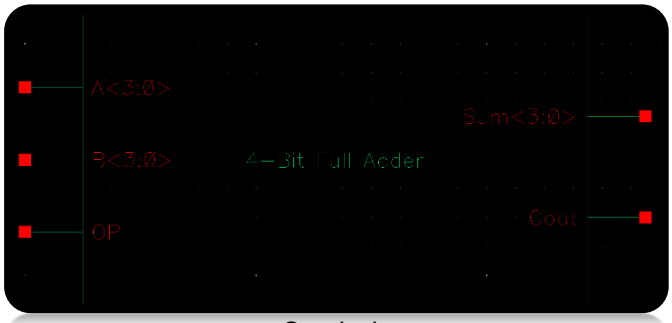
Furthermore, the CLA requires **more transistors and layout area**, which is not ideal for a compact 4-bit ALU. CLA's performance advantage becomes noticeable in **32-bit or 64-bit systems**, where ripple carry delay is significant—but in our case, it adds unnecessary complexity without any speed benefit.

Therefore, we concluded that the **CLA is not suitable for a 4-bit ALU**, and instead selected the **Mirror Adder** for its better performance and lower delay in this context.

Full Adder (4 bit)



Schematic



Symbol

Testbench

Why use Full Adder with mux instead of Subtractor

In our ALU design, we initially implemented a dedicated **subtractor circuit**. However, after analysis and testing, we decided **not to use the separate subtractor** in the final design. Instead, we used a **Full Adder combined with a multiplexer (MUX)** to perform both addition and subtraction using the same hardware.

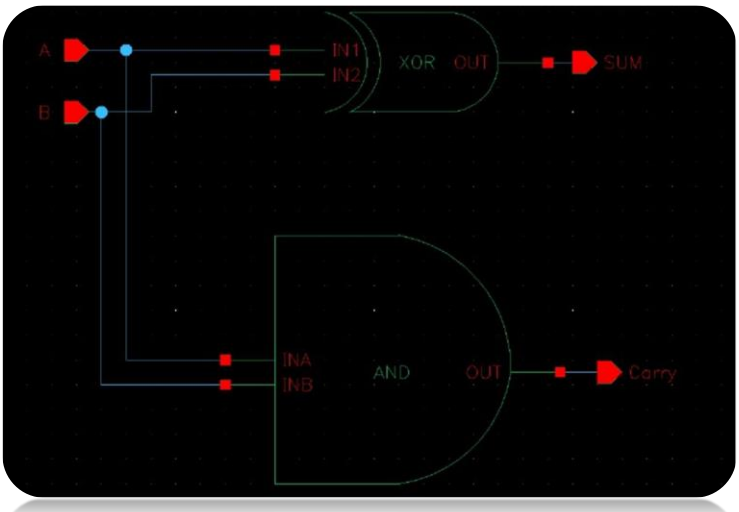
This approach is based on the **2's complement method** for subtraction, where A - B is computed as A + (~B + 1). By inverting the second operand (B) and adding 1 through the carry-in input of the adder, we can perform subtraction using the same full adder used for addition. A **MUX** is used to select whether the second input should be B (for addition) or ~B (for subtraction), and whether the carry-in should be 0 or 1.

The reasons for using this method are:

- **Hardware Efficiency**: Reusing the same adder circuit for both operations reduces the total number of gates and transistors.
- **Simpler Design**: Fewer separate blocks mean a more compact and manageable circuit.
- **Faster Operation**: Using a well-optimized full adder (like the Mirror Adder) with a small MUX introduces less delay compared to a dedicated subtractor path.
- **Flexibility**: The MUX-controlled design allows for easy switching between addition and subtraction through a control signal.

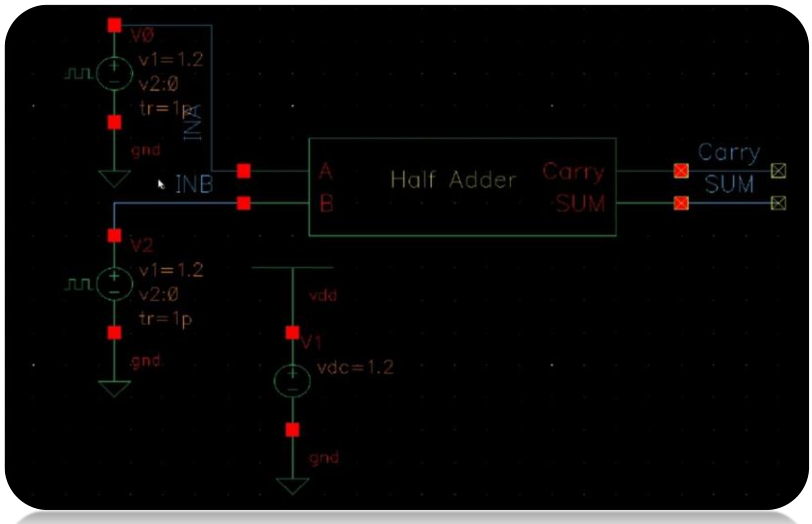
Therefore, to achieve a more **efficient, flexible, and faster design**, we chose to eliminate the separate subtractor and implement subtraction through a **Full Adder + MUX configuration** in our 4-bit ALU.

Half Adder

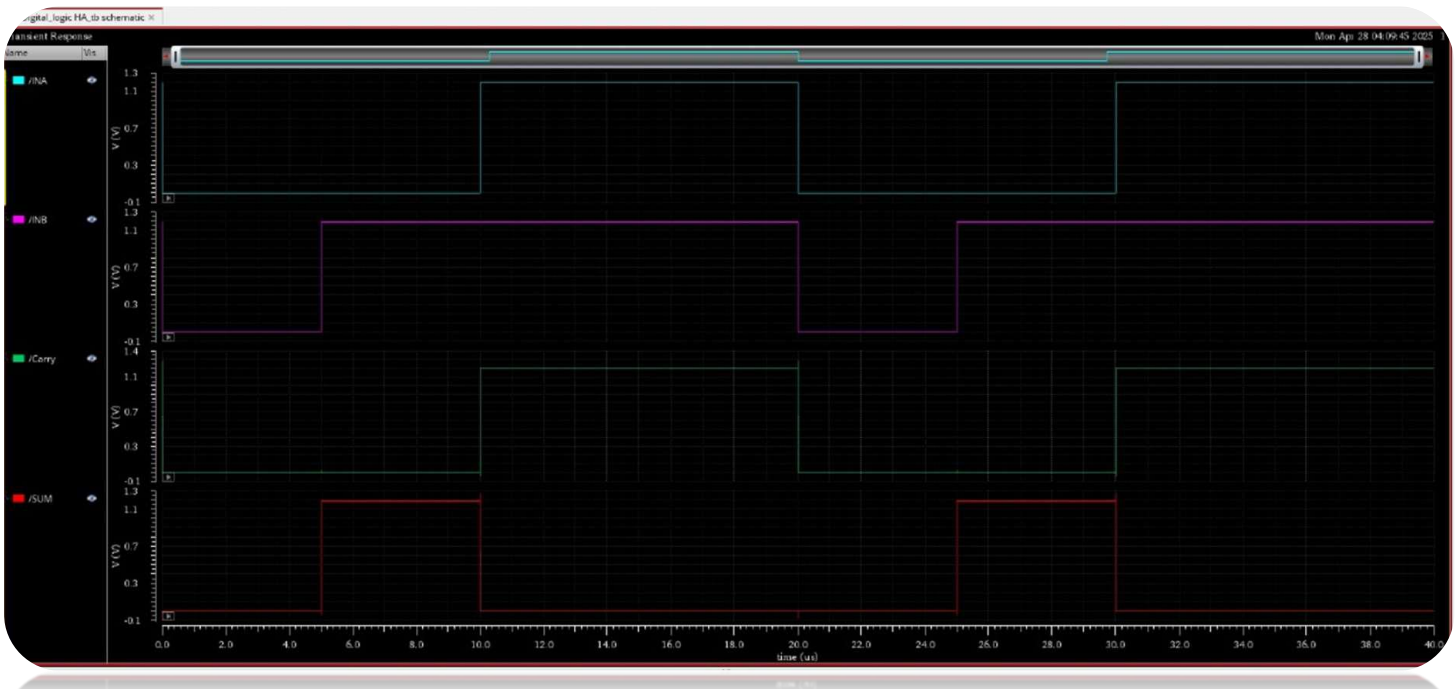


Schematic

Symbol



Testbench



Waveform

Comparison between Multiplier Architecture

In the process of designing the multiplication unit for our 4-bit ALU, we explored and analyzed four different multiplier architectures:

1. **Array Multiplier**
2. **Wallace Tree Multiplier**
3. **Dadda Tree Multiplier**
4. **Booth Multiplier**

Each architecture offers unique trade-offs in terms of **speed, complexity, area**, and **ease of implementation**. Below is a detailed comparison:

1. Array Multiplier

The array multiplier is a straightforward structure that uses a regular grid of **half adders and full adders** to compute partial products and sum them. It is easy to design and implement, especially for small bit-widths like 4-bit. However, it has a **longer delay** compared to tree-based architecture due to its **ripple carry-like structure**.

2. Wallace Tree Multiplier

The Wallace Tree multiplier uses a **reduction tree** to compress partial products as quickly as possible using carry-save adders. It reduces the number of addition stages, resulting in **faster performance** compared to the array multiplier, but it is **more complex** to implement and wire.

3. Dadda Tree Multiplier

The Dadda multiplier is similar to the Wallace tree but uses a slightly different strategy to minimize the number of adders used, making it more **area-efficient** while maintaining high speed. It is still complex to design and better suited for larger bit-widths.

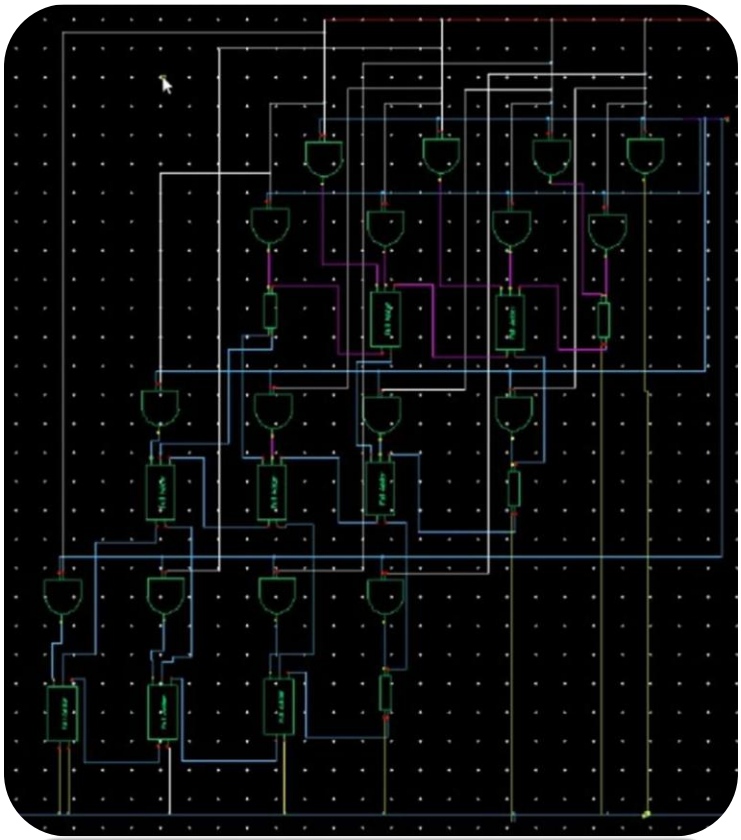
Multiplier Architecture Comparison Table

Feature / Architecture	Array Multiplier	Wallace Tree	Dadda Tree
Speed	Moderate	Fast	Fast
Complexity	Low	High	High
Area	Moderate	High	Lower than Wallace
Ease of Implementation	Easy	Hard	Hard
Best for Bit-width	Small (4–8 bits)	Large (16–64+)	Large (16–64+)

Conclusion

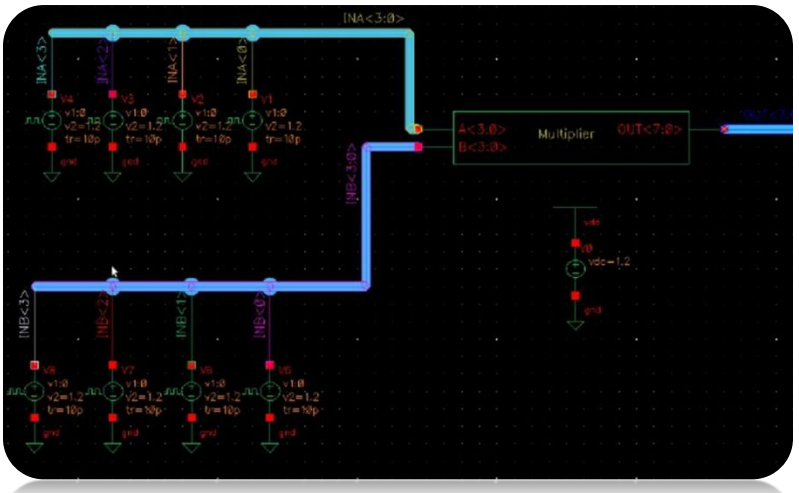
After analyzing all architectures and simulating their performance, we chose to implement the **Array Multiplier** for our 4-bit ALU. The **speed advantage of tree-based multipliers** like Wallace and Dadda is **not significant at 4-bit level**, and their complexity does not justify the small gain in delay. The **Array Multiplier** provides a **simpler and more practical solution** for small bit-width operations, making it ideal for our design goals.

Multiplier (unsigned)

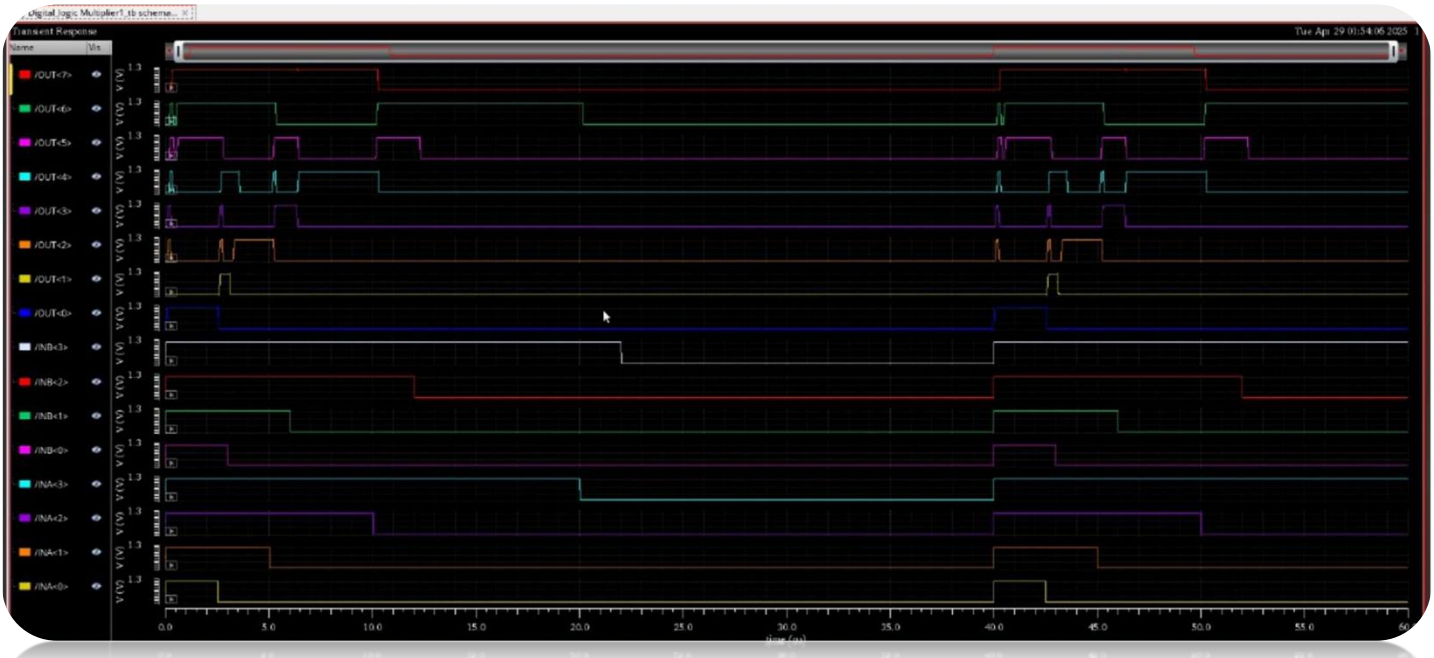


Schematic

Symbol

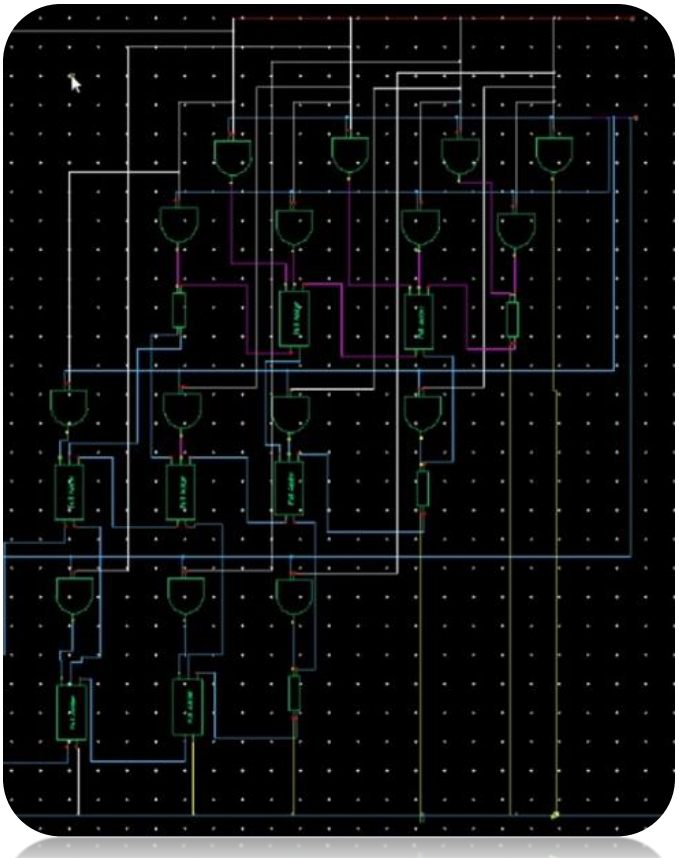


Testbench



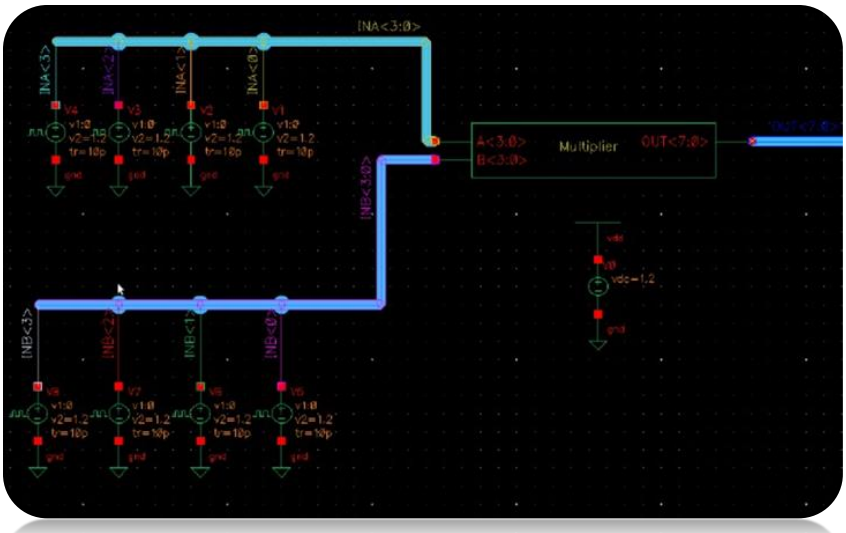
Waveform

Multiplier (Signed)

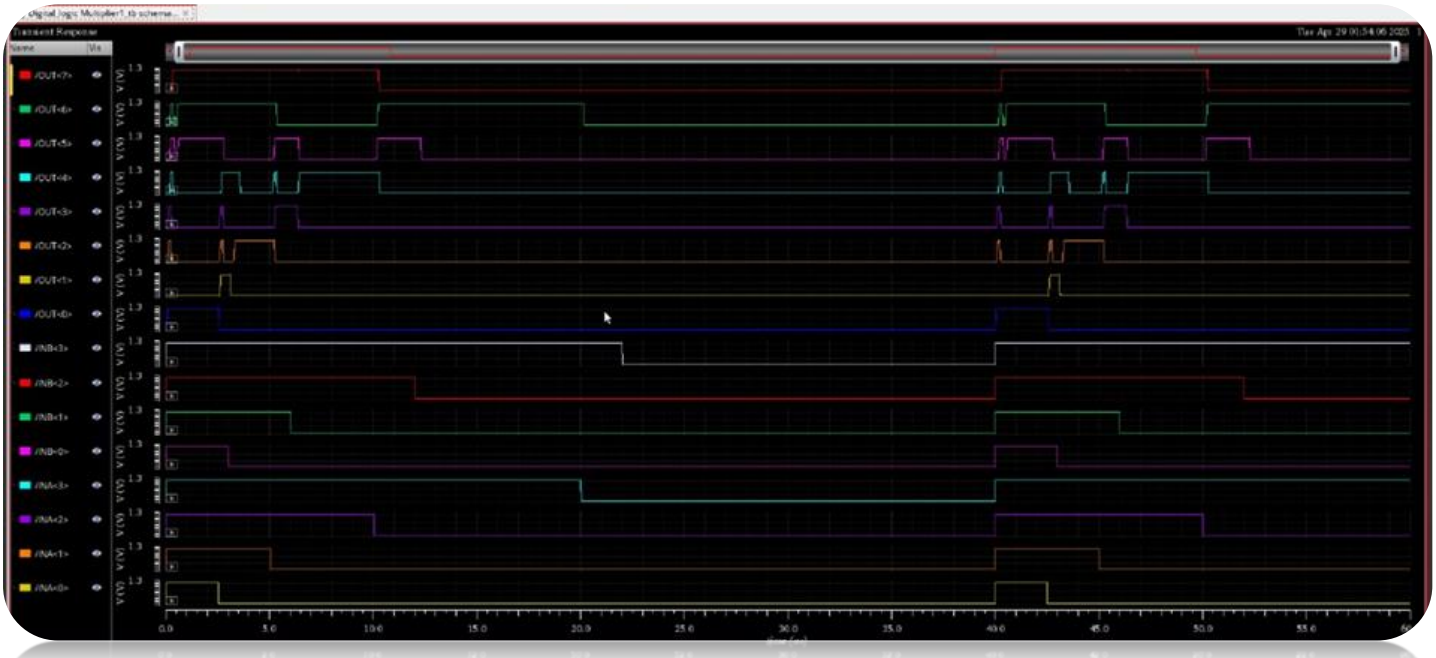


Schematic

Symbol

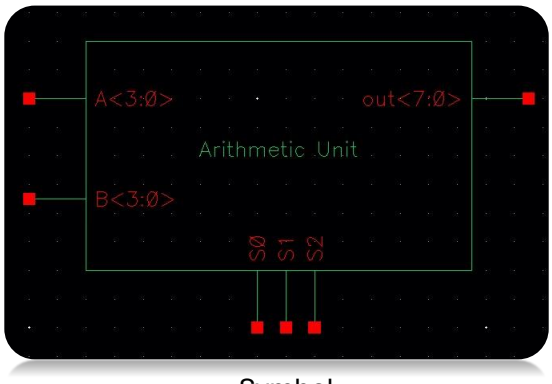


Testbench

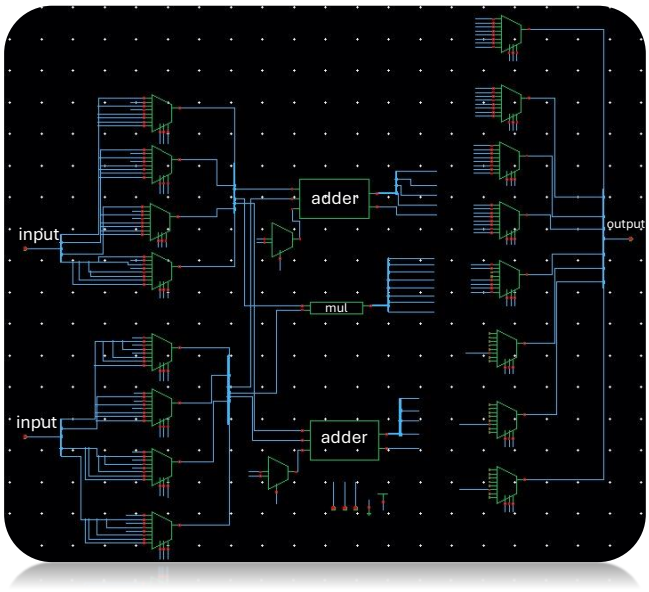


Waveform

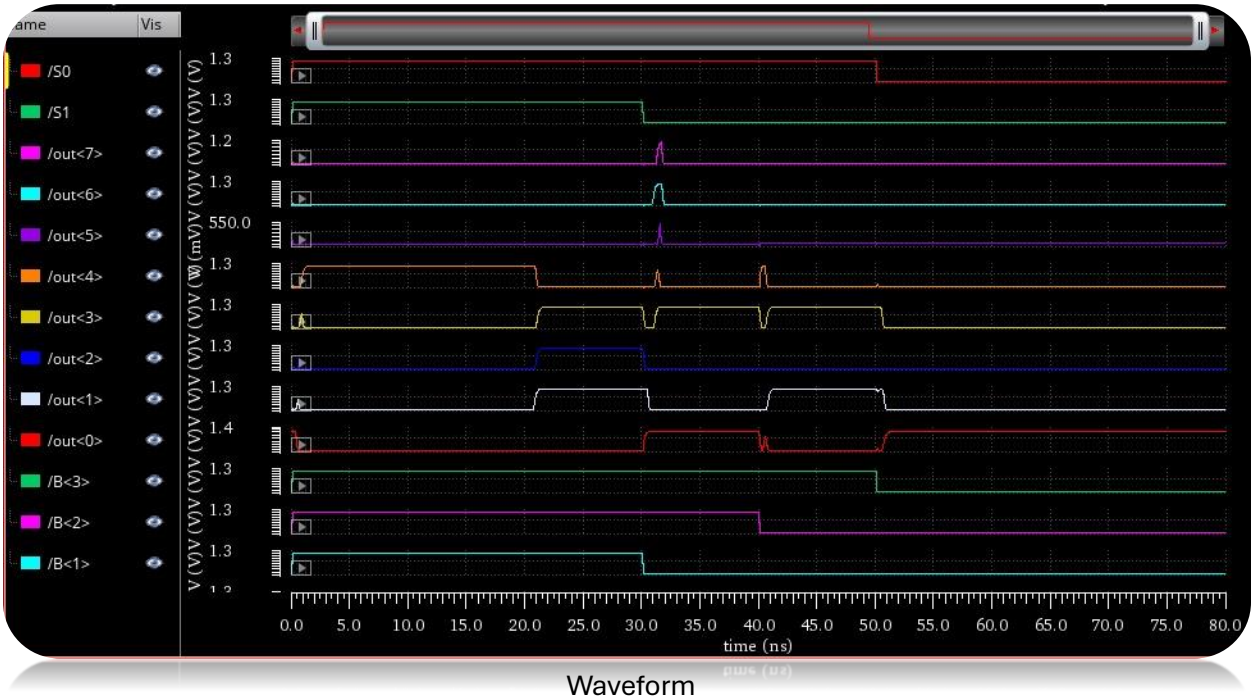
Arithmetic unit



Symbol

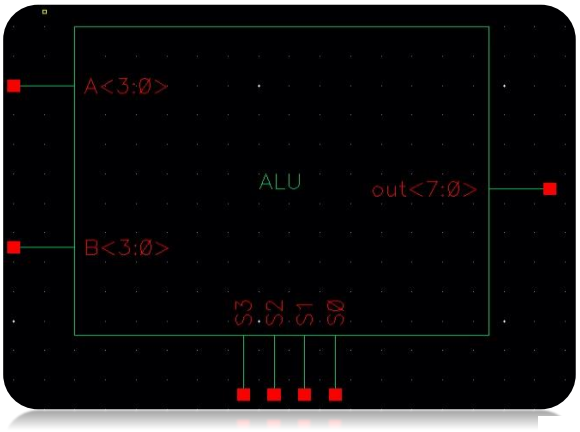


Schematic

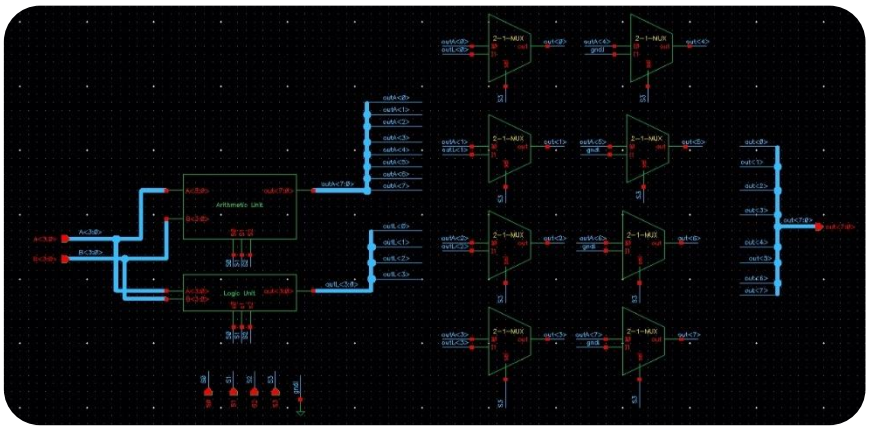


Waveform

ALU

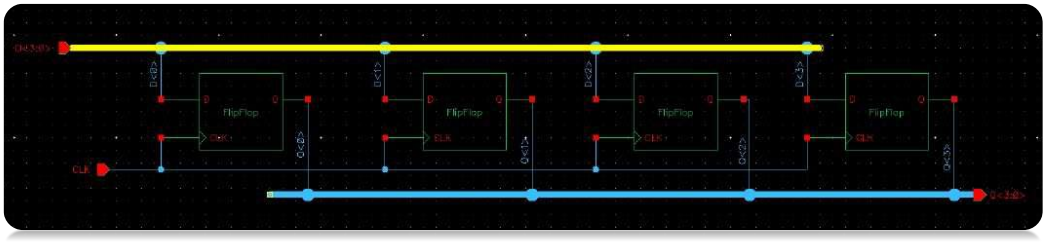


Symbol



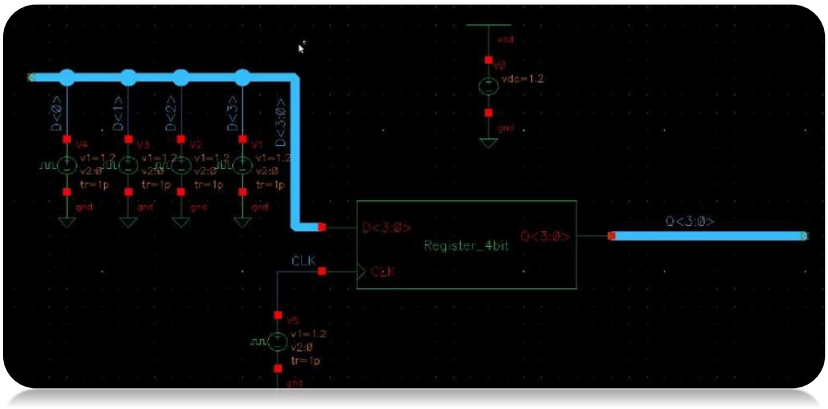
Schematic

Register (4bit)

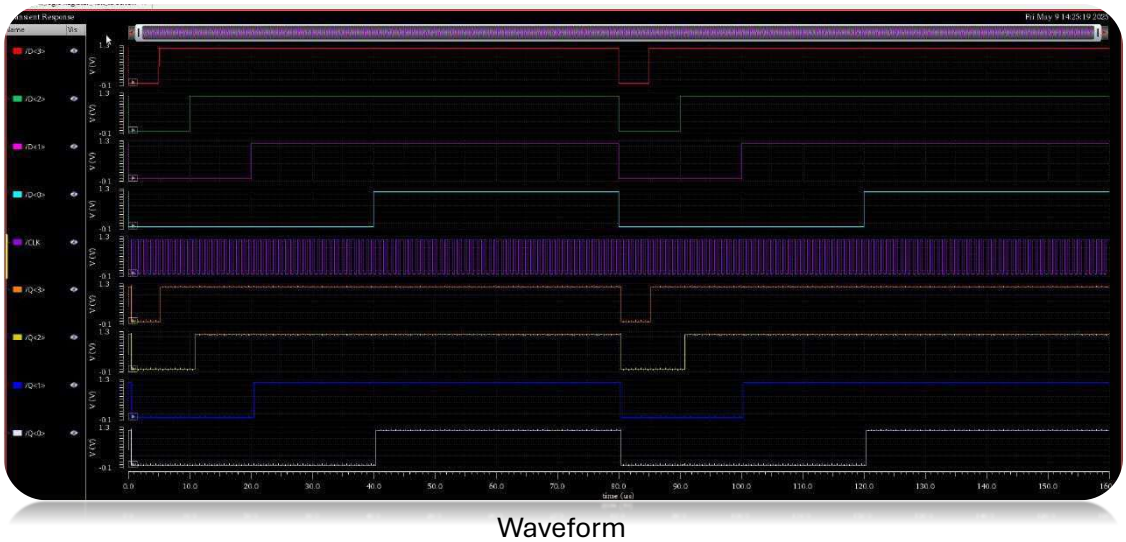


Schematic

Symbol

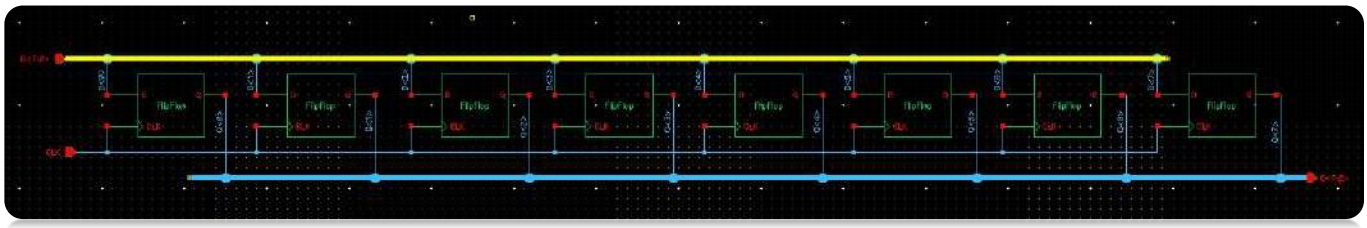


Testbench

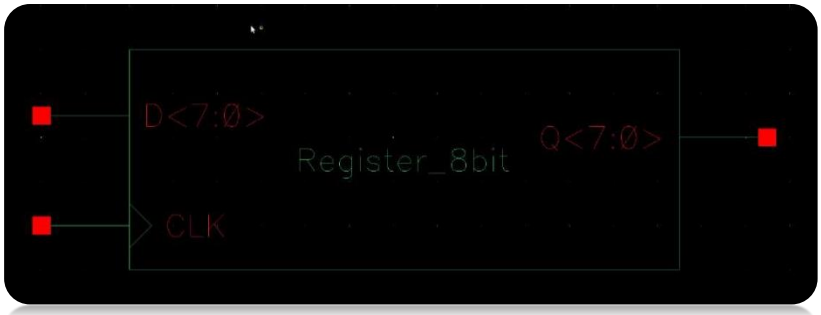


Waveform

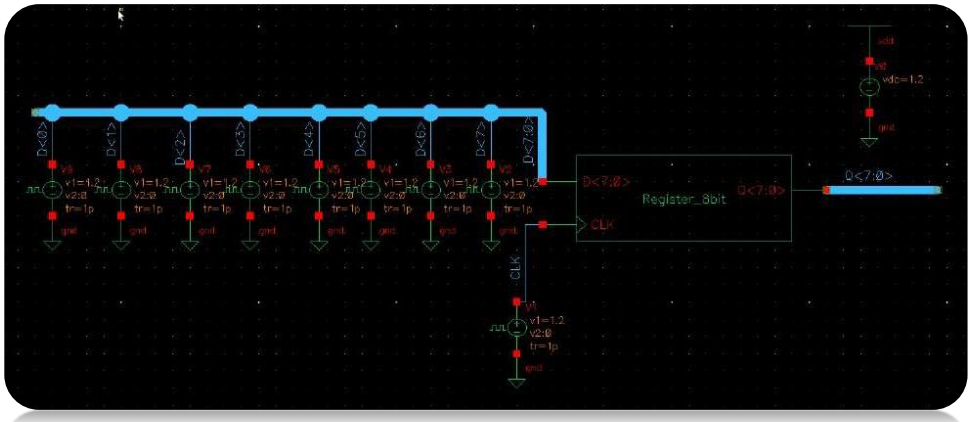
Register (8 bit)



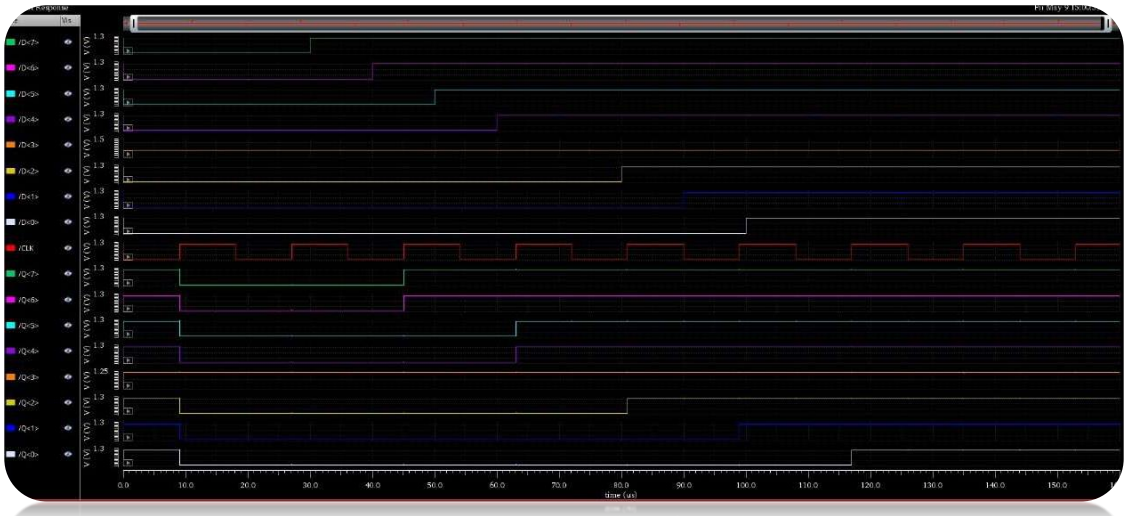
Schematic



Symbol



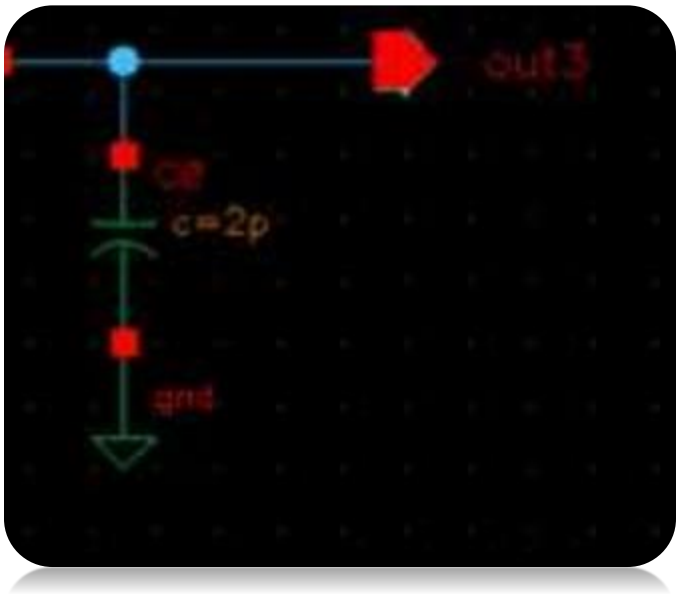
Testbench



Waveform

Capacitor

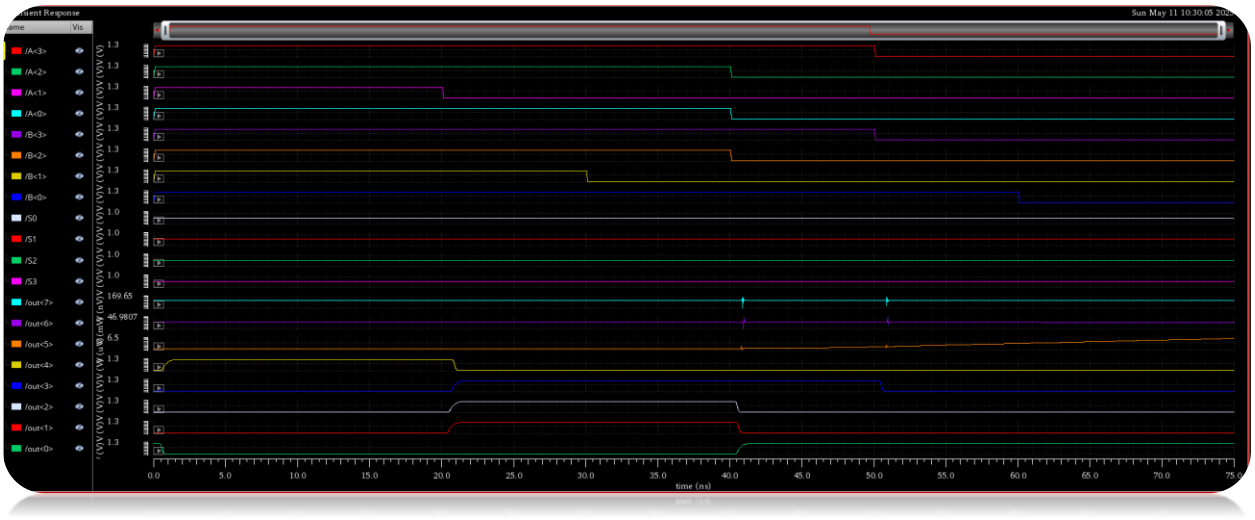
Circuit



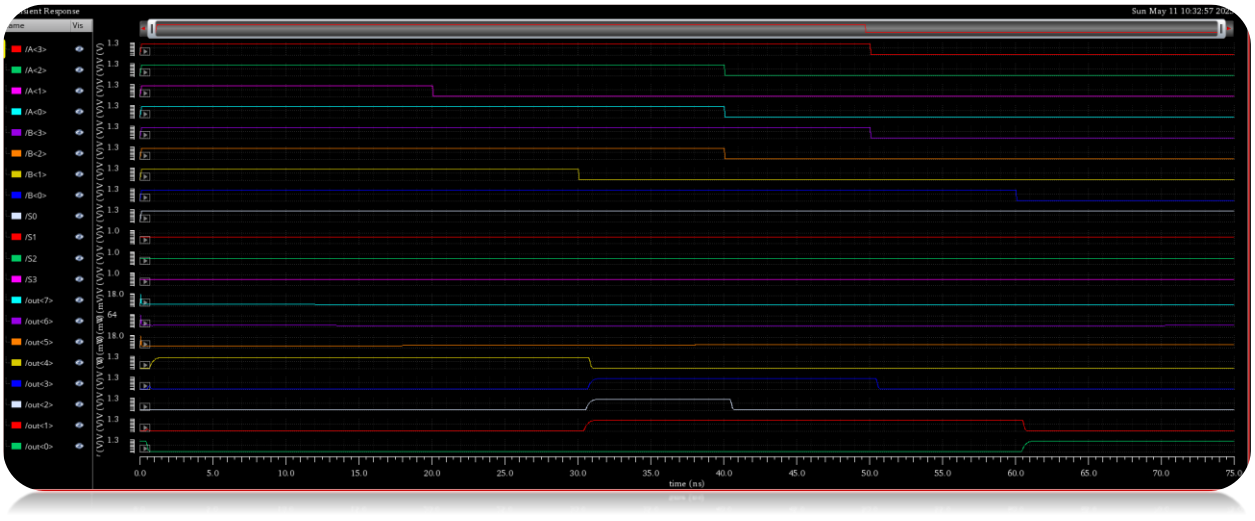
Simulation: Operations Waveforms

Arithmetic unit

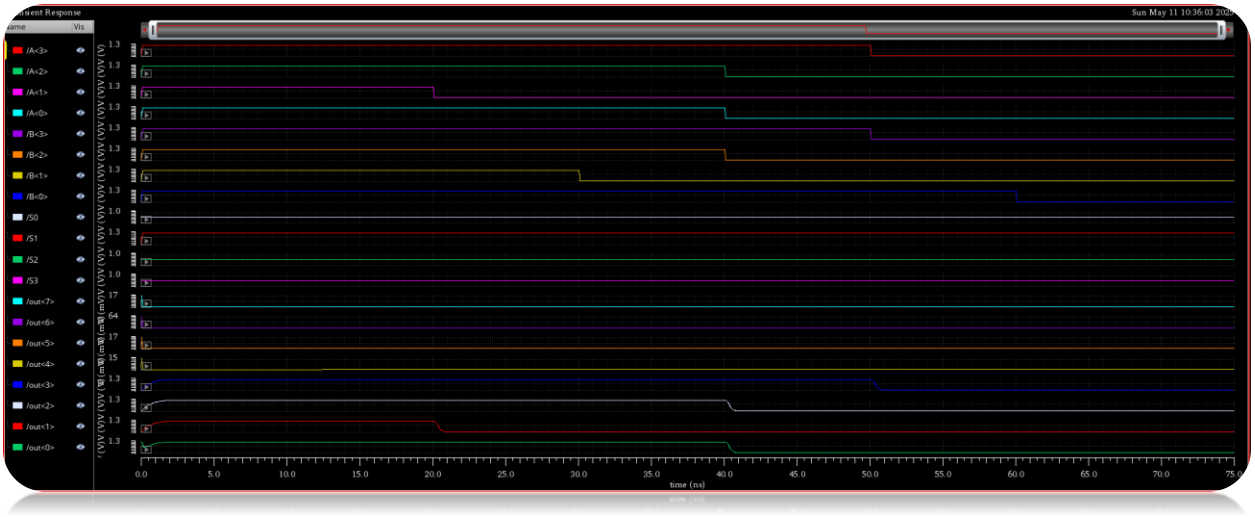
Increment a



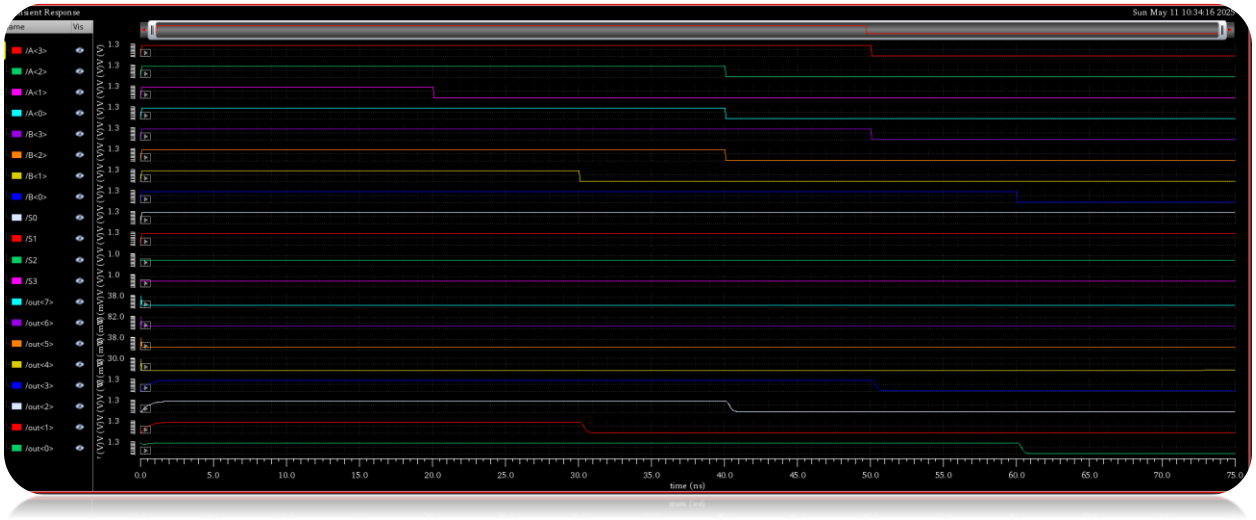
Increment b



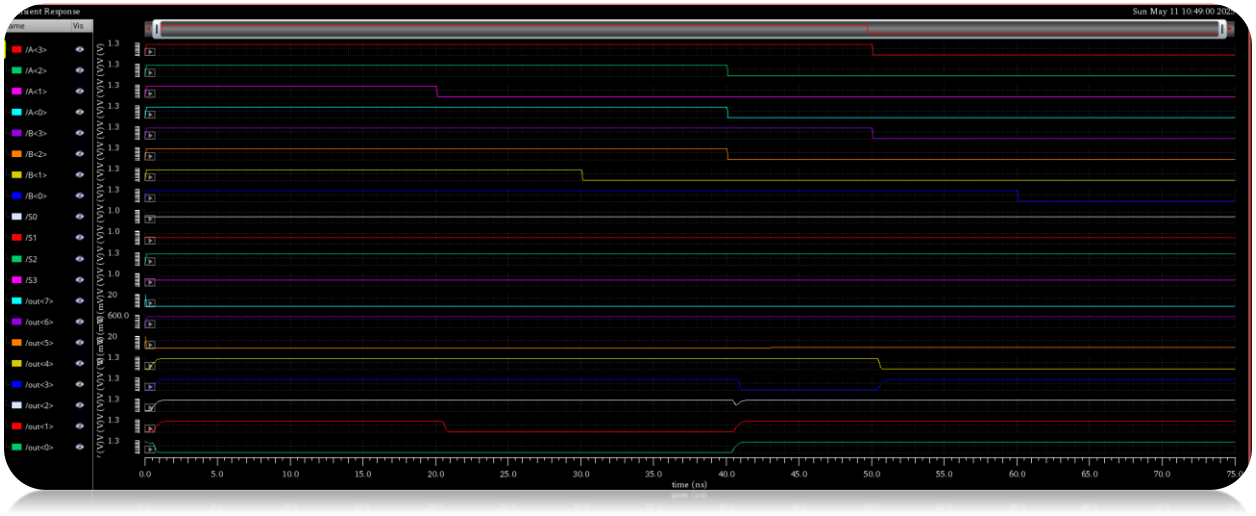
Transfer a



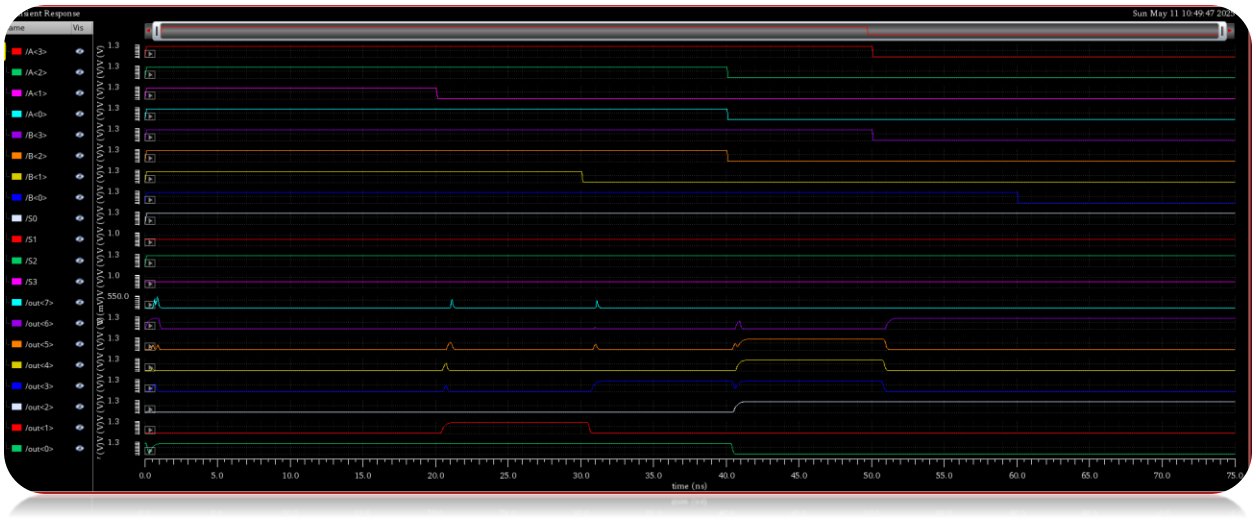
Transfer b



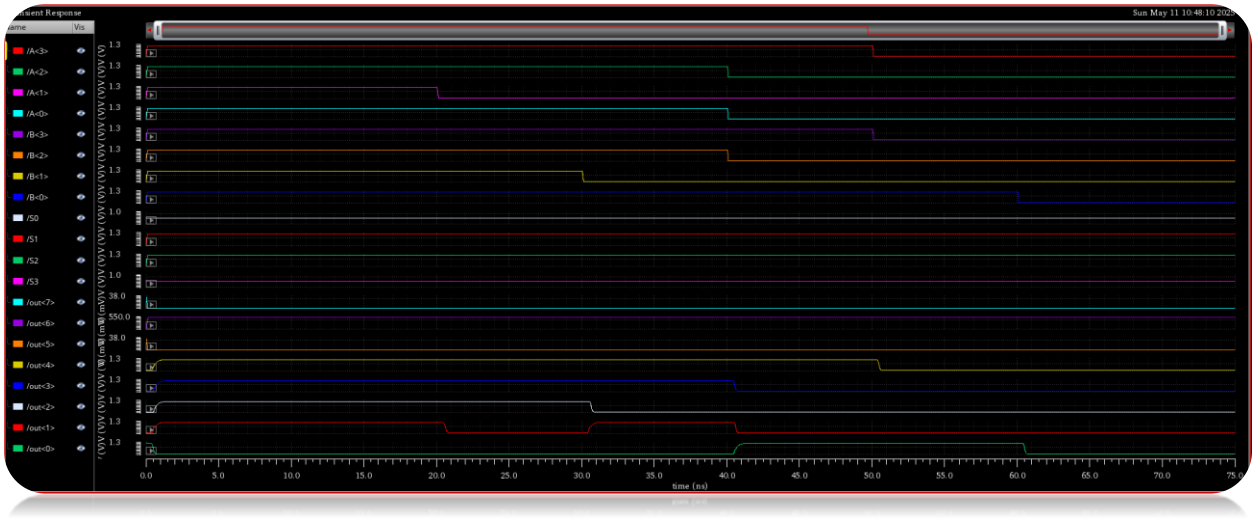
Decrement a



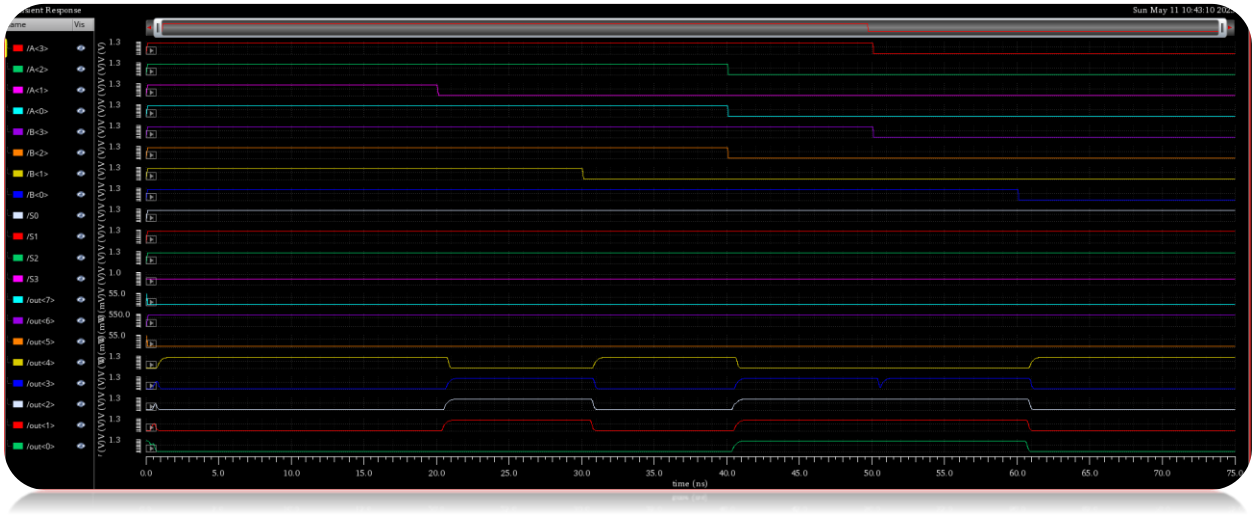
Multiply a and b



Add a and b

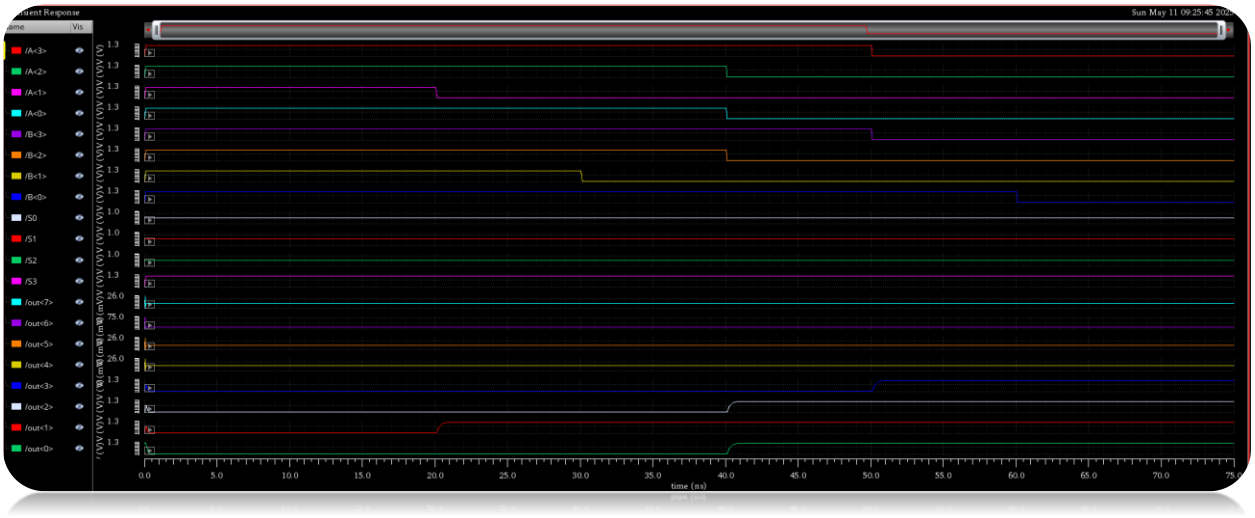


Subtract a and b

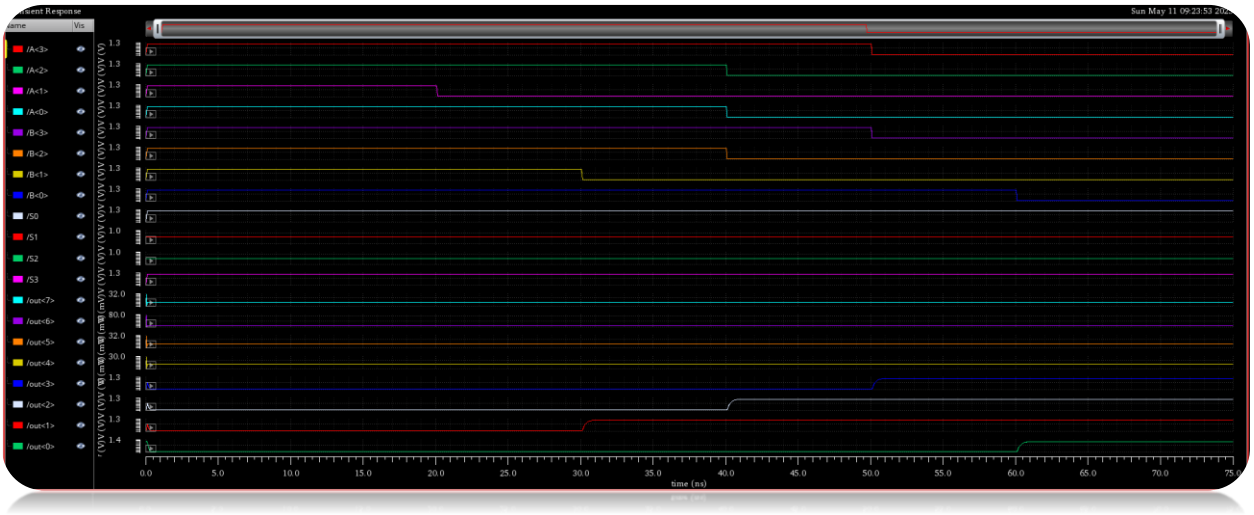


Logical unit

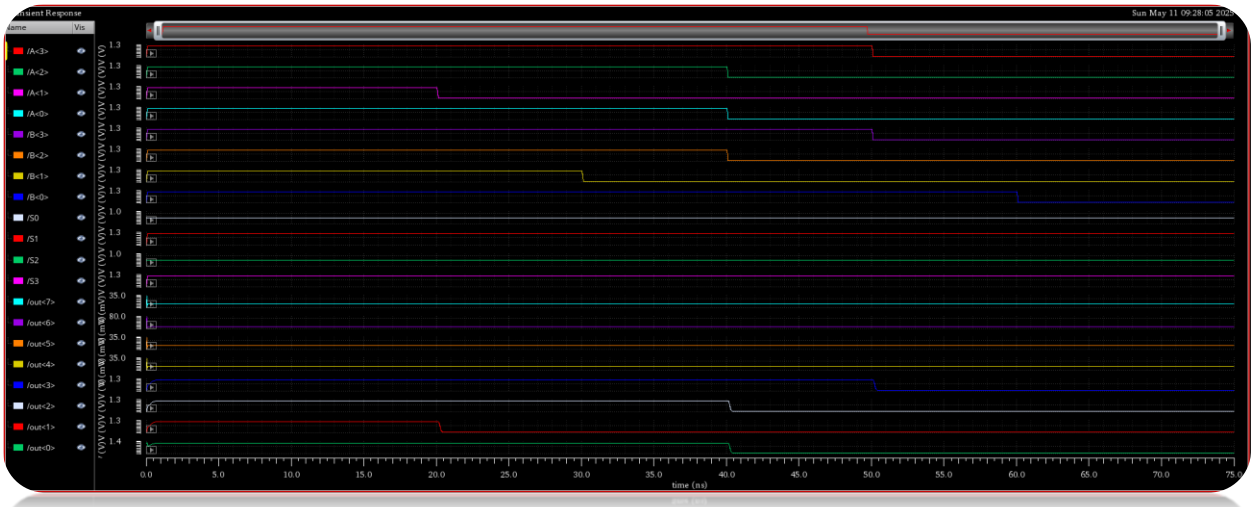
Complement a



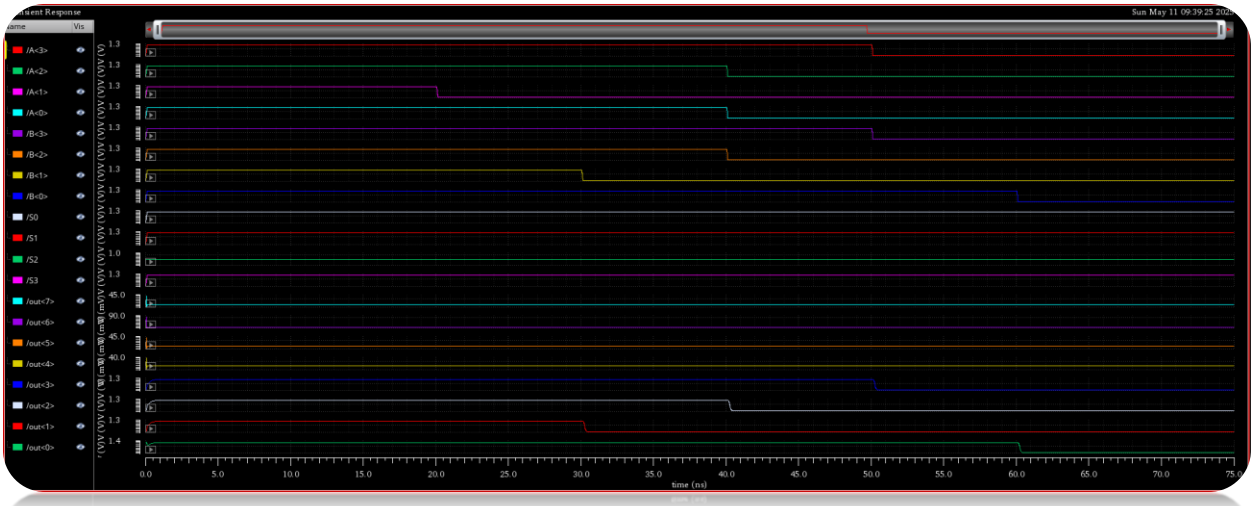
Complement b



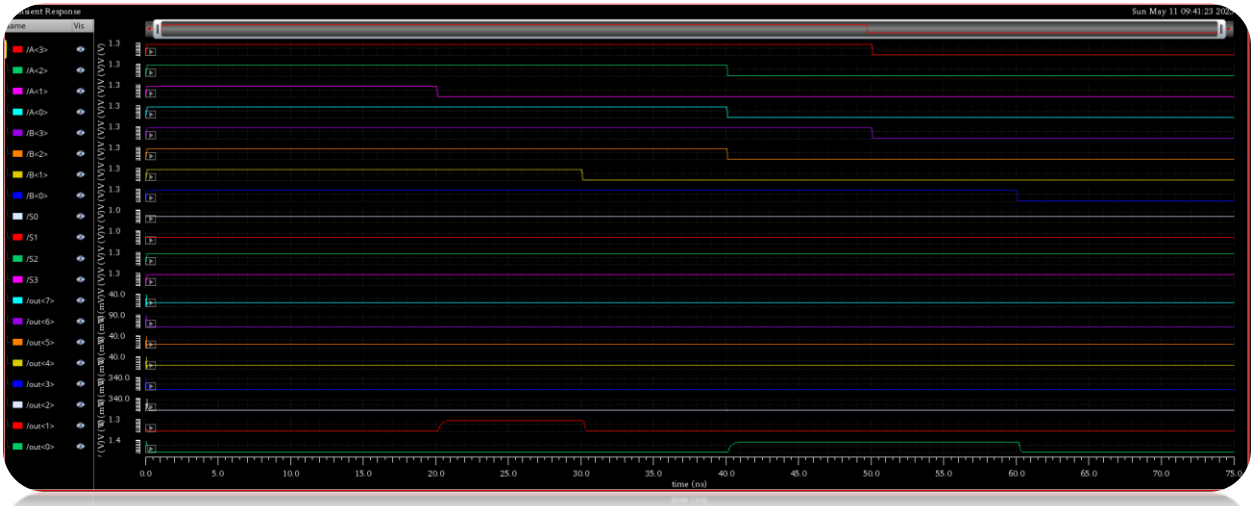
AND


Transient Response
Sun May 11 09:28:05 2025
/A<3>
/A<2>
/A<1>
/A<0>
/B<3>
/B<2>
/B<1>
/B<0>
/S0
/S1
/S2
/S3
/out<7>
/out<6>
/out<5>
/out<4>
/out<3>
/out<2>
/out<1>
/out<0>
0.0 5.0 10.0 15.0 20.0 25.0 30.0 35.0 40.0 45.0 50.0 55.0 60.0 65.0 70.0 75.0
time (ns)

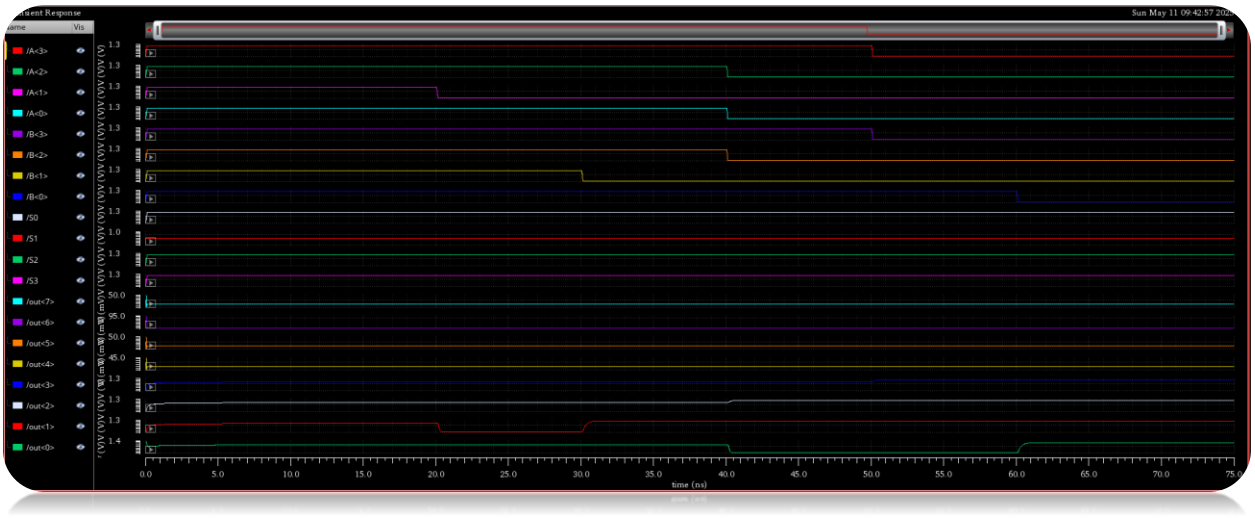
OR


Transient Response
Sun May 11 09:39:25 2025
/A<3>
/A<2>
/A<1>
/A<0>
/B<3>
/B<2>
/B<1>
/B<0>
/S0
/S1
/S2
/S3
/out<7>
/out<6>
/out<5>
/out<4>
/out<3>
/out<2>
/out<1>
/out<0>
0.0 5.0 10.0 15.0 20.0 25.0 30.0 35.0 40.0 45.0 50.0 55.0 60.0 65.0 70.0 75.0
time (ns)

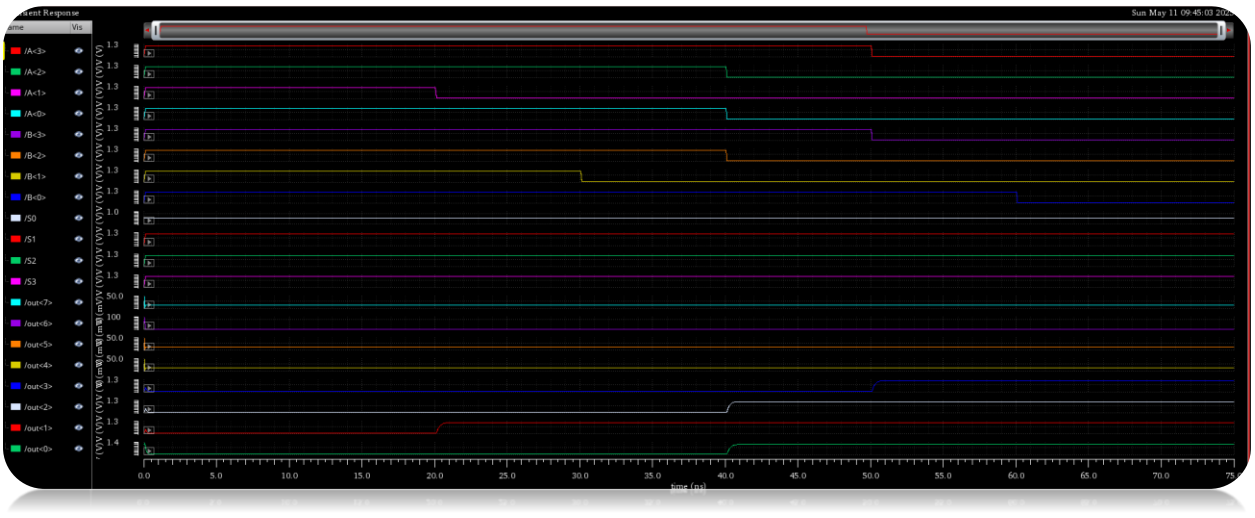
XOR


Transient Response
Sun May 11 09:41:23 2025
/A<3>
/A<2>
/A<1>
/A<0>
/B<3>
/B<2>
/B<1>
/B<0>
/S0
/S1
/S2
/S3
/out<7>
/out<6>
/out<5>
/out<4>
/out<3>
/out<2>
/out<1>
/out<0>
0.0 5.0 10.0 15.0 20.0 25.0 30.0 35.0 40.0 45.0 50.0 55.0 60.0 65.0 70.0 75.0
time (ns)

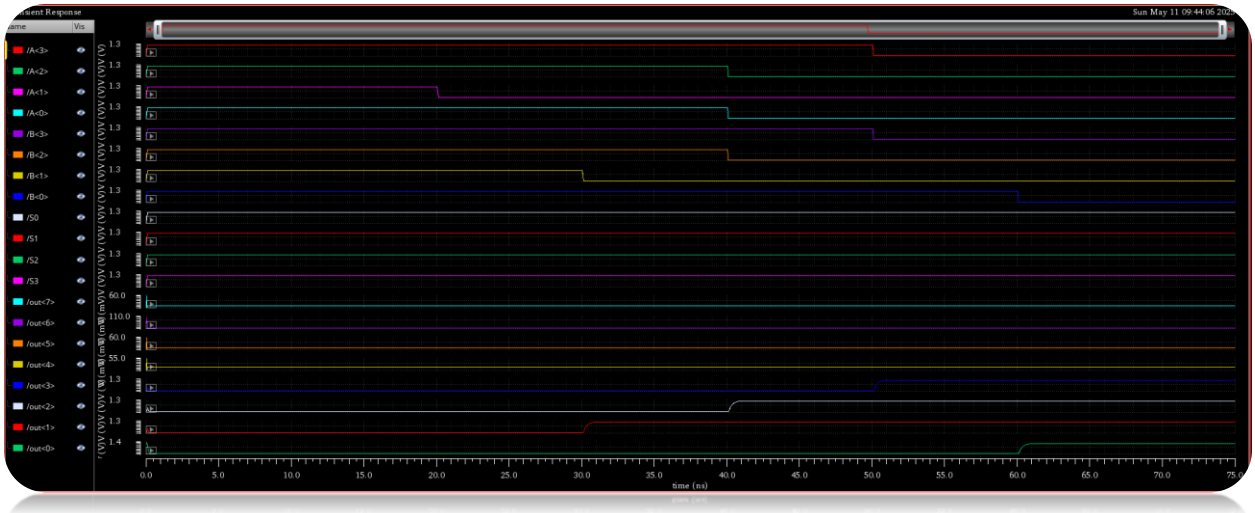
XNOR



NAND



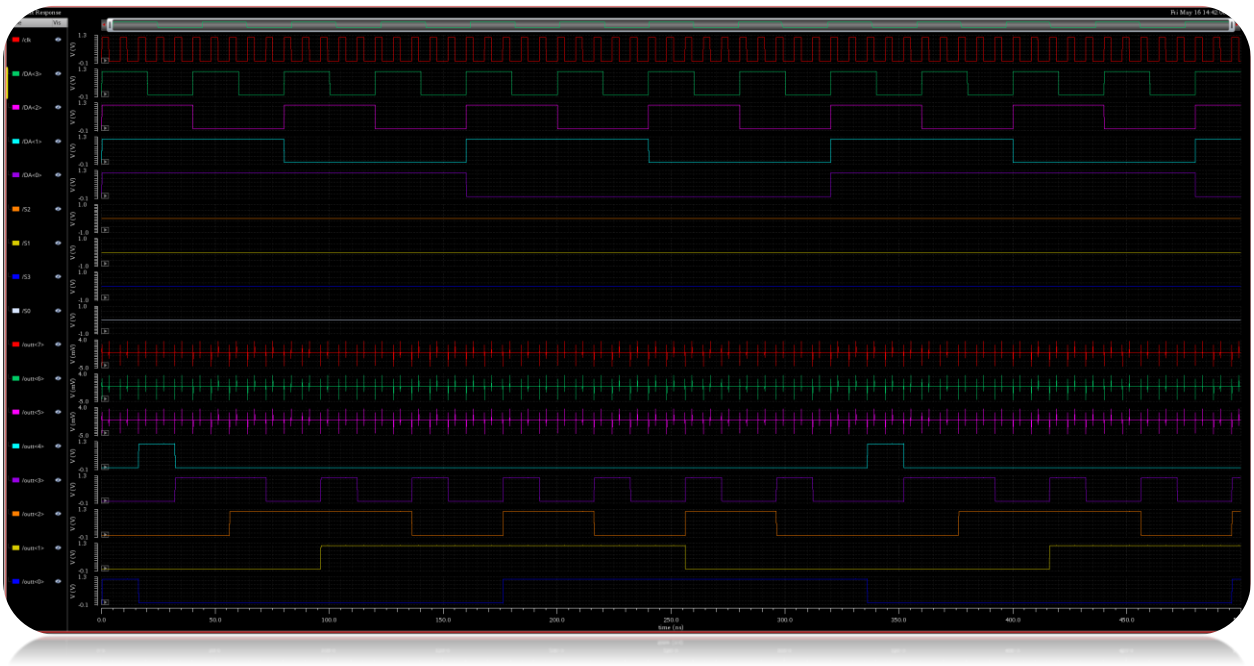
NOR



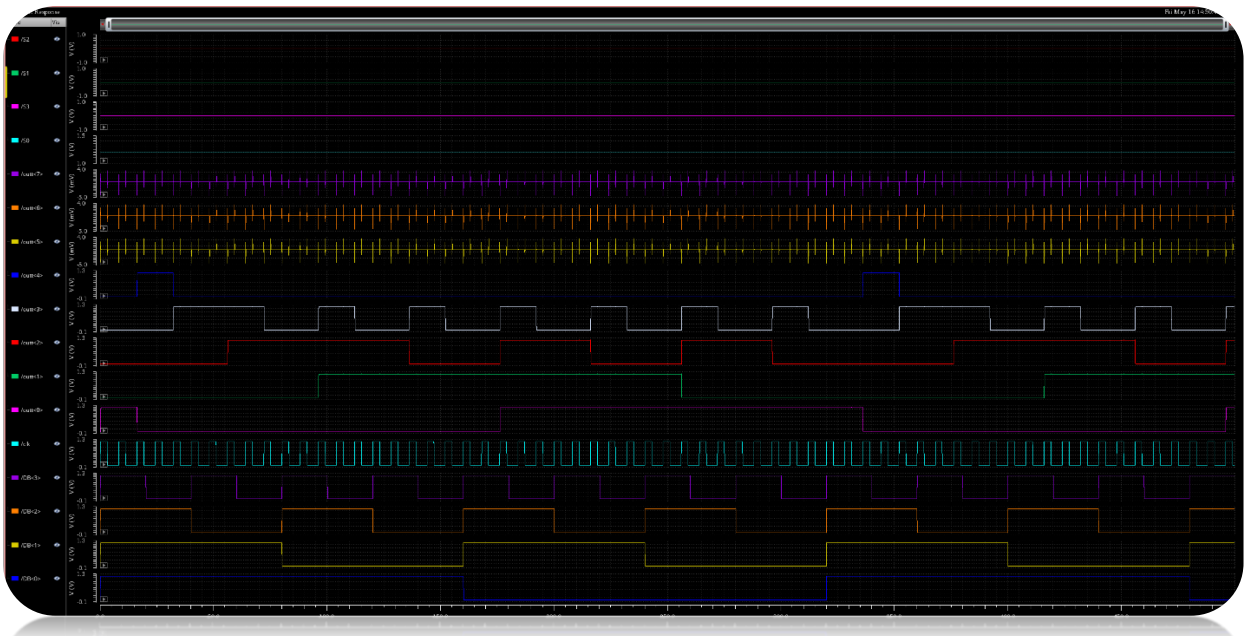
Simulation: Transient simulations with the flip-flops at max f(340 MHz)

Arithmetic unit

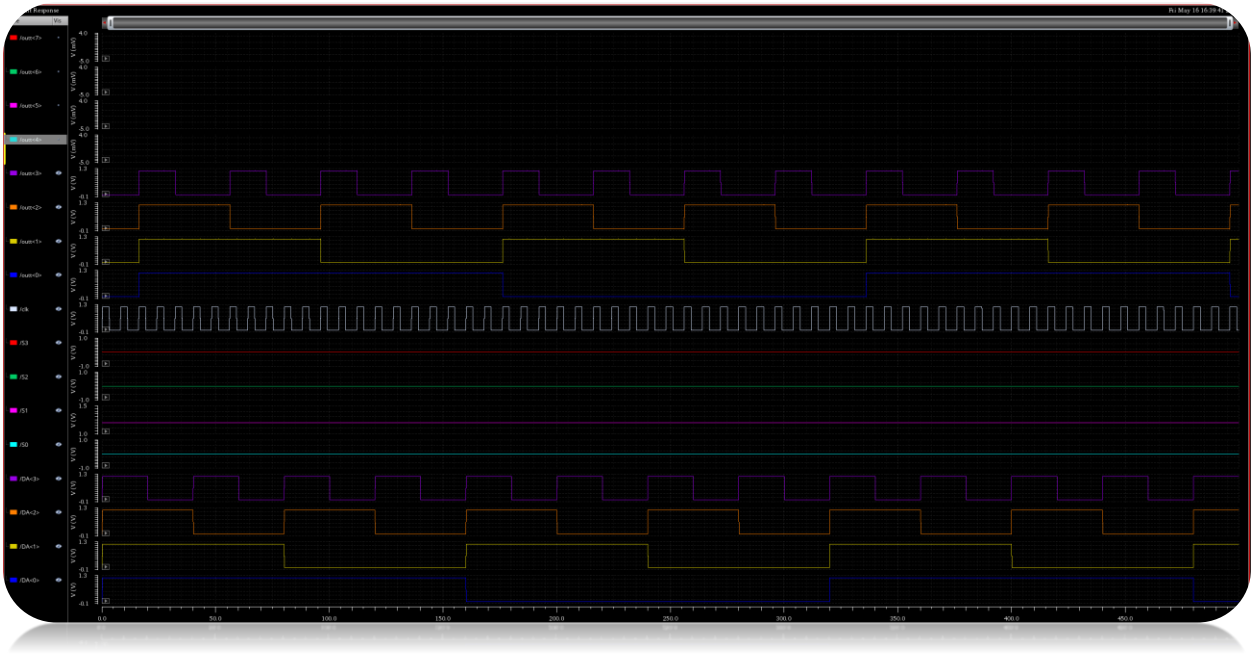
Increment a



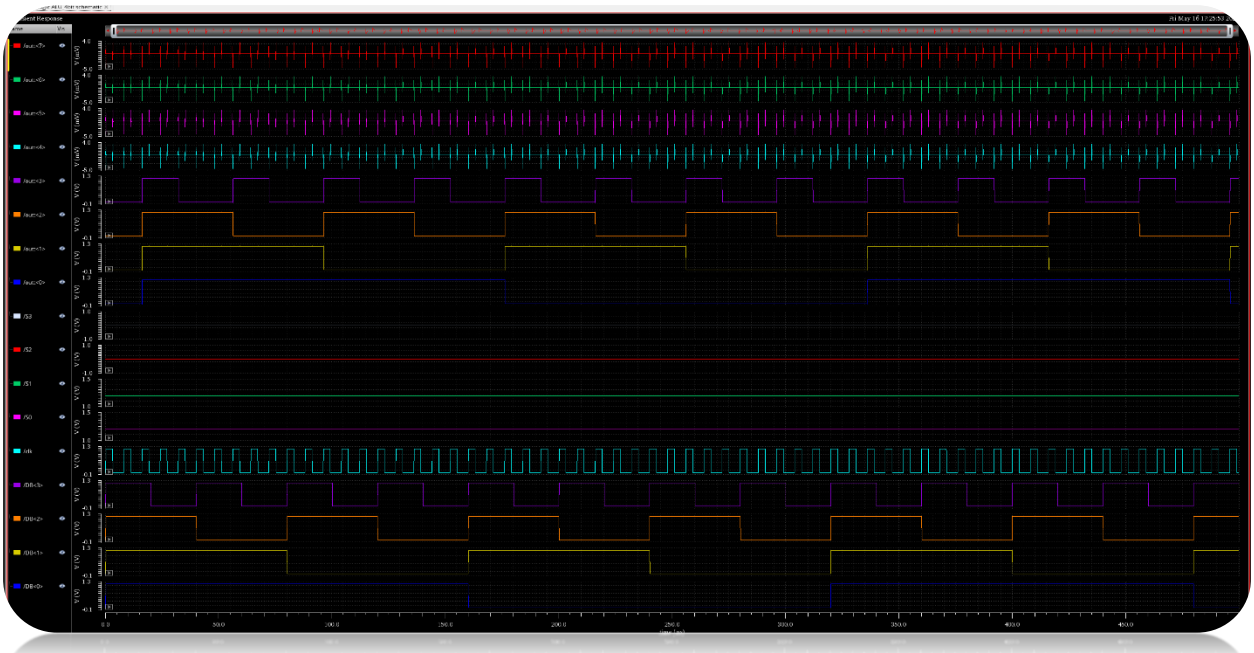
Increment b



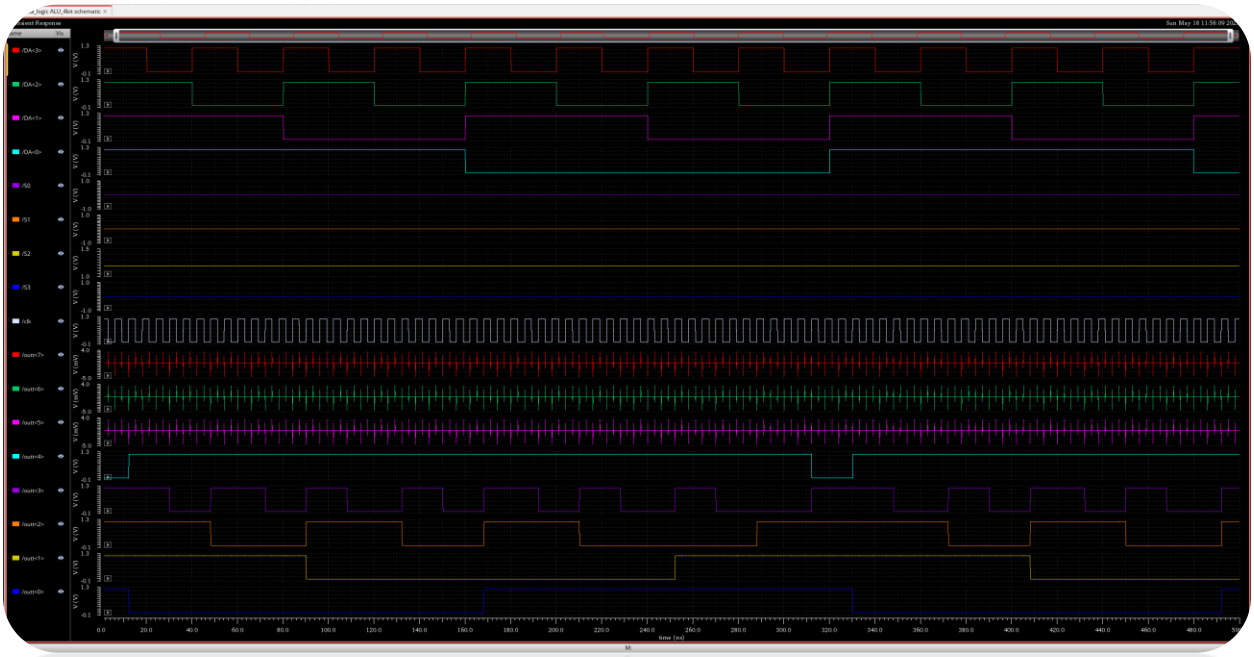
Transfer a



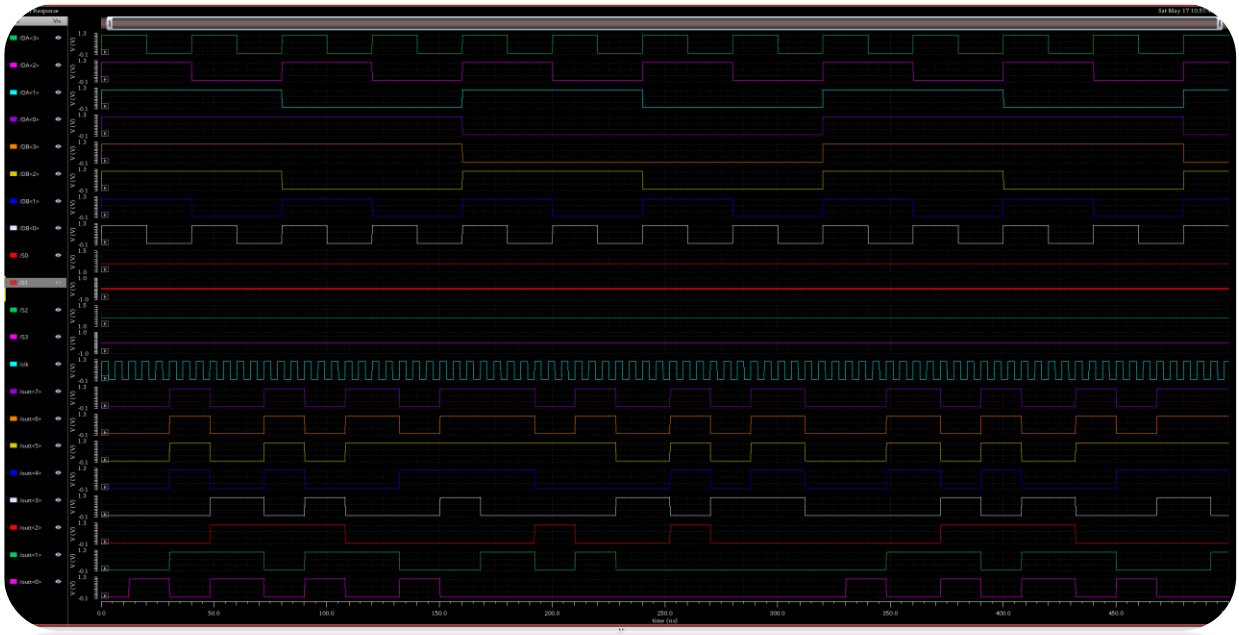
Transfer b



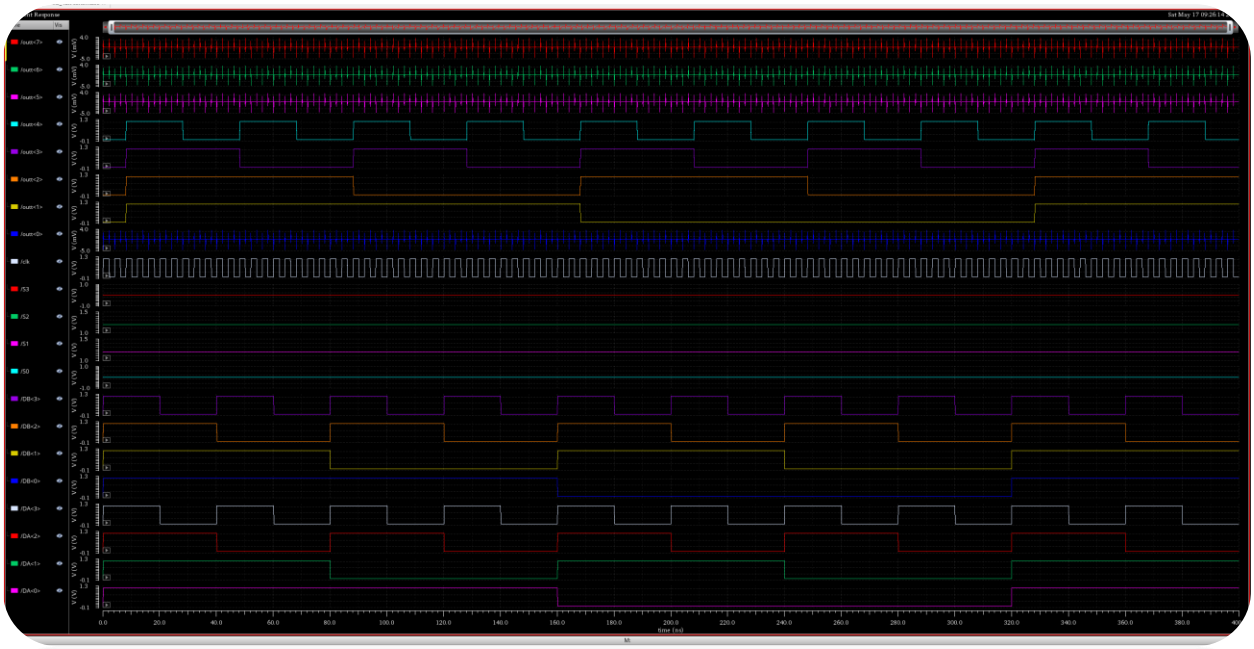
Decrement a



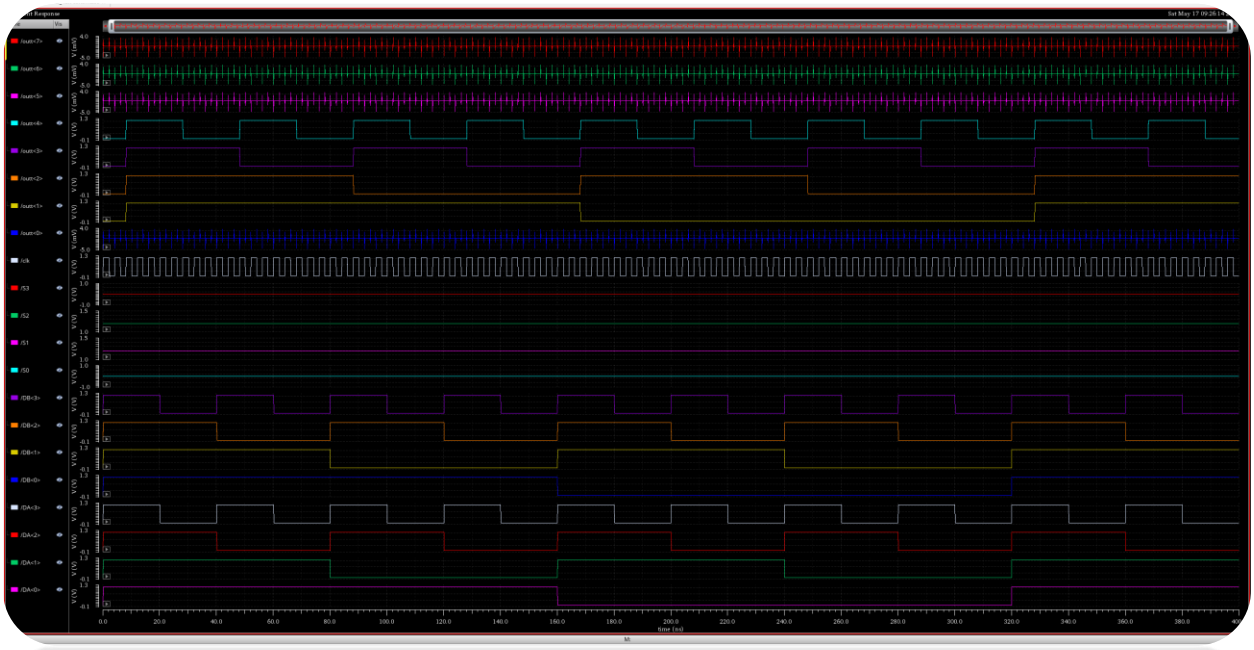
Multiply a and b



Add a and b

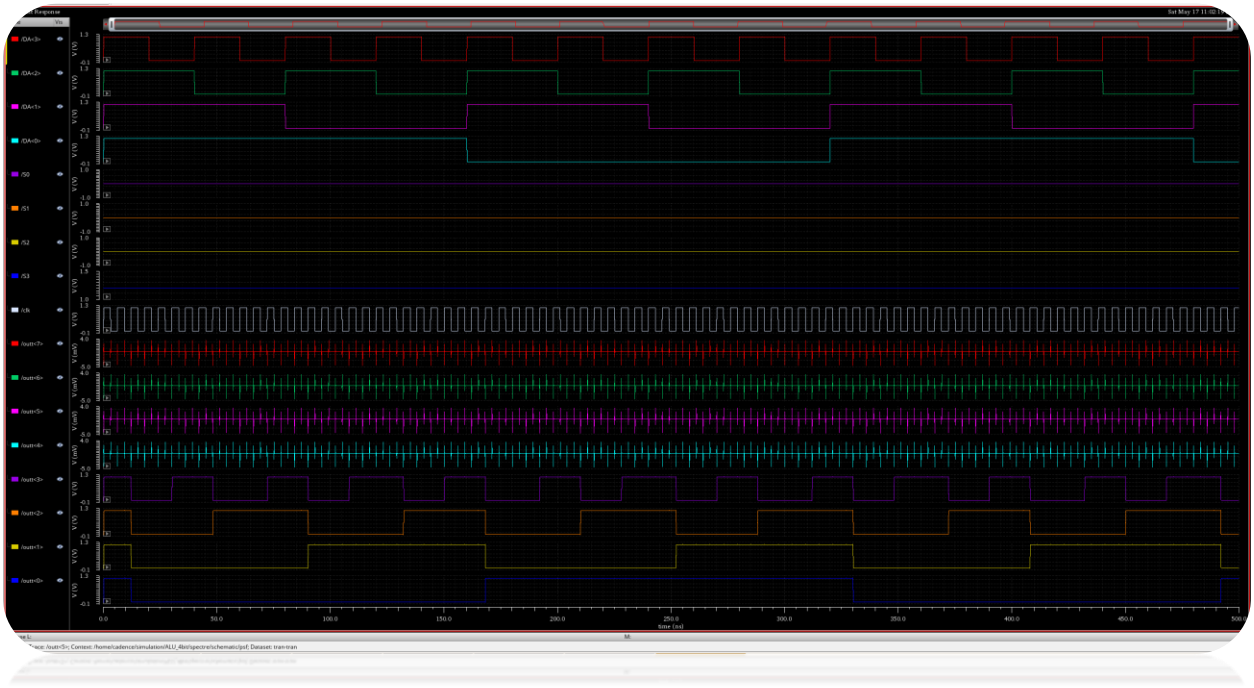


Subtract a and b

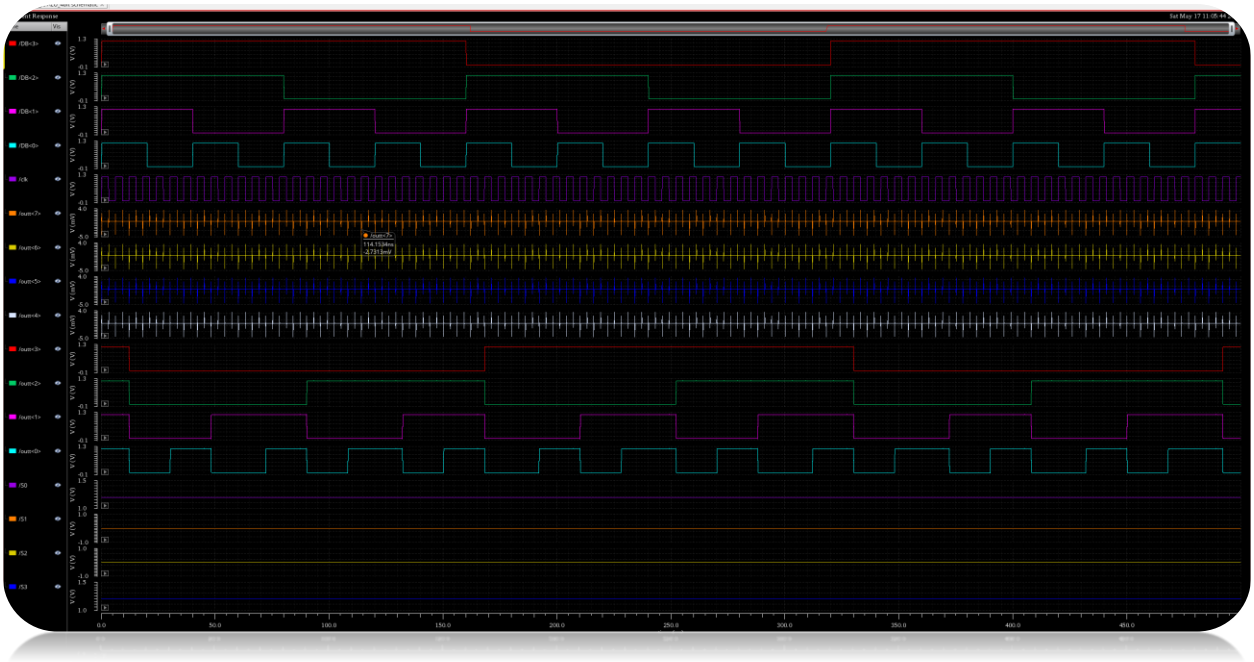


Logical unit

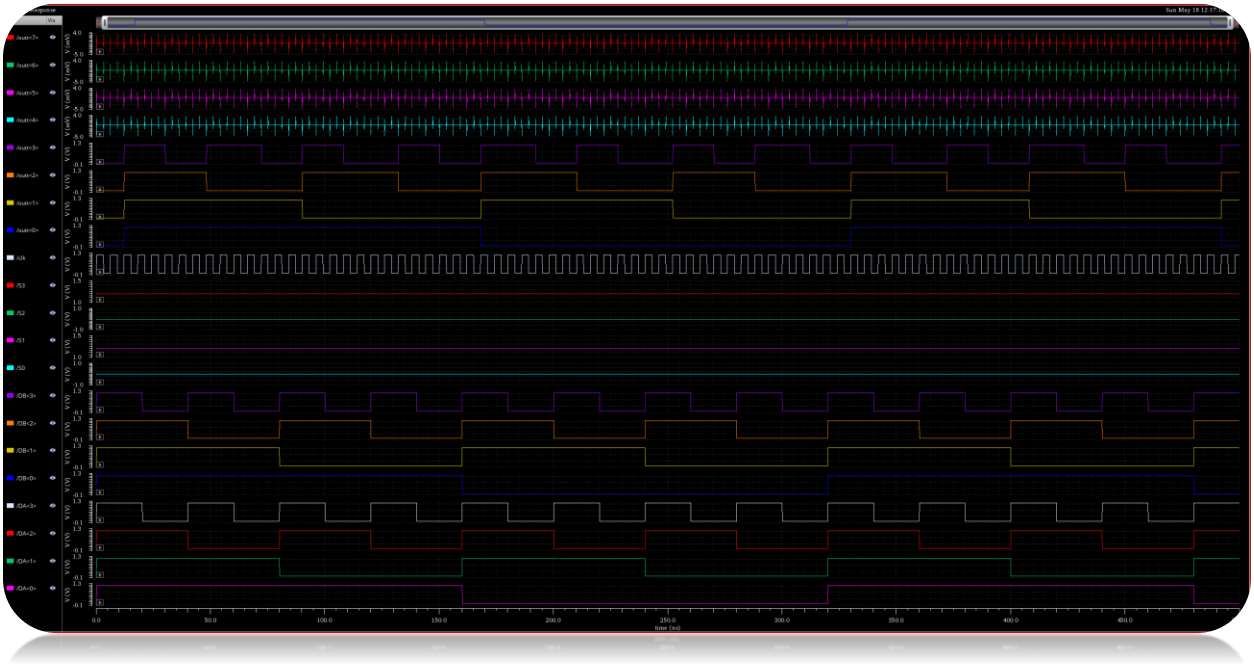
Complement a



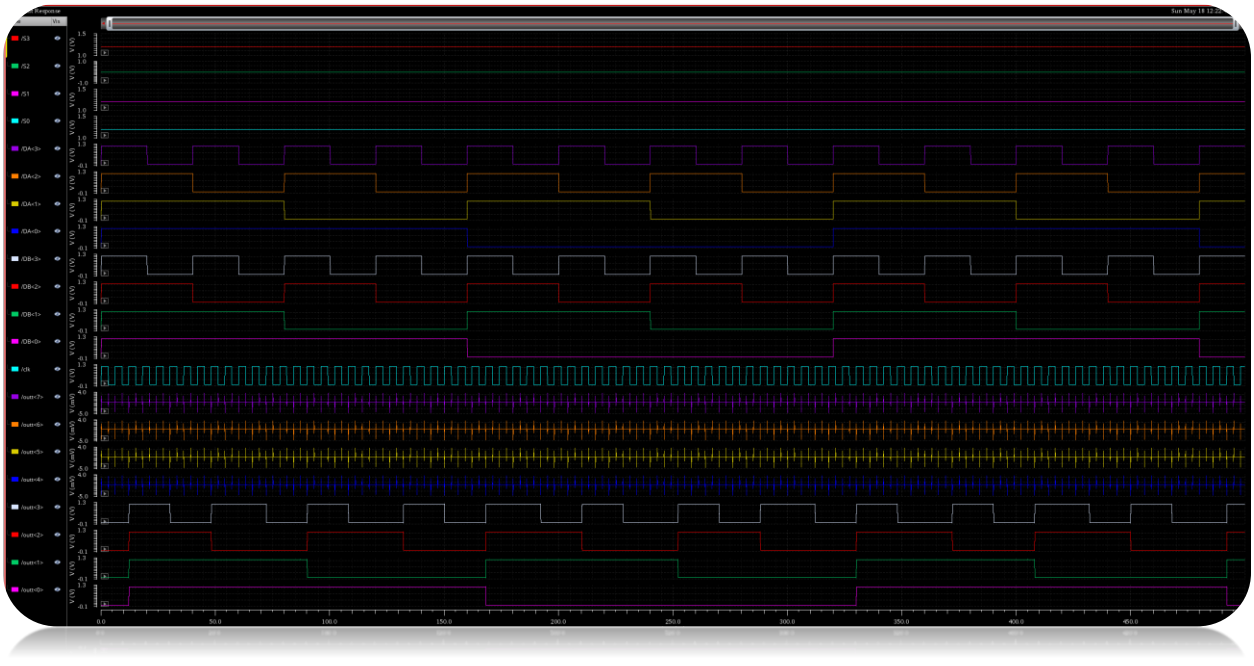
Complement b



AND



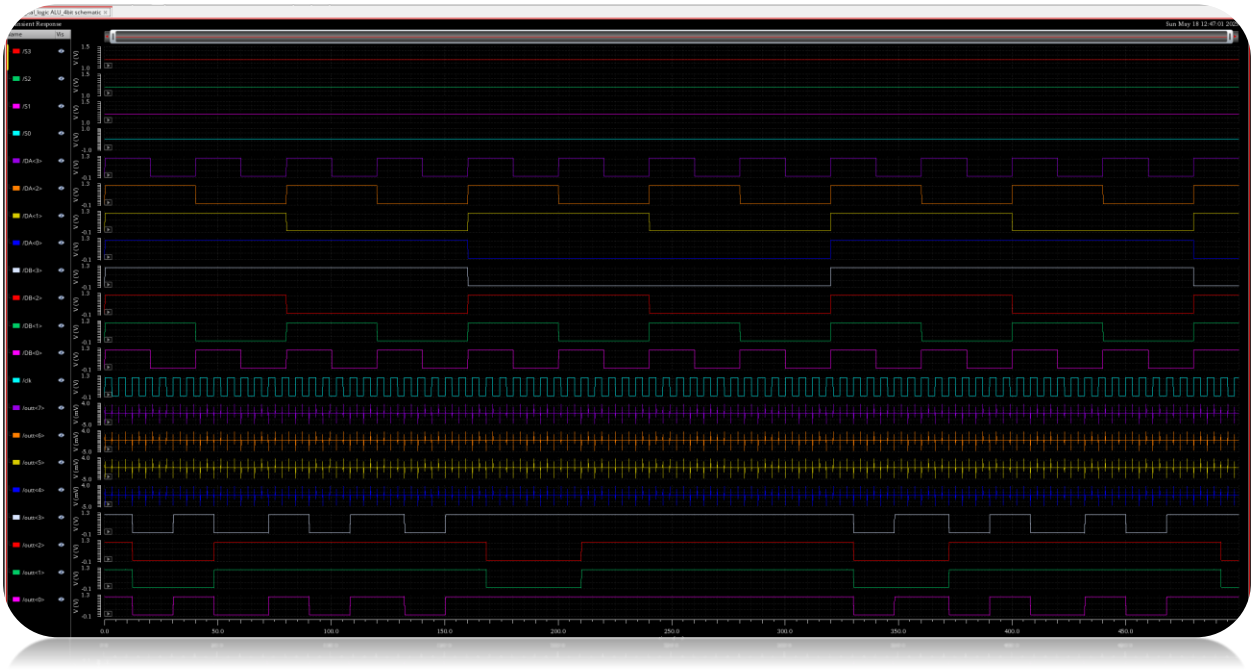
OR



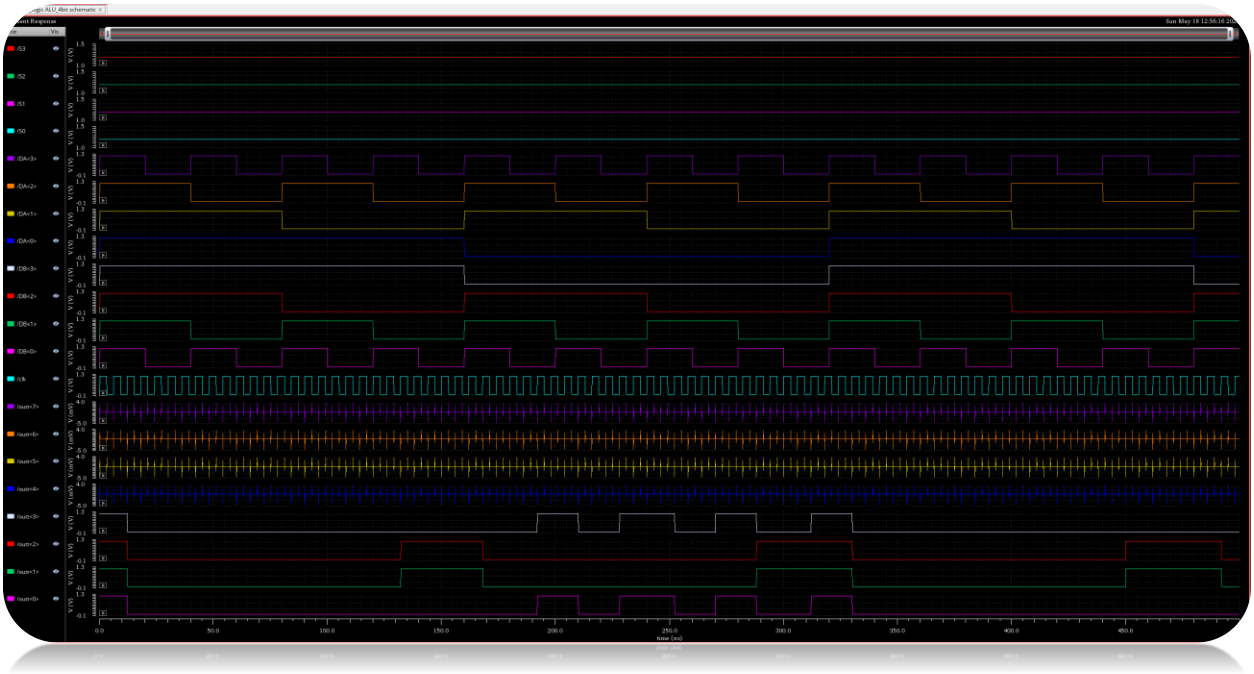
XOR

XNOR

NAND

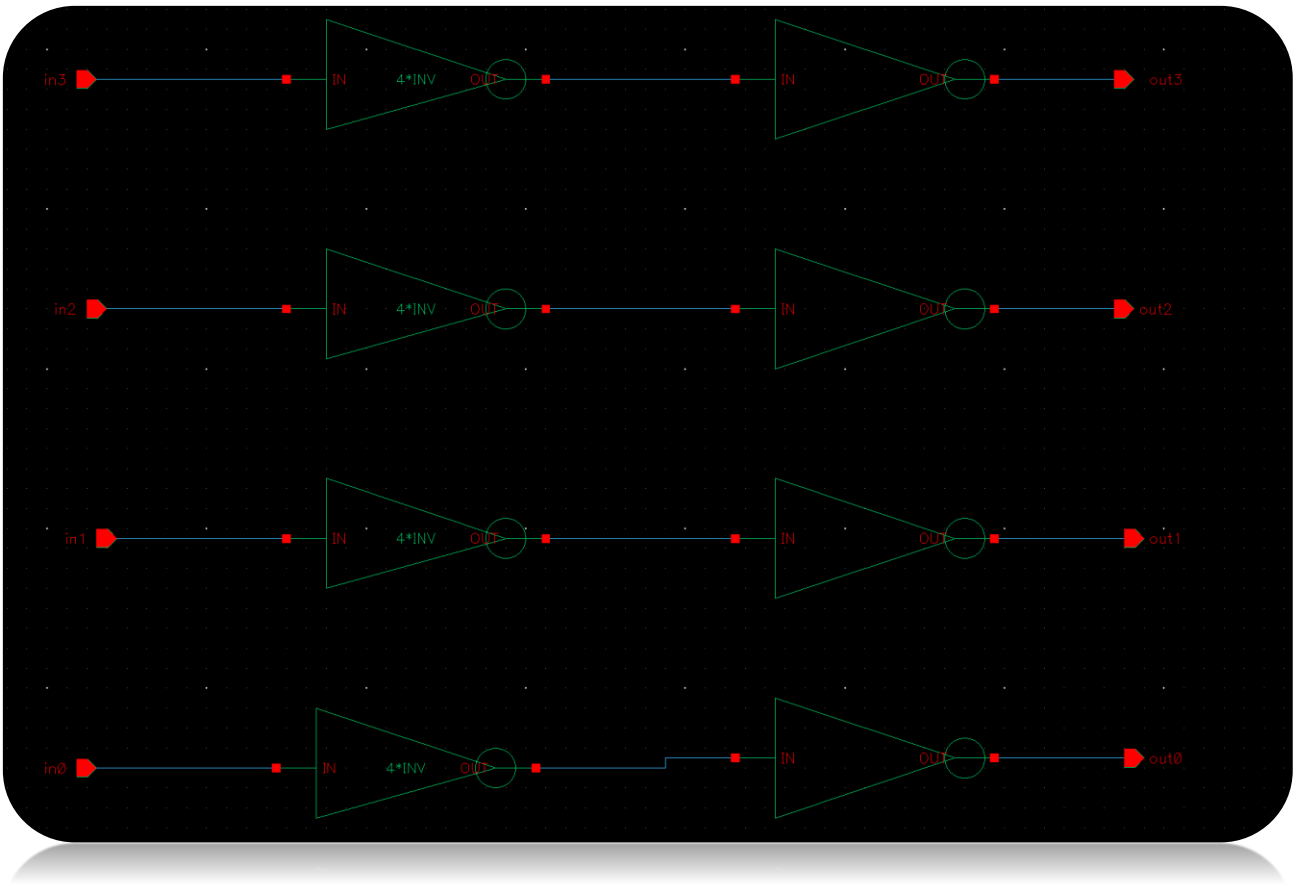


NOR

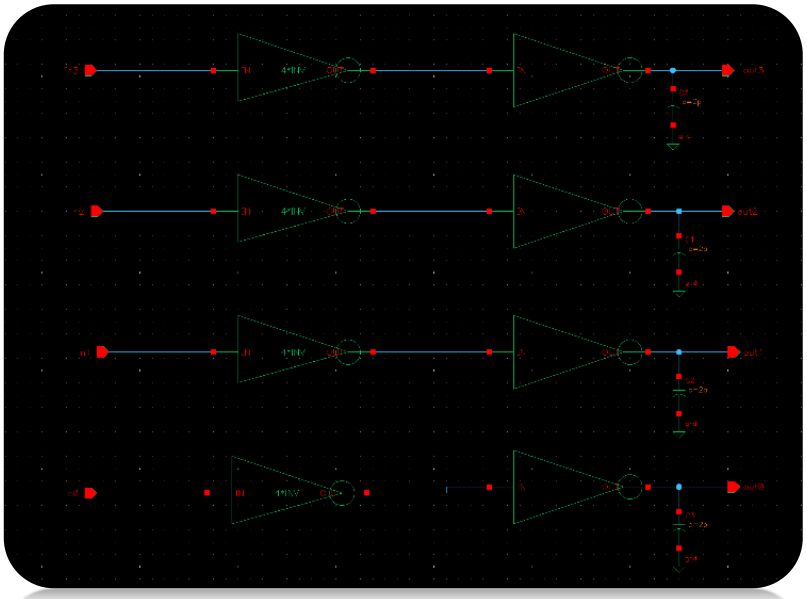


Regeneration stage

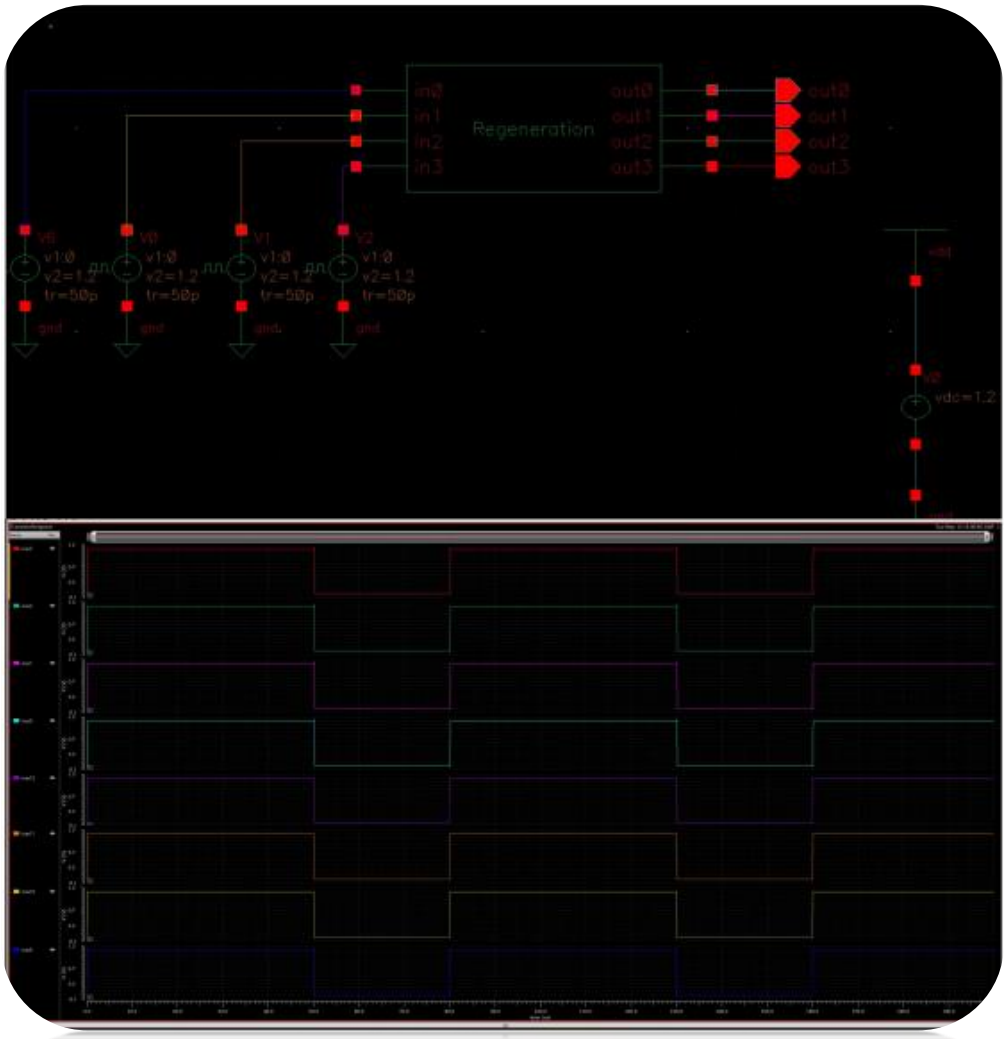
A regeneration stage block may be needed for different reasons in this circuit design, as the ALU has a relatively large input capacitance while the previous stage can not drive nor charge this large capacitance on its own so a regeneration stage with a fan-out of 4 had to be added as an intermediary phase between the ALU and previous stage. This regeneration block was made by adding a 4*inv in series with a 16*inv using our designed reference inverter mentioned previously. The delay of the regeneration stage on its own is negligible or almost nonexistent (note: I may add/remove to this part when re-assessing calculations)



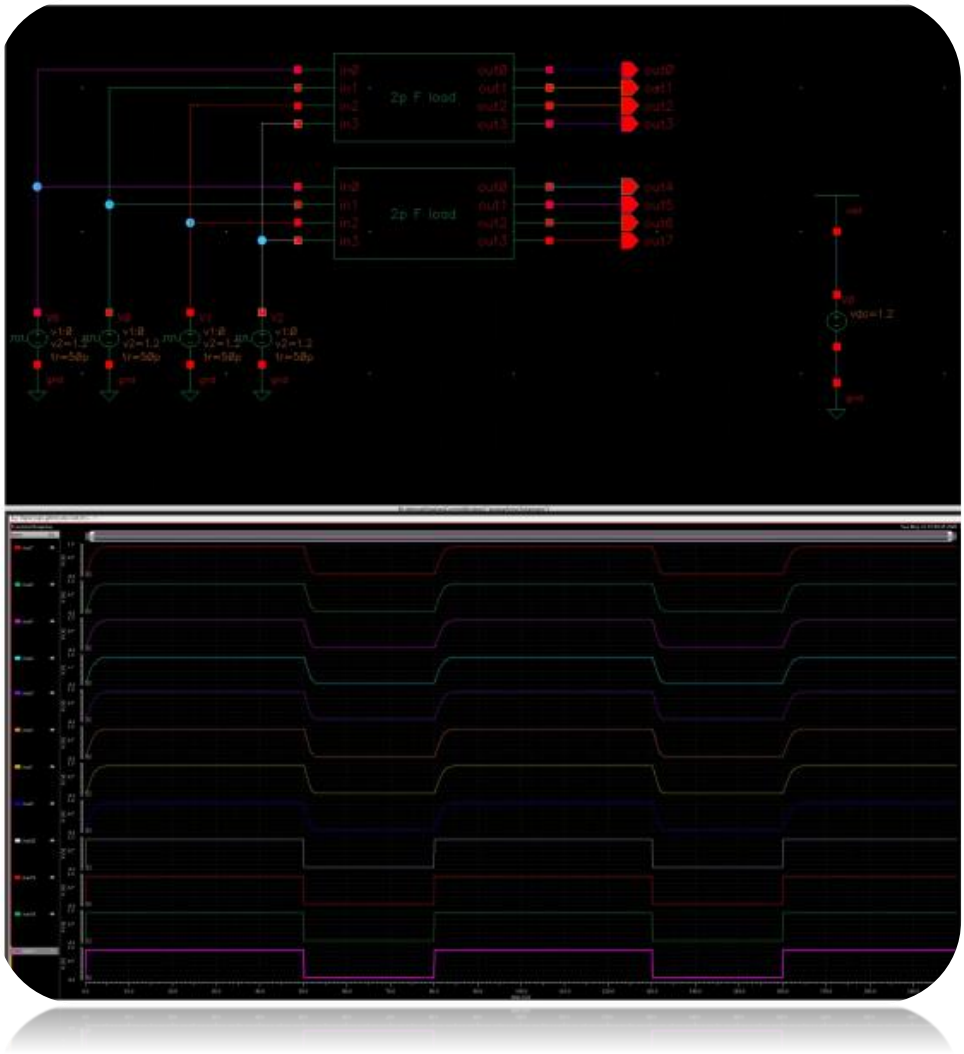
After taking the output from the ALU, the wires are connected to two load blocks each having 4 inputs and 4 outputs to accommodate for the total 8 outputs of our ALU. This load block is very similar in design on circuit level to the regeneration block with just the addition of 2 picoFarad capacitors at the output terminals.



Regeneration test bench:

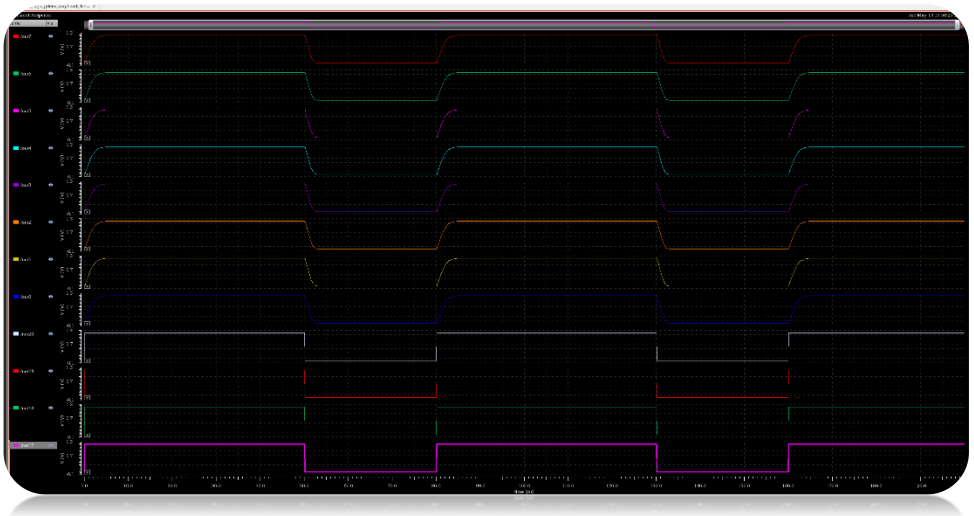


Load Test Bench:



Another waveform with varying inputs:

In this test bench, each input was adjusted to have different pulse width to test the change of individual inputs separately and its effect on the output.



Calculations

TP logic output bit number 7

T_{pcq} = 0.1 nano seconds

T_{plogic} = 1.85 nano seconds

T_{su} = 1 nano seconds

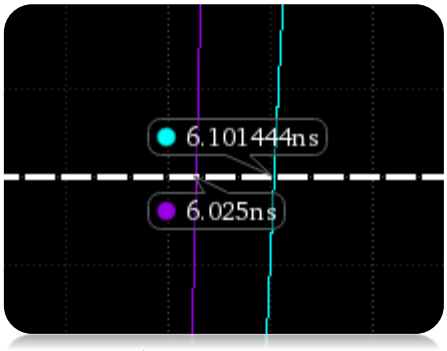


Figure 1: Tpcq

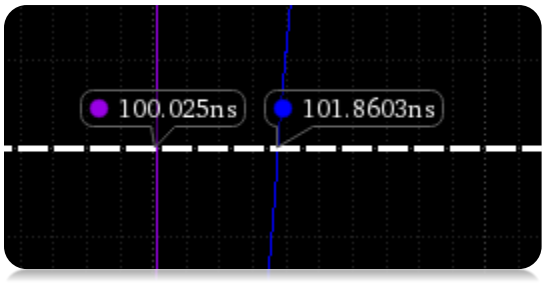


Figure 3: Tplogic

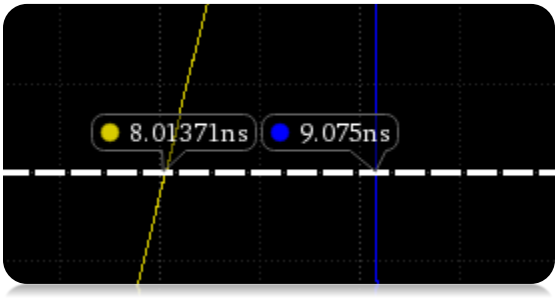


Figure 2: Tsu

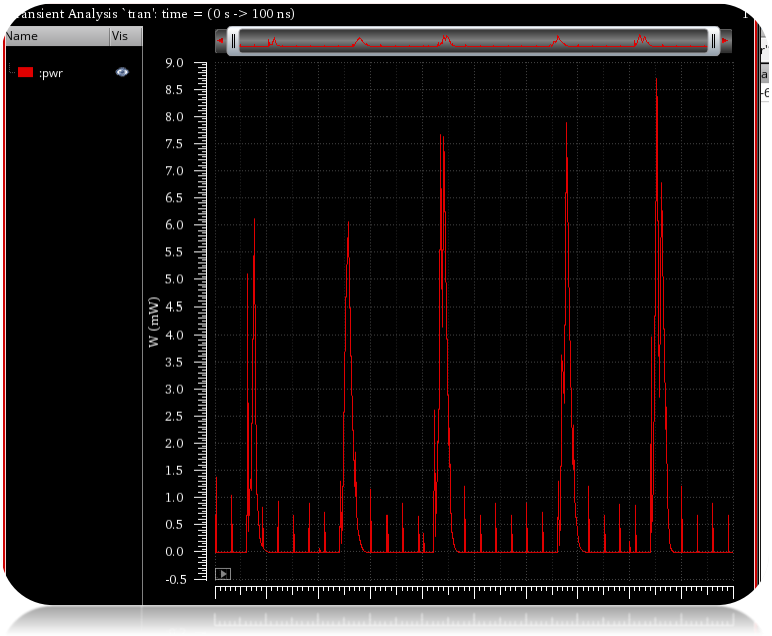
$$T_{clk} \geq t_{pcq} + t_{p\log ic} + t_{su} = 2.95 \text{ nS}$$

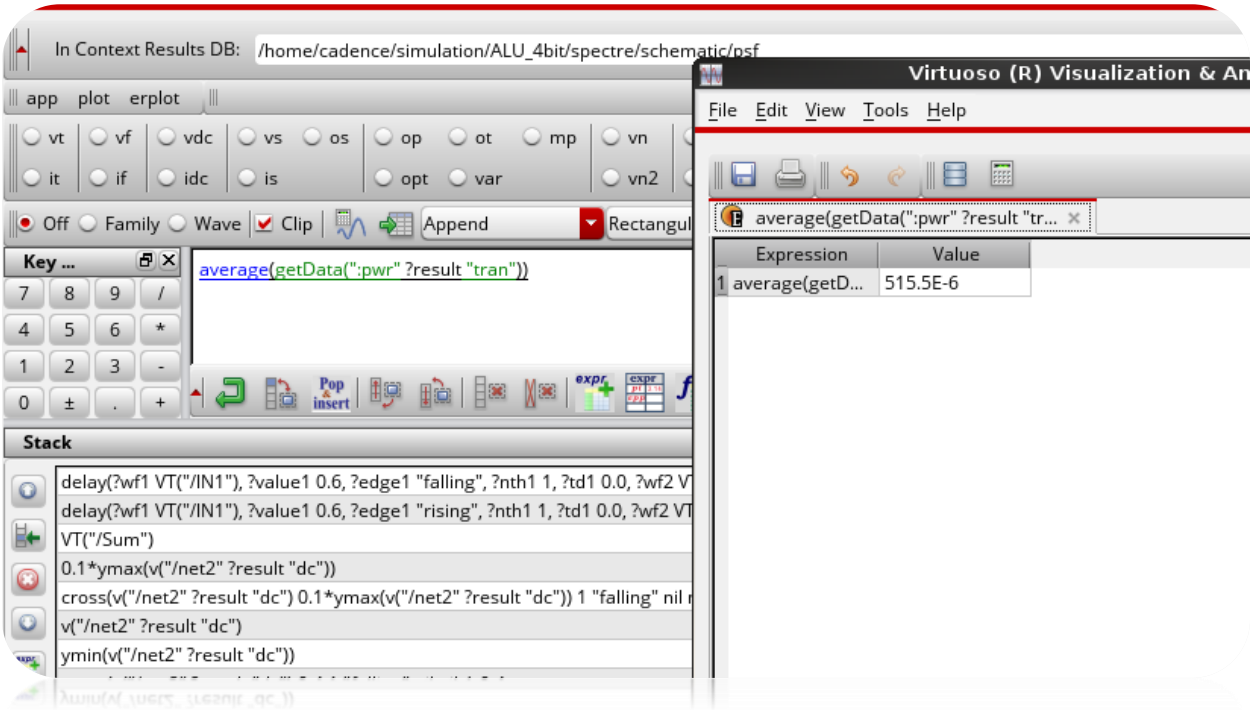
$F_{max} = 1/T_{CLK} \approx 340$ MHz

For maximum delay, selectors were forced to do the multiplication operation and inputs were configured as that the A input changes from -8 to +7 and input B stays at +7 the whole time as it was found this is the worst case in this ALU design.

Tpcq was found as the time between clock reaching 50% and the output waveform reaching the same 50% while the Tsu was found as the minimum amount of time required for the input to be stable before the clock cycle.

Power:





Avg power = 0.515 mW

Bonus: FPGA Implementation of the semicustom implementation

The objective of implementing the ALU design on an **FPGA** is to **validate the hardware functionality** of our Verilog-based ALU design through real-time operation and visualization. This **semi-custom implementation** demonstrates the transition from simulation to a working physical system and provides hands-on verification of design logic.

Target Platform and Tools

- **Target FPGA Board**: Intel **Cyclone IV EP4CE6E22**
- **Design Tools Used**: Intel **Quartus Prime**
- **Source Code**: Verilog HDL

▶️ **Project files**: Drive Link

Implementation Details

- **Inputs:**
  - **4 LEDs** on the FPGA are used as **selector inputs** to choose the desired ALU operation (in binary)
  - **Selectors** changes through the counter on Verilog code.
  - Two 4-bit operands **A and B** are assigned using input pins through Quartus (set in the Pin Planner).
- **Outputs:**
  - The ALU's output is displayed on the **7-segment display** of the FPGA.
  - The **operation mode** is selected based on the **most significant bit (MSB)** of the 4-bit selector (`SEL[3]`):
    - `SEL[3] = 0`: **Arithmetic operations** (e.g., addition, subtraction)
    - `SEL[3] = 1`: **Logical operations** (e.g., AND, OR, XOR)
  - **For arithmetic operations**:
    - **3 digits** show the **magnitude** of the 4-bit result in **decimal**.
    - **1 digit** indicates the **sign** (positive or negative) of the result.
  - **For logical operations**:
    - The result is displayed as a **4-bit binary** value using the **7-segment display**, with one digit per bit (or grouped as needed).

- **Functionality:**
  - The ALU performs operations such as addition, subtraction, AND, OR, etc., based on the selector input.
  - Real-time results are visible on the 7-segment display, enabling easy debugging and validation.

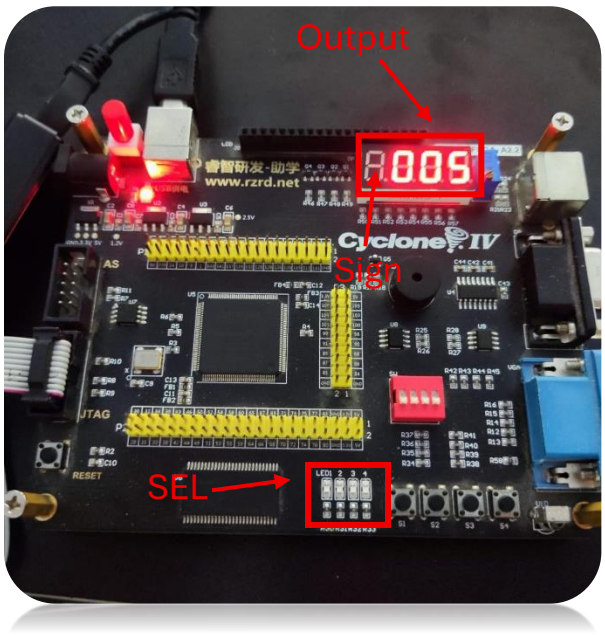
Verification and Testing

- The implemented design was verified through **physical testing on the FPGA board**.
- A demonstration video showing the ALU in operation is available here:
  ▶ **Demo Video**: Drive Link
- Additional screenshots and labeled images of selector configuration and output display will be included below.

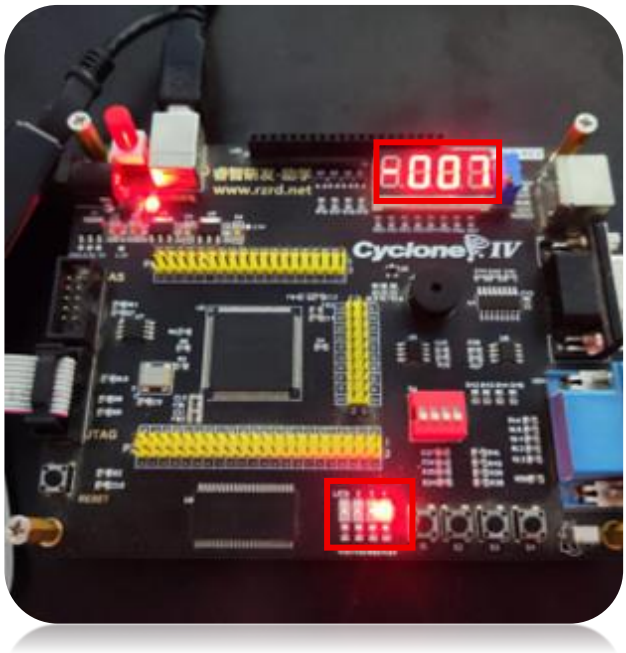
A=0100, B=1000

A=4, B=-8

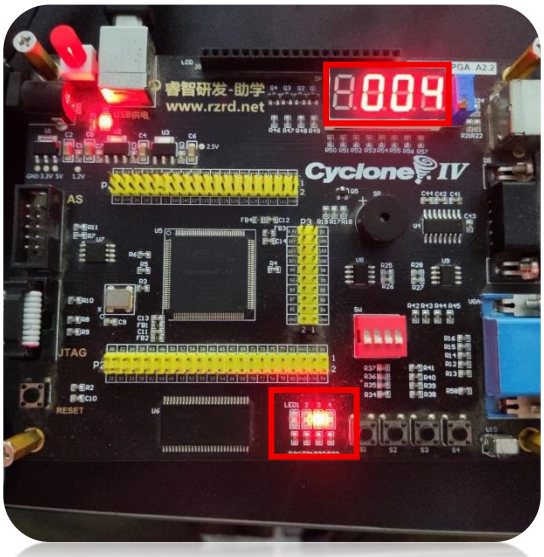
1. SEL=0000 (Increment A)



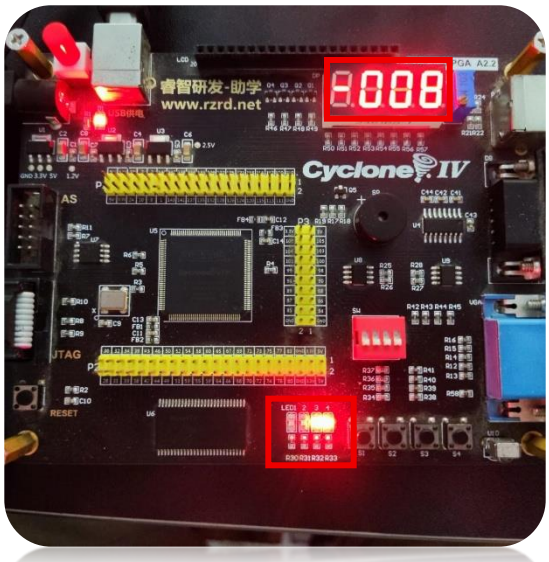
2.SEL=0001 (Increment B)



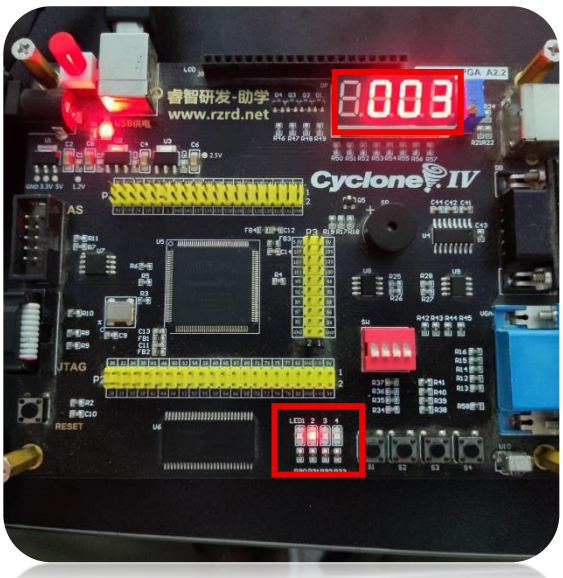
3. SEL=0010 (Transfer A)



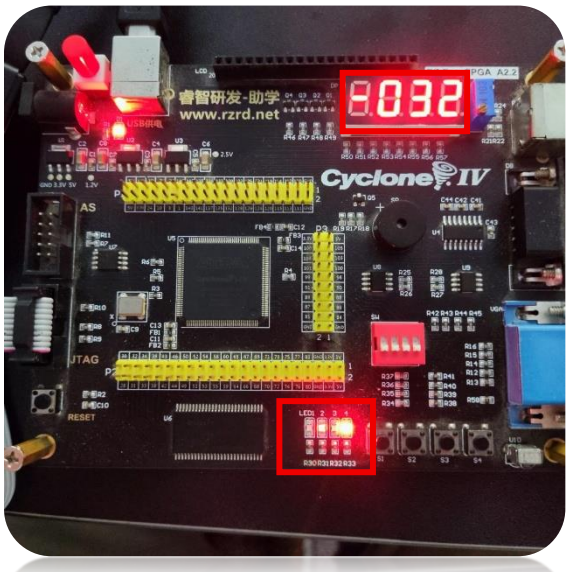
4.SEL=0011 (Transfer B)



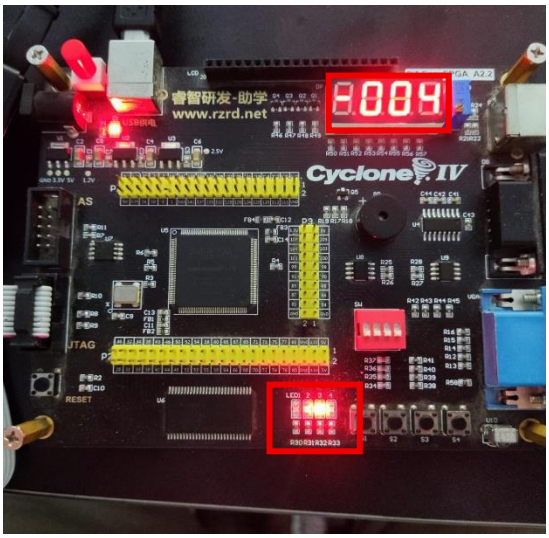
5. SEL =0100 (Decrement A)



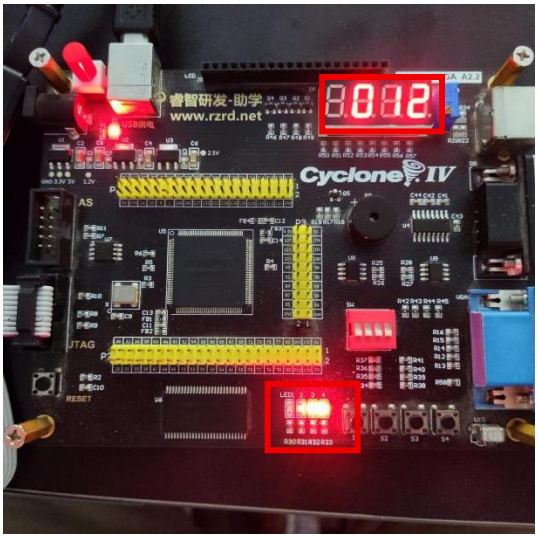
6. SEL= 0101 (Multiply A and B)

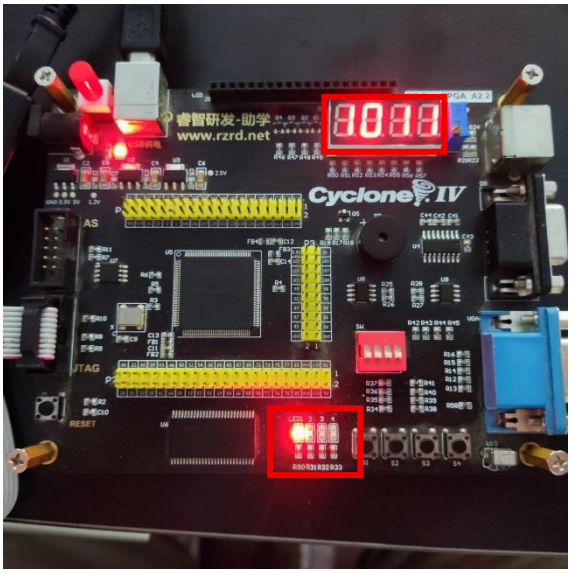


7. SEL=0110 (Add A and B)

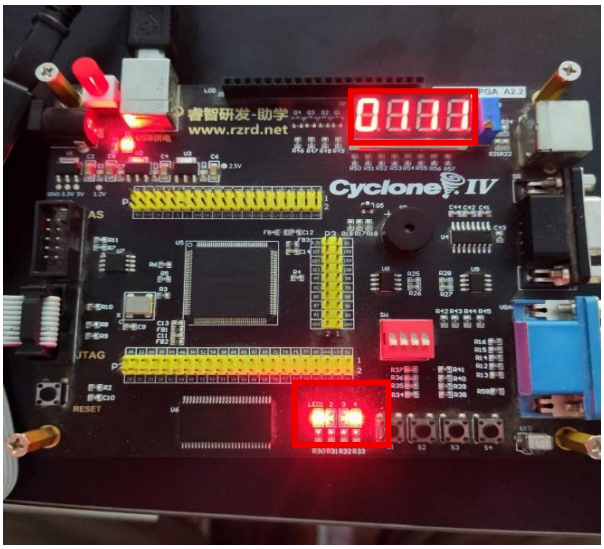


8.SEL=0111 (Subtract A and B)

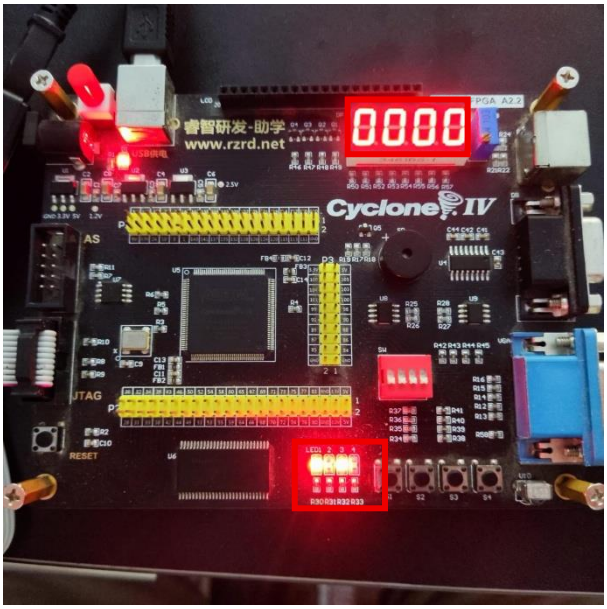




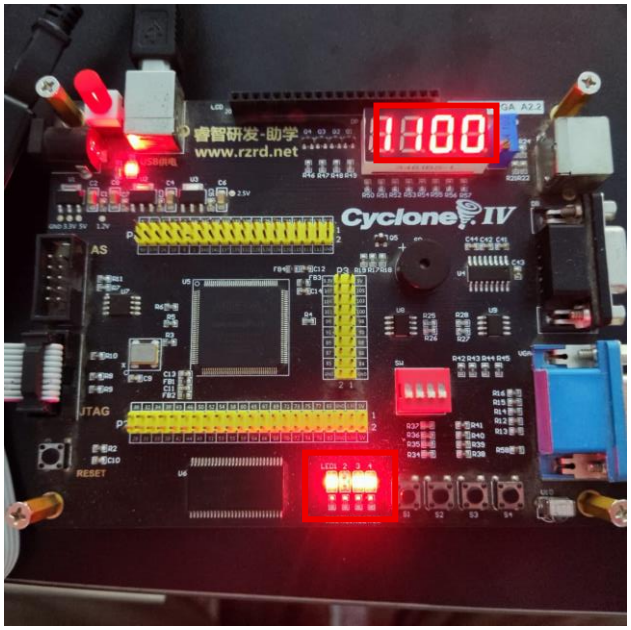
9. SEL=1000 (~A)



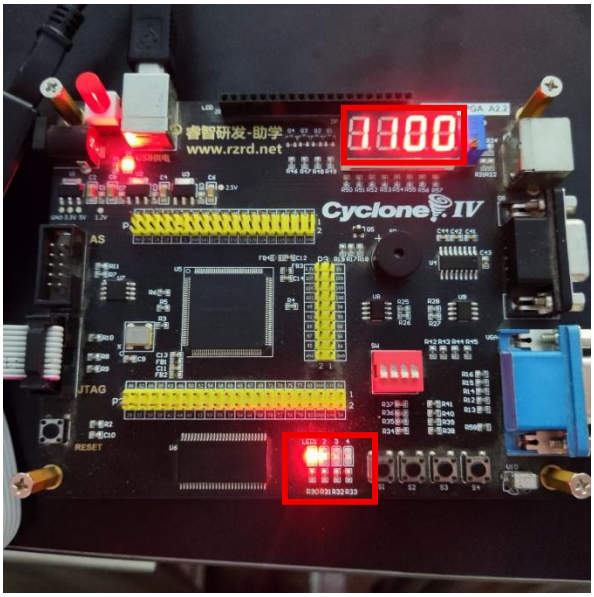
10. SEL=1001 (~B)



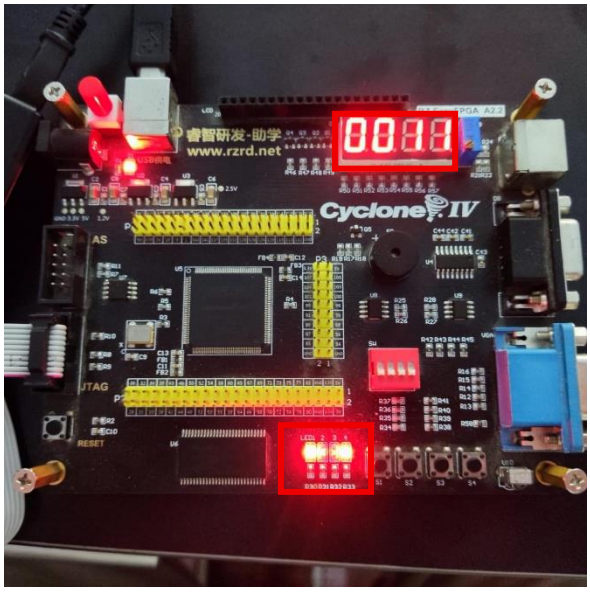
11. SEL=1010 (AND A & B)



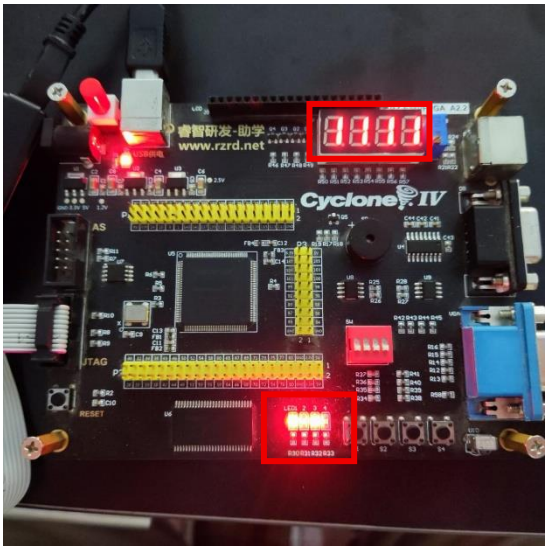
12. SEL=1011(OR A | B)



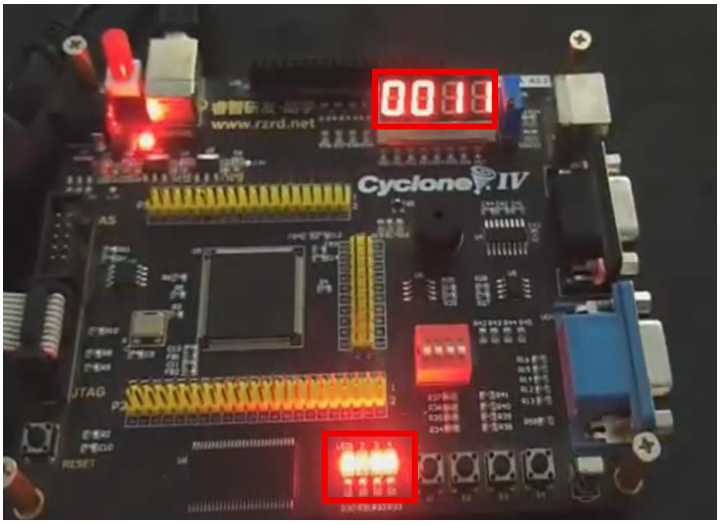
13. SEL=1100 (XOR A^B)



14. SEL=1101 (XNOR ~(A^B))



15.SEL=1110 (NAND ~(A&B))



16. SEL=1111 (NOR ~(A|B)

Conclusion of FPGA Section

The FPGA implementation successfully demonstrated the **practical functionality** of the 4-bit ALU design. The use of real hardware allowed us to **verify the correctness of our Verilog code** beyond simulation. Although this design targets small bit-width operations, it reflects a complete digital system pipeline—from logic design to hardware realization. This semi-custom implementation confirms that the design is **functional, reliable, and adaptable** for future extensions on larger FPGAs.

GitHub Repository

You can access the complete Verilog source code, testbenches, and Quartus project files for this ALU design through the GitHub repository below:

Repository Link

This repository includes:

- Verilog modules for Arithmetic and Logical Unit
- Modeling using C++
- Testbenches for simulation
- FPGA configuration files for Quartus (Cyclone IV)
- Documentation and block diagrams