# ALU

Muhammed Adel Ahmed

# Project description

## About The Project:

It is required to design the ALU shown in Fig.1. This ALU can execute arithmetic and logical operations. The operation of the ALU is described by table 1. The output (arithmetic or logical) is selected by the MSB of the selection line, while the required operation is selected by the other 3 bits. It is also required to design flip-flops at the inputs and outputs of the ALU.

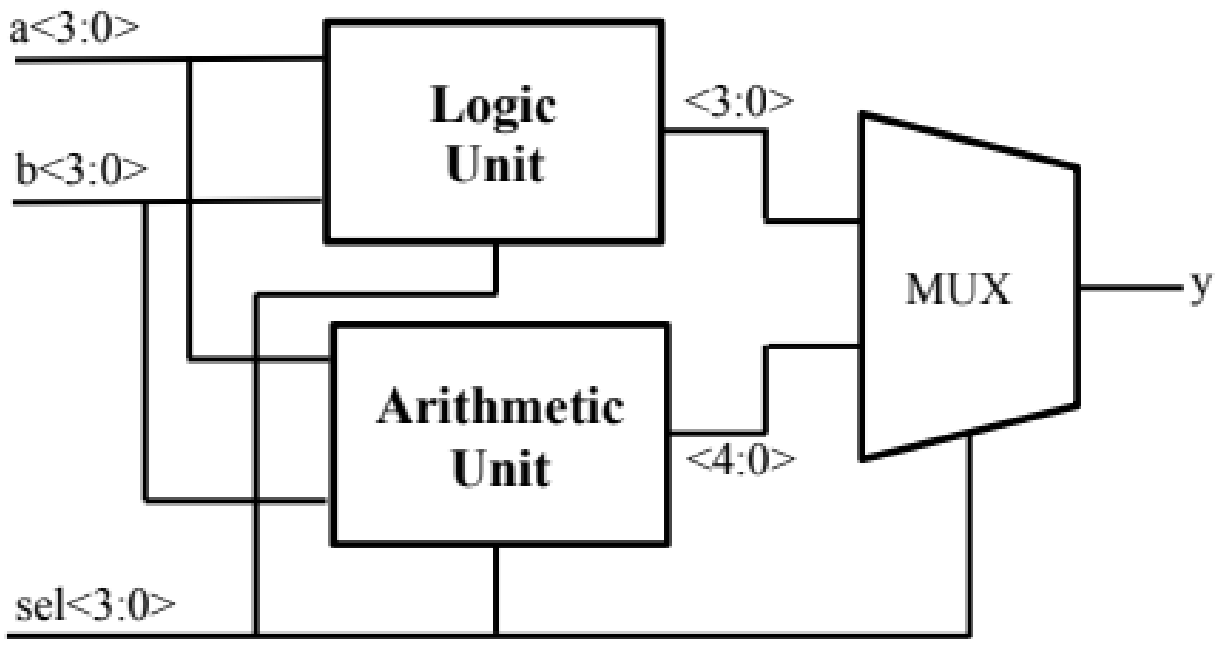



**Fig. 1: ALU Block Diagram**

**Table 1: ALU Operations**

| Sel | Operation | Unit |
|---|---|---|
| 0000 | Increment a | Arithmetic |
| 0001 | Decrement a | |
| 0010 | Multiply a by 2 | |
| 0011 | Increment b | |
| 0100 | Decrement b | |
| 0101 | Multiply b by 2 | |
| 0110 | Add a and b | |
| 0111 | Multiply a by 4 | |
| 1000 | Complement a (1's complement) | Logic |
| 1001 | Complement b (1's complement) | |
| 1010 | AND | |
| 1011 | OR | |
| 1100 | XOR | |
| 1101 | XNOR | |
| 1110 | NAND | |
| 1111 | NOR | |

# 2. Verilog Code

## ALU Code:

### 1- *signed arithmetic and unsigned logic:*

```
module alu(
input [3:0] A,      // 4 – bit input operand A
input [3:0] B,      // 4 – bit input operand B
input [3:0] Sel,    // 4 – bit selection signal (Sel[3] determines arithmetic or logical operations)
output reg [5:0] Y  // 6
– bit output result (result width allows for extended range for arithmetic and logical results)
);

always @(*) begin
// Check the most significant bit of Sel (Sel[3])
// If Sel[3] is 0, perform arithmetic operations
if (!Sel[3]) begin
// Use the lower three bits of Sel to choose the arithmetic operation
case({Sel[2], Sel[1], Sel[0]})
3'b000 : Y <= $signed(A) + 4'b0001;  // Increment A by 1
3'b001 : Y <= $signed(A) – 4'b0001;  // Decrement A by 1
3'b010 : Y <= $signed(A) << 2;       // Shift A left by 2 (multiply A by 4)
3'b011 : Y <= $signed(B) + 4'b0001;  // Increment B by 1
3'b100 : Y <= $signed(B) – 4'b0001;  // Decrement B by 1
3'b101 : Y <= $signed(B) << 2;       // Shift B left by 2 (multiply B by 4)
3'b110 : Y <= $signed(A) + $signed(B);// Add A and B
3'b111 : Y <= $signed(A) << 4;       // Shift A left by 4 (multiply A by 16)
endcase
end
// If Sel[3] is 1, perform logical operations
else begin
// Use the lower three bits of Sel to choose the logical operation
case({Sel[2], Sel[1], Sel[0]})
3'b000 : Y <= ~$signed(A);           // Bitwise NOT of A
3'b001 : Y <= ~$signed(B);           // Bitwise NOT of B
3'b010 : Y <= $signed(A) & $signed(B); // Bitwise AND of A and B
3'b011 : Y <= $signed(A) | $signed(B); // Bitwise OR of A and B
3'b100 : Y <= $signed(A) ^ $signed(B); // Bitwise XOR of A and B
3'b101 : Y <= ~($signed(A) ^ $signed(B)); // Bitwise XNOR of A and B
3'b110 : Y <= ~($signed(A) & $signed(B)); // Bitwise NAND of A and B
3'b111 : Y <= ~($signed(A) + $signed(B)); // Negated sum of A and B (two's complement)
endcase
end
end
```

*endmodule*

---

# 2-signed arithmetic and signed logic (Bonus):

```
module alu_2(
    input signed [3:0] A,       // 4-bit input operand A
    input signed [3:0] B,       // 4-bit input operand B
    input signed [3:0] Sel,     // 4-bit selection signal (Sel[3] determines arithmetic or logical operations)
    output reg [5:0] Y   // 6-bit output result (result width allows for extended range for arithmetic and logical
results)
);
always @(*) begin
    // Check the most significant bit of Sel (Sel[3])
    // If Sel[3] is 0, perform arithmetic operations
    if (!Sel[3]) begin
        // Use the lower three bits of Sel to choose the arithmetic operation
        case({Sel[2], Sel[1], Sel[0]})
            3'b000 : Y <= A + 4'b0001;   // Increment A by 1
            3'b001 : Y <= A - 4'b0001;   // Decrement A by 1
            3'b010 : Y <= A << 2;        // Shift A left by 2 (multiply A by 4)
            3'b011 : Y <= B + 4'b0001;   // Increment B by 1
            3'b100 : Y <= B - 4'b0001;   // Decrement B by 1
            3'b101 : Y <= B << 2;        // Shift B left by 2 (multiply B by 4)
            3'b110 : Y <= A + B;// Add A and B
            3'b111 : Y <= A << 4;        // Shift A left by 4 (multiply A by 16)
    endcase
end
// If Sel[3] is 1, perform logical operations
else begin
    // Use the lower three bits of Sel to choose the logical operation
    case({Sel[2], Sel[1], Sel[0]})
        3'b000 : Y <= ~(A);             // Bitwise NOT of A
        3'b001 : Y <= ~(B);             // Bitwise NOT of B
        3'b010 : Y <= (A & B); // Bitwise AND of A and B
        3'b011 : Y <= (A | B); // Bitwise OR of A and B
        3'b100 : Y <= (A ^ B); // Bitwise XOR of A and B
```

```
            3'b101 : Y <= ~(A ^ B); // Bitwise XNOR of A and B
            3'b110 : Y <= ~(A & B); // Bitwise NAND of A and B
            3'b111 : Y <= ~(A + B); // Negated sum of A and B (two's complement)
        endcase
    end
end
endmodule
```

# 3. Test Bench Code

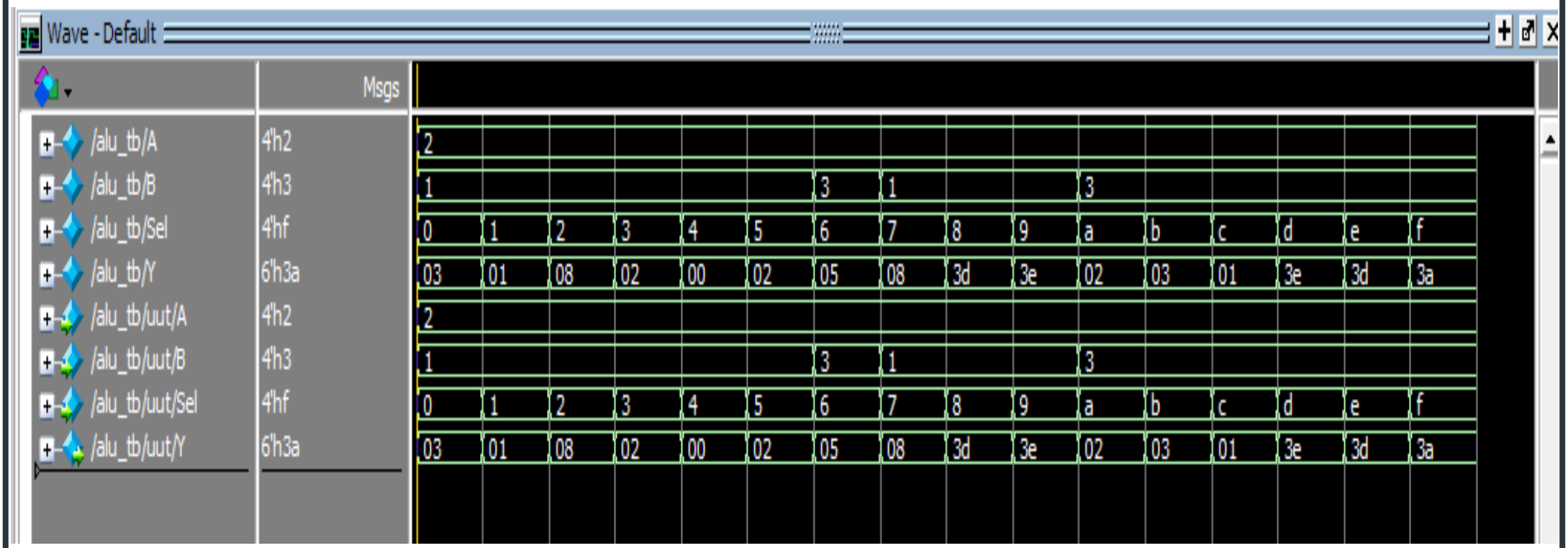
## Test Bench Code:

```
module alu_tb2;

// Testbench signals
reg [3:0] A;        // 4-bit test input operand A
reg [3:0] B;        // 4-bit test input operand B
reg [3:0] Sel;      // 4-bit test input selection signal
wire [5:0] Y;       // 6-bit test output result (wire)

// Instantiate the ALU module
alu uut (
   .A(A),
   .B(B),
   .Sel(Sel),
   .Y(Y)
);
integer i;
initial begin
   // Display header
   $display("Time\tA\tB\tSel\t\tY");

   // Initialize inputs
   A = 4'b0000;
   B = 4'b0000;
   Sel = 4'b0000;

   // Apply test cases
   for(i = 1; i <= 30; i = i + 1) begin
      $display("Test Case %d", i);
      A = $random % 16;  // Limit to 4-bit values (0-15)
      B = $random % 16;  // Limit to 4-bit values (0-15)
      Sel = $random % 16;  // Limit to 4-bit values (0-15)
      #10;
      $display("%0t\t%b\t%b\t%b\t", $time, A, B, Sel, Y);
   end

   // Stop simulation
   $stop;
end
endmodule
```

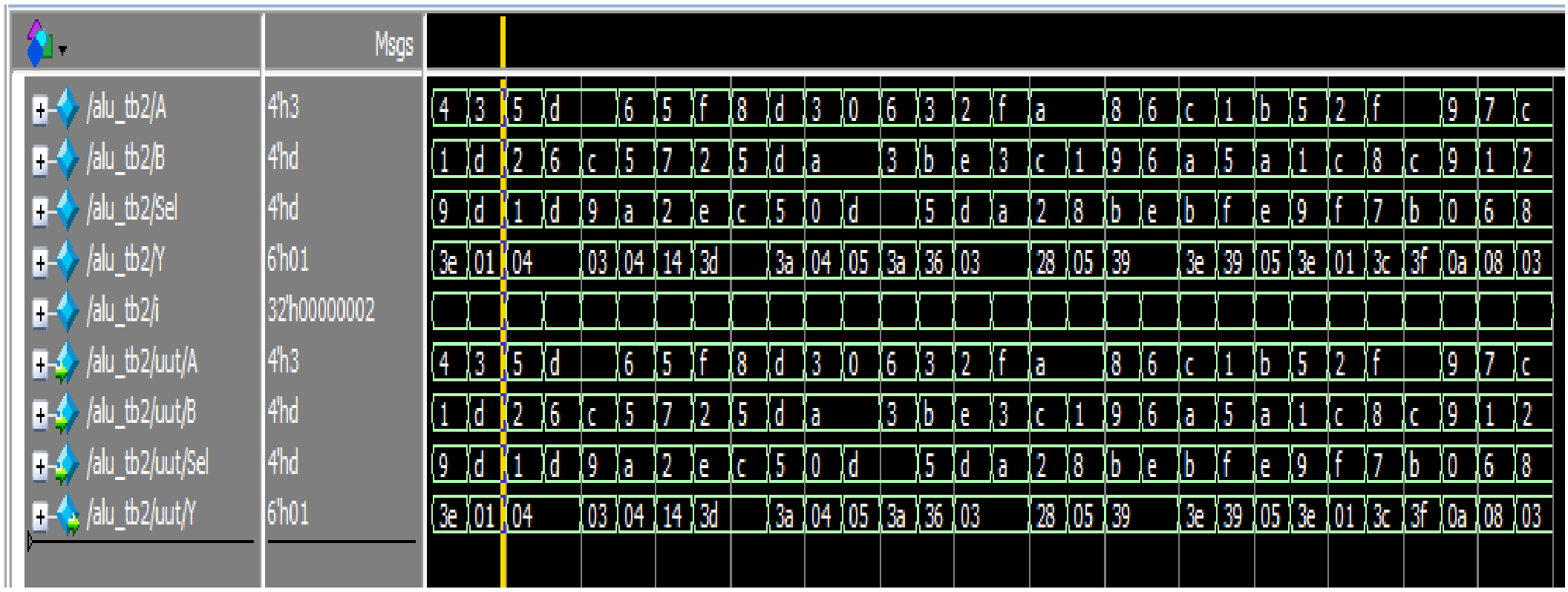
```
module alu_tb;
// Testbench signals
reg [3:0] A;         // 4-bit test input operand A     reg [3:0] B;         // 4-bit test input operand B
reg [3:0] Sel;       // 4-bit test input selection signal      wire [5:0] Y;        // 6-bit test output result (wire)
// Instantiate the ALU module
alu uut ( .A(A),.B(B),.Sel(Sel),.Y(Y)    );
initial begin
   // Display header
   $display("Time\tA\tB\tSel\t\tY (Actual)\tY (Expected)");

   // Apply test vectors for Arithmetic Operations (Sel[3] = 0)
   A = 4'b0010; B = 4'b0001; Sel = 4'b0000;
   #10;
   if(Y == 6'b000011)
      $display("Test Case 1 Passed");  else         $display("Test Case 1 Failed");
   $display("%0t\t%b\t%b\t%b\t\t%b\t\t%b", $time, A, B, Sel, Y, 6'b000011); // A + 1

   A = 4'b0010; B = 4'b0001; Sel = 4'b0001;
   #10;
   if(Y == 6'b000001)
      $display("Test Case 2 Passed");    else     $display("Test Case 2 Failed");
$display("%0t\t%b\t%b\t%b\t\t%b\t\t%b", $time, A, B, Sel, Y, 6'b000001); // A - 1

A = 4'b0010; B = 4'b0001; Sel = 4'b0010;
#10;
if(Y == 6'b001000)
   $display("Test Case 3 Passed");     else      $display("Test Case 3 Failed");
$display("%0t\t%b\t%b\t%b\t\t%b\t\t%b", $time, A, B, Sel, Y, 6'b001000); // A << 2

A = 4'b0010; B = 4'b0001; Sel = 4'b0011;  #10;
if(Y == 6'b000010)
   $display("Test Case 4 Passed");    else   $display("Test Case 4 Failed");
$display("%0t\t%b\t%b\t%b\t\t%b\t\t%b", $time, A, B, Sel, Y, 6'b000010); // B + 1

A = 4'b0010; B = 4'b0001; Sel = 4'b0100;#10;
if(Y == 6'b000000)
   $display("Test Case 5 Passed");   else    $display("Test Case 5 Failed");
$display("%0t\t%b\t%b\t%b\t\t%b\t\t%b", $time, A, B, Sel, Y, 6'b000000); // B - 1

A = 4'b0010; B = 4'b0001; Sel = 4'b0101;#10;
if(Y == 6'b000010)
   $display("Test Case 6 Passed");  else   $display("Test Case 6 Failed");
$display("%0t\t%b\t%b\t%b\t\t%b\t\t%b", $time, A, B, Sel, Y, 6'b000100); // B << 1

A = 4'b0010; B = 4'b0011; Sel = 4'b0110;  #10;
if(Y == 6'b000101)
   $display("Test Case 7 Passed");     else   $display("Test Case 7 Failed");
$display("%0t\t%b\t%b\t%b\t\t%b\t\t%b", $time, A, B, Sel, Y, 6'b000101); // A + B

A = 4'b0010; B = 4'b0001; Sel = 4'b0111;   #10;
if(Y == 6'b001000)
   $display("Test Case 8 Passed");  else  $display("Test Case 8 Failed");
$display("%0t\t%b\t%b\t%b\t\t%b\t\t%b", $time, A, B, Sel, Y, 6'b001000); // A << 4
```

```
// Apply test vectors for Logical Operations (Sel[3] = 1)
   A = 4'b0010; B = 4'b0001; Sel = 4'b1000;  #10;
   if(Y == 6'b111101)
      $display("Test Case 9 Passed");  else  $display("Test Case 9 Failed");
   $display("%0t\t%b\t%b\t%b\t\t%b\t\t%b", $time, A, B, Sel, Y, 6'b111101); // NOT A

   A = 4'b0010; B = 4'b0001; Sel = 4'b1001; #10;
   if(Y == 6'b111110)
      $display("Test Case 10 Passed");  else   $display("Test Case 10 Failed");
   $display("%0t\t%b\t%b\t%b\t\t%b\t\t%b", $time, A, B, Sel, Y, 6'b111110); // NOT B

   A = 4'b0010; B = 4'b0011; Sel = 4'b1010;    #10;
   if(Y == 6'b000010)
      $display("Test Case 11 Passed");    else $display("Test Case 11 Failed");
   $display("%0t\t%b\t%b\t%b\t\t%b\t\t%b", $time, A, B, Sel, Y, 6'b000010); // A AND B

   A = 4'b0010; B = 4'b0011; Sel = 4'b1011;
   #10;
   if(Y == 6'b000011)
      $display("Test Case 12 Passed");
   else
      $display("Test Case 12 Failed");
   $display("%0t\t%b\t%b\t%b\t\t%b\t\t%b", $time, A, B, Sel, Y, 6'b000011); // A OR B

   A = 4'b0010; B = 4'b0011; Sel = 4'b1100;
   #10;
   if(Y == 6'b000001)
      $display("Test Case 13 Passed");
   else
      $display("Test Case 13 Failed");
   $display("%0t\t%b\t%b\t%b\t\t%b\t\t%b", $time, A, B, Sel, Y, 6'b000001); // A XOR B

   A = 4'b0010; B = 4'b0011; Sel = 4'b1101;
   #10;
   if(Y == 6'b111110)
      $display("Test Case 14 Passed");
   else
      $display("Test Case 14 Failed");
   $display("%0t\t%b\t%b\t%b\t\t%b\t\t%b", $time, A, B, Sel, Y, 6'b111110); // A XNOR B

   A = 4'b0010; B = 4'b0011; Sel = 4'b1110;
   #10;
   if(Y == 6'b111101)
      $display("Test Case 15 Passed");
   else
      $display("Test Case 15 Failed");
   $display("%0t\t%b\t%b\t%b\t\t%b\t\t%b", $time, A, B, Sel, Y, 6'b111101); // NAND A and B

   A = 4'b0010; B = 4'b0011; Sel = 4'b1111;
   #10;
   if(Y == 6'b111010)
      $display("Test Case 16 Passed");
   else
      $display("Test Case 16 Failed");
   $display("%0t\t%b\t%b\t%b\t\t%b\t\t%b", $time, A, B, Sel, Y, 6'b111010); // Nor A and B
$stop;
end
endmodule
```

# 2- different test cases for different inputs:

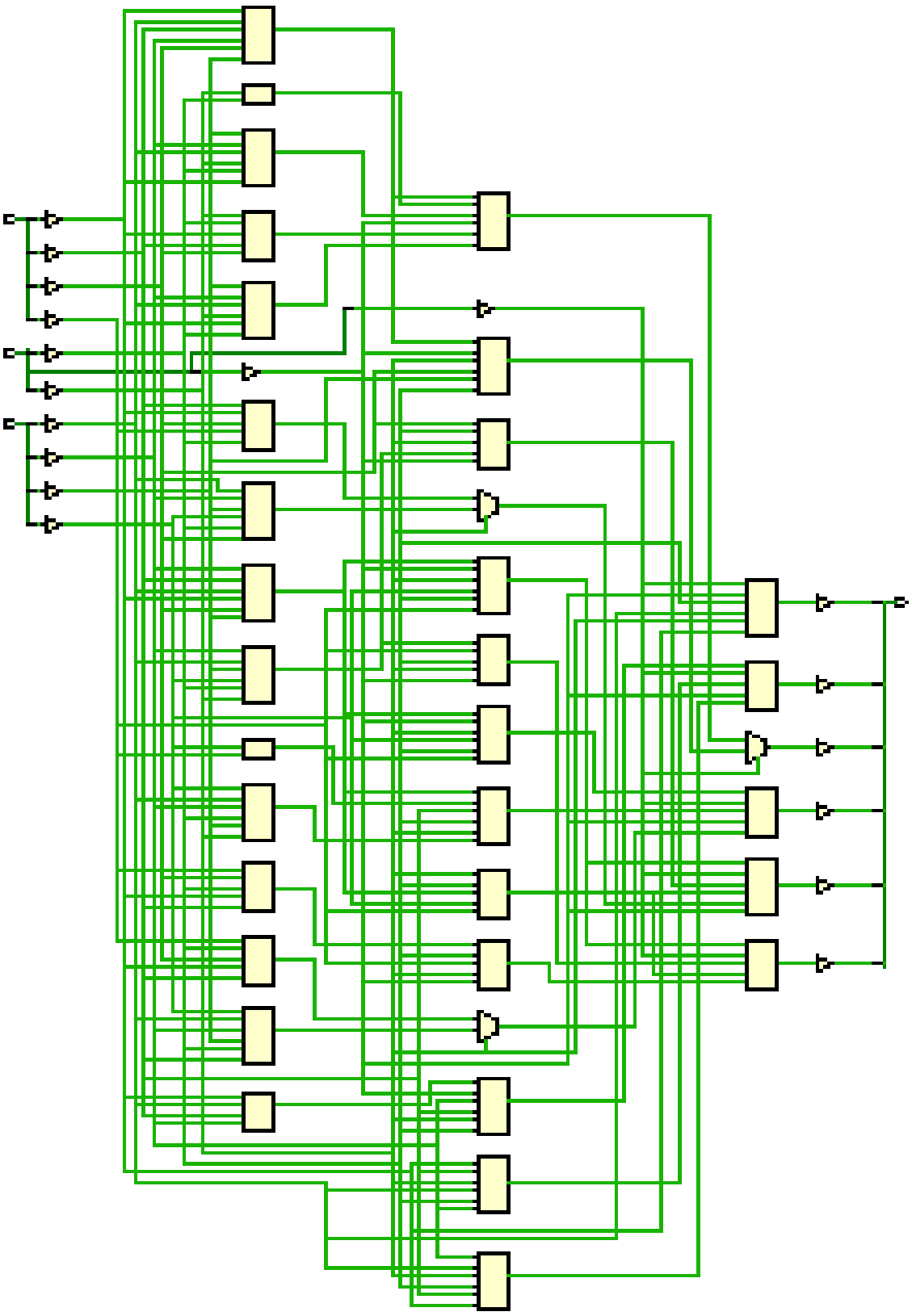


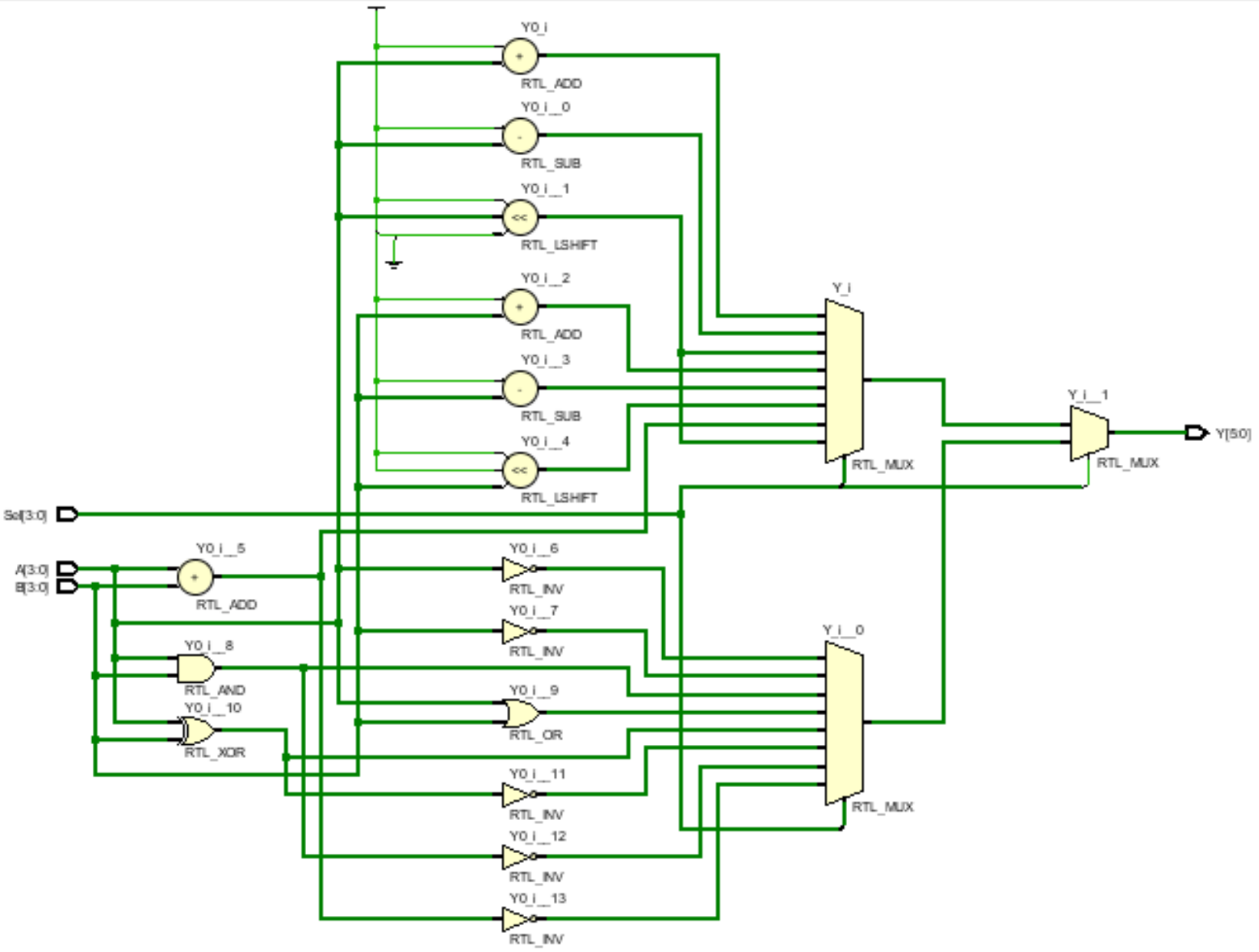

```
# Time  A       B       Sel             Y (Actual) Y (Expected)
# Test Case 1 Passed
# 10    0010    0001    0000            000011          000011
# Test Case 2 Passed
# 20    0010    0001    0001            000001          000001
# Test Case 3 Passed
# 30    0010    0001    0010            001000          001000
# Test Case 4 Passed
# 40    0010    0001    0011            000010          000010
# Test Case 5 Passed
# 50    0010    0001    0100            000000          000000
# Test Case 6 Passed
# 60    0010    0001    0101            000010          000100
# Test Case 7 Passed
# 70    0010    0011    0110            000101          000101
# Test Case 8 Passed
# 80    0010    0001    0111            001000          001000
# Test Case 9 Passed
# 90    0010    0001    1000            111101          111101
# Test Case 10 Passed
# 100   0010    0001    1001            111110          111110
# Test Case 11 Passed
# 110   0010    0011    1010            000010          000010
# Test Case 12 Passed
# 120   0010    0011    1011            000011          000011
# Test Case 13 Passed
# 130   0010    0011    1100            000001          000001
# Test Case 14 Passed
# 140   0010    0011    1101            111110          111110
# Test Case 15 Passed
# 150   0010    0011    1110            111101          111101
# Test Case 16 Passed
# 160   0010    0011    1111            111010          111010
```




```
# Time  A       B       Sel             Y
# Test Case          1
# 10    0100    0001    1001    62
# Test Case          2
# 20    0011    1101    1101     1
# Test Case          3
# 30    0101    0010    0001     4
# Test Case          4
# 40    1101    0110    1101     4
# Test Case          5
# 50    1101    1100    1001     3
# Test Case          6
# 60    0110    0101    1010     4
# Test Case          7
# 70    0101    0111    0010    20
# Test Case          8
# 80    1111    0010    1110    61
# Test Case          9
# 90    1000    0101    1100    61
# Test Case         10
# 100   1101    1101    0101    58
# Test Case         11
# 110   0011    1010    0000     4
# Test Case         12
# 120   0000    1010    1101     5
# Test Case         13
# 130   0110    0011    1101    58
# Test Case         14
# 140   0011    1011    0101    54
# Test Case         15
# 150   0010    1110    1101     3
# Test Case         16
# 160   1111    0011    1010     3
# Test Case         17
# 170   1010    1100    0010    40
# Test Case         18
# 180   1010    0001    1000     5
# Test Case         19
# 190   1000    1001    1011    57
```

```
# Test Case         19
# 190   1000    1001    1011    57
# Test Case         20
# 200   0110    0110    1110    57
# Test Case         21
# 210   1100    1010    1011    62
# Test Case         22
# 220   0001    0101    1111    57
# Test Case         23
# 230   1011    1010    1110     5
# Test Case         24
# 240   0101    0001    1001    62
# Test Case         25
# 250   0010    1100    1111     1
# Test Case         26
# 260   1111    1000    0111    60
# Test Case         27
# 270   1111    1100    1011    63
# Test Case         28
# 280   1001    1001    0000    10
# Test Case         29
# 290   0111    0001    0110     8
# Test Case         30
# 300   1100    0010    1000     3
```



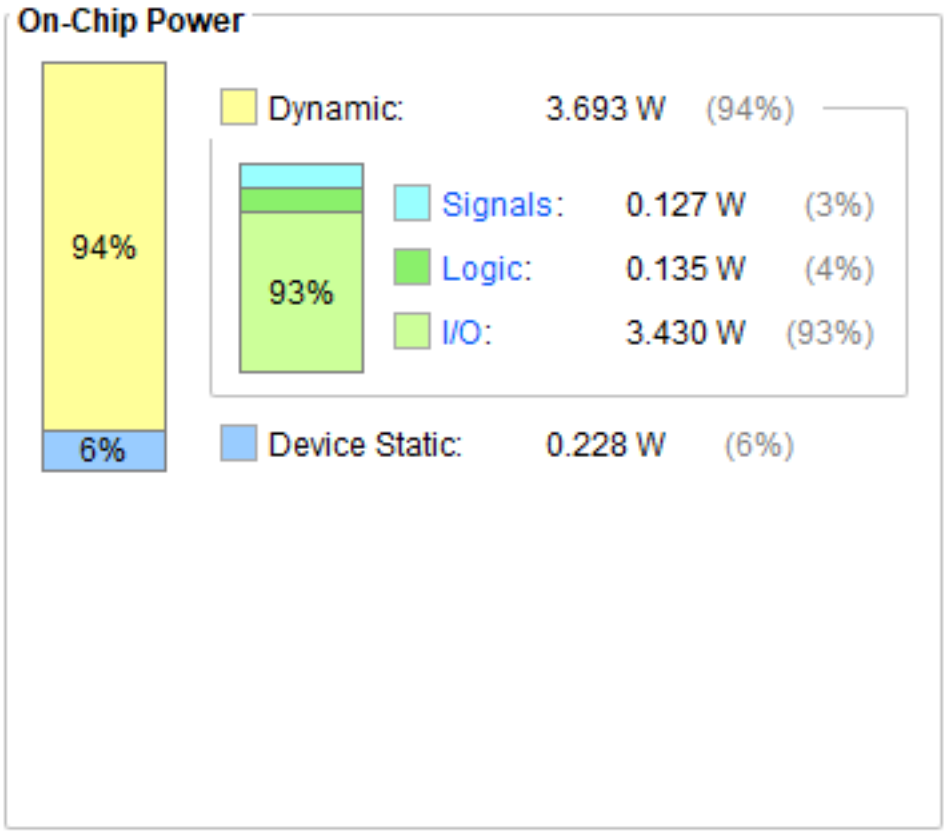

Y0_i
RTL_ADD
Y0_i__0
RTL_SUB
Y0_i__1
RTL_LSHIFT
Y0_i__2
RTL_ADD
Y0_i__3
RTL_SUB
Y0_i__4
RTL_LSHIFT
Y_i
RTL_MUX
Y_i__1
RTL_MUX
Y[5:0]
Sel[3:0]
A[3:0]
B[3:0]
Y0_i__5
RTL_ADD
Y0_i__6
RTL_INV
Y0_i__7
RTL_INV
Y_i__0
RTL_MUX
Y0_i__8
RTL_AND
Y0_i__9
RTL_OR
Y0_i__10
RTL_XOR
Y0_i__11
RTL_INV
Y0_i__12
RTL_INV
Y0_i__13
RTL_INV

## Summary

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

| | |
|---|---|
| **Total On-Chip Power:** | **3.921 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **70.2°C** |
| Thermal Margin: | 14.8°C (1.2 W) |
| Effective ϑJA: | 11.5°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

Launch Power Constraint Advisor to find and fix invalid switching activity




Hierarchy

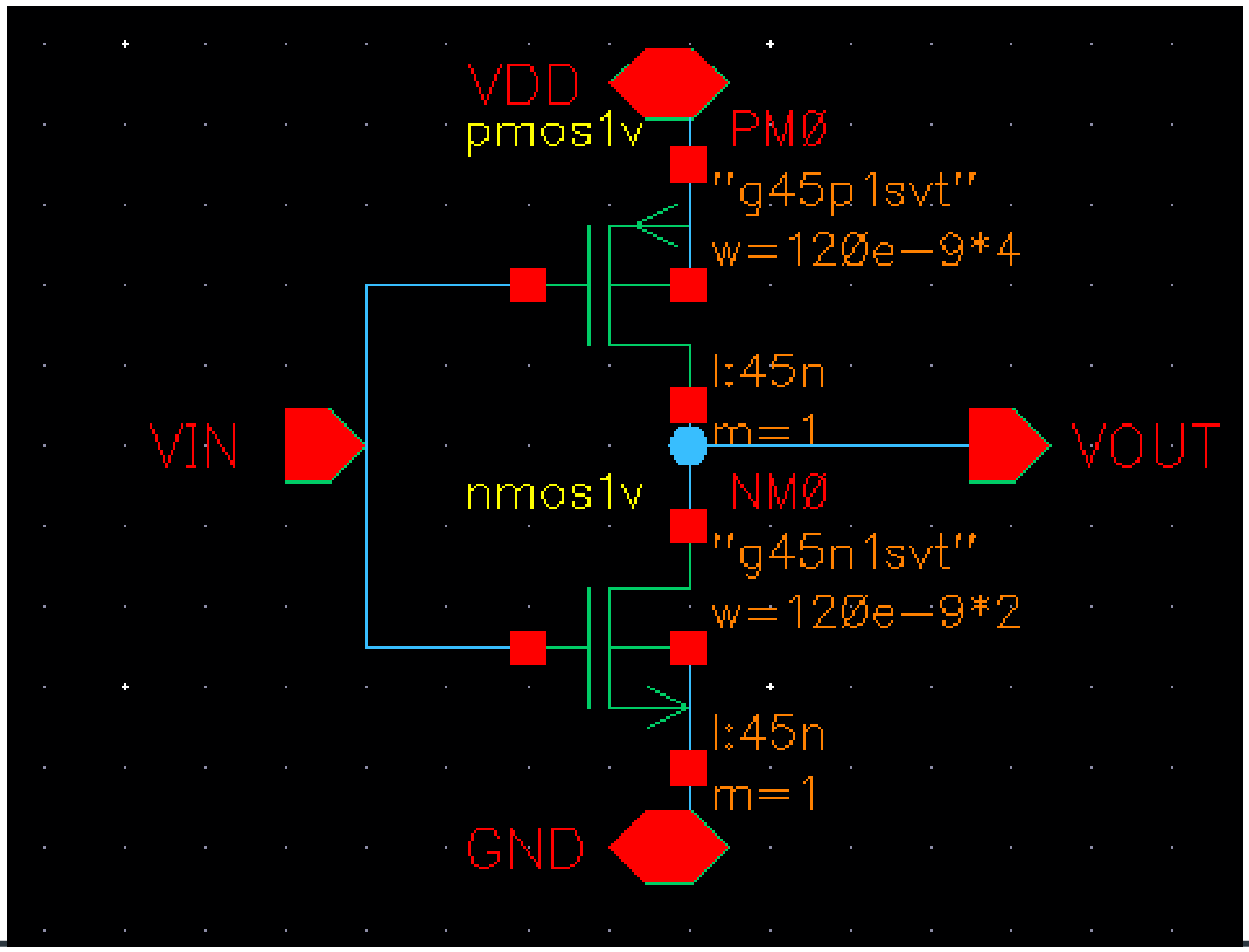
| Name 1 | Slice LUTs (53200) | F7 Muxes (26600) | Bonded IOB (200) |
|---|---|---|---|
| N alu | 30 | 3 | 18 |

# 4. Cadence schematics

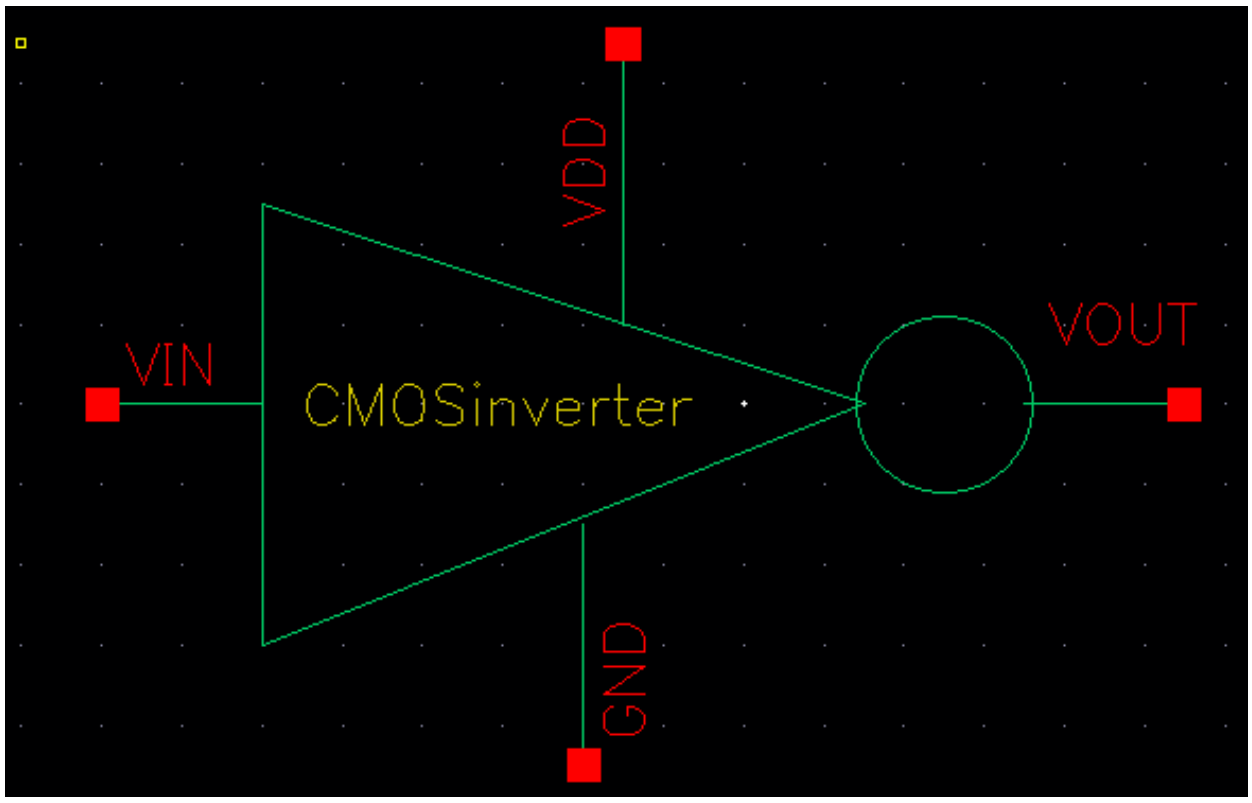
- In this section, we undertake the individual design of the ALU's schematics for each component and strive to optimize them.
- Our initial step involves designing the essential gates, serving as the primary level of abstraction.
- We will provide a scheme and a symbol for each component used in the design

## 4.1 Inverter Schematic

Defining the Wp and Wn values is a crucial aspect of the reference inverter, and it is advantageous to assume Wp : Wn = (2:1). According to Cadence guidelines, the minimum width available in this technology corresponds to a width of 120nm, considering the minimum length. Therefore, we will assign Wn as 120nm and set Wp as 2 times Wn. Defining FP (number of fingers for pmos), FN (number of fingers for nmos), to switch the best value of them to choose the minimum propagation delay.



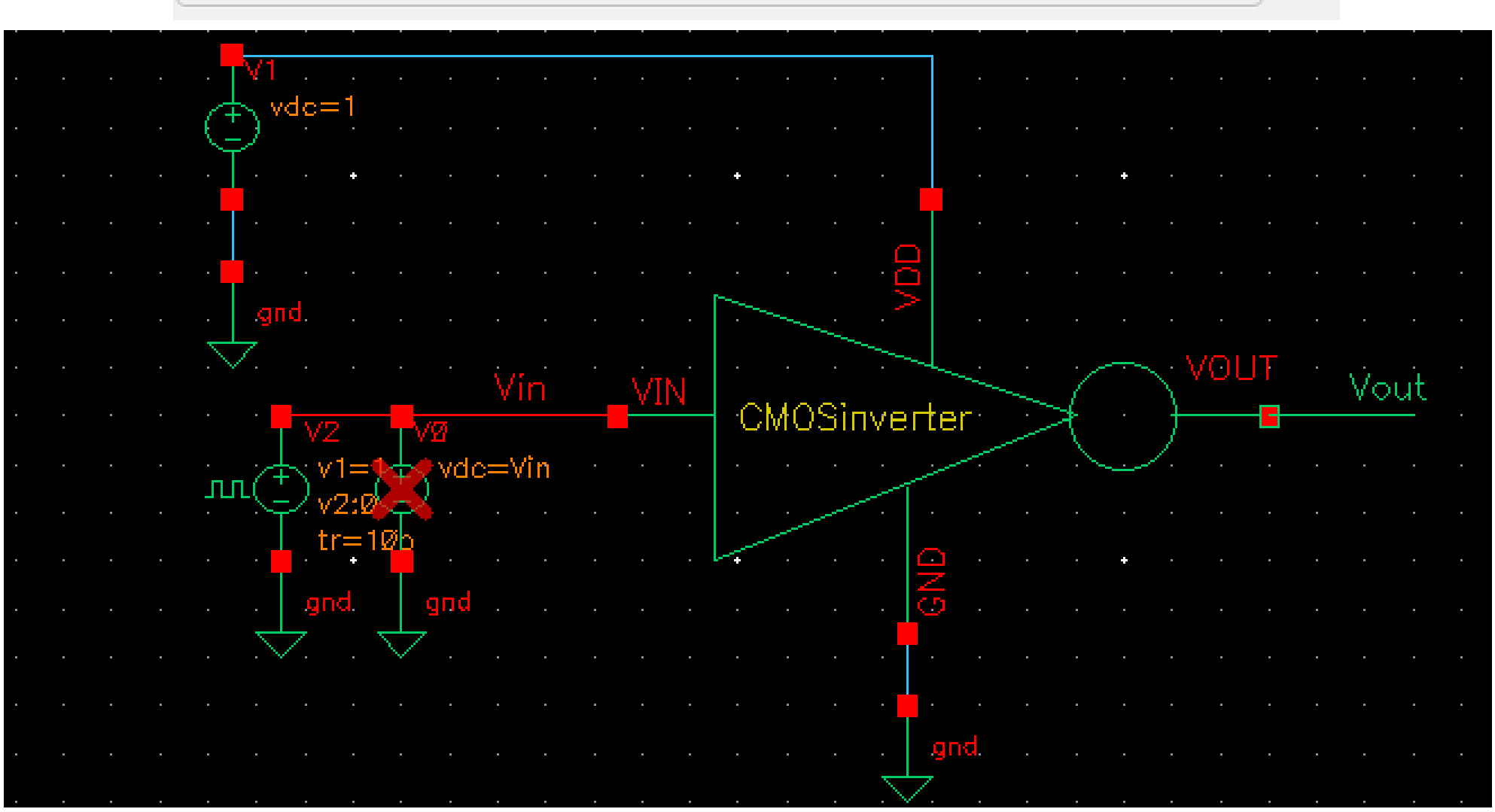
# 4.1 Inverter Symbol




# 4.1 Inverter Testbench

Initially defining the Fp and Fn values:

| CDF Parameter | Value | Display |
|---|---|---|
| FN | 1 | off |
| FP | 2 | off |
| H | 1 | off |



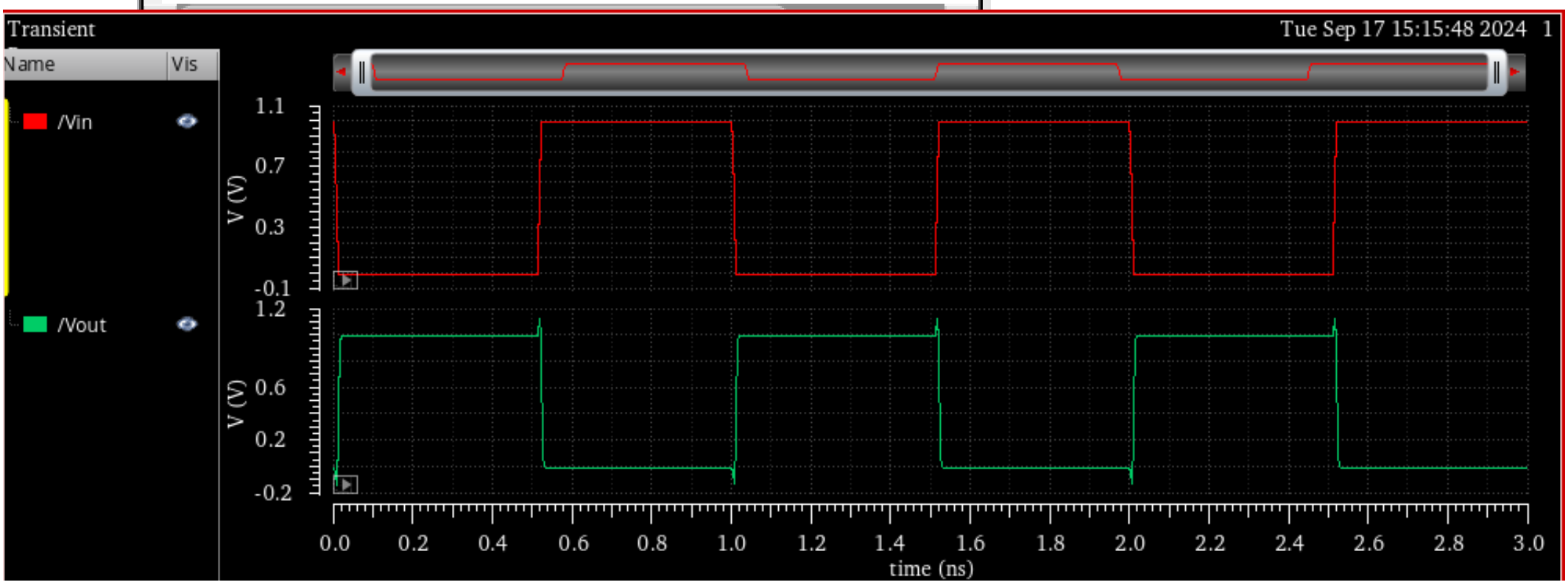
$$Tpd = 5.80\,ps$$

**Analyses**

| | Type | Enable | Arguments |
|---|---|---|---|
| 1 | tran | ☑ | 0 3n moderate |

**Outputs**

| | Name/Signal/Expr | Value | Plot | Save | Save Optio |
|---|---|---|---|---|---|
| 1 | Vin | | ☑ | ☐ | allv |
| 2 | Vout | | ☑ | ☐ | allv |
| 3 | tpdf | 6.365p | ☑ | ☐ | |
| 4 | tpdr | 5.24293p | ☑ | ☐ | |
| 5 | tpd | 5.80396p | ☑ | ☐ | |




**Try to increase the widths by increase the number of fingers of each:**

| CDF Parameter | Value | Display |
|---|---|---|
| FN | 2 | off |
| FP | 4 | off |
| H | 1 | off |

**So:**

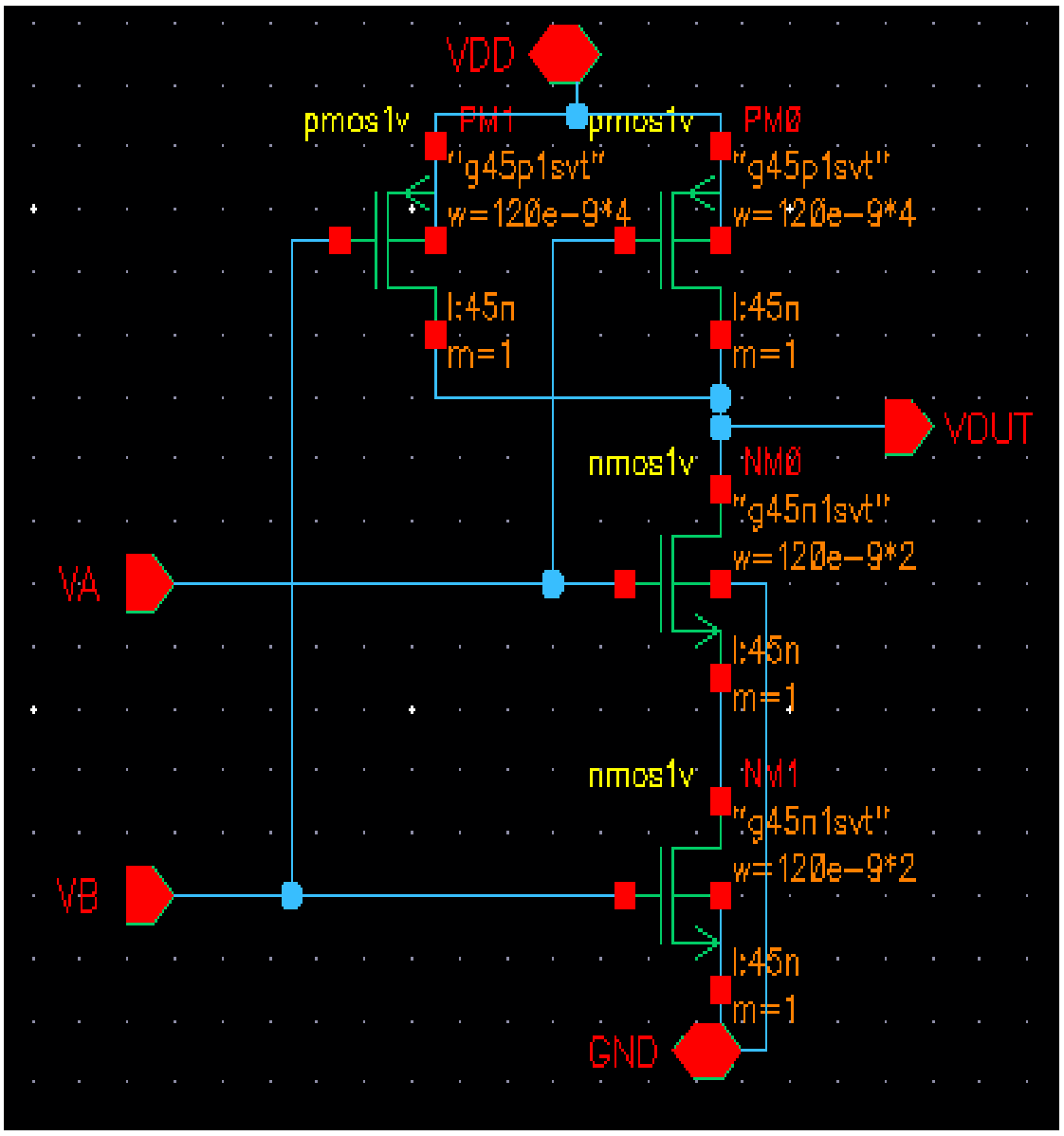
$$Tpd = 5.53\,ps$$

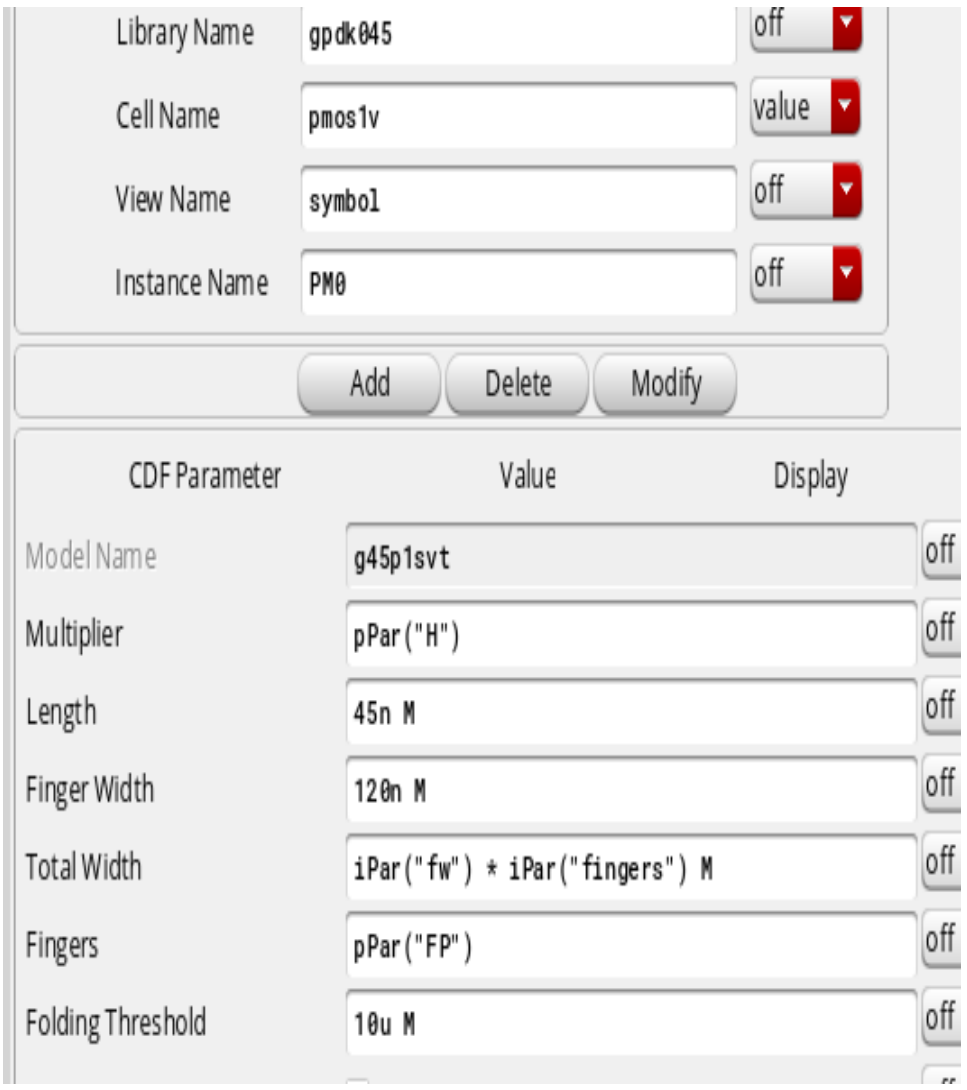
**Outputs**

| | Name/Signal/Expr | Value | Plot | Save | Save Optio |
|---|---|---|---|---|---|
| 1 | Vin | | ☑ | ☐ | allv |
| 2 | Vout | | ☑ | ☐ | allv |
| 3 | tpdf | 5.96527p | ☑ | ☐ | |
| 4 | tpdr | 5.10663p | ☑ | ☐ | |
| 5 | tpd | 5.53595p | ☑ | ☐ | |

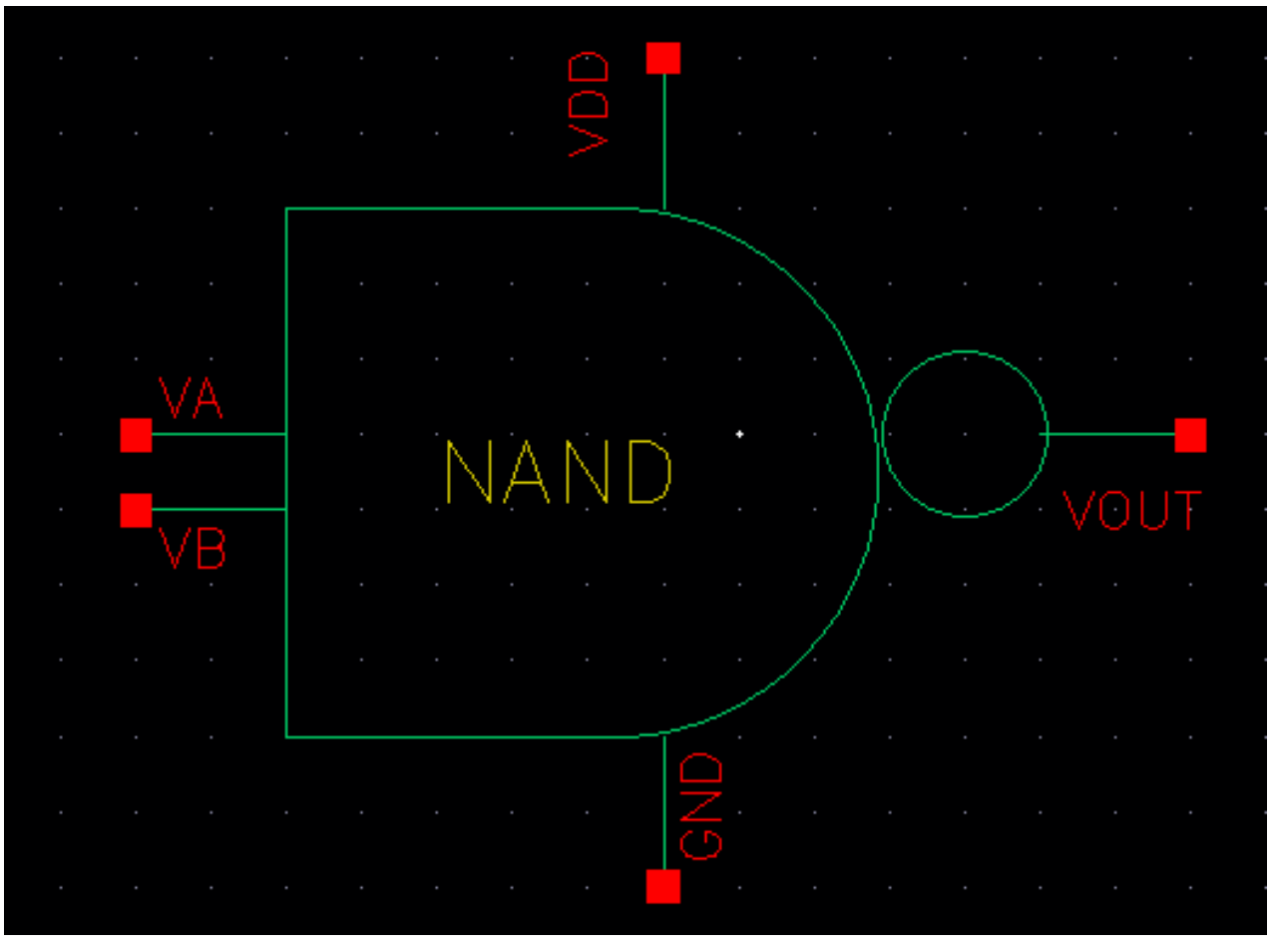
# 4.2 NAND Schematic

- Initially, we determine the gate size based on the reference inverter, considering the worst-case scenario. $Wn = 2Wnref =$ 320$n$m $Wp = Wpref =$ 320$n$m

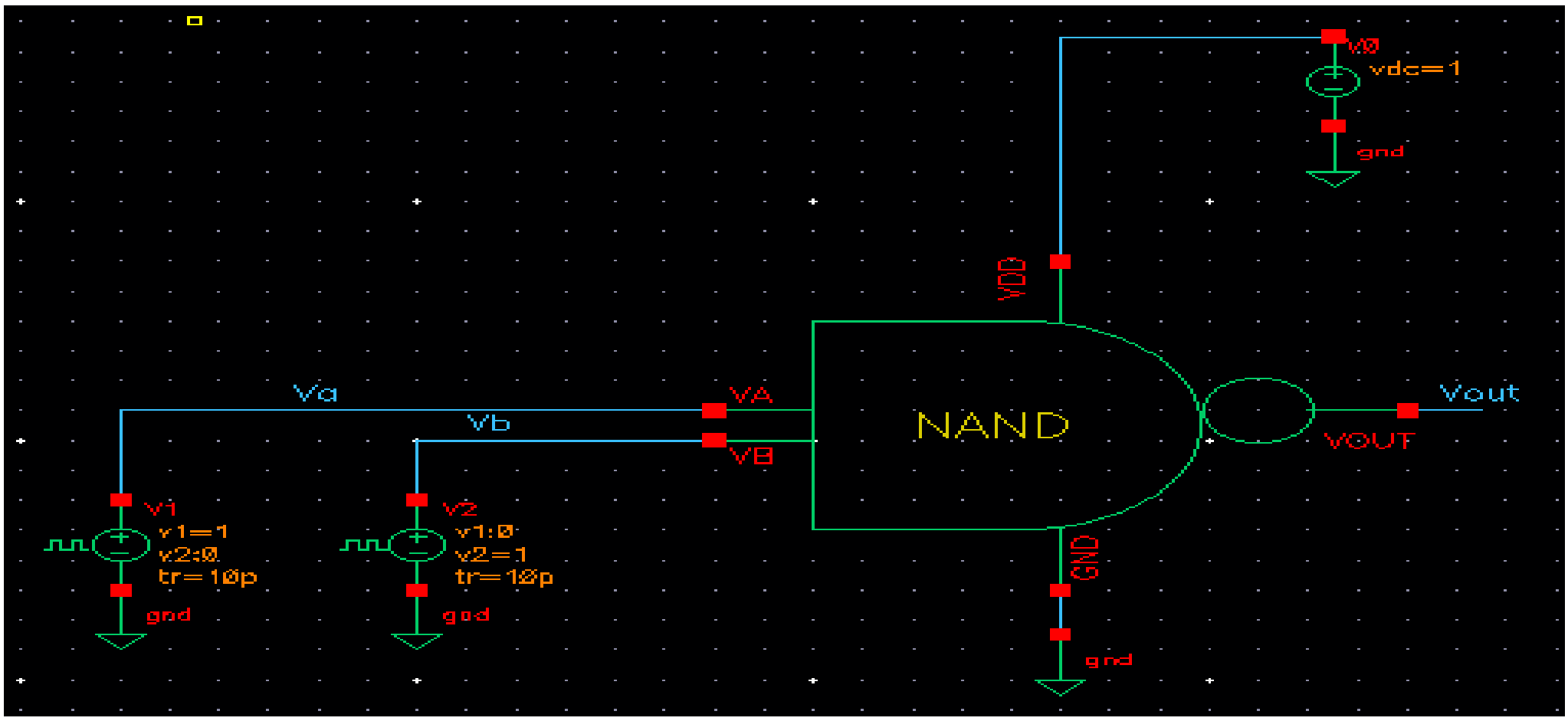



# 4.2 NAND Symbol

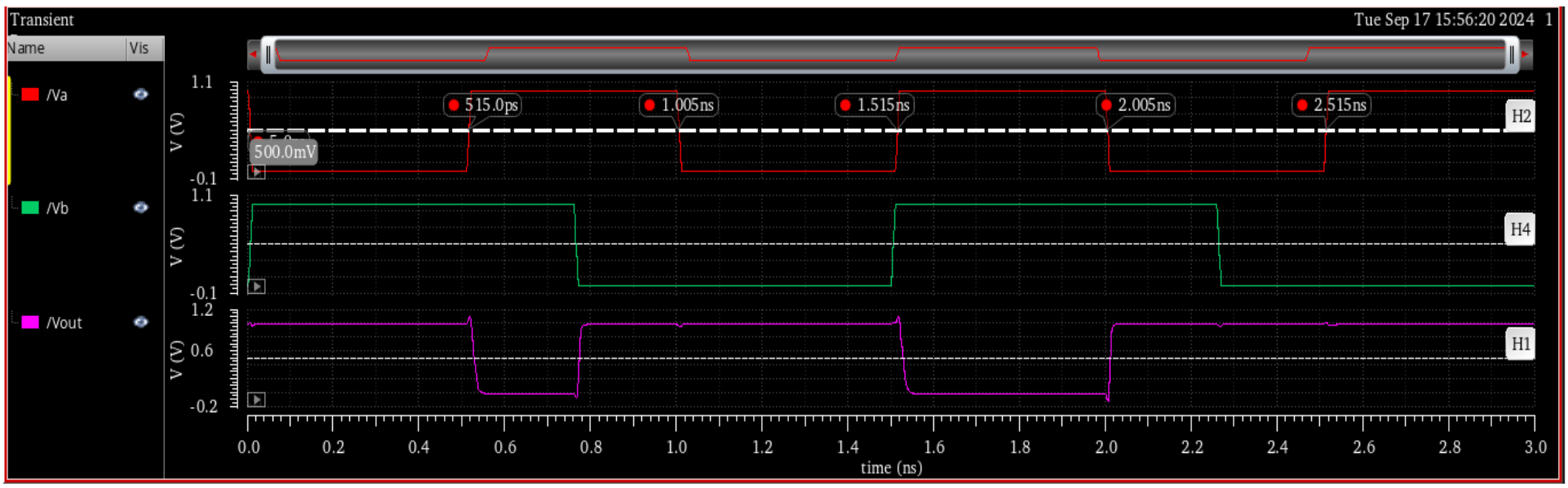


# NAND Testbench




Initially

| CDF Parameter | Value |
|---|---|
| FP | 2 |
| FN | 1 |
| H | 1 |



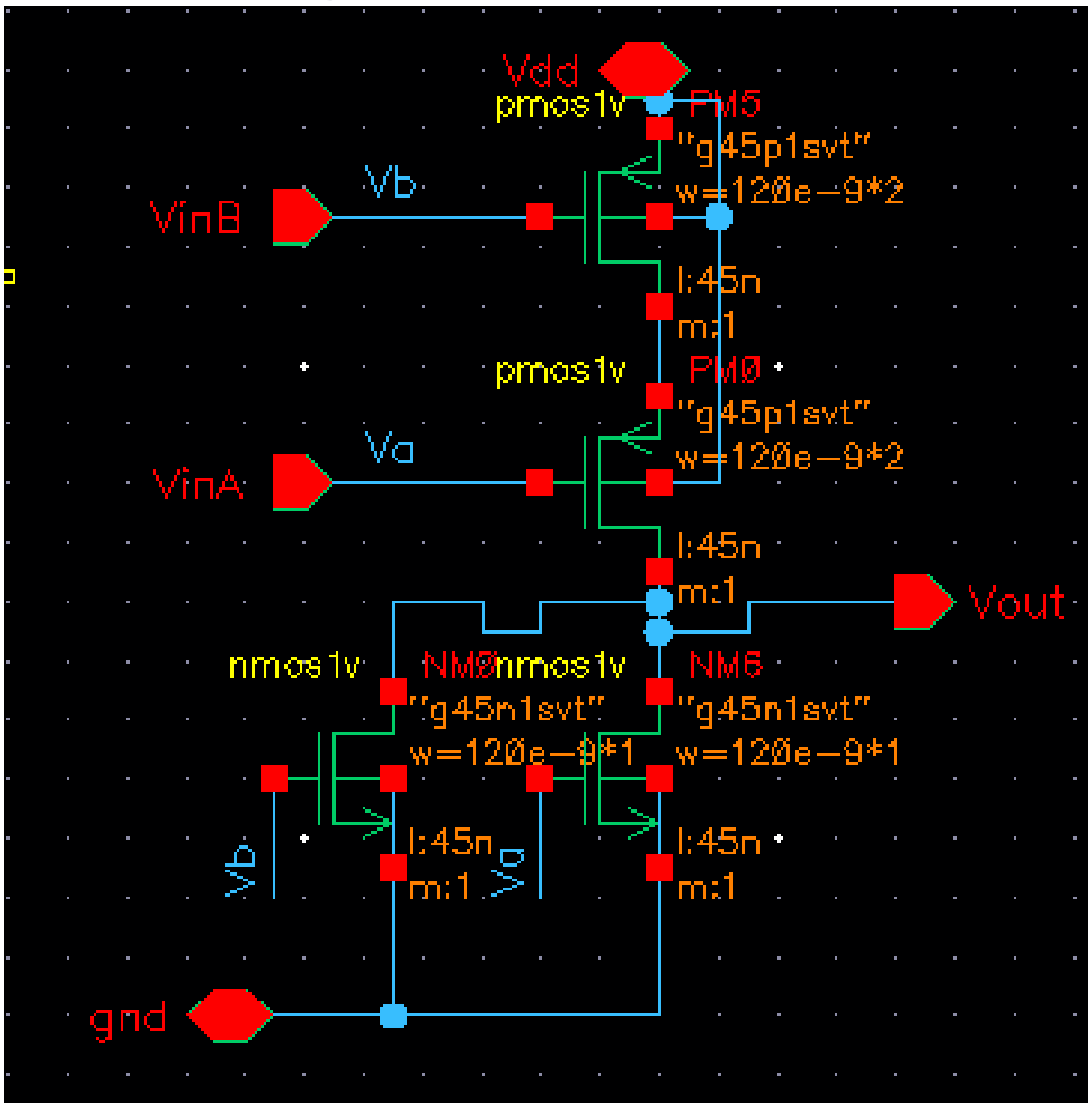

**So:** $Tpd = 9.683\,ps$

| CDF Parameter | Value |
|---|---|
| FP | 4 |
| FN | 2 |

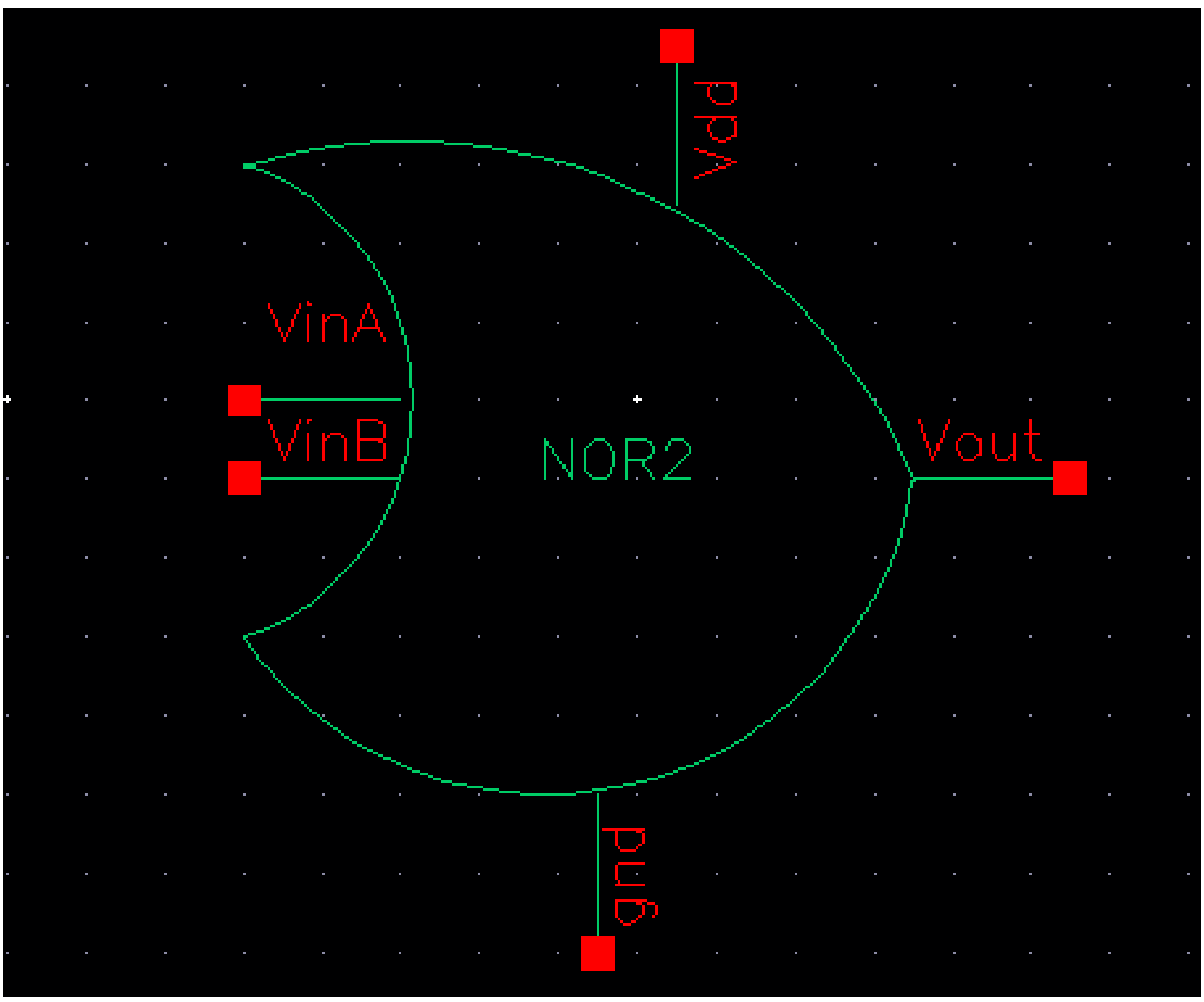
**So:** $Tpd = 9.17\,ps$

# 4.3 NOR Schematic

- Initially, we determine the gate size based on the reference inverter while taking the worst-case scenario into account.




# 4.3 NOR Symbol



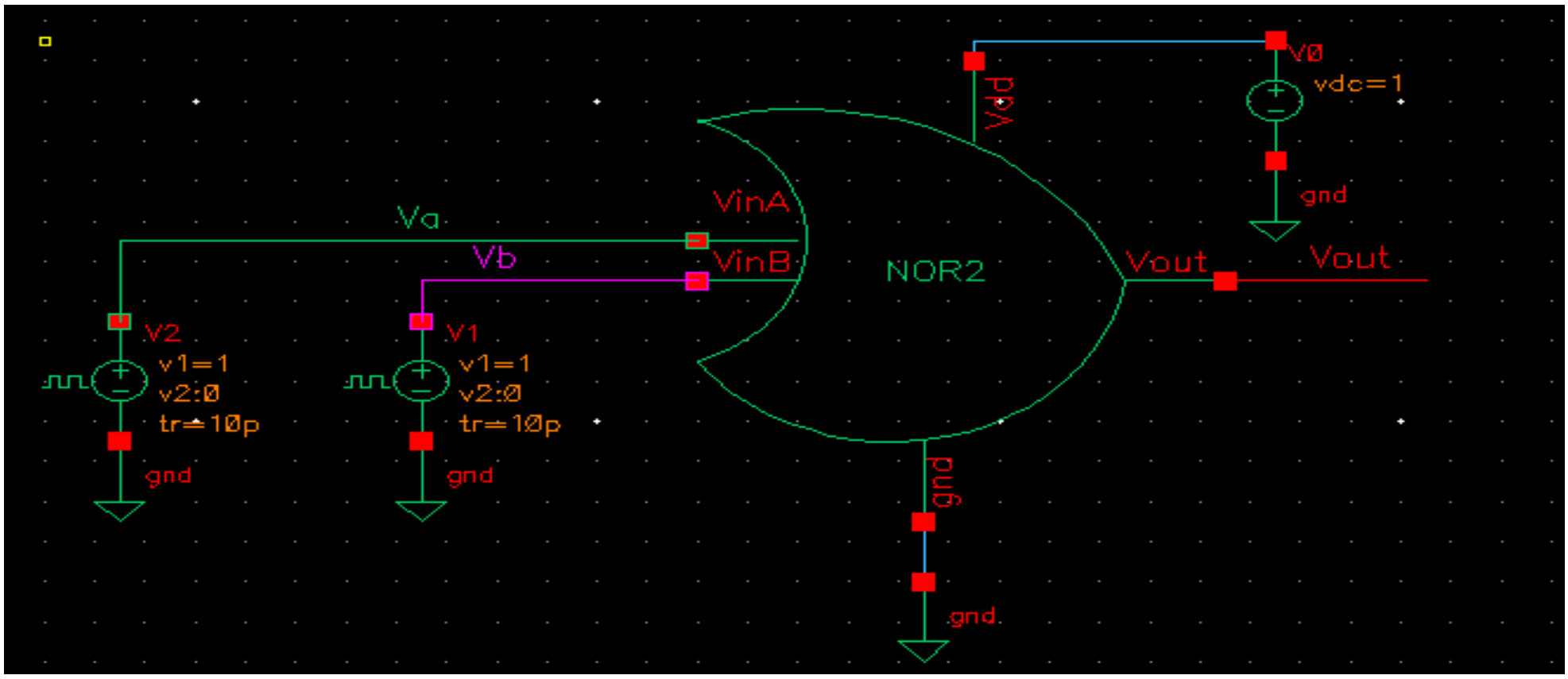
# 4.1 NOR Testbench

| CDF Parameter | Value |
|---|---|
| FN | 1 |
| FP | 2 |


**So:** $Tpd = 12\,ps$

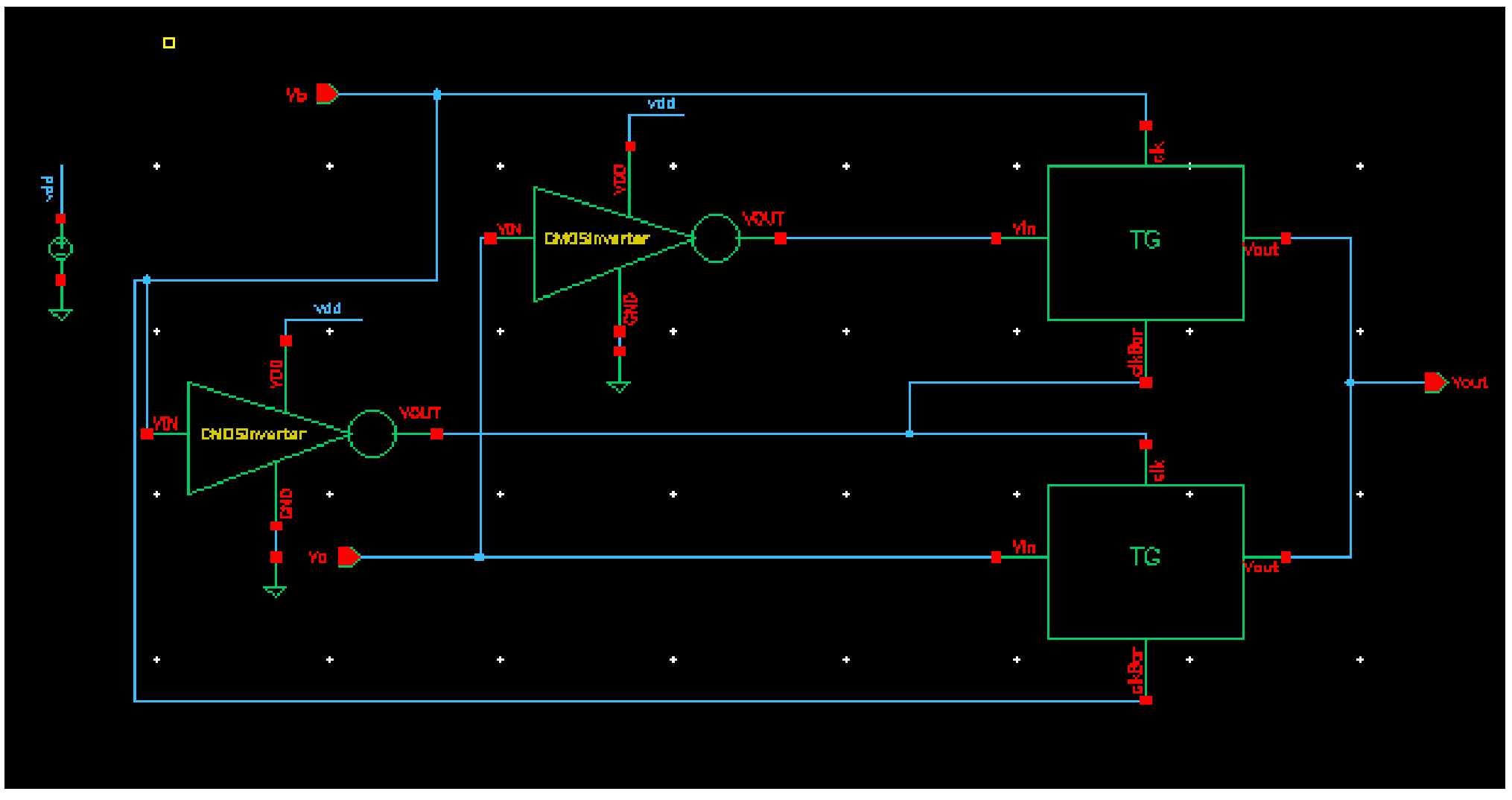
| CDF Parameter | Value | Display |
|---|---|---|
| FN | 2 | off |
| FP | 4 | off |


**So:** $Tpd = 9.91\,ps$

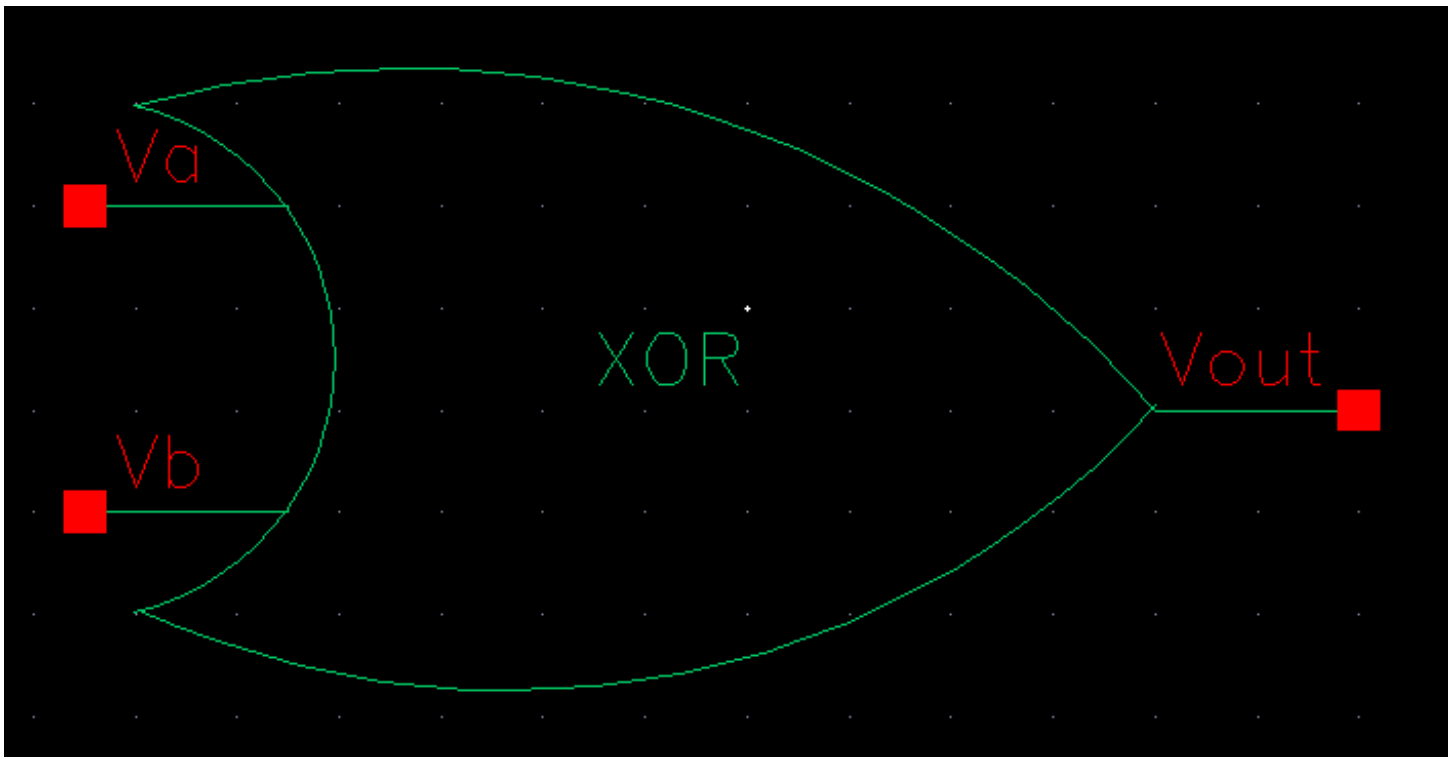



# XOR Schematic

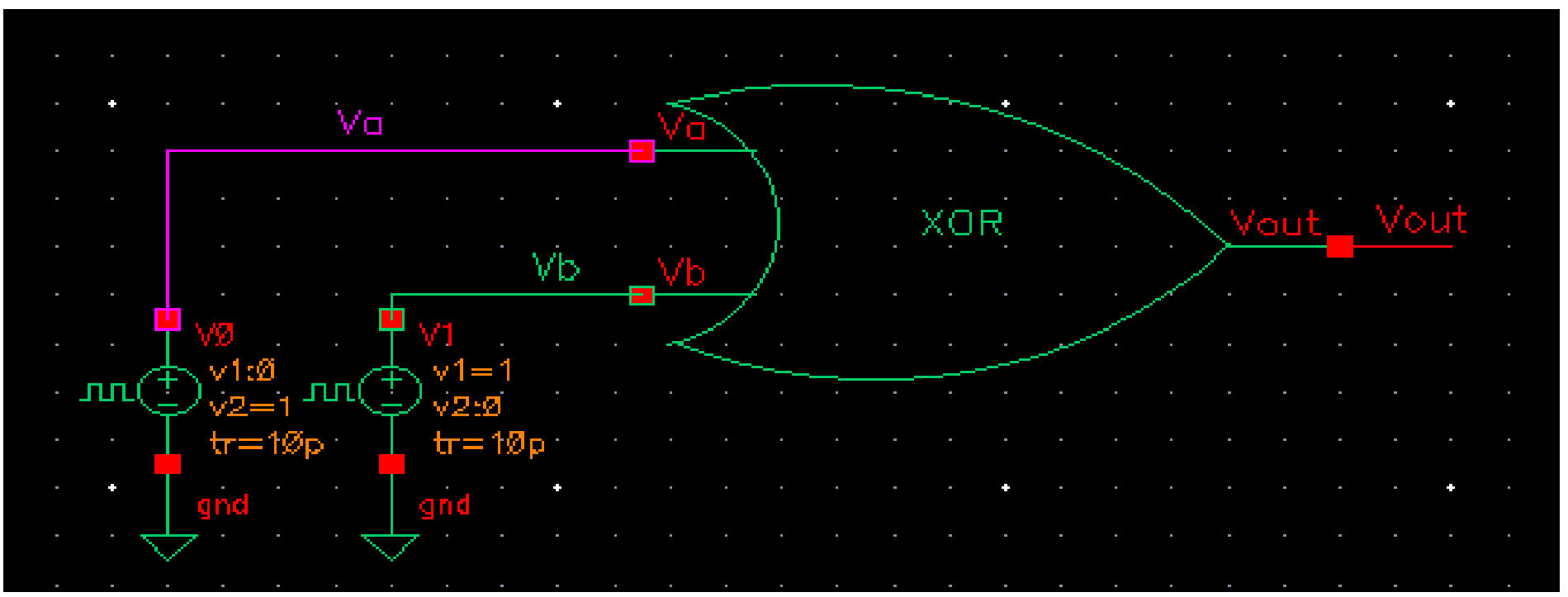
- To achieve the optimal implementation of the XOR functionality in terms of delay time, we utilized Transmission Gate Logic (TGL) for the XOR gate.

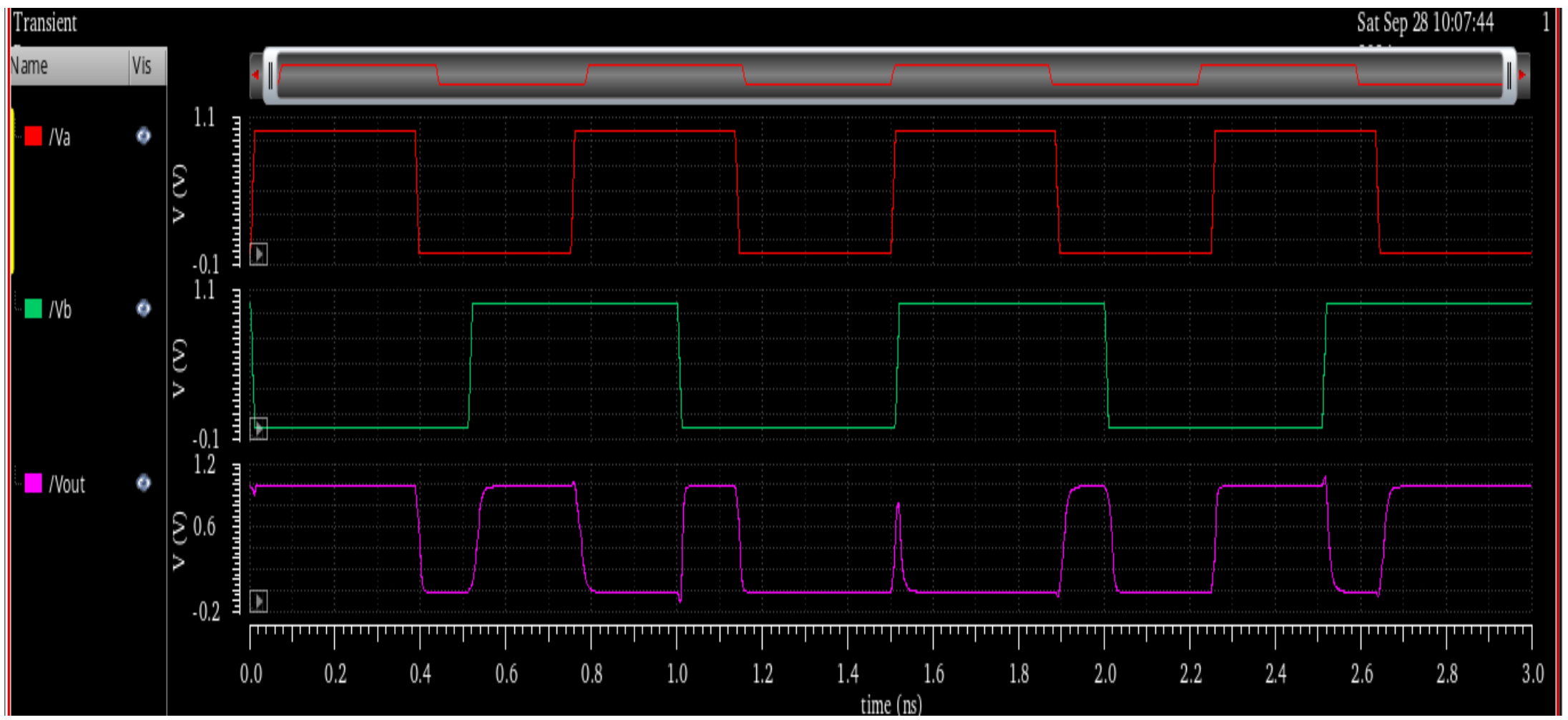


# 4.6 XOR Symbol



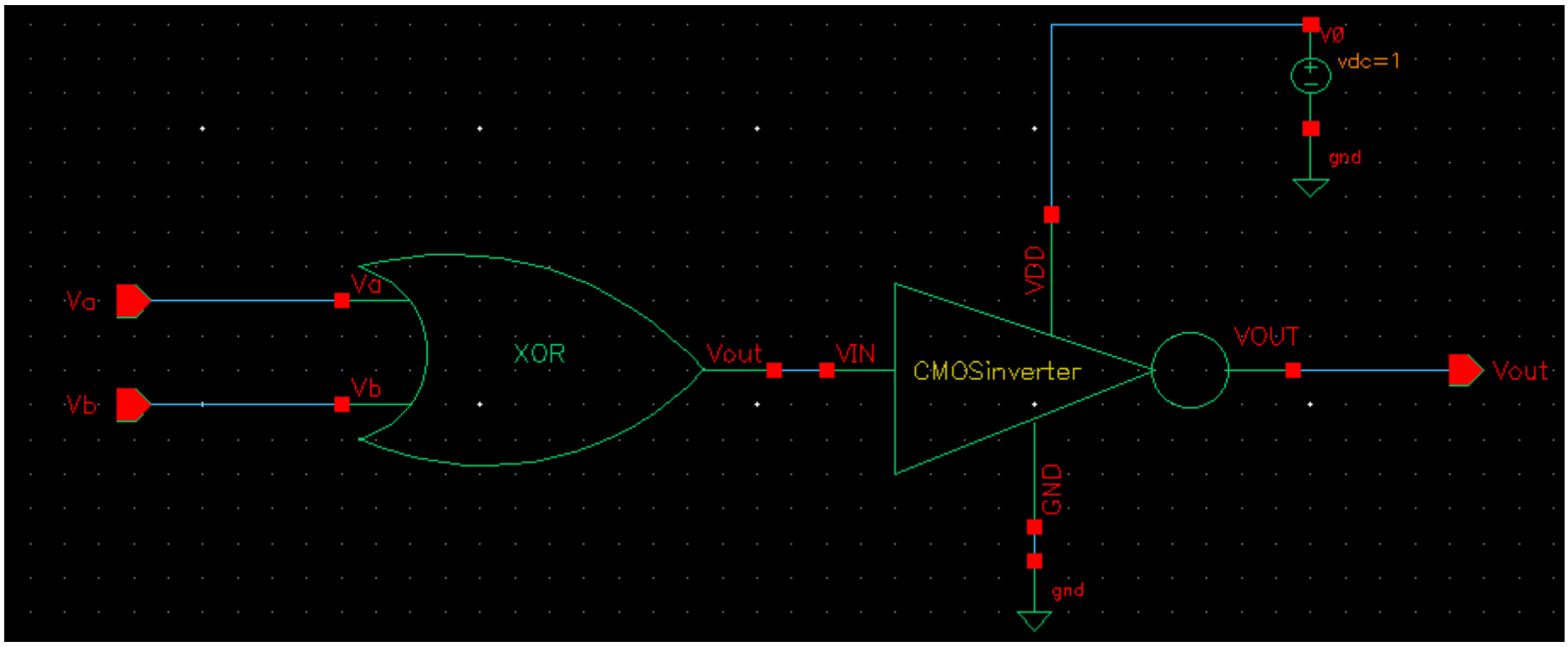

# XOR Testbench



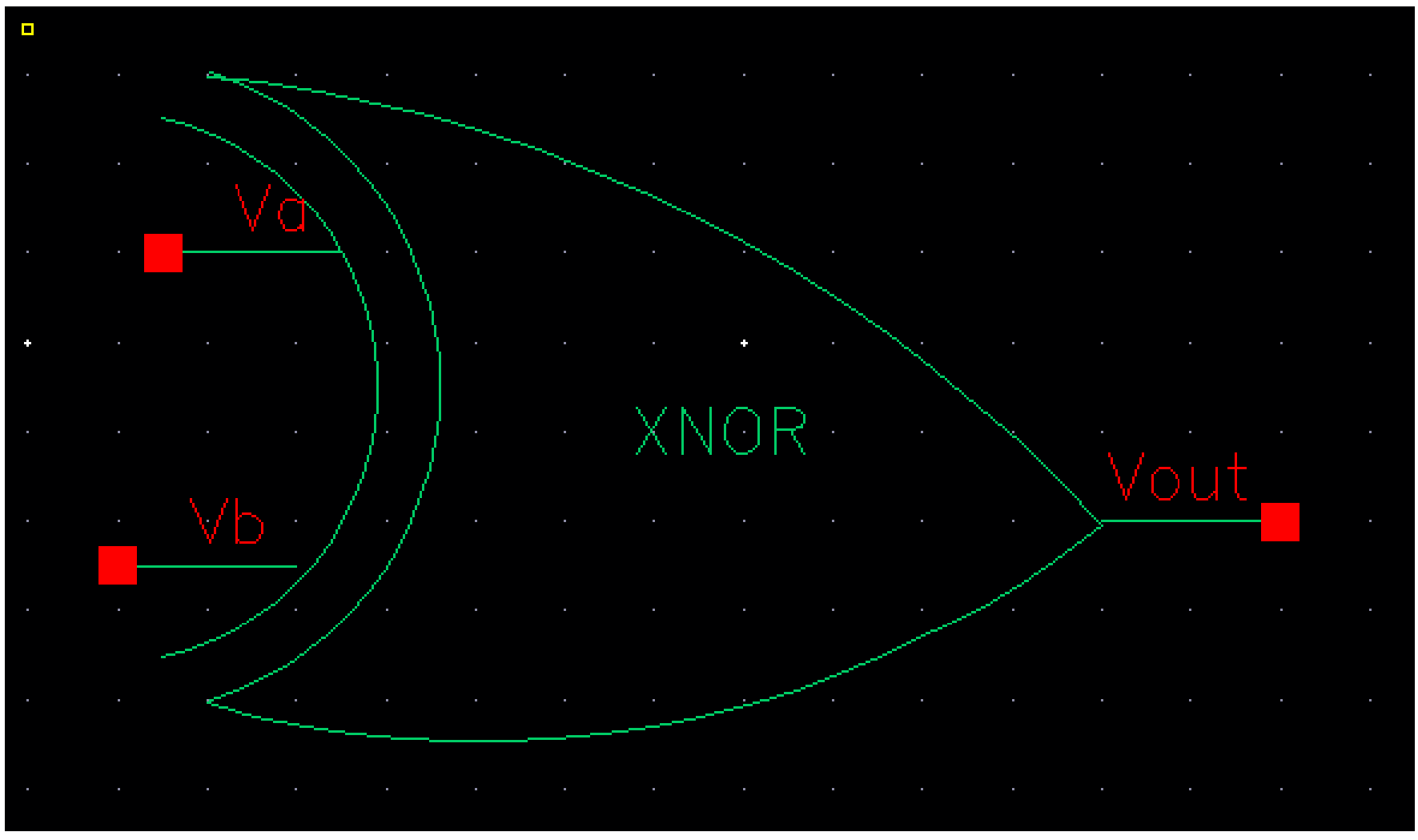



## 4.8 XNOR Schematic

- To ensure the most optimal implementation of the XNOR functionality in terms of delay time, we employed Transmission Gate Logic (TGL) for the XNOR gate.

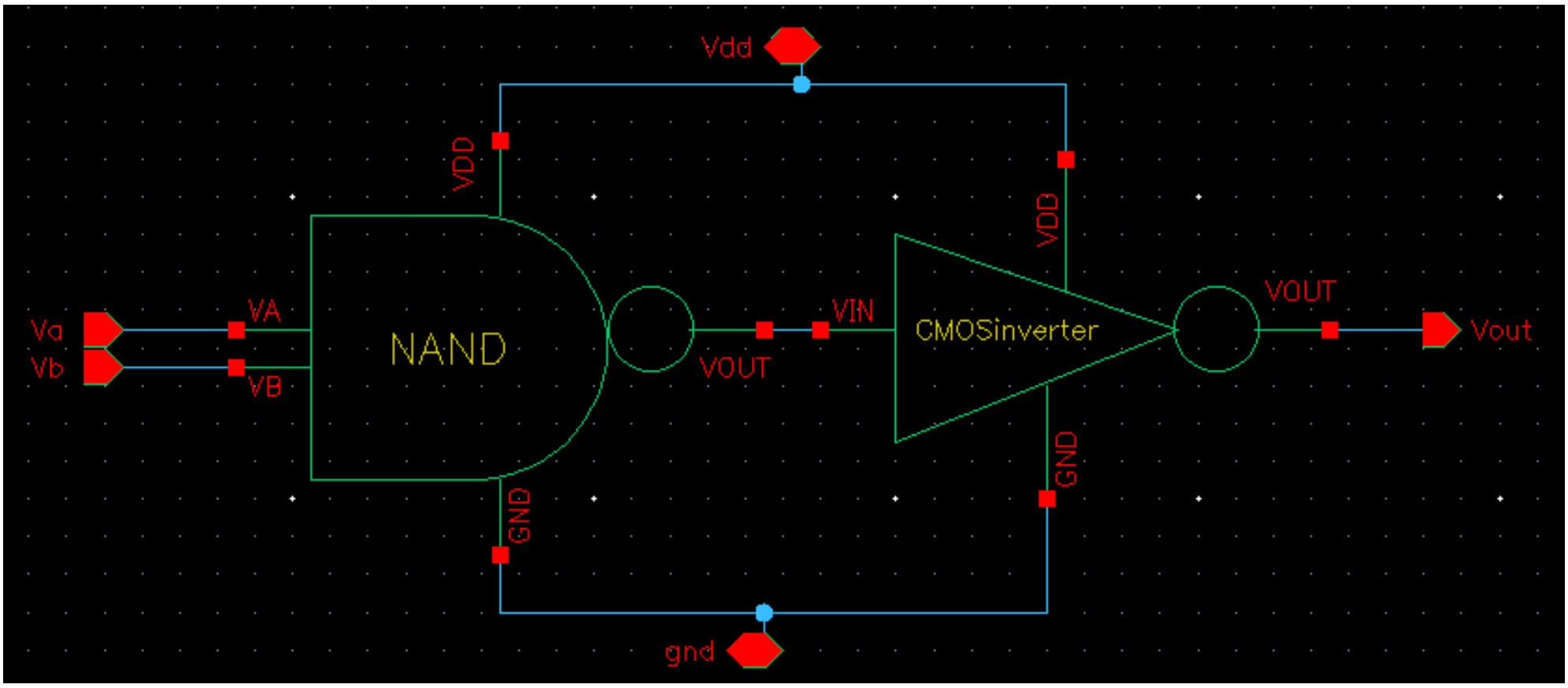



## 4.2 XNOR Symbol

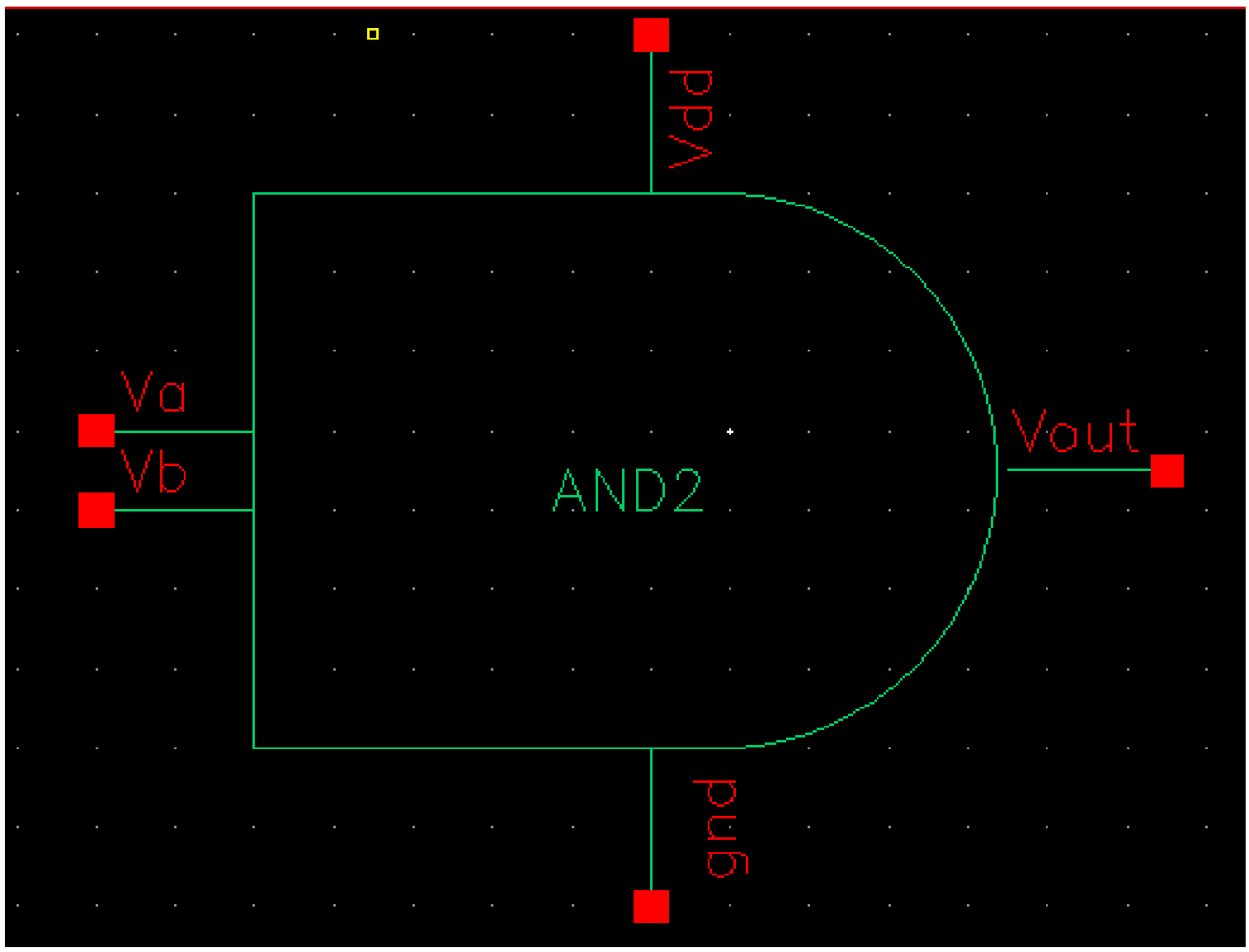


# 4.3 AND Schematic

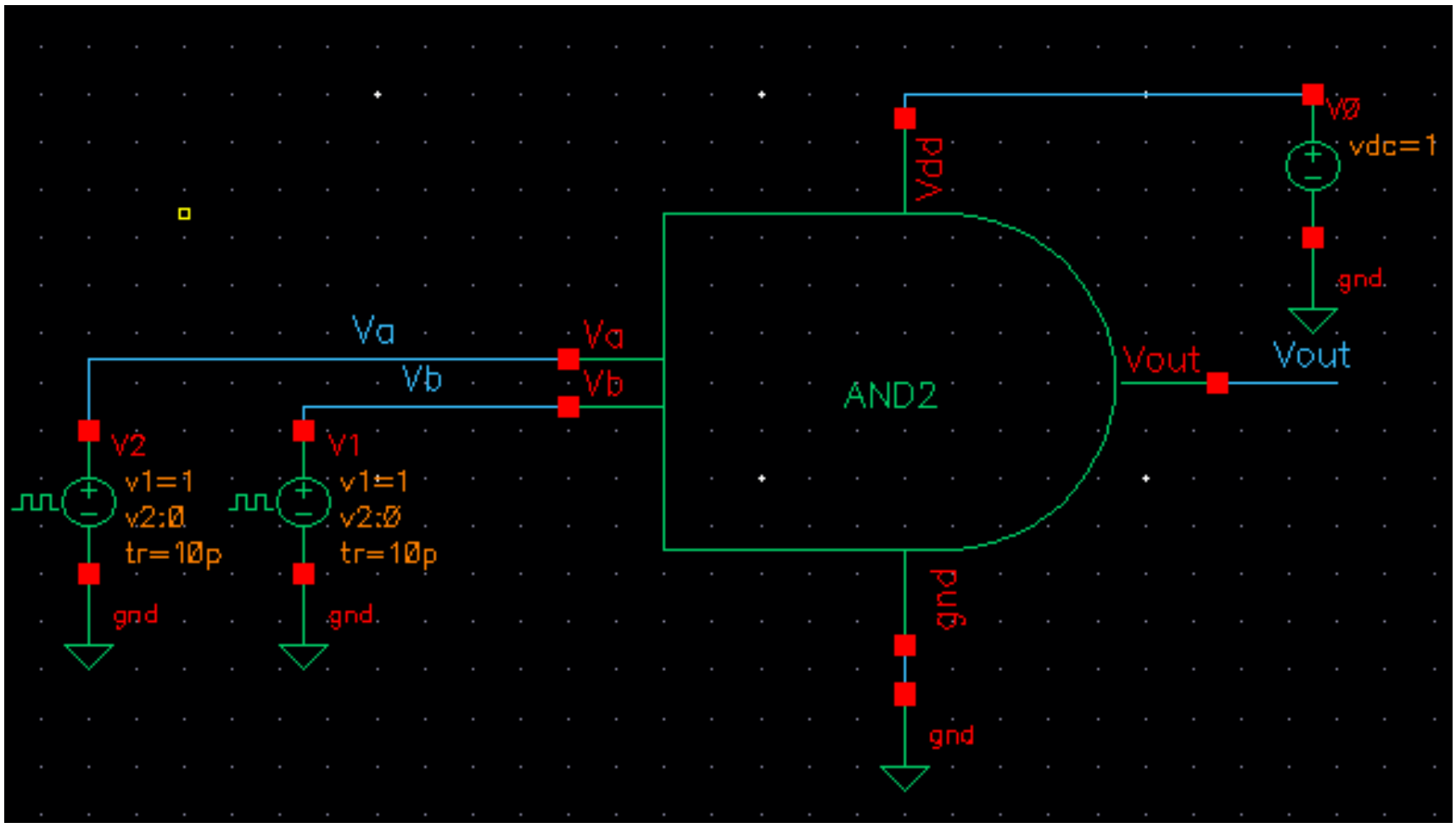
- By utilizing the NAND and NOR gates we have implemented, we can easily obtain the AND and OR functionality by incorporating an inverter after each respective gate. Additionally, the buffer functionality is achieved by employing a sequence of two inverters.

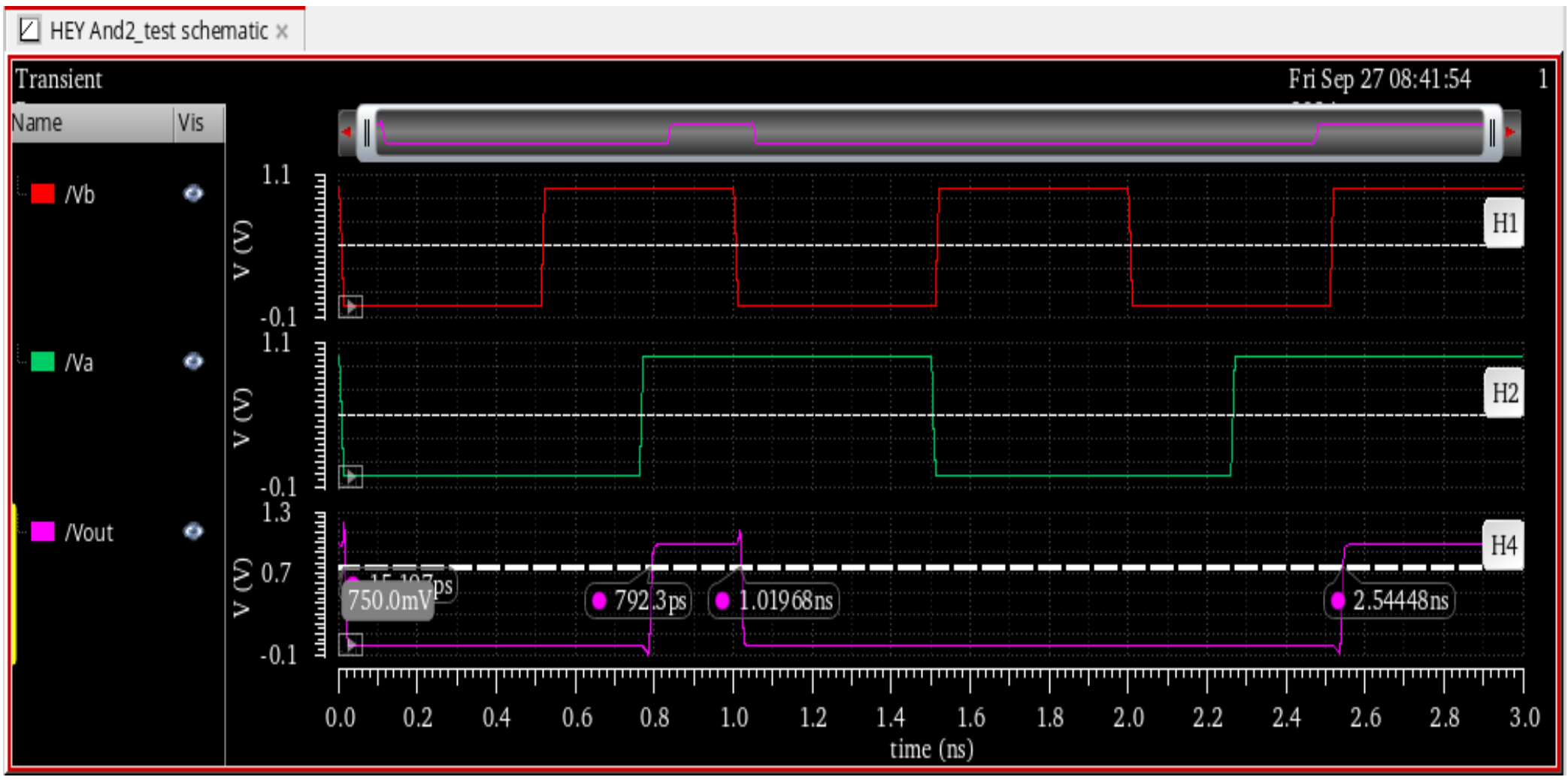



# 4.9 AND Symbol



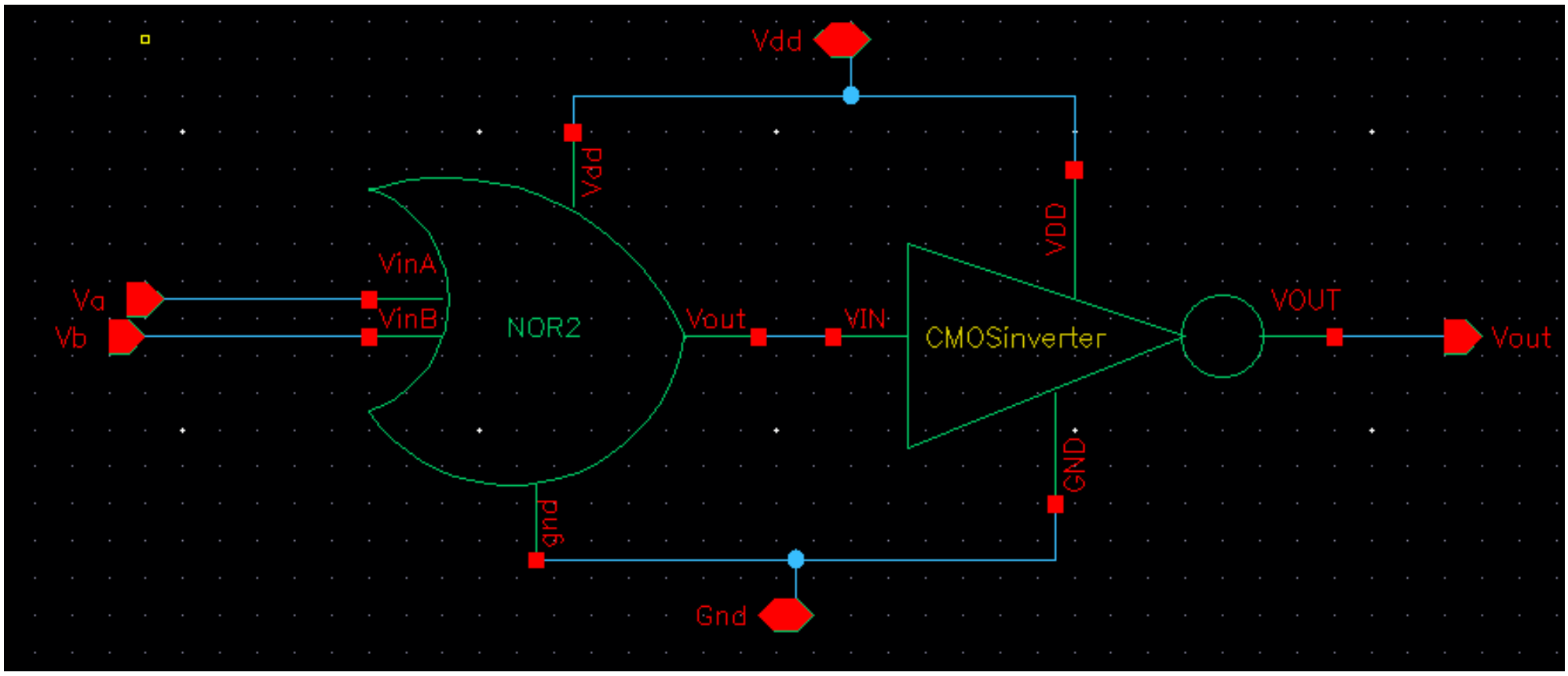
# 4.1 AND Testbench



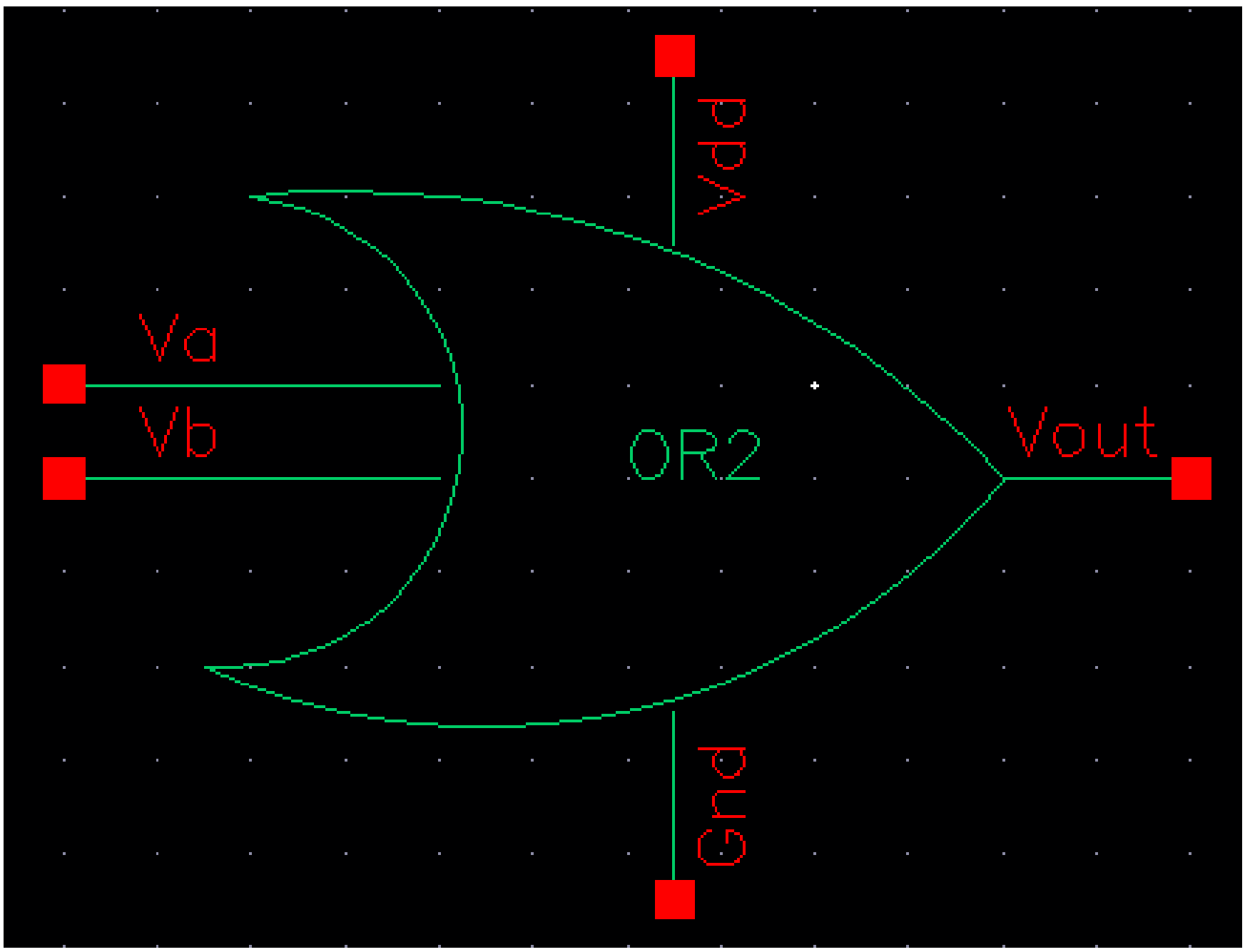




**So:** $\boldsymbol{Tpd = 14.68\,ps}$

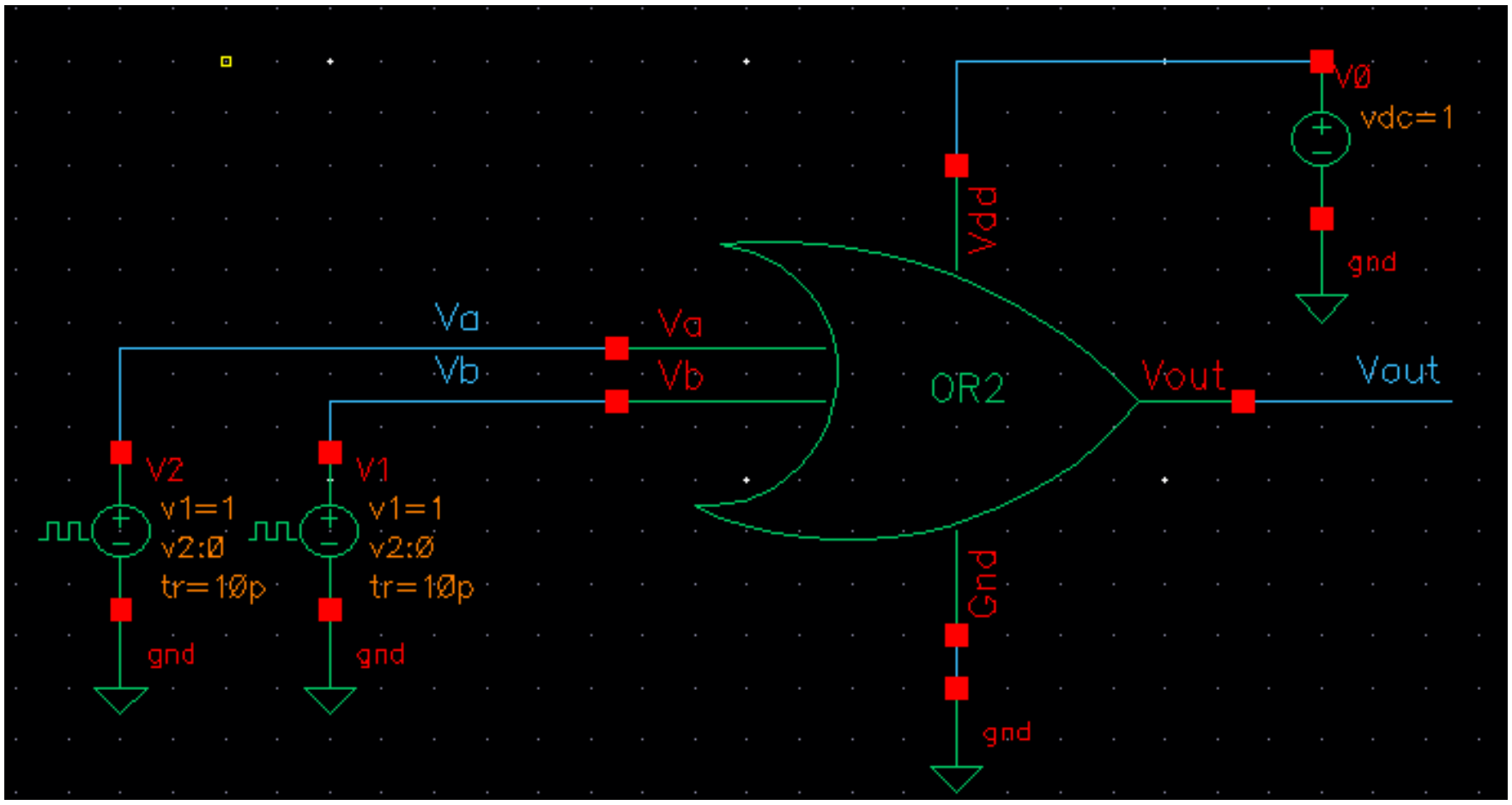
# 4.2 OR Schematic

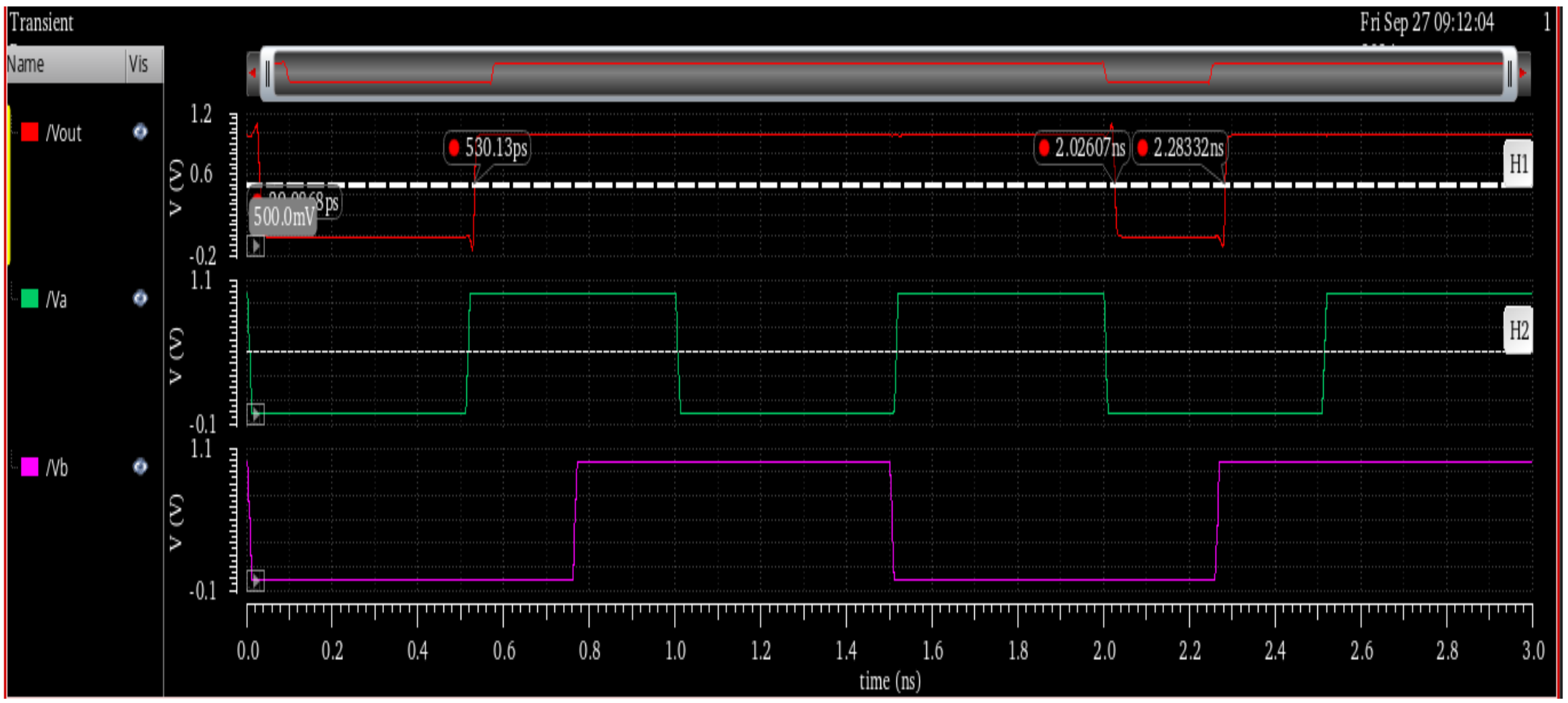



# 4.4 OR Symbol



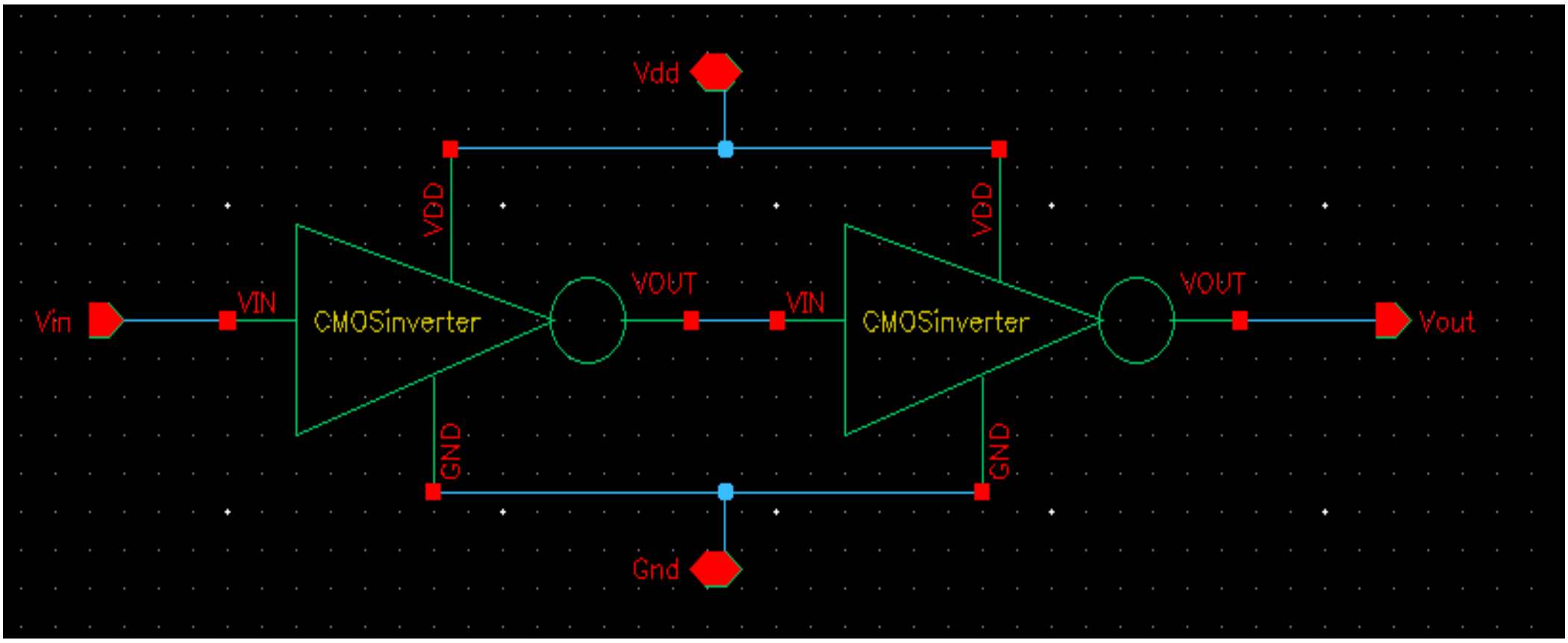
# 4.1 OR Testbench



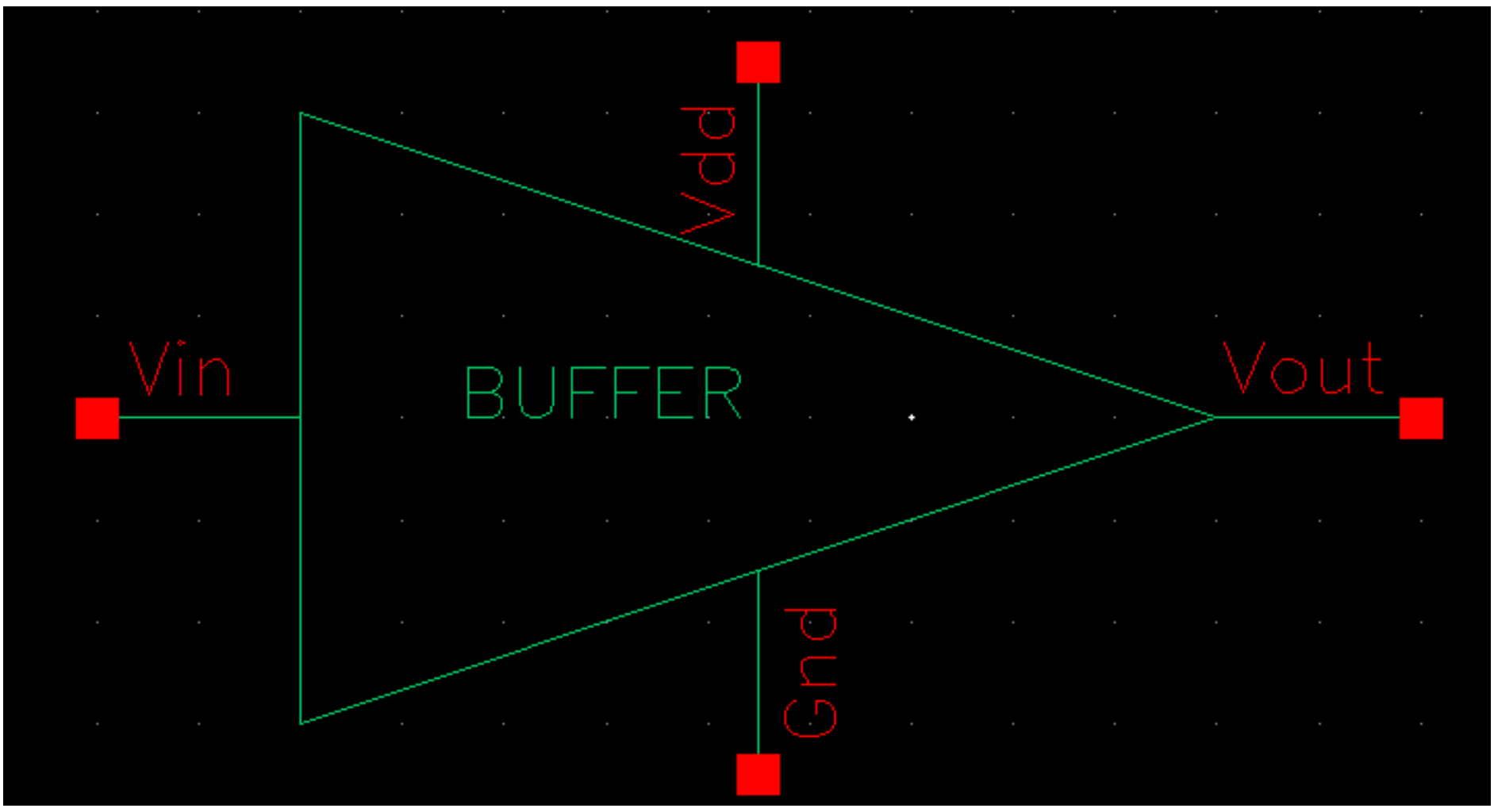




**So:** $\boldsymbol{Tpd = 18.1\,ps}$

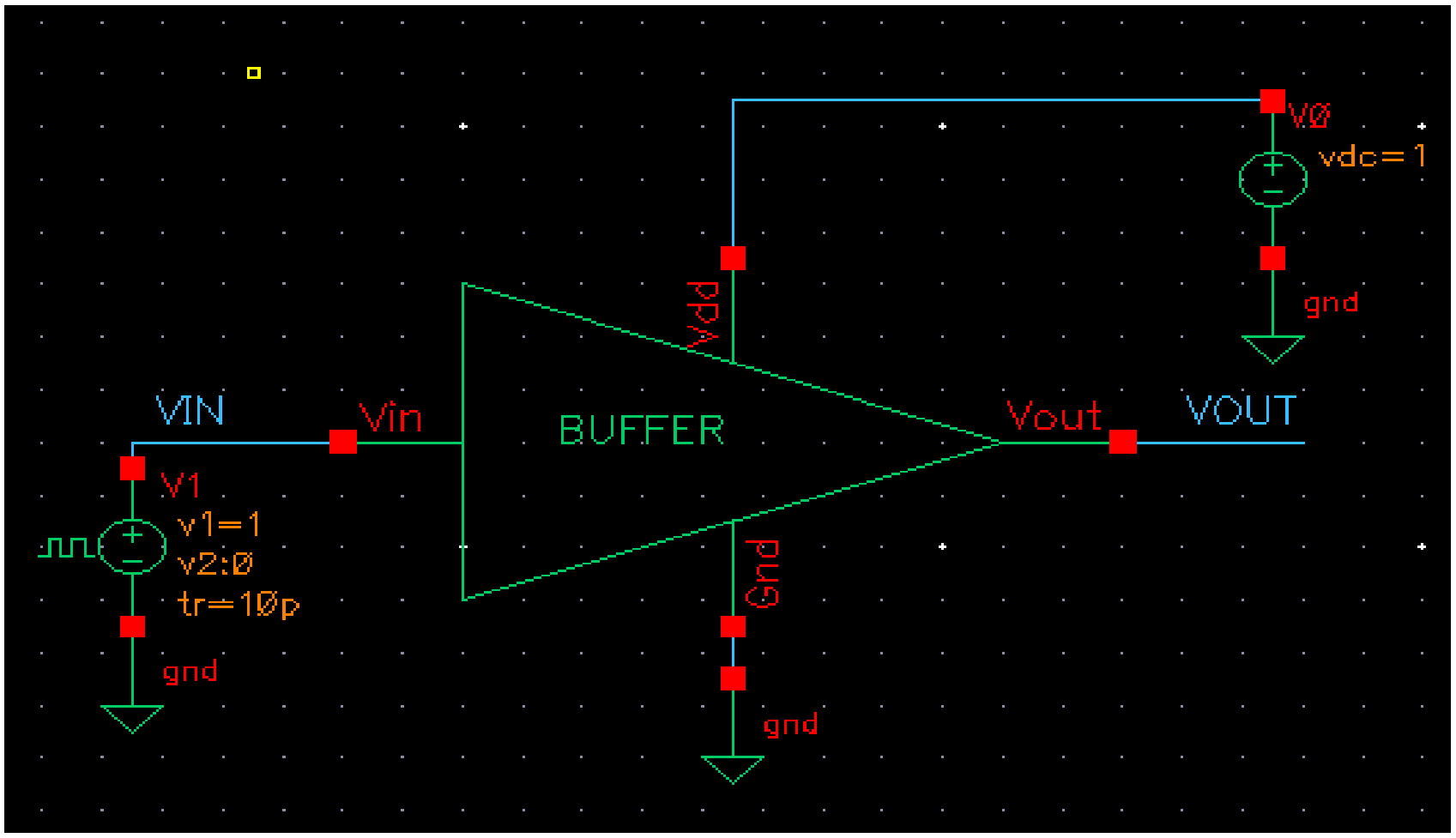
# 4.5 Buffer Schematic

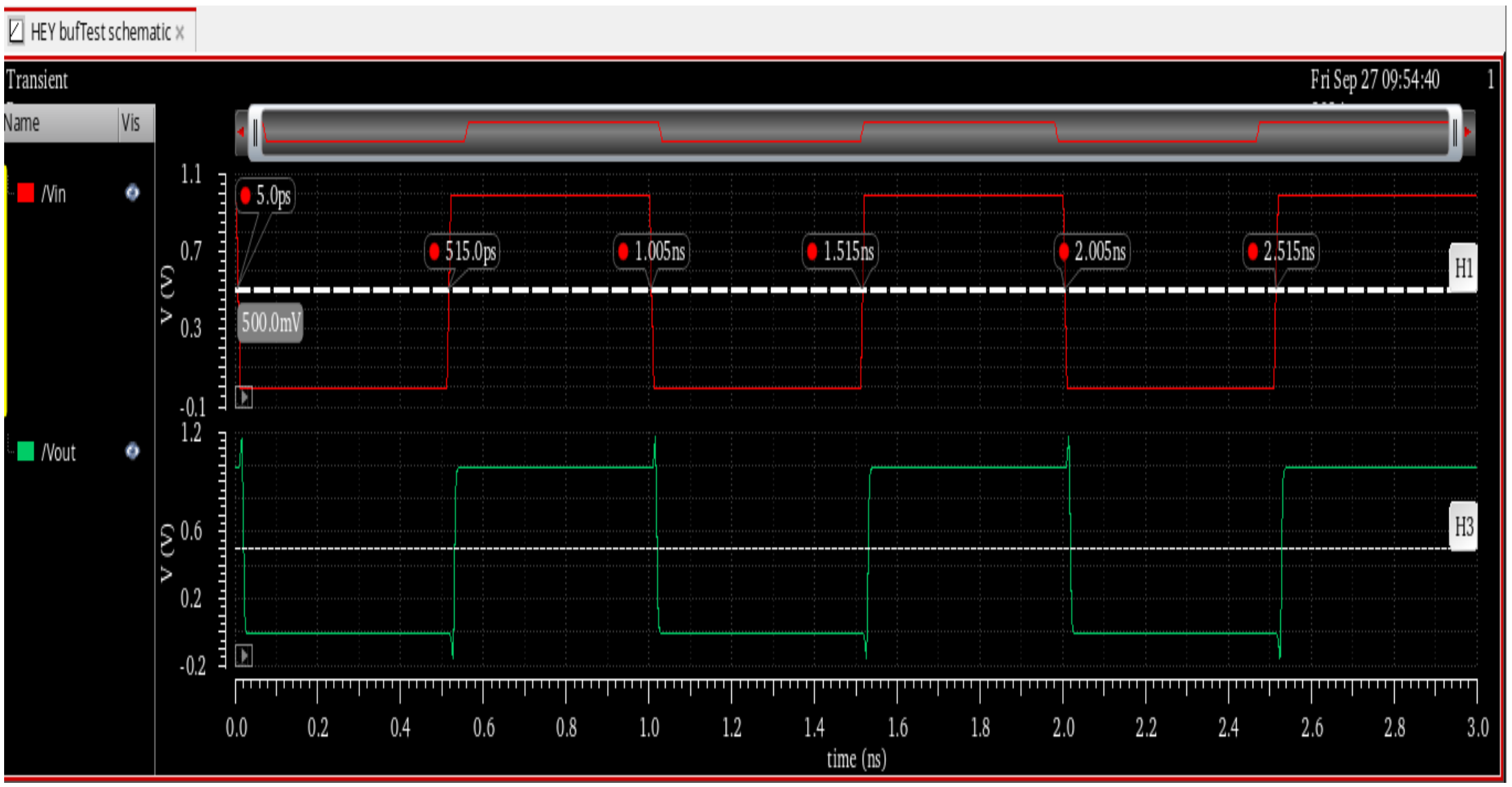



# 4.2 Buffer Symbol



# Buffer Testbench




So: $Tpd = 13.74\,ps$

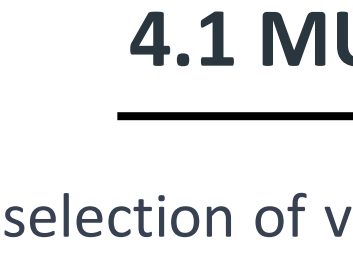
# 4.1 MUX 2x1 Schematic

- To enable the selection of various options, it is necessary to design a multiplexer (MUX) with different configurations such as 2x1, 4x1, or 8x1. To accomplish this, we begin by designing the fundamental building block, which is the 2x1 MUX.
- (TG Multiplexer) is the most efficient one
- The truth table corresponding to the 2x1 MUX is as follow

## 2-to-1 Multiplexer

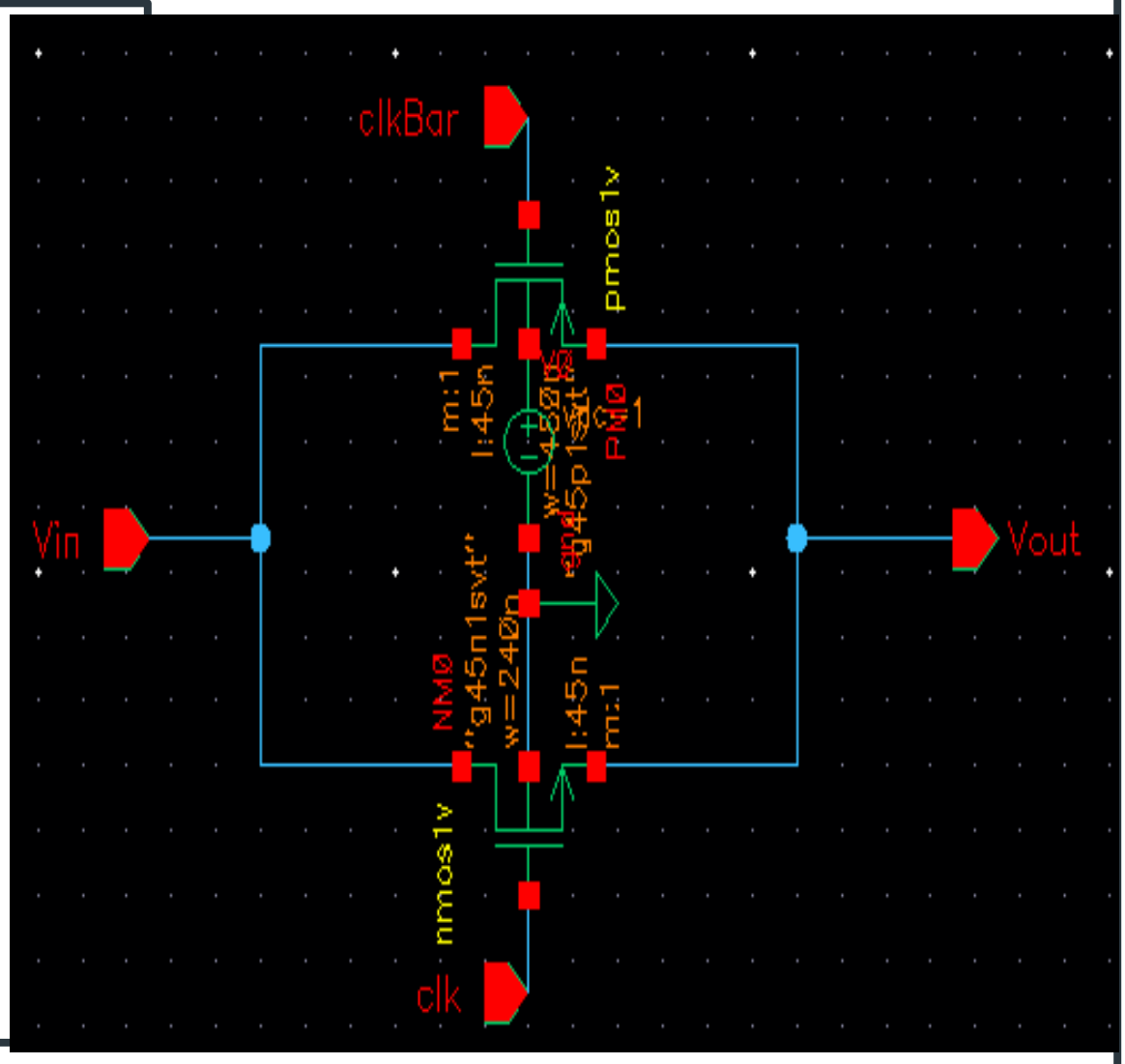
| S | $I_0$ | $I_1$ | Y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

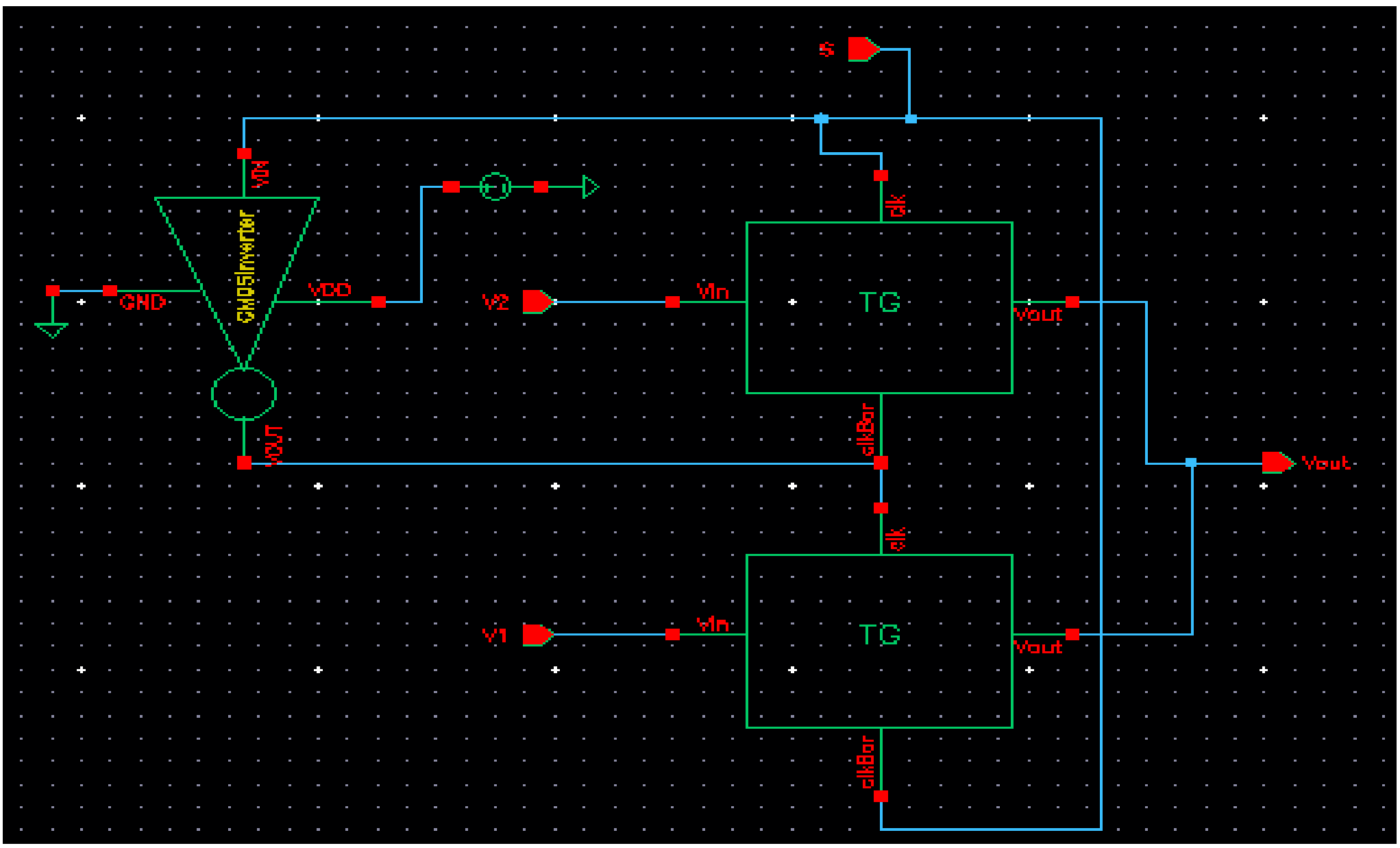
Truth table for 2x1 MUX



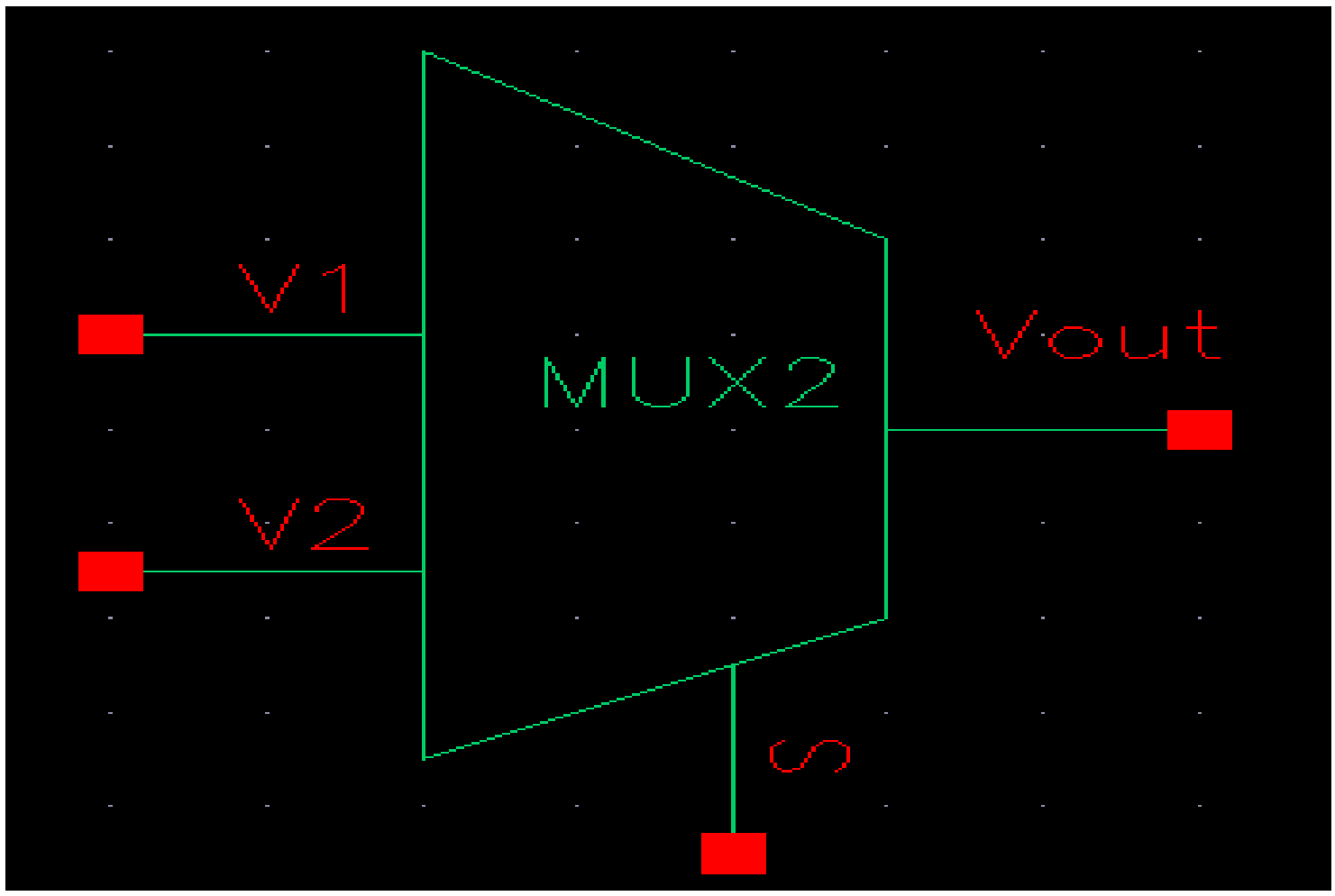

| S | Y |
|---|---|
| 0 | $I_0$ |
| 1 | $I_1$ |

Condensed Truth Table

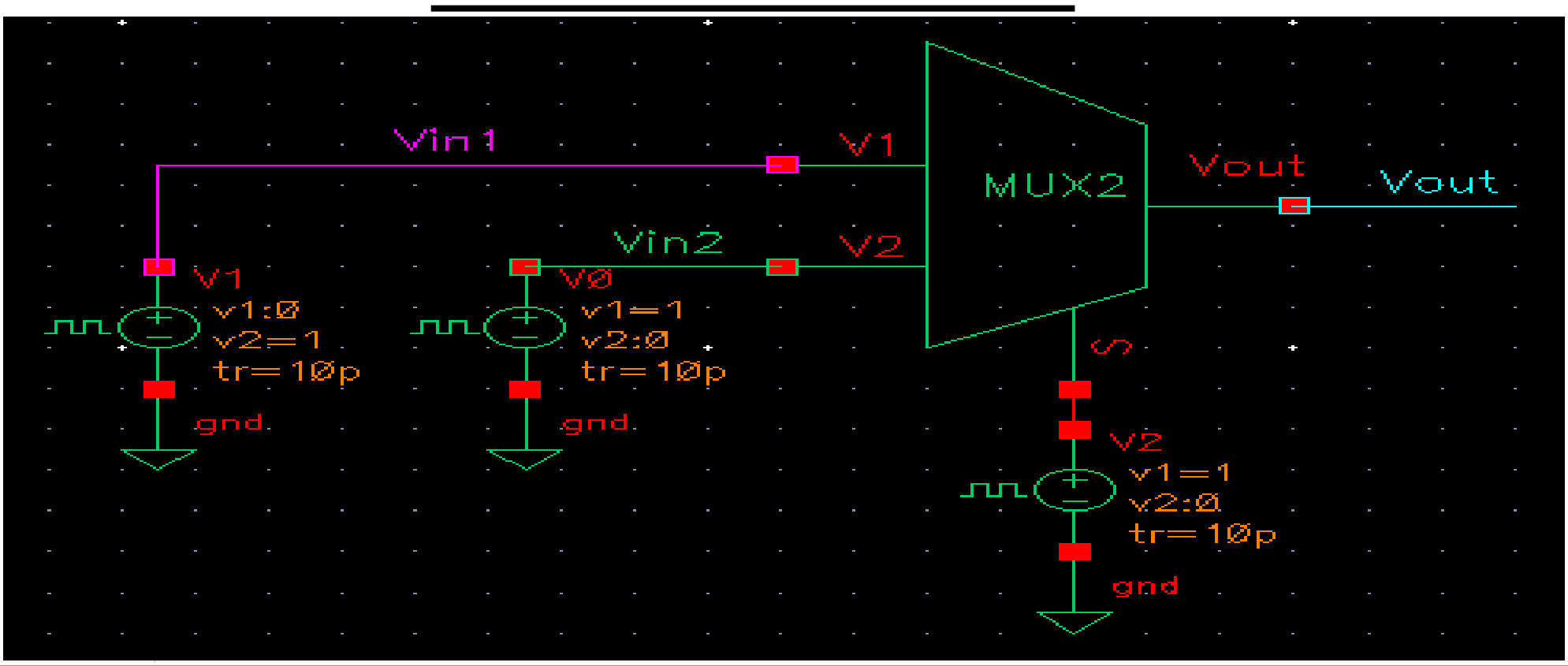


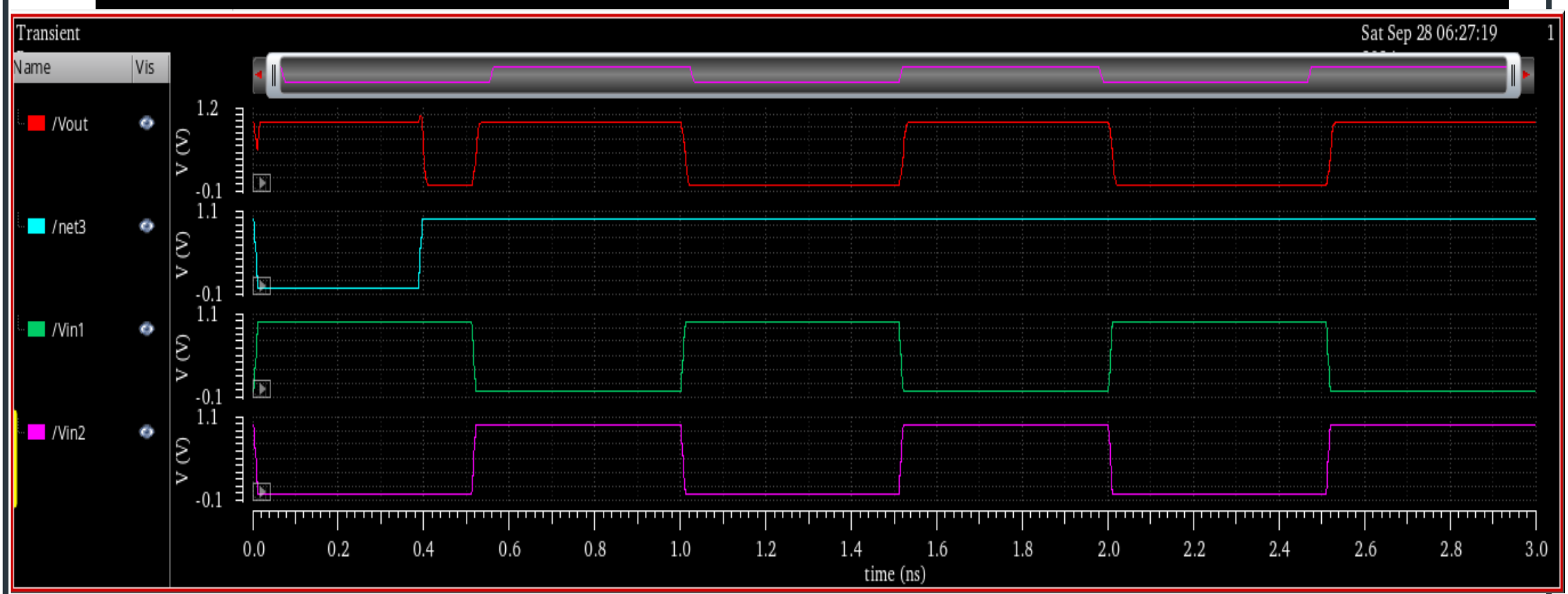



# MUX 2x1 Symbol



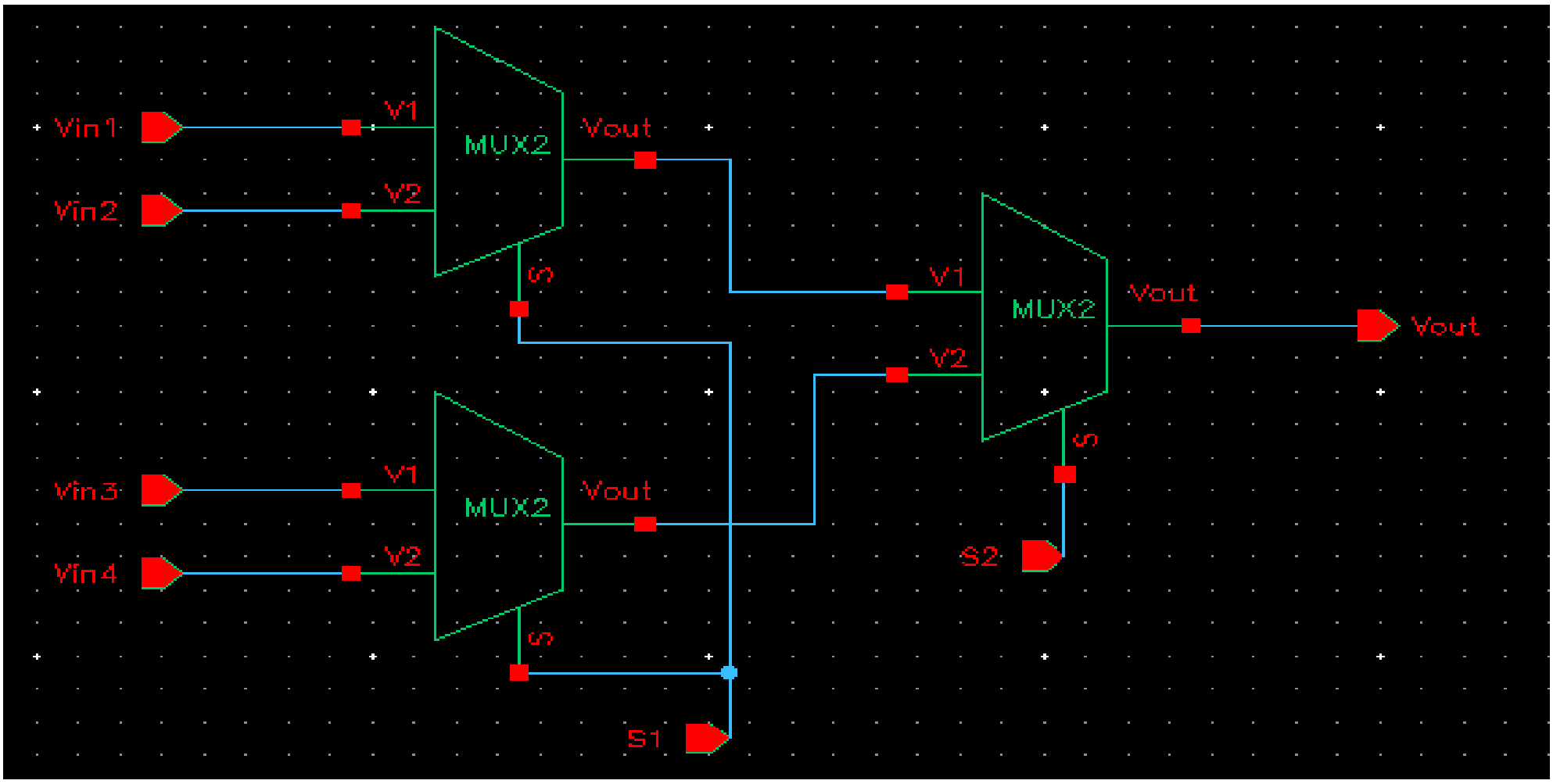

# MUX 2x1 Testbench



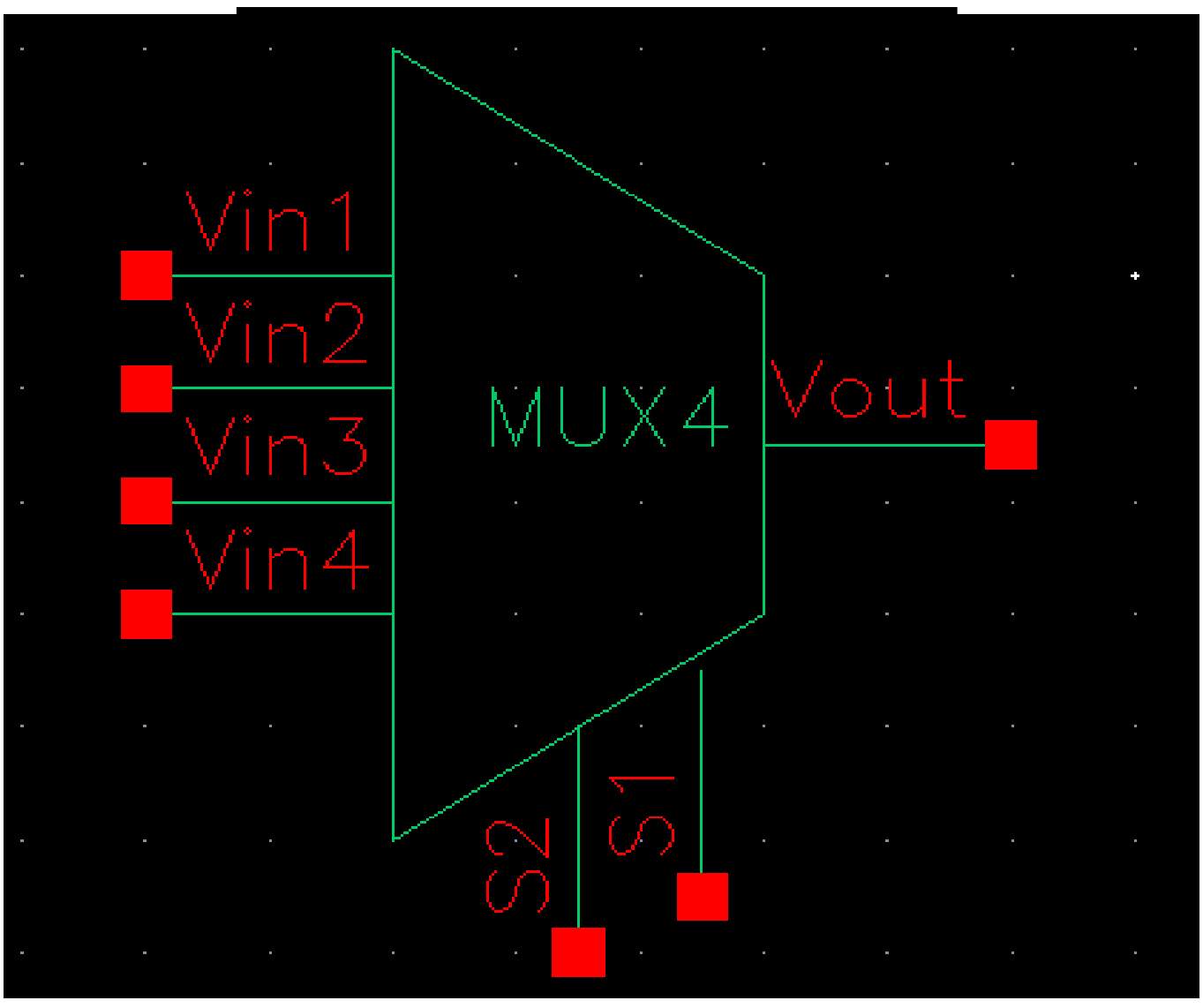



# 4.2 MUX 4x1 Schematic

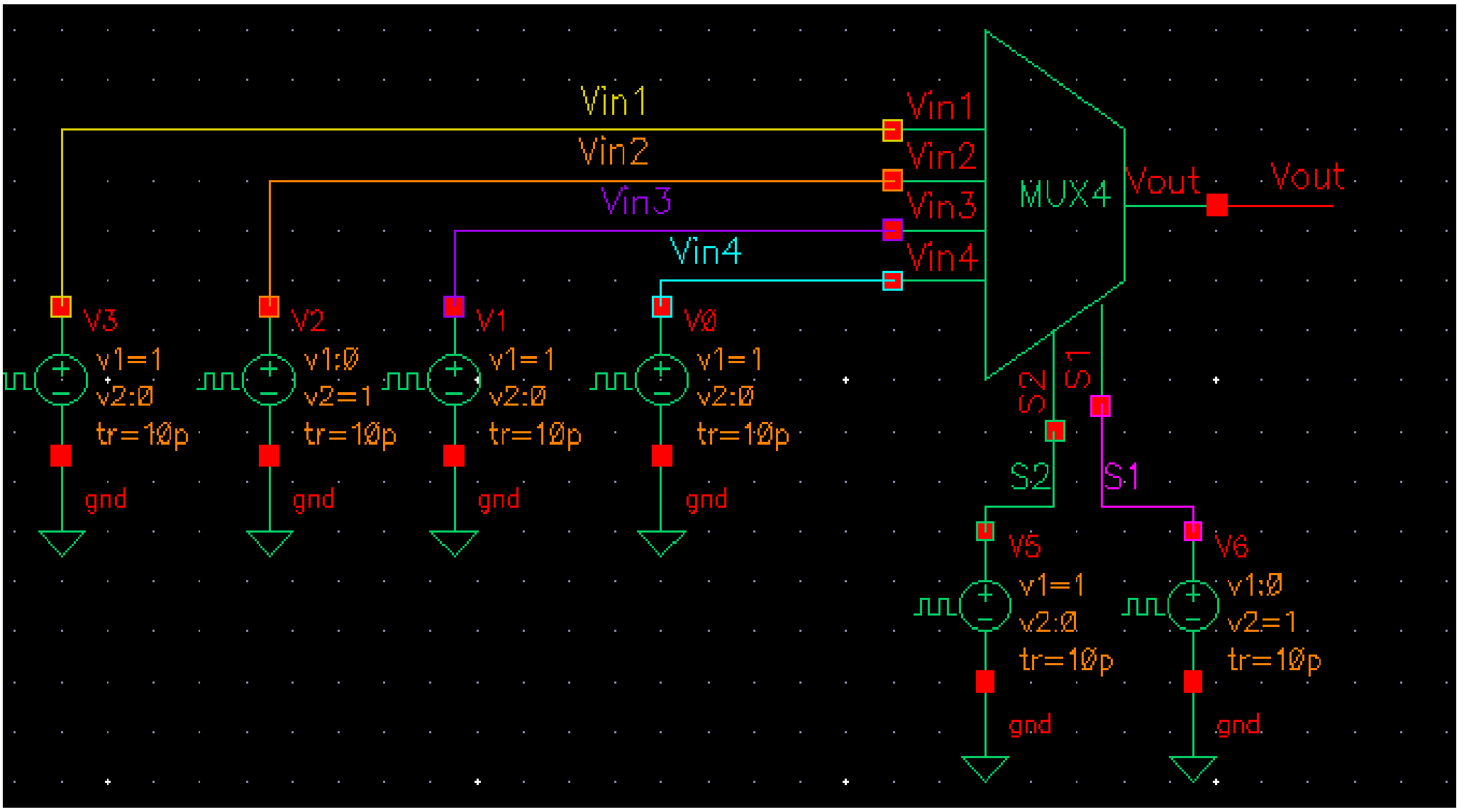
- Having established the foundational building block, we can now proceed to design larger multiplexers (MUXs) with expanded input options. By utilizing the 2x1 MUX as a building block, we can construct MUXs of various sizes, such as 4x1, 8x1, and beyond, to cater to our specific needs and requirements.

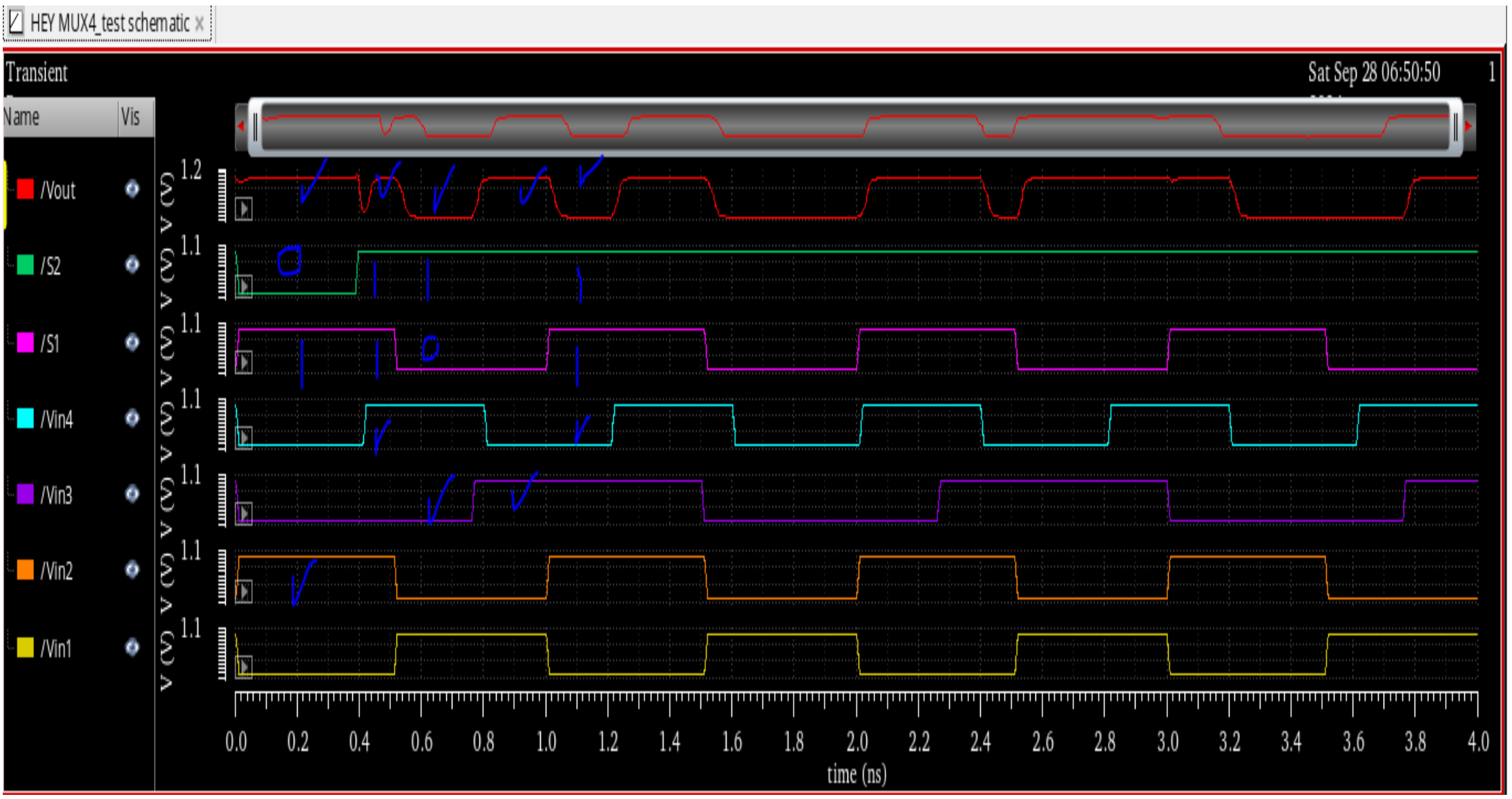



# 4.6 MUX 4x1 Symbol

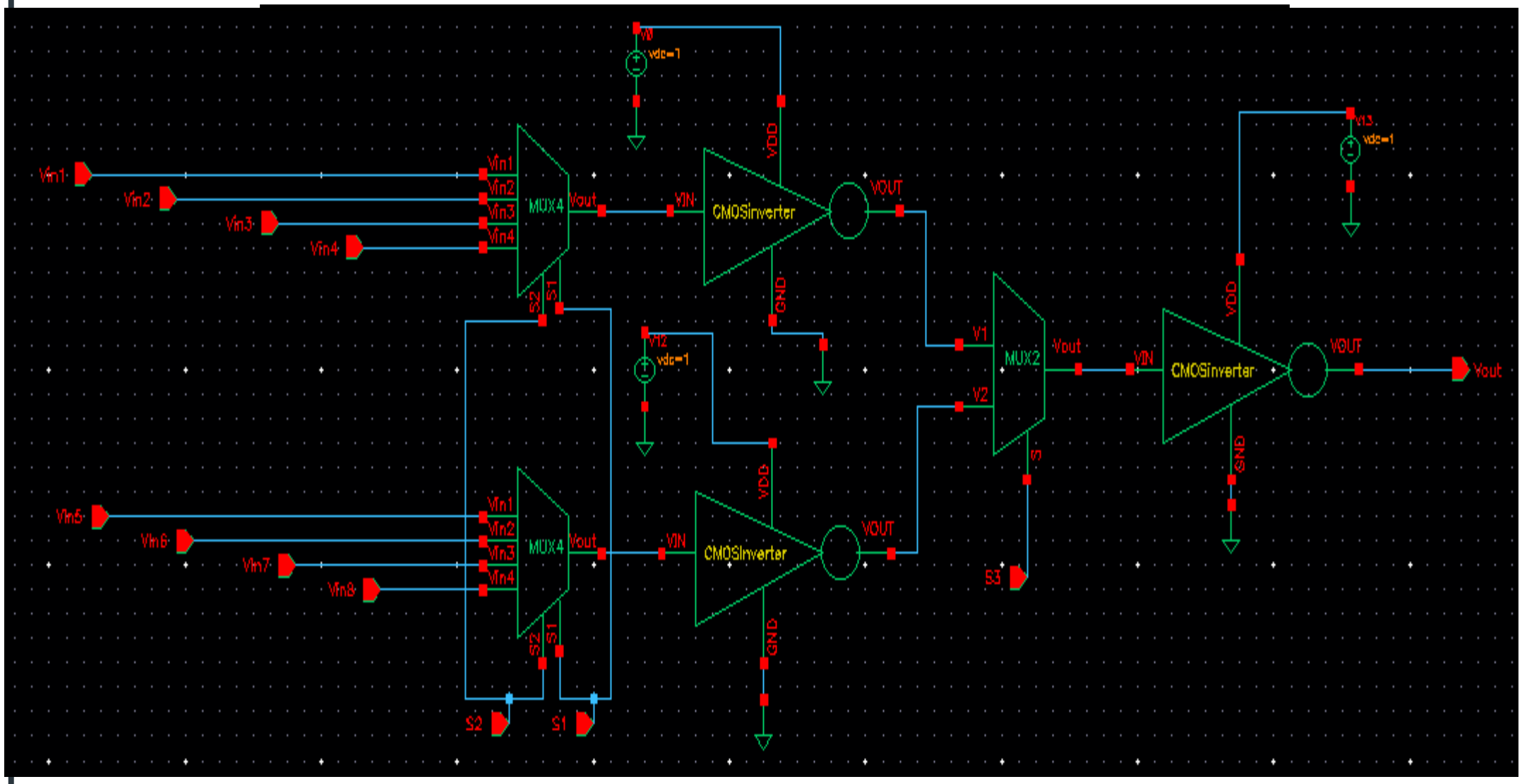


# 4.1 MUX 4x1 Testbench

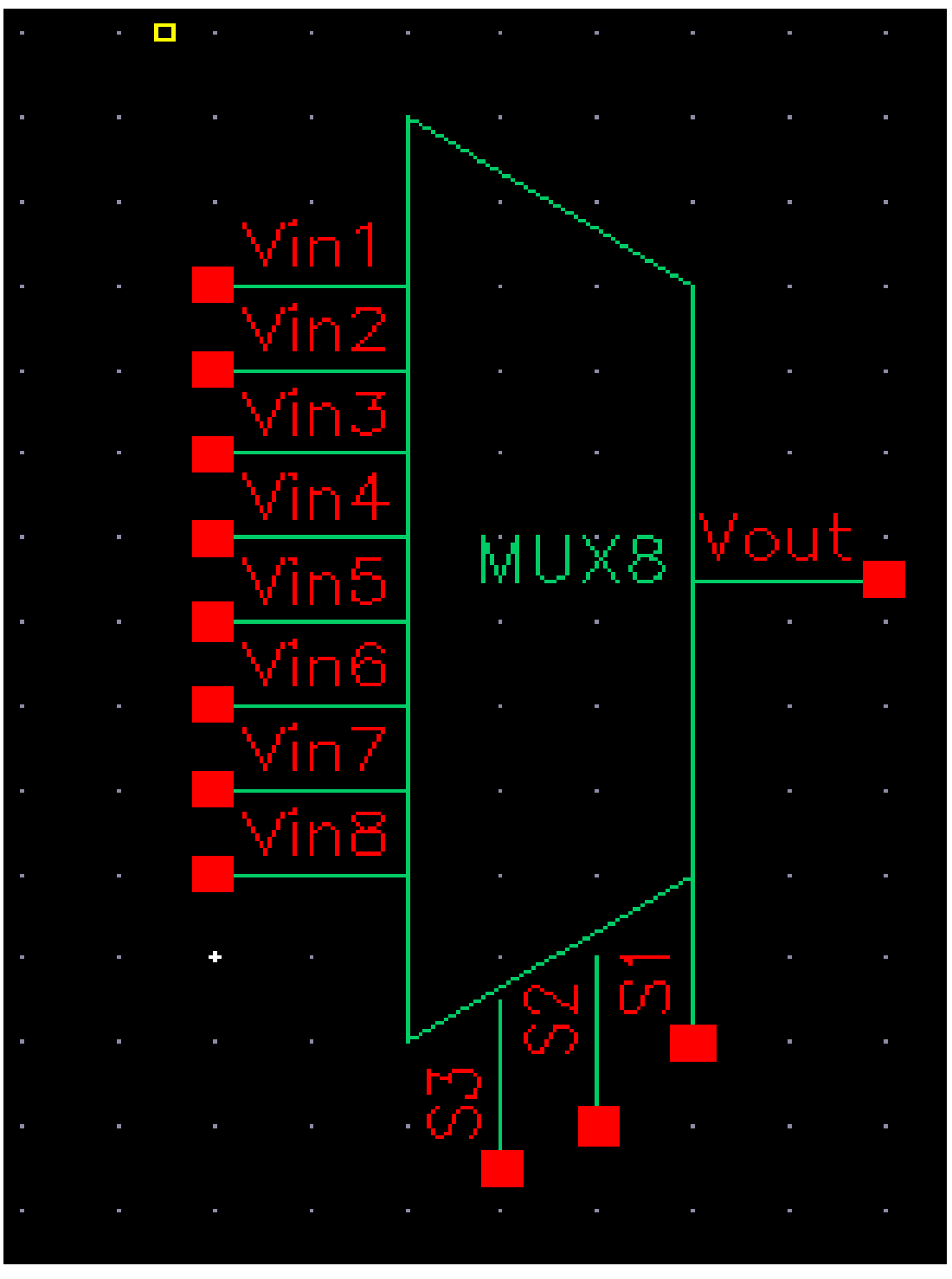


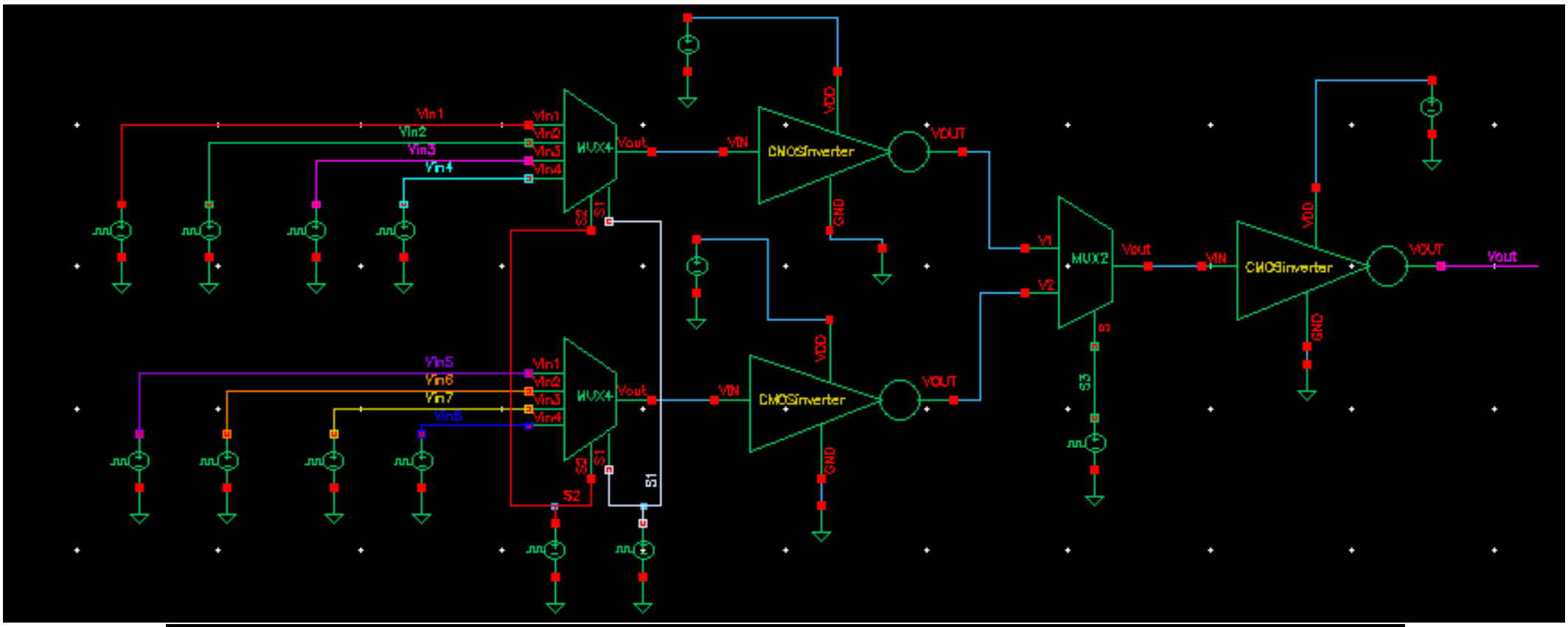
# 4.2 MUX 8x1 Schematic

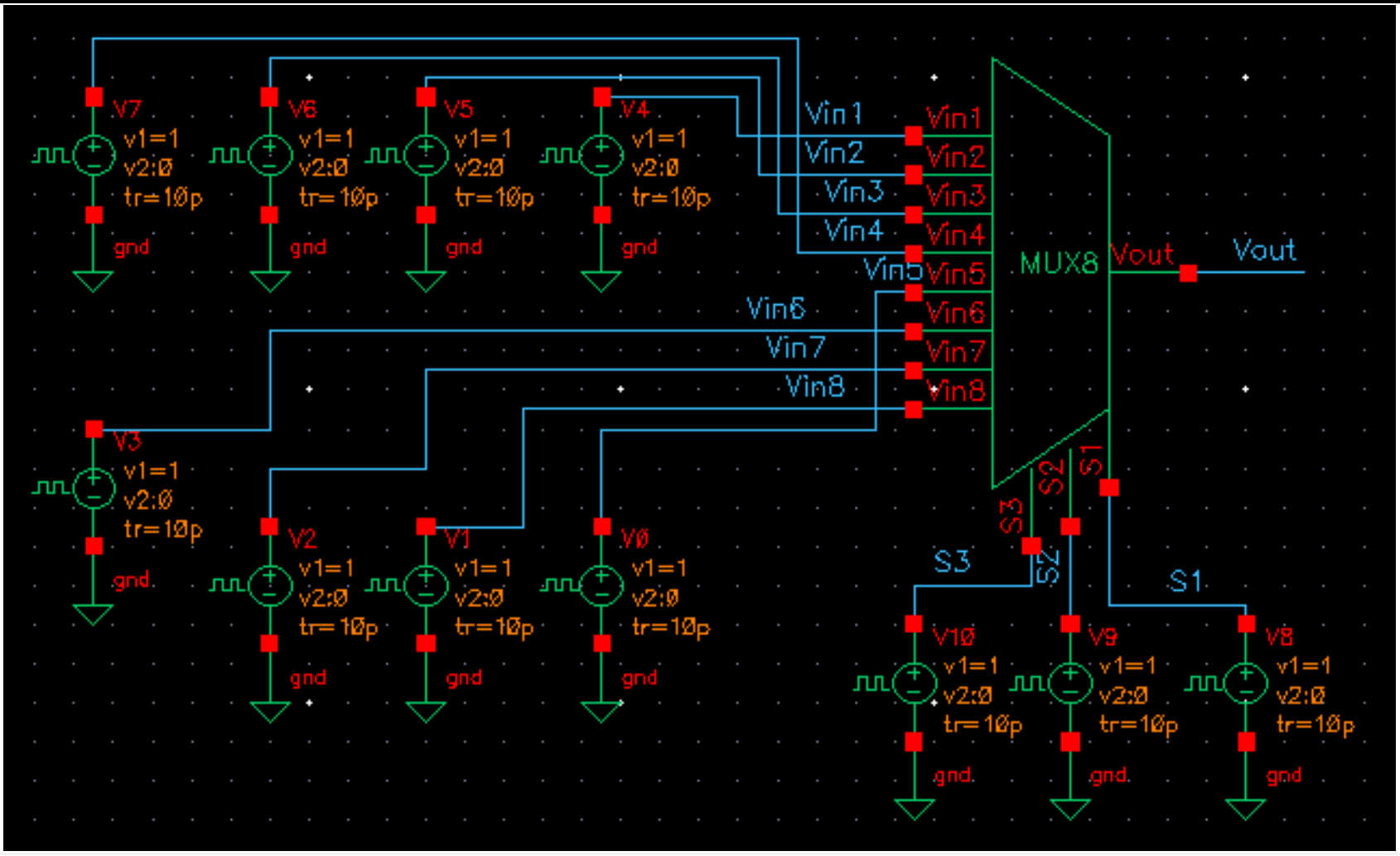


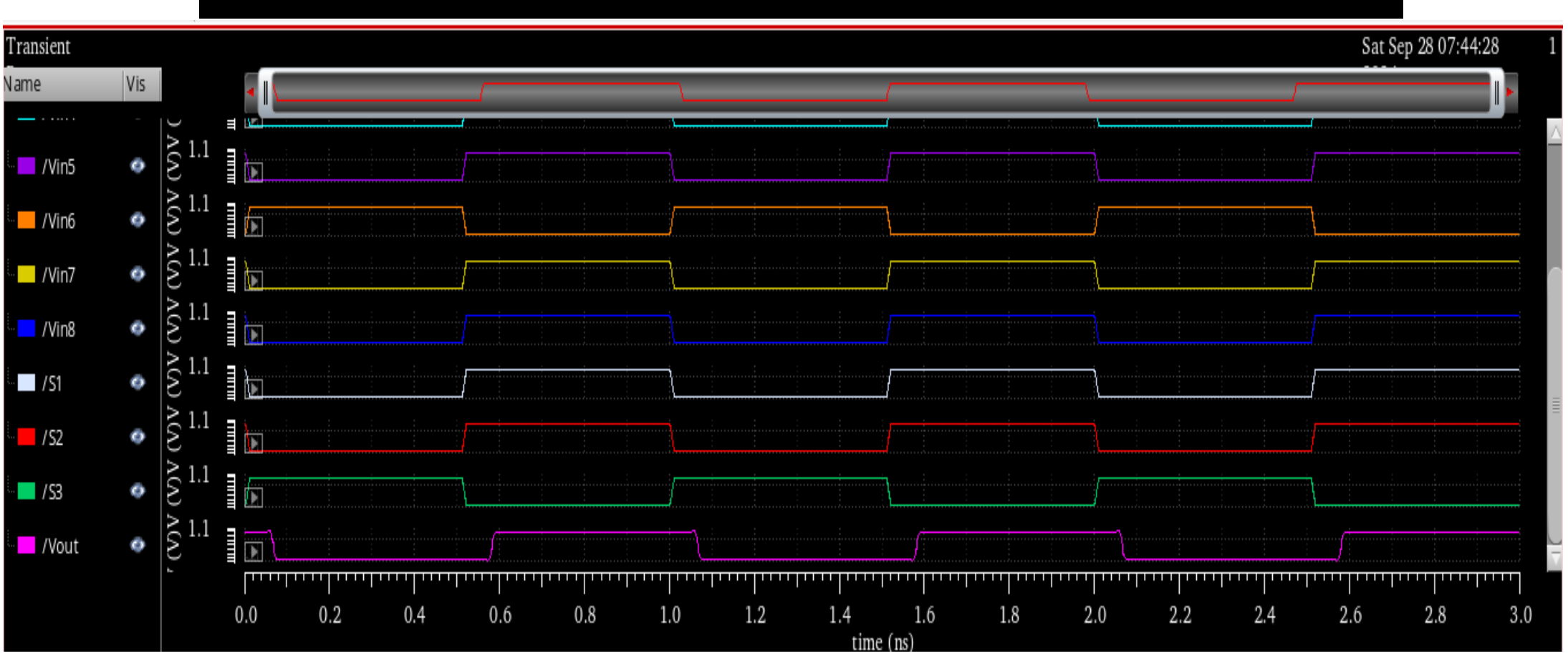

# 4.3 MUX 8x1 Symbol



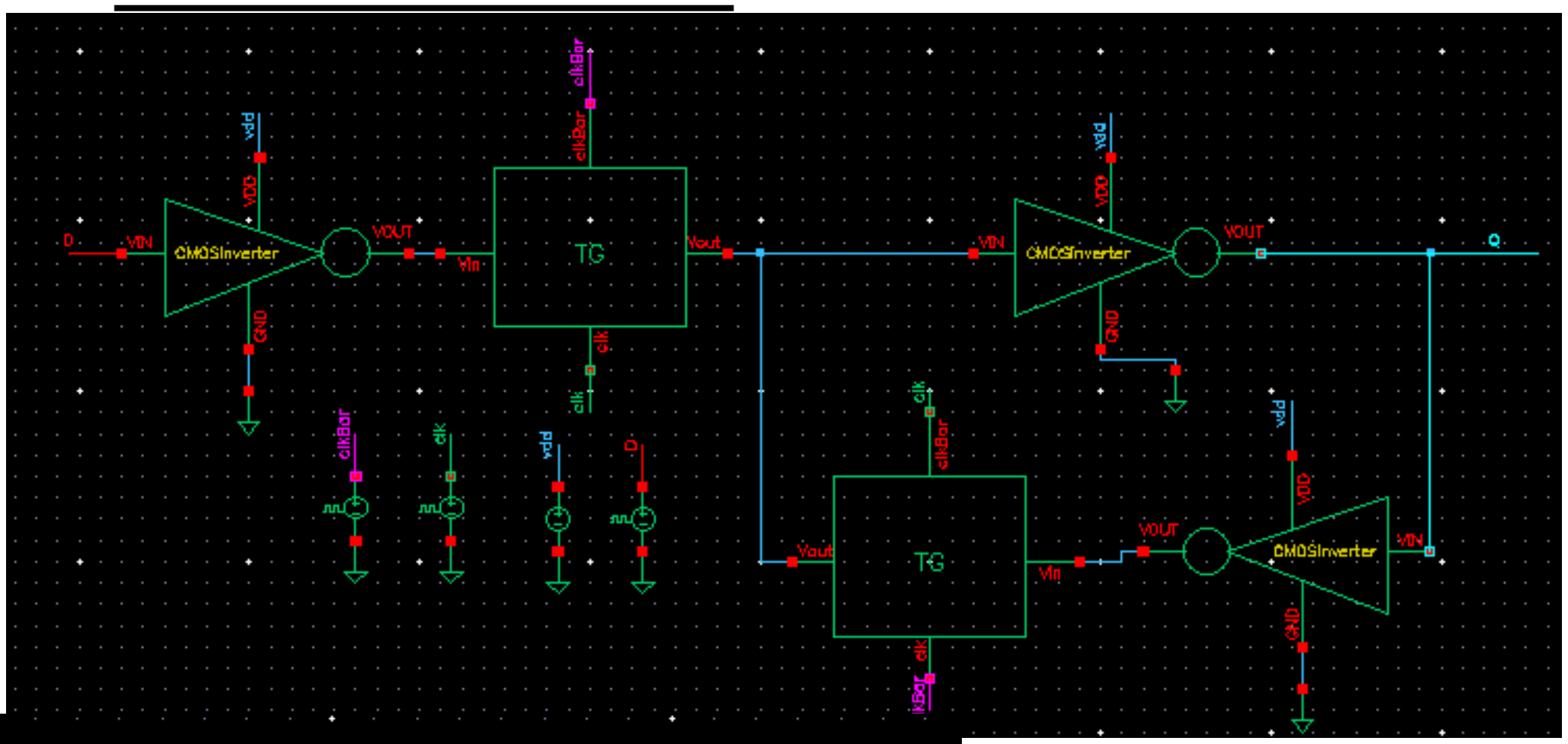
# MUX 8x1 Testbench

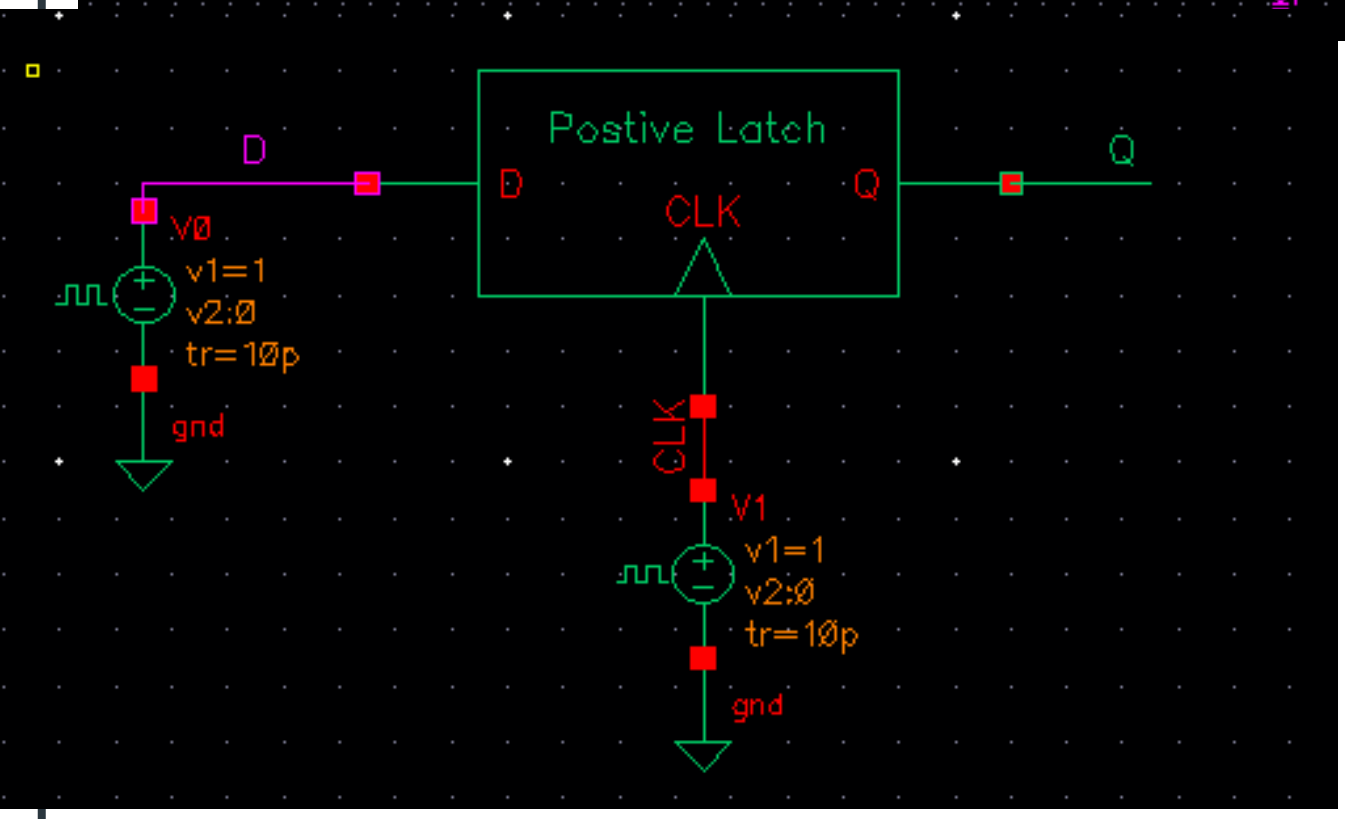


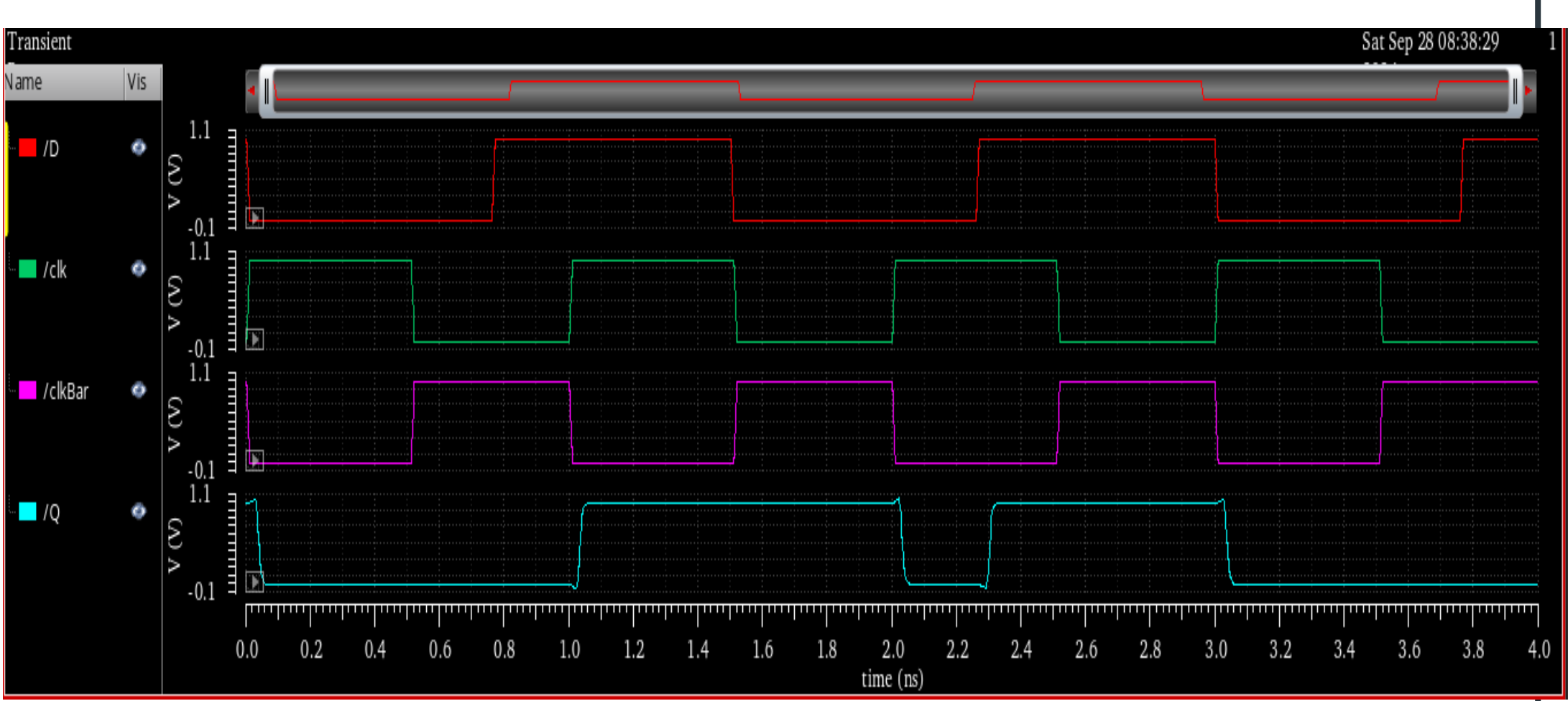



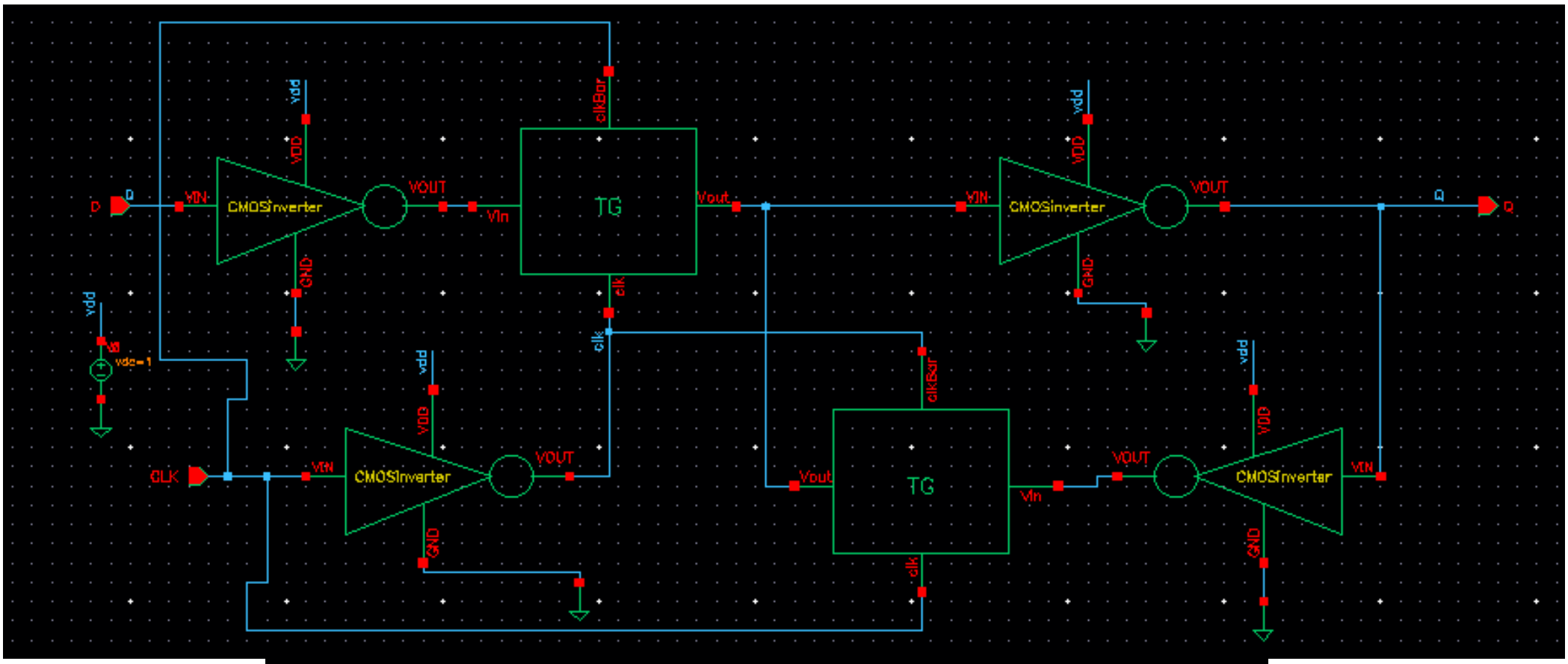



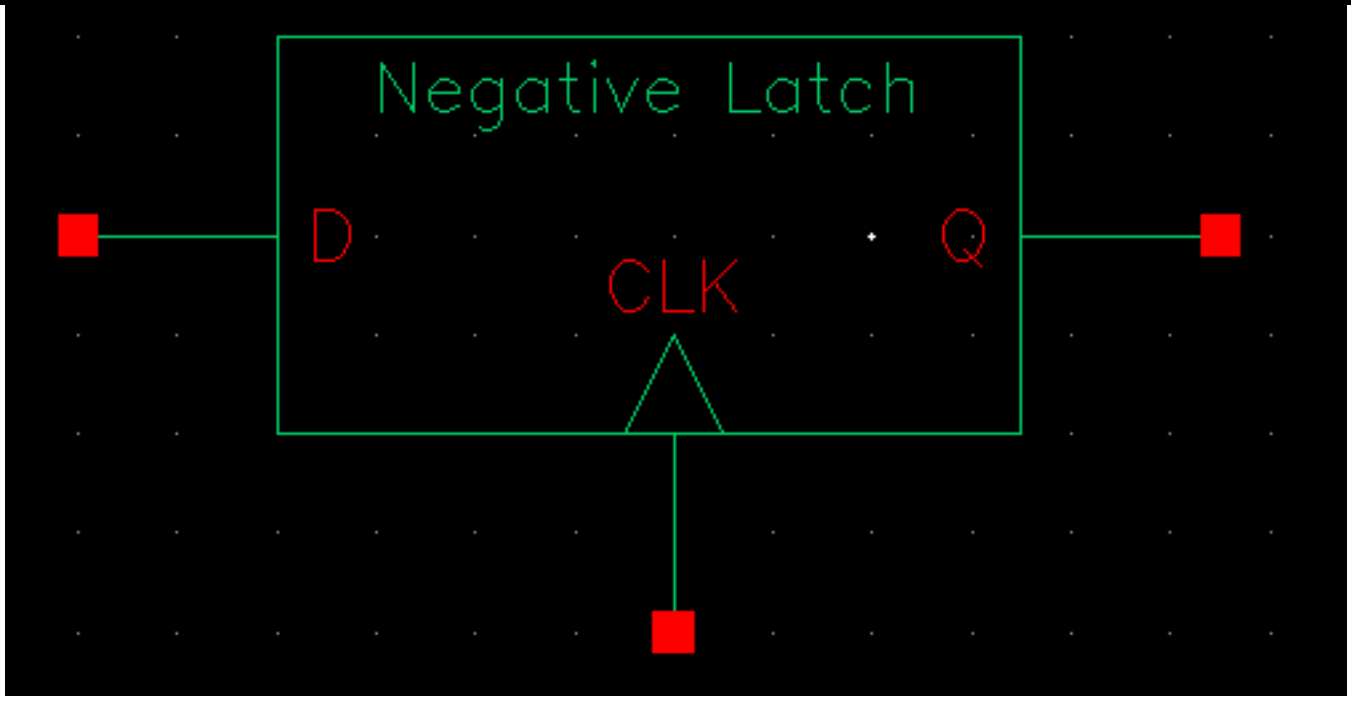
# Flip Flops Schematic

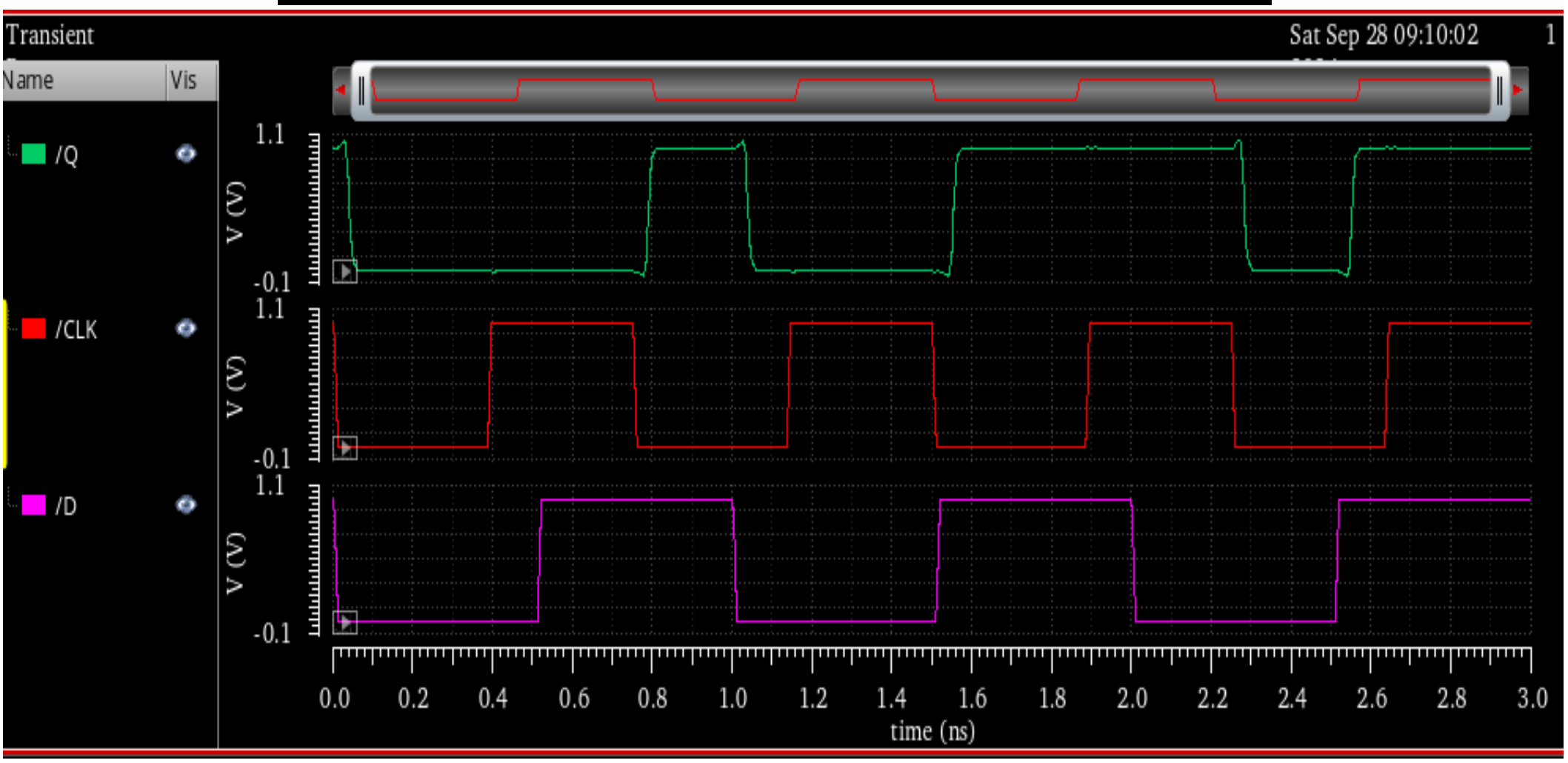


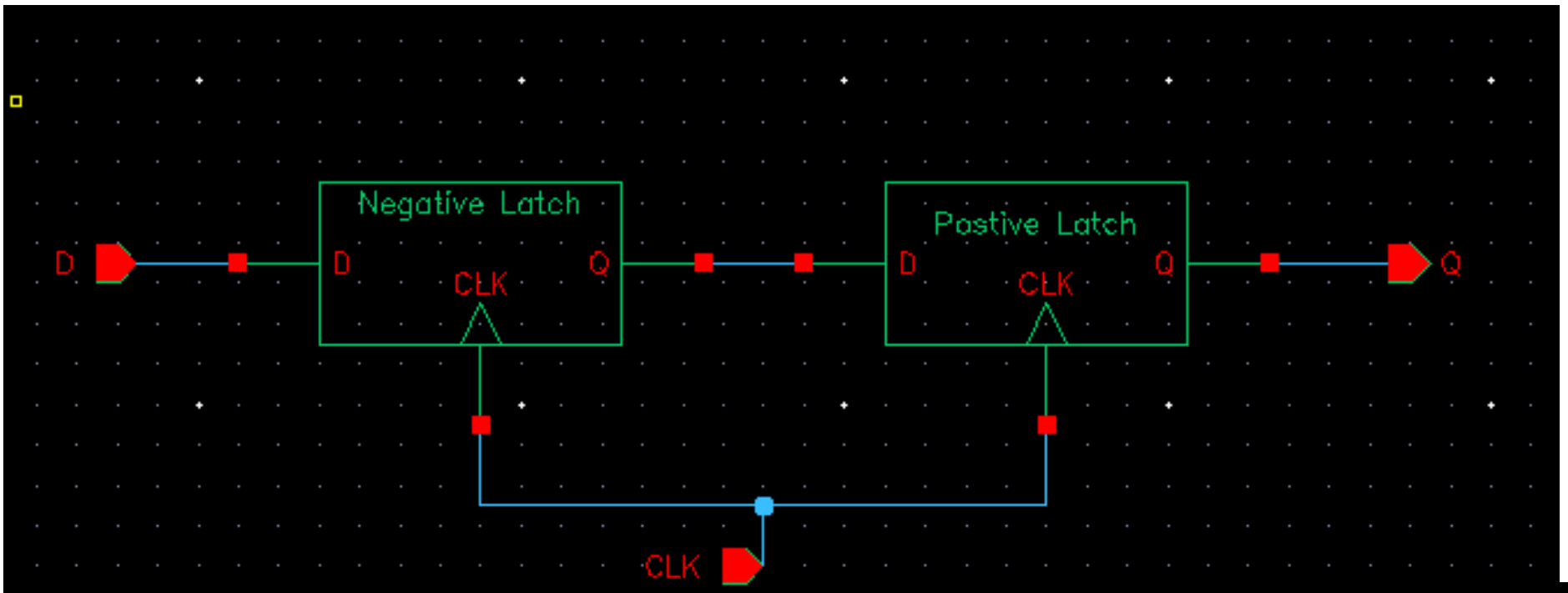



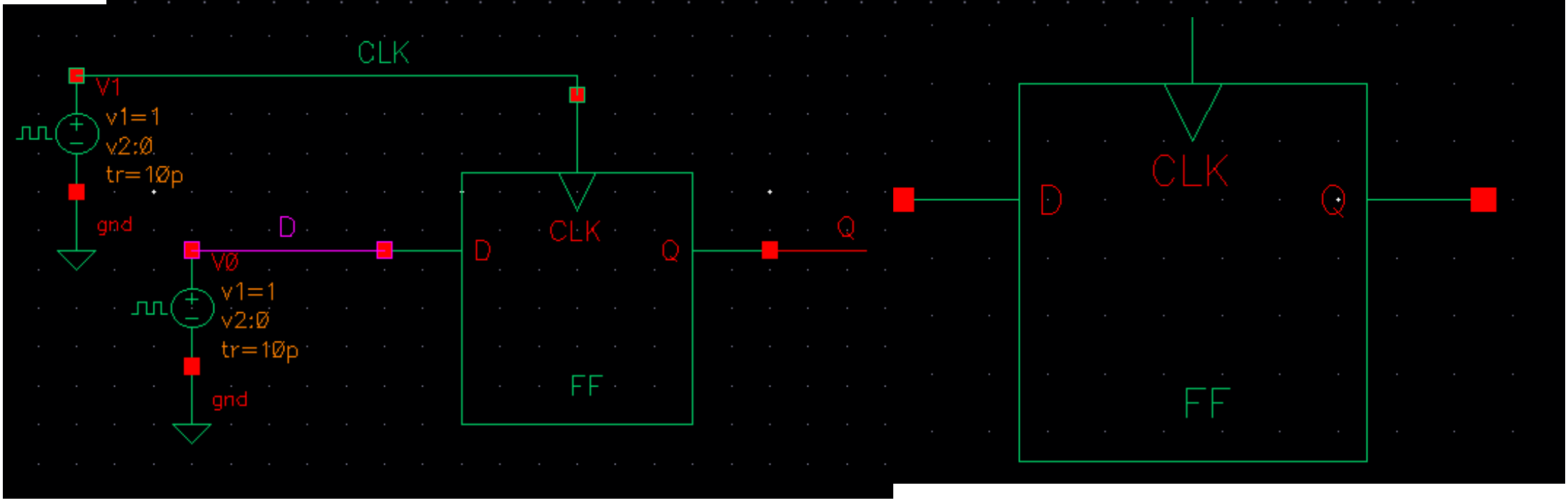



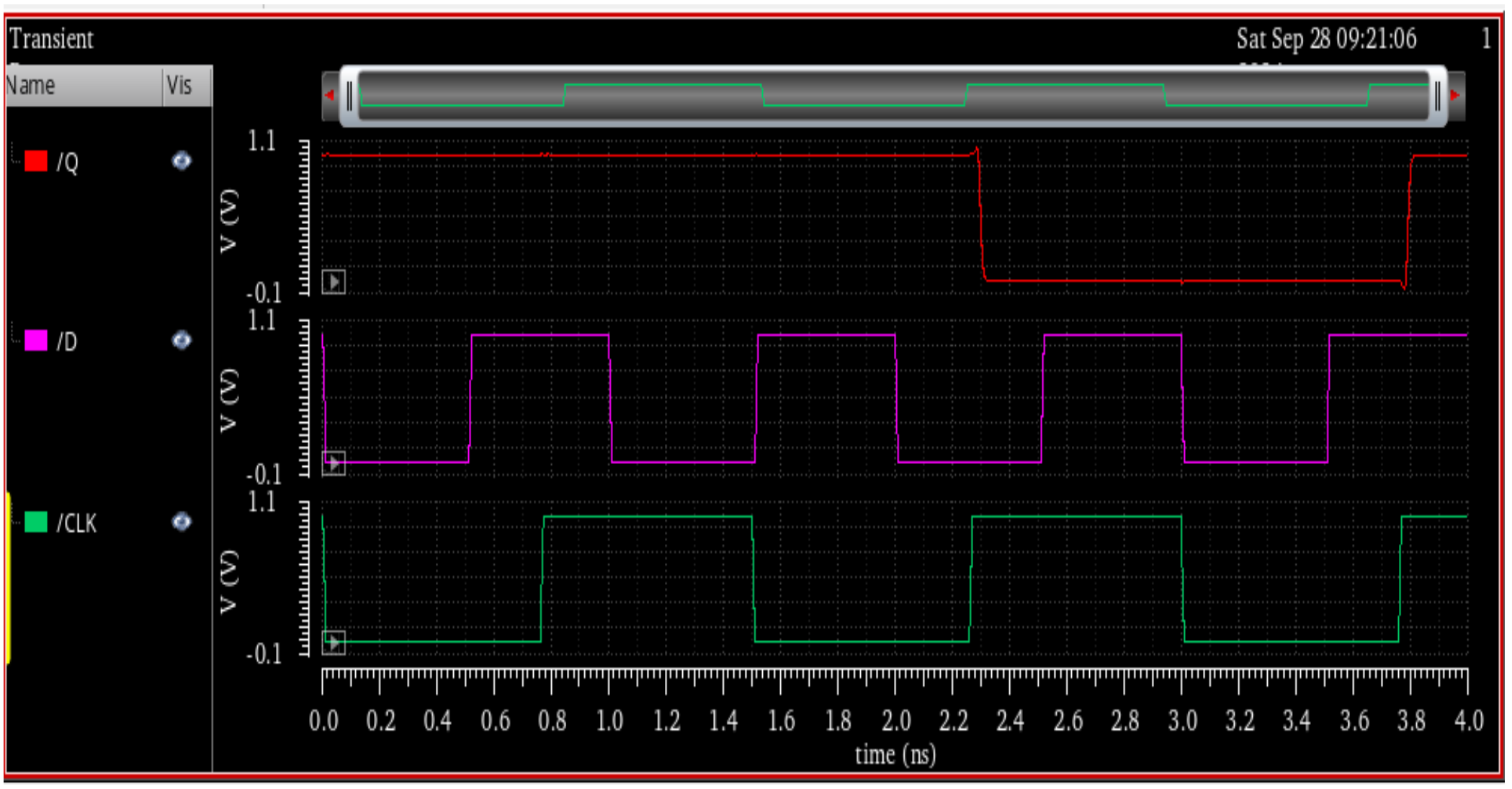

D
Q
VIN
CMOSInverter
VOUT
VDD
GND
vdd
Vin
TG
Vout
clkBar
clk
vdc=1
CLK


Negative Latch
D
Q
CLK

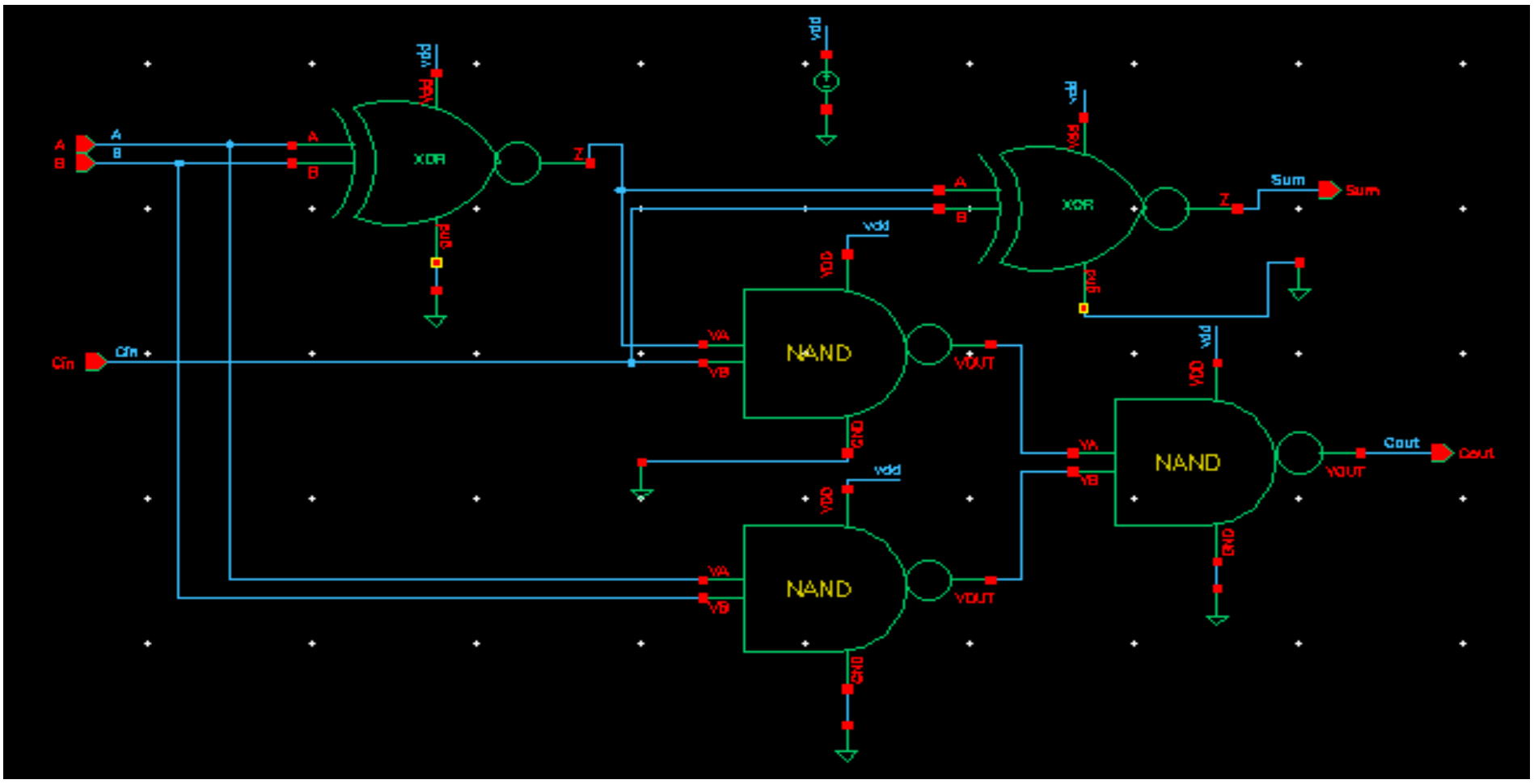

Transient
Sat Sep 28 09:10:02
1
Name
Vis
/Q
/CLK
/D
V (V)
1.1
-0.1
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
1.8
2.0
2.2
2.4
2.6
2.8
3.0
time (ns)


Negative Latch
Postive Latch
D
Q
CLK

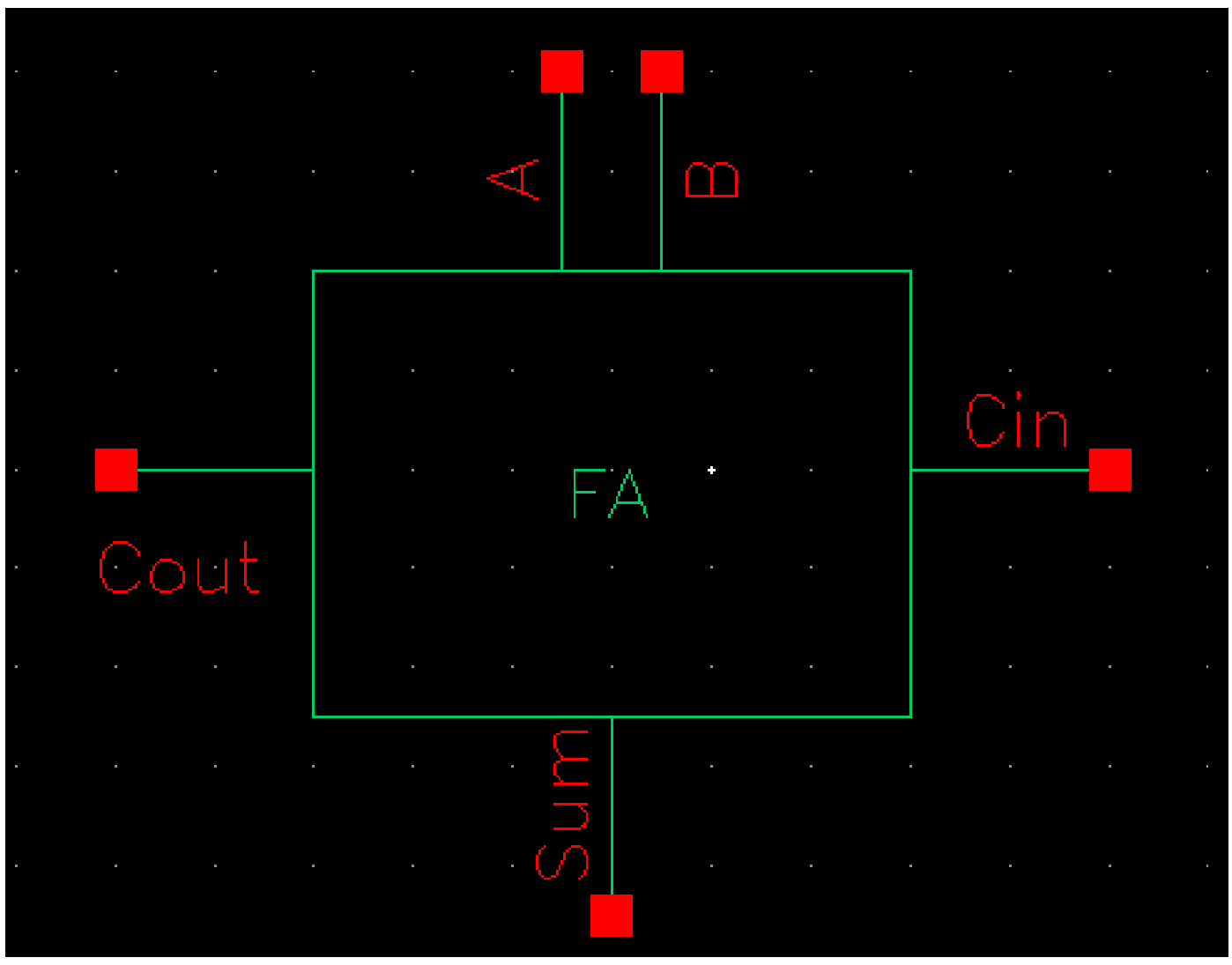

CLK
V1
v1=1
v2:0
tr=10p
gnd
D
V0
v1=1
v2:0
tr=10p
gnd
CLK
FF
Q
CLK
FF


Transient
Sat Sep 28 09:21:06
1
Name
Vis
/Q
/D
/CLK
V (V)
1.1
-0.1
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0 3.2 3.4 3.6 3.8 4.0
time (ns)

# 4.3 Full Adder Schematic

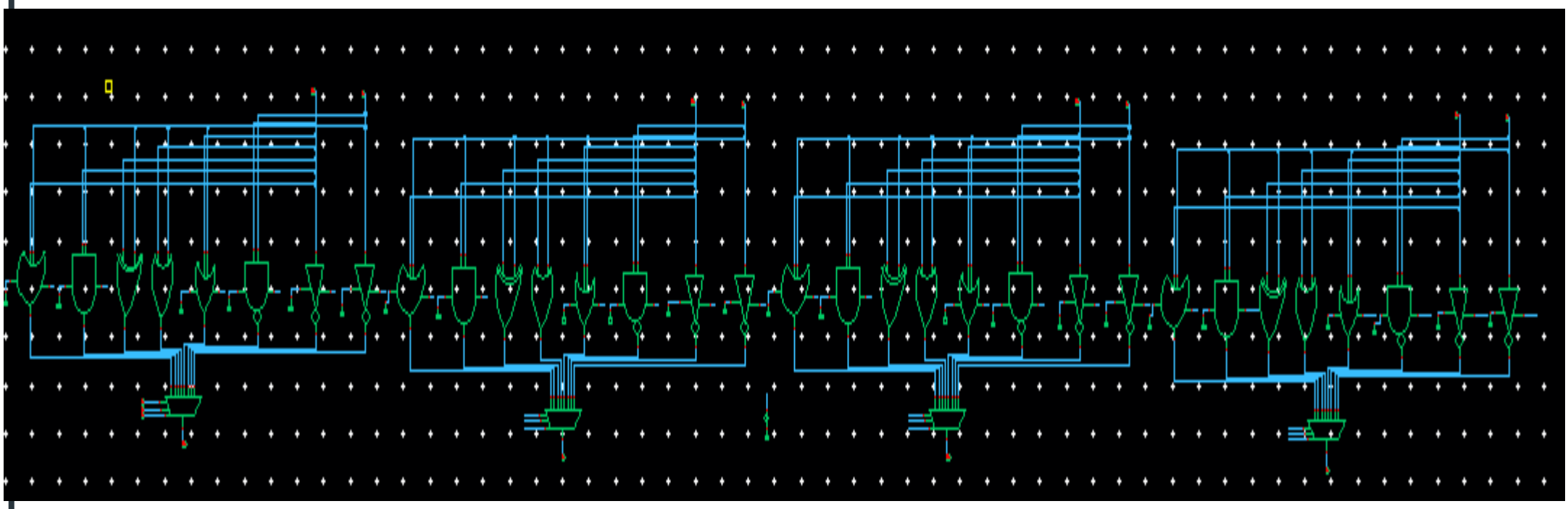



# 4.4 Full Adder Symbol

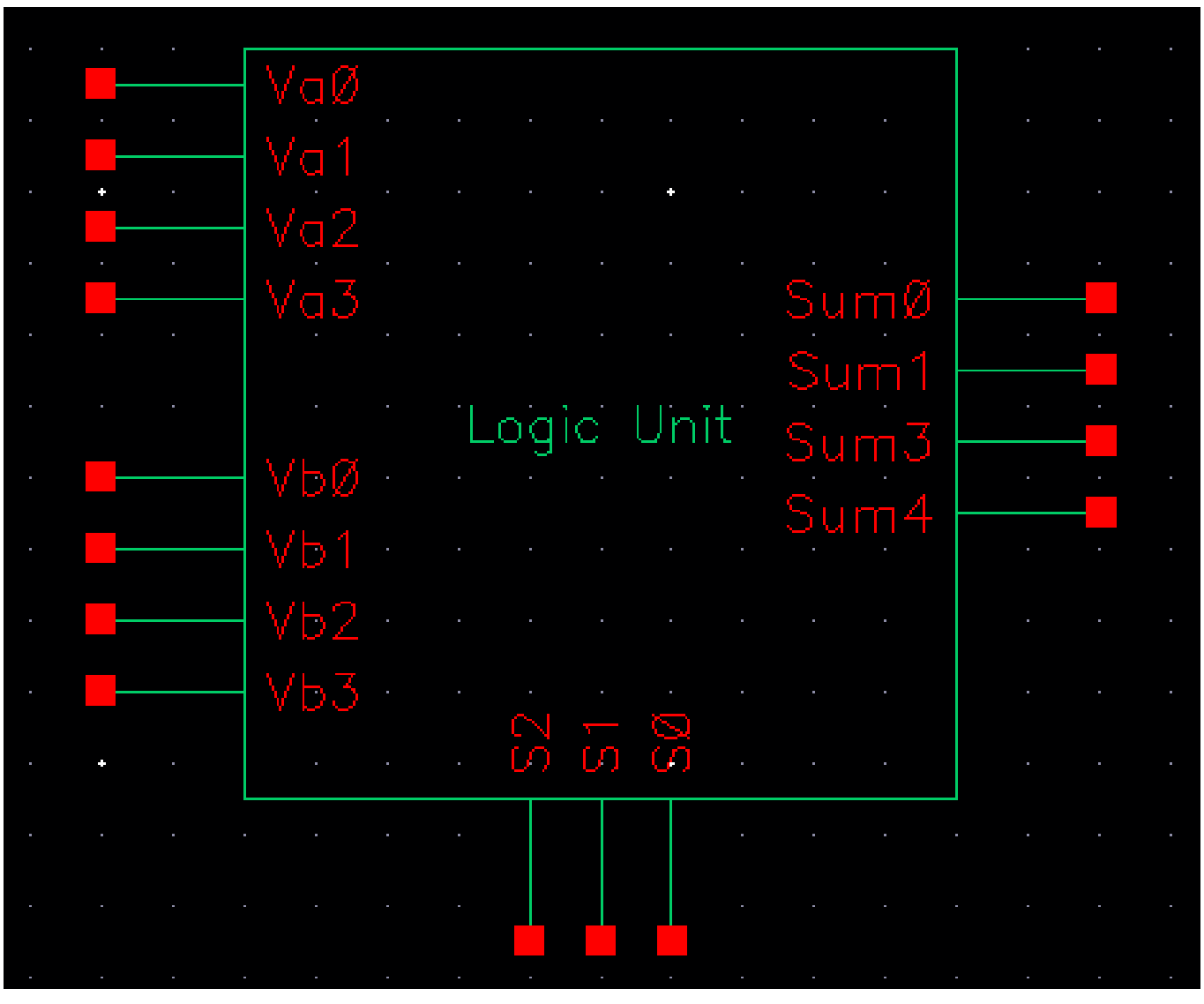


## 4.5 Logic Unit Schematic

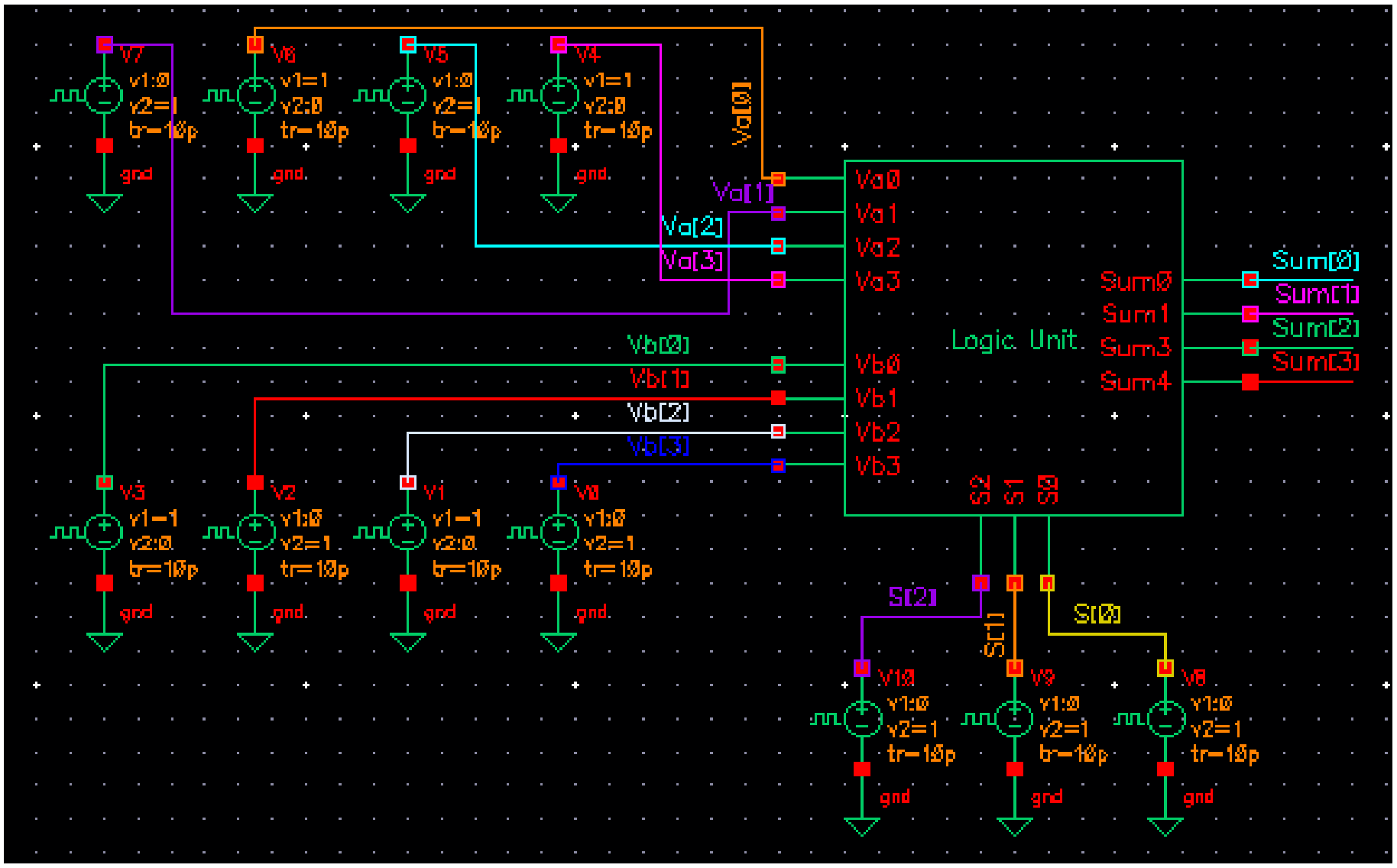
- With all the essential components and building blocks at our disposal, we can proceed to construct the complete Arithmetic Logic Unit (ALU), which comprises two primary blocks: the Logic Unit and the Arithmetic Unit. Our initial focus will be on building the Logic Unit.



## 4.4 Logic Unit Symbol



# 4.1 Logic Unit Testbench




bles Outputs Simulation Results To

**Analyses**

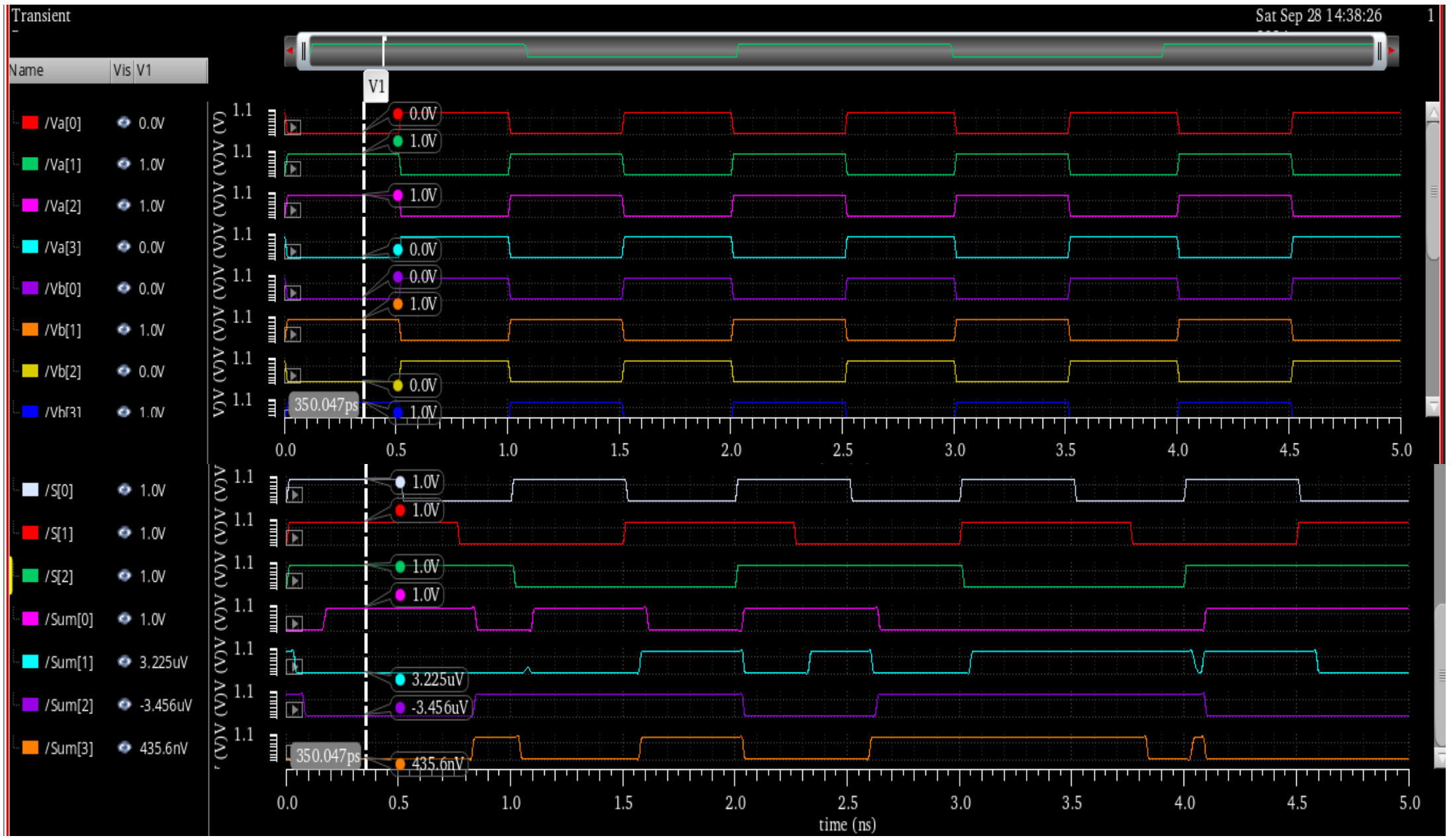
| | Type | Enable | |
|---|---|---|---|
| 1 | tran | ☑ | 0 5n moderate |

**Outputs**

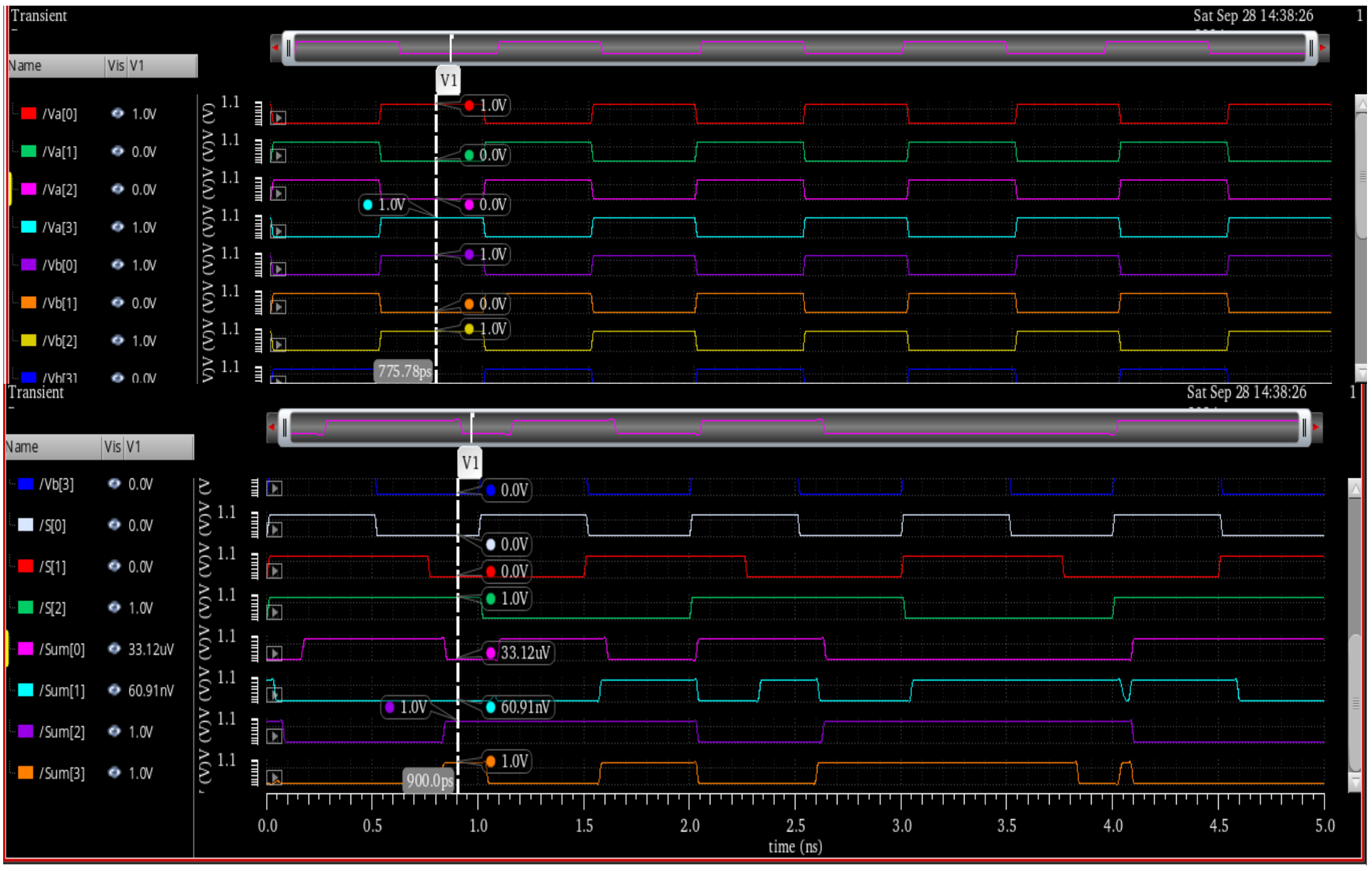
| | Name/Signal/Expr | Value | Plot | Save | Save Options |
|---|---|---|---|---|---|
| 1 | Va[0] | | ☑ | ☐ | allv |
| 2 | Va[1] | | ☑ | ☐ | allv |
| 3 | Va[2] | | ☑ | ☐ | allv |
| 4 | Va[3] | | ☑ | ☐ | allv |
| 5 | Vb[0] | | ☑ | ☐ | allv |
| 6 | Vb[1] | | ☑ | ☐ | allv |
| 7 | Vb[2] | | ☑ | ☐ | allv |
| 8 | Vb[3] | | ☑ | ☐ | allv |
| 9 | S[0] | | ☑ | ☐ | allv |
| 10 | S[1] | | ☑ | ☐ | allv |
| 11 | S[2] | | ☑ | ☐ | allv |
| 12 | Sum[0] | | ☑ | ☐ | allv |
| 13 | Sum[1] | | ☑ | ☐ | allv |
| 14 | Sum[2] | | ☑ | ☐ | allv |
| 15 | Sum[3] | | ☑ | ☐ | allv |

Plot after simulation: Auto  Plotting mode: Replace

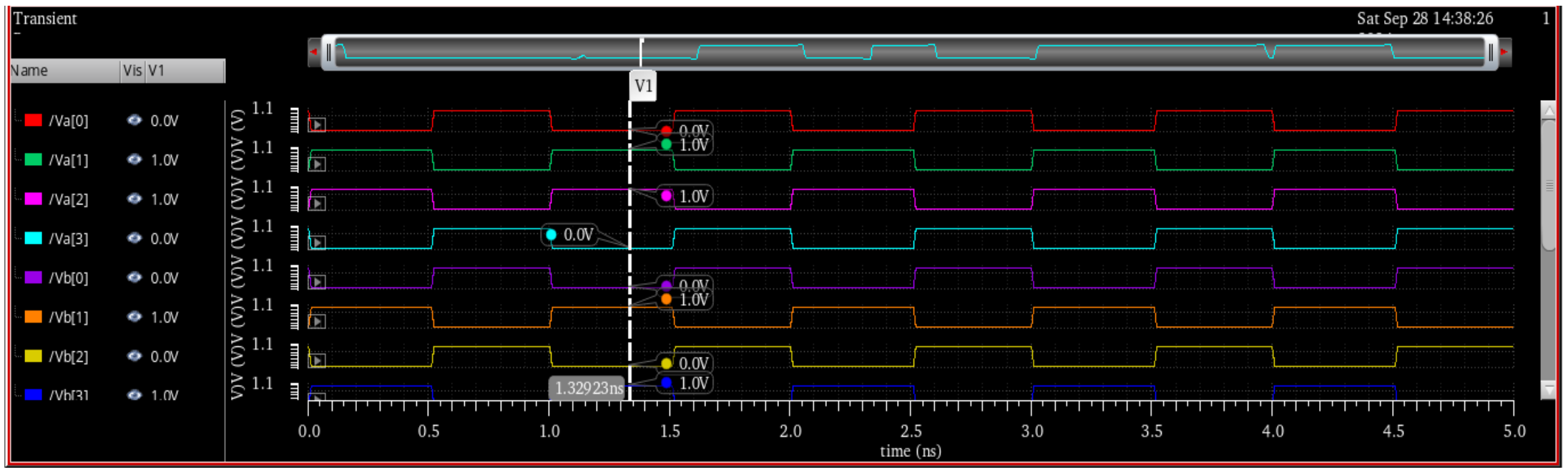
Testcase: $Va = \ 0110, Vb = \ 1010, S = 111, Result = 0001$

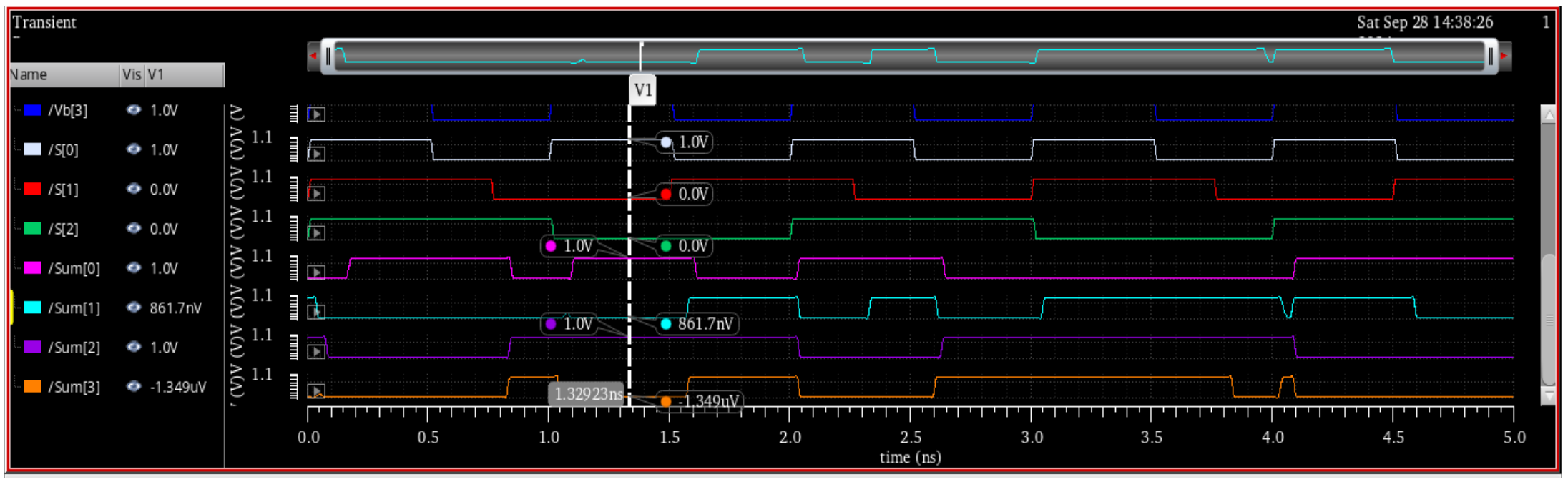



Testcase: $Va = \ 1001, Vb = \ 0101, S = 100, Result = 1100$

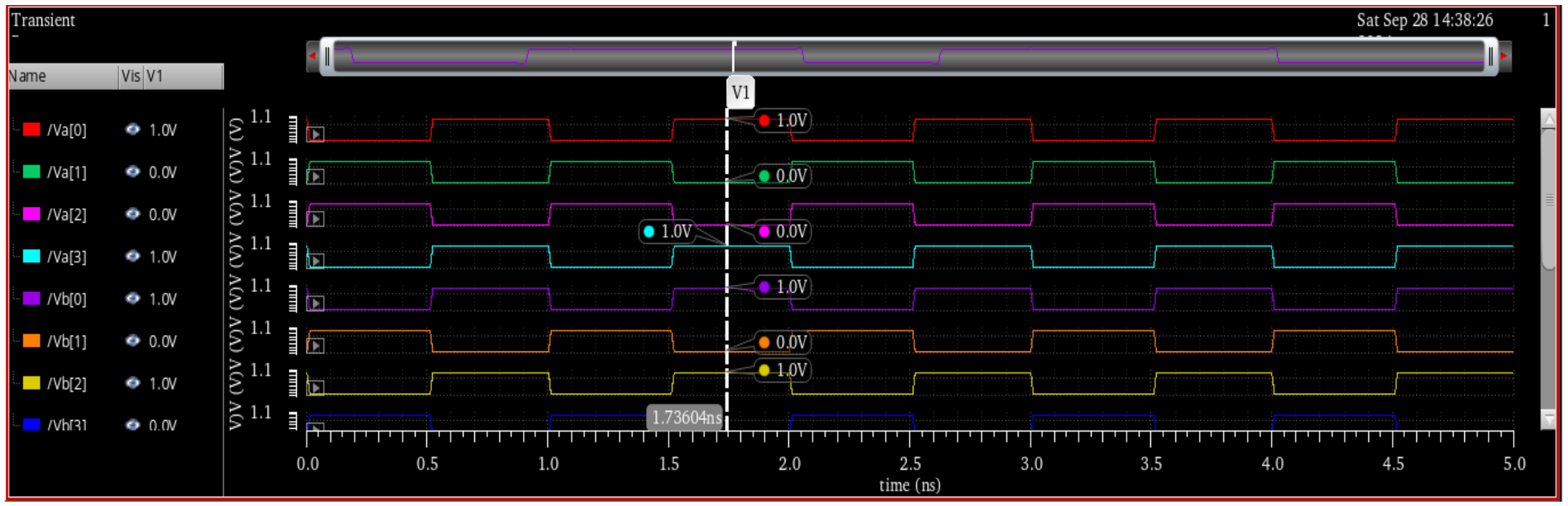


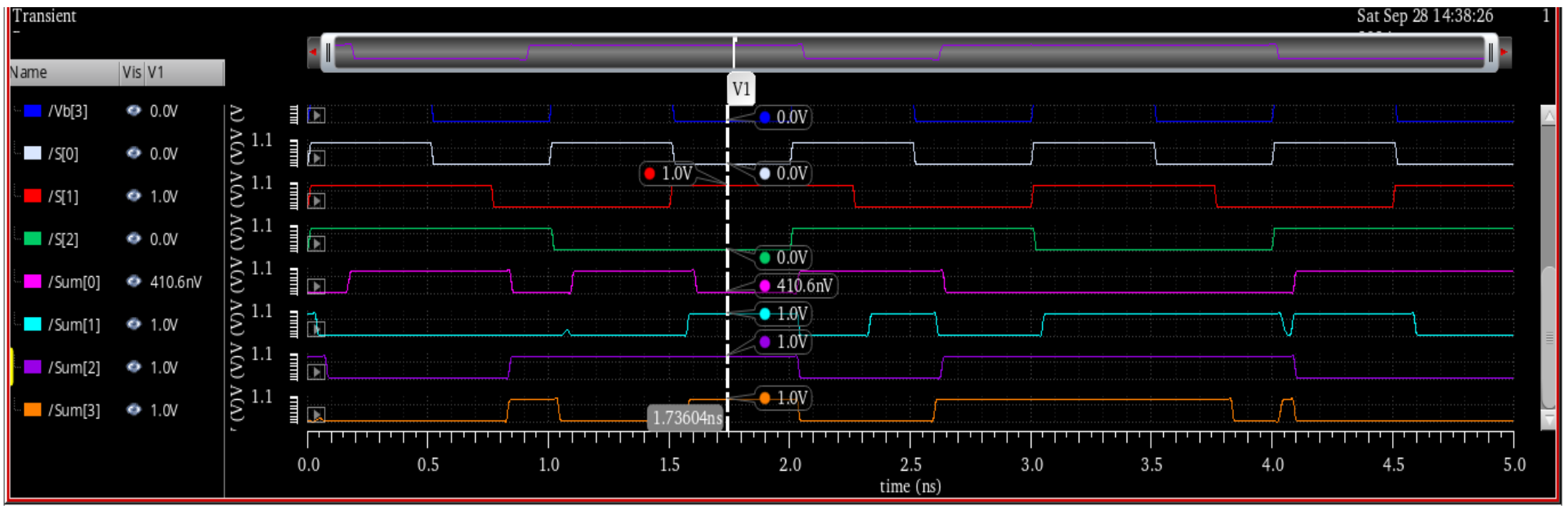
Testcase: $Va = \ 0110, Vb = \ 1010, S = 001, Result = 0101$



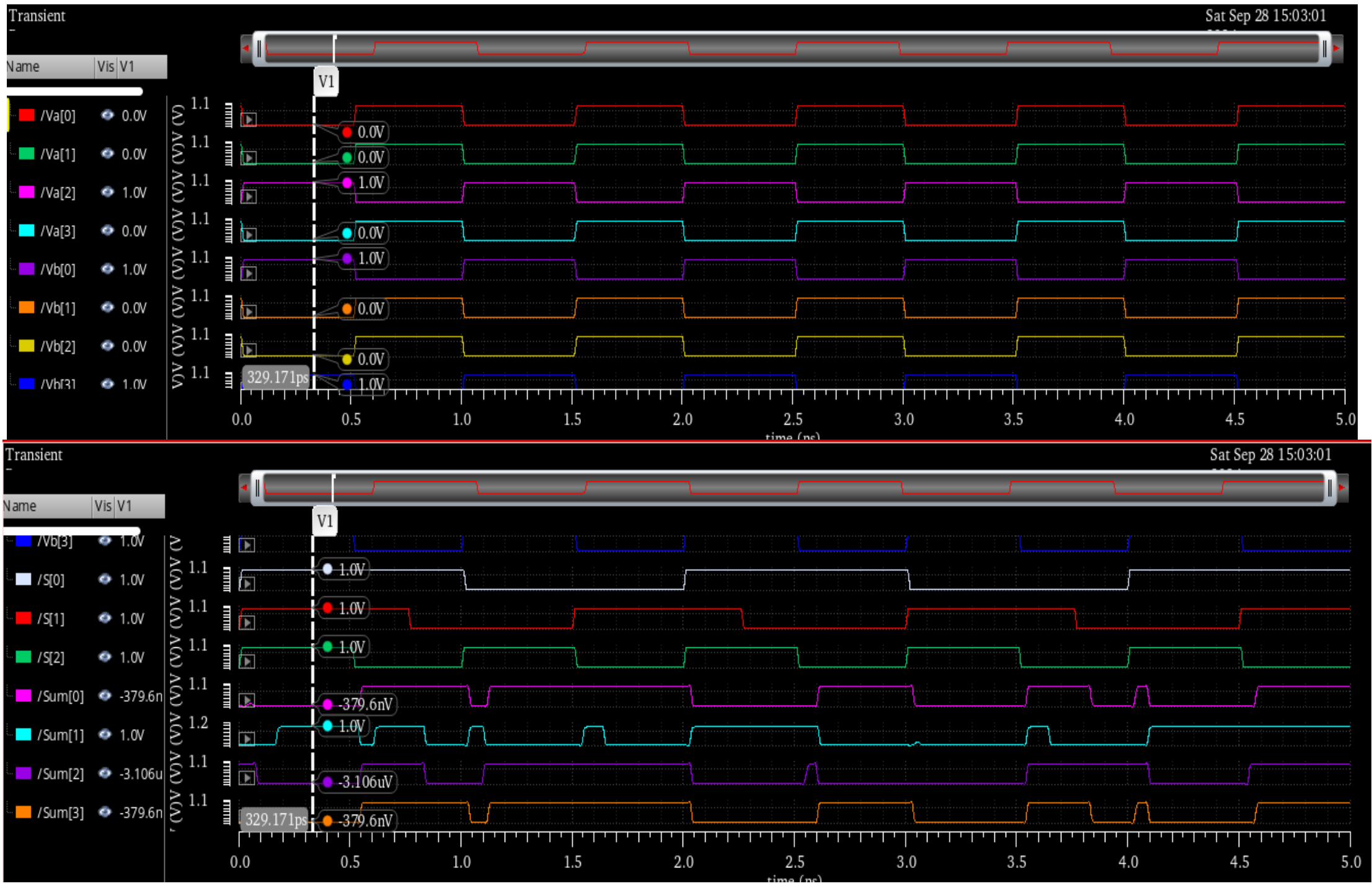




Testcase: $Va = \ 1001, Vb = \ 0101, S = 010, Result = 1110$

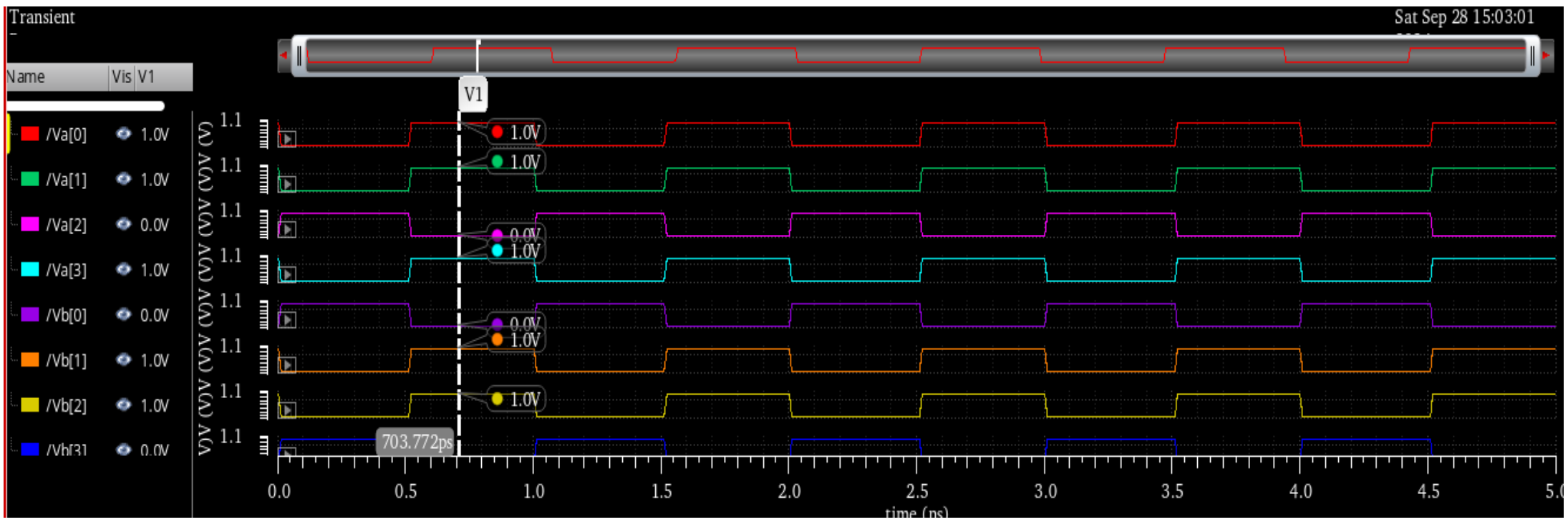


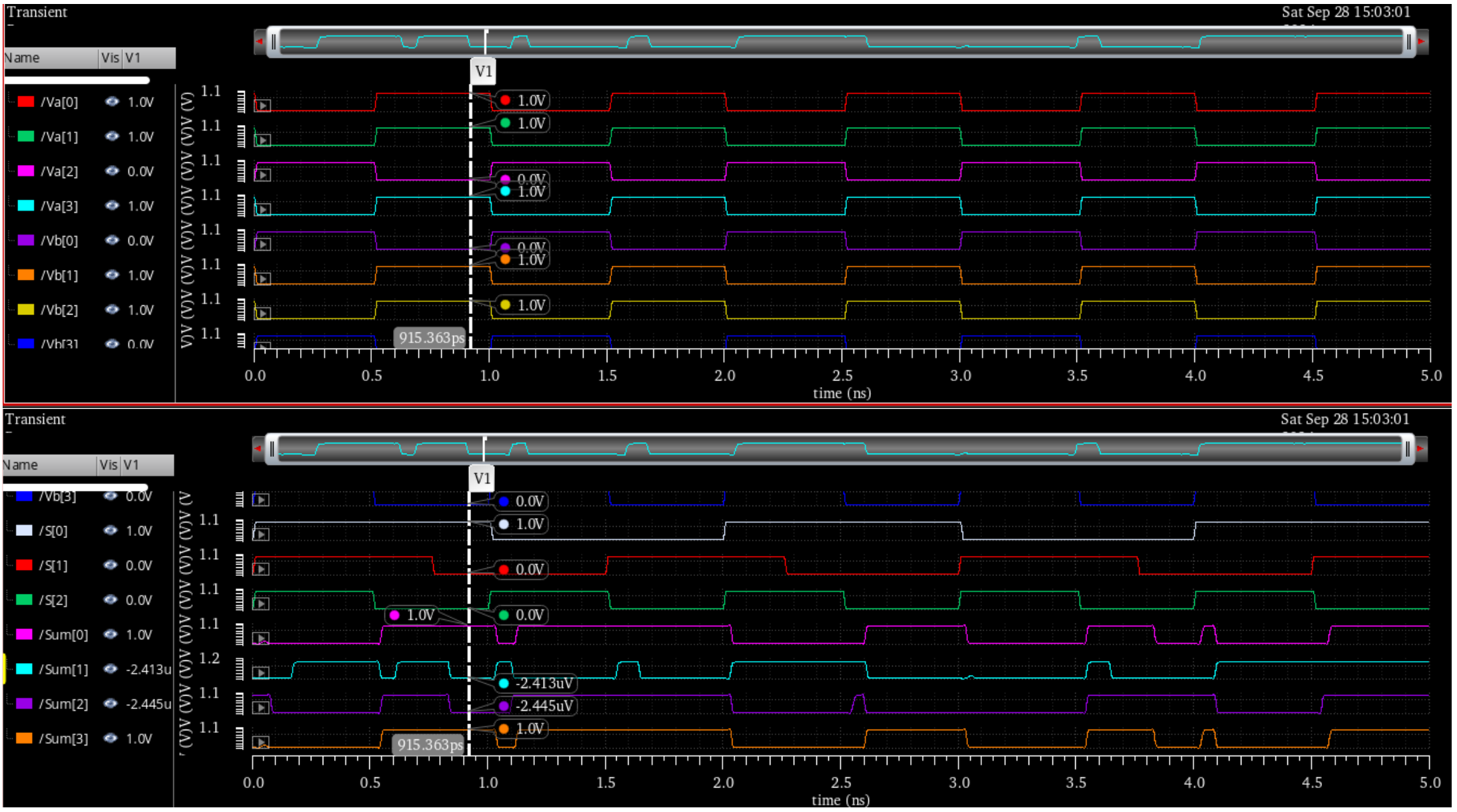
Testcase: $Va = \ 0100, Vb = \ 1001, S = 111, Result = 0010$



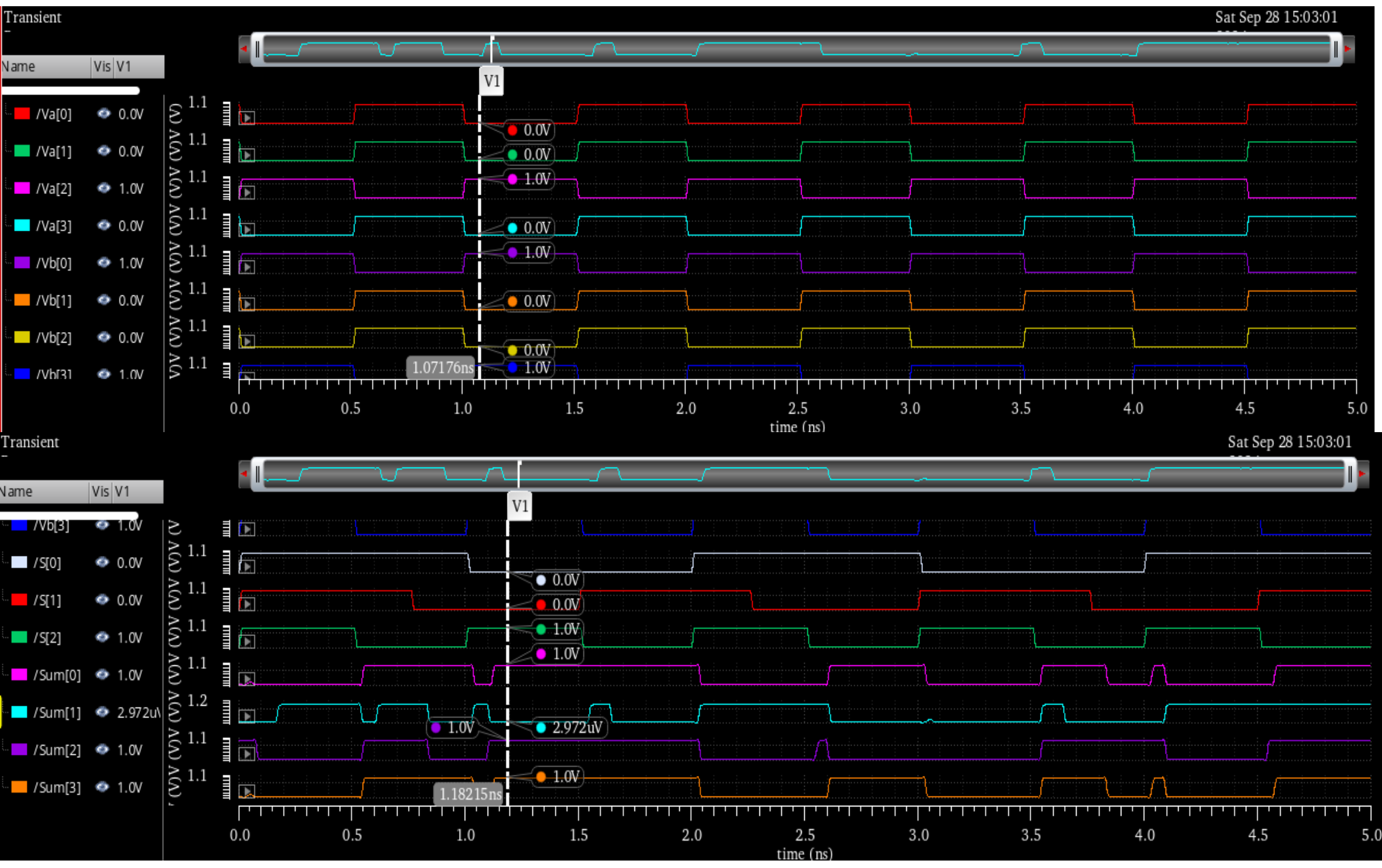

Testcase: $Va = \ 1011, Vb = \ 0110, S = 011, Result = 1111$



Testcase: $Va = \ 1011, Vb = \ 0110, S = 001, Result = 1001$

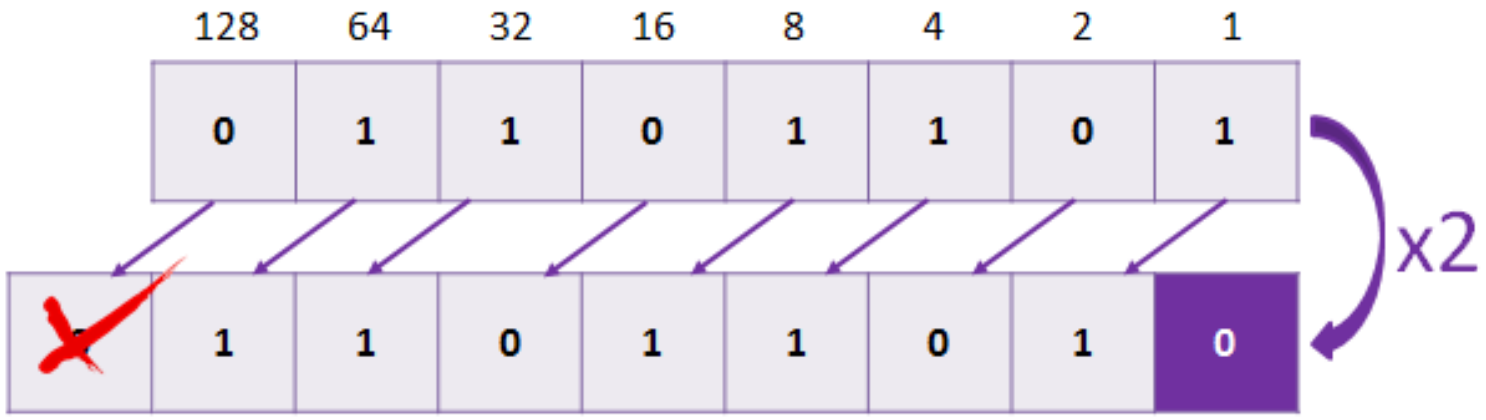


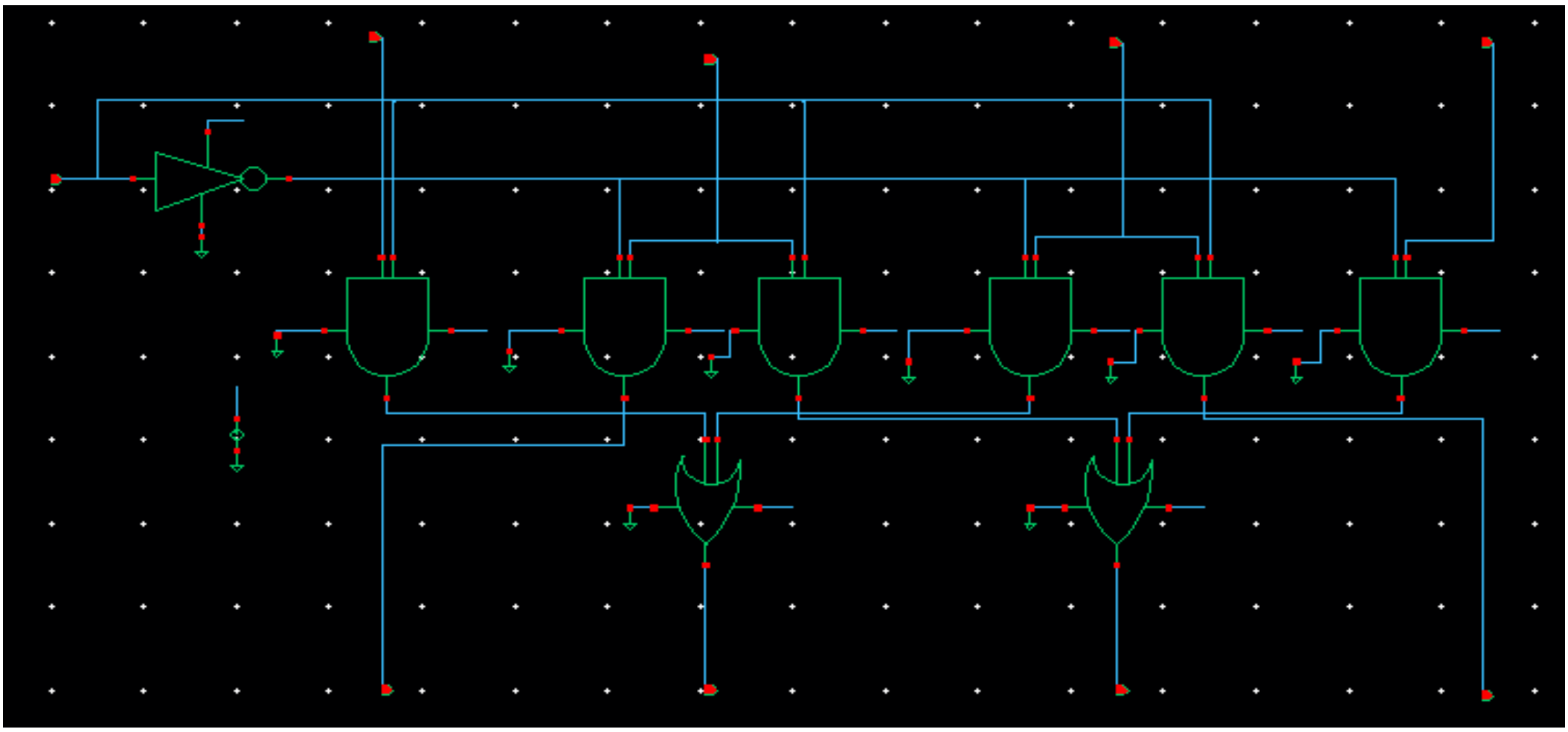

Testcase: $Va = \ 0100, Vb = \ 1001, S = 100, Result = 1101$



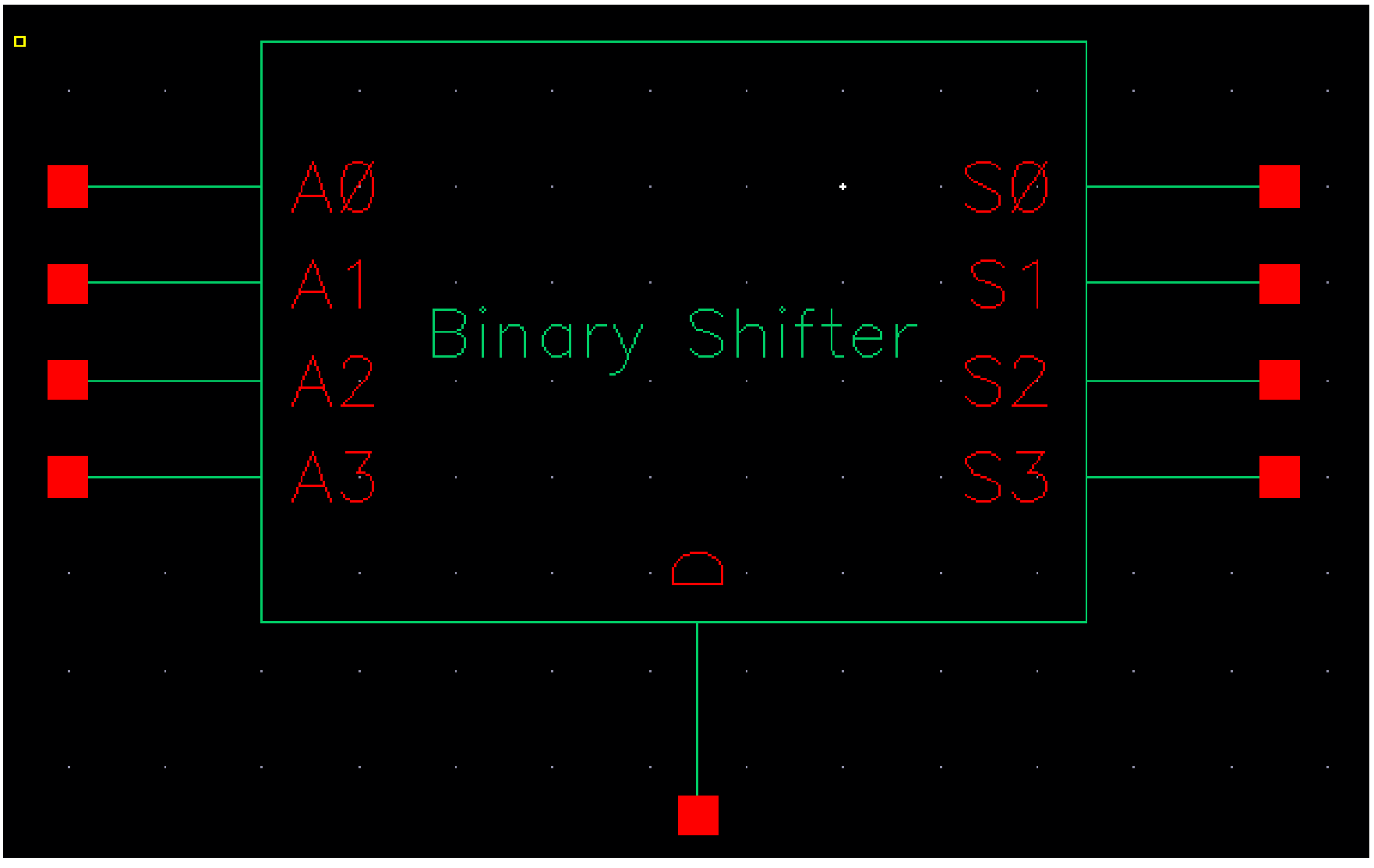
# 2-bit Binary Shifter Schematic

A **binary left shift** is used to multiply a binary number by two. It consists of shifting all the binary digits to the left by 1 digit and adding an extra digit at the end with a value of 0.

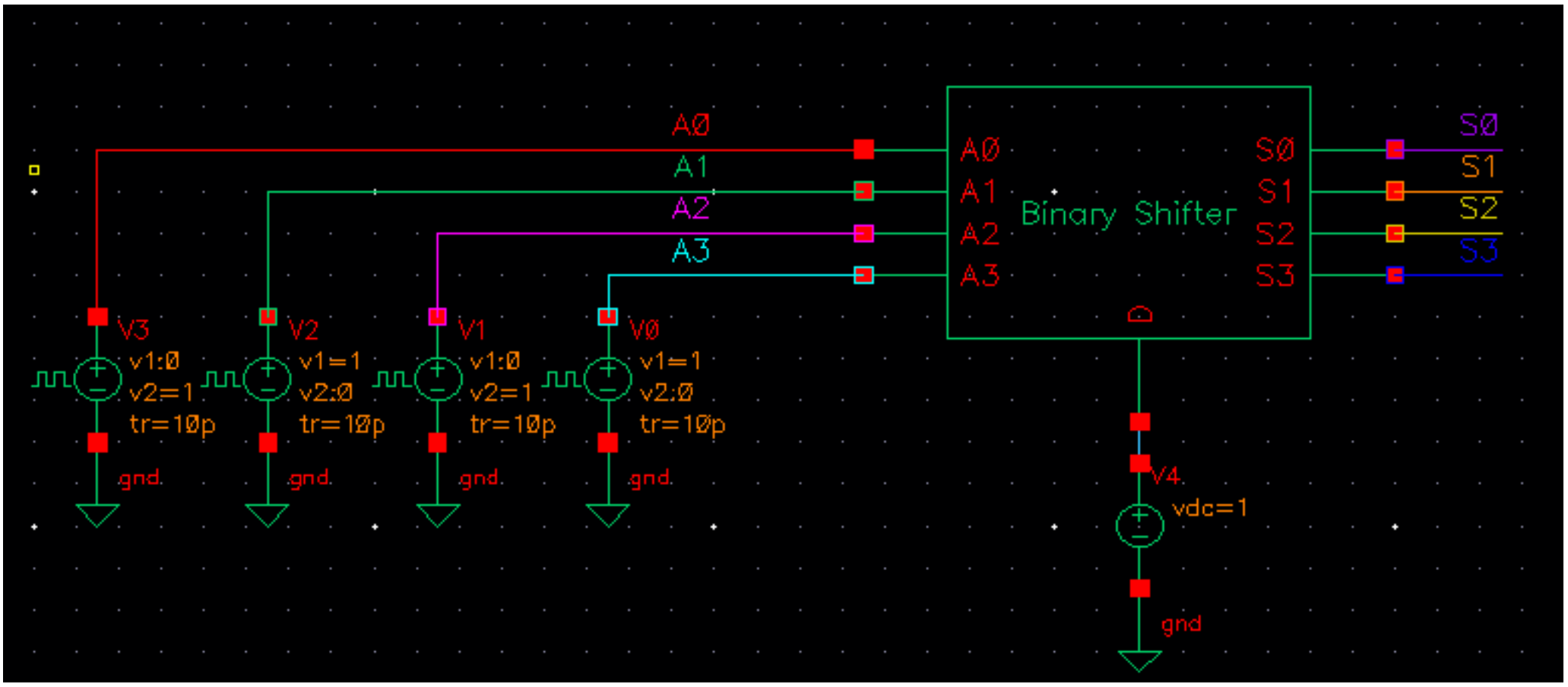


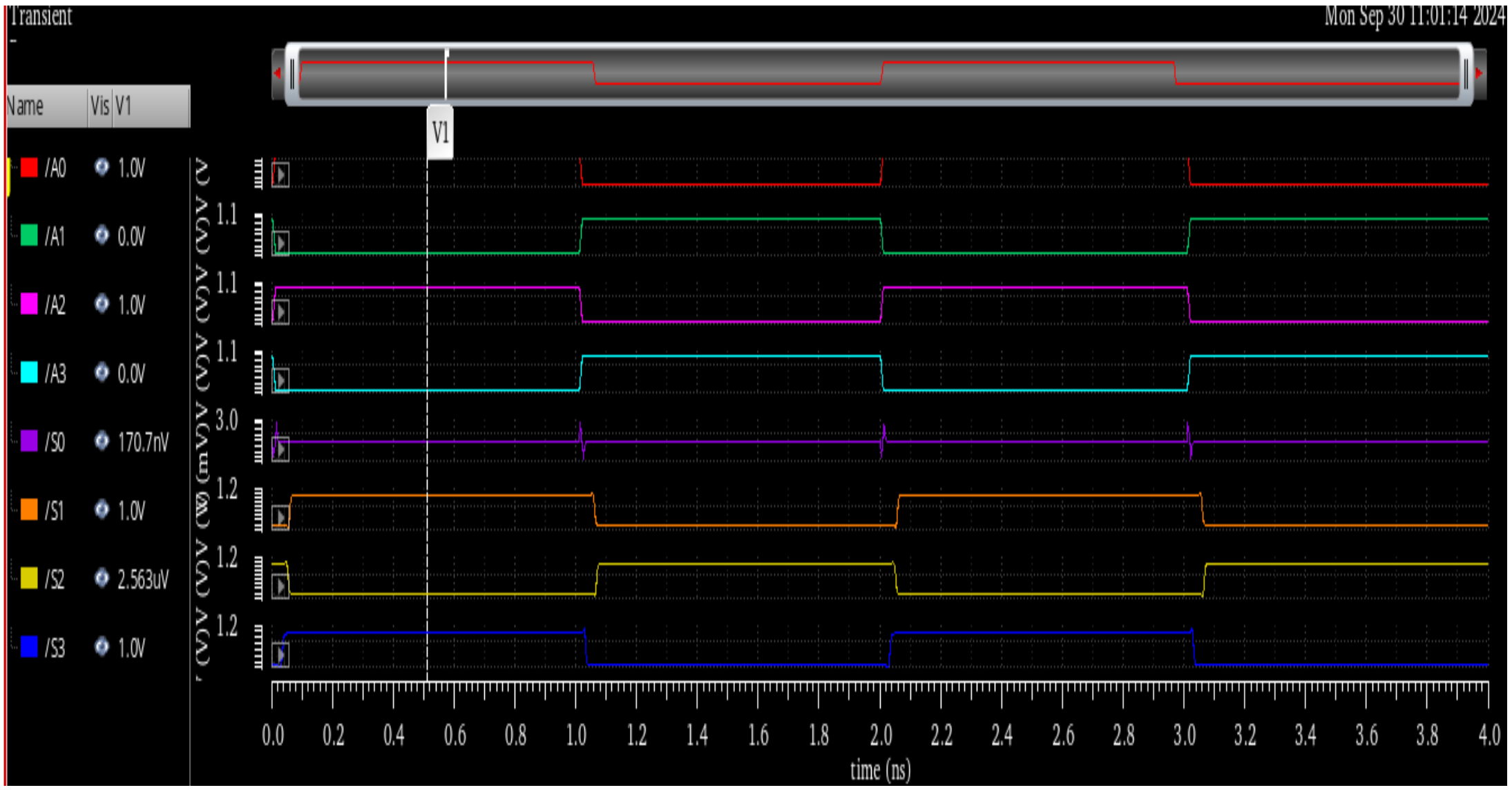



Input D is used to decide whether a **left shift (D=1)** or a **right shift (D=0)** is applied.

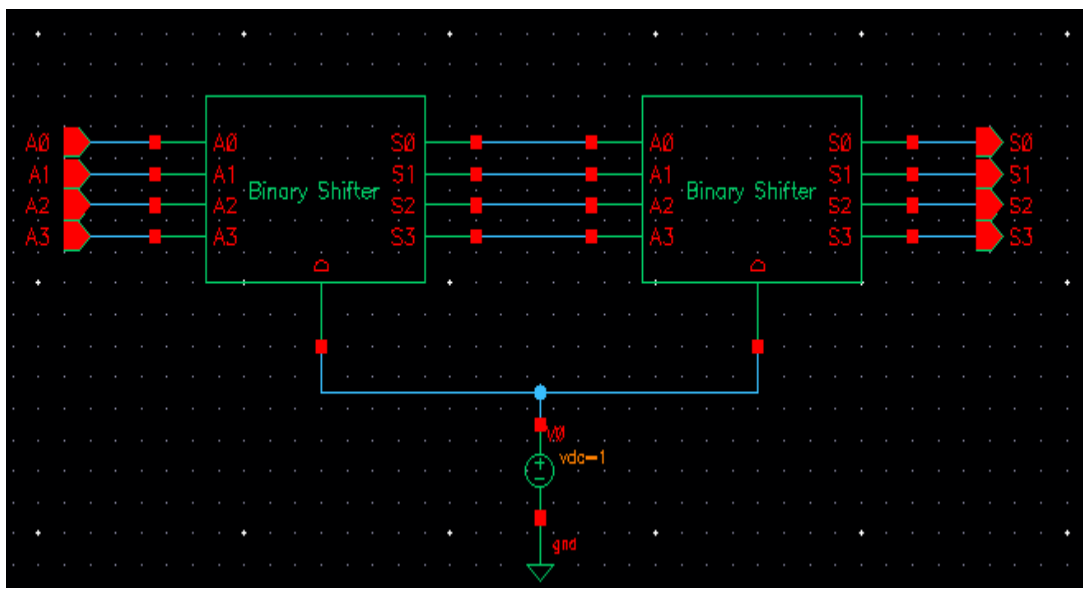


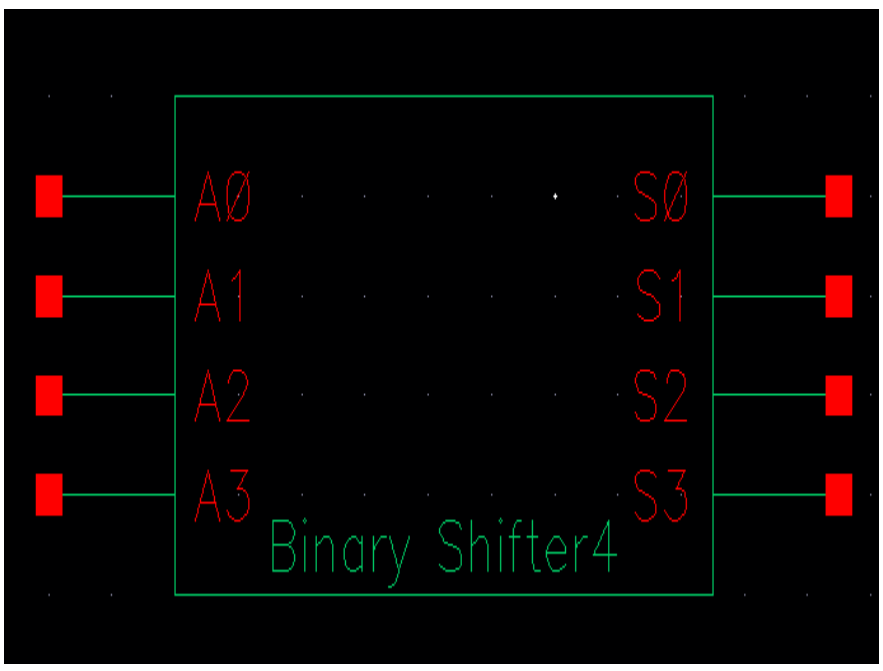
# 2-bit Binary Shifter Test



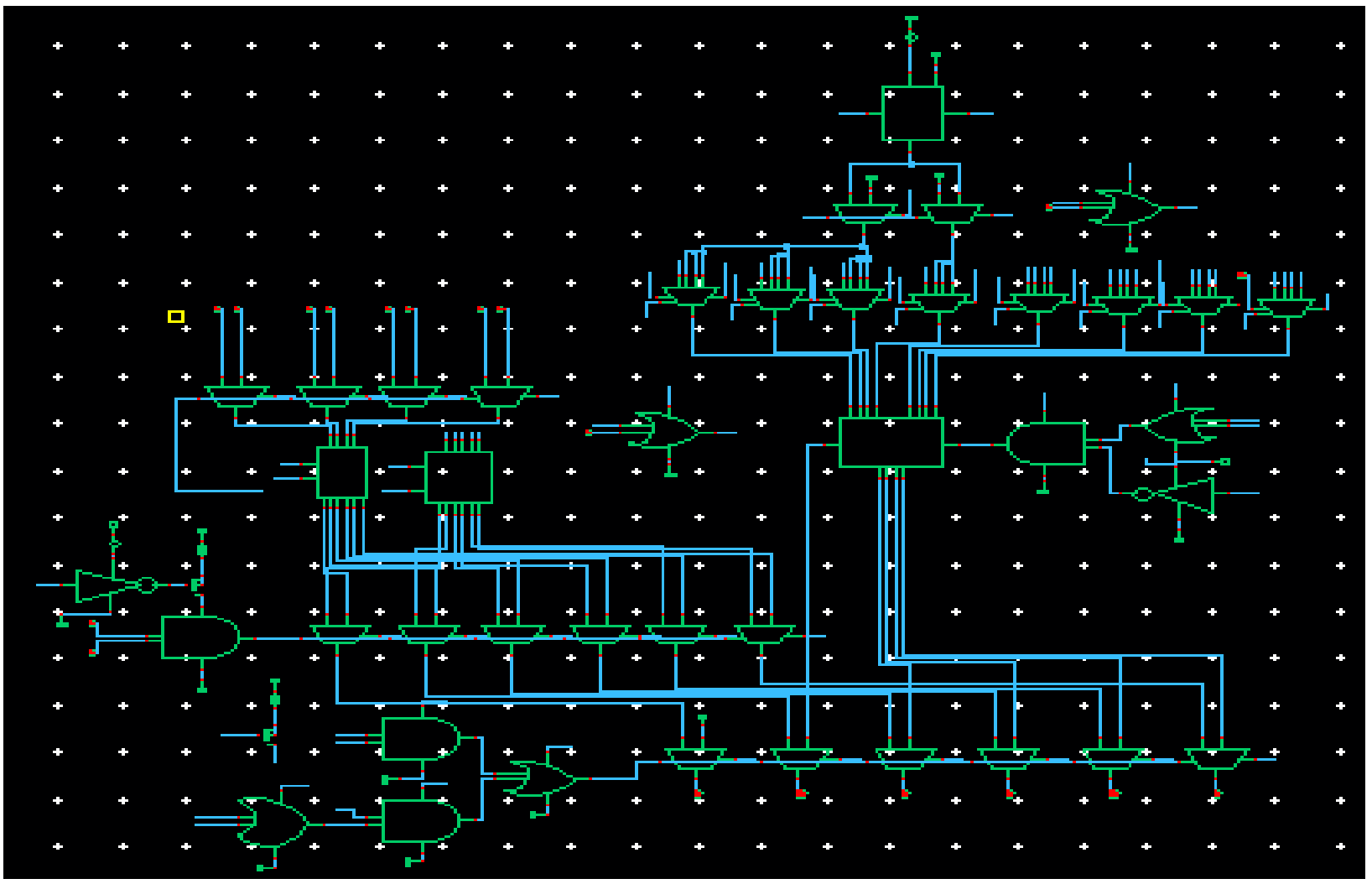

# 4x Binary Shifter



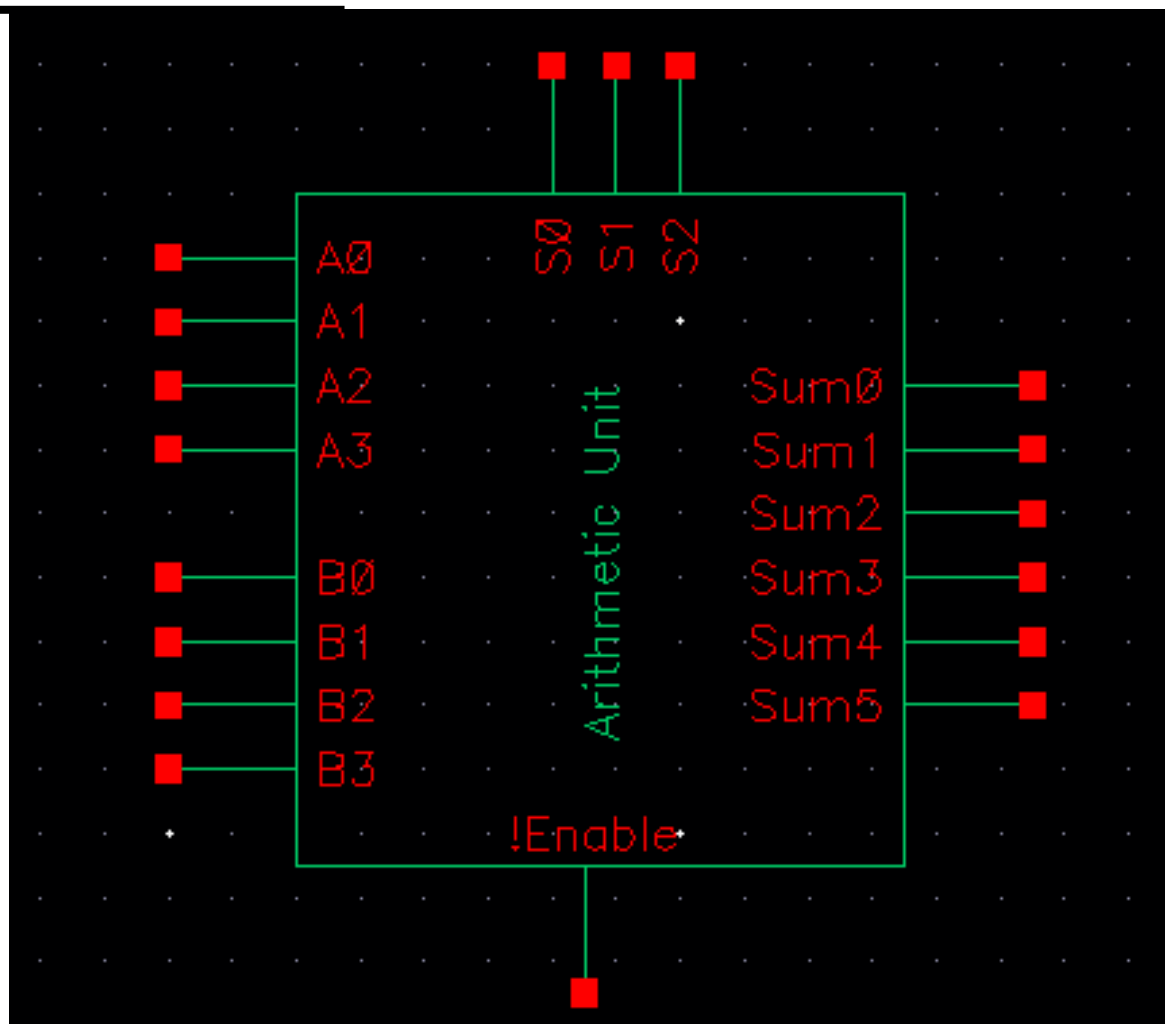



## 4.2 Arithmetic Unit Schematic

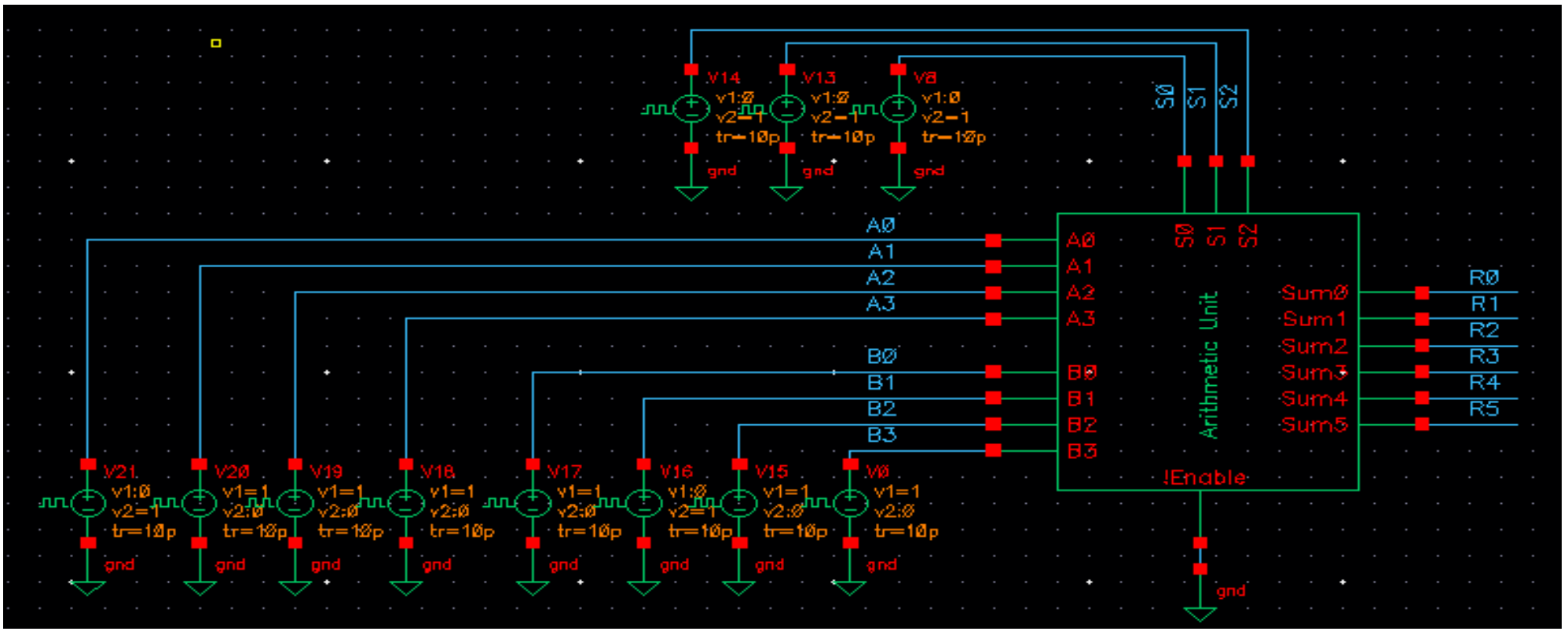
- To design the Arithmetic Unit, we incorporate full adders and place two 8x1 multiplexers (MUX) before each full adder. The purpose of these multiplexers is to select the appropriate operation based on our three selectors. It is important to note that the first full adder requires three MUXs, while the subsequent full adders utilize two MUXs each. This selection mechanism allows us to choose the desired operation within the Arithmetic Unit based on the given selectors.



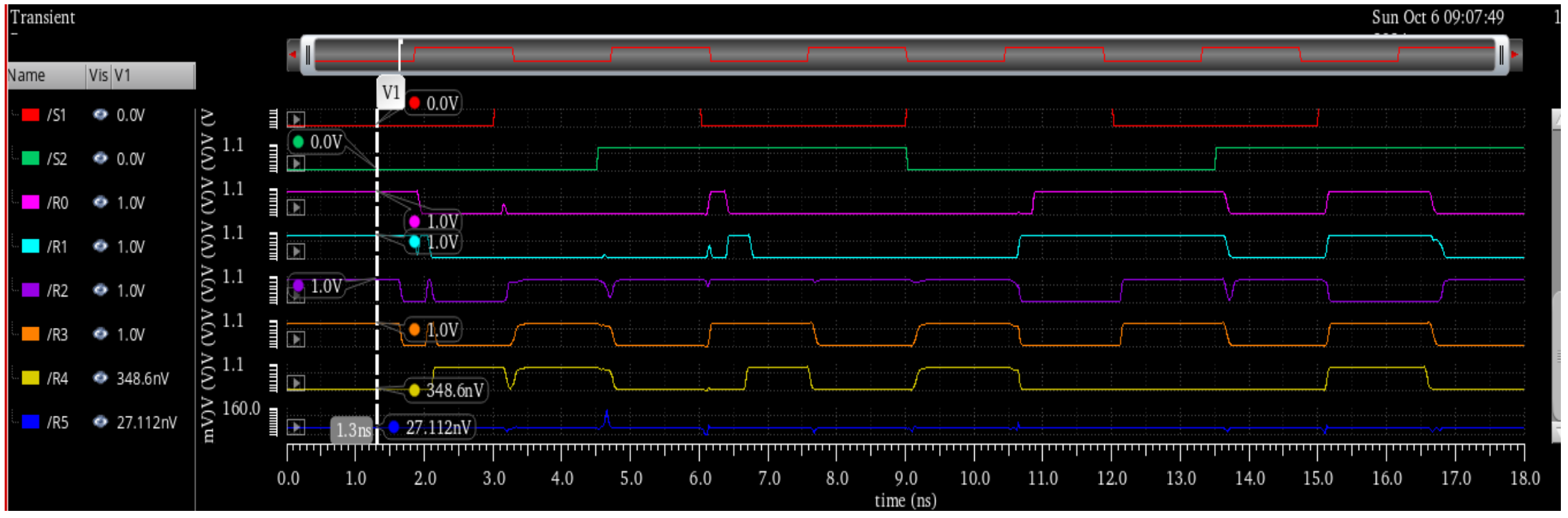
## Arithmetic Unit Symbol



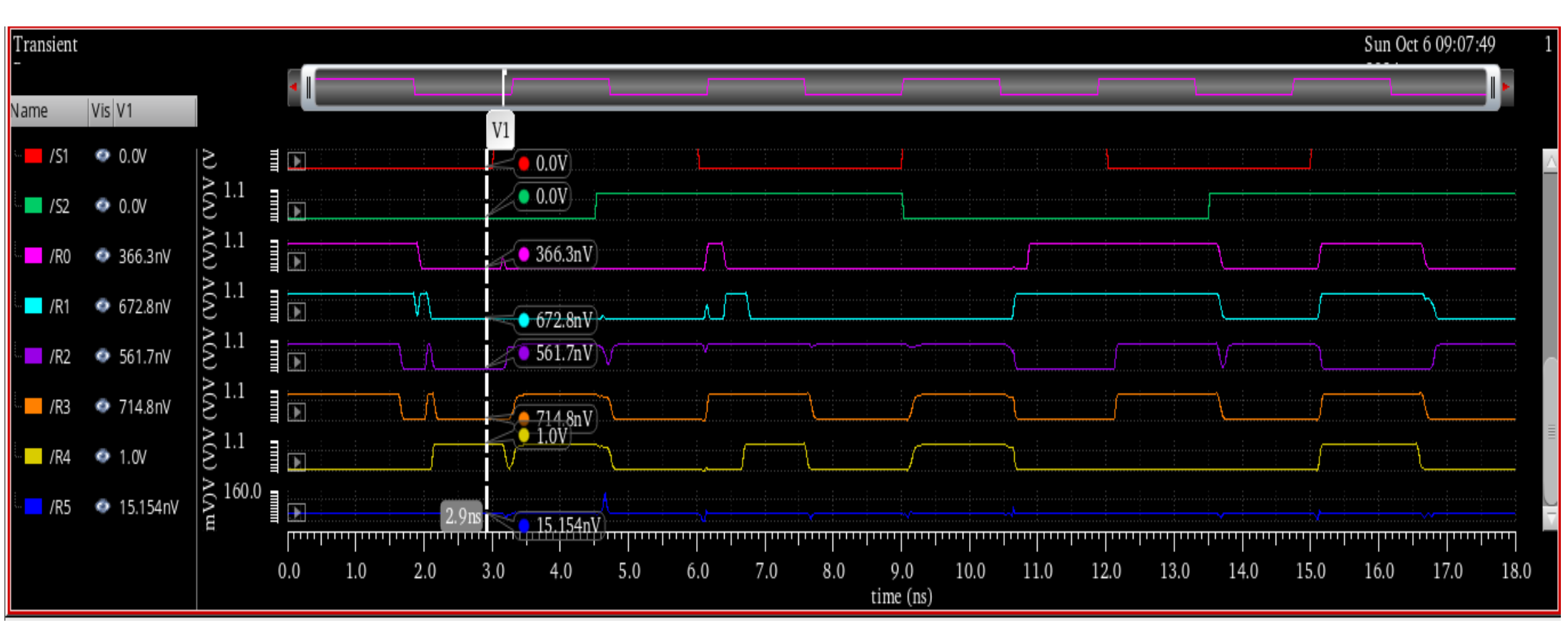
# Arithmetic Unit Test



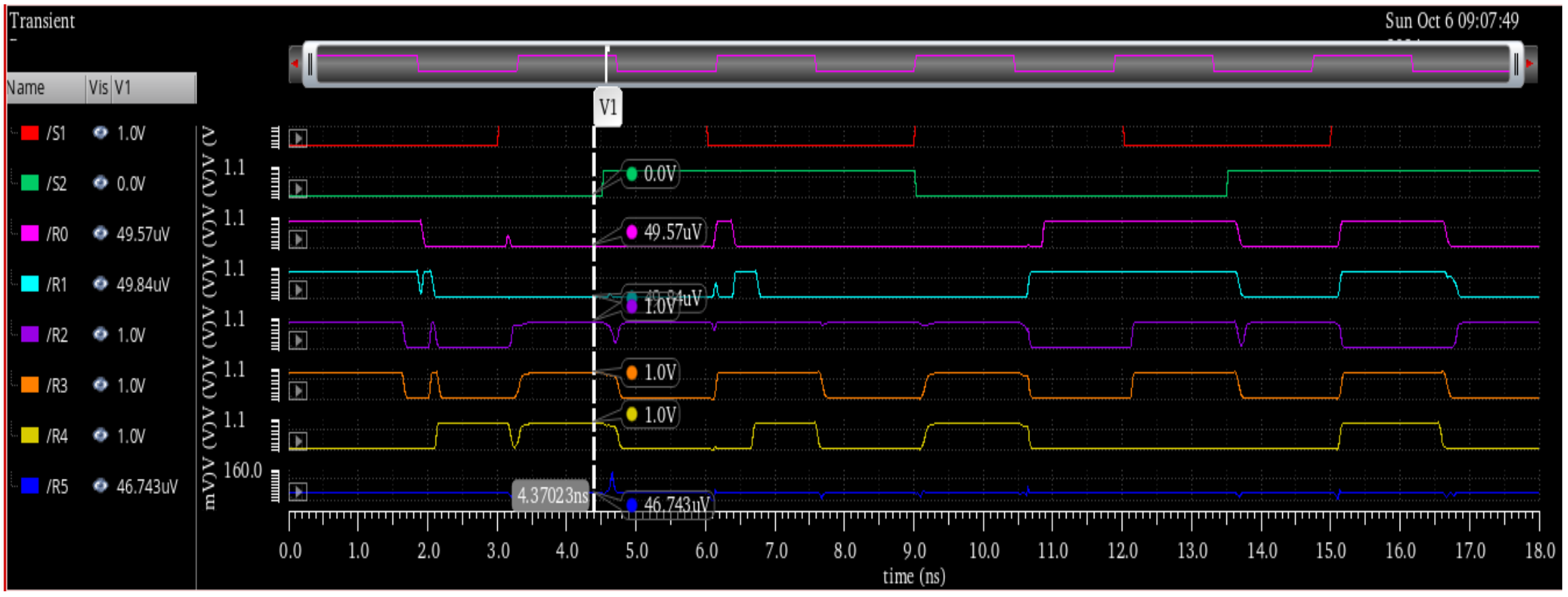

Testcase: $Va = \ 1110, Vb = \ 1101, S = 000, Result = 001111$



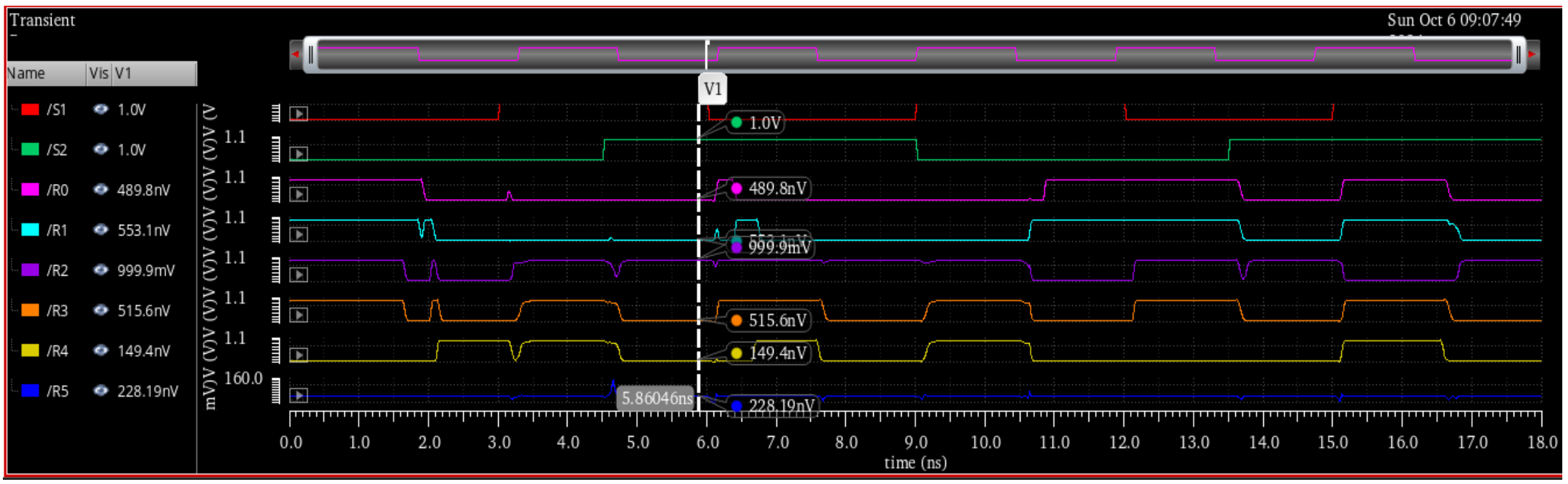

Testcase: $Va = \ 0001, Vb = \ 0010, S = 001, Result = 010000$



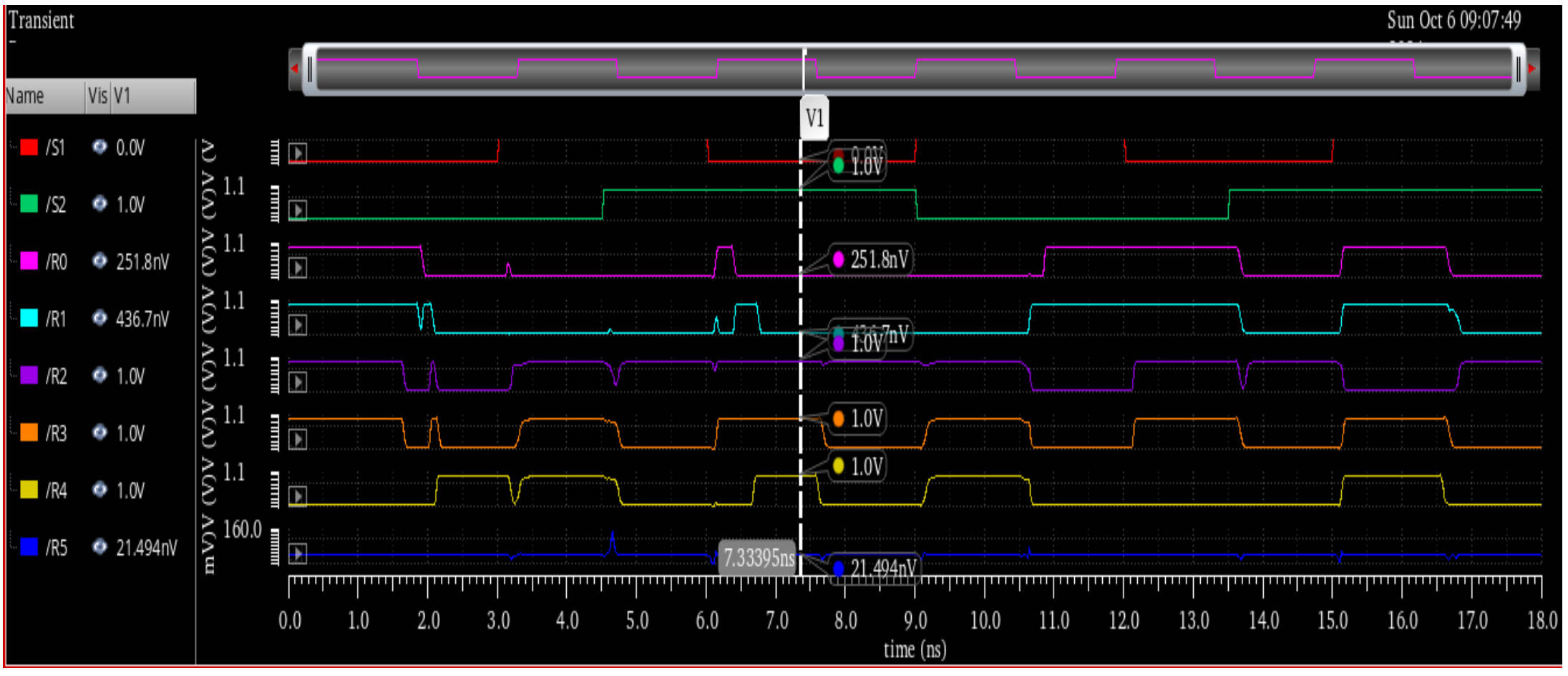
Testcase: $Va = \ 1110, Vb = \ 0010, S = 010, Result = 011100$



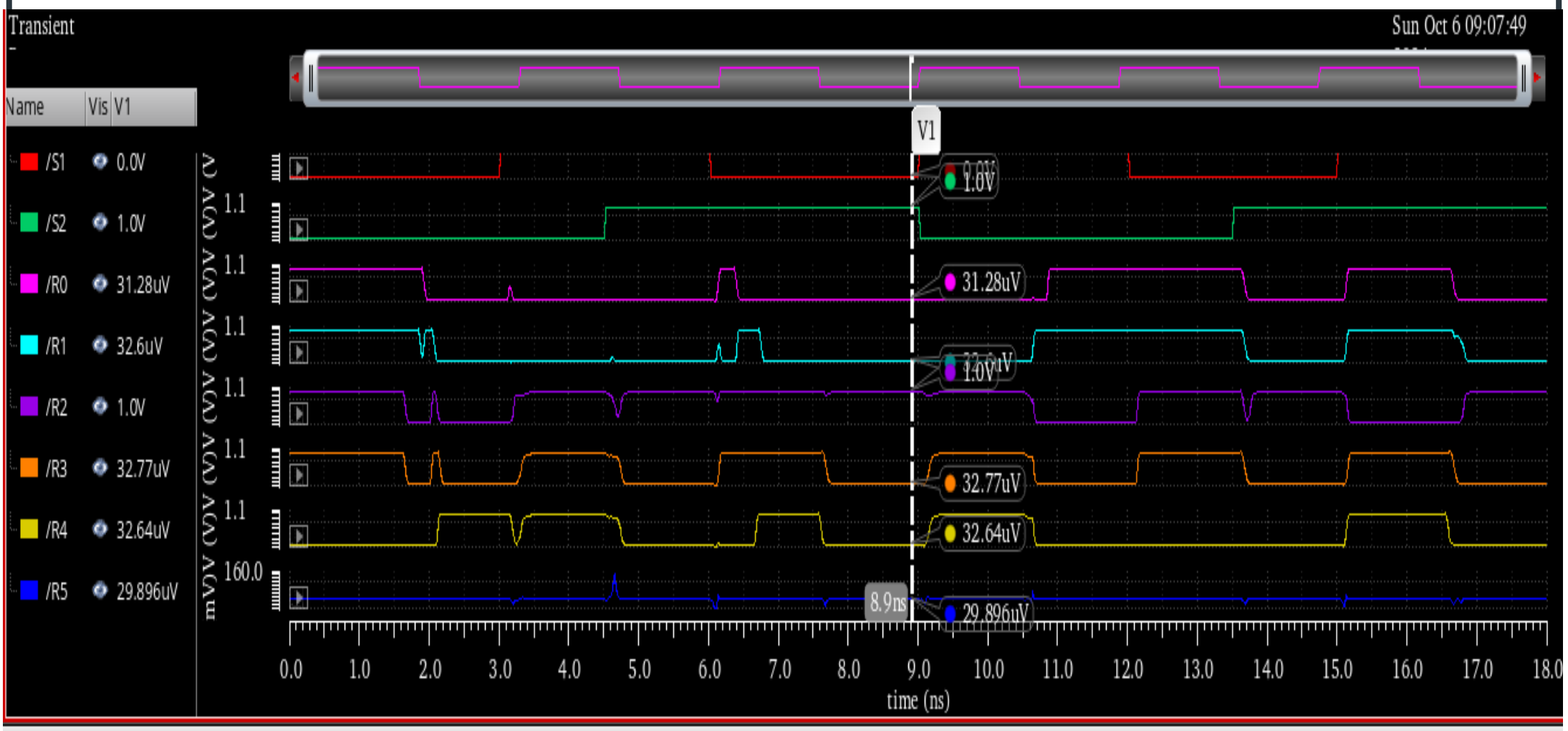

Testcase: $Va = \ 0001, Vb = \ 0010, S = 111, Result = 000100$




Testcase: $Va = \ 1110, Vb = \ 1101, S = 100, Result = 011100$



Testcase: $Va = 0001, Vb = 0010, S = 101, Result = 000100$




# 5. ALU Design

## 5.1 ALU Schematic

- To incorporate both logical and arithmetic operations within the ALU, an additional selector line, S3, is required. S3 serves as the most significant bit of the selectors and distinguishes between the two types of operations. If S3 = 1, the operation is a logical one, while if S3 = 0, it is an arithmetic operation.

- Furthermore, it is important to note that the output of the logic unit consists of 4 bits, whereas the output of the arithmetic unit is 6 bits. To ensure consistency in output length, the last two bits of the logic unit output is set to always be zero (grounded). This adjustment allows for uniformity in the bit length of the ALU's output.

# ALU Symbol