EE5311 - Digital IC Design

Project Final Report

Srivenkat A (EE18B038)
Hemanth Ram G K (EE18B132)
Sidesh S (EE18B032)

December 10, 2021

Contents

List of Tables

Problem Definition

Design a signed 8 bit carry save multiplier with a single stage pipeline. The idea is to show that the frequency of operation can be doubled and data can be fed to the multiplier at twice the rate through pipelining. This would require the following components:

- NAND2/ AND2 gate
- Full adder
  - Carry out generation circuit
  - Sum generation circuit
- Flip flop for pipelining

You have built your library with the above components and submitted each one as an assignment. Now you need to put them together and make the top level circuit.

The project report must contain

- A table summarizing the following information:
  - Maximum clock frequency w/o pipelining for the schematic and the layout extracted netlist
  - Maximum clock frequency with pipelining using the layout extracted netlist
  - Area of the DRC and LVS clean CSM
  - Number of test patterns you verified the functionality of your CSM against
- Block diagram of the 8 bit Carry Save Multipler (CSM) - Highlight how you used inverting and non-inverting adders to optimize delay
- Area of the CSM layout
- Maximum operating frequency of the CSM (without Pipeline)
- Location of the flip flop to double the frequency of operation
- SPICE simulation showing the frequency of operation with and without pipelining using RC extracted netlists for all sub-blocks (you should have the layout of all sub-blocks)
- Input combinations that you tested your CSM to see if it's functionally correct - Showing different kinds of multiplications
- SPICE simulations showing how the use of a faster adder in the Vector merge stage sped up over all delay of the CSM.

Summary

Table 2.1: Layout Parameters Summary

Layout Dimensions	
Height	999.75λ
Width	803λ
Aspect Ratio	1.245 : 1
Area	$802799.25\ \lambda^2 = 97.138\mu m^2$
DRC & LVS Clean	Yes

Table 2.2: CSM Analysis using Schematic

Metric from Schematic	**Unpipelined**	**Pipelined**
Comb. Delay	403.75ps	-
Minimum Tclk	560ps	310ps
Max. Frequency	1.78GHz	3.22GHz
Pipeline Gain	1.81×	

Table 2.3: CSM Analysis using RC extracted Netlist

Metric from RC extracted Netlist	**Unpipelined** (Full RC)	**Pipelined** (Stage level RC)
Comb. Delay	623.96ps	-
Minimum Tclk	910ps	460ps
Max. Frequency	1.1GHz	2.17GHz
Pipeline Gain	1.98×	

Table 2.4: 8-bit CSM Parameters Summary

Degradation due to Layout Parasitics (in unpipelined ckt)	1.618×
Degradation due to Layout Parasitics (in pipelined ckt)	1.484×
Alternate Vector Merge Stage	Carry Look-Ahead Adder & Carry Select Adder
# test patterns checked	30-40 (scripted using Python)

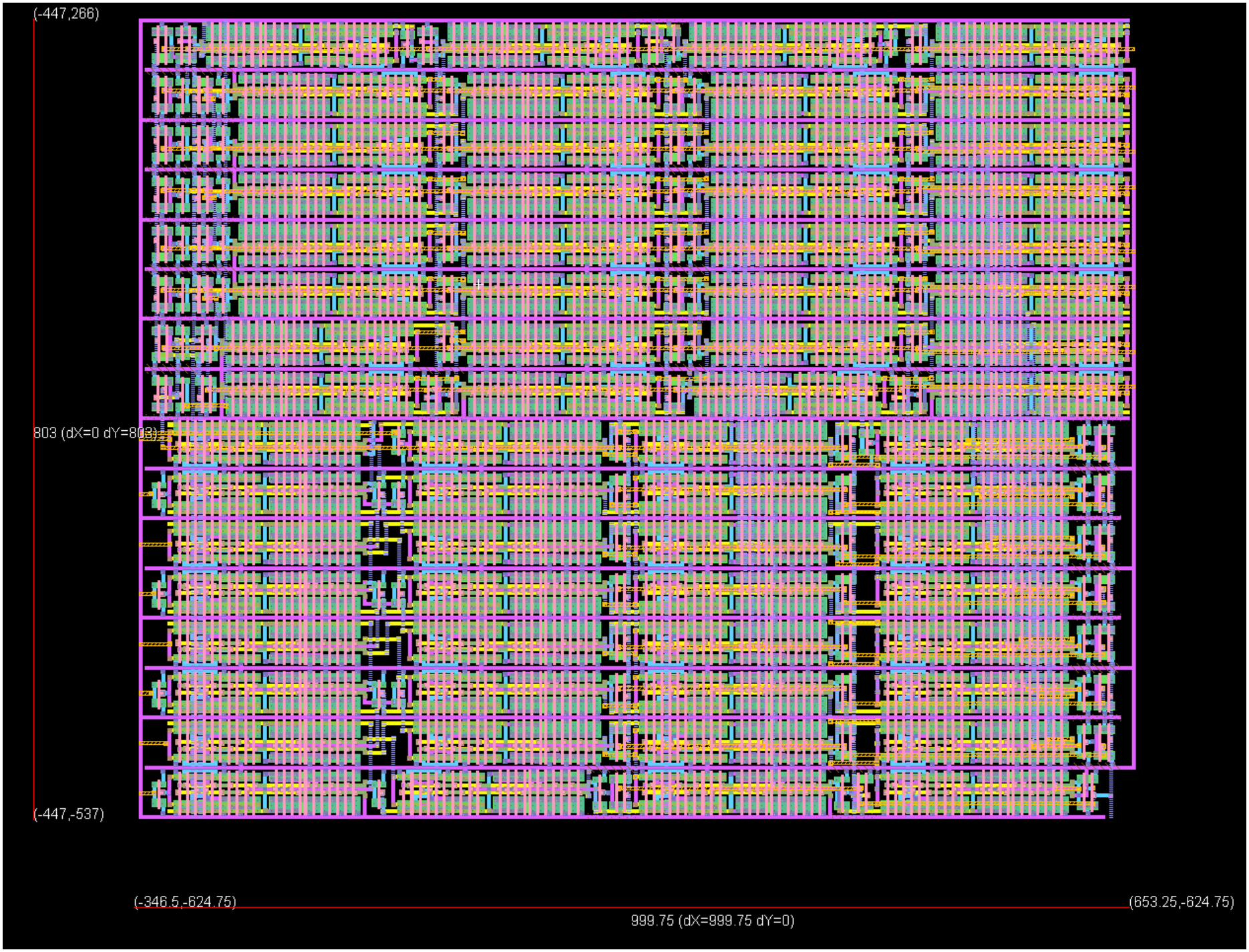


Figure 2.1: CSM Layout Image

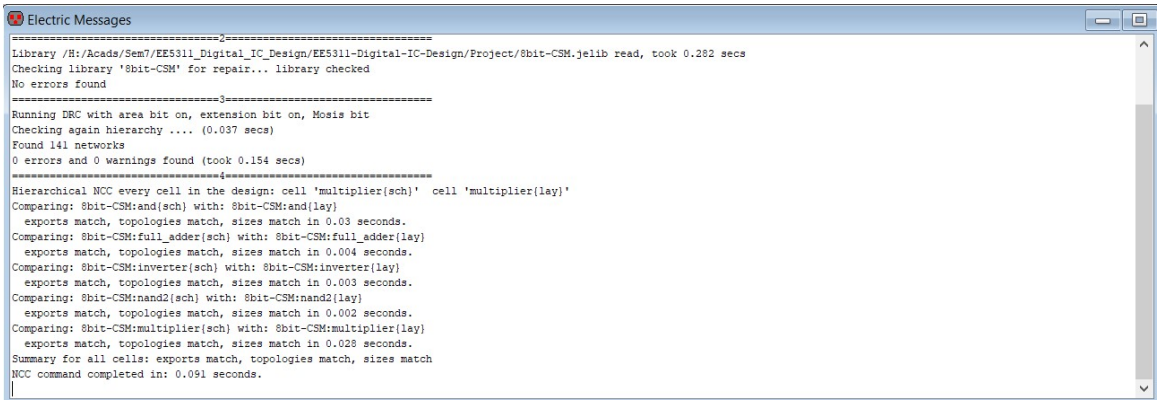


Figure 2.2: DRC LVS Clean Snapshot

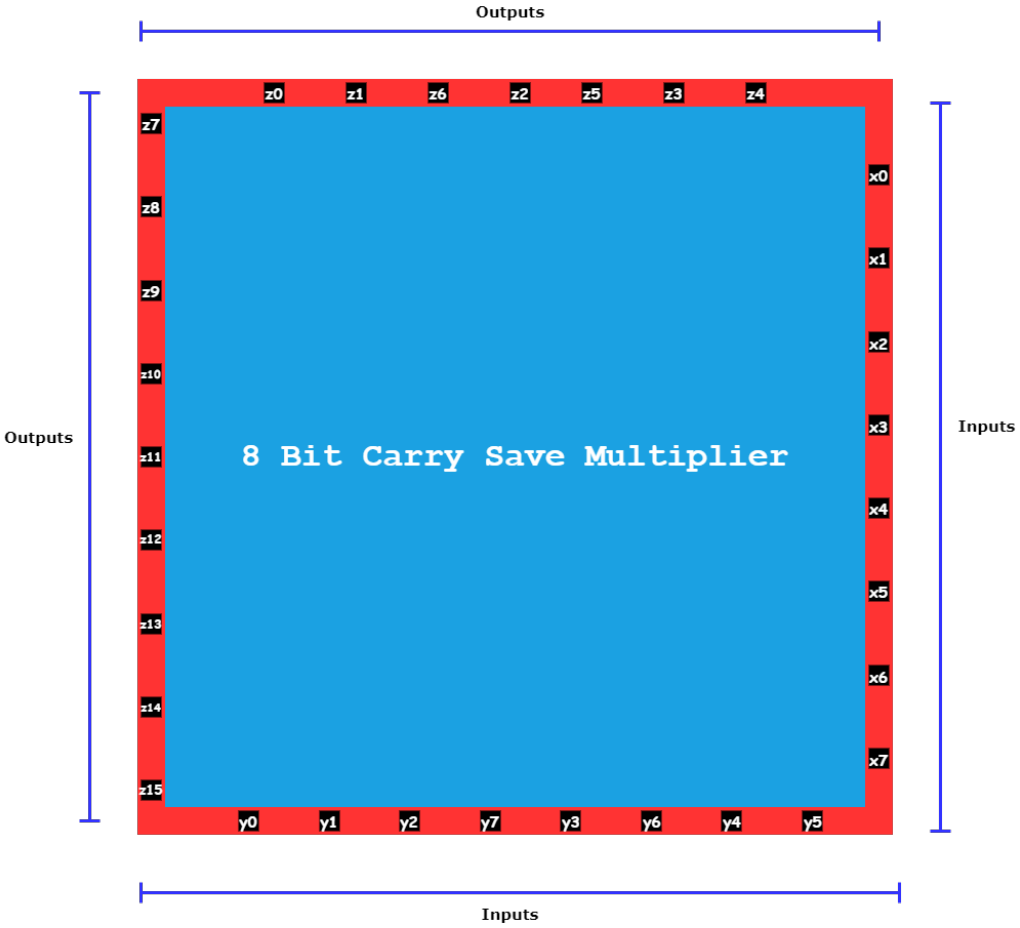


Figure 2.3: Pin Diagram

8-bit Carry Save Multiplier Schematic & Layout

3.1 Schematic of FA used

The CSM uses a 3x sized complementary Carry Out circuit, 1x sized complementary Sum circuit and 1x sized standard cells.

The schematic of the Full Adder is attached below:

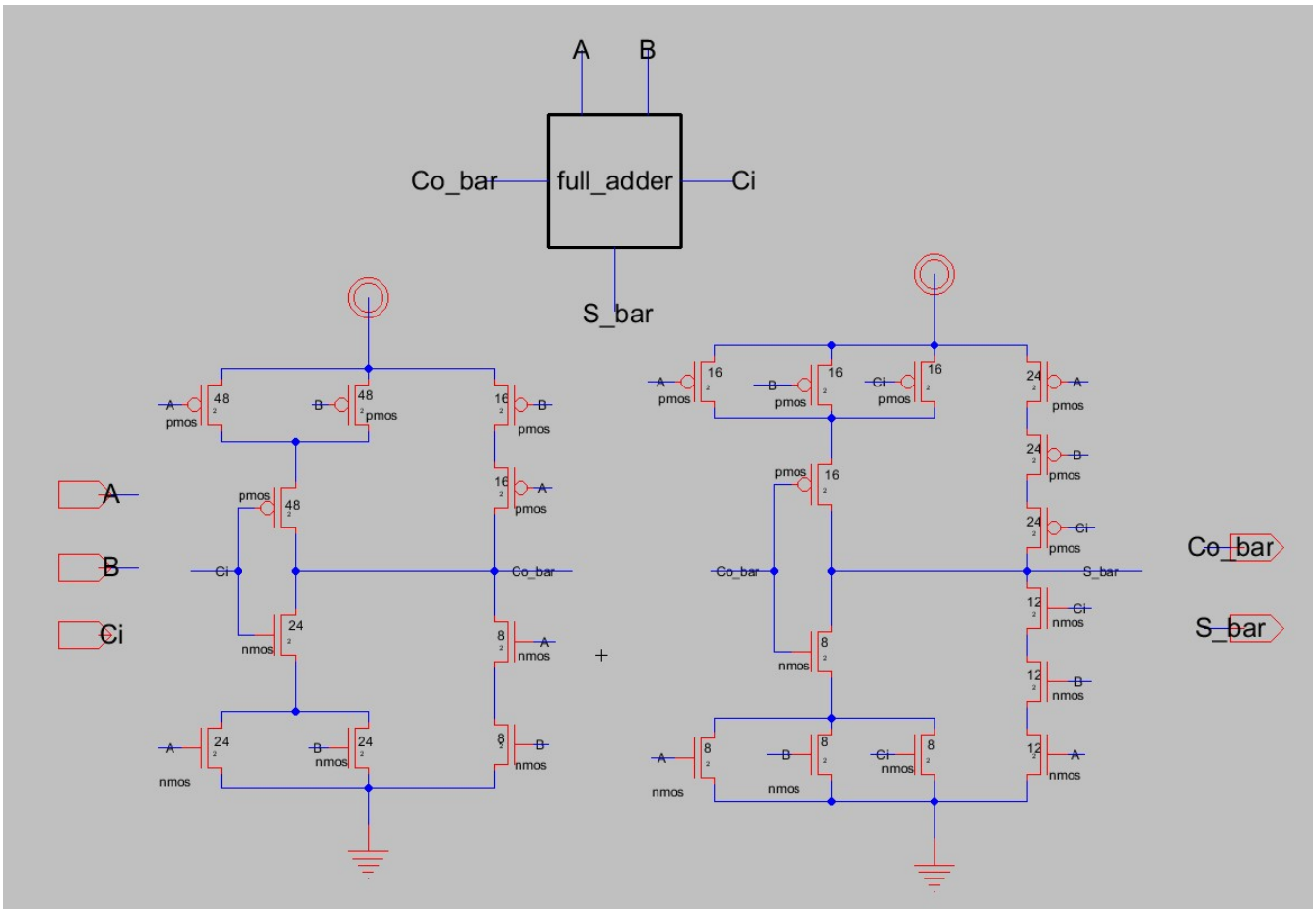


Figure 3.1: Full Adder

3.2 Block Diagram of 8-bit CSM

Instead of using inverted adders with inverters, we take advantage of the fact that the inverted sum and the carry out of the inverted added are actually special functions and give the correct outputs when all the inputs are inverted. So, to reduce the number of inverters in the critical path, we just use the above functionality in alternate rows. The inverting full adders shown in red in Figure 2.2 take inverted inputs and give the correct values.

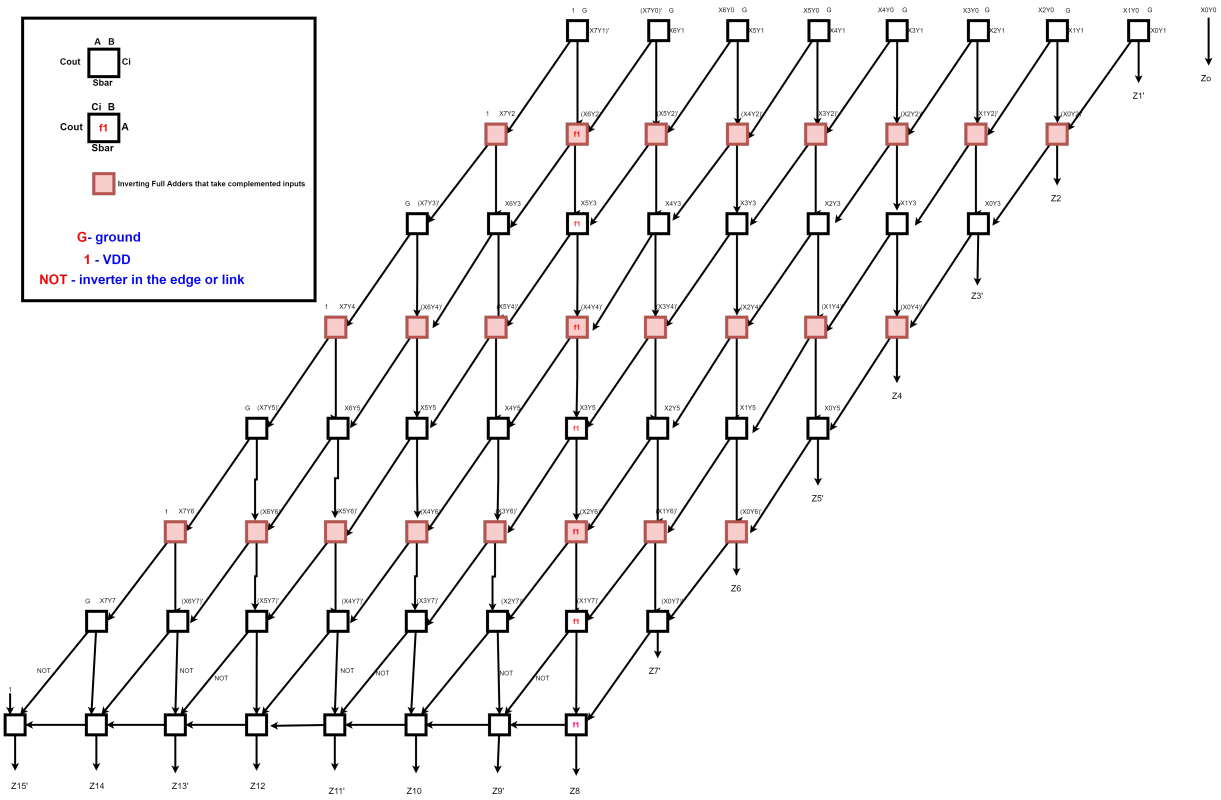


Figure 3.2: Block Diagram

3.3 Schematic of 8-bit CSM

The schematic of the Carry Save Multiplier is:

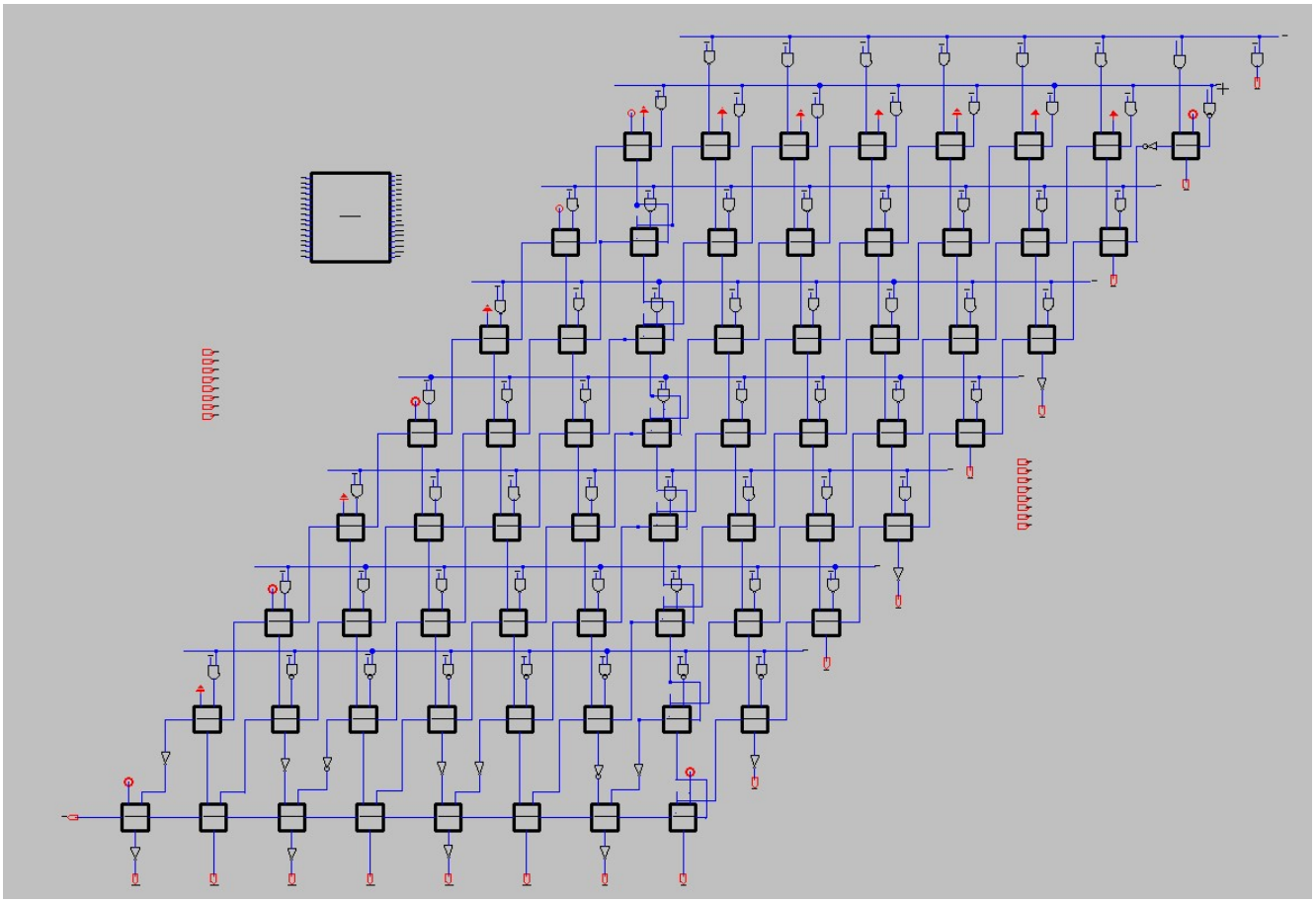


Figure 3.3: CSM Schematic

The schematic uses 64 inverting full adder circuits connected in the form of an 8x8 array with the last row being used to implement a Ripple Carry Adder Vector Merge stage.

3.4 Layout of FA used

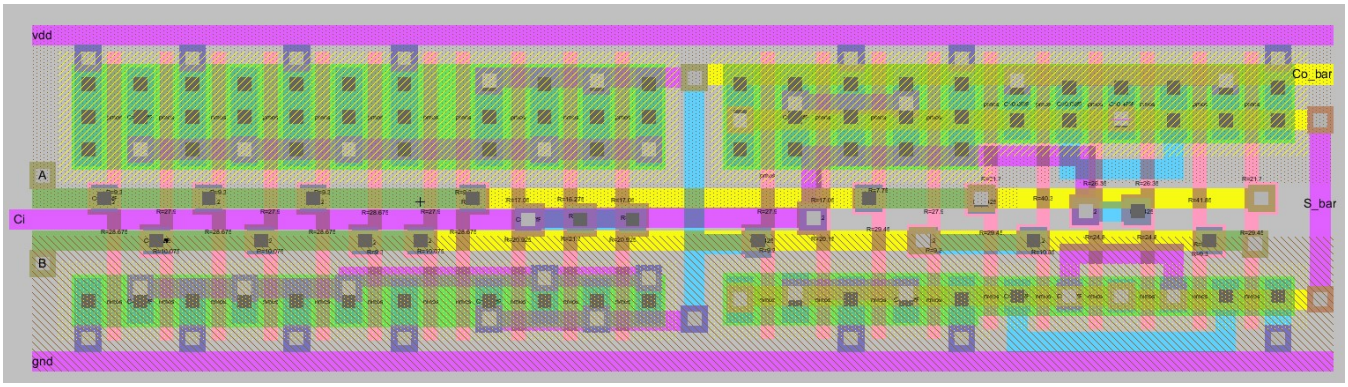


Figure 3.4: Full Adder Layout

All the subcircuits have a y-pitch of 50λ and pass DRC-LVS conditions. Only the FA layout is shown here. The back-annotated parasitics are displayed in the image.

```
================================3================================
Running DRC with area bit on, extension bit on, Mosis bit
Checking again hierarchy .... (0.01 secs)
Found 62 networks
Checking cell '8bit-CSM__full_adder{lay}'
        No errors/warnings found
0 errors and 0 warnings found (took 0.385 secs)
================================4================================
Hierarchical NCC every cell in the design: cell 'full_adder{sch}'  cell '8bit-CSM__full_adder{lay}'
Comparing: 8bit-CSM:full_adder{sch} with: 8bit-CSM:8bit-CSM__full_adder{lay}
  exports match, topologies match, sizes match in 0.056 seconds.
Summary for all cells: exports match, topologies match, sizes match
NCC command completed in: 0.082 seconds.
```

Figure 3.5: FA: DRC LVS Clean Snapshot

3.5 Layout of 8-bit CSM

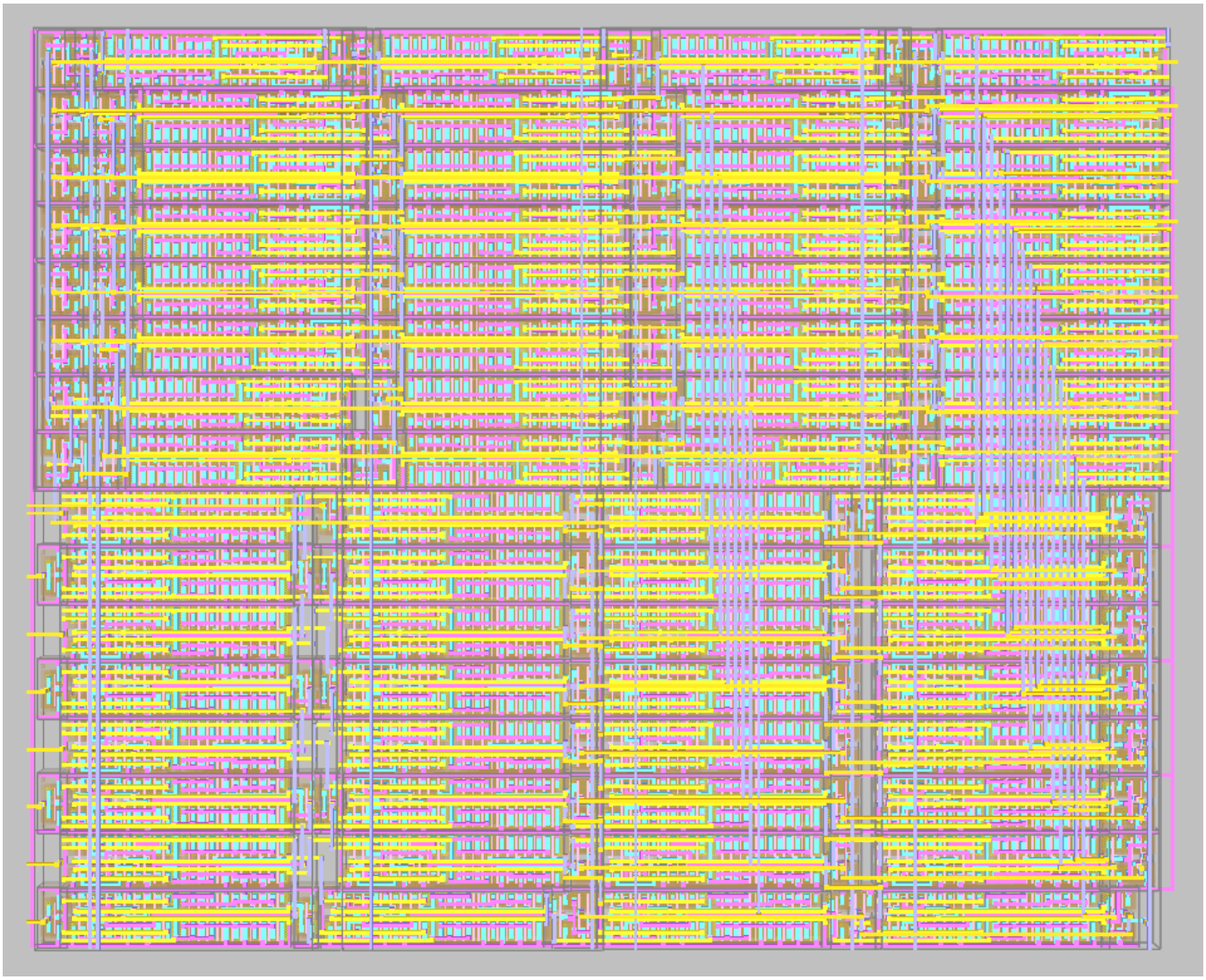
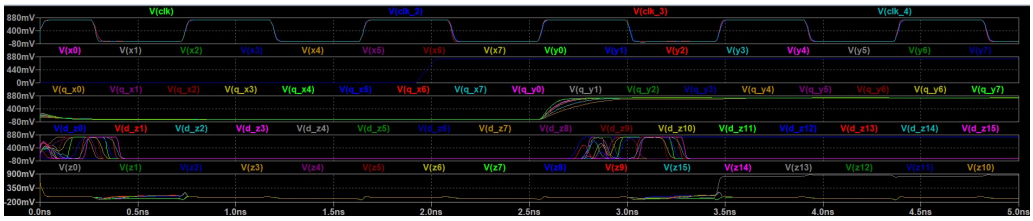


Figure 3.6: Layout

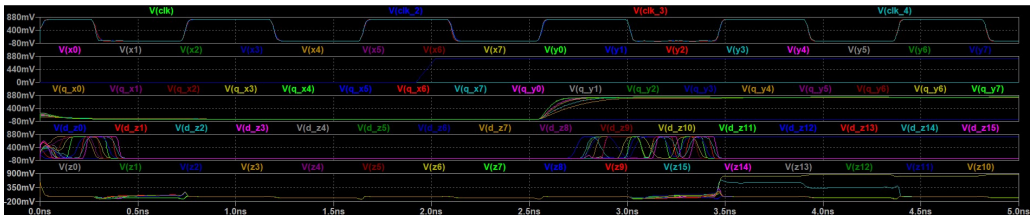
Area of Layout = 999.75λ * 803λ =802799.25 λ^2 sq. units = **97.138**μm^2

3.6 Simulation Outputs with extracted RC Netlist

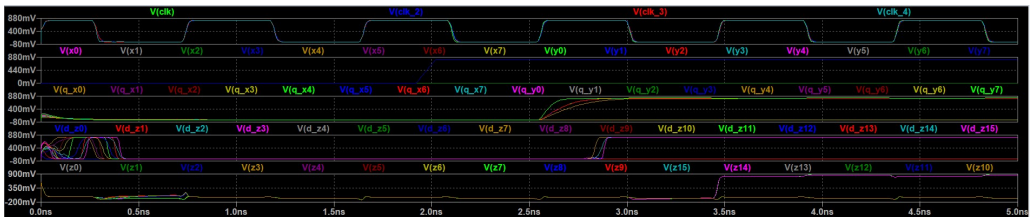
The **Multiplier functionality** was tested with **multiple (30-40) input combinations** using an automated python script that updates the spice netlist file with pulse voltage sources depending on the decimal inputs given by the user. Some of the simulation outputs are shown below:



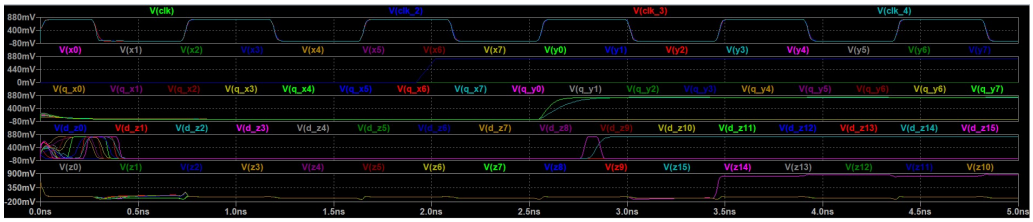
(a) -1 * -1 = +1



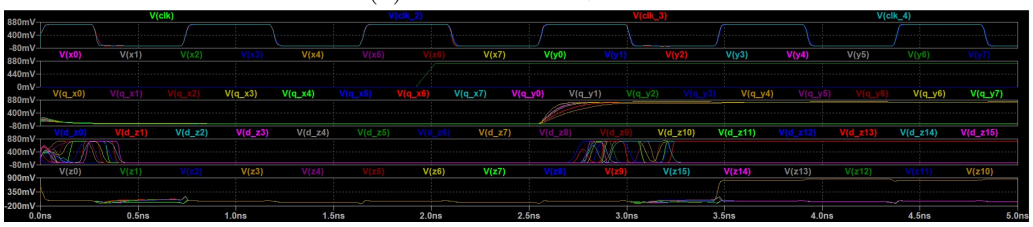
(b) -11 * -11 = +121



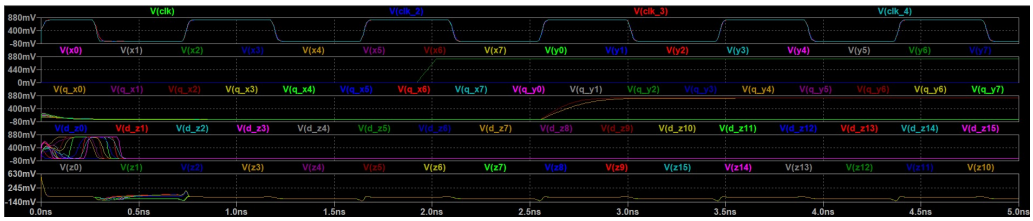
(c) 127 * -128 = -16256



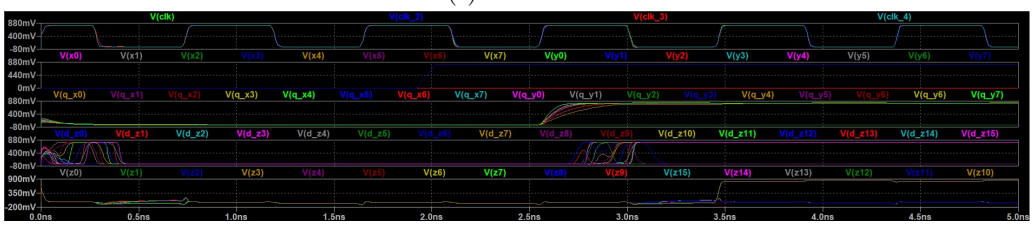
(d) -128 * -128 = +16384



(e) 127 * 127 = 16129



(f) 5* 0 = 0



(g) 71 * -5 = -355

Figure 3.7: SPICE Model Simulation Outputs

3.7 Performance Analysis of 8-bit CSM

The following testbench is used to obtain the previous simulation results:

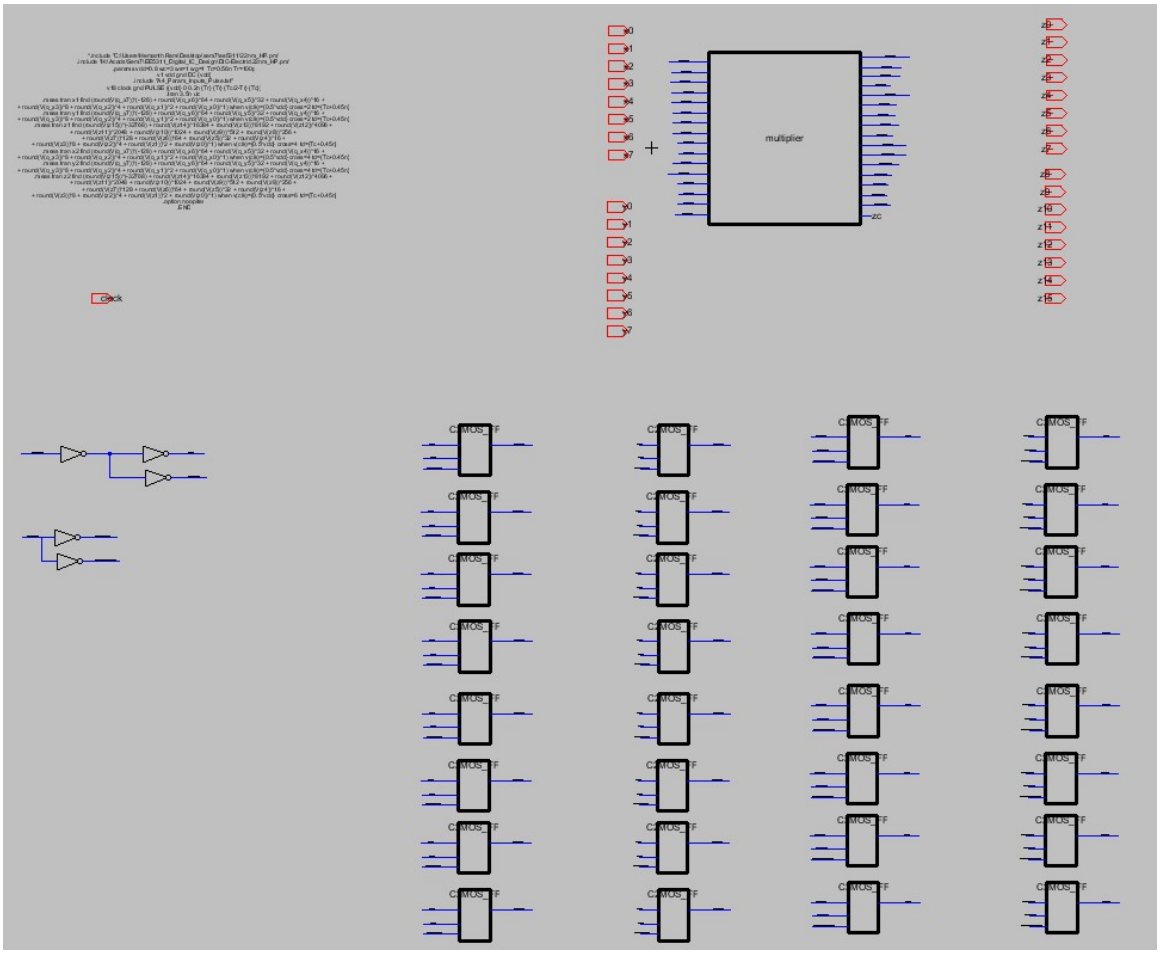


Figure 3.8: CSM Testbench

It uses a set of Launch and Capture flops (explained below) and identifies the min clock period that can be used to read the correct product using the capture flops at the next clock cycle.

To estimate the pure combinational delay of the CSM, the following testbench is used: For an input combination of -1*-1. the **combinational delay** observed is **403.75ps** for the **schematic** and **623.96ps** for the **RC extracted netlist**.

The combinational delay of the CSM included with the propagation delay of the flop can be used to theoretically estimate the min Tclk of the CSM. Taking a flop propagation delay of 51ps (derived in the next section), Such an estimation gives min Tclk as **454.75ps** for the schematic and **674.96ps** for the RC extracted netlist.

By manually reducing the clock period while ensuring the output is captured for different input combinations, the min Tclk observed is **560ps** for the schematic and **910ps** for the RC extracted netlist.

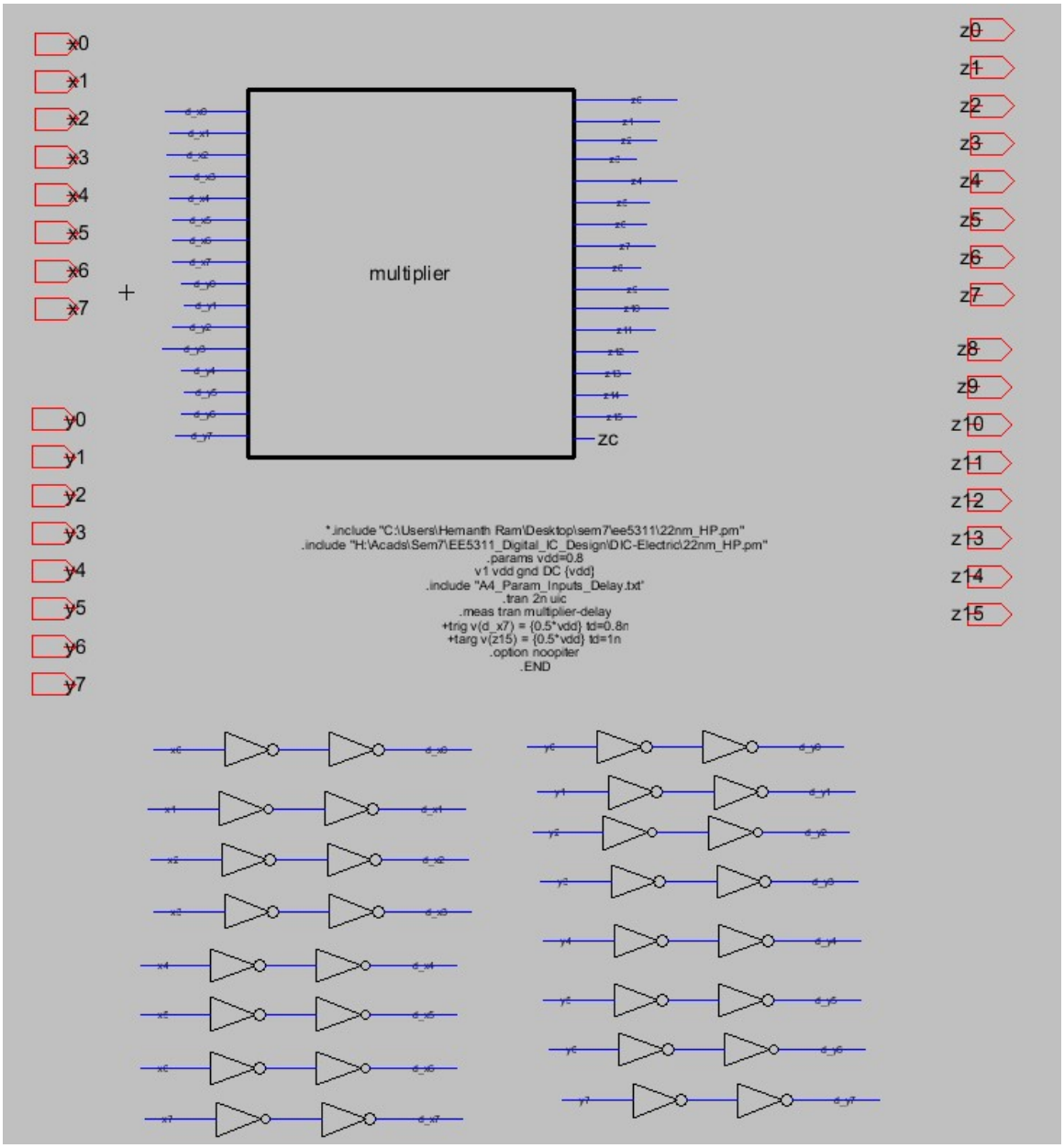


Figure 3.9: CSM Testbench-delay

Table 3.1: Performance Analysis

Theoretical Calc.	**Min Tclk**	**Max Freq**
Schematic	454.75ps	2.2GHz
RC Netlist	674.96ps	1.48GHz
Observed Results	**Min Tclk**	**Max Freq**
Schematic	560ps	1.78GHz
RC Netlist	910ps	1.1GHz

Single Stage Pipelined 8-bit CSM Schematic

4.1 Flip-Flop Schematic

The chosen implementation for the DFF is a Dynamic C2MOS Flop. It is insensitive to clock overlaps and offers reasonable setup times. The schematic is:

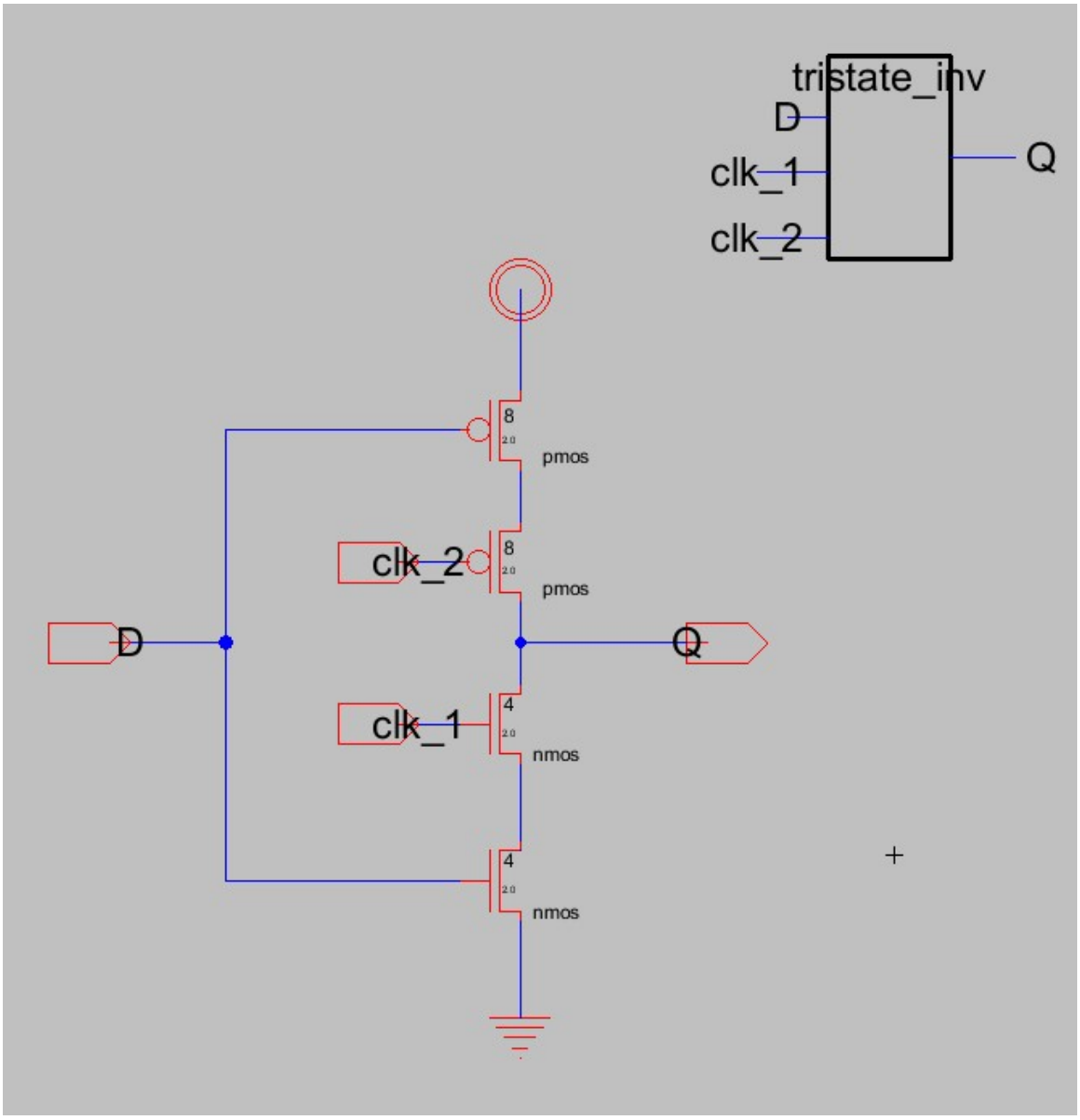


Figure 4.1: Tristate Inverter

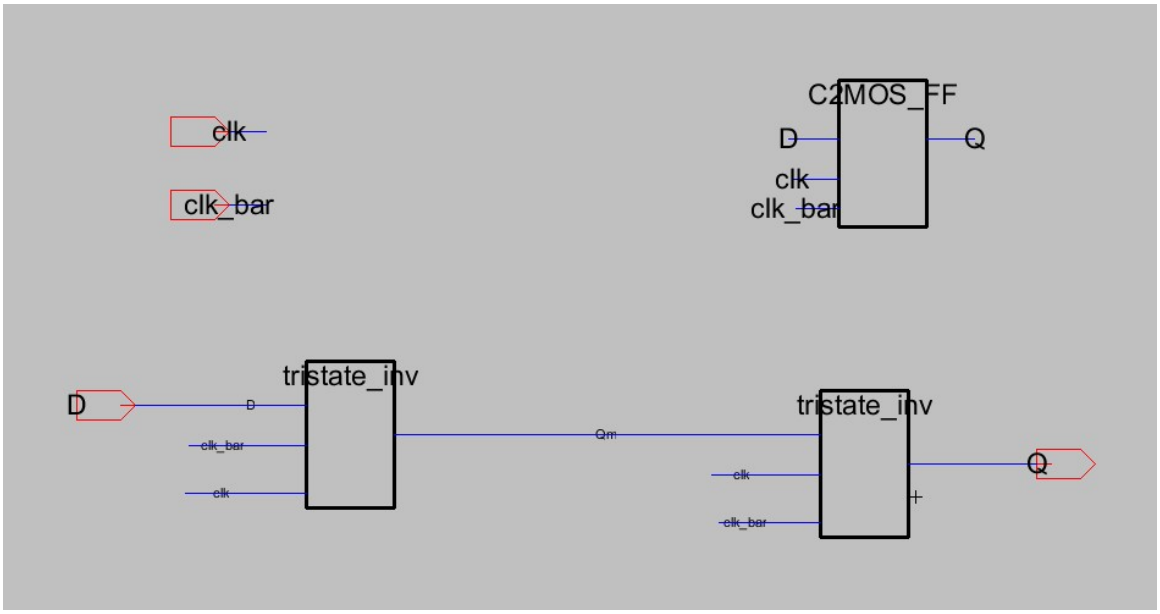


Figure 4.2: DFF

4.2 DFF Characterisation

To identify the setup time and propagation delay of the flop, the following test bench is used:

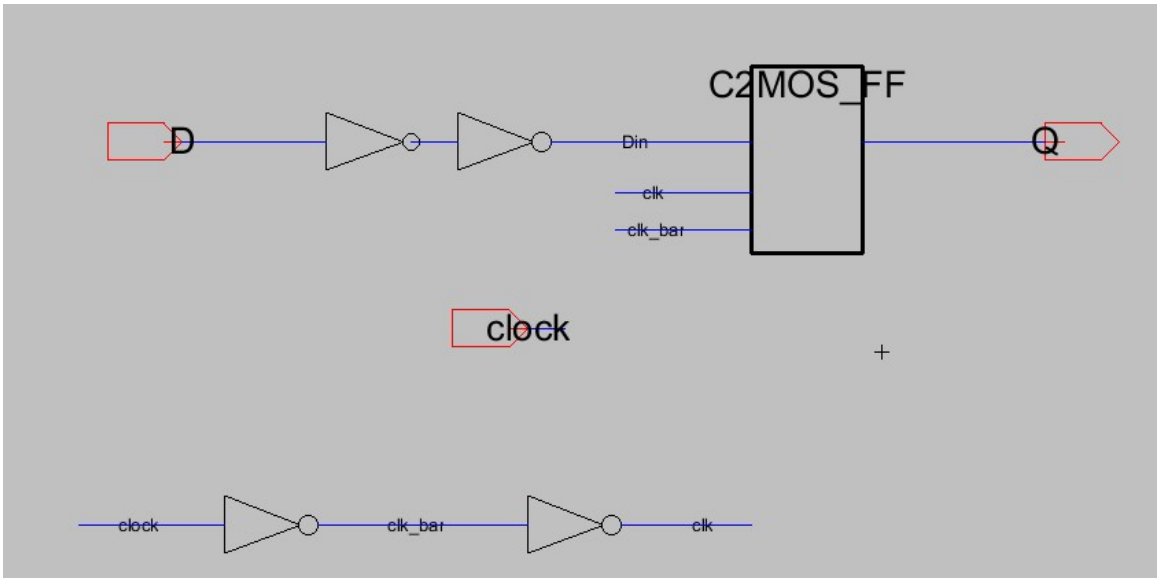


Figure 4.3: DFF Testbench

In the rising input case,

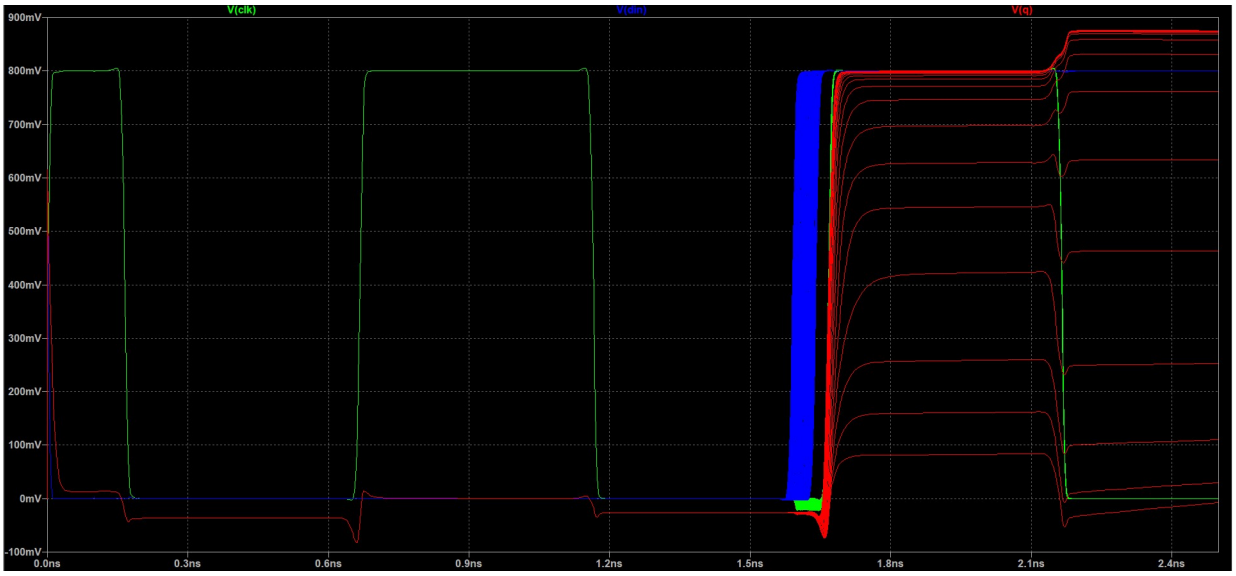


Figure 4.4: Rising input Simulation

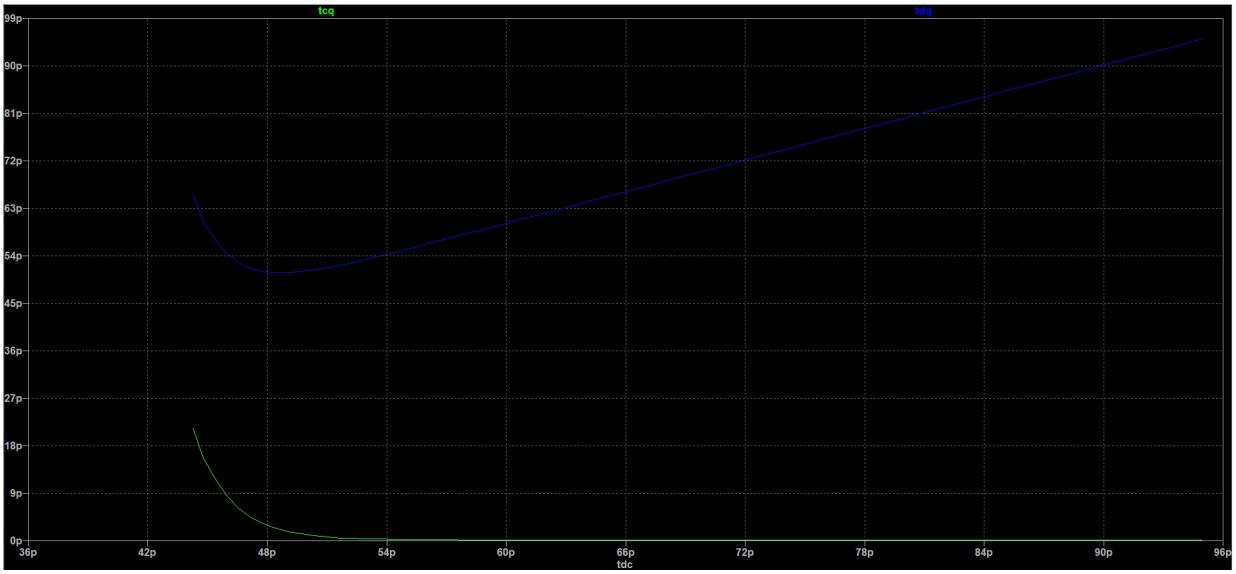


Figure 4.5: Setup-Rise time

In the falling input case,

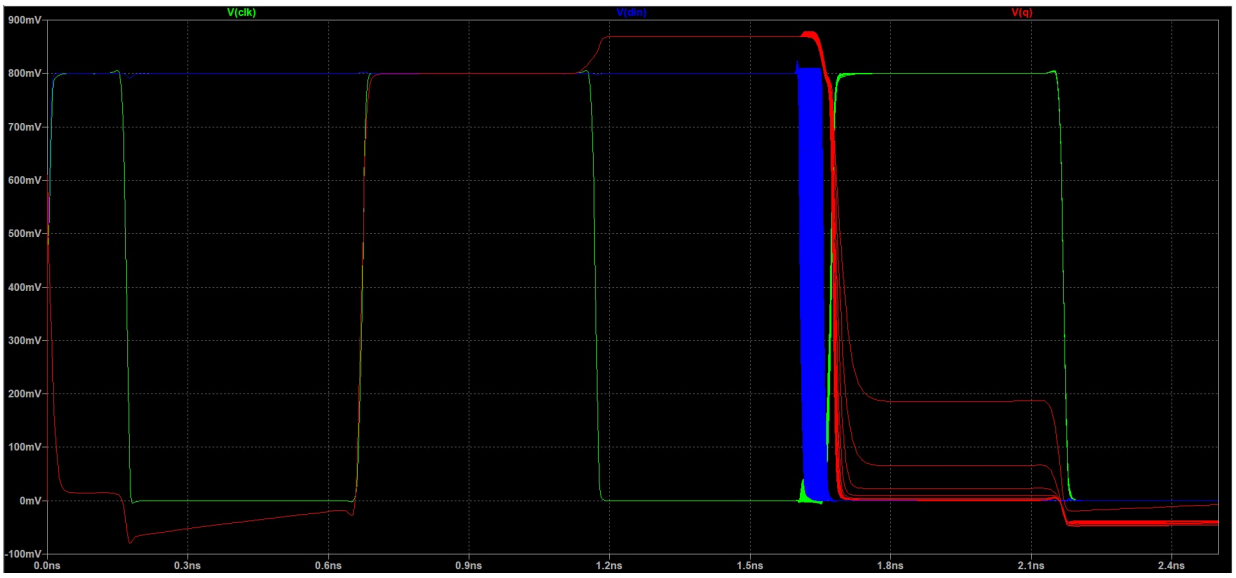


Figure 4.6: Falling input Simulation

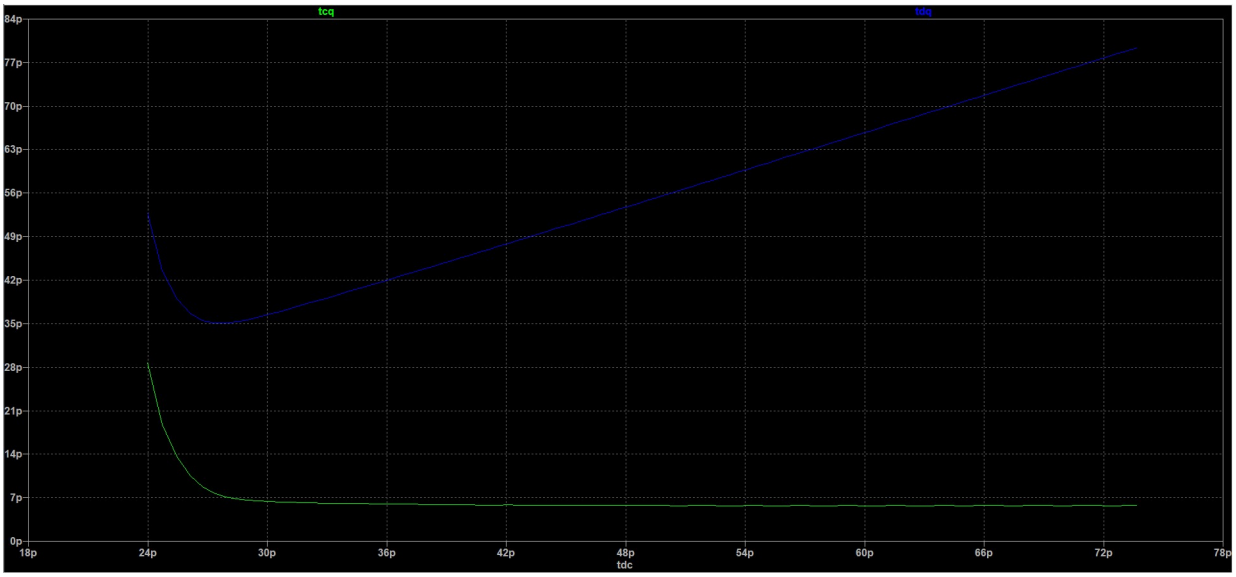


Figure 4.7: Setup-Fall time

Table 4.1: DFF Characterisation

Parameter	**Rise**	**Fall**
Tdc	48.71ps	27.91ps
Tcq	2.08ps	7.15ps
Tdq	50.79ps	35.06ps

So, the worst case **propagation delay** of the flop is **50.79ps** with a **setup time** of **48.71ps**

4.3 Identifying Location of DFFs

In order to double the throughput thorugh pipelining, the array of flops must be placed such that it divides the entire combinational block of the multiplier into 2 parts of approximately equal propagation delays. To identify that,

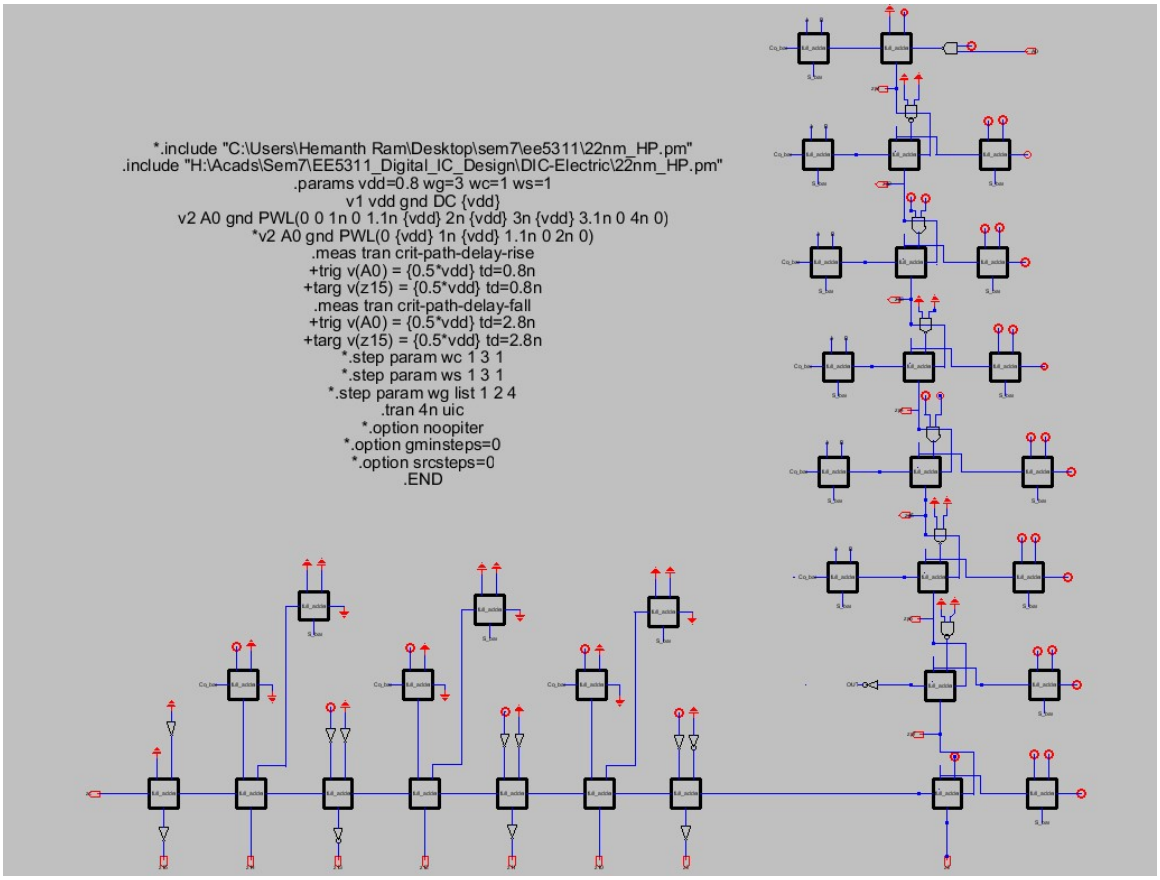


Figure 4.8: Critical Path

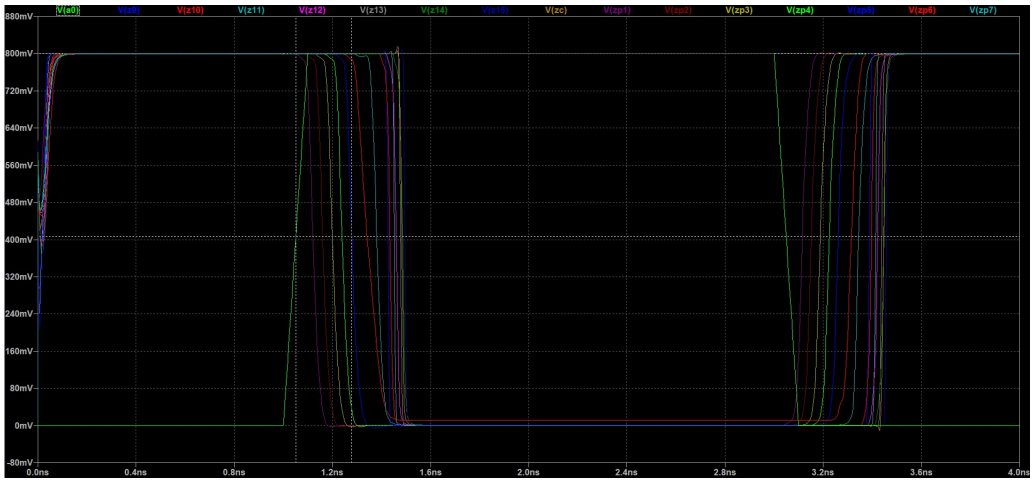


Figure 4.9: Identifying Flop Locations

The critical path delay is approx 400ps and the delay from the input to the output of the 5th adder is approx 200ps. So, the array of flops is added **after the 5th row of Full Adders**.

4.4 Schematic of Pipelined 8-bit CSM

The schematic of the pipelined 8-bit CSM is:

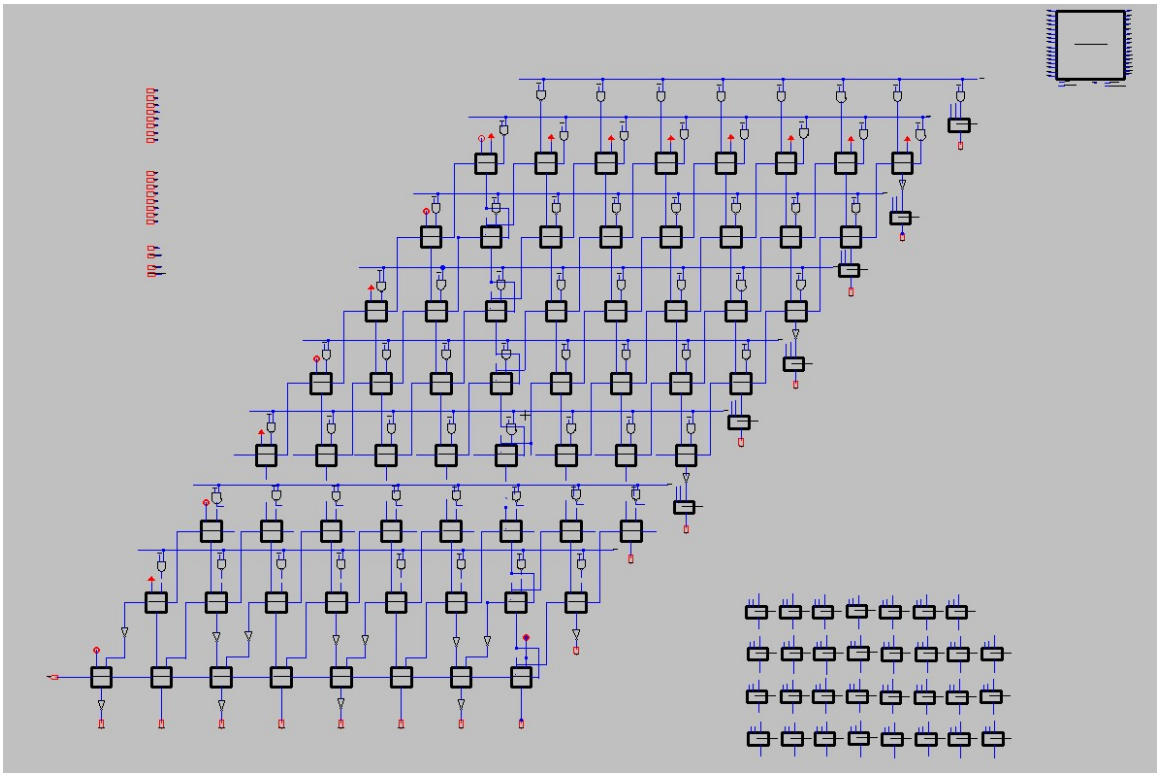
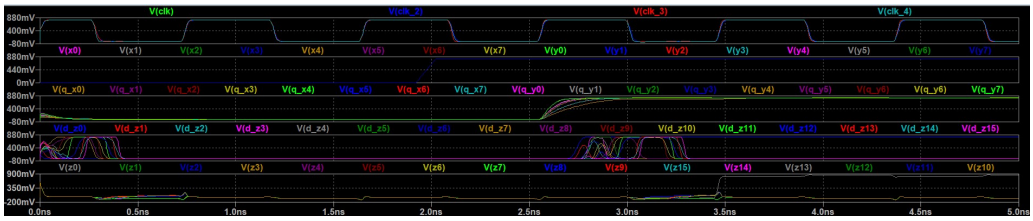


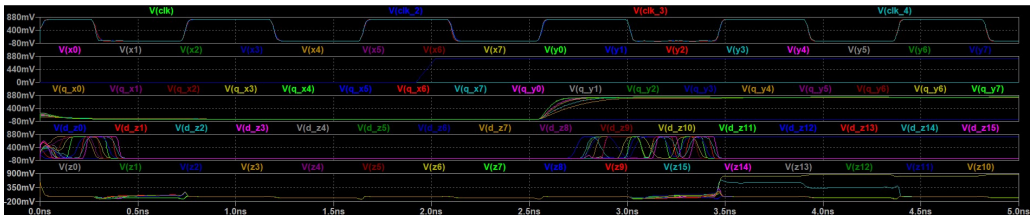
Figure 4.10: Pipelined 8-bit CSM

4.5 Simulation Outputs with RC extracted Netlist

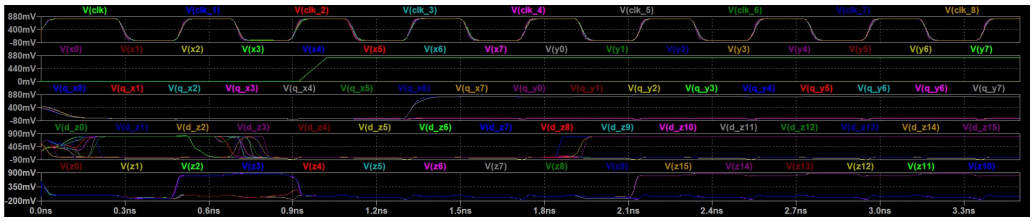
After pipelining, to check functionality and maximising clock frequency, we test the schematic with multiple input combinations. The simulations are done with block level RC extracted netlists and hence include the layout parasitics. Shown in Figure 3.11



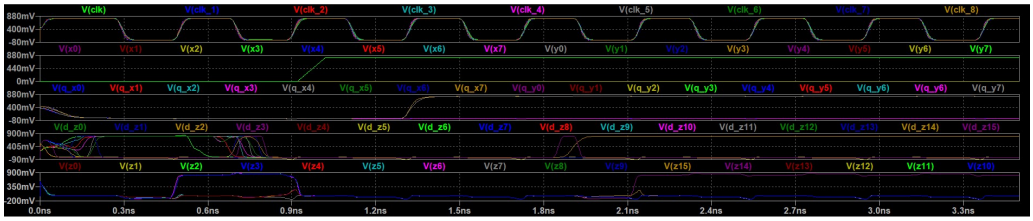
(a) -1 * -1 = +1



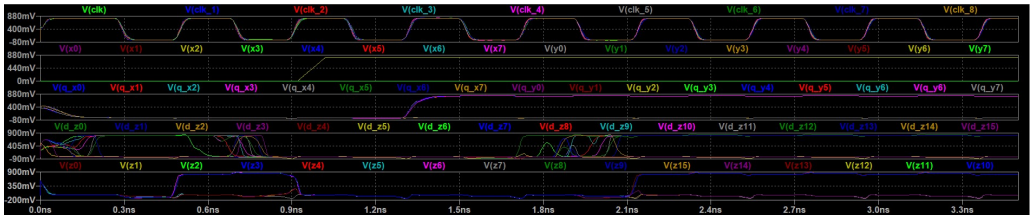
(b) -11 * -11 = +121



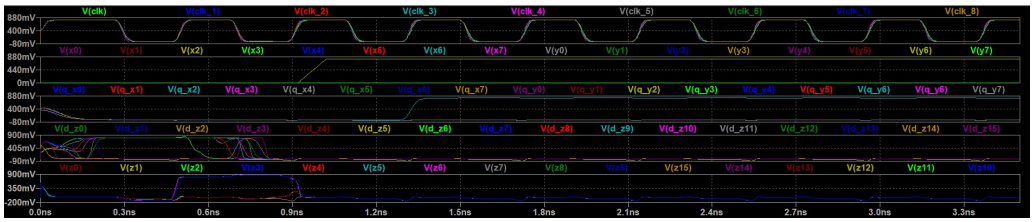
(c) 127 * -128 = -16256



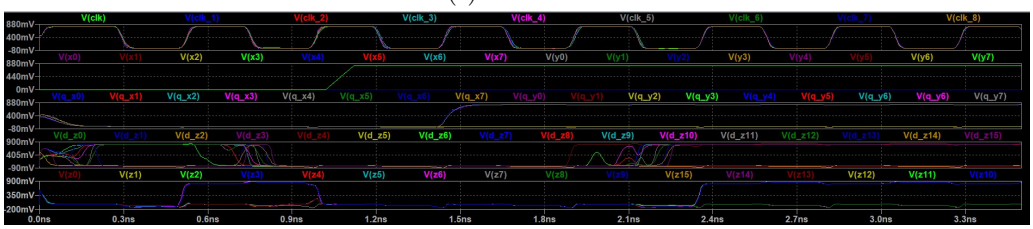
(d) -128 * -128 = +16384



(e) 127 * 127 = 16129



(f) 5* 0 = 0



(g) 71 * -5 = -355

Figure 4.11: SPICE Pipelined Model Simulation Outputs

4.6 Performance Improvement from Pipelining

By using the following testbench, above simulation plots are obtained:

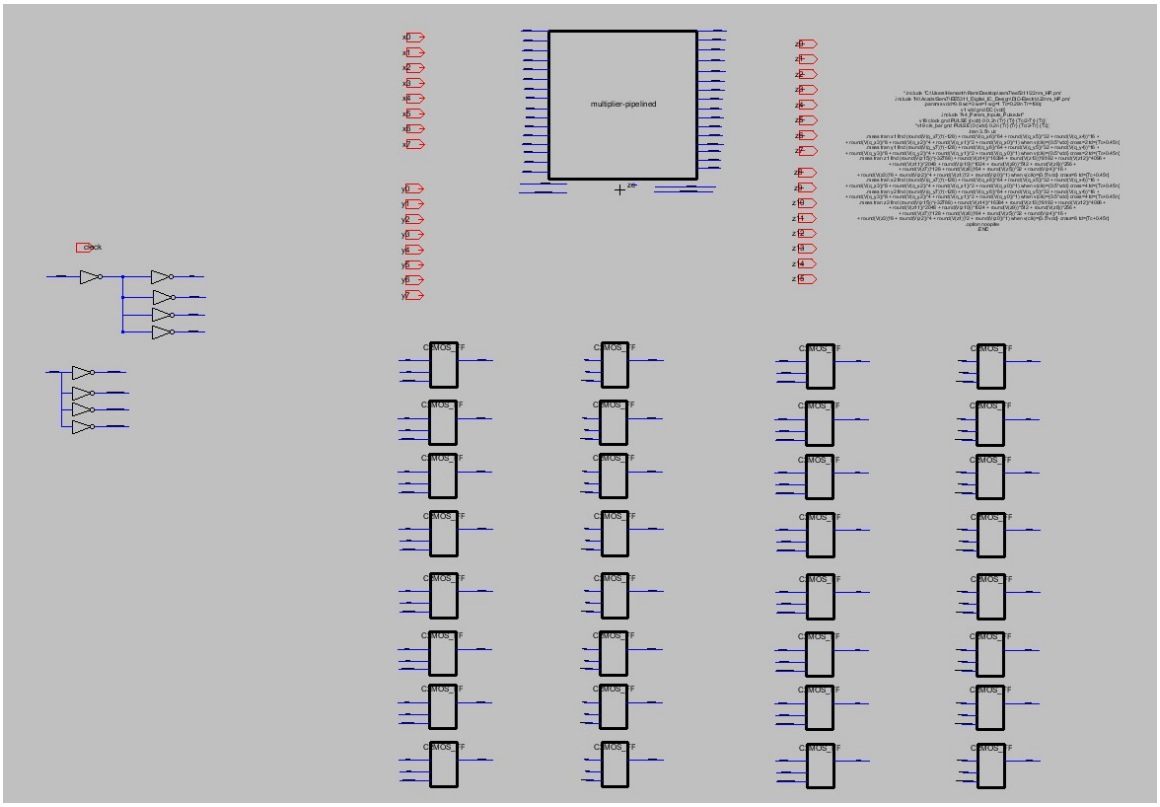


Figure 4.12: CSM Testbench-pipelined

Contrary to the approach used in the unpipelined version, here we minimise the clock period ensuring that the correct product is read at the capture flop after 2 clock cycles.

Half the combinational critical path delay is calculated as **201.87ps** for the **schematic** and **311.98ps** for the **RC extracted netlist**, when included with the sequential overhead gives a min Tclk of **252.87ps** and **352.98ps** respectively.

But by simulating over multiple input combinations, the min Tclk observed is **310ps** for the **schematic** and **460ps** for the **RC extracted netlist**

The RC netlist simulations are done using the block level RC extracted netlists and may contain less parasitics than the actual layout generated netlist. And the **decrement in performance** because of using the RC extracted netlist instead of the schematic is by **1.484** times in the **pipelined** case and **1.618** times in the **un-pipelined** case.

Table 4.2: Performance Analysis - Effect of Pipelining

Theoretical Calc.	**Min Tclk**	**Max Freq** (pipelined)	**Max Freq** (un-pipelined)	**Improvement factor**
Schematic	252.87ps	3.95GHz	2.2GHz	1.79
RC Netlist	352.98ps	2.83GHz	1.48GHz	1.91
Observed Results	**Min Tclk**	**Max Freq** (pipelined)	**Max Freq** (un-pipelined)	**Improvement factor**
Schematic	310ps	3.22GHz	1.78GHz	1.81
RC Netlist	460ps	2.17GHz	1.1GHz	1.98

8-bit CSM with Alternate Vector Merge Stage

5.1 Ripple Carry Vector Merge Stage

The current CSM schematic uses a Ripple-Carry adder in the vector merge stage. Here, each full adder takes its Cin from the Cout of the previous full adder. Hence, the critical path includes the time taken for the carry to propagate throughout the VM stage.

5.2 Carry Look-Ahead Added Vector Merge Stage

The ripple carry vector merge stage is replaced with the carry look ahead-adder vector merge stage. To avoid the MOSFET stack size from becoming too high, we cascade 2 4-bit CLA to get a 8-bit CLA. The CLA VM stage will be:

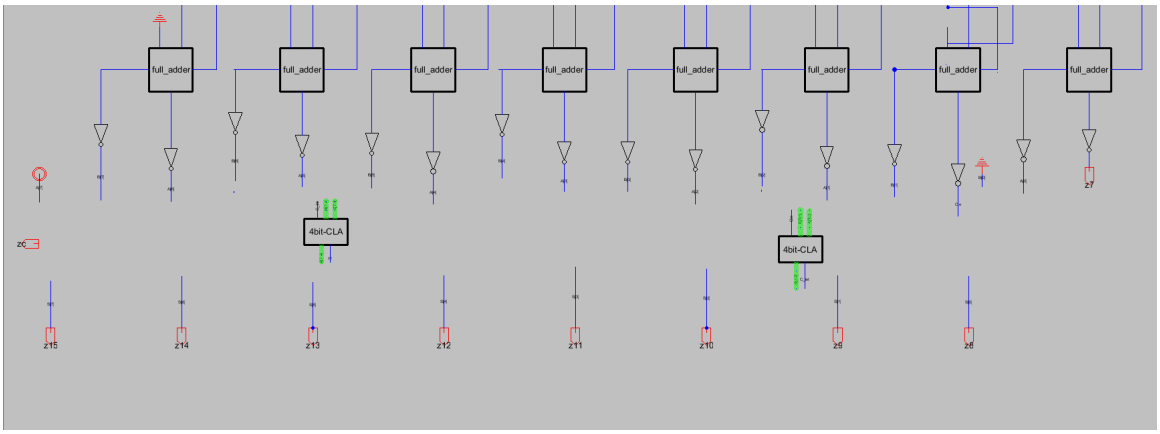


Figure 5.1: CLA VM Stage

The complete optimised multiplier schematic with a CLA VM is:

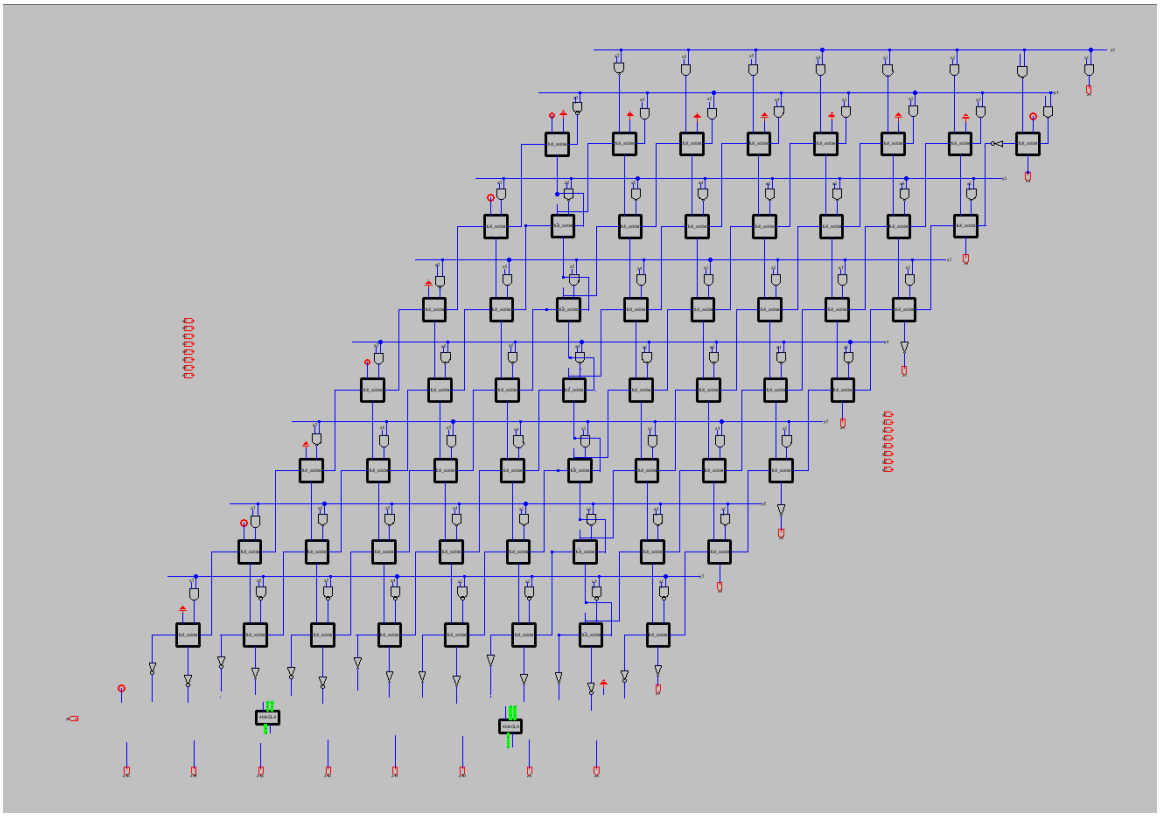


Figure 5.2: CSM with CLA VM

5.3 Carry Look-Ahead Adder Schematics

The CLA uses XOR gates that aren't used anywhere else in the circuit. The schematics of the XOR gate are:

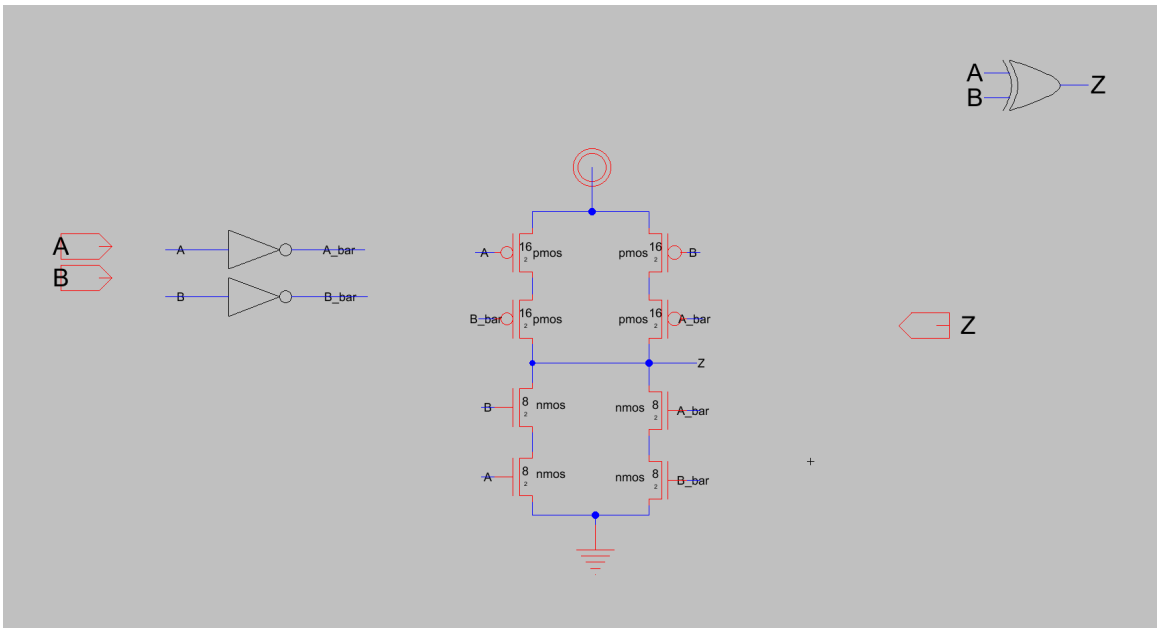


Figure 5.3: XOR Gate

Internal schematics of the CLA:

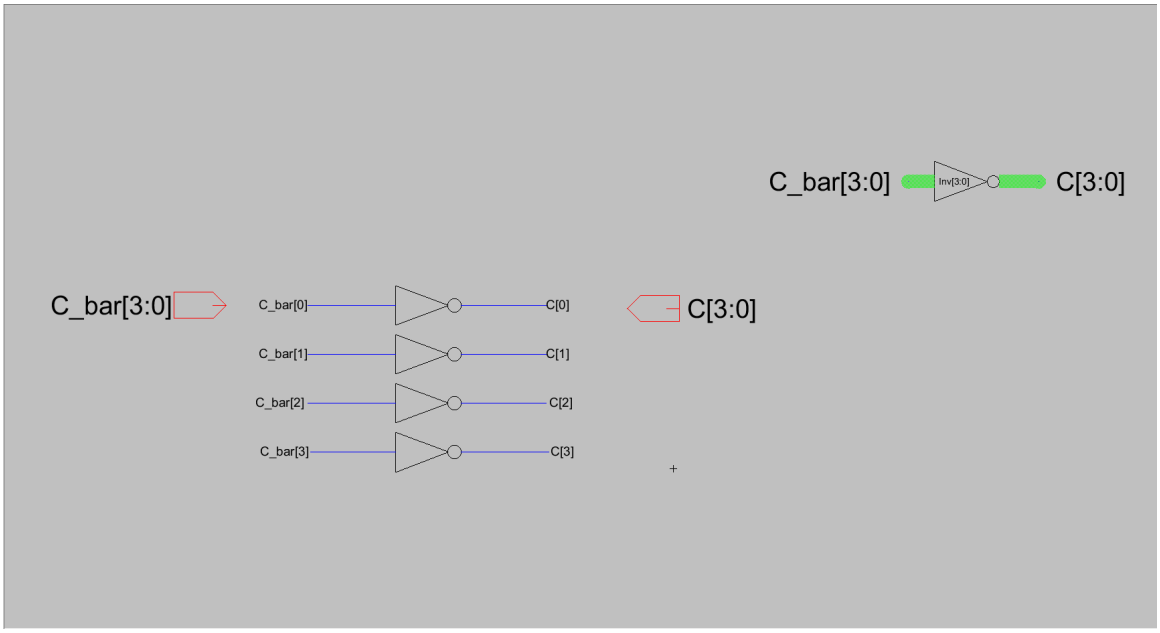


Figure 5.4: 4-bit NOT Gate

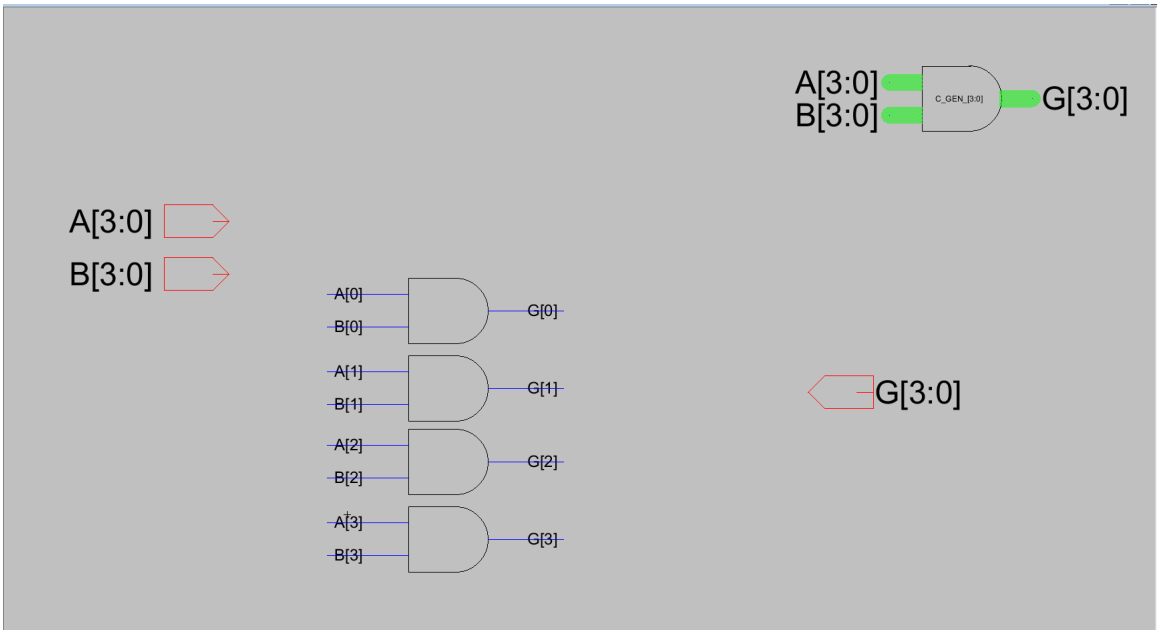


Figure 5.5: Carry Gen stage

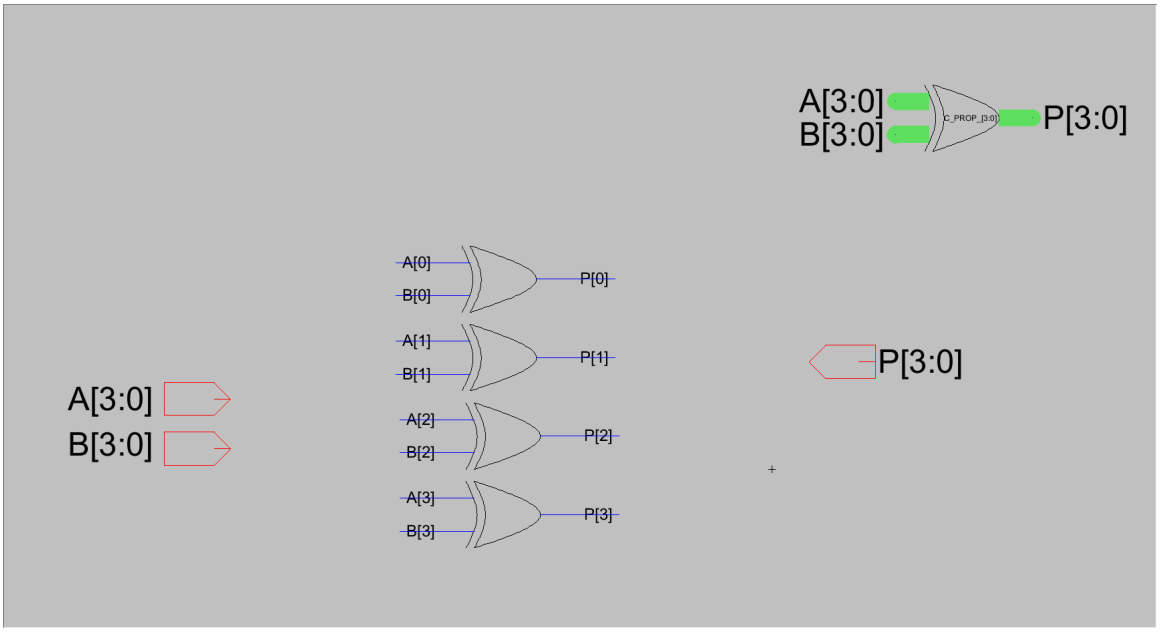


Figure 5.6: Carry Prop stage

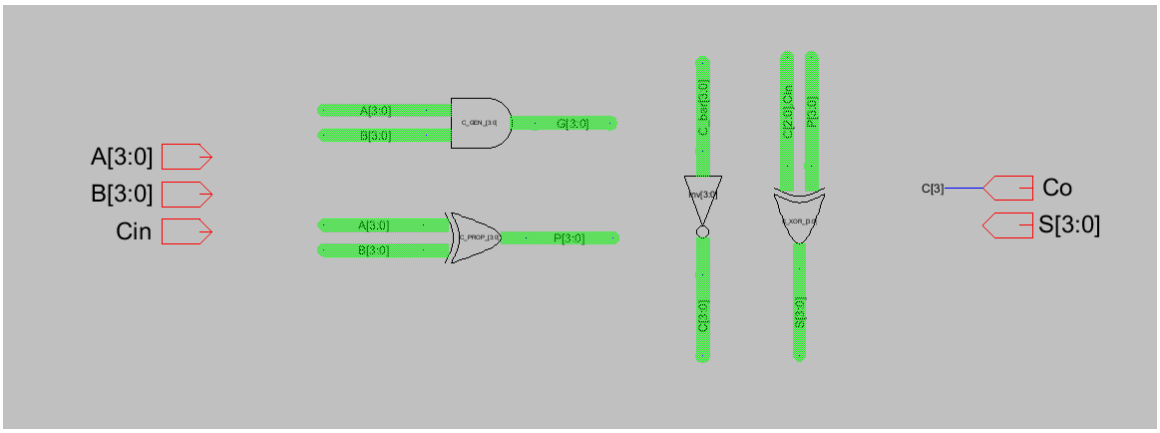


Figure 5.7: P,G generation

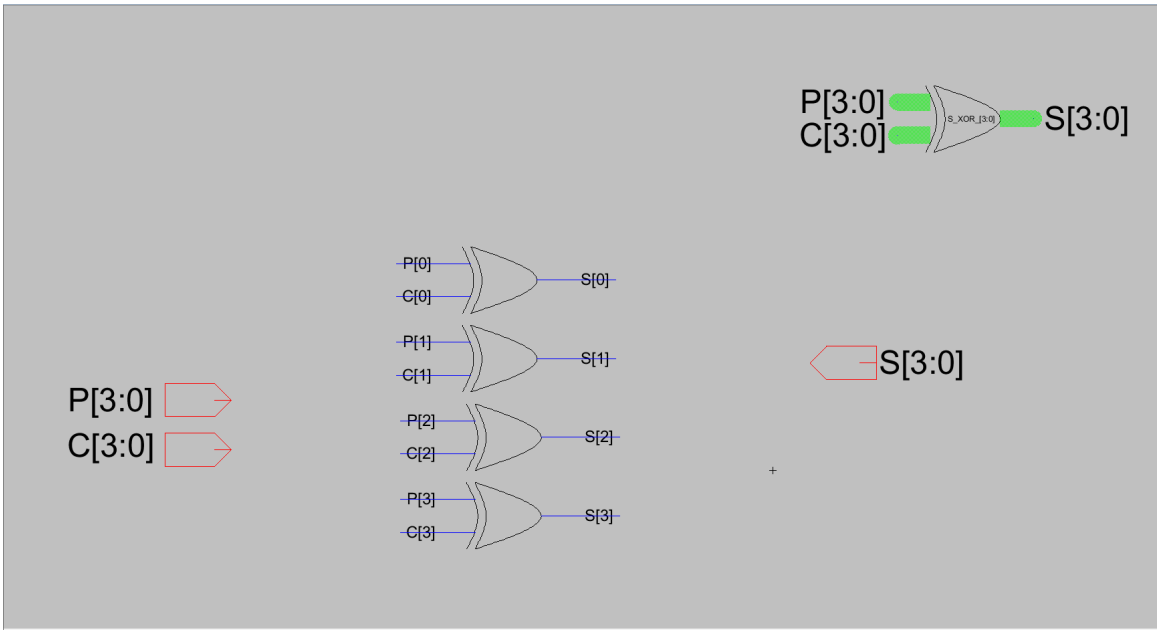


Figure 5.8: Sum schematic

The schematic of the complete CLA:

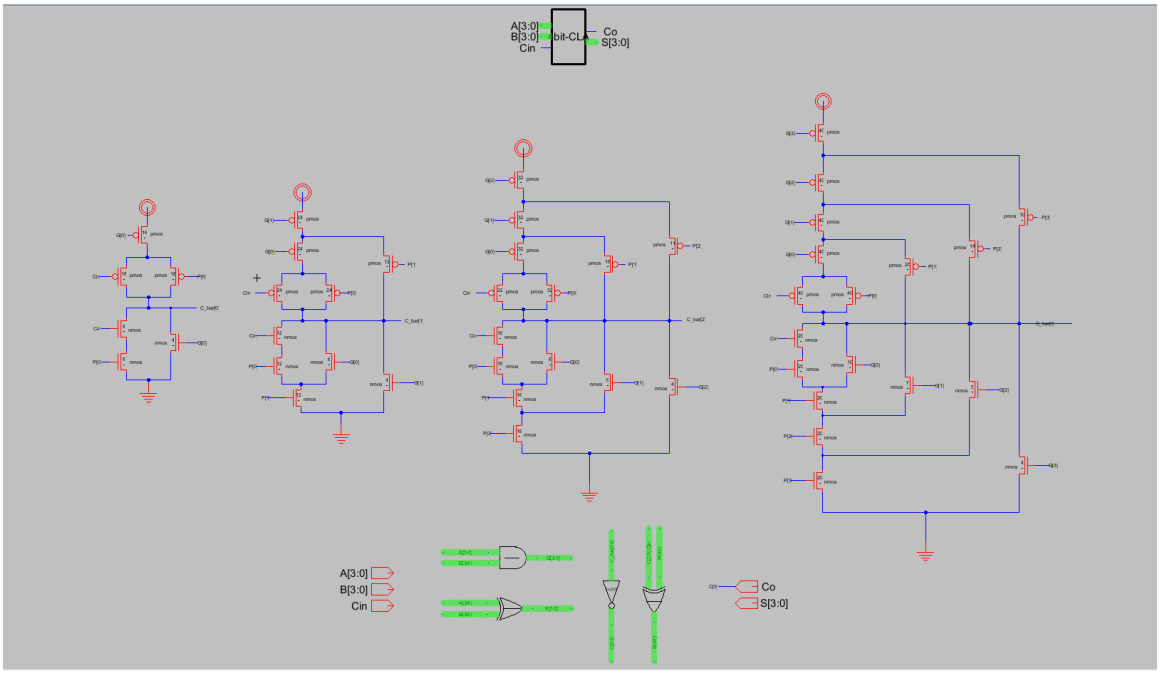


Figure 5.9: CLA Schematic

A zoomed in version to see sizes:

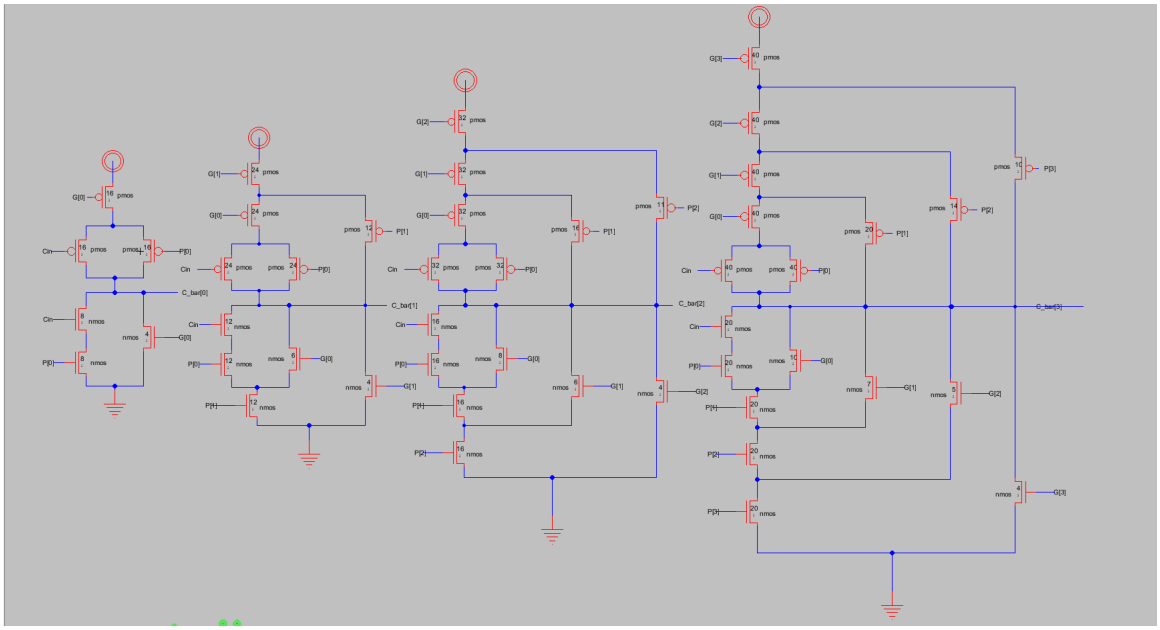


Figure 5.10: CLA Schematic-zoomed in

5.4 Performance Improvement from CLA

The VM stage is entirely a part of the critical path in the 8-bit Carry Save Multiplier. So, optimising the VM stage will give a performance improvement only in those cases where the VM stage is triggered. We analysed 3 specific input combinations and analysed the performance improvement:

Case 1: **1 * 1**: 30.3% improvement

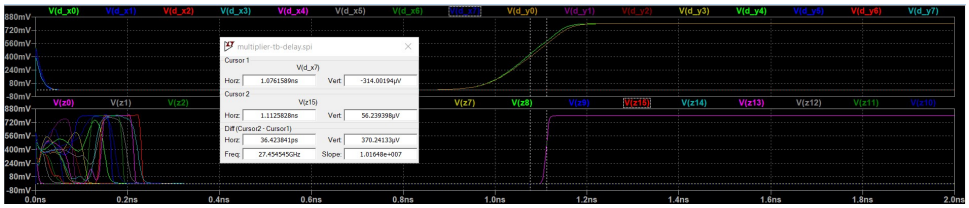


Figure 5.11: 1*1: Ripple Carry VM

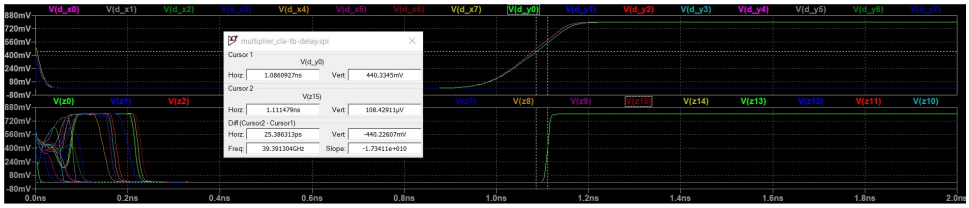


Figure 5.12: 1*1: CLA VM

Case 2: **-11 * -11**: 13.22% improvement

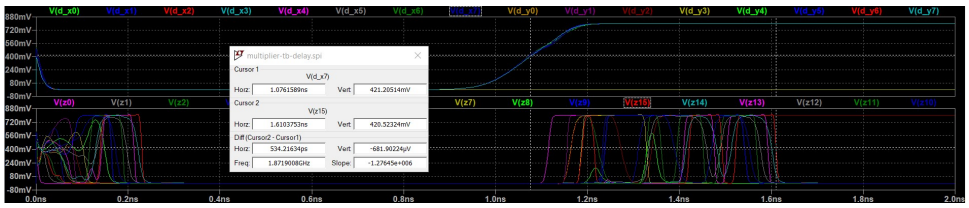


Figure 5.13: -11*-11: Ripple Carry VM

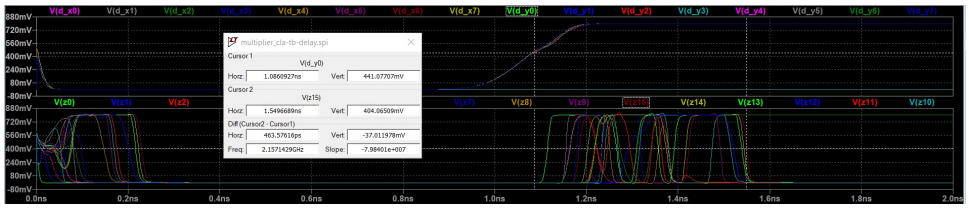


Figure 5.14: -11*-11: CLA VM

Case 3: **-75 * -1**: 4.48% improvement

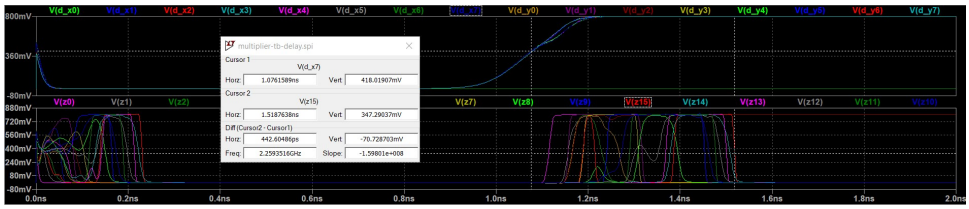


Figure 5.15: -75*-1: Ripple Carry VM

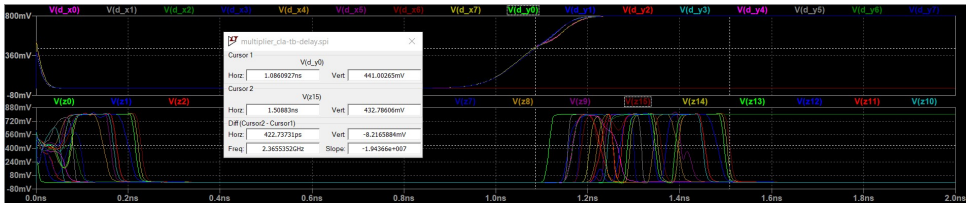


Figure 5.16: -75*-1: CLA VM

5.5 Linear Carry Select Adder Vector Merge Stage

The Carry select adder is optimal when the carry input is the critical input and the other input bits are already available. So, it calculates the output for both possible carry in values and feeds them into a 2:1 multiplexer. When the carry in arrives, it just switches the mux to fix the right output. So, the critical path delay only includes the delay of the mux instead of the delay of the ripple carry portion.
We build a 4-bit carry select adder and cascade 2 of them to form a 8-bit adder.The alternate VM stage using the CS Adder is:

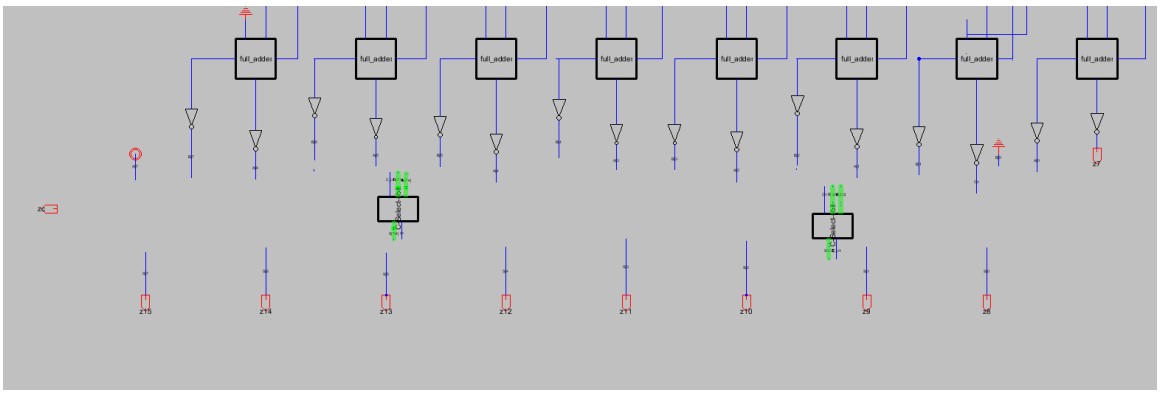


Figure 5.17: CS Adder VM Stage

5.6 Carry Select Adder Schematics

The internal schematics used in the Carry Select Adder are:

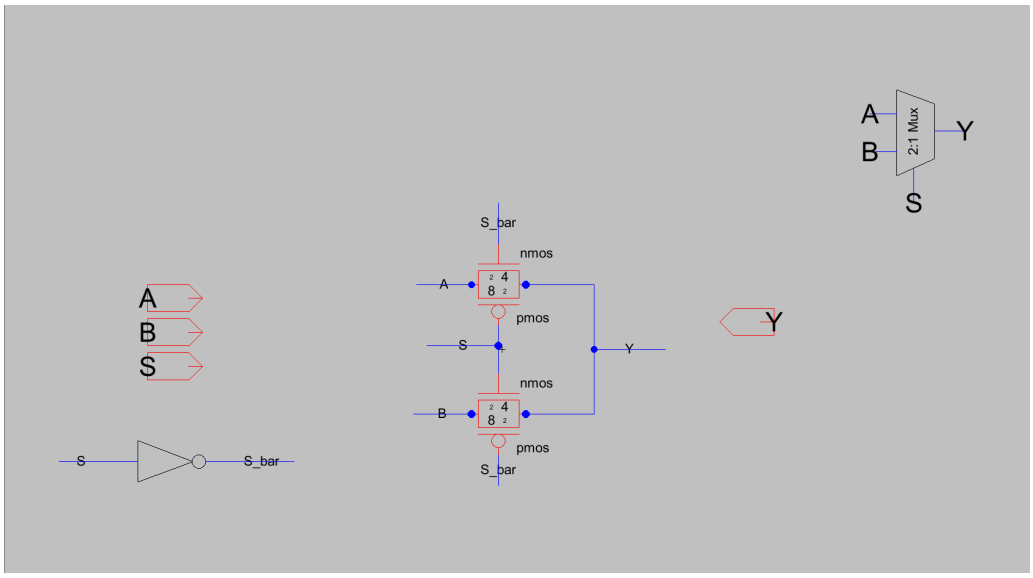


Figure 5.18: 2:1 Mux

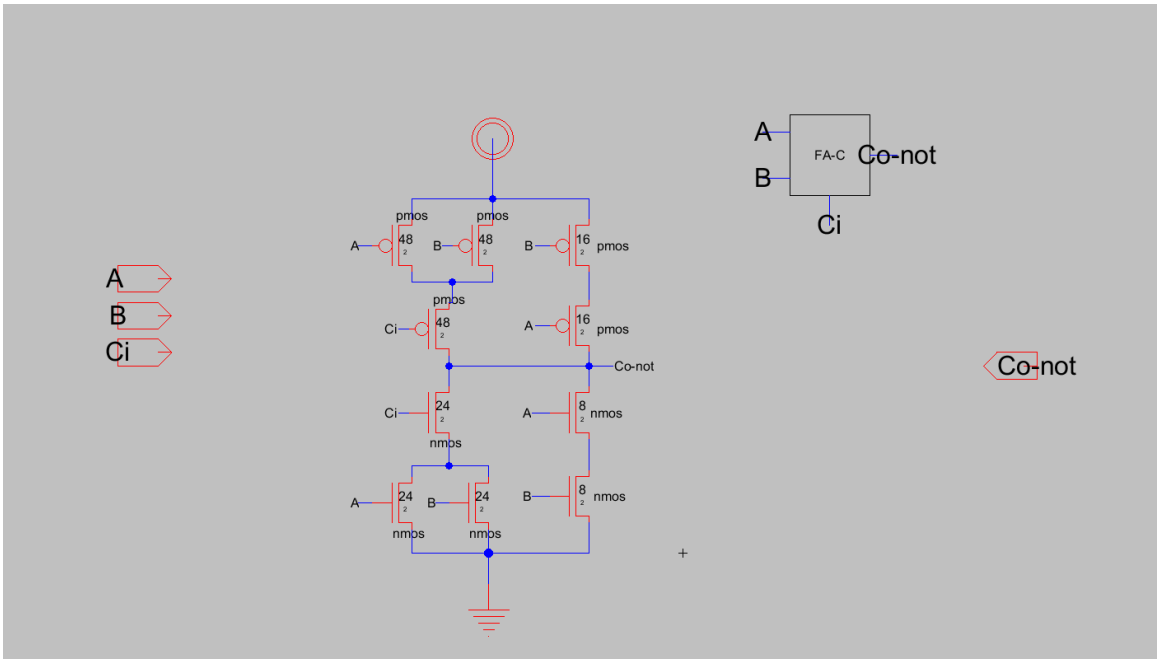


Figure 5.19: FA used in Carry Select adder

The complete schematic of the Carry Select Adder used is:

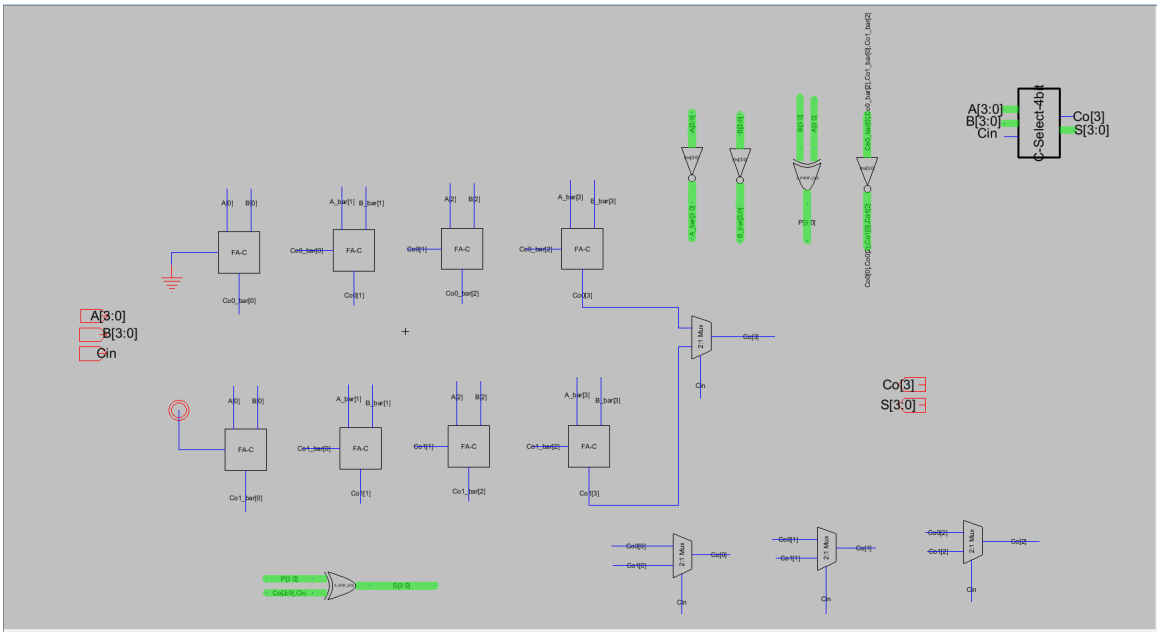


Figure 5.20: CS Adder Schematic

5.7 Performance Improvement from CS Adder

Similar to the analysis with the Carry Look-Ahead adder, we passed the same input combinations to the multiplier with a Carry Select Adder VM and analysed the combinational delay. The improvements are calculated w.r.t the ripple carry VM stage.

Case 1: **1 * 1**: 47.5% improvement

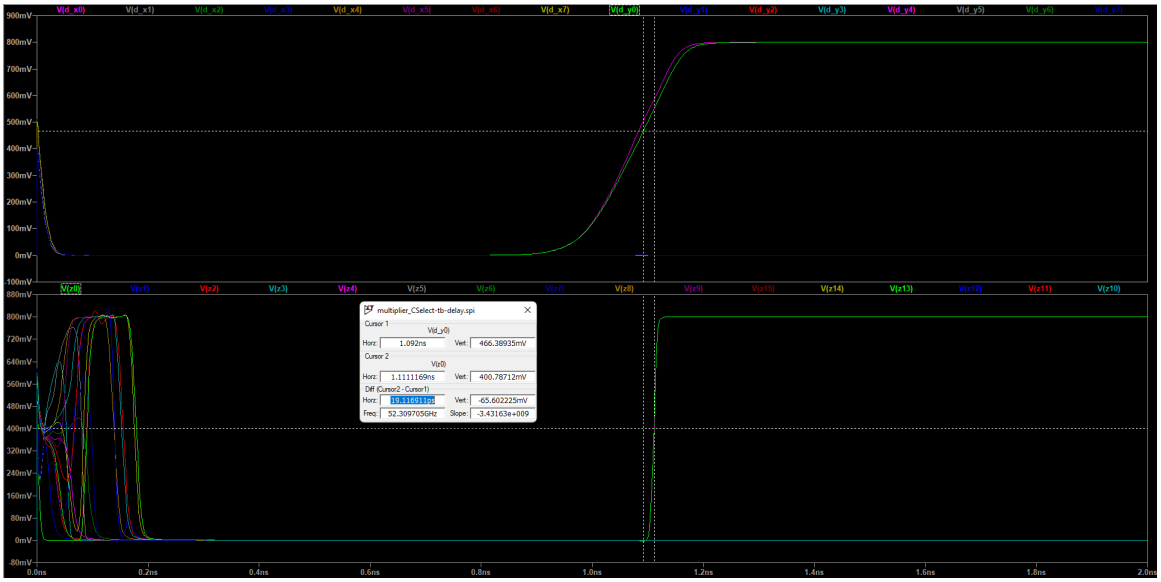


Figure 5.21: 1*1: CS_A VM

Case 2: **-11 * -11**: 5.91% improvement

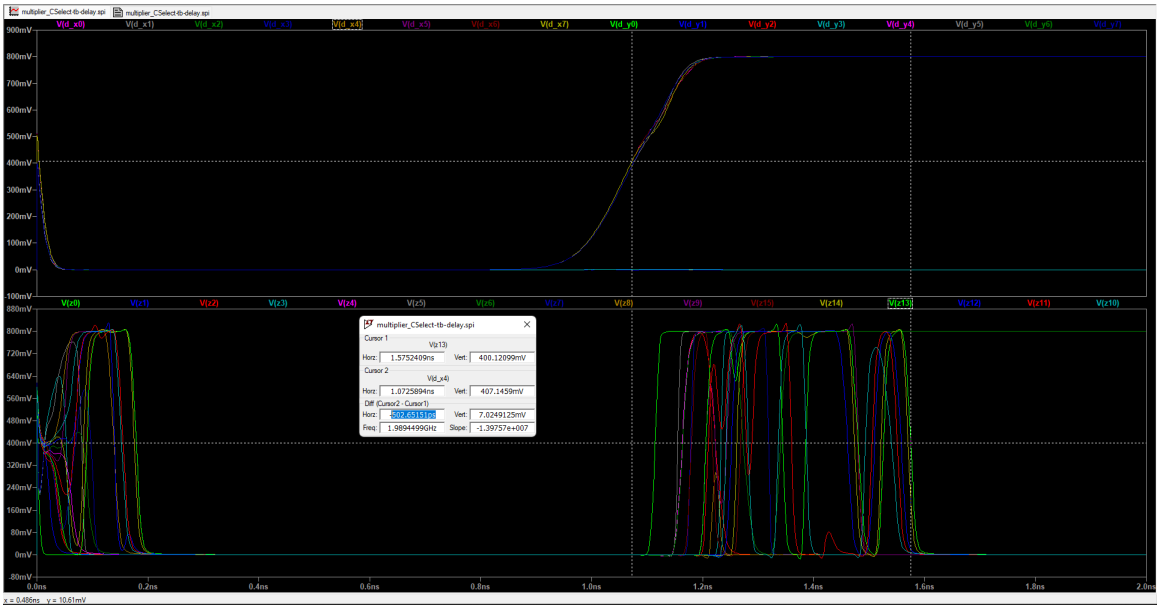


Figure 5.22: -11*-11: CS_A VM

Case 3: **-75 * -1**: 19.96% improvement

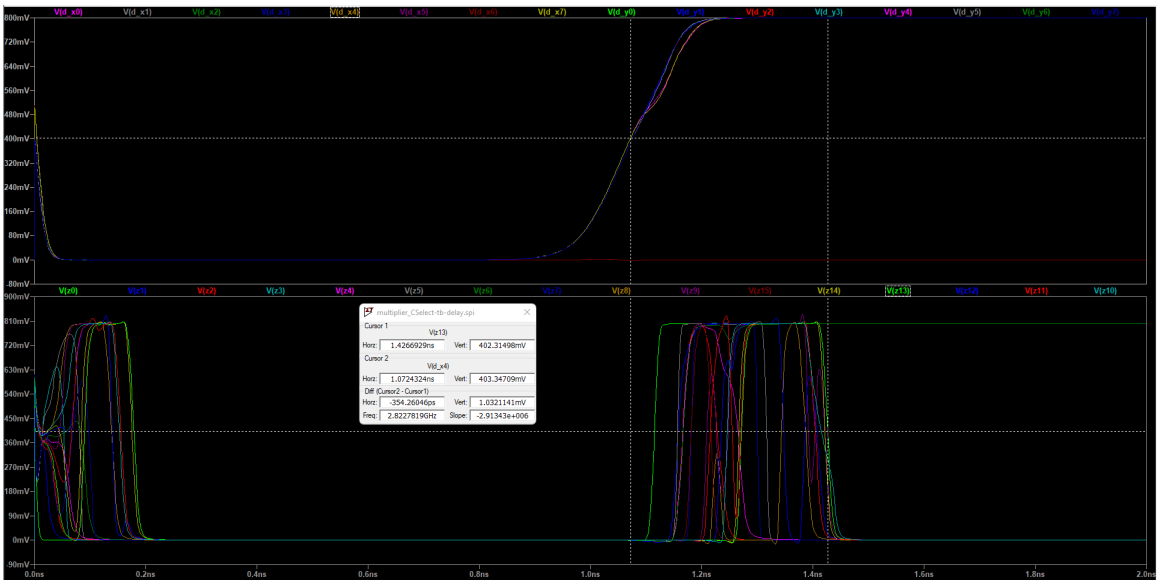


Figure 5.23: -75*-1: CS_A VM

5.8 Cascading 4-bit Carry Look-Ahead Adder and 4-bit Carry Select Adder

- In the Carry-Save multiplier, in both the above alternate VM implementations, we implement the 8-bit VM stage using two 4-bit adders.
- In the 4-bit CLA, the 4th bit is the critical bit and hence when they are cascaded, the net delay will be the direct sum of the delays of the individual 4-bit CLAs.
- For the first 4-bit adder in the VM stage, the 4-bit CLA will theoretically have a lower delay that the 4-bit CSA. But for subsequent stages, when we cascade the 4-bit adders, the additional delay added in the CSA will only be the delay of a mux and a xor gate.
- So optimally, choosing the first 4-bit adder in the VM as a CLA and the next stage as a CSA is expected to give min delay.

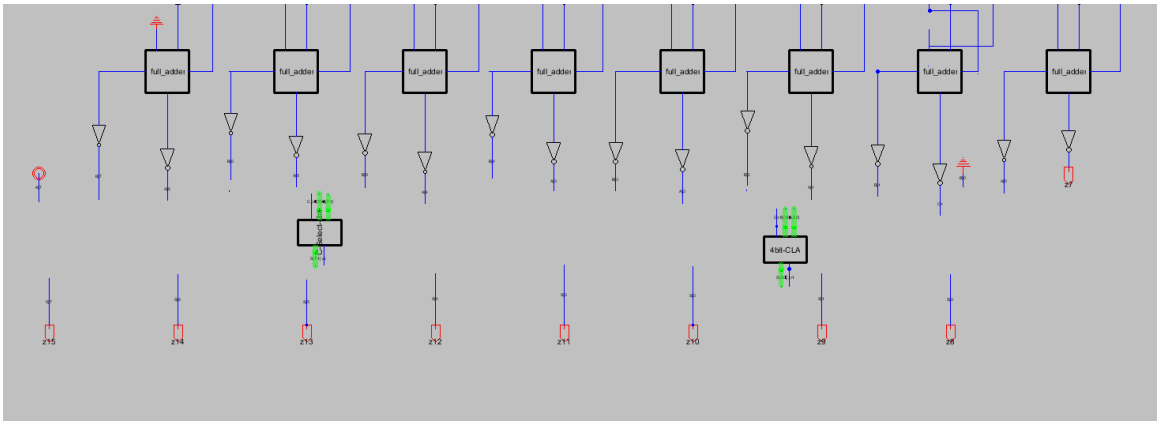


Figure 5.24: CLA-CS_A Cascaded VM Stage

To analyse the performance of this cascaded multiplier, we pass the same 3 inputs and analyse the combinational delay.

Case 1: **1 * 1**: 5.71% improvement

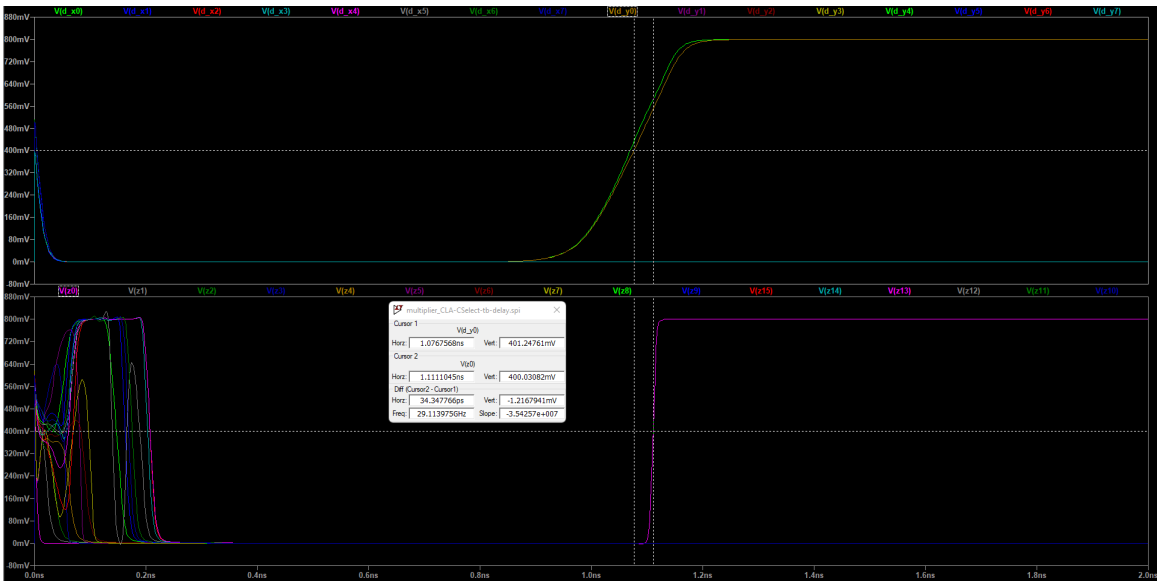


Figure 5.25: 1*1: CLA-CS_A Cascaded VM Stage

Case 2: **-11 * -11**: 14% improvement

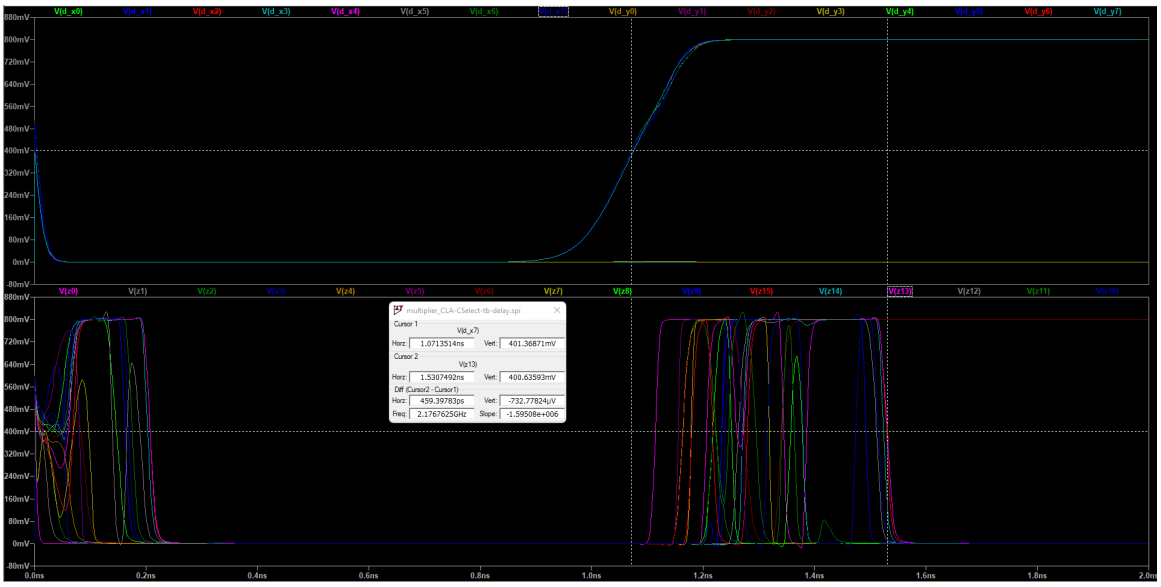


Figure 5.26: -11*-11: CLA-CS_A Cascaded VM Stage

Case 3: **-75 * -1**: 6.65% improvement

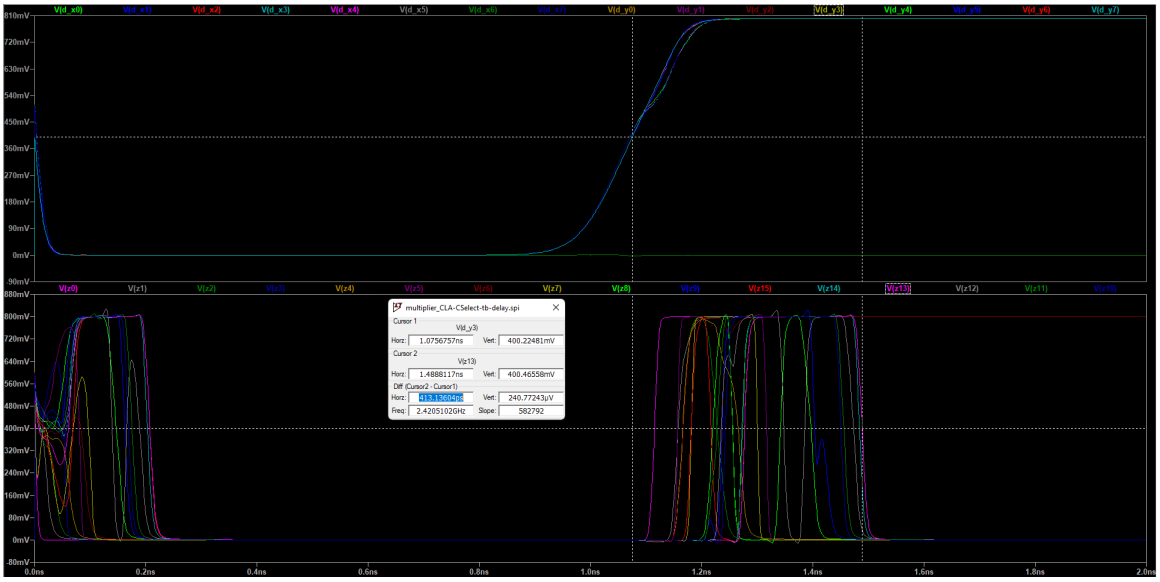


Figure 5.27: -75*-1: CLA-CS_A Cascaded VM Stage

Appendix

6.1 Python Code for spice input voltage sources definition

The following python code generates a text file with required voltage sources for user defined multiplicand and multiplier. It aligns the input voltage transitions at consecutive clock edges ensuring they are read correctly by the launch flops.

```
print("Enter 1st set of values \n")
x1 = int(input("Enter the Multiplicand value: "))
y1 = int(input("Enter the Multiplier Value: "))

if x1<-128 or x1>127 or y1<-128 or y1>127:
  raise ValueError("Please enter valid inputs for a 8 bit multiplier!!")

x1_2s = ("{0:0>8}").format(bin(x1 & int("1"*8, 2))[2:])
y1_2s = ("{0:0>8}").format(bin(y1 & int("1"*8, 2))[2:])

with open("A4_Param_Inputs_Pulse.txt",'w') as f:
  #for x input----
  for i in range(8):
    bit1 = "{VDD}" if x1_2s[i]==str(1) else "0"
    f.write("v"+str(i+2)+" x"+str(7-i)+" gnd PWL (0 0 {100p+2*Tc} 0 {200p+2*
   Tc} "+bit1+" 3n "+bit1+")"+"\n")
  for i in range(8):
    bit1 = "{VDD}" if y1_2s[i]==str(1) else "0"
    f.write("v"+str(i+10)+" y"+str(7-i)+" gnd PWL (0 0 {100p+2*Tc} 0 {200p
   +2*Tc} "+bit1+" 3n "+bit1+")"+"\n")
print("Text File Updated!!")
```

This python code takes 2 sets of input combinations from the user and aligns the input transitions along consecutive edges of the clock. This will demonstrate the multiplier processing 2 different input combinations in 1 .tran simulation run.

```
print("Enter 1st set of values \n")
x1 = int(input("Enter the Multiplicand value: "))
y1 = int(input("Enter the Multiplier Value: "))

print("\n Enter 2nd set of values \n")
x2 = int(input("Enter the Multiplicand value: "))
y2 = int(input("Enter the Multiplier Value: "))

if x1<-128 or x1>127 or y1<-128 or y1>127 or x2<-128 or x2>127 or y2<-128 or
    y2>127:
  raise ValueError("Please enter valid inputs for a 8 bit multiplier!!")

x1_2s = ("{0:0>8}").format(bin(x1 & int("1"*8, 2))[2:])
y1_2s = ("{0:0>8}").format(bin(y1 & int("1"*8, 2))[2:])

x2_2s = ("{0:0>8}").format(bin(x2 & int("1"*8, 2))[2:])
y2_2s = ("{0:0>8}").format(bin(y2 & int("1"*8, 2))[2:])
```

```
17
18
19 with open("A4_Param_Inputs_Pulse.txt",'w') as f:
20   #for x input----
21   for i in range(8):
22     bit1 = "{VDD}" if x1_2s[i]==str(1) else "0"
23     bit2 = "{VDD}" if x2_2s[i]==str(1) else "0"
24     f.write("v"+str(i+2)+" x"+str(7-i)+" gnd PWL (0 0 {100p+2*Tc} 0 {200p+2*
     Tc} "+bit1+" {100p+3*Tc} "+bit1+" {200p+3*Tc} "+bit2+" 3n "+bit2+")"+"\n"
     )
25   for i in range(8):
26     bit1 = "{VDD}" if y1_2s[i]==str(1) else "0"
27     bit2 = "{VDD}" if y2_2s[i]==str(1) else "0"
28     f.write("v"+str(i+10)+" y"+str(7-i)+" gnd PWL (0 0 {100p+2*Tc} 0 {200p
     +2*Tc} "+bit1+" {100p+3*Tc} "+bit1+" {200p+3*Tc} "+bit2+" 3n "+bit2+")"+"
     \n")
29 print("Text File Updated!!")
```

6.2 Sample SPICE code from testbench for reading output voltages

This spice code reads 15 different voltage values (z0 - z15) at a specifc instant of time and converts the voltages in 2's complement notation into decimal format and display it in the spice error log,

```
* Spice Code nodes in cell cell 'multiplier-tb{sch}'
*.include "C:\Users\Hemanth Ram\Desktop\sem7\ee5311\22nm_HP.pm"
*.include "C:\Users\welcome\Desktop\DIC\22nm_HP.pm"
.include "H:\Acads\Sem7\EE5311_Digital_IC_Design\DIC-Electric\22nm_HP.pm"
*Schematic
.params vdd=0.8 Tc=0.56n Tr=100p
*Netlist
*.params vdd=0.8 Tc=0.91n Tr=100p
v1 vdd gnd DC {vdd}
.include "A4_Param_Inputs_Pulse.txt"
v18 clock gnd PULSE ({vdd} 0 0.2n {Tr} {Tr} {Tc/2-Tr} {Tc})
.tran 5n uic
.meas tran x find (round(V(q_x7))*(-128) + round(V(q_x6))*64 + round(V(q_x5))*32 + round(V(q_x4))*16 +
    + round(V(q_x3))*8 + round(V(q_x2))*4 + round(V(q_x1))*2 + round(V(q_x0))*1) when v(clk)={0.5*vdd} cross=2 td={2*Tc+0.45n}
.meas tran y find (round(V(q_y7))*(-128) + round(V(q_y6))*64 + round(V(q_y5))*32 + round(V(q_y4))*16 +
    + round(V(q_y3))*8 + round(V(q_y2))*4 + round(V(q_y1))*2 + round(V(q_y0))*1) when v(clk)={0.5*vdd} cross=2 td={2*Tc+0.45n}
.meas tran z find (round(V(z15))*(-32768) + round(V(z14))*16384 + round(V(z13))*8192 + round(V(z12))*4096 +
    + round(V(z11))*2048 + round(V(z10))*1024 + round(V(z9))*512 + round(V(z8))*256 +
    + round(V(z7))*128 + round(V(z6))*64 + round(V(z5))*32 + round(V(z4))*16 +
    + round(V(z3))*8 + round(V(z2))*4 + round(V(z1))*2 + round(V(z0))*1) when v(clk)={0.5*vdd} cross=4 td={2*Tc+0.45n}
.option noopiter
.END
.END
```

Figure 6.1: Multiplier testbench spice code