16 Bit Carry Skip Ahead Adder and 5 Bit Multiplier Simulation

EE 577 A

Lab 1 Part 2 Assignment Report

Submitted by –

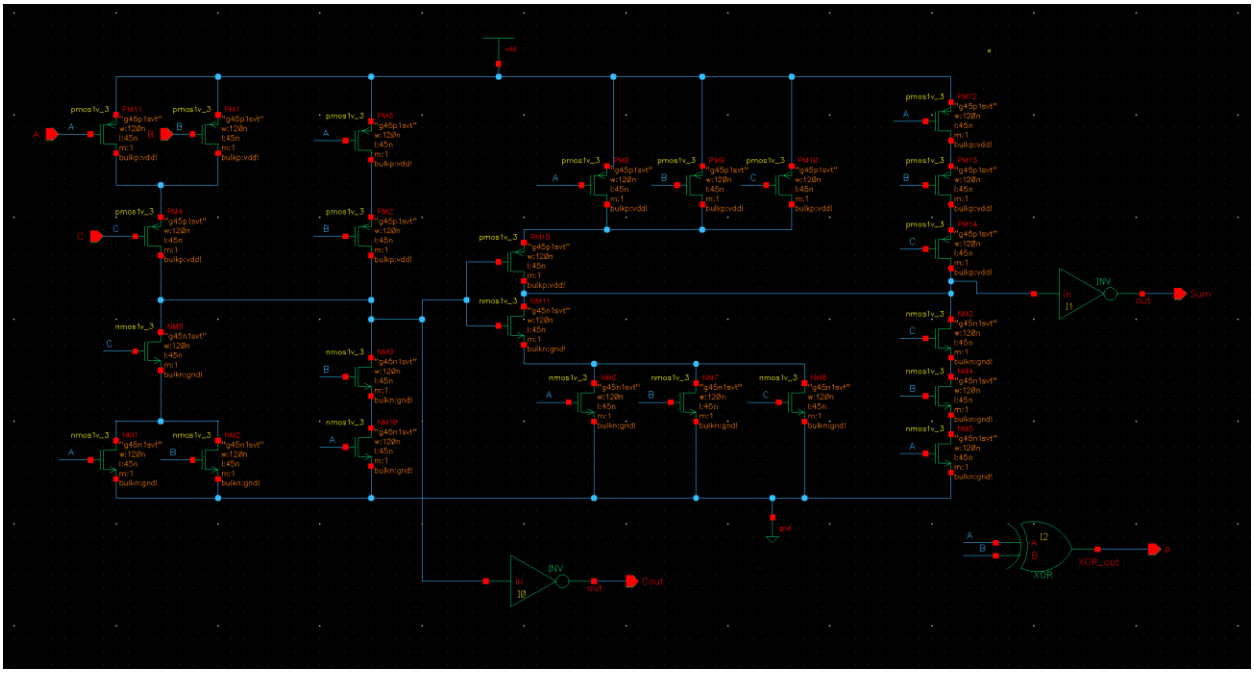
Rohit Kumar Singh

6689387841

Content:

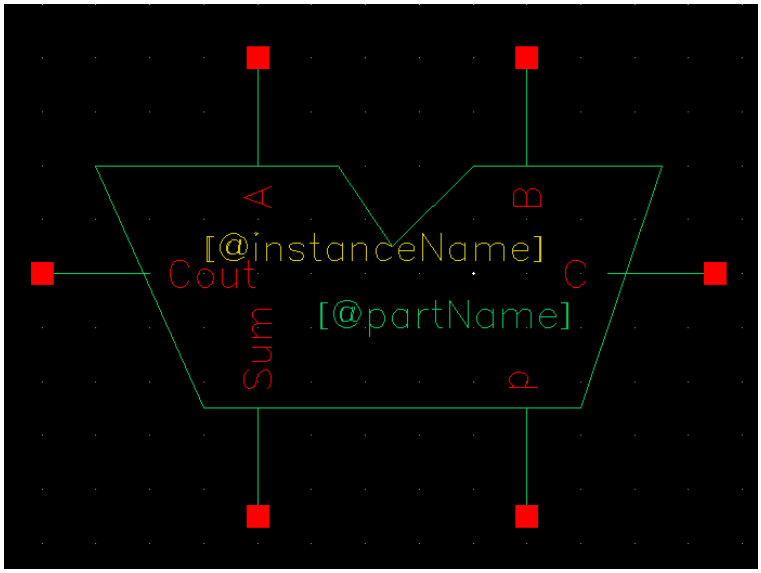
1-Bit Full Adder –

The 1-bit Full Adder is the fundamental building block of our 16 Bit Full Adder. We have constructed it using Duality.



Fiq 1 : 1- Bit Full Adder Schematic

Below is the symbol of our 1 bit Full Adder, we will use it to instantiate in our top level design in 16 bit Half adder and Multiplier.



Fiq 2: 1- Bit Full Adder Symbol

Functional Verification:

```
radix 1 1 1
io i i i
vname A B C
tunit ns
slope 0.005
vih 1.0
vil 0.0
trise 0.005
tfall 0.005


0   0 0 0
2   0 0 1
4   0 1 0
6   0 1 1
8   1 0 0
10  1 0 1
11  1 1 0
12  1 1 1
```

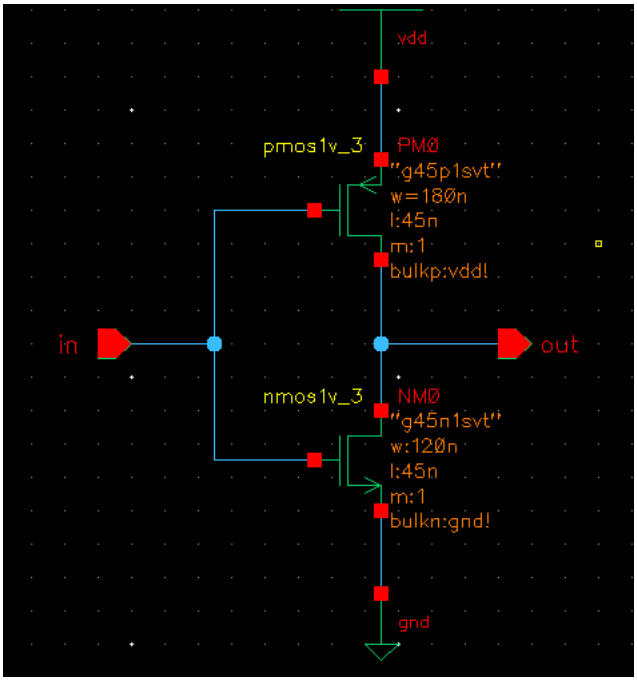
Fiq 3 : 1- Bit Full Adder Vector File

We are using python to generate the vector file. The vector file is inserted in our Specter manually by the user. After executing the vector file the output waveform is verified manually by the user with the golden outputs generated by the python script.

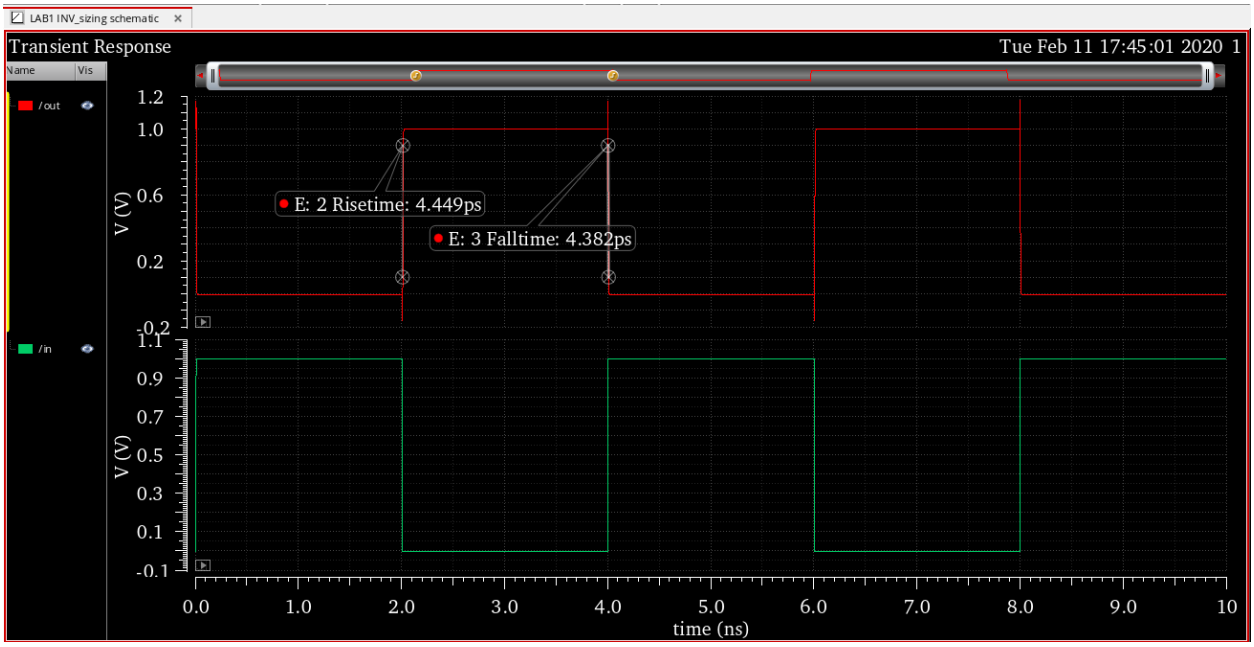
Fiq 4 : 1- Bit Full Adder Output File

Inverter Sizing –

In order to know the Beta ratio of the inverter, we sized our pmos, which enables the rise time and the fall time of our Output signal to be approximately equal.



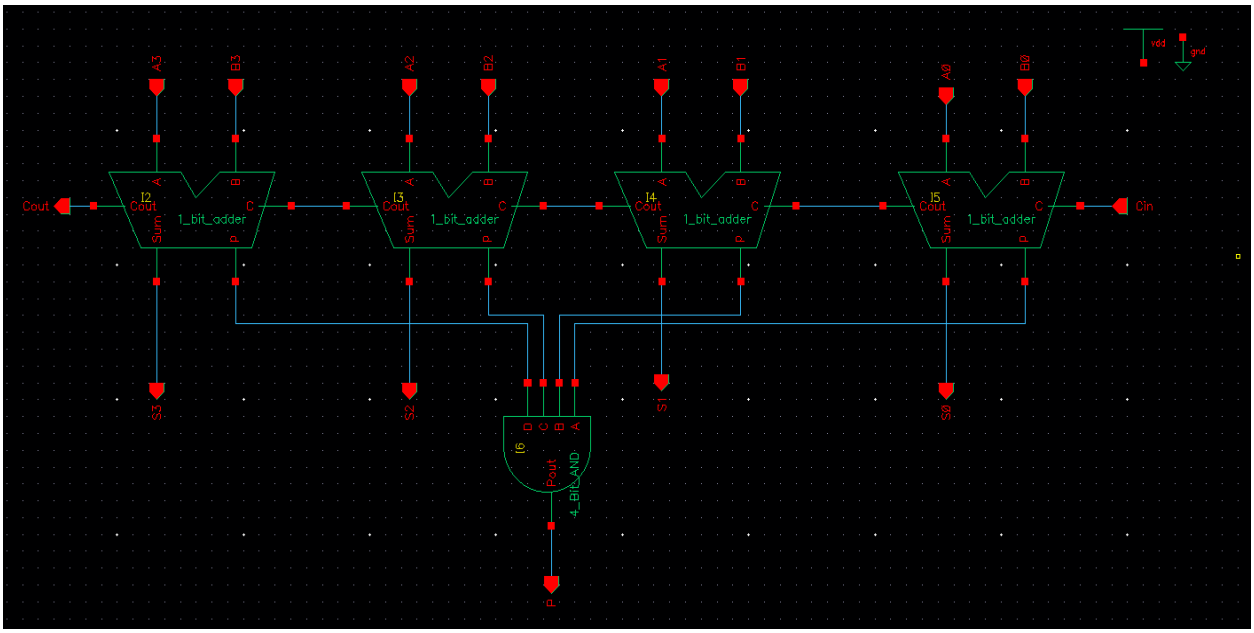
Fiq 1 : Sized Inverter Schematic



Fiq 1 : Sized Inverter Waveform

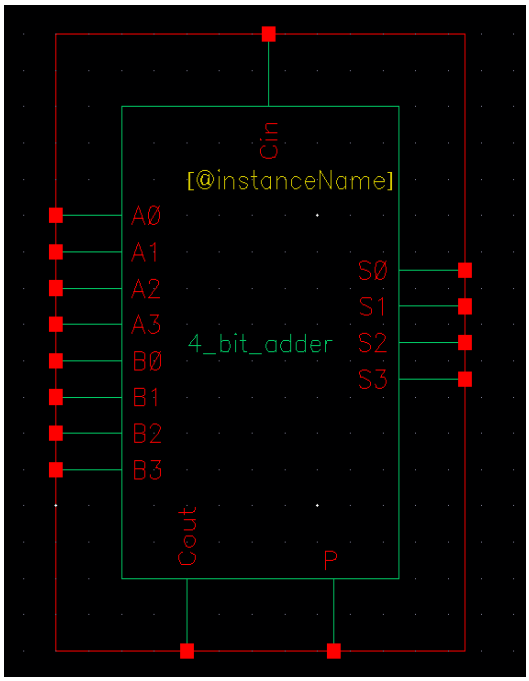
Hence according to the formula, the Beta ration is proportional to the ration of the width of pmos to that of nmos. The Gain ratio for the above inverter = $\beta_r = \frac{W_p}{W_n} = \frac{180}{120} = 1.5$

4-Bit Full Adder –



Fiq 1 : 4- Bit Full Adder Schematic

Below is the symbol of our 4-bit Full Adder, we will use it to instantiate in our top level design in 16 bit Half adder and Multiplier.



Fiq 2: 4- Bit Full Adder Symbol

Functional Verification:

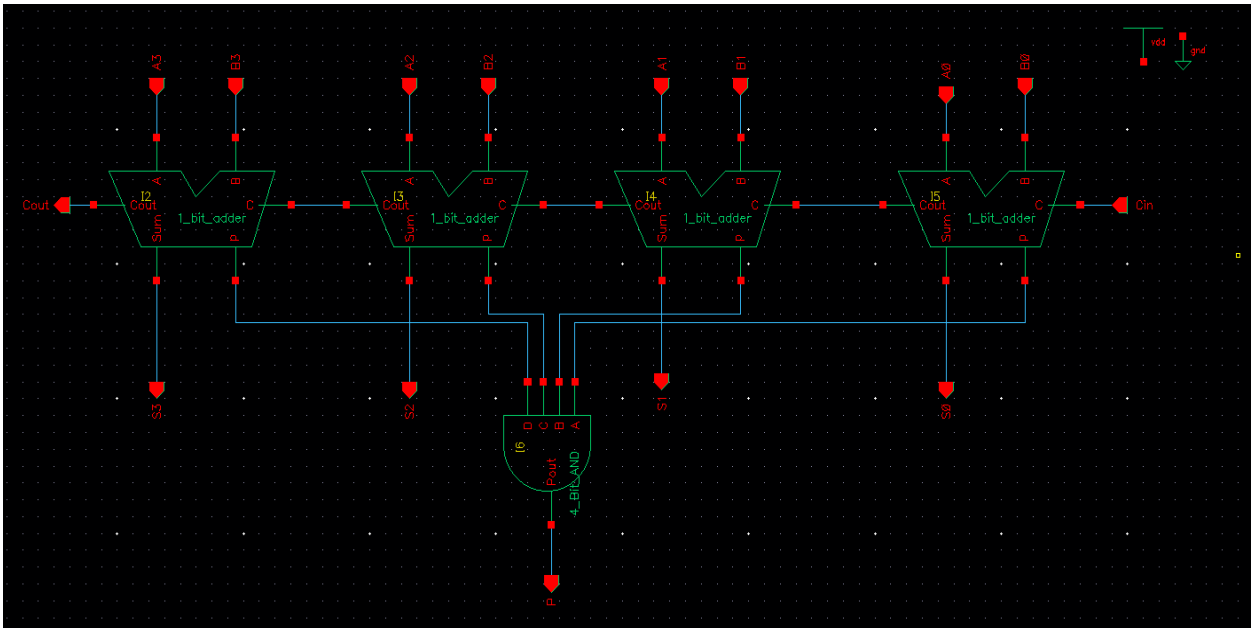
```
radix 4 4 1
io i i i
vname A<[3:0]> B<[3:0]> Cin
tunit ns
slope 0.005
vih 1.0
vil 0.0
trise 0.005
tfall 0.005

0  0 0 0
2  0 0 1
4  0 1 0
6  0 1 1
8  1 0 0
10 1 0 1
11 1 1 0
12 1 1 1
```

Fiq 3 : 4- Bit Full Adder Vector File

16-Bit Full Adder –

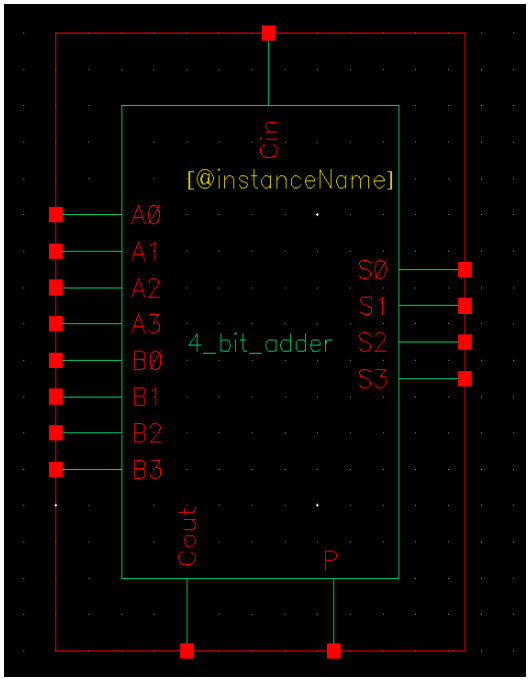
4-Bit Full Adder –

By utilizing four 1-bit Full Adder, we are constructing a 4-bit Full Adder. In the 4 bit Full Adder, we have introduced a propagate bit, which checks if the carry is being propagated from the input to output.



Fiq 1 : 4- Bit Full Adder Schematic

Below is the symbol of our 4-bit Full Adder, we will use it to instantiate in our top level design in 16 bit Half adder and Multiplier.



Fiq 2: 4- Bit Full Adder Symbol

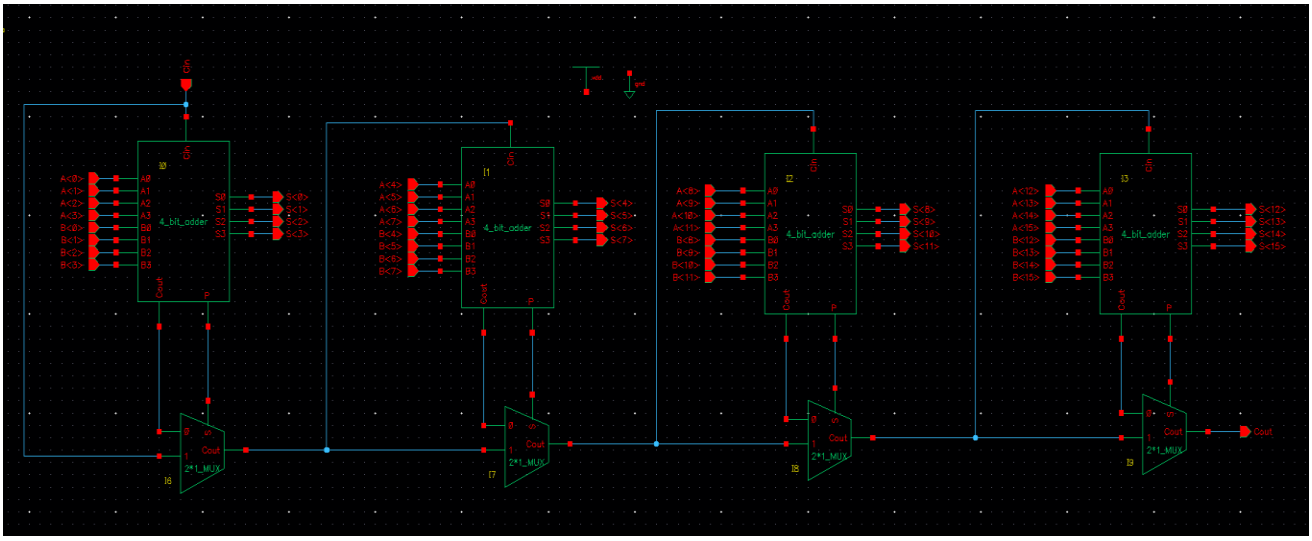
Functional Verification:

```
radix 4 4 1
io i i i
vname A<[3:0]> B<[3:0]> Cin
tunit ns
slope 0.005
vih 1.0
vil 0.0
trise 0.005
tfall 0.005

0   0 0 0
2   0 0 1
4   0 1 0
6   0 1 1
8   1 0 0
10  1 0 1
11  1 1 0
12  1 1 1
```

Fiq 3 : 4- Bit Full Adder Vector File

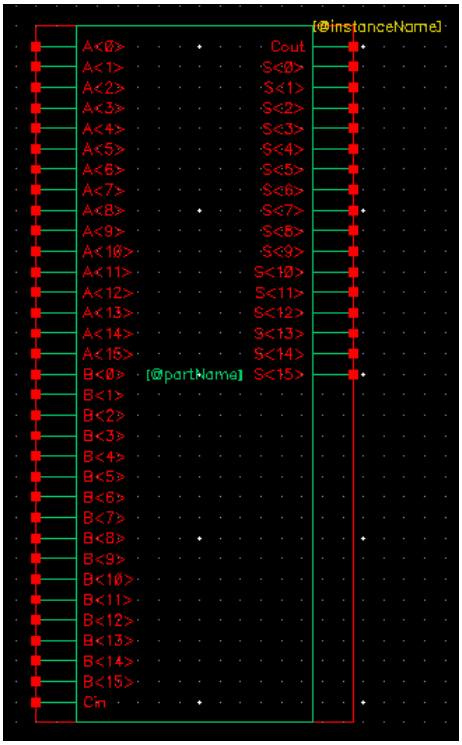
16-Bit Full Adder –

By utilizing four 4-bit Full Adders, we are constructing a 16-bit Full Adder. In the 16-bit Full Adder, we have introduced a multiplexer, which propagates the carry to the next adder module.



Fiq 1 : 16- Bit Full Adder Schematic

Below is the symbol of our 1 bit Full Adder, we will use it to instantiate in our top level design in 16 bit Half adder and Multiplier.



Fiq 2: 16- Bit Full Adder Symbol

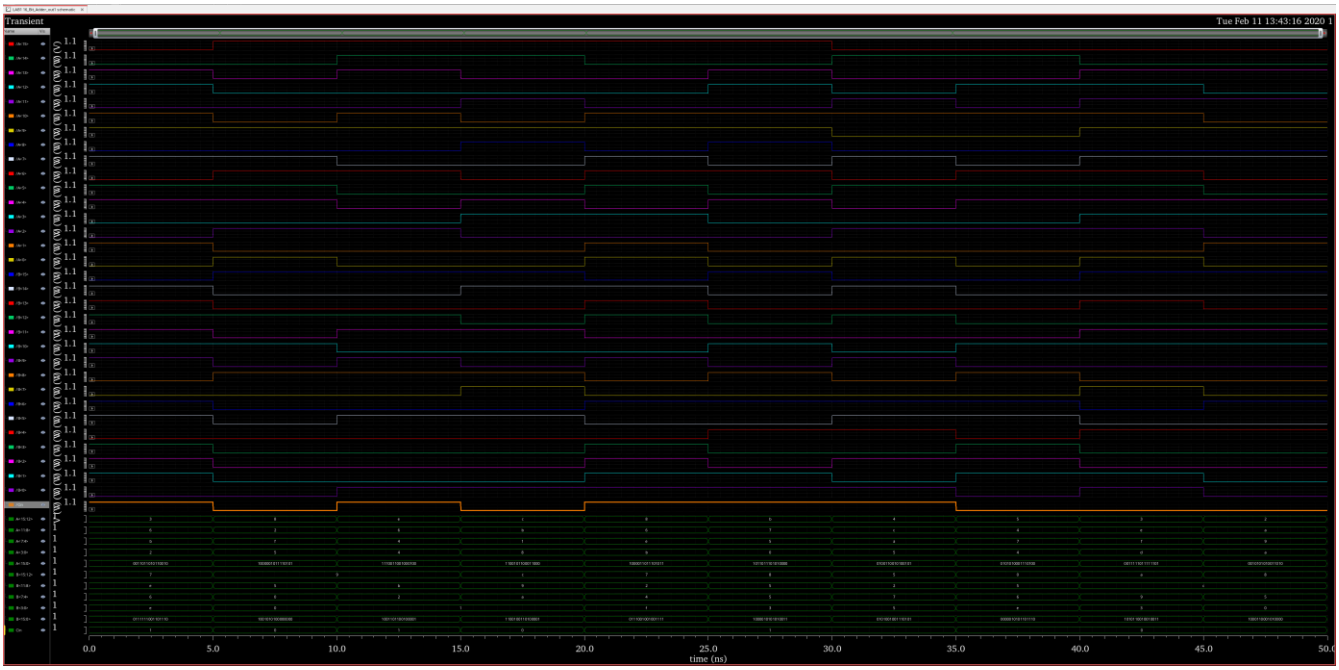
Functional Verification:

```
radix 4 4 4 4 4 4 4 4 1
io i i i i i i i i i
vname A<[3:0]> A<[7:4]> A<[11:8]> A<[15:12]> B<[3:0]> B<[7:4]>
B<[11:8]> B<[15:12]> Cin
tunit ns
slope 0.005
vih 1.0
vil 0.0

0 2 B 6 3 E 6 E 7 1
5 5 F 2 8 0 0 5 9 0
10 4 4 6 E 1 2 B 9 1
15 8 1 B C 1 A 9 C 0
20 B E 6 8 F 4 2 7 1
25 0 5 7 B 3 5 5 8 1
30 5 A C 4 5 7 2 5 1
35 4 7 4 5 E 6 5 0 0
40 D F E 3 3 9 C A 0
45 A 9 A 2 0 5 C 8 0
```

Fiq 3 : 16- Bit Full Adder Vector File

We are using python to generate the vector file. The vector file is inserted in our Specter manually by the user. After executing the vector file, the output waveform is verified manually by the user with the golden outputs generated by the python script

vector_file_Add_final.py



Fiq 4 : 16- Bit Full Adder Output File

Below is the golden output file, which is generated by Python Scripting and is utilized to verify the output waveform

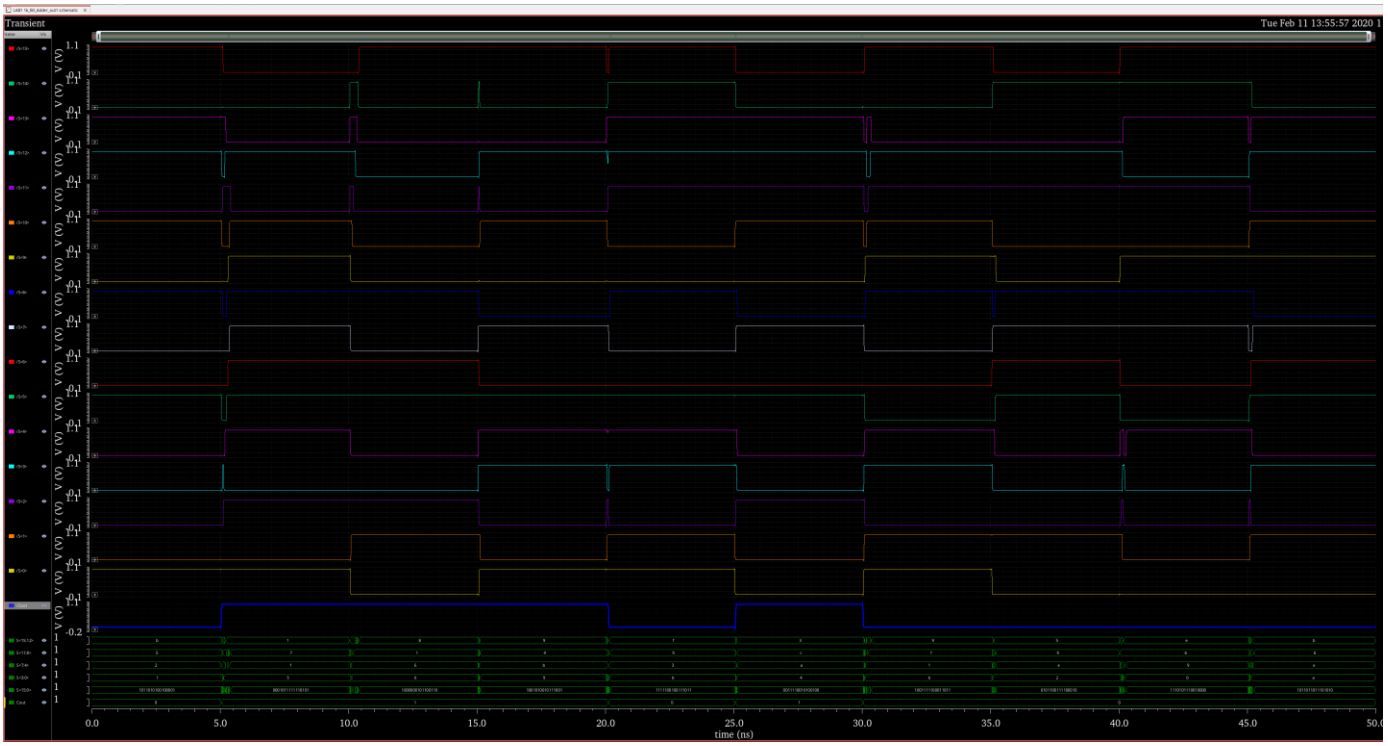
Golden Output File

A(DEC)	B(DEC)	Cin(DEC)	SUM(DEC)	A(BIN)	B(BIN)	Cin(BIN)	SUM(BIN)
14002	32366	1	46369	0011011010110010	0111111001101110	1	01011010100100001
33525	38144	0	71669	1000001011110101	1001010100000000	0	10001011111110101
58948	39713	1	98662	1110011001000100	1001101100100001	1	11000000101100110
51992	51617	0	103609	1100101100011000	1100100110100001	0	11001010010111001
34539	29263	1	63803	1000011011101011	0111001001001111	1	01111100100111011
46928	34131	1	81060	1011011101010000	1000010101010011	1	10011110010100100
19621	21109	1	40731	0100110010100101	0101001001110101	1	01001111100011011
21620	1390	0	23010	0101010001110100	0000010101101110	0	00101100111100010
16125	44179	0	60304	0011111011111101	1010110010010011	0	01110101110010000
10906	35920	0	46826	0010101010011010	1000110001010000	0	01011011011101010

Fiq 5 : 16- Bit Full Adder Golden Output Fil

Add_golden.txt

vector_file_Add_final.py



Fiq 6 : 16- Bit Full Adder Output File

16_bit_input.png

16_bit_output.png

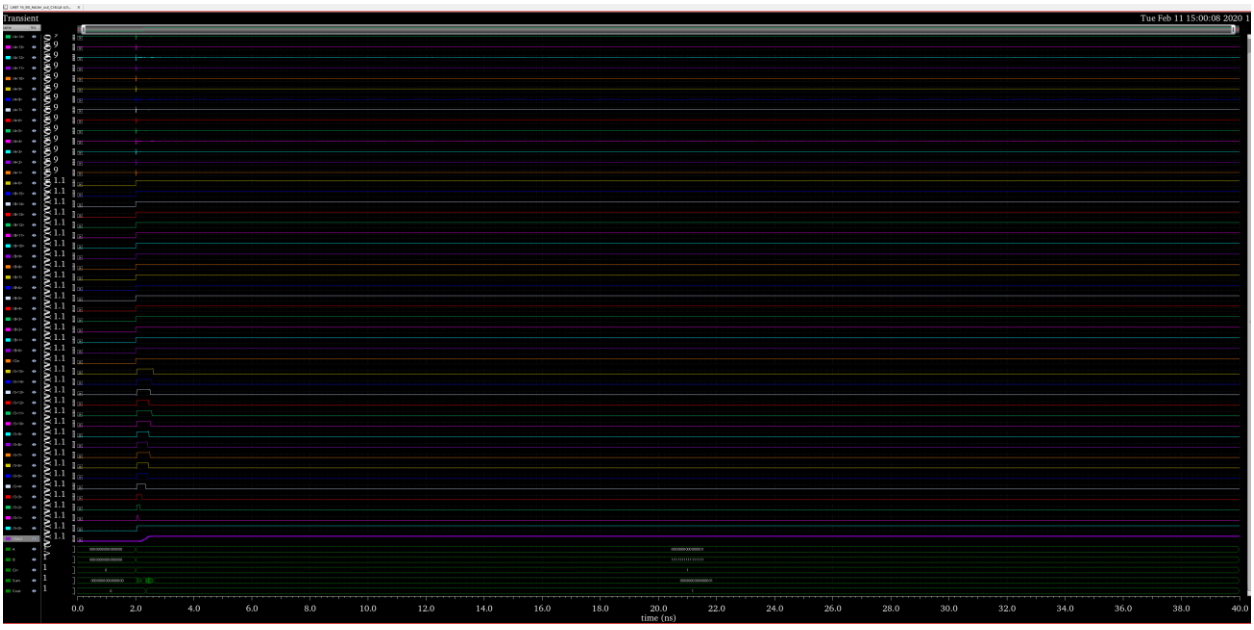
The Output were successfully verified and reviewed using the golden file and the output waveform generated.

We will investigate further to find the worst path - critical path.

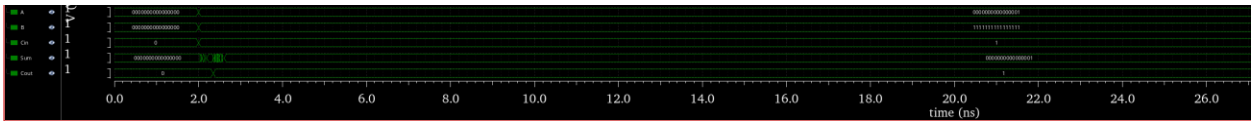
The critical path is the most resistive path, it is also the path having the maximum delay path. We considered all the inputs and finally concluded the critical path of our 16 bit Adder. There are two possibilities of worst case delays, first when the inputs are FFFF and FFFF, whereas the other one is 0001 and FFFF, it's noted in the waveform that the delay in inputs of 0001 and FFFF there is greater delay compared to FFFF to FFFF.

```
radix 4 4 4 4 4 4 4 4 1
io i i i
vname A<[15:12]> A<[11:8]> A<[7:4]> A<[3:0]> B<[15:12]> B<[11:8]> B<[7:4]> B<[3:0]> Cin
tunit ns
slope 0.005
vih 1.0
vil 0.0
trise 0.005
tfall 0.005


0  0 0 0 0 0 0 0 0 0
2  0 0 0 1 F F F F 1
```

Fiq 7 : 16- Bit Full Adder Critical Path Vector File



Fiq 8 : 16- Bit Full Adder Critical Path Output Waveform



Fiq 9 : 16- Bit Full Adder Critical Path Waveform

Result and Summary –

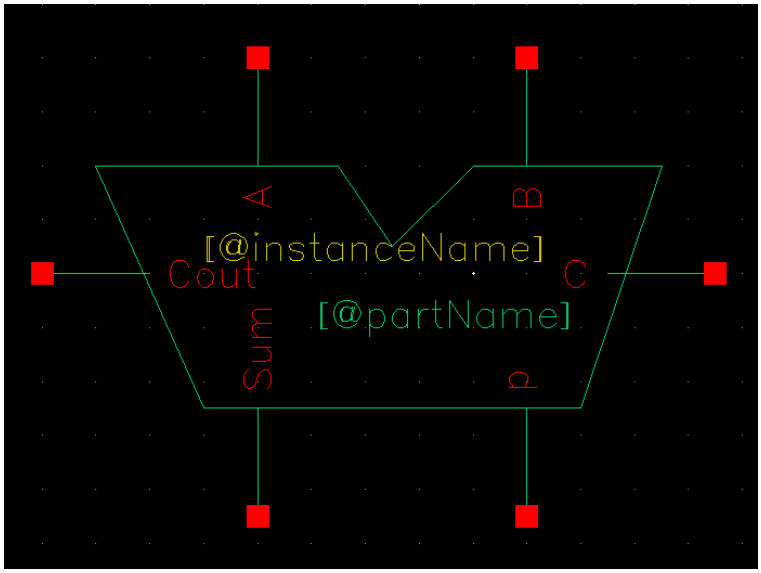
Hence the 16 bit Carry Skip Ahead Adder was designed and the proper golden files were generated using python scripting. The implementation of the vector files was executed manually and the verification of the results, showed positive correlation between the golden files and the generated output waveforms. We

calculated the Beta ration of our inverter and found out the critical path. Hence the Design and Verification of the 16_Bit Carry Skip Ahead Adder is successful.

5- Bit Multiplier:

1-Bit Full-Adder –

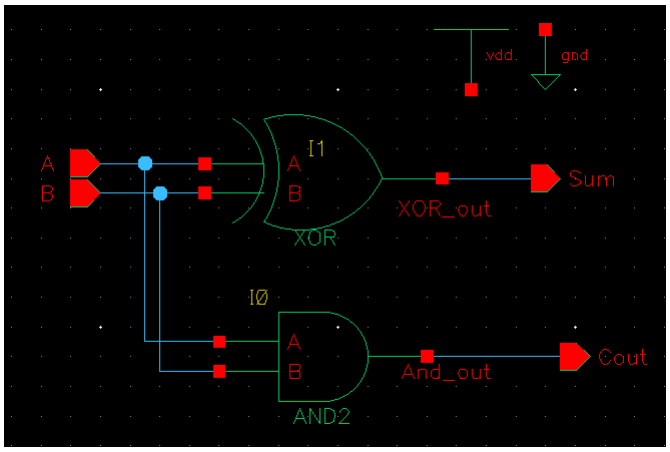
Below is the symbol of our 1 bit Full Adder, we will use it to instantiate in our top level design in 5 bit Multiplier.



Fiq 1 : 1- Bit Full Adder Schematic

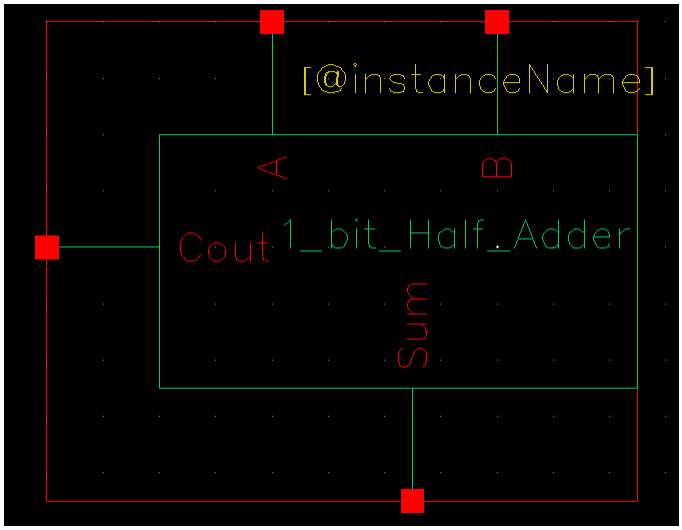
1-Bit Half Adder –

By utilizing a Xor and And gate we are forming a four 1-bit Full Adder, we are generating sum and carry out from the Xor and And Gates.



Fiq 2: 4- Bit Full Adder Symbol

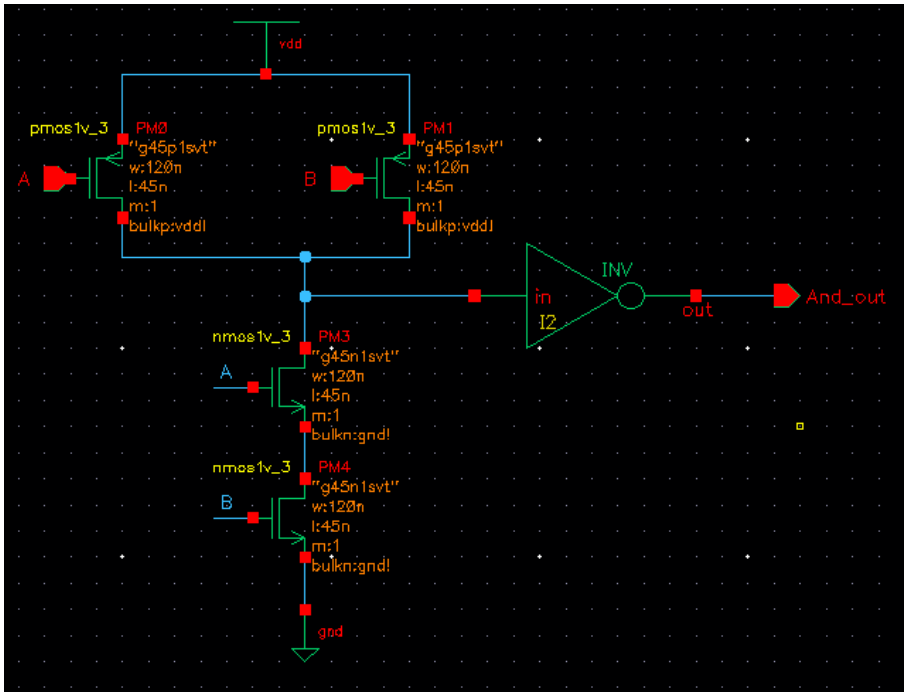
Below is the symbol of our 1 bit Half Adder, we will use it to instantiate in our top level design in 5 bit Multiplier.



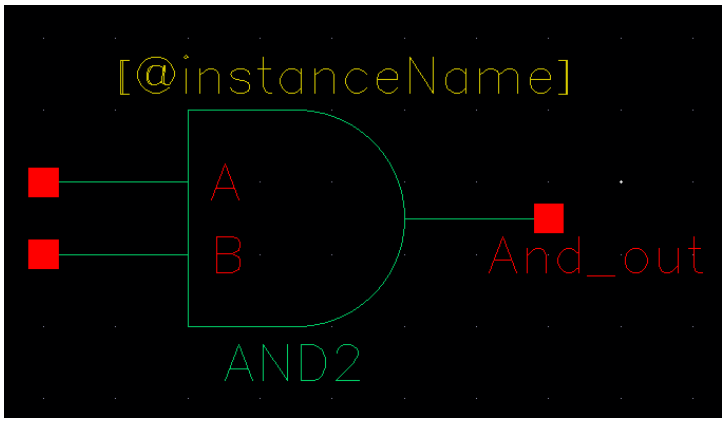
Fiq 3 : 4- Bit Full Adder Schematic

Basic Gates –

We are using pmos and nmos transistors to create an AND gate, we are using an inverter which we created on our own using a CMOS inverter circuit.

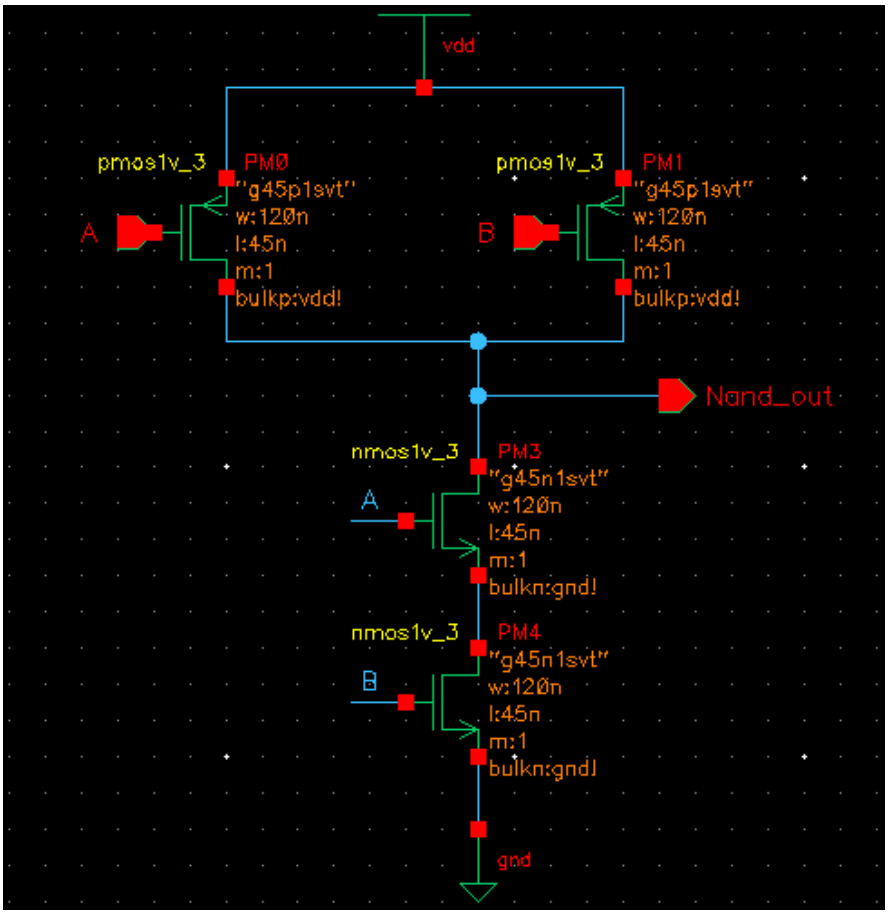


Fiq 4 : AND Gate

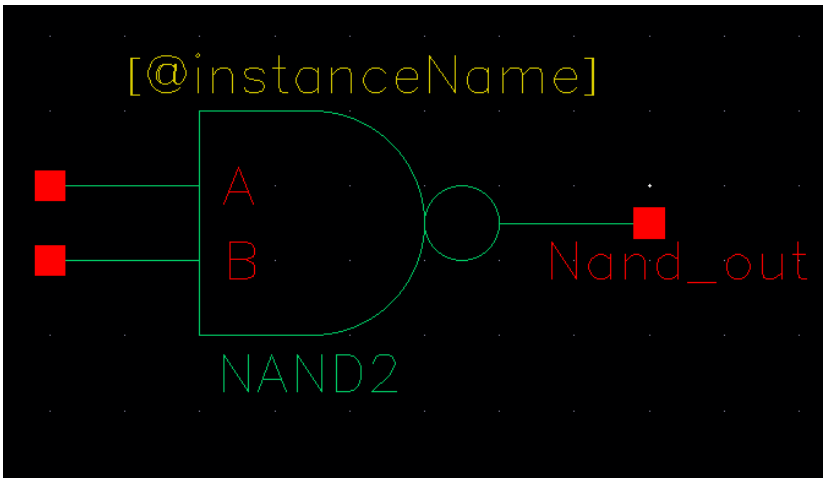


Fiq 5 : AND Gate Symbol

We are using pmos and nmos transistors to create an NAND gate.

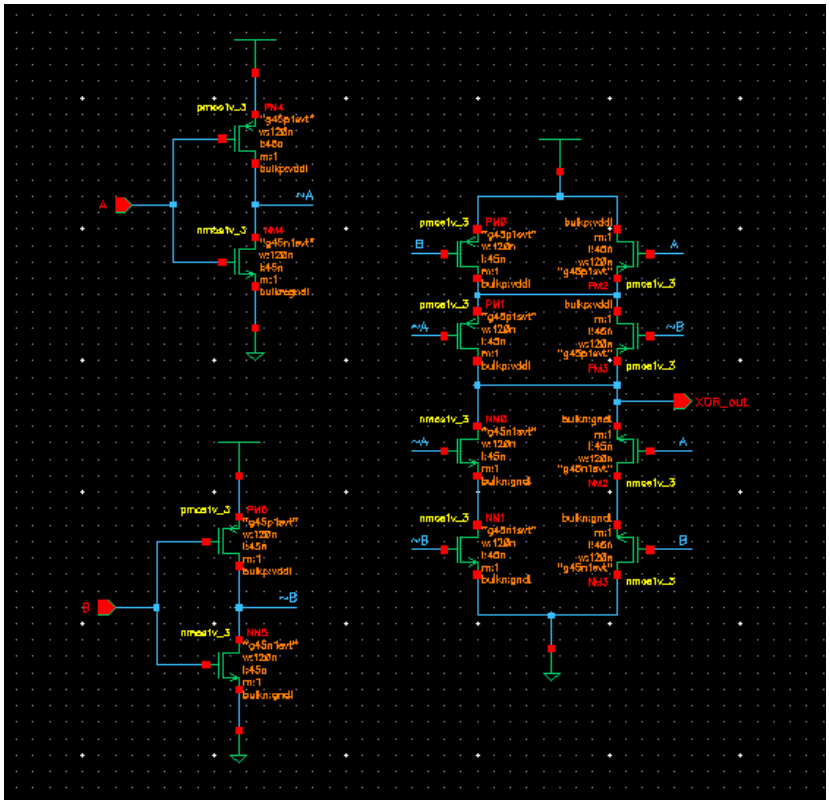


Fiq 6 : NAND Gate

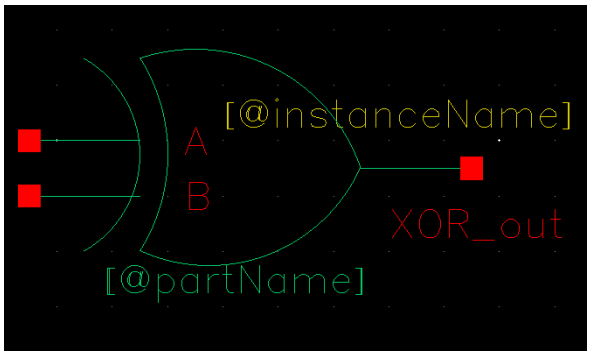


Fiq 7 : NAND Gate Symbol

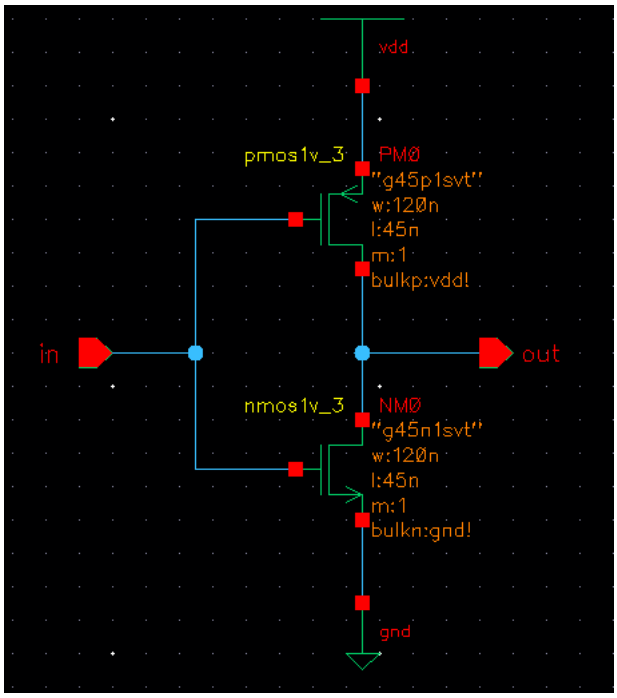
We are using pmos and nmos transistors to create an XOR gate.



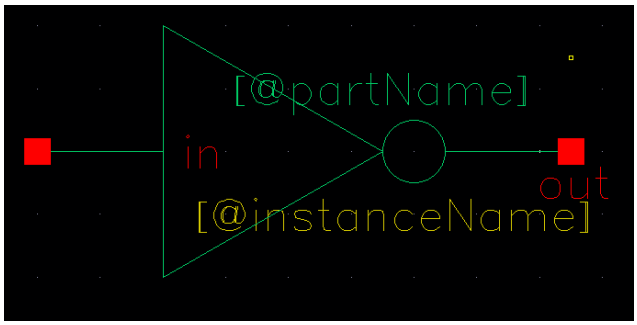
Fiq 8 : NAND Gate Symbol



Fiq 9 : NAND Gate Symbol

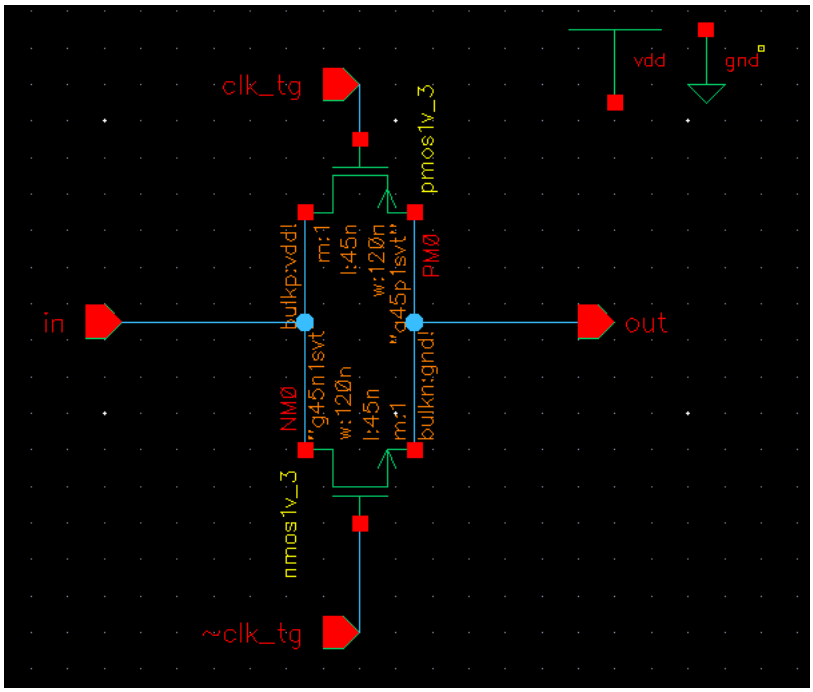


Fiq 10 : Inverter Gate Schematic

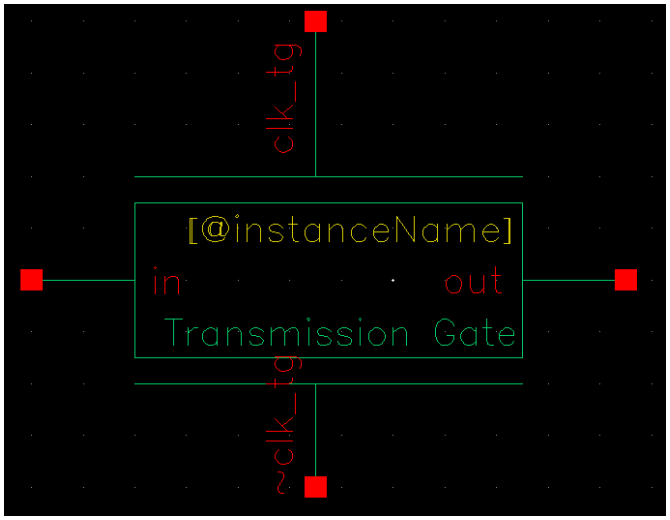


Fiq 11 : Inverter Gate Symbol

D Flip Flop –

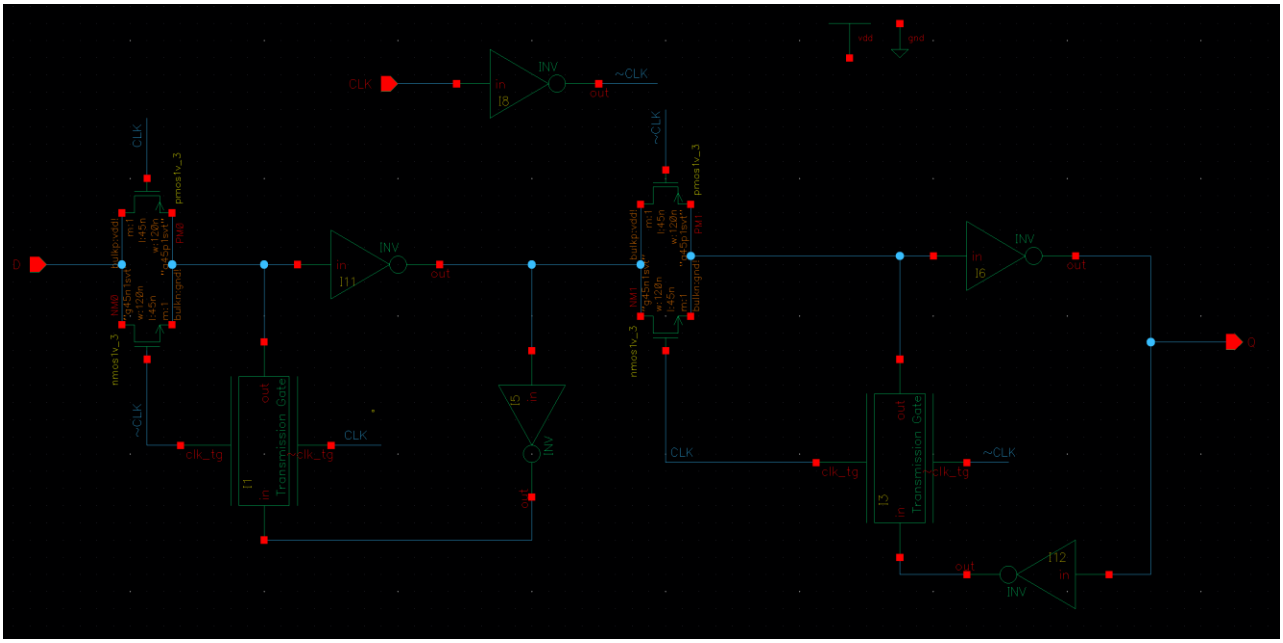


Fiq 10 : Transmission Gate

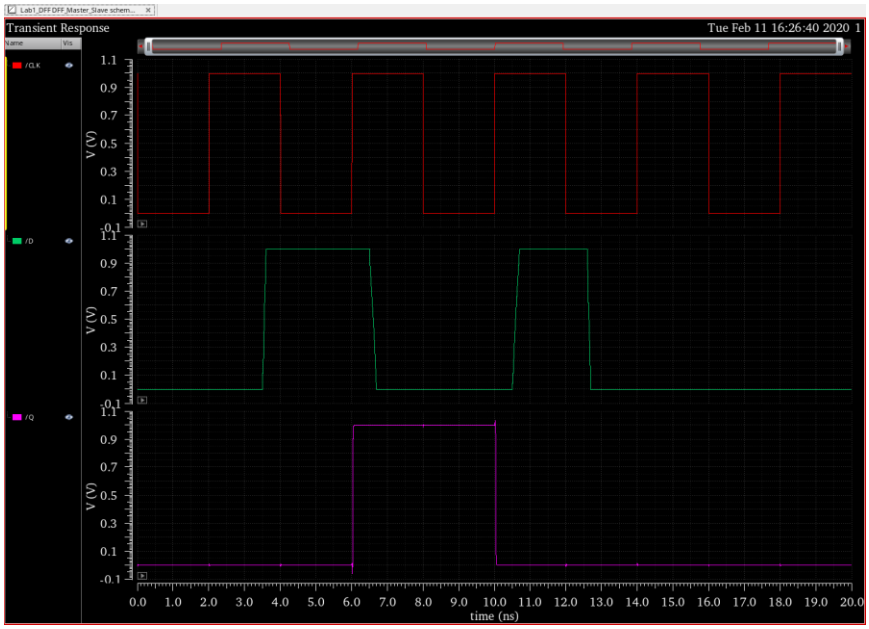


Fiq 11 : Transmission Gate Symbol

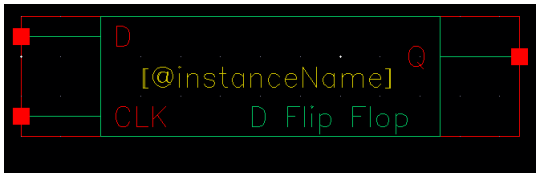
Instantiating the Inverter and the transmission gate, we are creating the Master Slave D Flip Flop, to test the functionality of the D Flip Flop, I am providing a PWL Signal as an input just before the set-up time.



Fiq 12 : D Flip Flop Schematic

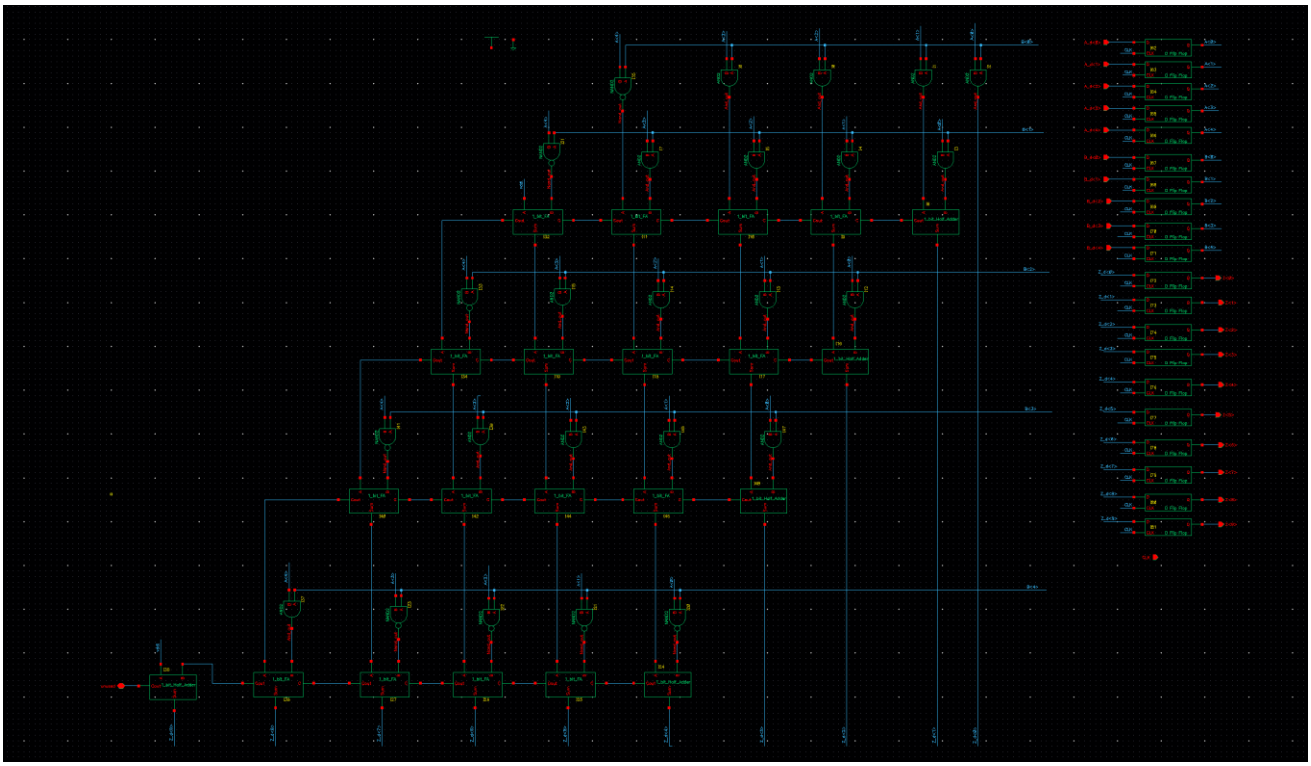


Fiq 13 : D Flip Flop Waveform



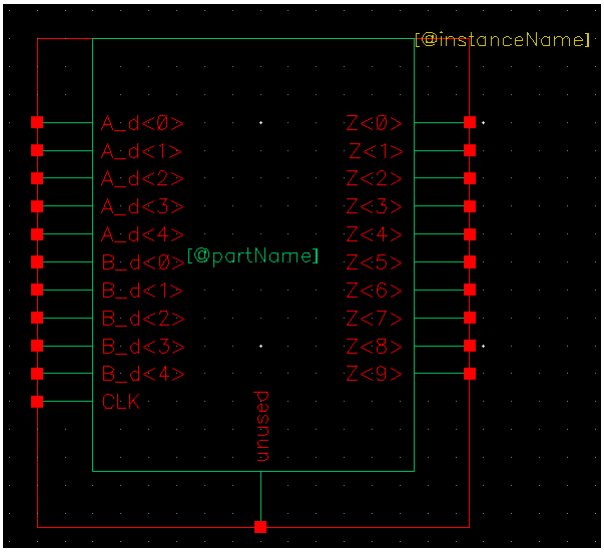
Fiq 14 : D Flip Flop Symbol

Multiplier -



Fiq 15 : Multiplier Schematic with D Flip Flop

As instructed we have used multiple D Flip Flops, for every input we are using one D Flip Flop and for every output we are using one D Flip Flop. The Flip Flop syncs the input and outputs with the clock.



Fiq 16 : Multiplier Schematic with D Flip Flop Symbol

Functional Verification:

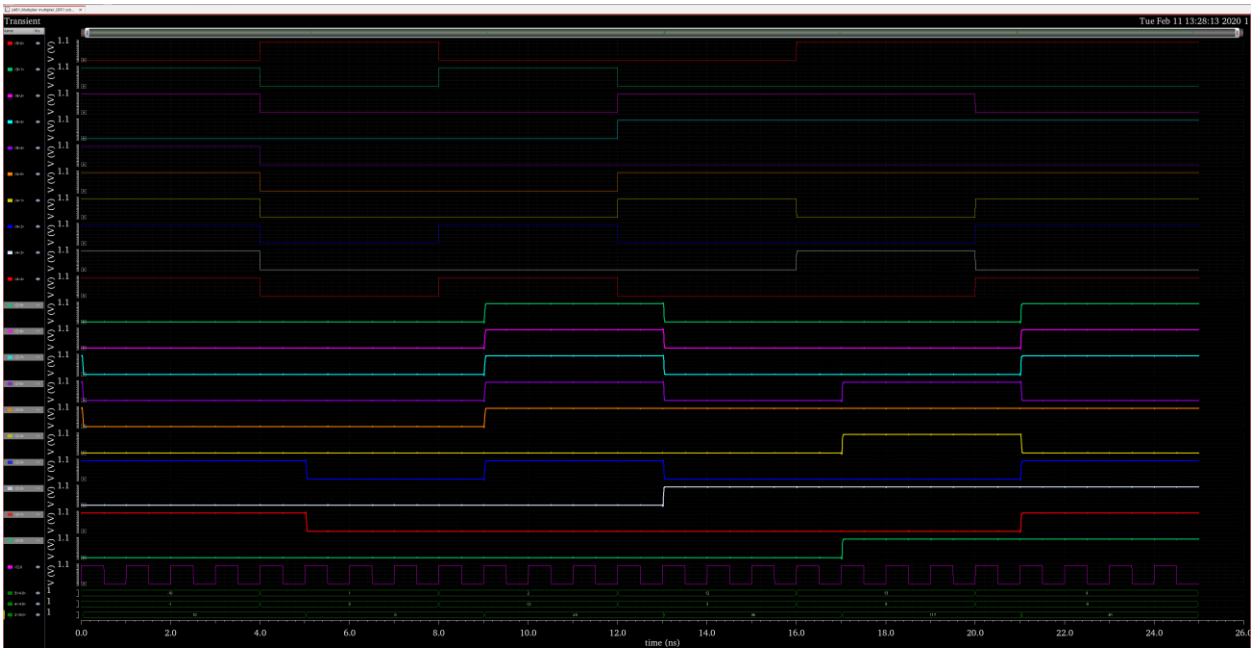
```
radix 1 4 1 4
io i i i i
vname B<4> B<[3:0]> A<4> A<[3:0]>
tunit ns
slope 0.005
vih 1.0
vil 0.0
trise 0.005
tfall 0.005

0  1 6 1 F
4  0 1 0 0
8  0 2 1 4
12 0 C 0 3
16 0 D 0 9
20 0 9 1 7
```

Fiq 17 : Multiplier Schematic with D Flip Flop Vector File

vector_file_Mul_final.py

Vector_mul_final.vec

We are using python to generate the vector file. The vector file is inserted in our Specter manually by the user. After executing the vector file, the output waveform is verified manually by the user with the golden outputs generated by the python script



Fiq 18 : Multiplier Output Waveform

Golden Output :

A(DEC)	B(DEC)	PROD(DEC)	A(BIN)	B(BIN)	PROD(BIN)
-10	-1	10	10110	11111	0000001010
1	0	0	00001	00000	0000000000
2	-12	-24	00010	10100	1111101000
12	3	36	01100	00011	0000100100
13	9	117	01101	01001	0001110101
9	-9	-81	01001	10111	1110101111

Fiq 18 : Multiplier Output Golden File

Result and Summary –

Hence the 5 bit Multiplier was designed and the proper golden files were generated using python scripting. The implementation of the vector files was executed manually and the verification of the results, showed positive correlation between the golden files and the generated output waveforms. Hence the Design and Verification of the 5 Bit Multiplier is successful.