THE FLOOR PLAN

Block Level Logical Diagram:

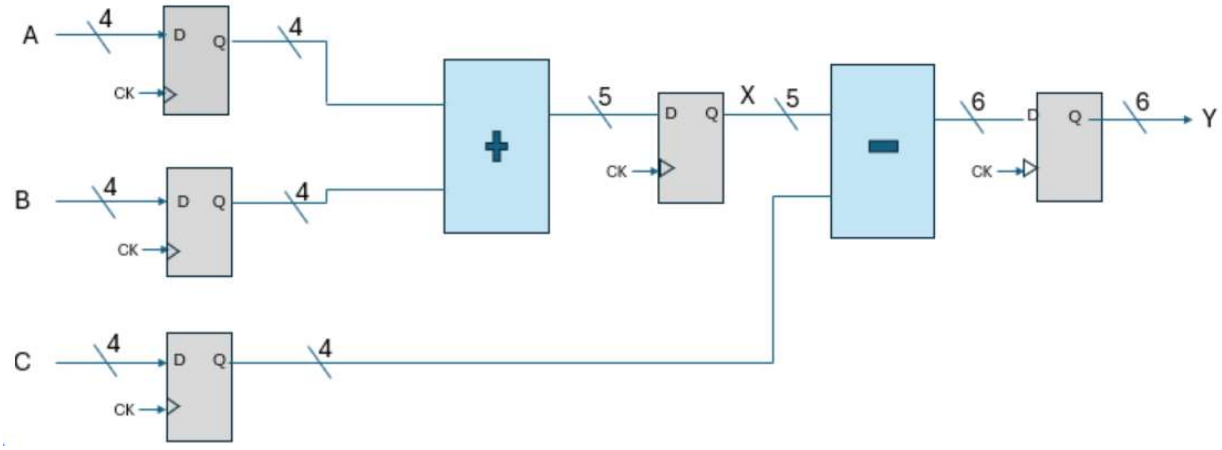


Figure 1

ALU Implementation:

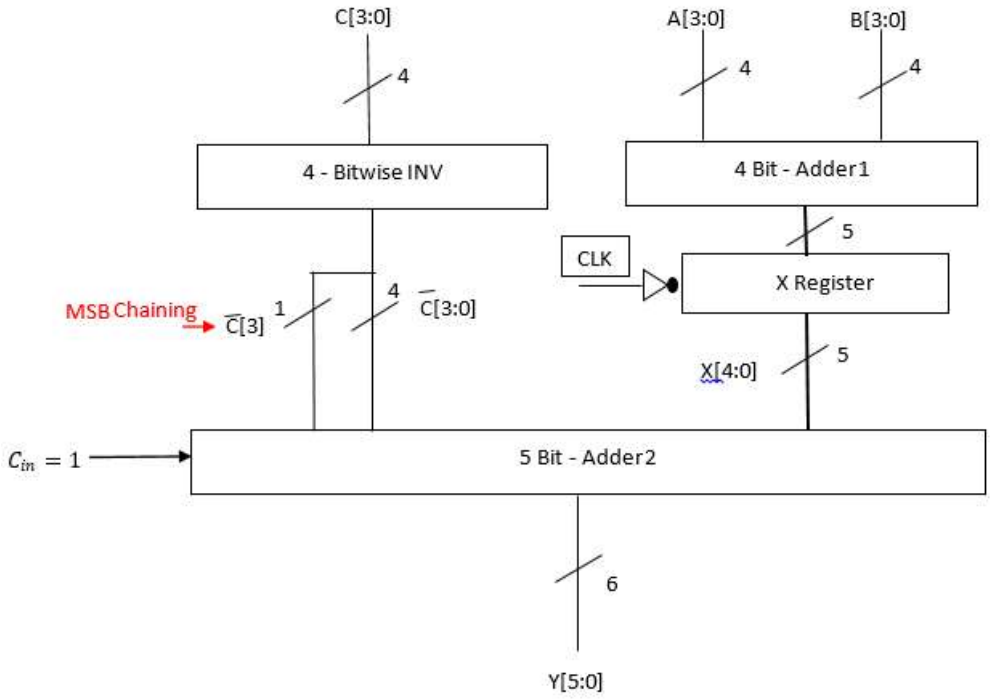


Figure 2

Bitwise 4 - bit inversion implementation:

We used the bitwise inverter to invert the C input bits:

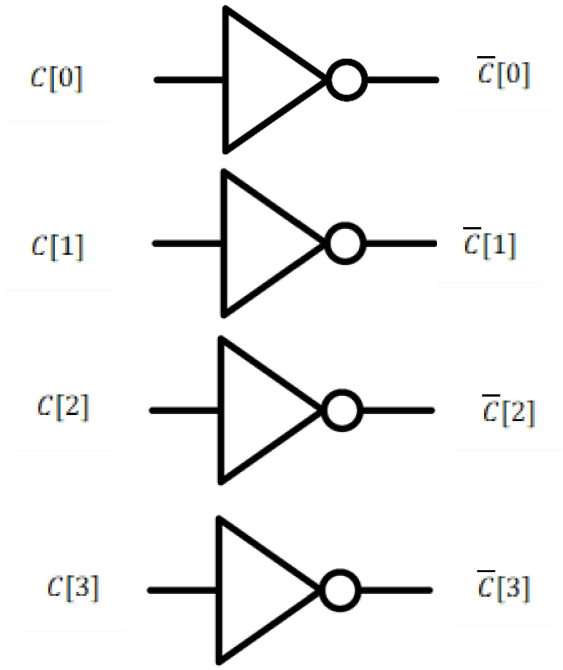
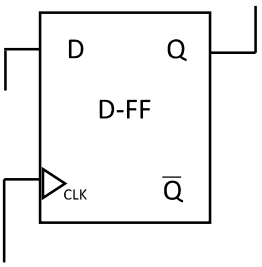


Figure 3

The Registers Implementations:

For register implementation we will use the D-FF cell



- 4 bit register implementation for inputs A ,B&C:

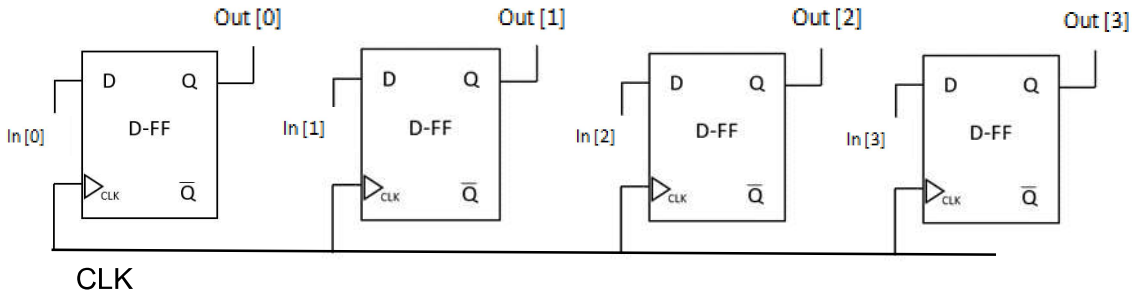


Figure 4

- 5-bit register implementation for X (the output of Adder 1)

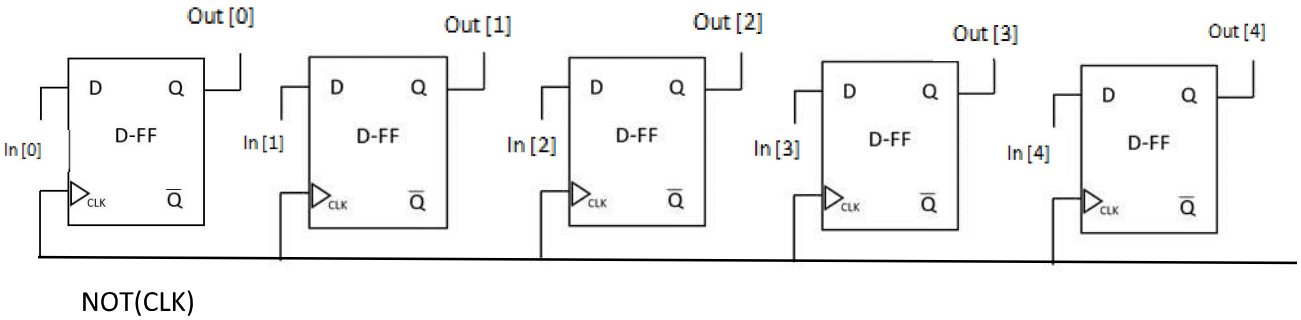


Figure 5

❖ 6-bit register implementation for the output Y

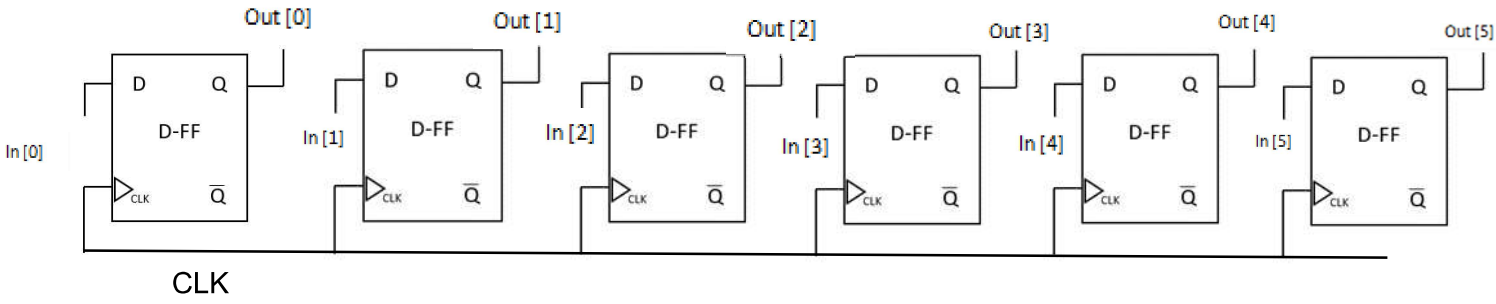


Figure 6

Adders Implementation: Kogge-Stone adder

❖ Full logical diagram of the 4- bit adder (Adder 1) & 5- bit adder (Adder 2)

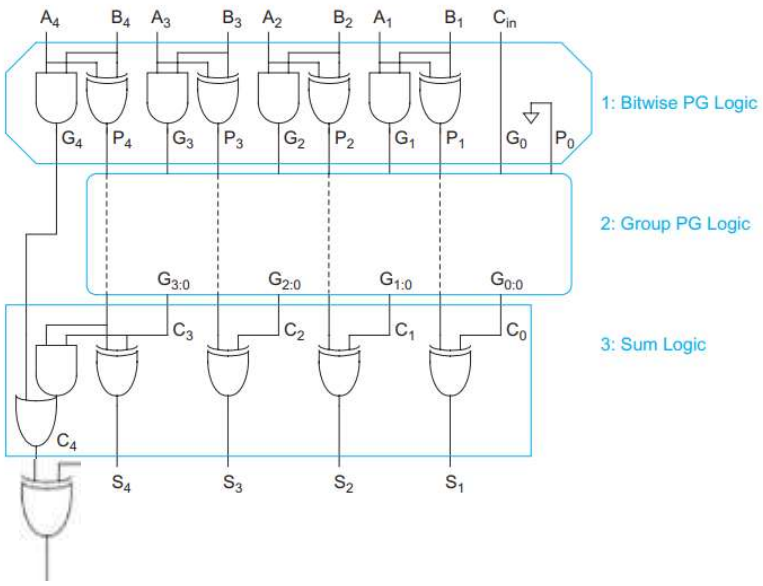


Figure 7 – 4- bit adder

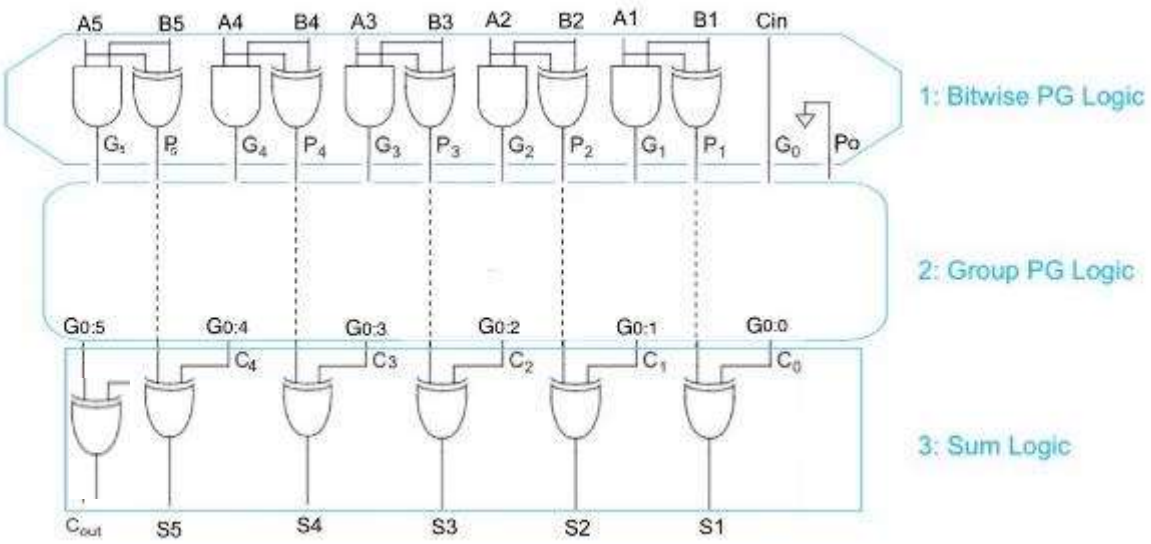


Figure 8 - 5-bit adder

- The tree adder PG network for 4-bit (PG logic)

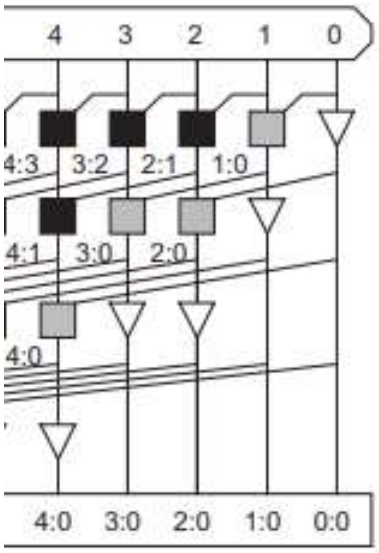


Figure 9

- The tree adder PG network for 5-bit (PG logic)

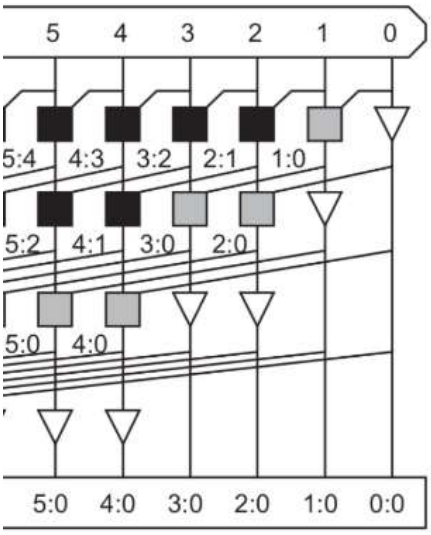


Figure 10

The black, Grey cells and the Buffer implementation:

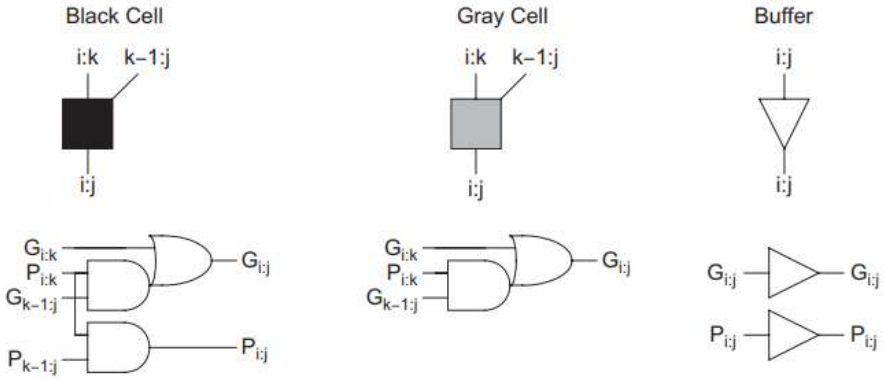


Figure 11

As we can see the inputs of the kogge stone adder are P and G (propagate, generate), so we are getting P and G from the bitwise PG logic stage as shown in figures

The Workflow of the ALU:

The inputs of the ALU are fed from three registers that store 4 bit inputs A, B and C.

Our goal is to compute ⟨Y⟩ = ⟨A⟩+⟨B⟩ −⟨C⟩, while ⟨Y⟩, ⟨A⟩, ⟨B⟩ and ⟨C⟩ are the two's complement representation of Y, A, B, and C respectfully.

The first adder (Adder 1) computes the sum of A [3:0] and B [3:0] strings, resulting in a 5-bit output X [4:0] which will be stored in a 5-bit low edge sensitive register. In the next stage we want to compute subtraction between X and C, thus we want to represent –C in two's complement representation and then apply ⟨X⟩ +⟨-C⟩ using 5-bit adder (Adder2).

In order to get ⟨-C⟩ we need to invert the bits of C and add 1 to the result, we will do that by applying a bitwise inversion on C [3:0], and then appending the MSB of the inversion result to the result in order to achieve a 5-bit length string ($\overline{C}[3]°\overline{C}\ [3:0]$) while saving the strings sign, so now we can fed ($\overline{C}[3]°\overline{C}\ [3:0]$) and X[4:0] into a 5-bit adder (Adder 2). The carry-in of Adder 2 is set to 1 to account for the +1 required when computing the two's complement to represent ⟨-C⟩. The result of the second adder is a 6-bit output Y [5:0] in two's complement representation (⟨Y⟩ =⟨X⟩ -⟨C⟩).

- We used low edge sensitive register for X in order to get a correct timing, in other words, to get the correct Y in the next high edge of the clock.

Block Level Floor Plan:

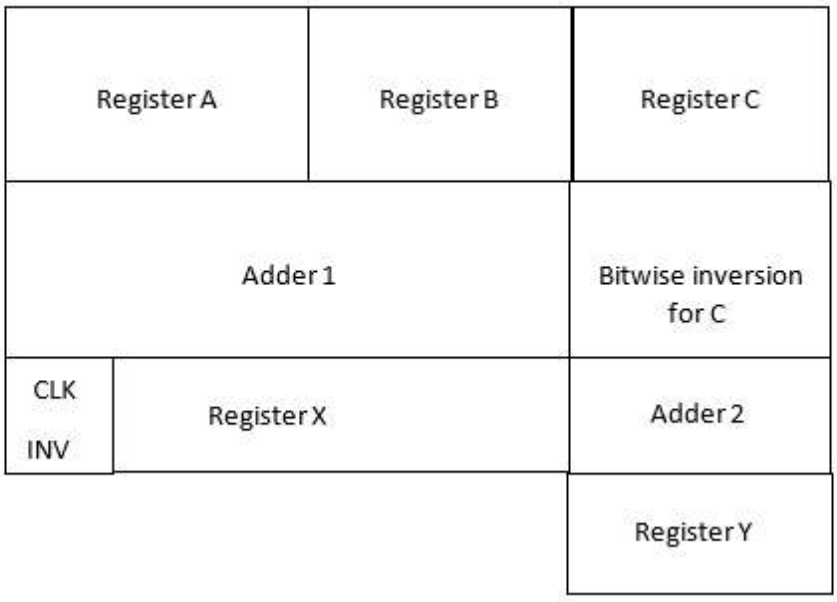


Figure 12

The number of cells:

- 3 x 4-bit register = 12 cells.
- 1 x 5-bit register =10 cells.
- 1 x 6-bit register = 6 cells.
- 1 x bitwise inverter = 4 cells.
- 1 x inverter = 1 cell.
- Adder 1 = 5 x buffer + 4 x gray cell + 4 x black cell = 5 + 4 x 2 + 4 x 3 = 25 cells.
- Adder 2 = 6 x buffer + 5 x gray cell + 6 x black cell = 6 + 5 x 2 + 6 x 3 = 34 cells.
- Bitwise PG logic (for Adder 1) = 8 cells.
- Sum logic (for Adder 1) = 4 cells.
- Bitwise PG logic (for Adder 2) = 10 cells.
- Sum logic (for Adder 2) = 5 cells.

→ We estimated in total 119 cells/components from gsclib045 that we are going to use in the ALU.

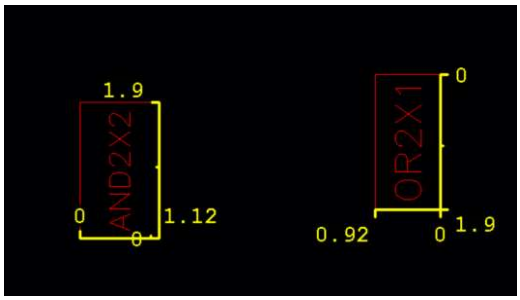
Component	Area $(um)^2$
AND2X2	2.128
OR2X1	1.748
XOR2X1	3.268
BUFX2	2.128
INVX1	0.988
DFFHQX1	6.308

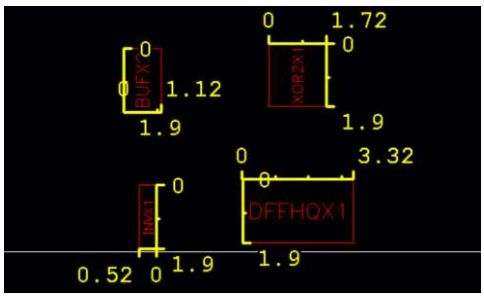
Sizes of used components:

- We measured the area using the detentions in layout XL, as shown in the images above.

Based on the number of cells and the sizes of each cell, we estimated the required area for a layout:

Size (23 x DFFHQX1 + 5 x INVX1 + 11 x BUFX2 +38 x AND2X2 + 19 x OR2X1 + 20 x XOR2X1) = 352.868$[(um)^2]$


1.9
AND2X2
0
0
1.12
OR2X1
0
0.92
0
1.9


0
1.72
0
XOR2X1
1.9
0
BUFX
0
1.12
1.9
0
3.32
0
0
INVX1
DFFHQX1
0.52
0
1.9
1.9

Schematic for the implementation:

We updated the schematic for the ALU that we did in the previous part, so that we added another 5-bit register after the bitwise inversion instead of NOT(T_{CLK}), from a consideration of getting a lower clock cycle time.

The final schematic of the full ALU in virtuoso:

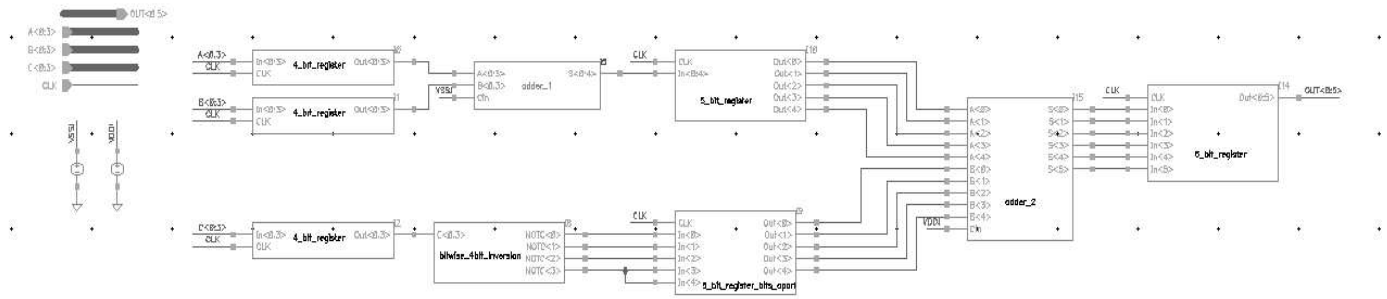


Figure 1 – ALU

Our implementation is made of several blocks, we will discuss each block individually.

1. **4-bit register:** a component that made of 4 D-FF + CLK which used to store the initial values of the inputs A, B and C.

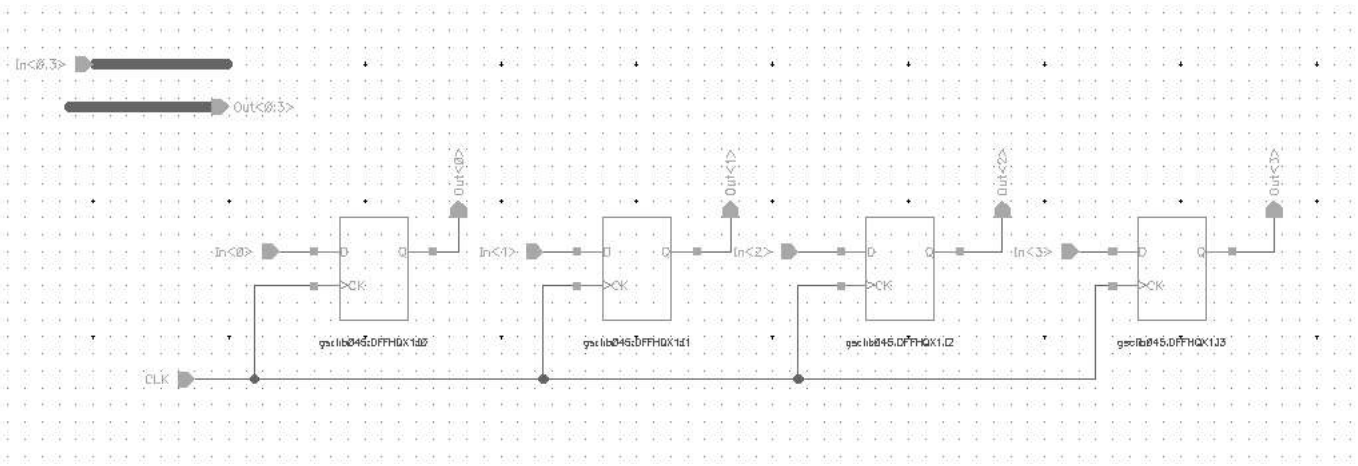


Figure 2 – 4-bit register

Figure 3 – symbol of the 4-bit register

2. **Adder_1 (4:5 adder):** used to calculate the inputs A + B, the output X is entered to 5-bit register.
   This adder is a Kogge Stone adder.

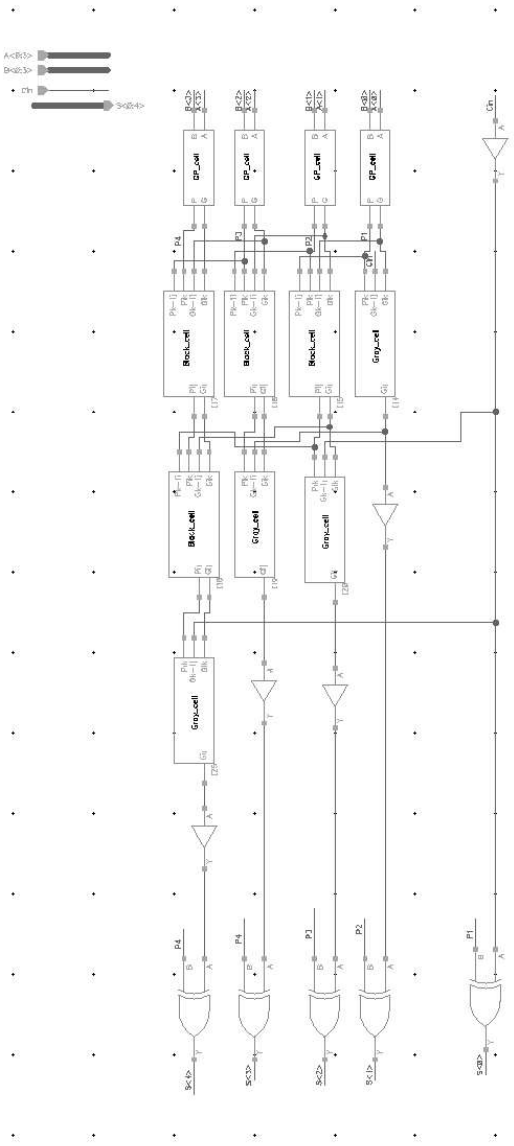


Figure 4 – adder_1 schematic

Figure 5 – adder_1 symbol

This adder, is made of 3 types of blocks:

a. BLACK CELL:

Schematic:

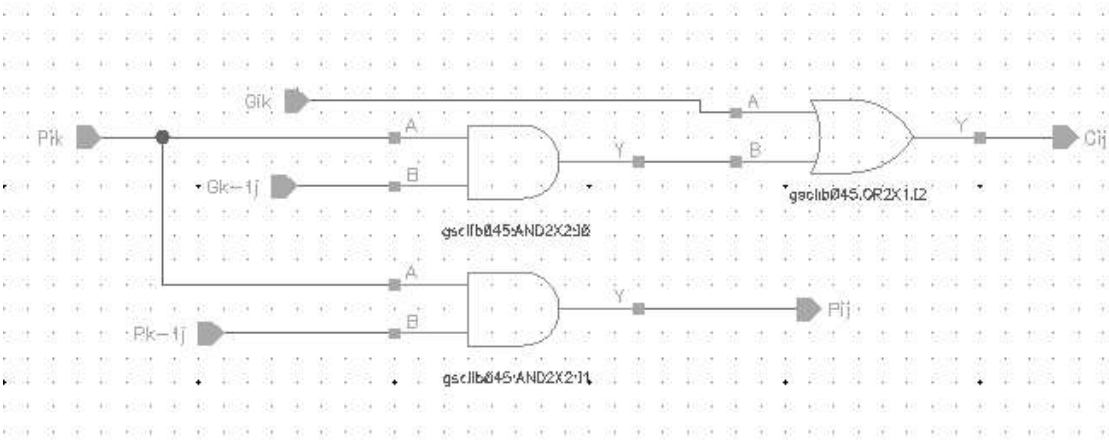


Figure 6 – Black cell schematic

Symbol:

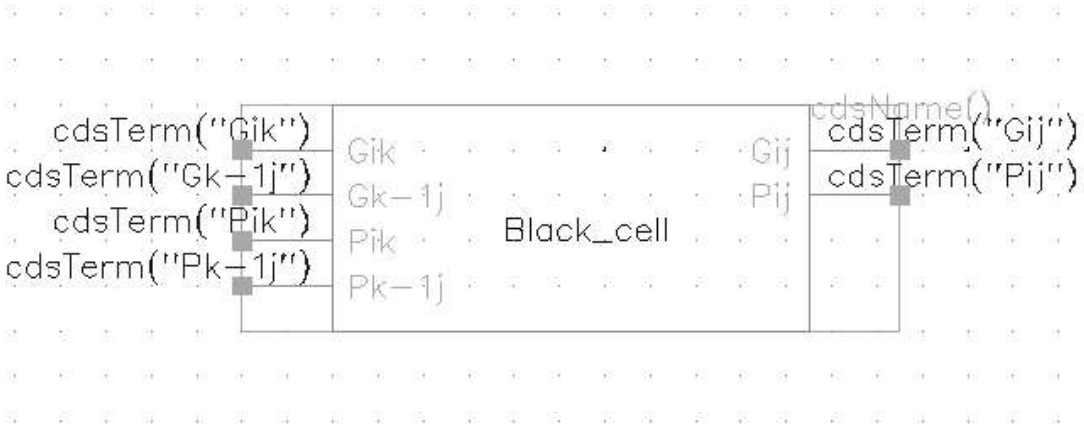


Figure7 – Black cell symbol

Layout:

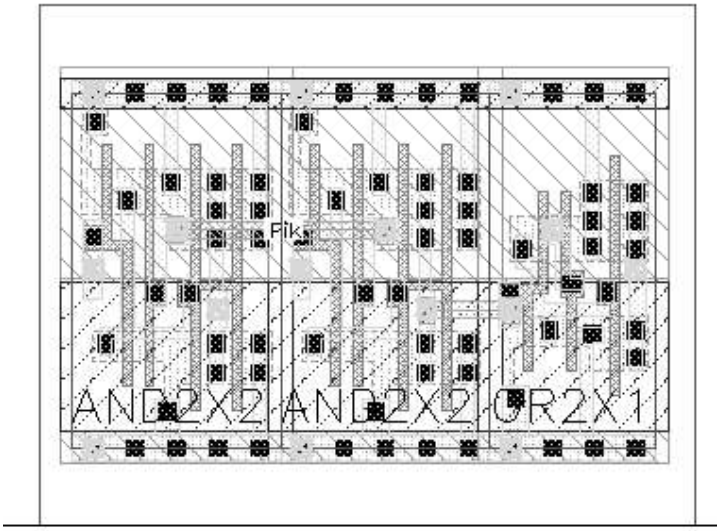


Figure8 – Black cell layout

QUANTUS, DRC, LVS:

b. GREY-CELL

Schematic:

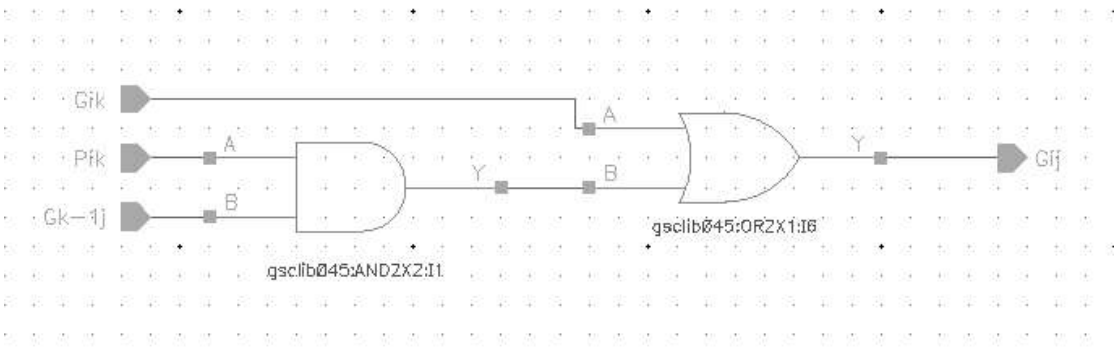


Figure 9 – Grey cell schematic

Symbol:

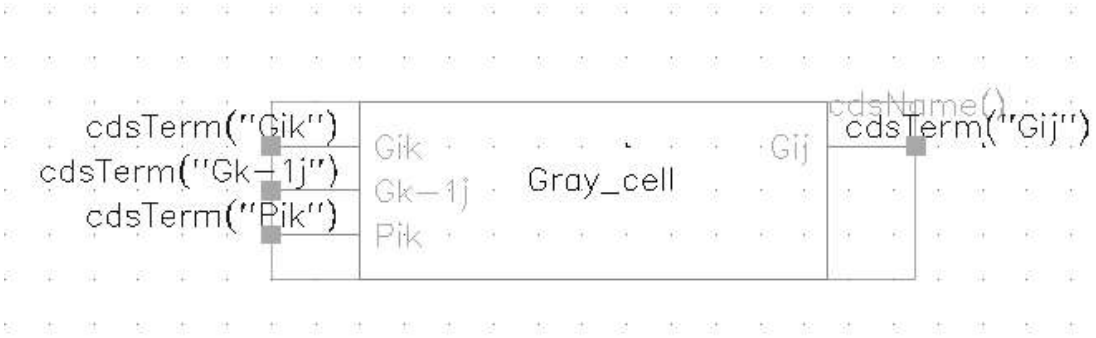


Figure10 – Grey cell symbol

Layout:

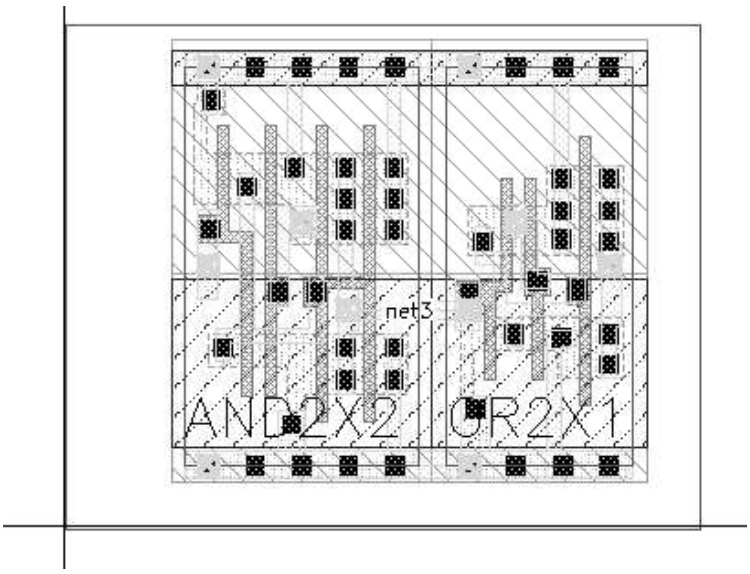
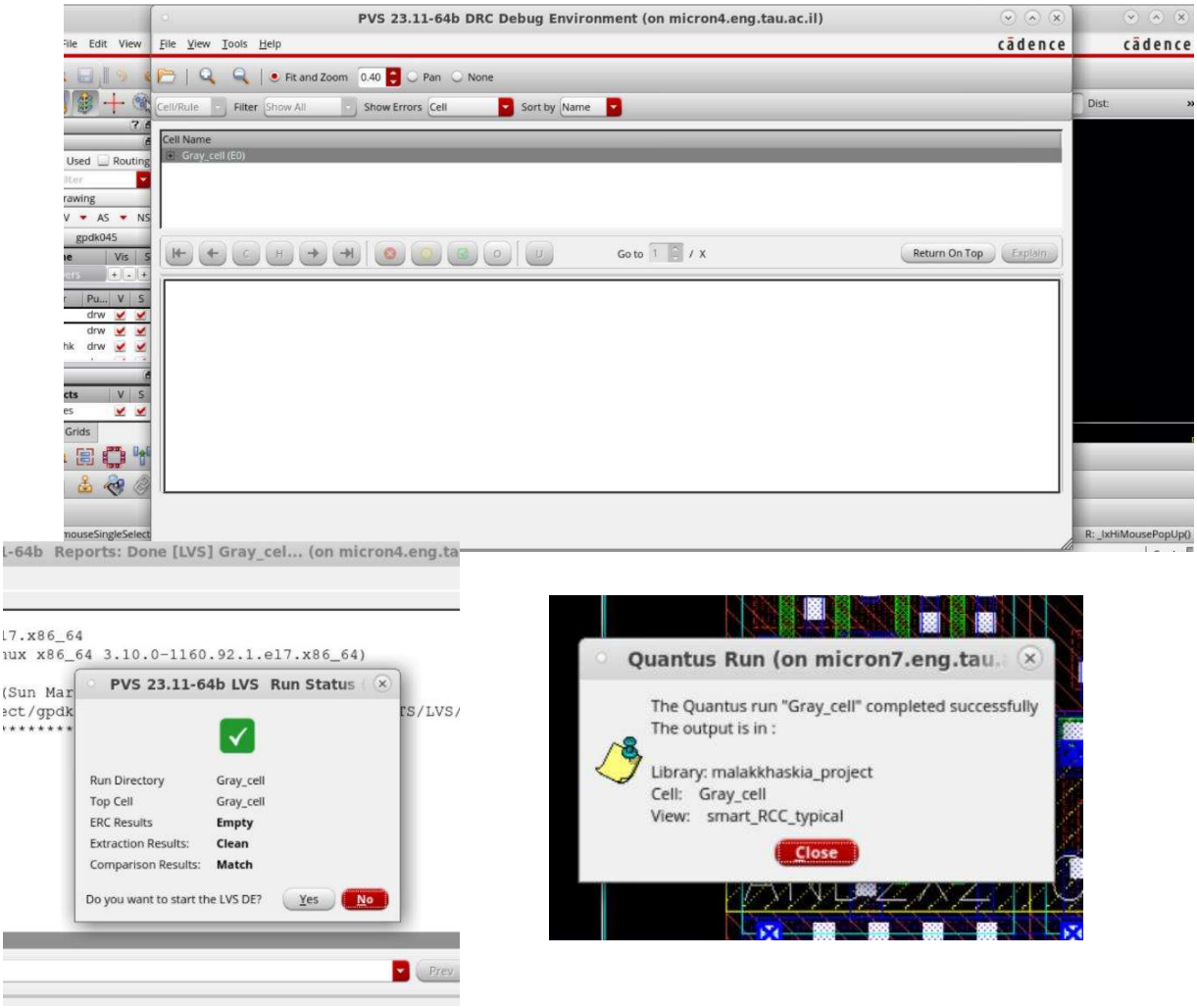


Figure11 – Grey cell layout

QUANTUS, DRC, LVS:



c. GP-CELL:

Schematic:

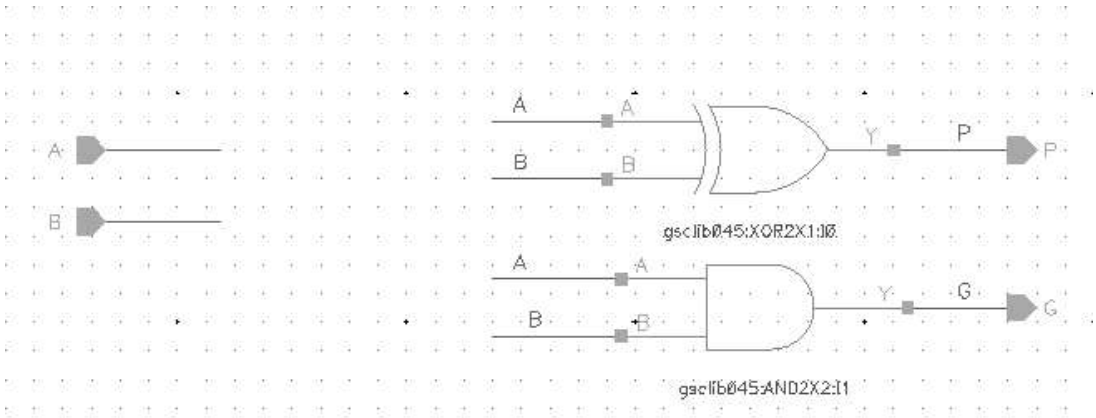


Figure12 – GP cell schematic

Symbol:

Figure13 – GP cell symbol

Layout

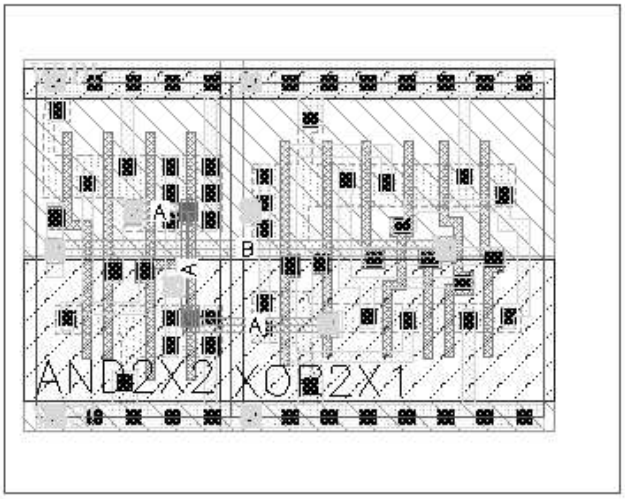
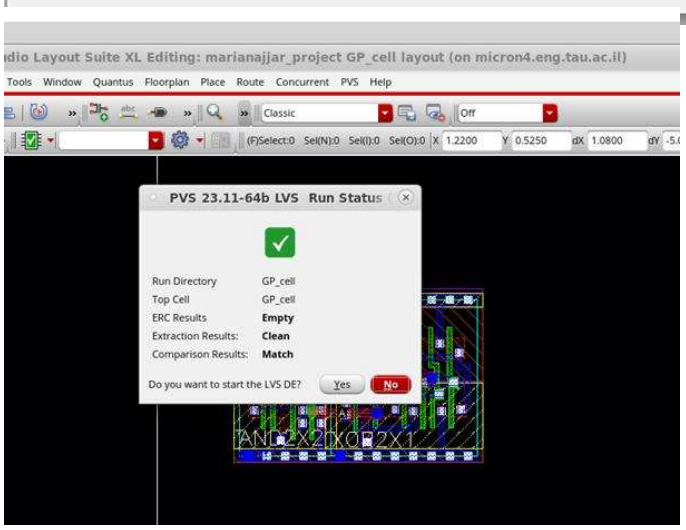
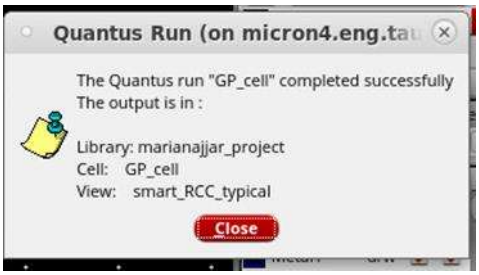


Figure14 – GP cell layout

QUANTUS, DRC, LVS:





3. 5-bit register: a component that made of 5 D-FF + CLK which used to store the values of X.

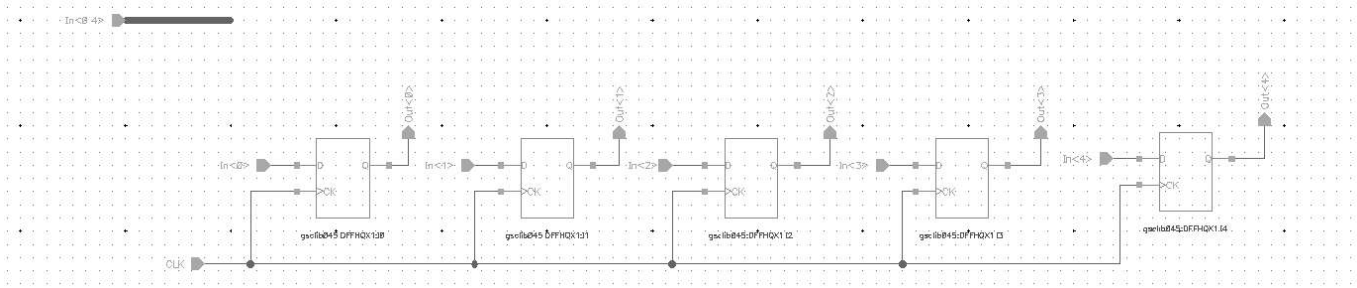


Figure15 – 5-bit register schematic

We used 2 types of symbols in the implementation:

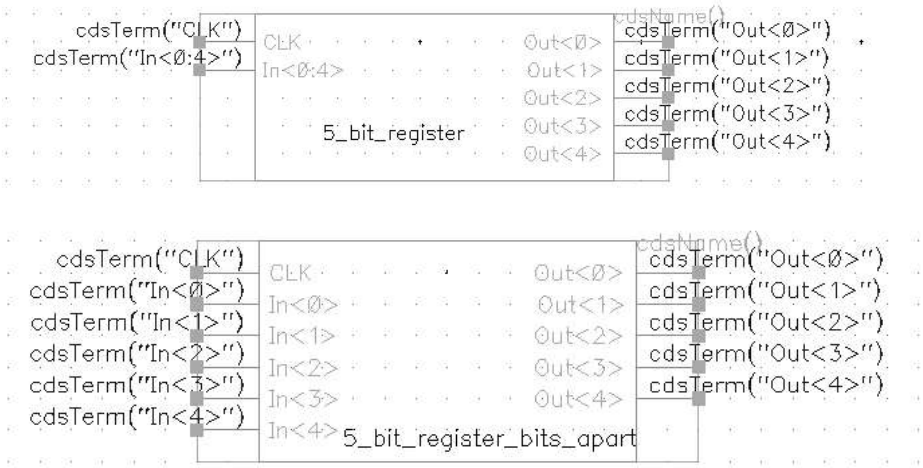


Figure16 – 5-bit register symbols

3. Bitwise_4bit_inverter: used to perform NOT for C.

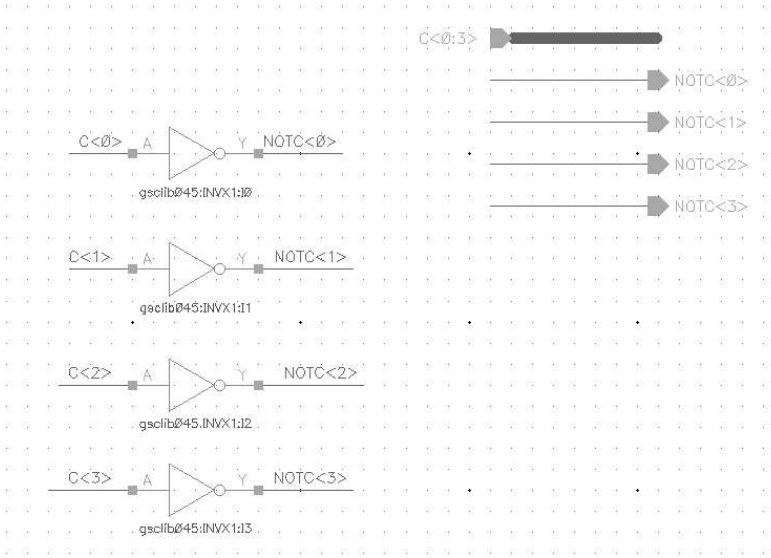


Figure17 – Bitwise inverter schematic

Figure18 – Bitwise inverter symbol

4. Adder_2 (5:6): used to calculate <X>-<C>, the output Y is entered to 6-bit register. This adder is a **Kogge Stone adder**.

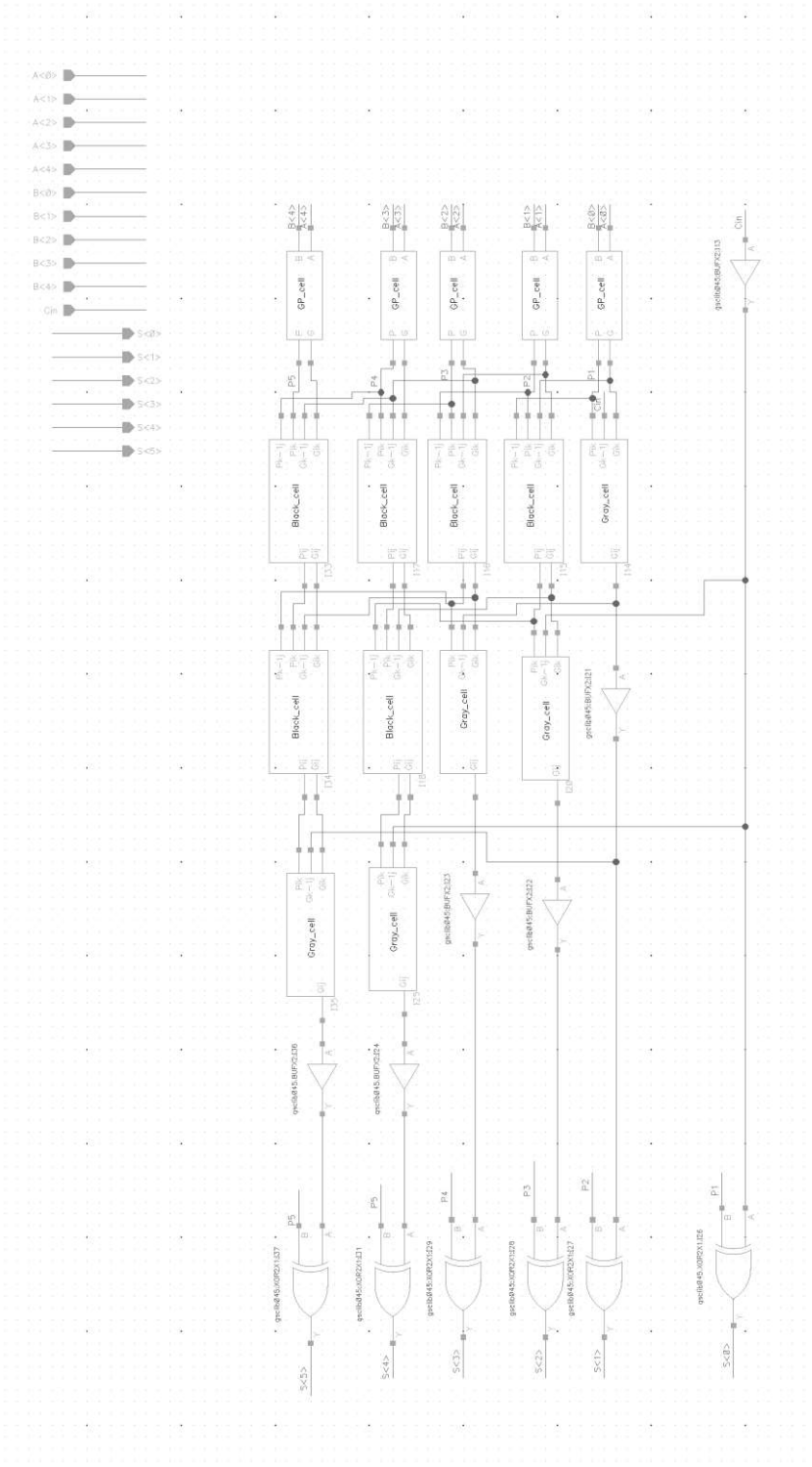


Figure19 – Adder_2 (5:6) schematic

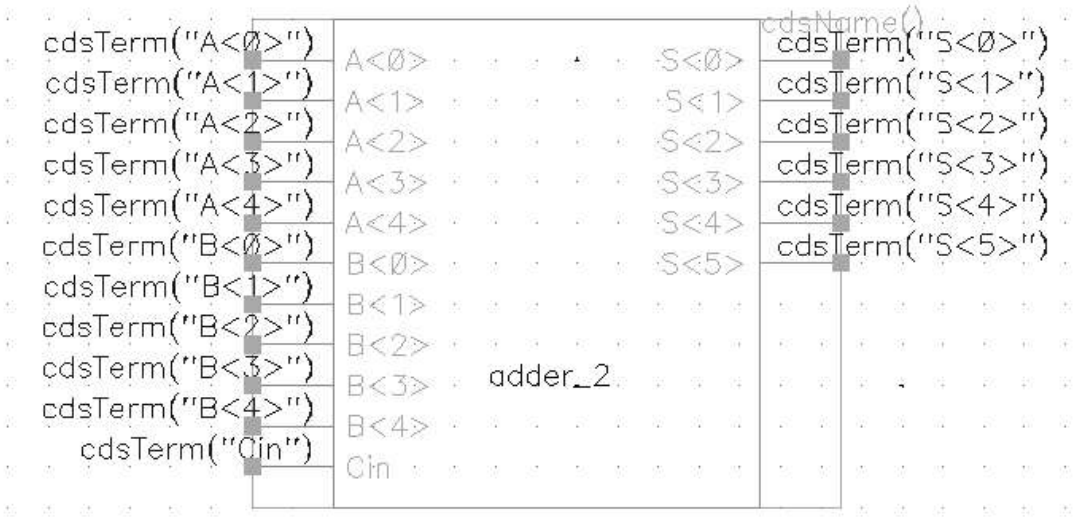


Figure20 – Adder_2 (5:6) symbol

Also, this adder is made of blocks as we shown in Adder_1.

5. 6-bit register: a component that made of 6 D-FF + CLK which used to store the values of <Y>=<X>-<C>.

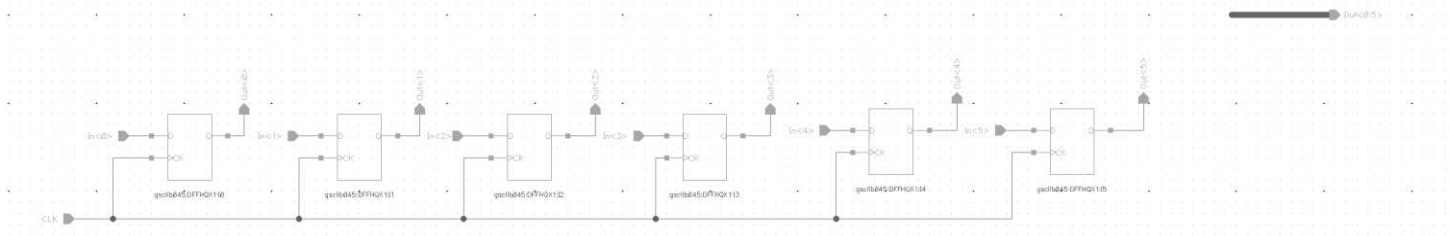


Figure21 – 6-bit register schematic

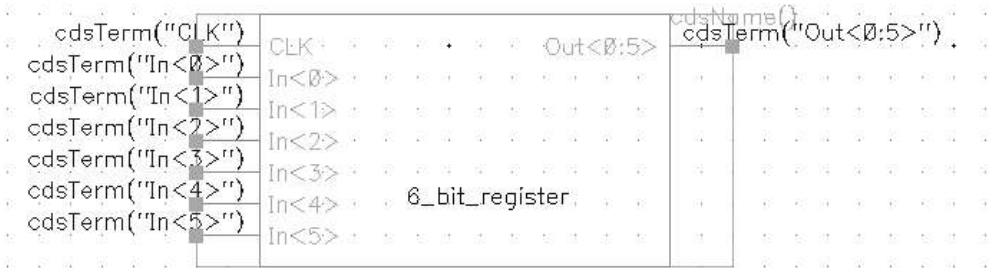


Figure22 – 6-bit register symbol

Tests for the ALU:

Adder_1(4:5):

TO verify that the 4- bit adder performs calculations correctly, we created the following benchmark:

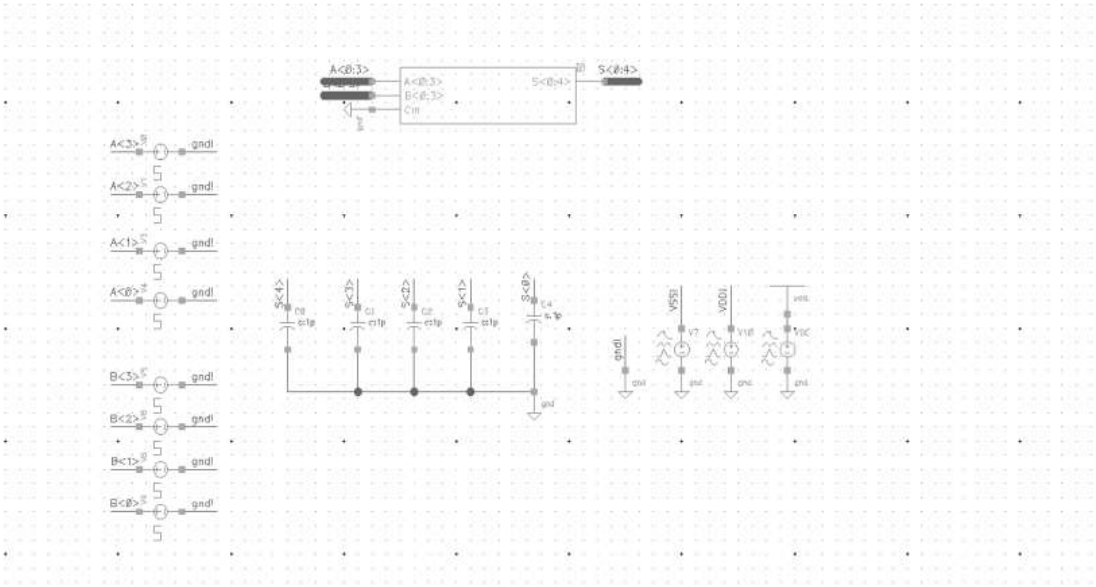


Figure23- Test schematic for adder 1

We selected a finite set of arbitrary inputs A and B and checked whether the output is <x> =<A>+ <B>

The results:

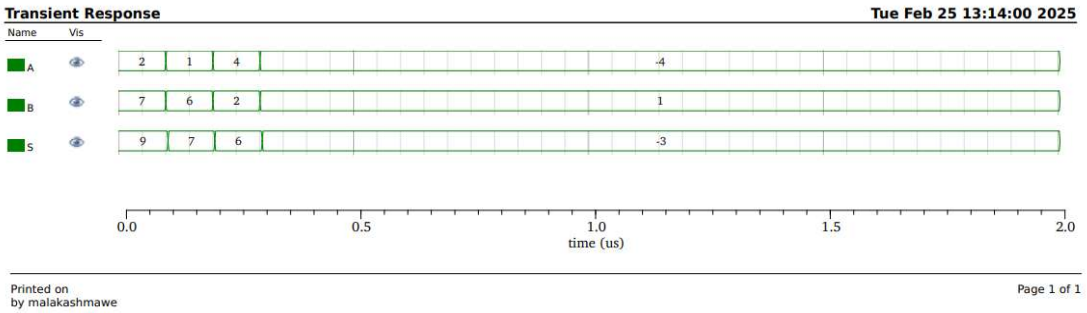


Figure24- Test results for adder 1

We can see that the adder functions correctly.

Adder2(5:6) Test:

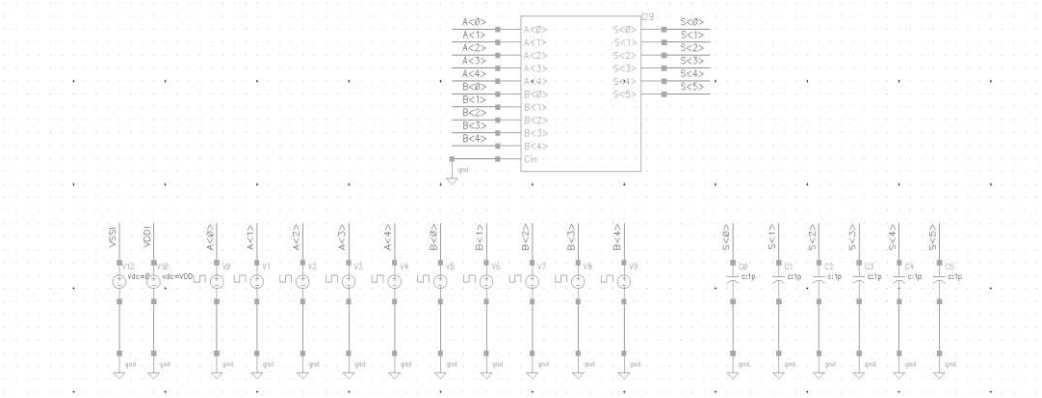


Figure25- Test schematic for adder 2

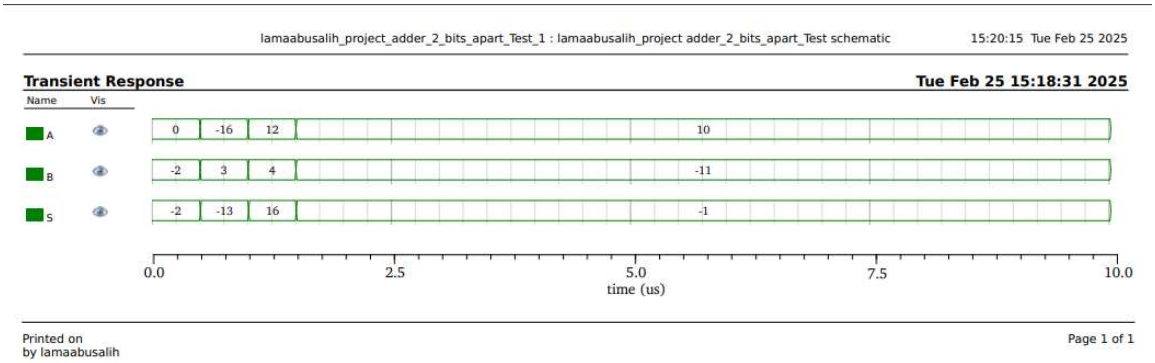


Figure26- Test results for adder 2

ALU TEST:

We used a similar test bench to check the accuracy of the full ALU

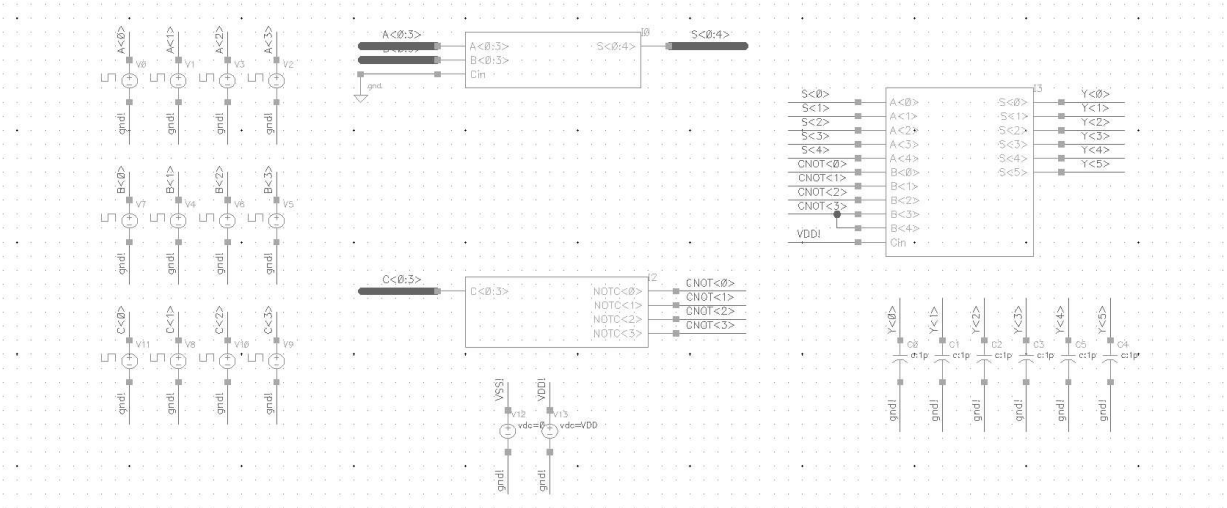


Figure27- Test schematic for the full ALU

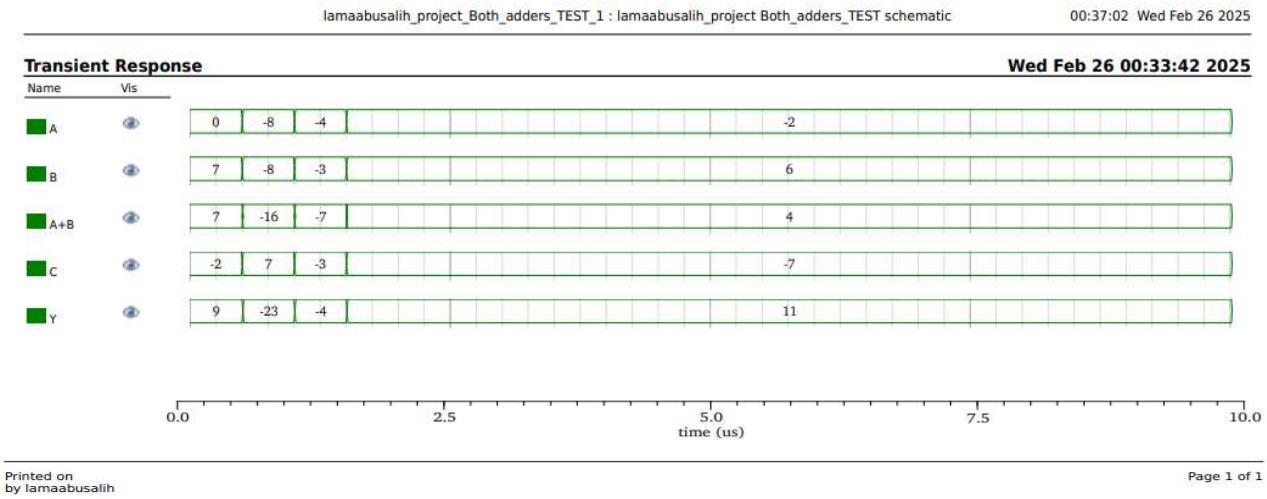


Figure28- Test results for the full ALU

We can easily notice that <y>= <A>+ <B>-<C>, thus the full ALU operates correctly from a logical perspective.

Measuring time delay for the critical path:

As we saw in class, in order to find $T_{CLK,min}$, we have to measure : t_{logic}, t_{cq}, t_{setup}.

Measuring t_{logic}:

First, we have to realize the critical path between 2 registers in our ALU.

Since our ALU is build to across 3 registers and each logical component surrounded between 2 registers so it's enough to measure the delay for the logical component itself. In conclusion we will measure the delay for bitwise inversion, adder1 and adder2.

Pay attention to that the bitwise inversion work in parallel so it's enough to measure the delay of a single inverter.

We defined $t_{logic} = \frac{t_{plh}+t_{phl}}{2}$, then we build the testbench in order the measure the delay for each component:

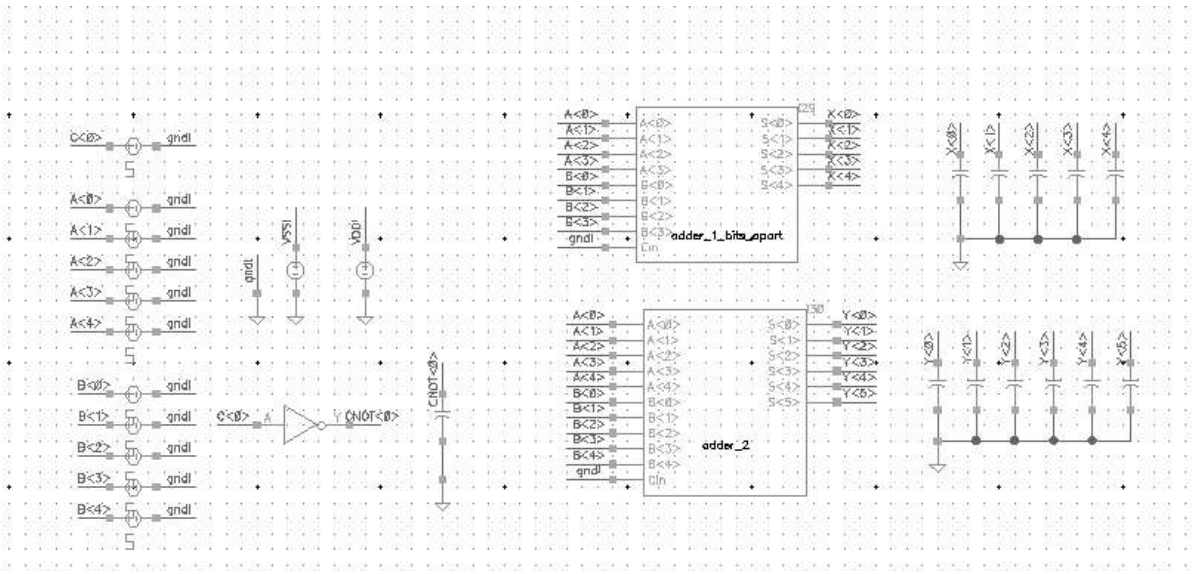
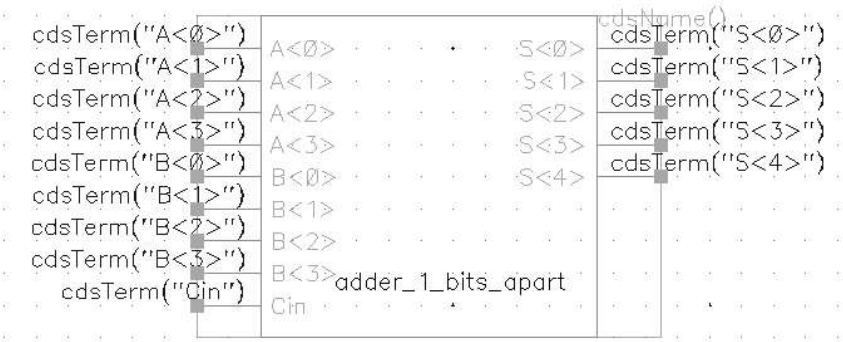


Figure29- Test bench for t_{logic}

Here, we used a symbol of adder_1 which called adder_1_bits_apart:



We insert for each v_bit a random sequence of bits and then we measured using markers the delays as we see in class. We took the max delay for each exit of the 3 components.

For V_{DD}=1.2V:

	t_{phl}[nsec]	t_{plh}[nsec]	t_{logic}[nsec]
INV	4.012	4.012	4.012
Adder_1	4.084	4.069	4.0765
Adder_2	4.07	4.03	4.05

For V_{DD}=900mV:

	t_{phl}[nsec]	t_{plh}[nsec]	t_{logic}[nsec]
INV	7.368	6.98	7.174
Adder_1	7.506	7.082	7.294
Adder_2	7.105	7.381	7.243

As we can see from the results above, we got the smallest delay for the inverter, and a very close delays for both adders, on order of [psec], so we can consider one of them as the component of the critical path, and because we did the layout for adder1, so we will continue the measurements for it.

After we did Quantas and did the measurements again, we got:

Adder_1(4:5)	VDD[V]	t_{phl}[nsec]	t_{plh}[nsec]	t_{logic}[nsec]
With RC extraction	1.2	4.323	6.022	5.173
	0.9	7.36	7.592	7.476
Without RC extraction	1.2	4.084	4.069	4.0765
	0.9	7.506	7.082	7.294

Measuring t_{cq}, t_{setup}:

As we know, t_{cq}, and also t_{setup}, are relevant to registers and not for the logical component, so it is enough to check these values for 1 D-FF (the unit that register is made of, which transfers information in parallel to parallel mode).

We built the next test-bench to measure both values:

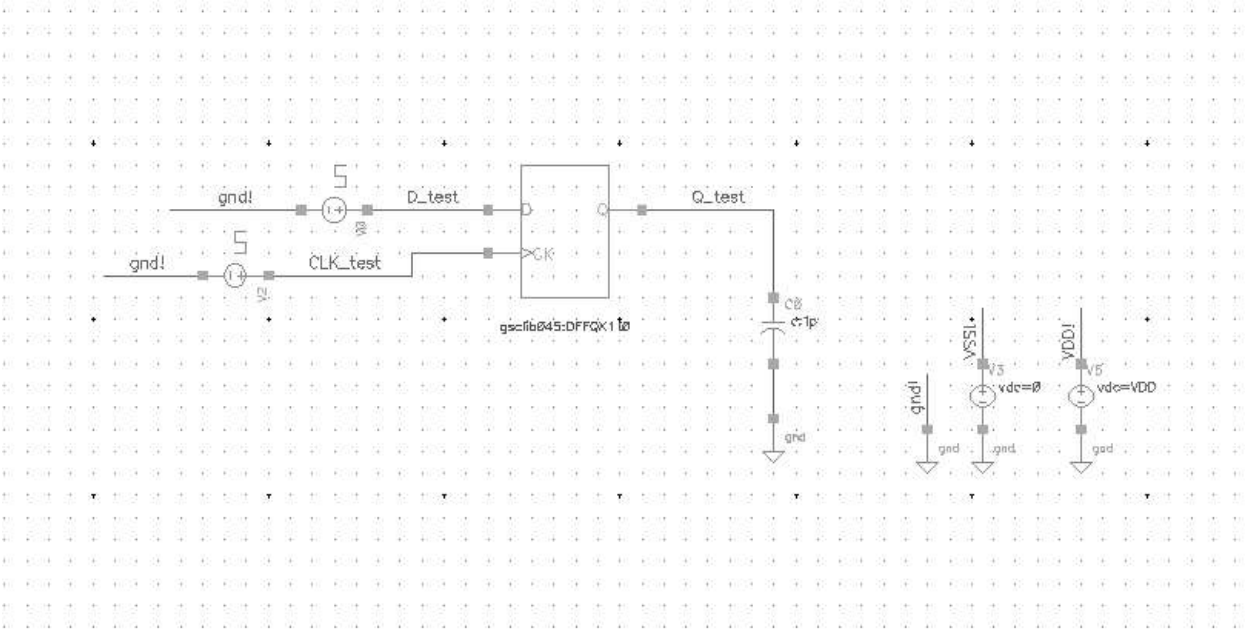


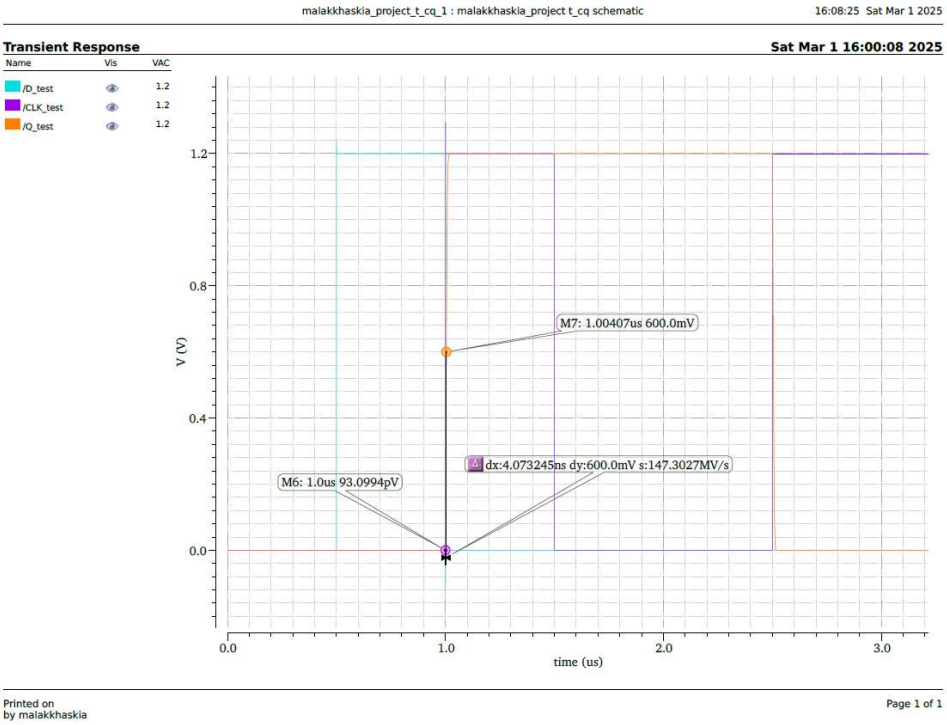
Figure 30- Test bench for t_{cq}

t_{cq}:

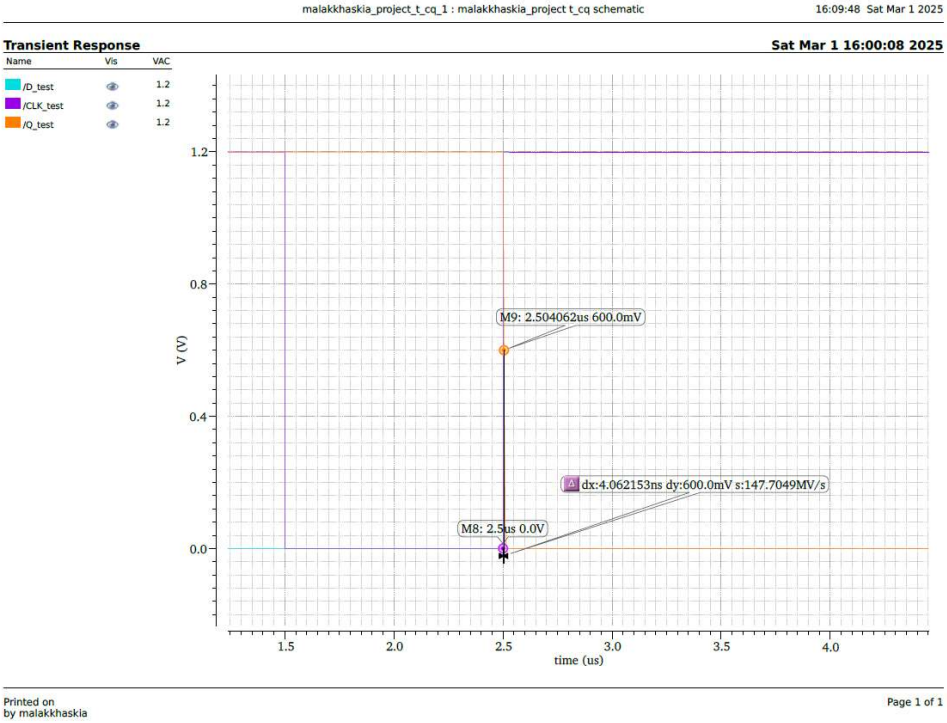
We defined $t_{cq} = \frac{t_{cq\_lh} + t_{cq\_hl}}{2}$

From the definition, we want to measure the delay between clock raising edge and getting a stable logical value in the output. (As we learned, a logical component identified the change of the logical state when the voltage across the threshold voltage = $\frac{V_{DD}}{2}$)

For V_{DD} = 1.2V:

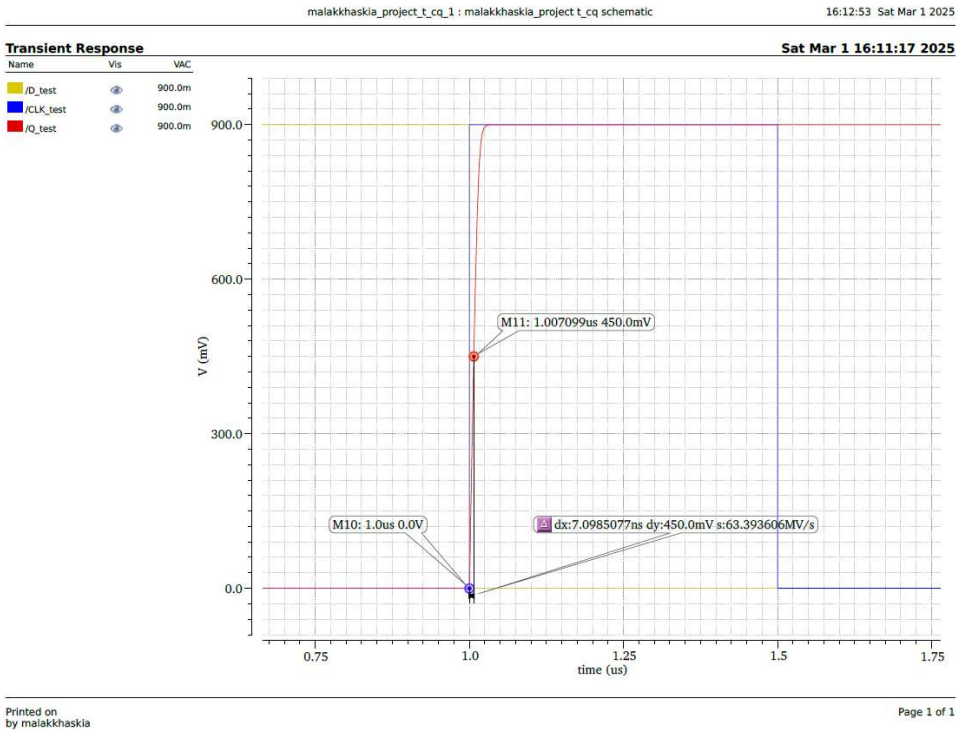


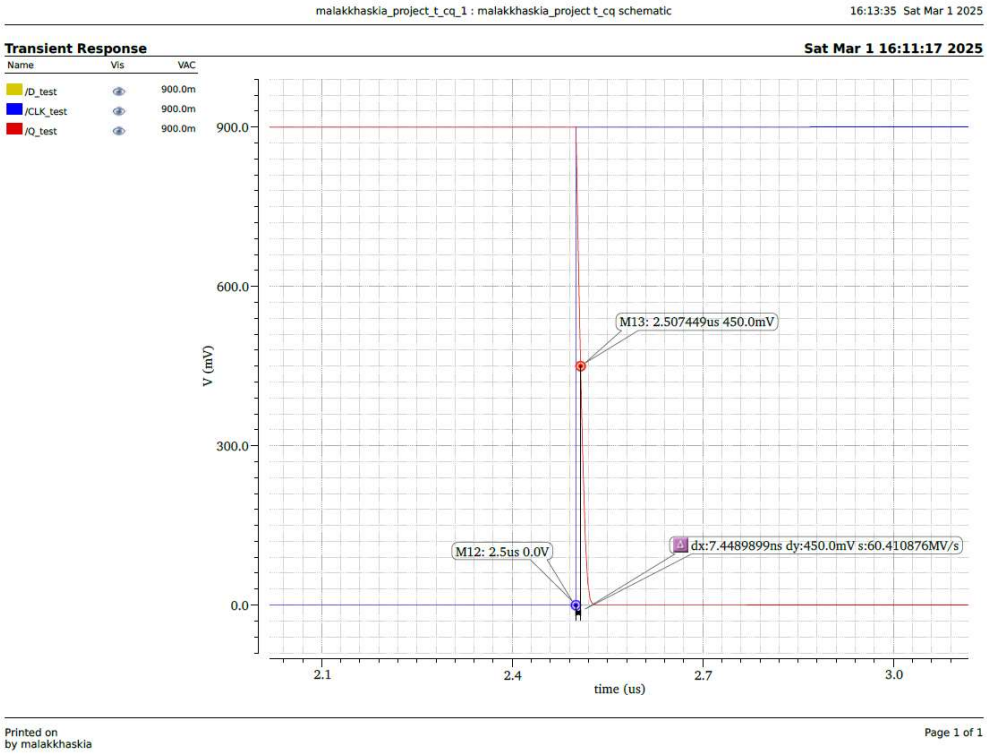
Measured Value	Time [nsec]
t_{cq_lh}	4.073
t_{cq_hl}	4.062



We got $t_{cq} = 4.0675\ [nsec]$

For V_{DD} = 900mV:





Measured Value	Time [nsec]
t_{cq_lh}	7.098
t_{cq_hl}	7.448

We got $t_{cq} = 7.273\ [nsec]$

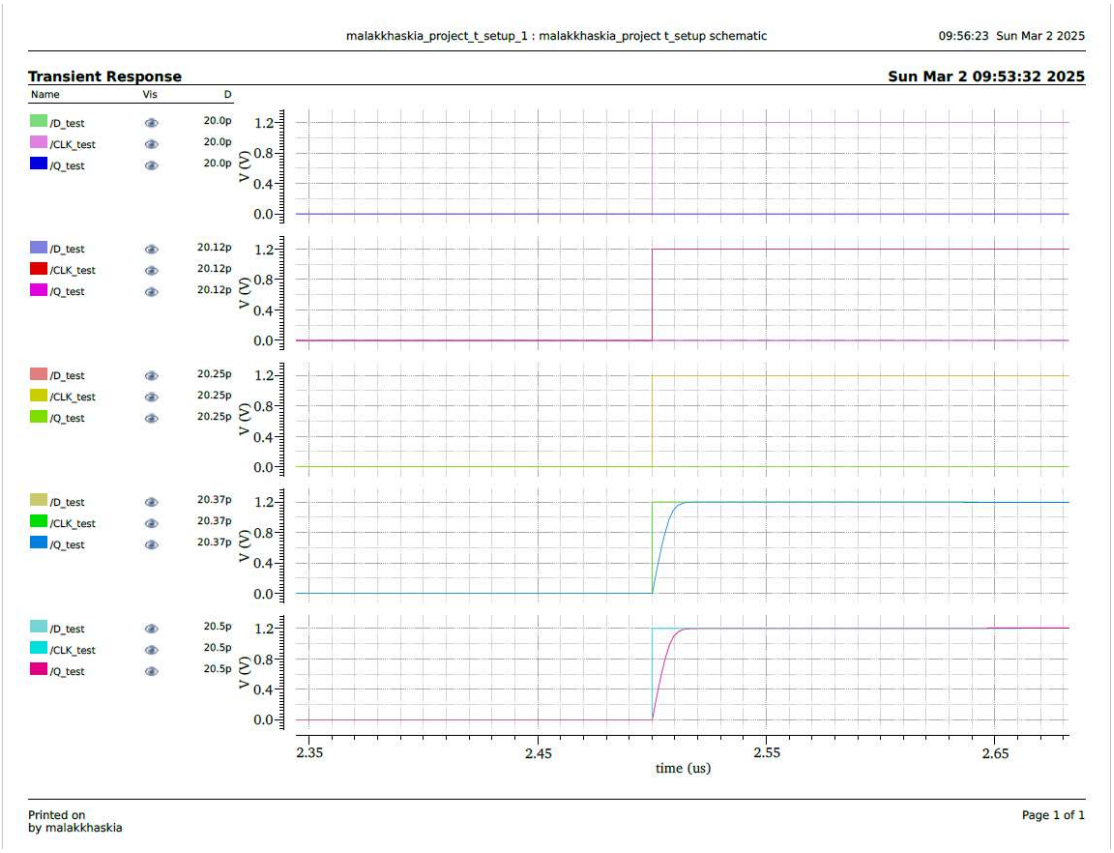
t_{setup}:

As we mentioned before, we use the same testbench for 1 D-FF.

From the definition, we want to find the minimum time for which the input of a FF will be stable before the edge of the CLK.

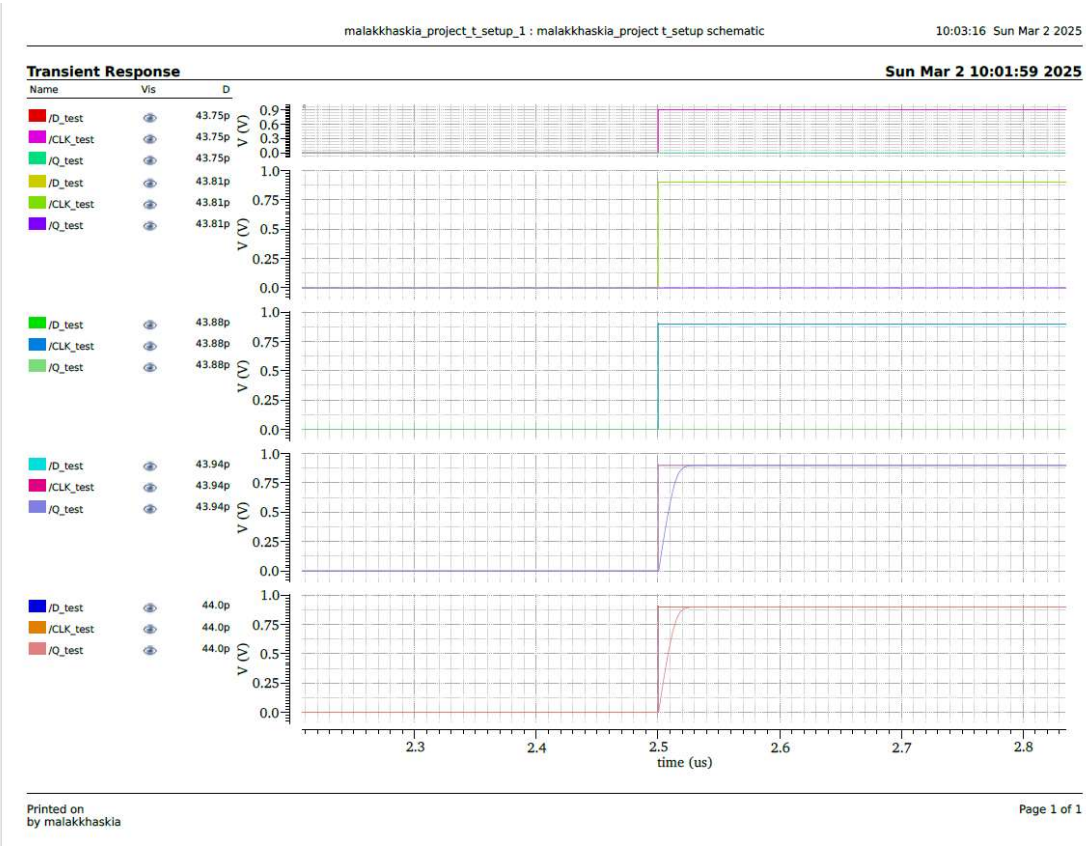
For this sake, we defined a new parameter D, refers to the delay between the CLK and the input, and take some values to find when the output starts to update with the raising edge of the CLK.

For V_{DD} = 1.2V:



The output starts to update after t_{setup}=20.37 [psec]

For V_{DD} = 900mV:



The output starts to update after t_{setup}=43.94[psec].

Measuring minimal T_{CLK} / maximal f_{CLK}:

We found all the requirement values to measure time period for the CLK:

$$T_{CLK} \geq T_{CLK,min} = t_{logic} + t_{cq} + t_{setup}$$

Quantas	V_{DD} [V]	t_{logic}[nsec]	t_{cq} $[nsec]$	t_{setup}[psec]	$T_{CLK,min}$[nsec]	f $[MHz]$
Without RC extraction	1.2	4.0765	4.0675	20.37	8.164	122.48
	0.9	7.294	7.273	43.94	14.611	68.44
With RC extraction	1.2	5.173	4.0675	20.37	9.261	107.98
	0.9	7.476	7.273	43.94	14.793	67.59

As expected the T_clk increases when the supply voltage VDD decreases, reasons for this are:

Reduced transistor drive strength because of lower supply voltage that means a smaller voltage drop on the transistors which build the gates. This reduces the gates ability to drive current and change its state quickly. Moreover, the transistors are weaker so they switch modes slowly, the slower switching directly translates to increased propagation delays through the gates.

Pass Fail tests:

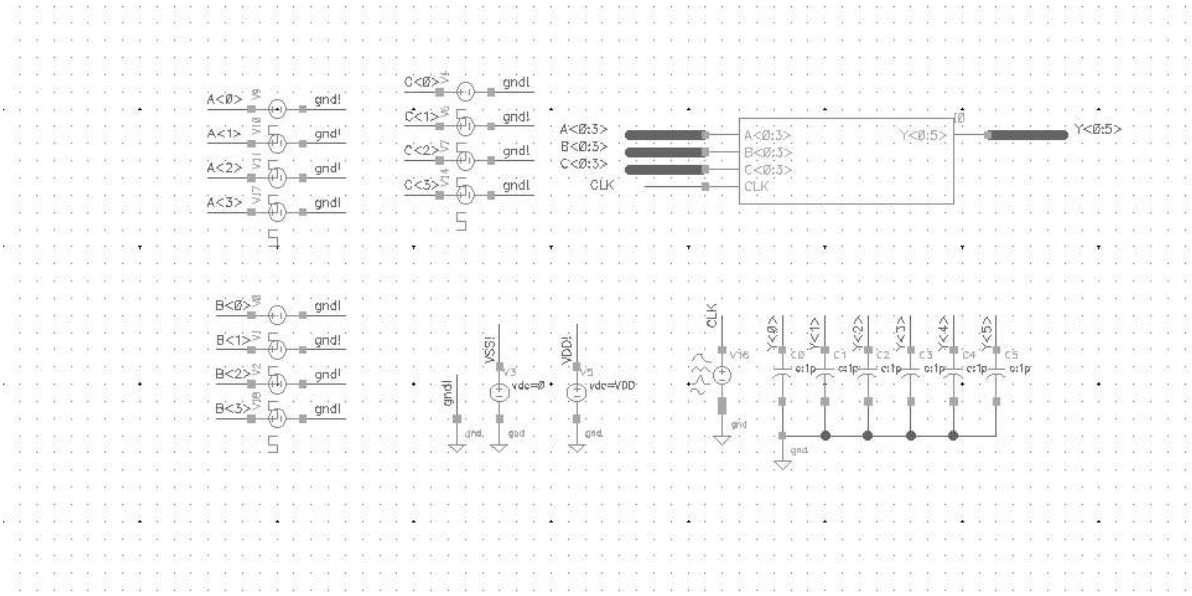
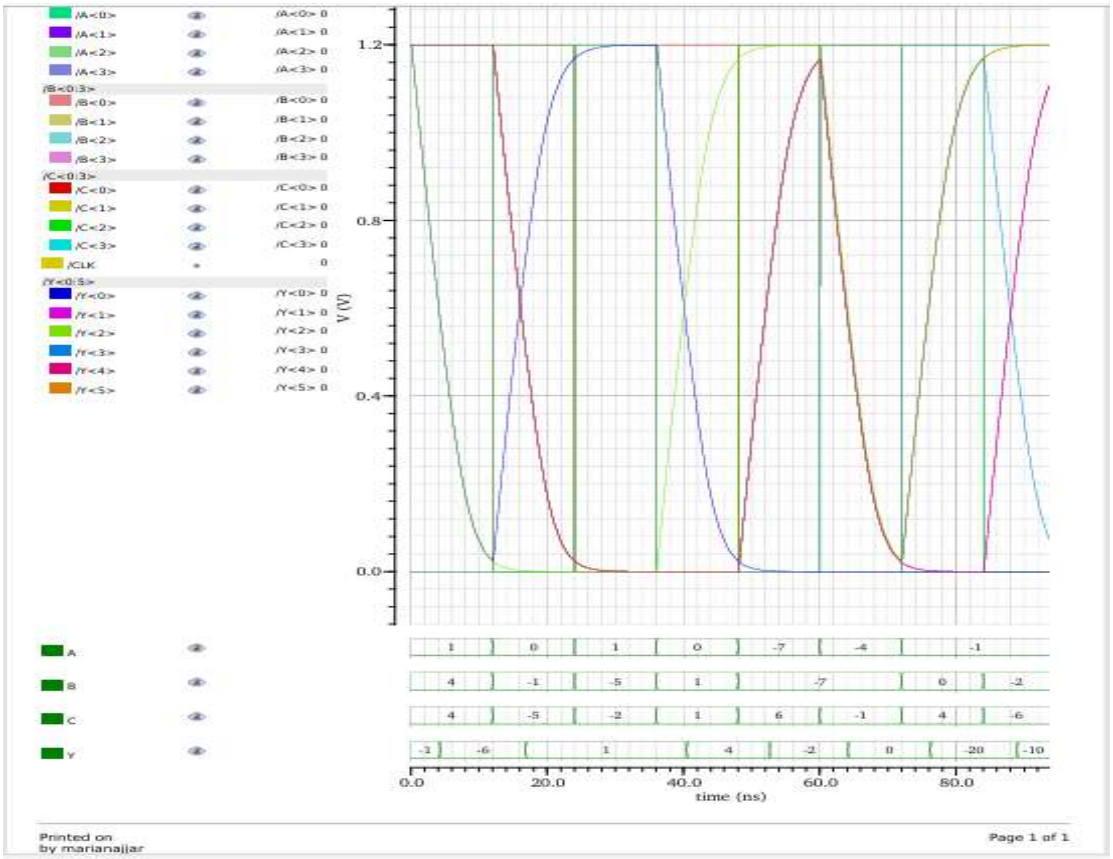


Figure 31- pass/fail test schematic

We did a pass fail tests for the full ALU with a CLOCK input and obtained the following results for the actual minimum clock period:

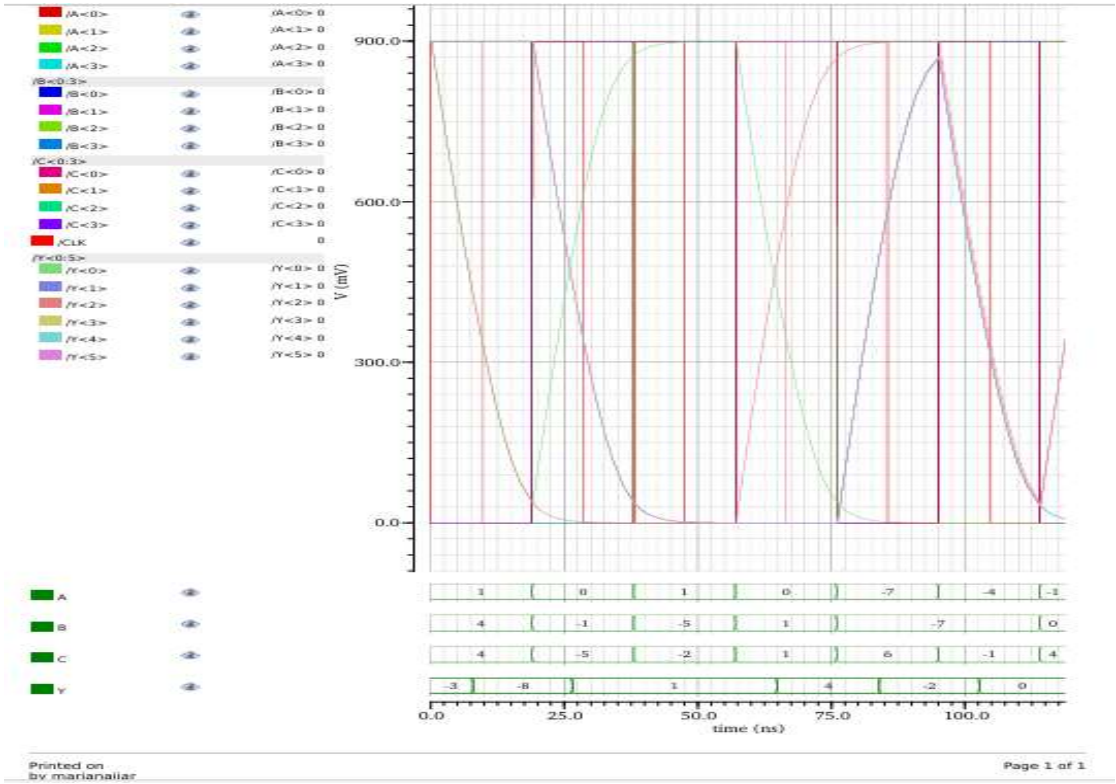
<u>For VDD= 1.2 V:</u>

$T_{clk\_minimal}$ = 10 nsec



For VDD= 0.9 V:

$T_{clk\_minimal}$ = 19 nsec



Layout for Adder_1 (4:5):

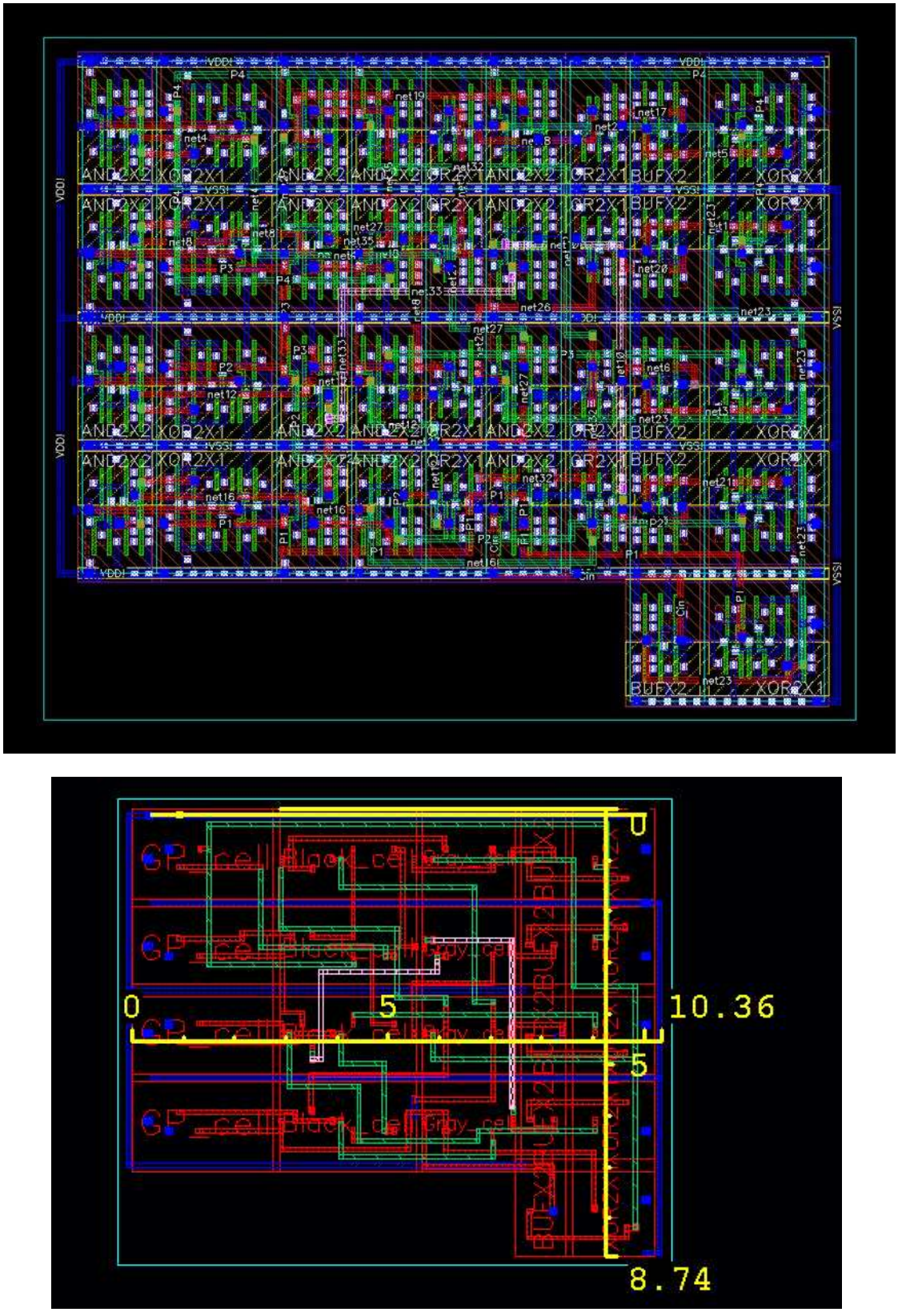


Figure 32- Layout of Adder_1 (4:5)

- <u>The 4-bit adder (Adder1(4:5)) consists of the following components:</u>

  5 *(buffer x2) +4 *(GP cell) +4 *(grey cell) + 4 *(black cell) +5*(XOR 2X1)

- The estimated total area calculation:

$$S_{total} = 4 * S_{(\text{GP cell})} + 5 * S_{\text{Buf x2}} + 4 * S_{(\text{grey cell})} + 4 * S_{(\text{black cell})} + 5 * S_{(\text{XOR 2X1})}$$
$$= 4 * 5.396 + 5 * 2.128 + 4 * 3.876 + 4 * 6.004 + 5 * 3.268$$
$$= 88.084\ [\text{um}^2]$$

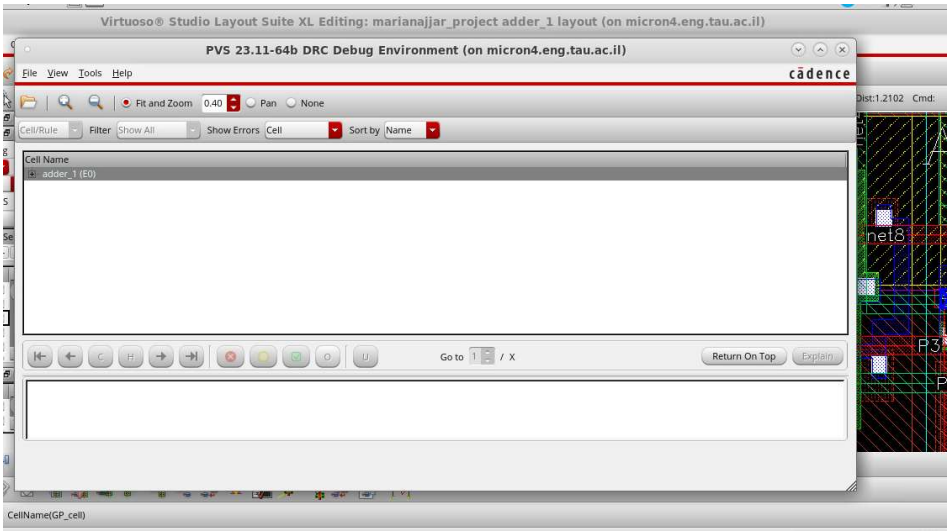
- The actual layout total area is: $S_{total} = 10.36 * 8.74 = 90.5464[\text{um}^2]$

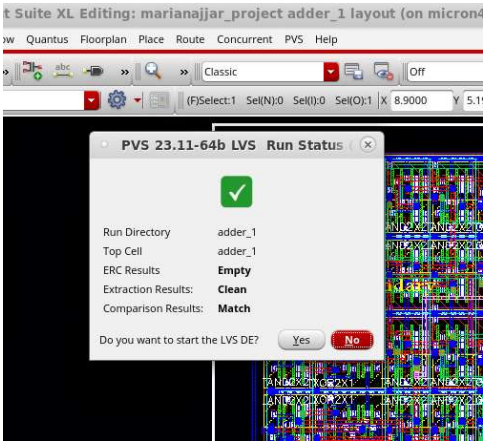
We can see that the actual area with only a 2.79% increase compared to the total area of all components, thus we met the limitations of area successfully.

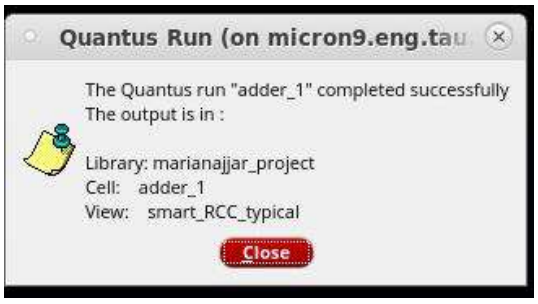
In order to achieve efficient routing results, we used up to 4 metal layers, and the most important data lines were assigned to metal 2 and metal 3.

Additionally, we performed DRC and LVS checks on our layout and the design was verified without any errors or warnings.

<u>QUANTUS, DRC, LVS:</u>

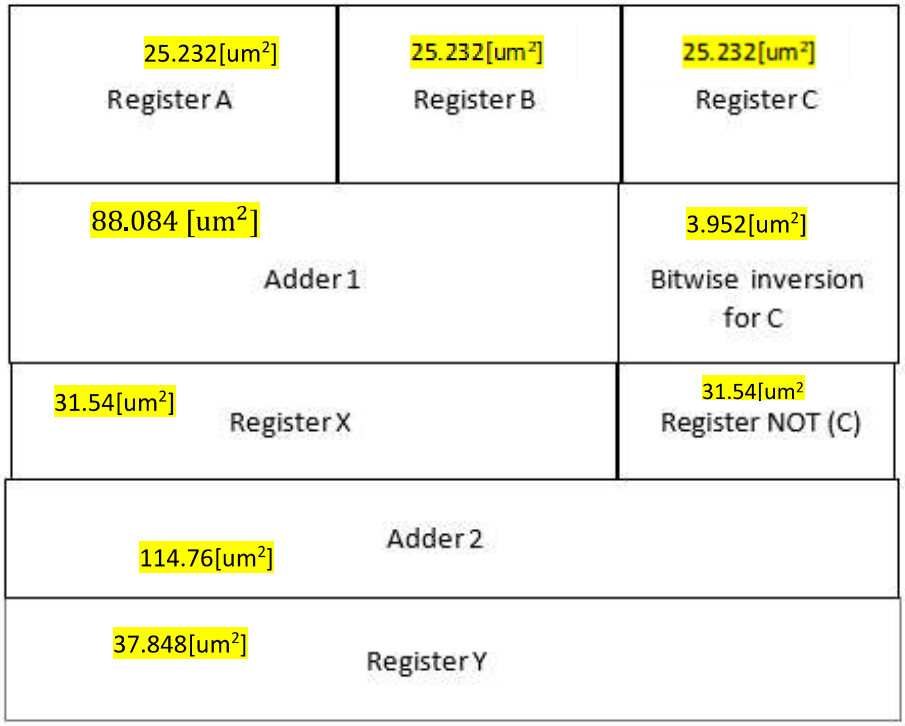






THE FINAL FLOOR PLAN:

The floor plan was designed in the first stage, However, we saw that the layout usage of adder 1 is very close to the total area of the components comprising the block (102.79% of the estimated area). Thus, we can refer to the total area of the blocks as approximately the sum of the areas of the components it consists of.



The total area is a sum of the areas of each component in the full ALU schematic.

From part 1 of the project, The estimated area of the full ALU (with the additional register):

$$S_{total-estimated} = 383.42[\mathrm{um}^2]$$