Introduction to VLSI – Final Project part 2

4-bit ALU – Implementation Stage

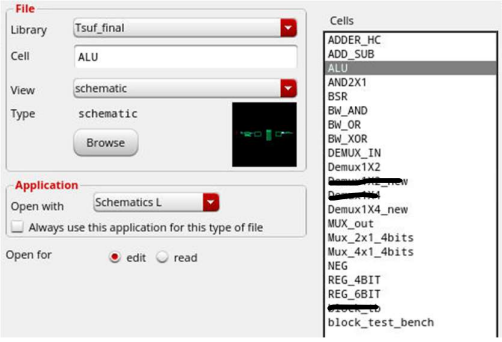
Roy Milshtein, (username – roymilshtein)

Tsuf Zamet, (username – tsufzamet)

Itay Havdala, (username – itayhavdala)

Inbal Shalit, (username – inbalshalit)

The virtuoso library that contains the project- Tsuf_final (Tsuf Zamet is its owner).
Names of our cells (the deleted ones are irrelevant):



*The cell names appear also next to their schematic/layout screenshots.

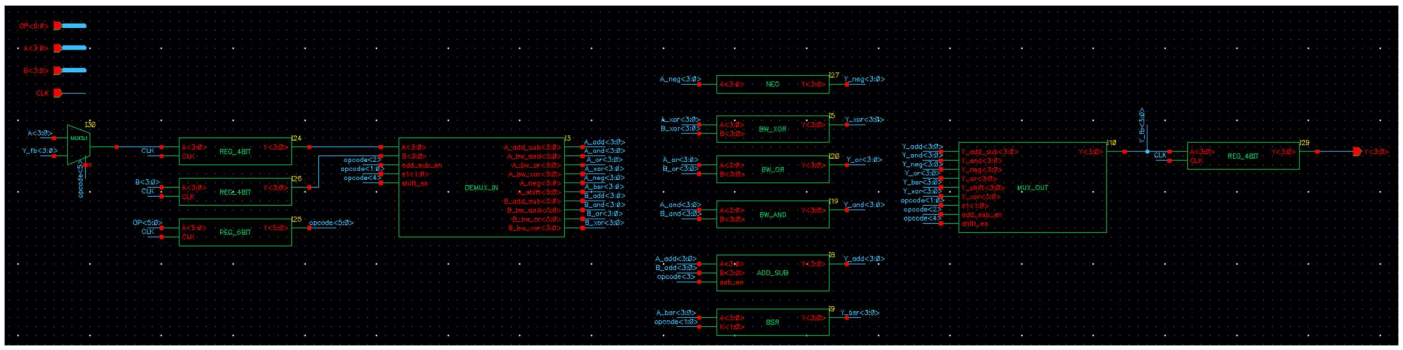
In the following report, we present the implementation stage of our project.
This report, combined with our planning stage report, fully describes our 4-bit ALU.

Content:

- Schematics:
  1. ALU schematic
  2. Block schematics in virtuoso
  3. Testbench
- Measurements:
  1. Logic tests (proof of good calculations)
  2. Delay measurements (including QRC measurements)
  3. Maximal clock frequency
- Floor plan:
  1. Final floor plan and size comparison
  2. Block layouts
  3. Full ALU layout
  4. DRC, LVS and QRC extraction
- Summary

Schematics implementations

Our ALU schematic:



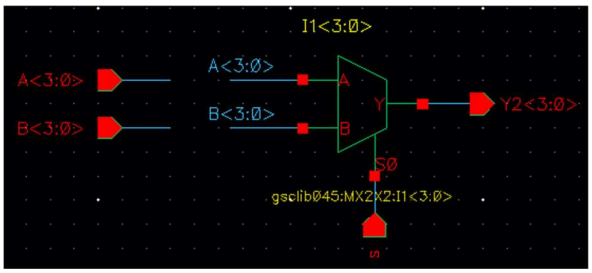
As described in the planning stage, we built our 4-bit ALU using smaller blocks.
Each block consists of basic cells from gsclib045 library, while some blocks were built using smaller blocks, according to our hierarchical design.

The ALU schematic design includes the following blocks (left to right):

- Feedback mux2:1
- 4-bit registers for inputs A and B and 6-bit register for the opcode
- Demux_IN
- Logic operations blocks (top to bottom) – Negation, bitwise xor, bitwise or, bitwise and, add_sub and right shift.
- Mux_out
- 4-bit register for output Y.

In the following section we will present our schematic implementation in virtuoso for each one of these blocks.

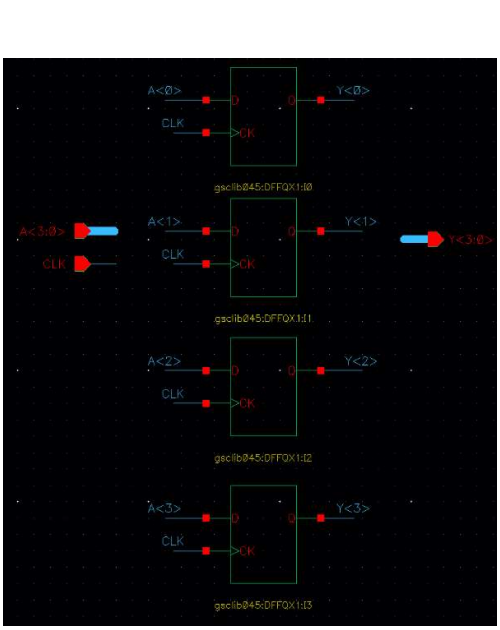
1. <u>Feedback mux</u>
   Cell name: Mux_2x1_4bits
   Getting two 4-bit inputs and a select bit (feedback enable, which is the opcode's MSB), and sets the output to be either the user's input A, or the 4-bit result of last ALU's operation, according to the select bit.
   Schematic:

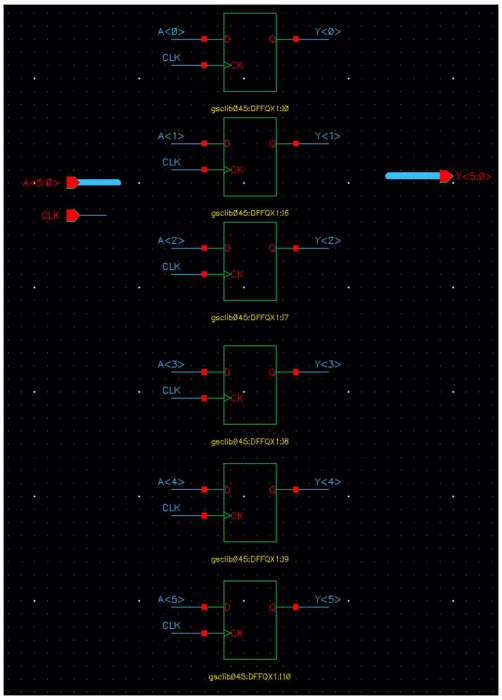


   Denote that this Mux 2:1 is made out of 4 Mux2x2 cells.

2. Input registers

   Cell names: REG_4BIT, REG_6BIT

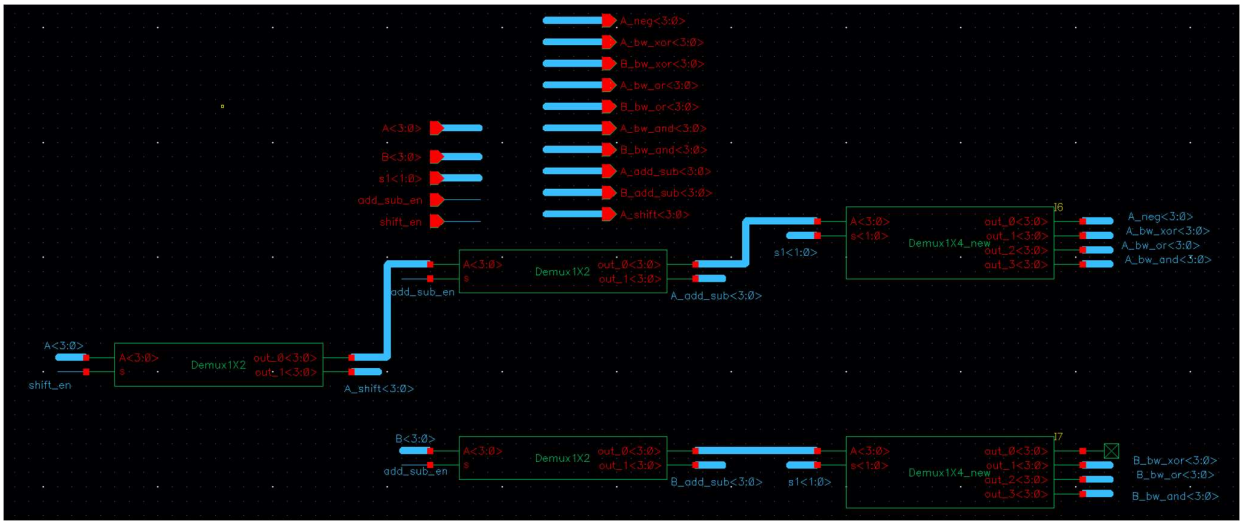




   The registers are constructed from 4 (or 6) DFF, same as in the planning stage.

3. Demux_in

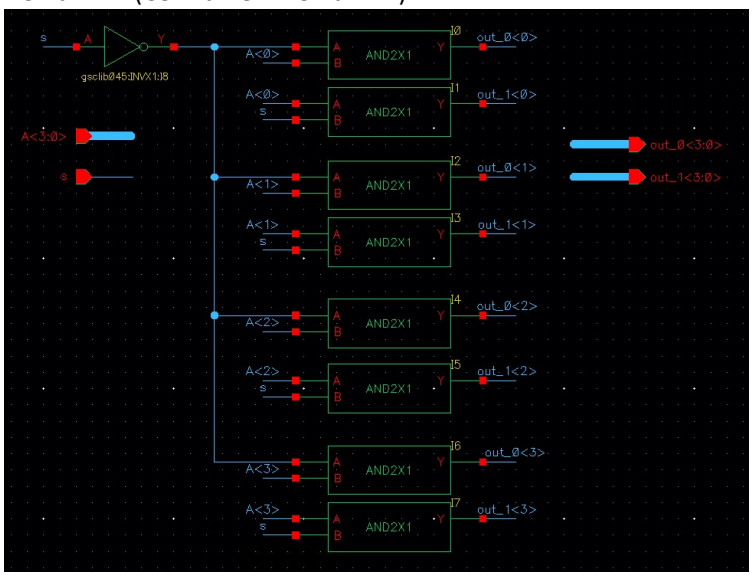
   Cell name: DEMUX_IN



   Constructed from 3 sub-blocks named Demux 1X2 and 2 sub blocks named Demux 1X4_new. One of the outputs of the lower Demux1X4 is not in use, therefore connected to noConn cell.

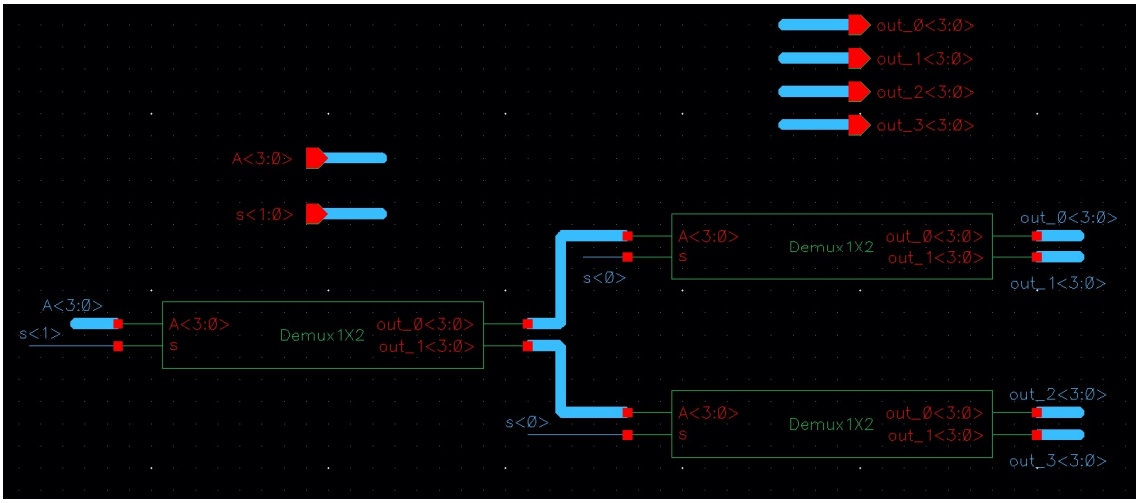
   Denote that this implementation is the same as planned in planning stage.

Demux1X2 (Cell name – Demux1x2):



Denote that the AND2X1 gates are a modification of the original basic cell of the same name from gsclib045 library. The modifications were done according to the project's instructions. We used this modified cell in all our blocks.

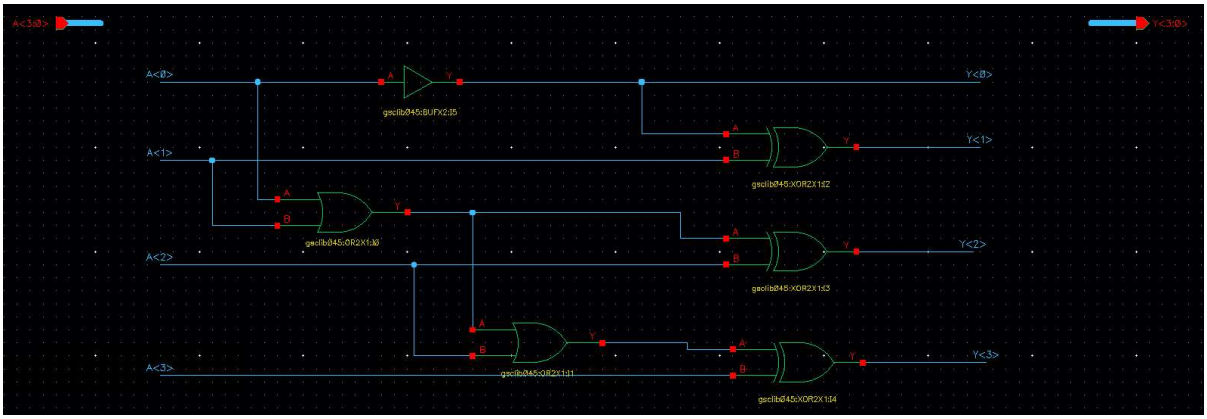
Demux1X4 (Cell name – Demux1x4_new):



This demux is made from 3 smaller blocks of demux1X2.

4. Logic operations blocks

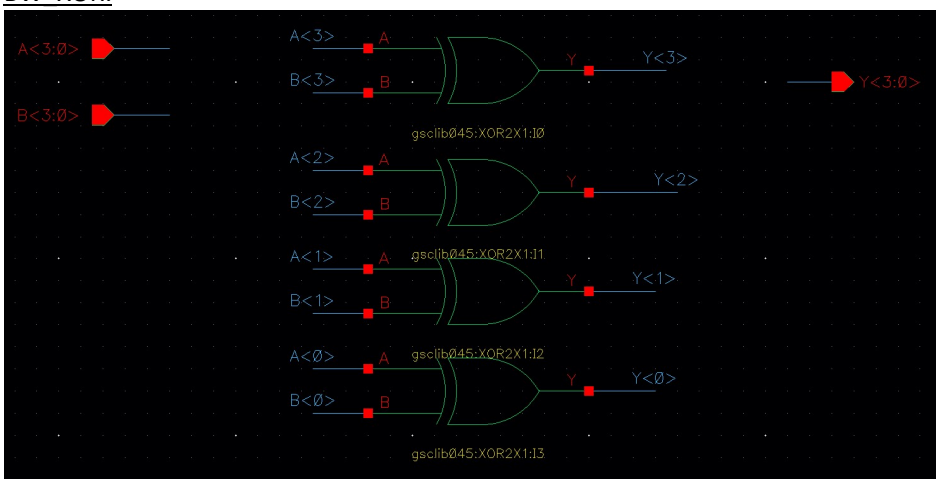
Neg:



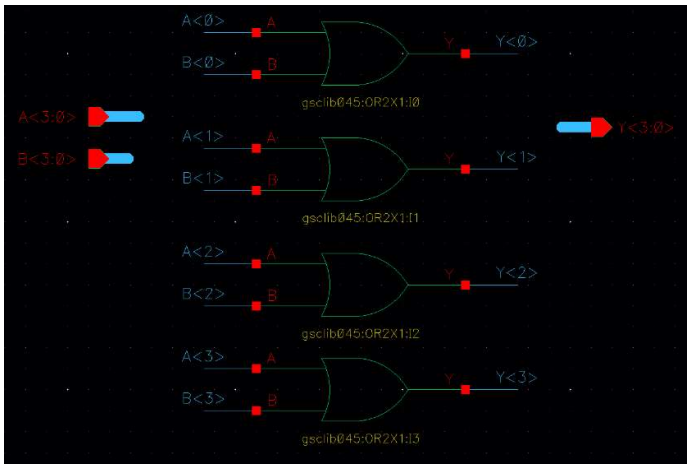
The implementation of neg is very similar to the one in the planning stage. The only difference is that we had to add a buffer for the Y<0> output to differentiate it from the input A<0> (as suggested by the T.A.).

Next, we have the bitwise operations: XOR, OR, AND. Each constructed using the relevant basic 2X1 logic cells.

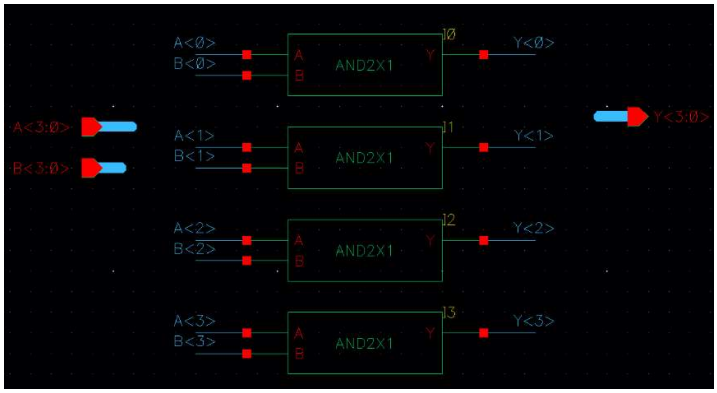
BW_XOR:



BW_OR:



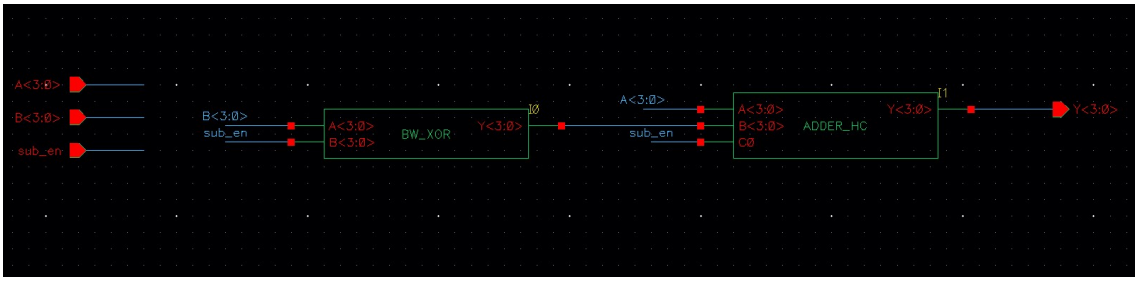
BW_AND:



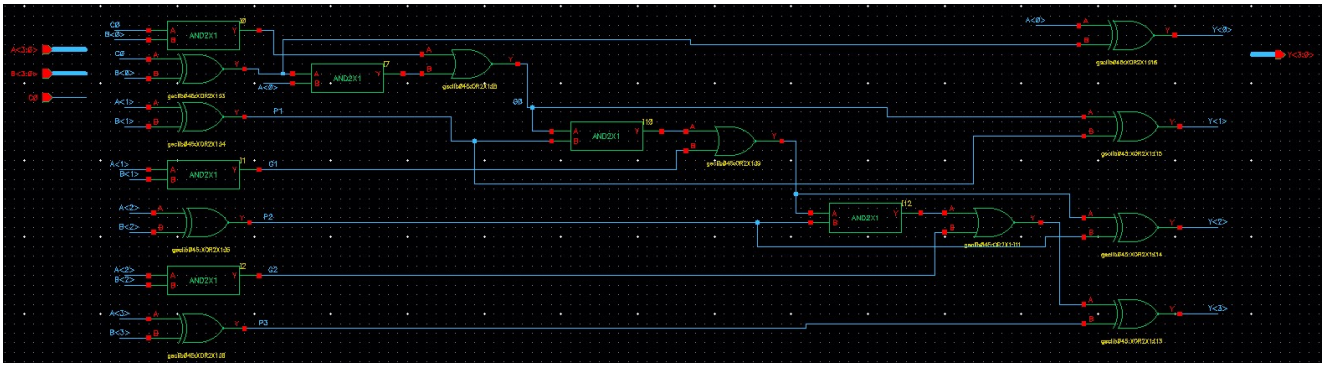
* As explained above – the AND cells used here are the modified ones, and not the ones from gsclib045 library.

ADD_SUB:

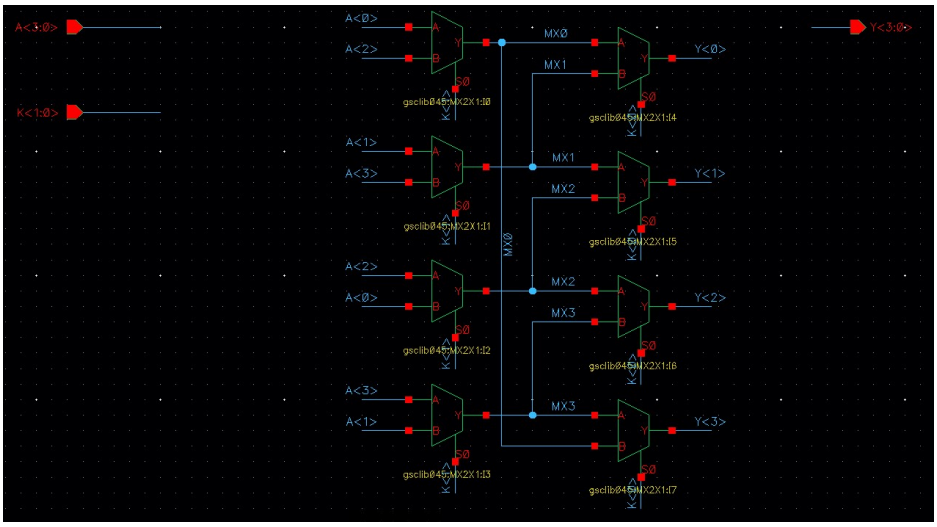
This block consists of a BW_XOR instance (schematic is shown above) and a block named ADDER_HC, which is our implementation on the Han-Carlson adder algorithm.
Using bitwise XOR between B input and the sub_enable bit allows us to choose between B and $\bar{B}$ as second input for our adder. Using the sub_enable bit as carry- in preforms the +1 addition required to negate B when preforming substruction.
Therefore, the ADD_SUB block enables the ability to choose whether to preform addition or substruction between A and B inputs.



ADDER_HC:



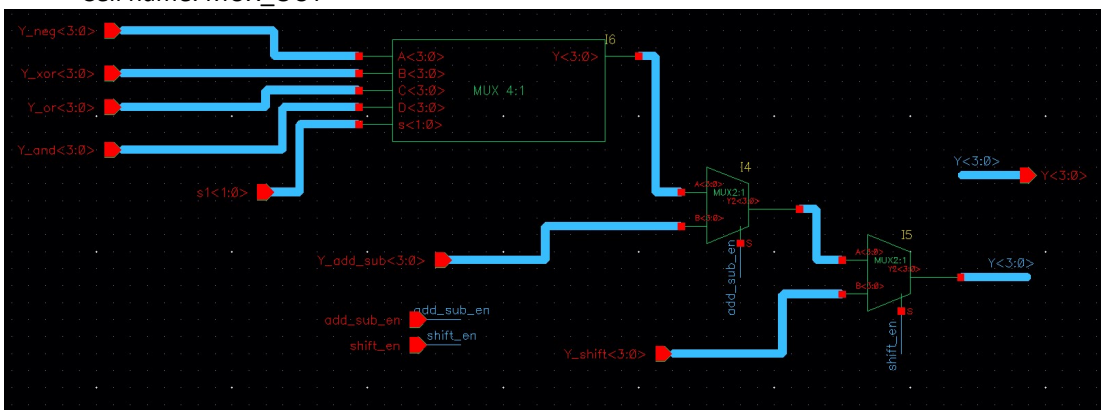
BSR (barrel shift right):



The barrel shifter is implemented the same as the planning stage, using 8 mux2X1 gates.

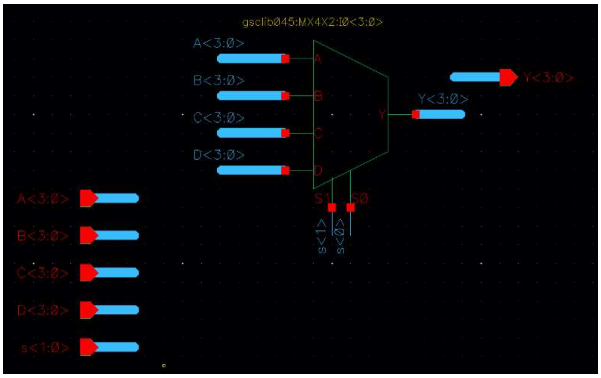
5. Mux_out

   Cell name: MUX_OUT



Constructed from a sub-block named Mux_4x1_4bits and 2 sub-blocks named Mux_2x1_4bits (same one we used in the feedback mux).

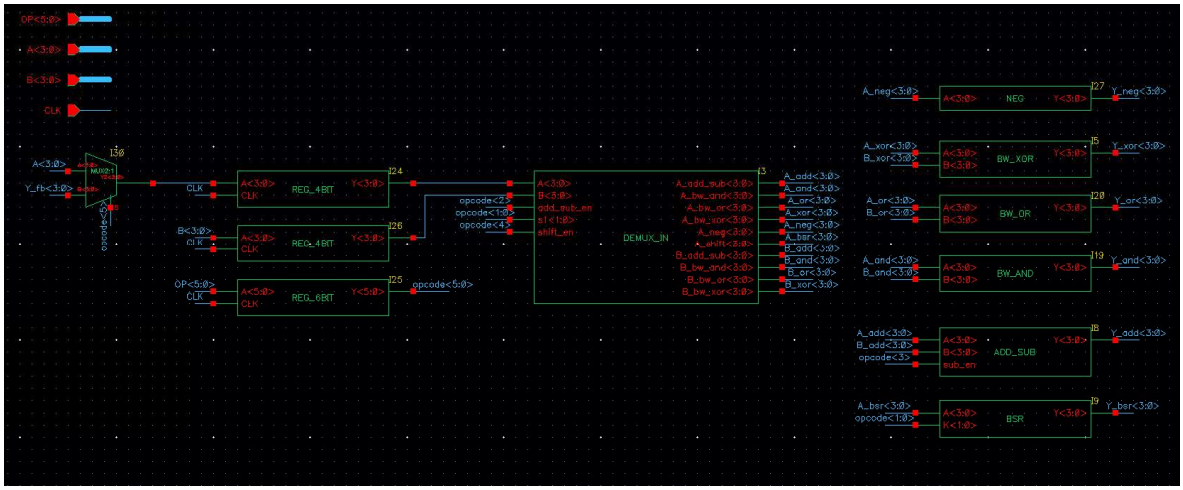
This is the implementation for a 4X1 mux of 4 bits (built from 4 MUX4X2 basic cells):

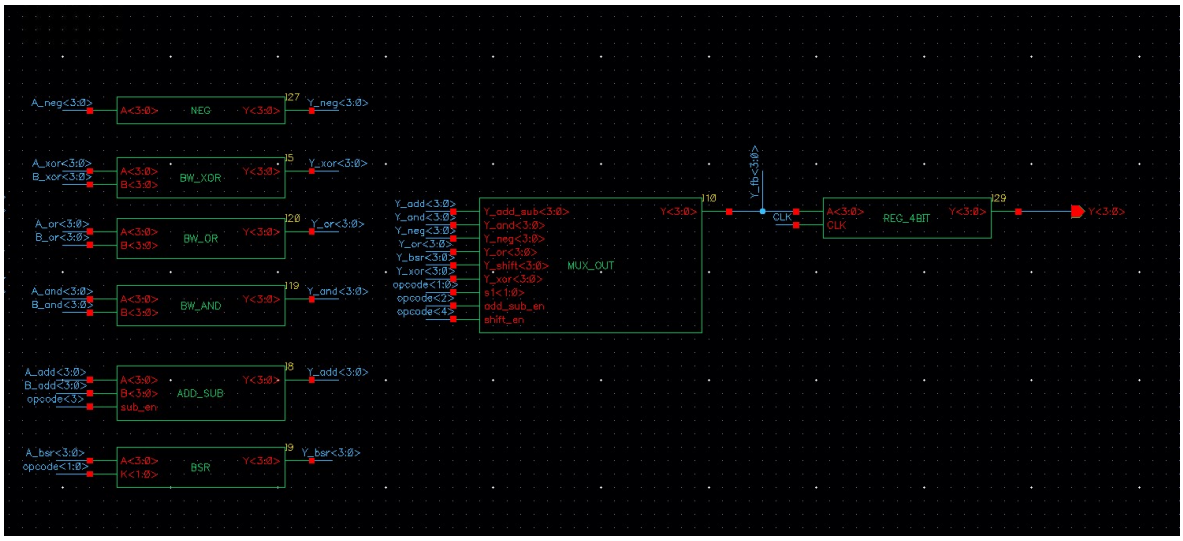


6. Output register

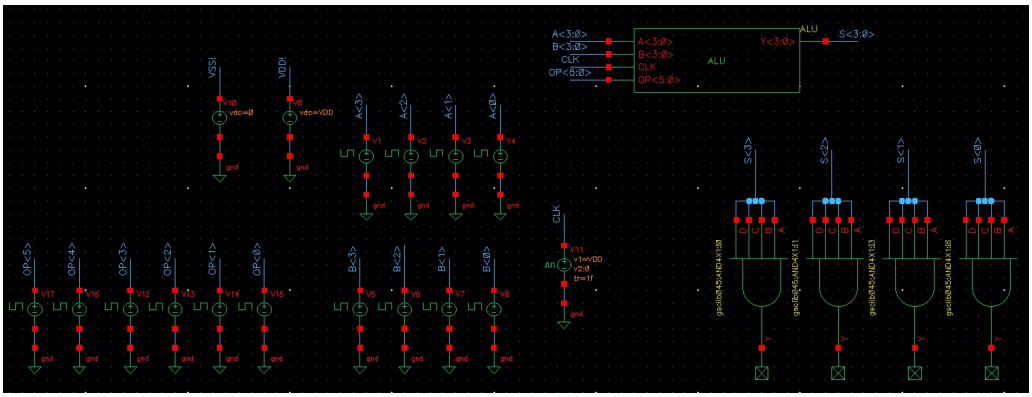
   We used the REG_4bit for the output register (schematic above in the input registers section).

In order to make the inputs and outputs to each block clearer, here are some zoom-in photos of our blocks inside the ALU schematic:





In order to simulate our ALU, we used the following testbench:



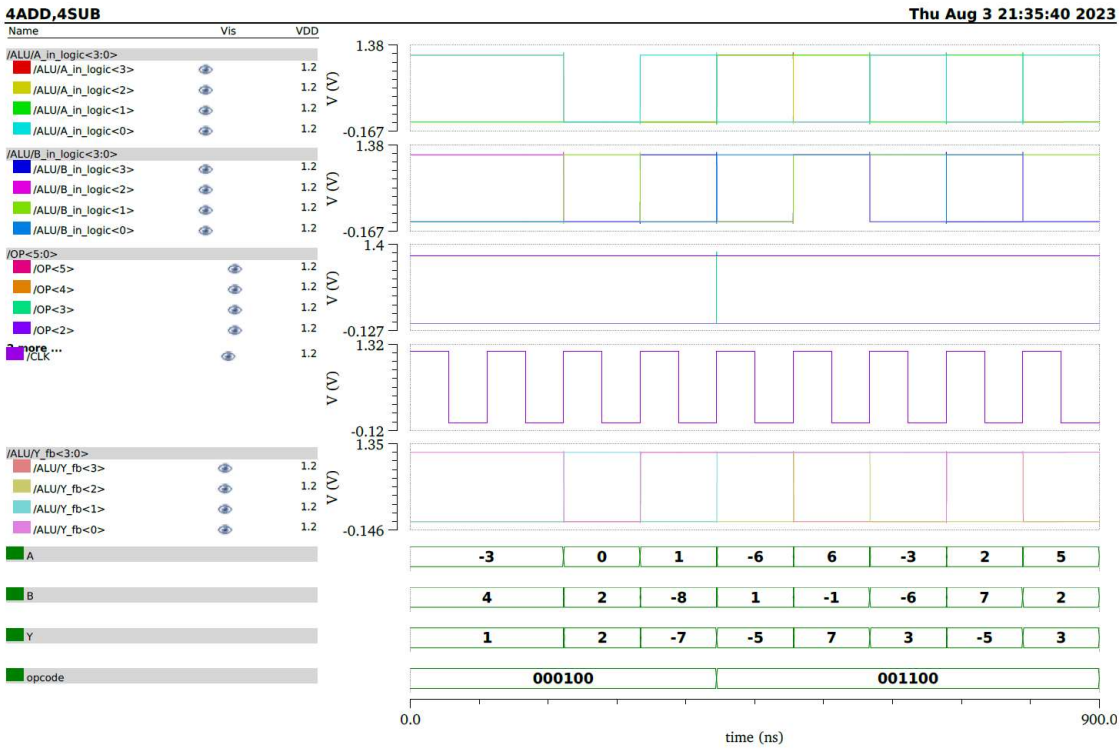
Denote that we chose to connect AND gates as load to each output bit, as the TA suggested.

Measurements

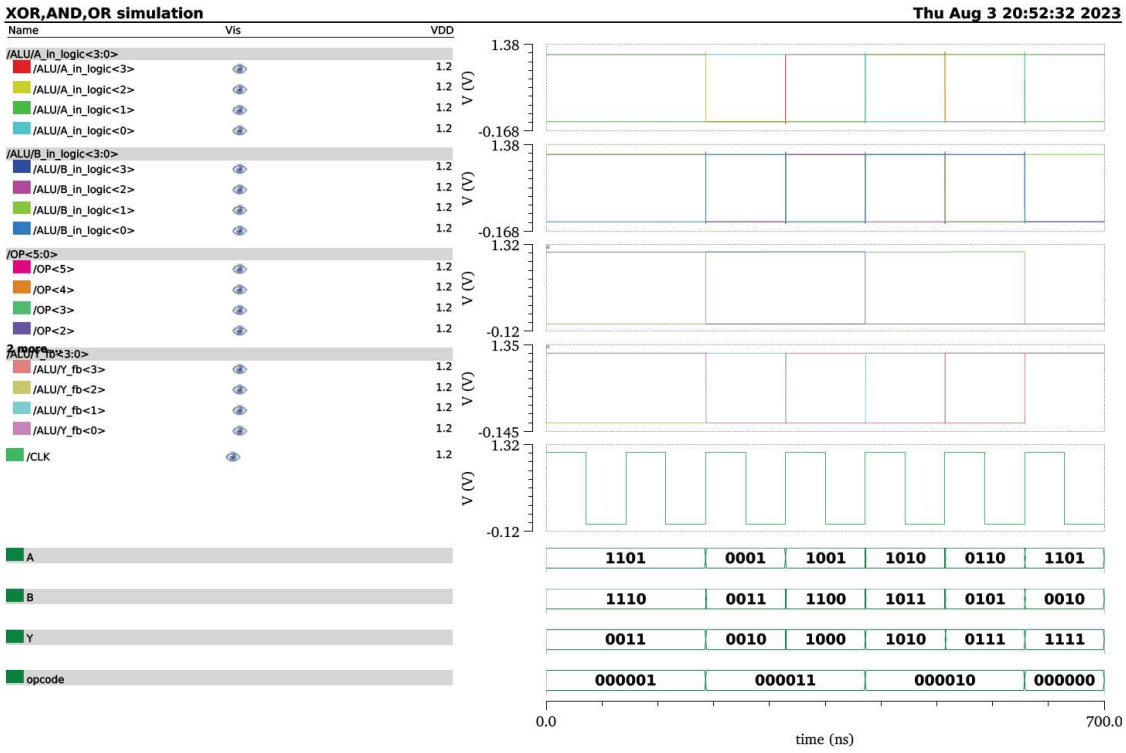
- **<u>Logic tests (proof of good calculations)</u>**
  We performed several logic simulations to measure the functionality of our ALU:

  <u>Note</u>: at the rise of the clock, it performs the operation of the previous clock (if changes during that clock), therefore we ignore first cycle for bits to pass the registers into the logic!

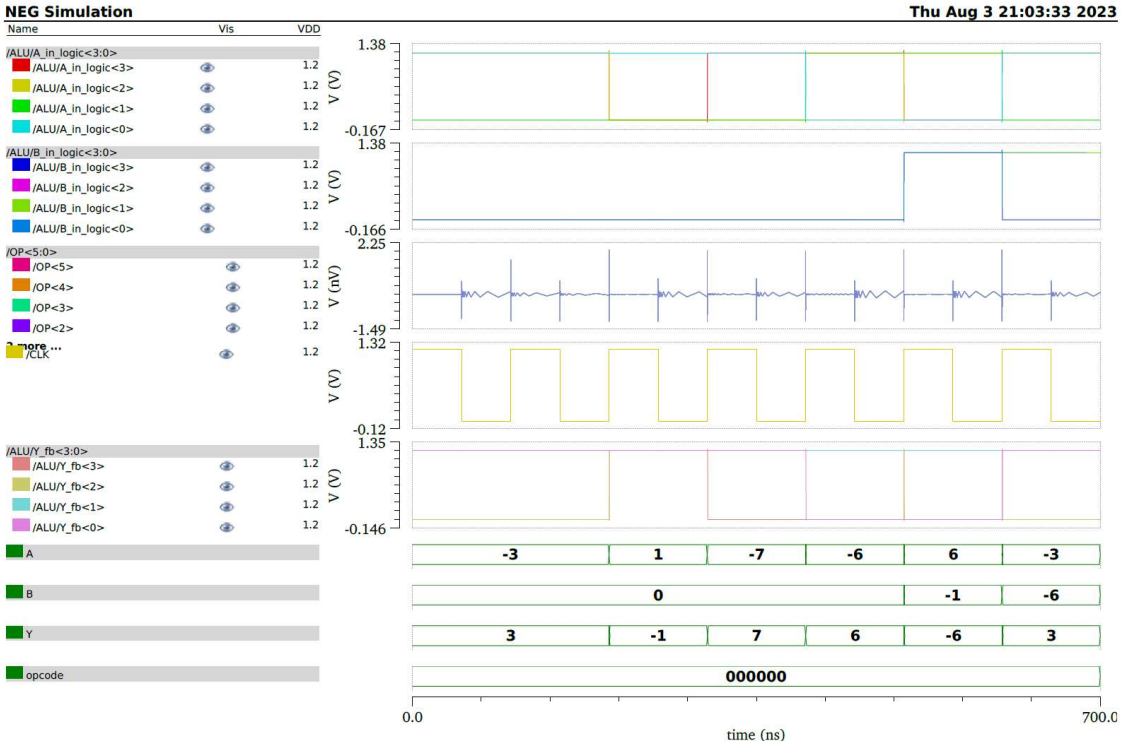
  1. <u>Add/Sub:</u>
     Preforming 4 add operations, then 4 sub operations:



     As we can see, the ALU performs the required operations for both positive and negative numbers in range.
  2. <u>Bitwise operations:</u>
     Performing 2 operations of each, in the order: XOR→AND→OR:



3. <u>Neg:</u>

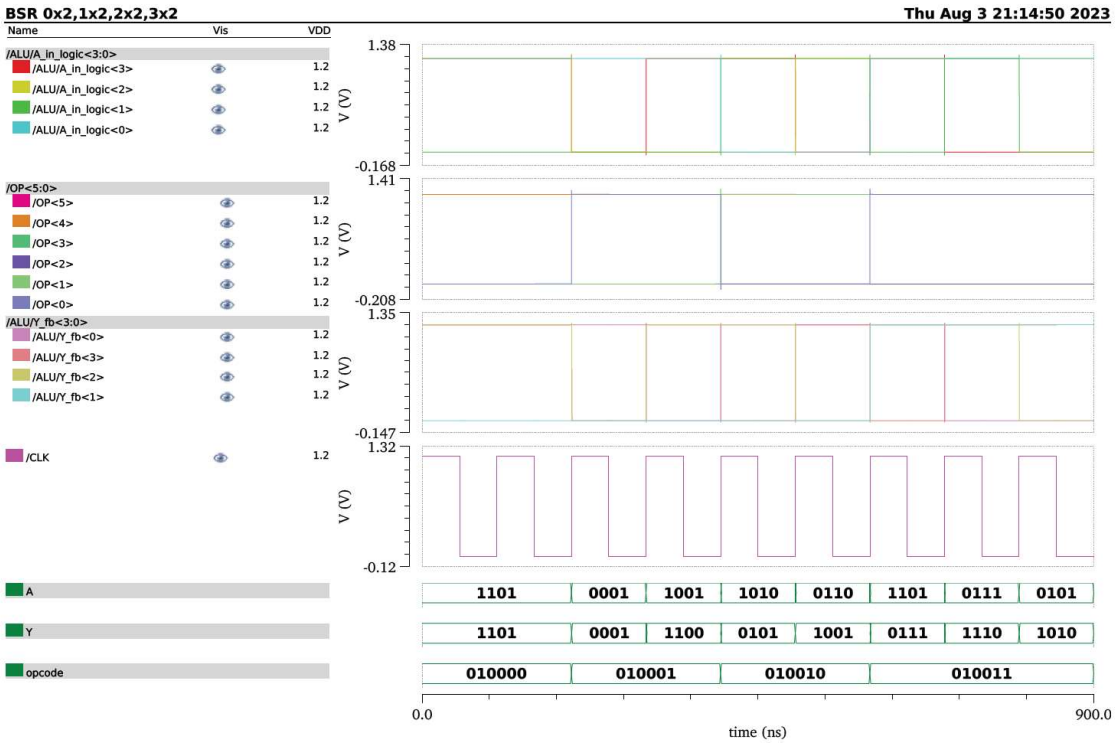


The signal B is irrelevant here. As we can see, even when its not zero, it doesn't affect the result.

The fluctuations seen in the opcode are on the scale of nanovolts and are also not affecting are results.

4. Barrel shift:

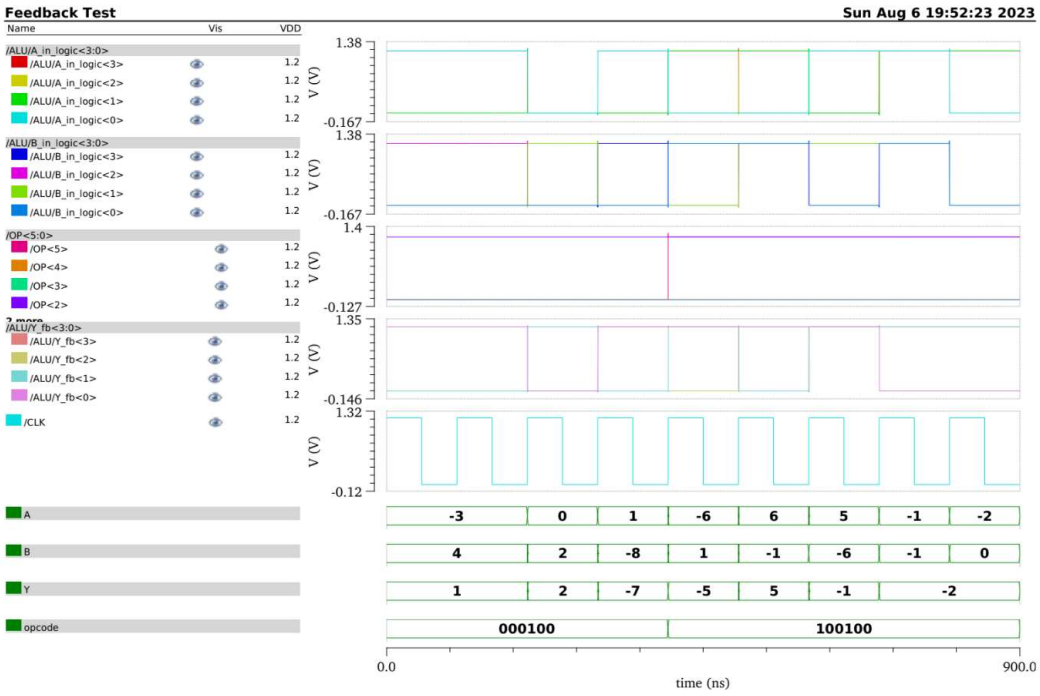
   2 of each shift: BSR by 0(same as 4) twice, then by 1 twice, by 2 twice, and by 3 twice.



5. Feedback:

   For feedback we set the MSB to be “1”.

   The following simulation performs the operation ADD, and the feedback turns on after a few cycles:



   As we can see, after “turning on” the feedback, the input A is equal to the output Y of the previous cycle!

Note: there is another clock cycle delay between the feedback and update in the value of A due to the extra register in the data path.

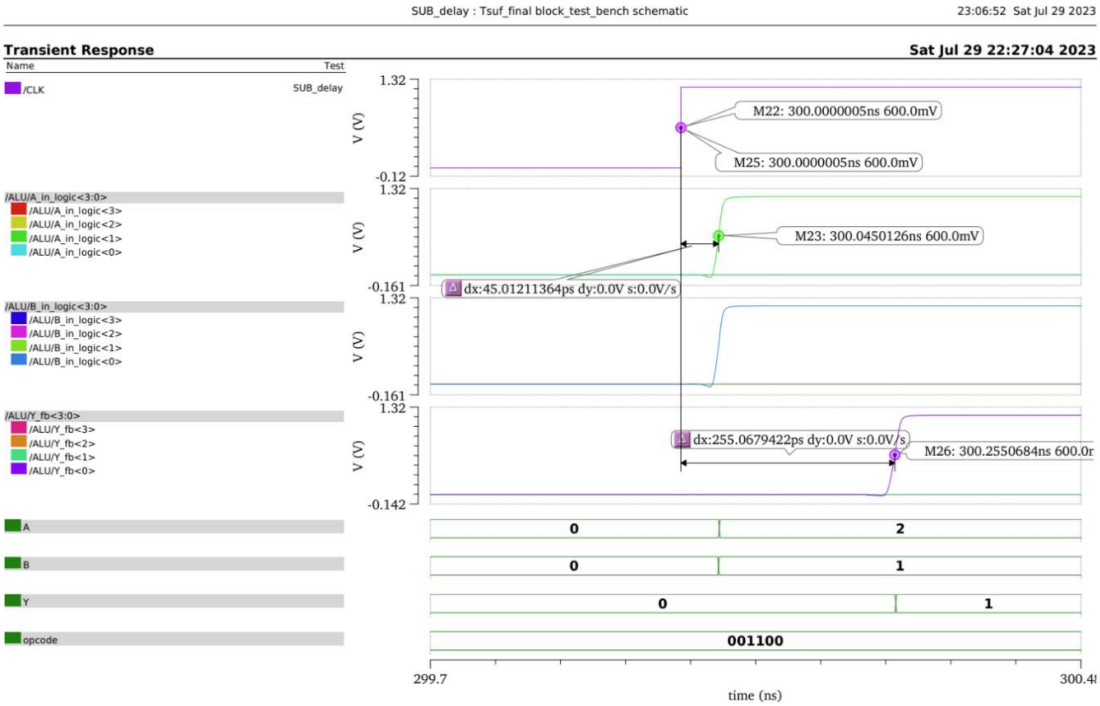
- **Delay measurements**

  This section includes simulations showing what is the critical path of our ALU and the delay of this path, for different supply voltages (with and without parasitics).
  First, in order to find what is the critical path, we checked 3 different ALU operations, which we suspected might be the slowest.
  According to our logic design, we suspected that the longest path would go through one of the following blocks:
  - ADD_SUB (preforming substruction)
  - BSR (shift operation)
  - Neg

  In order to find out which one is the slowest, we used our ALU test bench (schematic shown in previous section) with the relevant opcode for each operation.
  We measured the time between half-way clock rise to the output change (till voltage change of 50%). Input and output signals were taken from between the registers, meaning, A,B and the opcode taken from the output of their registers, and Y was taken as the output of the Mux_out block.
  By measuring the delay this way, the time it takes the data to go through the input registers is also considered ($t_{CQ,input\ regs}$). This is meant to help us in calculations of the maximal frequency that our ALU can operate in.
  The reason we suspected the negation block is that its input must go through three demux gates before getting inside the operation block, and three MUX gates before reaching to the output.
  Denote that the inputs for the bitwise operations blocks (bitwise XOR, bitwise OR and bitwise AND) also get to those blocks after 3 demux gates, but those operations are much simpler and therefore faster.

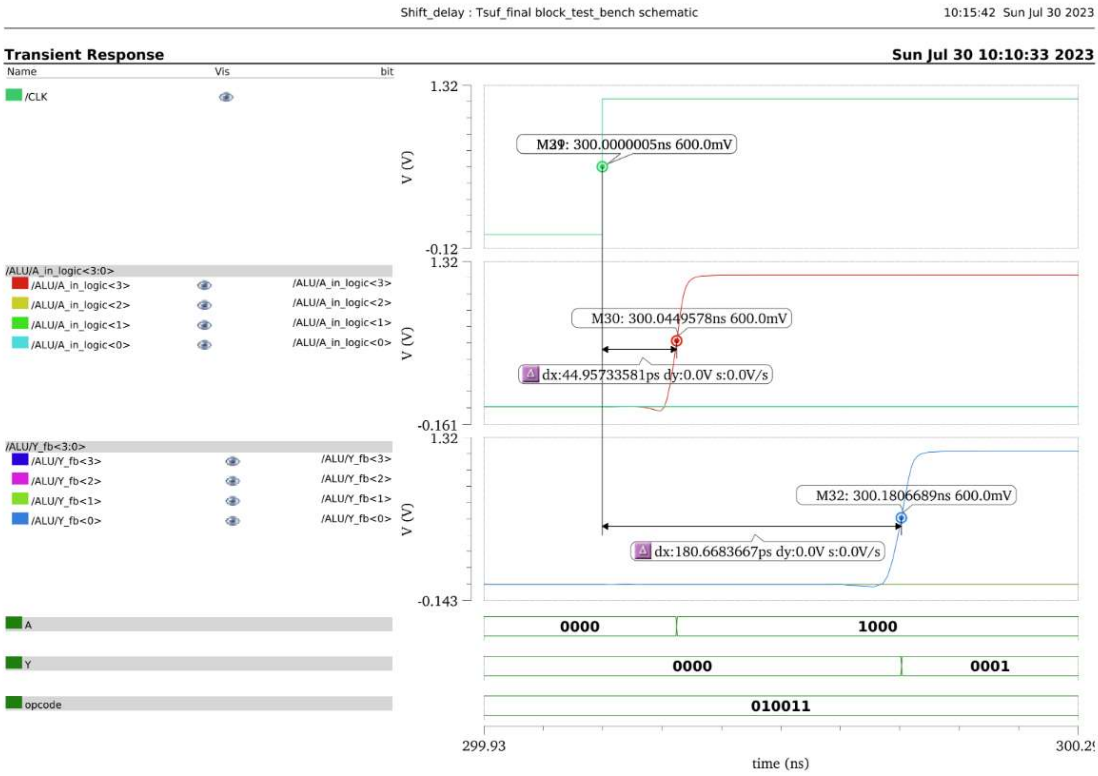
Here are the simulations for those three critical path candidates, all preformed using 1.2V supply voltage and no parasitics:

- ADD_SUB
  Denote that the relevant opcode for subtraction is 001100 as chosen in the planning stage.



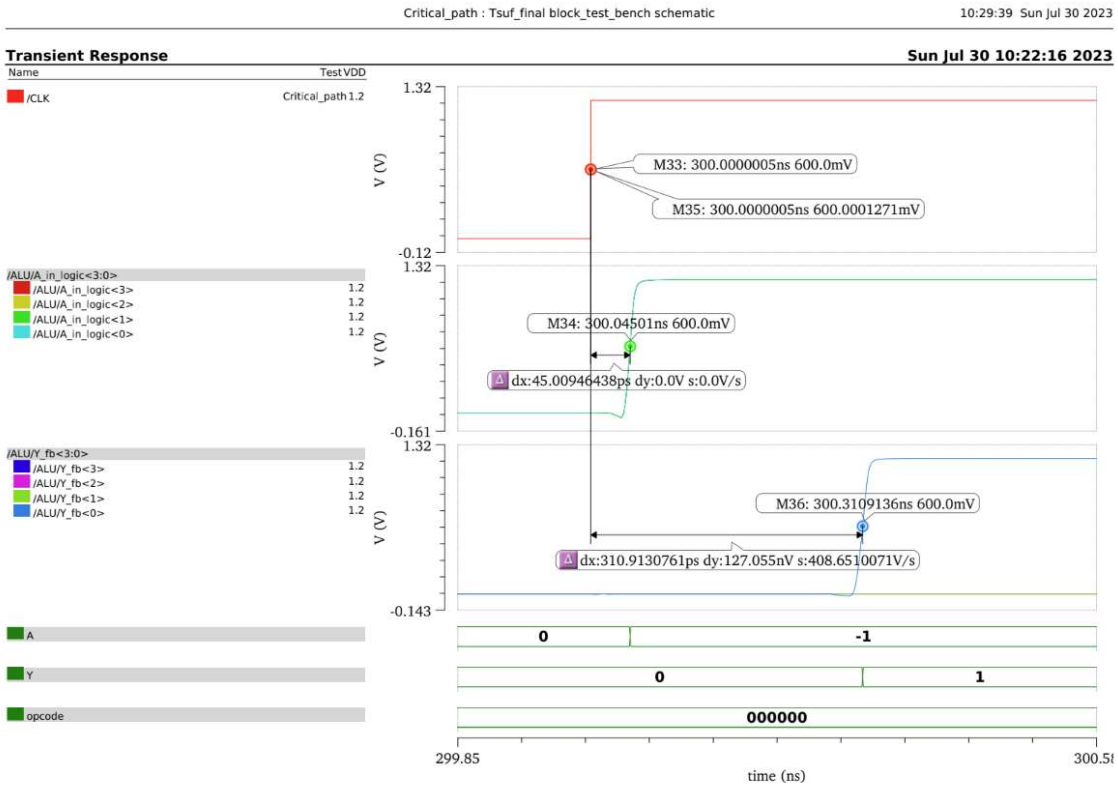
The delay of this path is $255.068\ ps$, as shown above.
We also marked the input registers delay: $t_{CQ} = 45.01ps$ (minor changes between inputs and operations).

- BSR



We checked for the case of shifting by 3, so the relevant opcode is 010011.
The delay of this path is $180.67\ ps$, as shown in the simulation above.

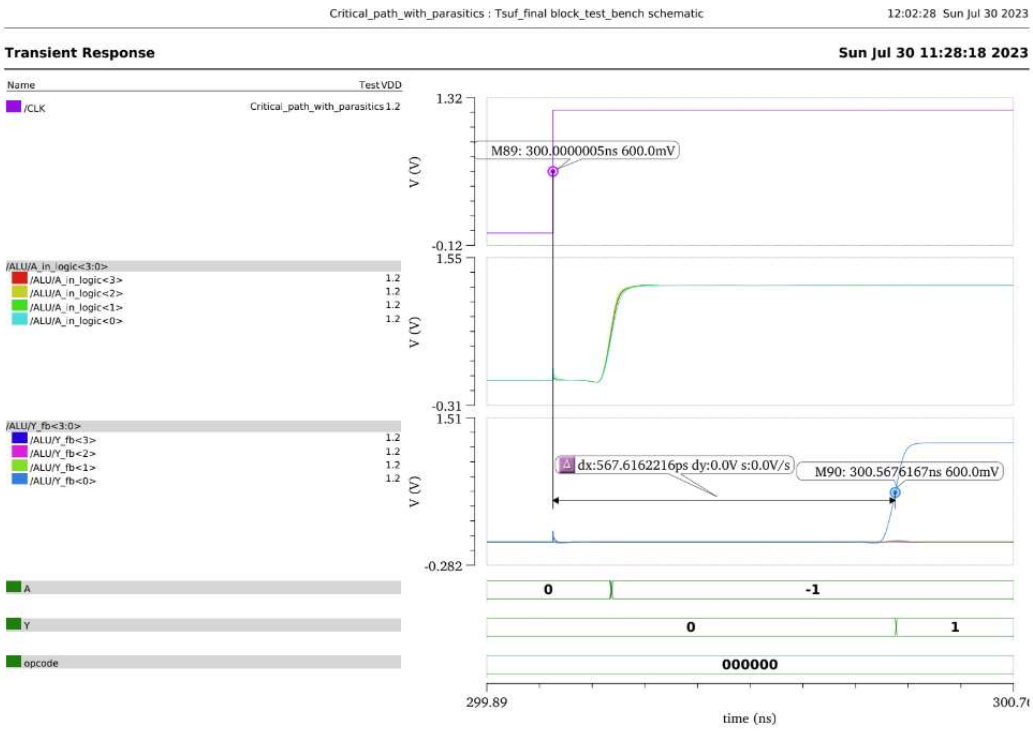
- Neg



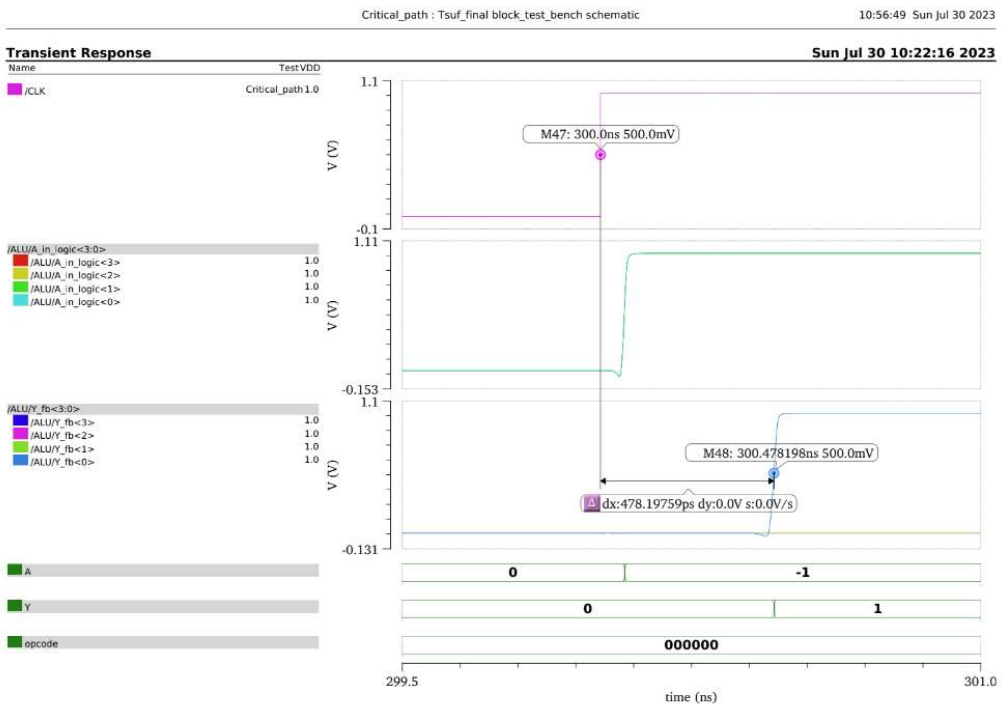
The relevant opcode is 000000.
The delay of this path is $310.91\ ps$, as shown in the simulation above.

Therefore, <u>the critical path is</u> A<3:0>→Demux_in→Neg→Mux_out→Y_fb<3:0> , while A<3:0> is the output of the input register A and Y_fb is the output of the Mux_out block (which is also the input of the output register and one of the inputs of the feedback mux).
In the following prints, there are simulation of the critical path for supply voltages of $900mV, 1V\ and\ 1.2V$, with and without parasitics.

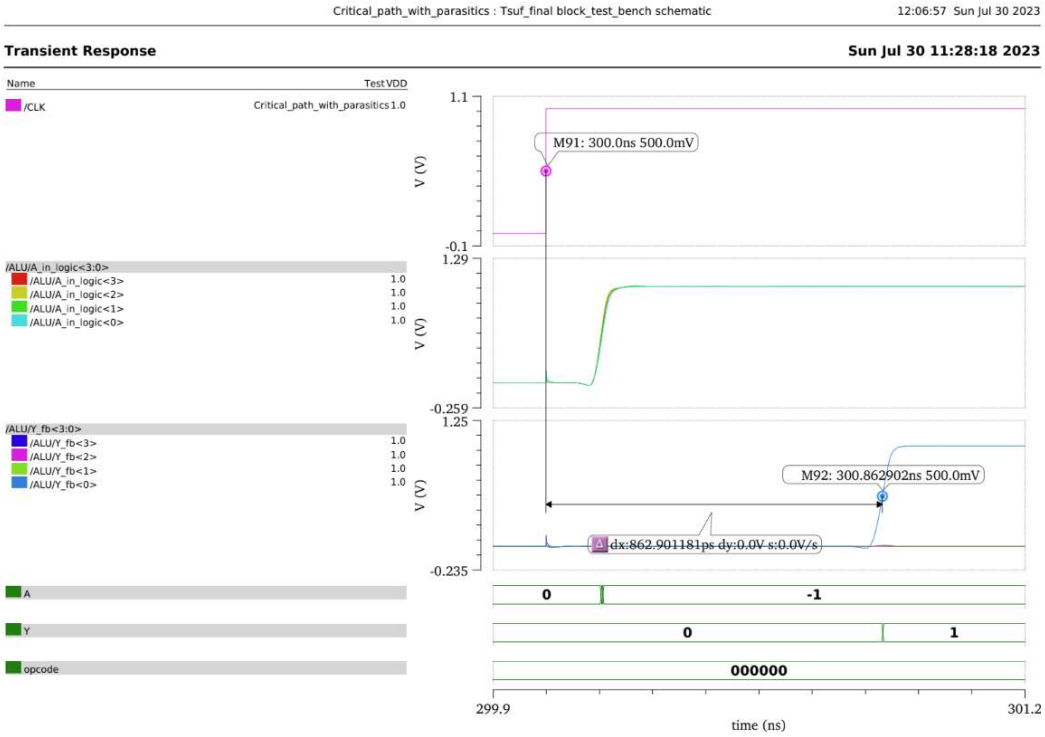
Critical path for supply voltage of $1.2V$ <u>with parasitics</u> (without parasitics is shown above):



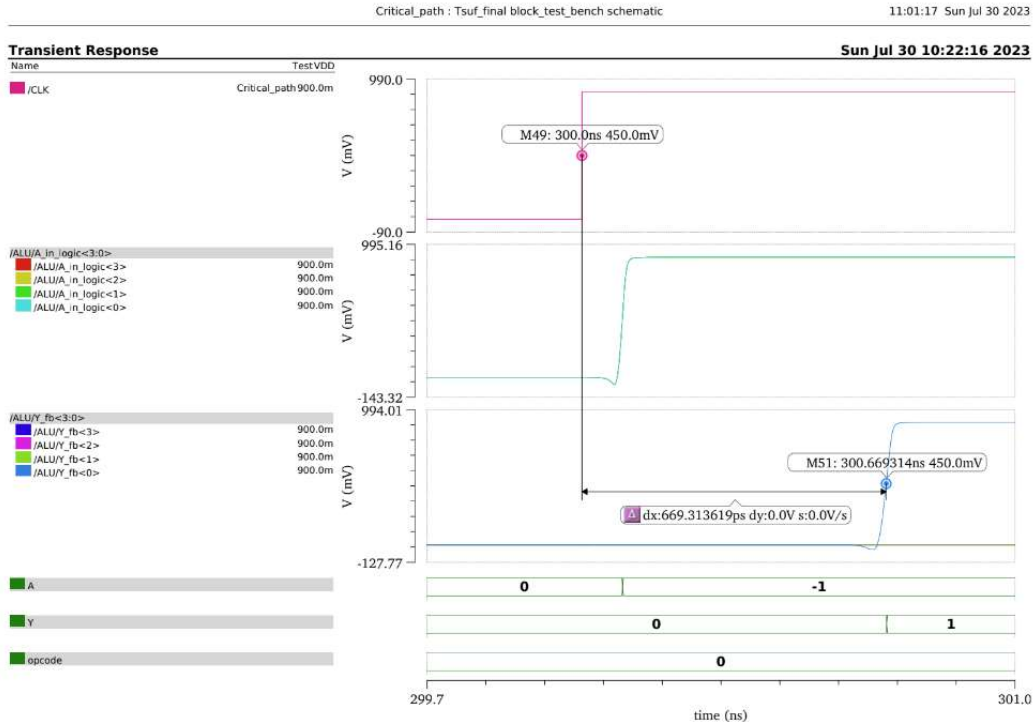
Critical path for supply voltage of $1V$ <u>without parasitics</u>:



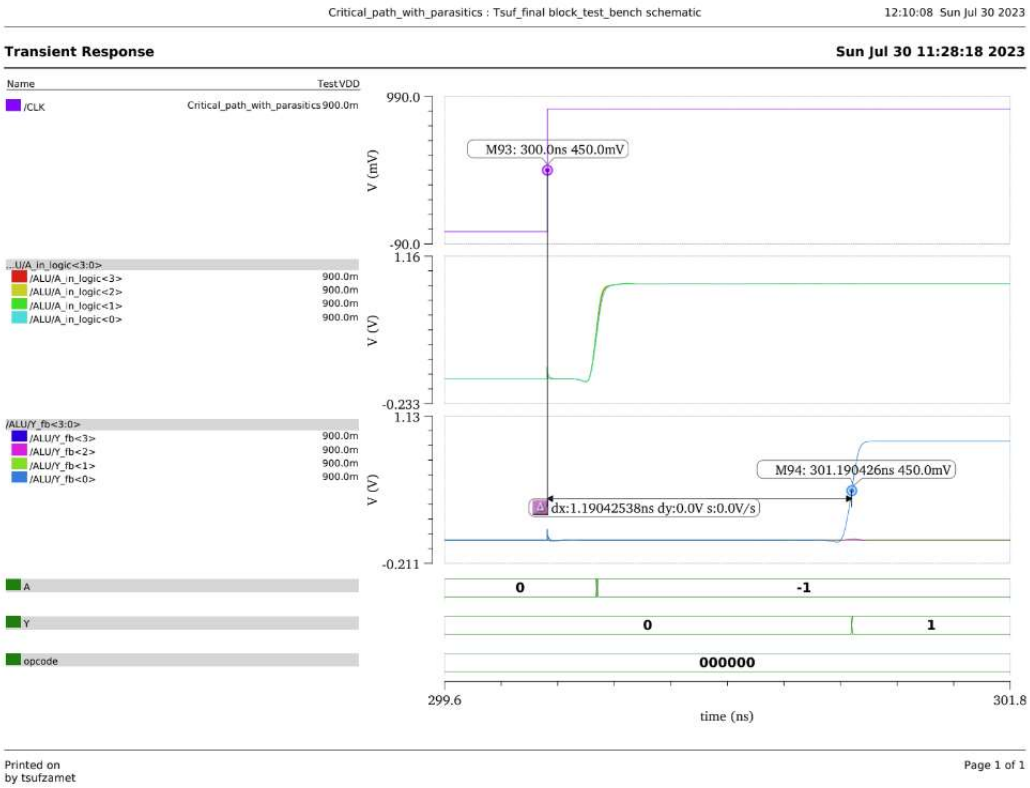
Critical path for supply voltage of $1V$ <u>with parasitics</u>:



Critical path for supply voltage of $900mV$ <u>without parasitics:</u>

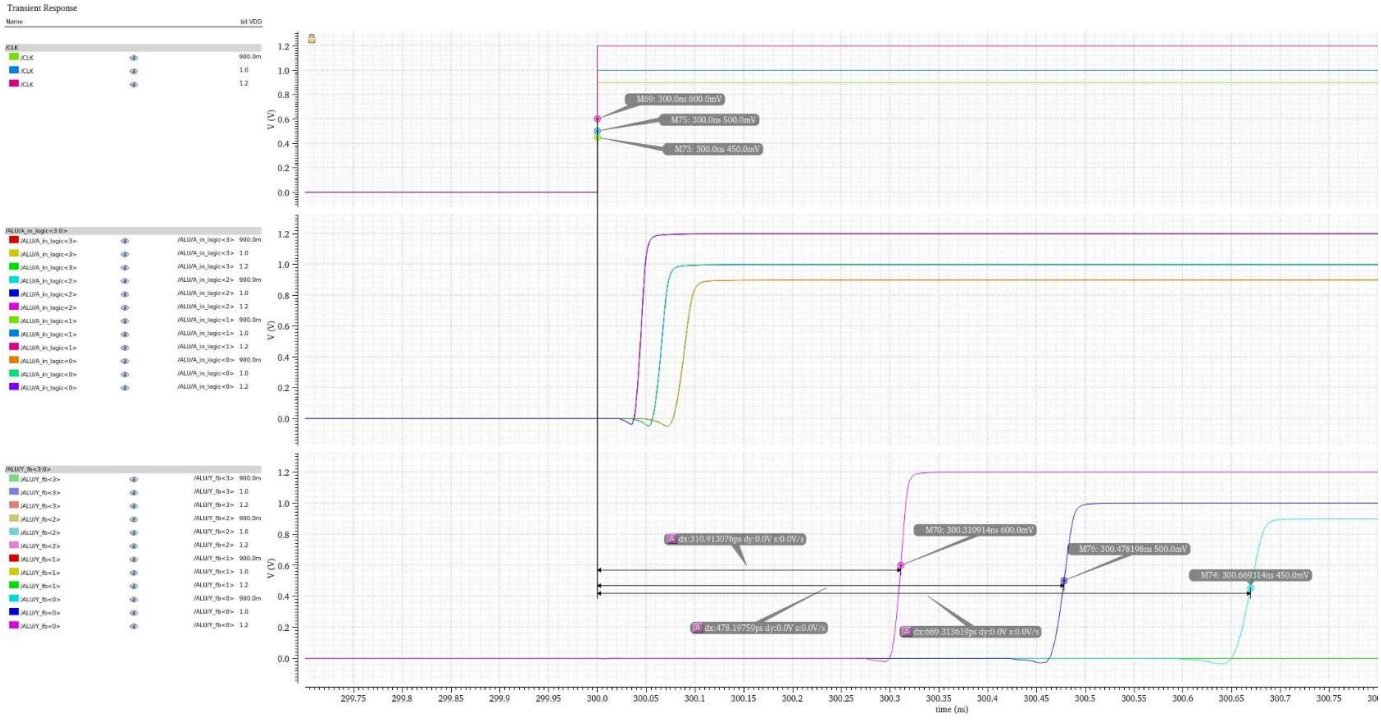


Critical path for supply voltage of $900mV$ <u>with parasitics:</u>

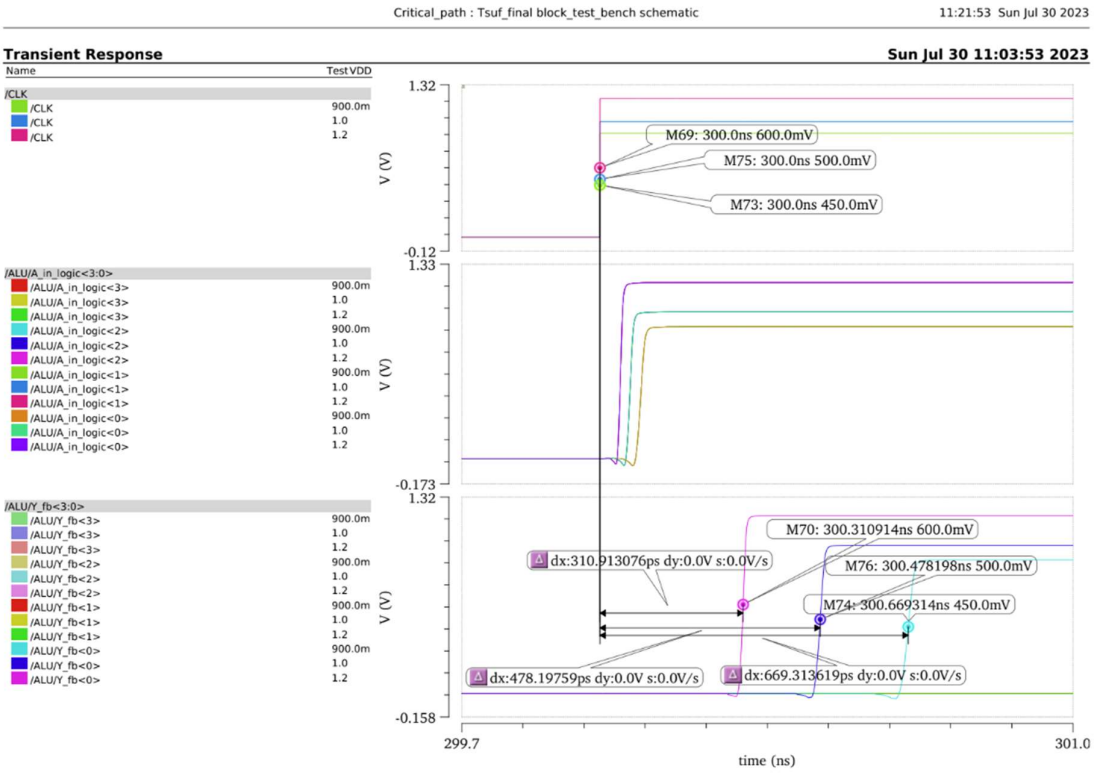


It's clear to see from simulation that the delay without parasitics is smaller than with them (as expected), and that the lower the supply voltage is – the longer the delay time (lower supply voltage implies lower current for charging).

Here's a simulation print showing the differences between the supply voltages, as described above (without parasitics):



Another version that might be more readable:



The following table summarizes the delay times from the simulations of the critical path:

	$1.2V$	$1V$	$900mV$
Critical path delay <u>without</u> parasitics	$310.9131\ ps$	$478.1976\ ps$	$669.3136\ ps$
Critical path delay <u>with</u> parasitics	$567.6162\ ps$	$862.9011\ ps$	$1.1904\ ns$

Denote that those times are the <u>longest</u> times (for each case) it takes for data coming from the input registers to go through the logic part of the ALU and arrive to the 'D' input of the output registers.

- **<u>Maximal clock frequency</u>**
  Based on the results from previous section (critical path delay measurements), in order to find what is the maximal clock frequency for our ALU to operate correctly, we will need to add to our considerations also the t_{setup} time of the output registers.
  As we learned in class, the minimal clock period is:
  $$T_{clk,min} = t_{CQ} + t_{pd,logic} + t_{setup}$$
  While t_{CQ} is the time it takes for data to propagate through the input registers after each clock rise, $t_{pd,logic}$ is the logic delay of the critical path (from between the registers) and t_{setup} is the time we need the data entering the output register to be ready before the next rising edge of the clock.
  As we explained in the previous section, our critical path measurements include the first two components, $t_{CQ}\ and\ t_{pd,logic}$, so we will only need to add t_{setup} to the results we already have.

In order to estimate t_{setup} , we used the 4-bit shift register we built in recitation 11.
We fixed the skew variable to 0, the clock period to 100n and the t_delay_logic variable to be less than one clock period (95n). Then, we preformed several sweep simulations over the t_delay_data_clock variable in order to find the value that causes fail.
We used the same expression as we used in class and in HW3, with 1% tolerance instead of 10%.
First sweep results:

Point	t_delay_data_clk	Pass/Fail	/D<0>	/Q<3>	/CLK	/Q<0>	/D<1>	/Q<1>	/D<2>	/Q<2>	/D<3>	ng" ?edge2 "ris
1	-20p	pass										302n
2	-17.78p	pass										302n
3	-15.56p	pass										302n
4	-13.33p	pass										302n
5	-11.11p	pass										302n
6	-8.889p	pass										302n
7	-6.667p	pass										302n
8	-4.444p	pass										302n
9	-2.222p	fail										401n
10	0	fail										401n

Second sweep:

Point	Corner	_delay_data_clk	Pass/Fail	/D<0>	/Q<3>	/CLK	/Q<0>	/D<1>	/Q<1>	/D<2>	/Q<2>	/D<3>	ng" ?edge2 "ris
1	nom	-5p	pass										302n
2	nom	-4.444p	pass										302n
3	nom	-3.889p	pass										302n
4	nom	-3.333p	fail										401n
5	nom	-2.778p	fail										401n
6	nom	-2.222p	fail										401n
7	nom	-1.667p	fail										401n
8	nom	-1.111p	fail										401n
9	nom	-555.6f	fail										401n
10	nom	0	fail										401n

Third sweep:

20 rows

Point	Corner	_delay_data_clk	Pass/Fail	/D<0>	/Q<3>	/CLK	/Q<0>	/D<1>	/Q<1>	/D<2>	/Q<2>	/D<3>	ng" ?edge2 "ris
1	nom	-4p	pass										302n
2	nom	-3.789p	pass										302n
3	nom	-3.579p	fail										401n
4	nom	-3.368p	fail										401n
5	nom	-3.158p	fail										401n
6	nom	-2.947p	fail										401n
7	nom	-2.737p	fail										401n
8	nom	-2.526p	fail										401n
9	nom	-2.316p	fail										401n
10	nom	-2.105p	fail										401n
11	nom	-1.895p	fail										401n
12	nom	-1.684p	fail										401n
13	nom	-1.474p	fail										401n
14	nom	-1.263p	fail										401n
15	nom	-1.053p	fail										401n
16	nom	-842.1f	fail										401n
17	nom	-631.6f	fail										401n
18	nom	-421.1f	fail										401n
19	nom	-210.5f	fail										401n
20	nom	0	fail										401n

→ $t_{setup} \cong 3.7\ ps$
Note that $t_{setup}$is much smaller than the logic delay, so this estimation should be good enough in order to get idea of this the maximal frequency.

Now, the minimal clock period will be the sum of t_{setup} and $t_{critical\_path}$ calculated above for the critical path:

$$T_{min} = t_{setup\_reg} + t_{critical\_path}$$

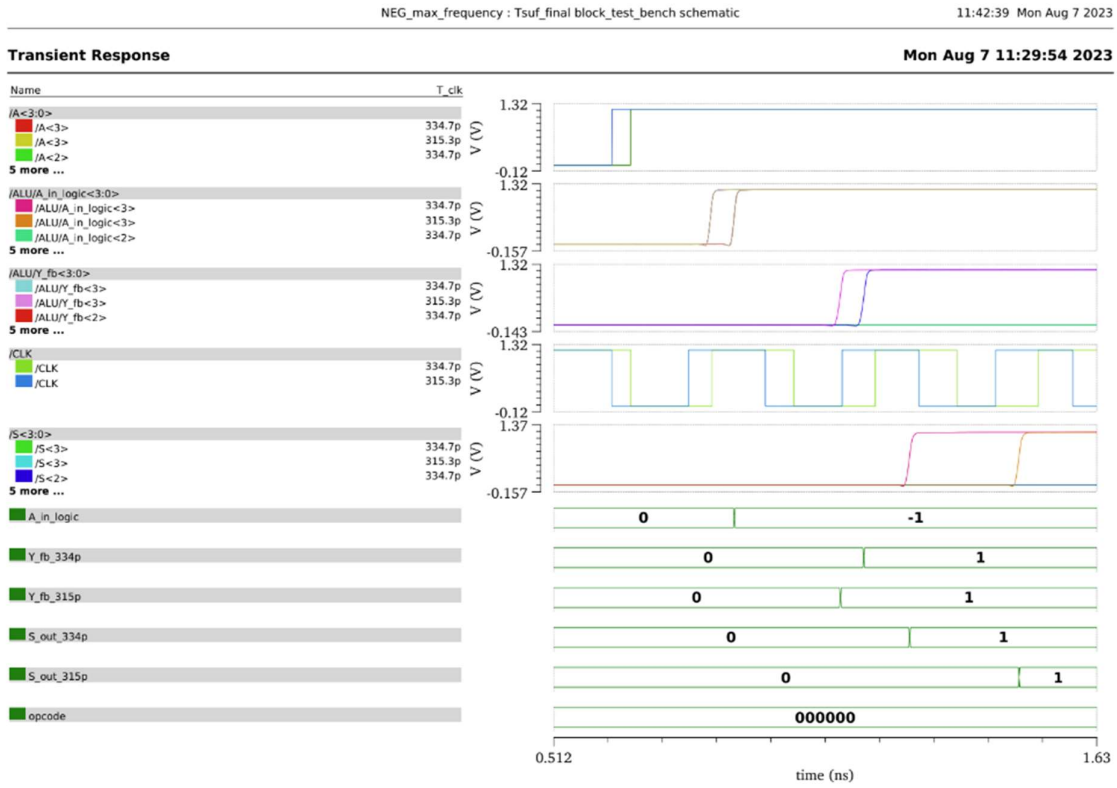
From the following connection:

$$f_{max} = \frac{1}{T_{min}}$$

We will get the maximal frequency for each case:

	$1.2V$	$1V$	$900mV$
Maximal frequency <u>without</u> parasitics	$3.1785\ GHz$	$2.075\ GHz$	$1.485\ GHz$
Maximal frequency <u>with</u> parasitics	$1.75\ GHz$	$1.154\ GHz$	$0.837\ GHz$

From the table above, we can see that lower supply voltage → lower maximal clock frequency.
In order to check our results, we simulated the critical path (through the NEG block) with supply voltage of 1.2V.
From our calculations, the minimal clock period possible for our ALU to preform correctly (for supply voltage of 1.2V and no parasitics) is: $T_{clk,min} = 0.3146\ nsec$.
In the following plot, we present the results we get from simulation for $T_{clk} = 315.3\ psec$ and for



$T_{clk} = 334.7\ psec$:

The input is A[3:0]=1111, and the output is 0001, as expected (two's complement negation).
Y_fb is the output the combinatoric logic, which is the input for the output register. As can be seen in the plot above, for $T_{clk} = 315psec$, the ALU output, S, changes two clock periods after the inputs get into the logic blocks → the ALU cannot function correctly in this frequency.
On the other hand, for $T_{clk} = 334psec$, the output from the logic is ready just in time to get through the output register in its correct value. This shows that our frequency calculations were almost accurate.

- **Rise/ Fall times:**

  We measured rise and fall times of few inputs that are branching and going to several places. We set a pass/ fail test with the spec of $< 50pF$ and all the inputs we tested have passed. We used the same test bench to perform these measurements like we used previously, and with an AND gate used as a load. The results are below:

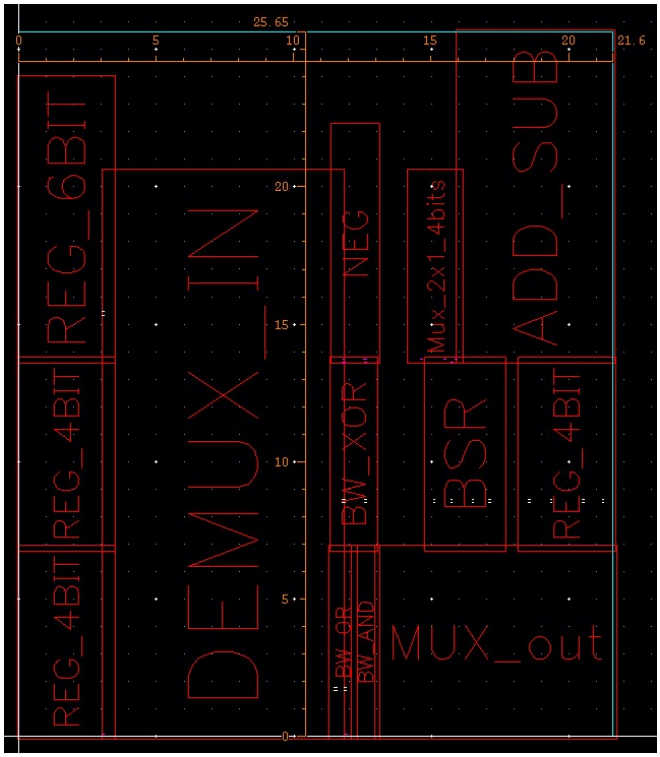
Test	Output	Nominal	Spec	eig	Pass/Fail
Filter	Filter	Filter	Filter	Fil	Filter
ltay_final_block_test_bench_1	riseTime(VT("/I24/opcode<5>") 0 nil 1.2 nil 10 90 nil "time")	23.81p	< 50p		pass
ltay_final_block_test_bench_1	riseTime(VT("/I24/opcode<4>") 0 nil 1.2 nil 10 90 nil "time")	34.4p	< 50p		pass
ltay_final_block_test_bench_1	riseTime(VT("/I24/opcode<3>") 0 nil 1.2 nil 10 90 nil "time")	30.86p	< 50p		pass
ltay_final_block_test_bench_1	riseTime(VT("/I24/opcode<2>") 0 nil 1.2 nil 10 90 nil "time")	45.23p	< 50p		pass
ltay_final_block_test_bench_1	riseTime(VT("/I24/net2<0>") 0 nil 1.2 nil 10 90 nil "time")	10.5p	< 50p		pass
ltay_final_block_test_bench_1	riseTime(VT("/I24/net1<0>") 0 nil 1.2 nil 10 90 nil "time")	10.87p	< 50p		pass
ltay_final_block_test_bench_1	riseTime(VT("/I24/Y_fb<0>") 0 nil 1.2 nil 10 90 nil "time")	14.3p	< 50p		pass
ltay_final_block_test_bench_1	fallTime(VT("/I24/opcode<5>") 1.2 nil 0 nil 10 90 nil "time")	22.75p	< 50p		pass
ltay_final_block_test_bench_1	fallTime(VT("/I24/opcode<4>") 1.2 nil 0 nil 10 90 nil "time")	34.93p	< 50p		pass
ltay_final_block_test_bench_1	fallTime(VT("/I24/opcode<3>") 1.2 nil 0 nil 10 90 nil "time")	30.46p	< 50p		pass
ltay_final_block_test_bench_1	fallTime(VT("/I24/opcode<2>") 1.2 nil 0 nil 10 90 nil "time")	46.39p	< 50p		pass
ltay_final_block_test_bench_1	fallTime(VT("/I24/net2<0>") 1.2 nil 0 nil 10 90 nil "time")	10.36p	< 50p		pass
ltay_final_block_test_bench_1	fallTime(VT("/I24/net1<0>") 1.2 nil 0 nil 10 90 nil "time")	10.36p	< 50p		pass
ltay_final_block_test_bench_1	fallTime(VT("/I24/Y_fb<0>") 1.2 nil 0 nil 10 90 nil "time")	14.37p	< 50p		pass

  The inputs in the table, from top to bottom, are:

  - OP<5> (feedback_en)- branches to 4 muxes.
  - OP<4> (shift_en)- branches to 4 muxes and 4 demuxes.
  - OP<3> (sub_en)- branches into 5 places (4 to the XOR before the adder and 1 to the carry in of the adder).
  - OP<2> (add_sub_en)- branches to 4 muxes and 4 demuxes.
  - A<0> and B<0> which enters demux_in.
  - Y_fb (the output of mux_out and an input of the feedback mux).

Floor Plan and layout

Our final floor plan in virtuoso is:



Measured size of $\boldsymbol{21.6\mu m \times 25.65\mu m}$ and total area of $\boldsymbol{554.04\mu m^2}$

Compared to the original floor plan size of $20.8\mu m \times 25.65\mu m$ and total area of $533.52\mu m^2$. This difference is because of a $0.8\mu m$ increase in the width of the demux_in block, which we'll discussed later.
The total area of our ALU is 140% of the total area of the cells we used, which is less than the maximal realization area (250% of gates area).

This layout follows the design of the schematic with “left to right” flow of the data, while also using the space efficiently, and leaving room for wiring between the blocks.

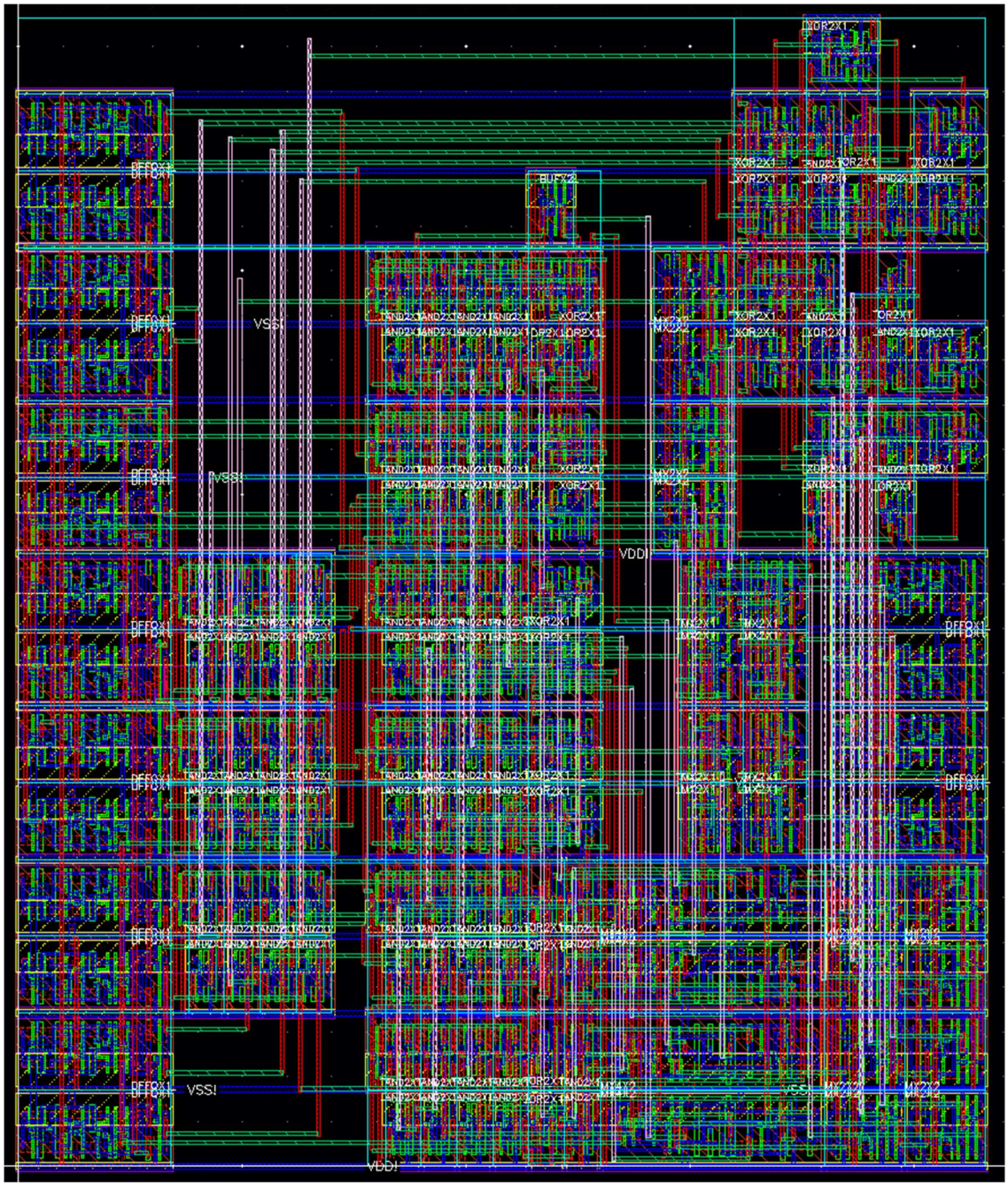
All of layouts we built are constructed by placing each component above the other, so that every pair of components is sharing one voltage source (VDD or VSS). This was done to save as much space as possible and by the staff's recommendation. Also, all the components were placed in an orientation that the input is on the left and the output is on the right.

We decided to use this following convention about the metals:
Horizontal metals- metal 1 and 3.
Vertical metals- metal 2 and 4.
This was to prevent DRCs in the integration of all the layouts and to make this process easier.
The layouts of the blocks in the full ALU consist of 3 layers of metal, while the full ALU has 4.

Below are screenshots of the layout of each block and its size and cell name.

Denote that every screenshot of a layout in this section includes a ruler to show the size of it.
The unit of the ruler is microns.

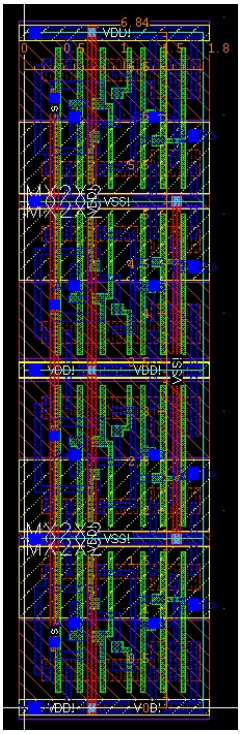
First, let’s admire the full layout of the ALU:



We can see the blocks as shown in the floorplan connected with metals as specified above.
Now we will dive in to specific blocks:

1. Feedback mux

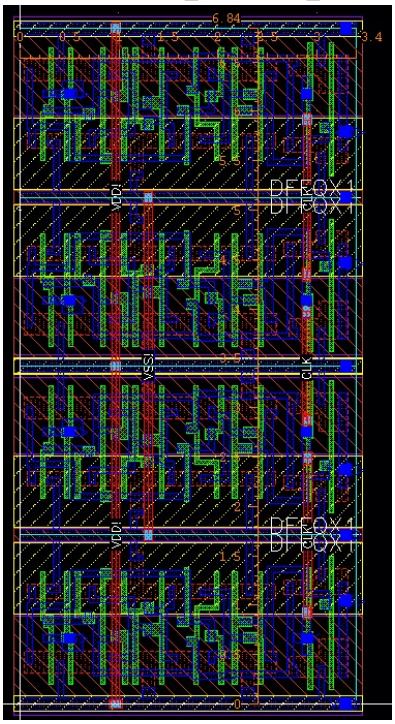
   Cell name: Mux_2x1_4bits

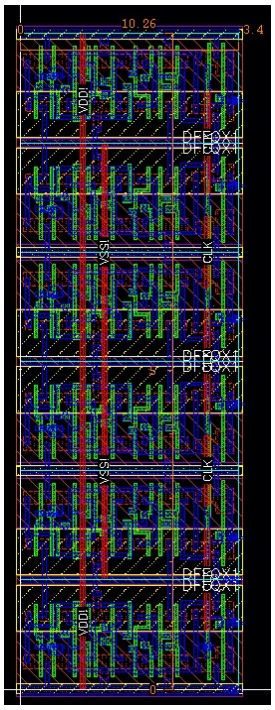


   Size: $1.8\mu m \times 6.84\mu m$

2. Input registers

   Cell names: REG_4BIT, REG_6BIT

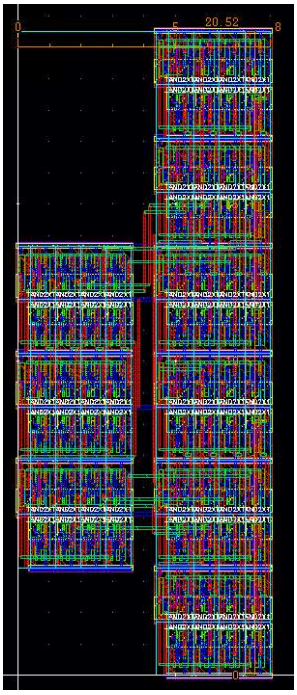




   Size: $3.4\mu m \times 6.84\mu m$ $(4\ bits)$, $3.4 \times 10.26\mu m$ $(6\ bits)$

3. Demux_in

Cell name: DEMUX_IN

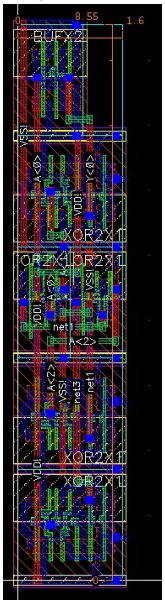


Size: $8\mu m \times 20.52\mu m$

In the planning state we estimated this block will be in the size of $7.2\mu m \times 20.52\mu m$. In the implementation we realized we'll need more space because we couldn't attach the 2 parts, and passing all the interconnects between them, without getting DRCs. We decided to leave a gap there because we saw that we are still upholding the space requirement.

4. Logic operations blocks

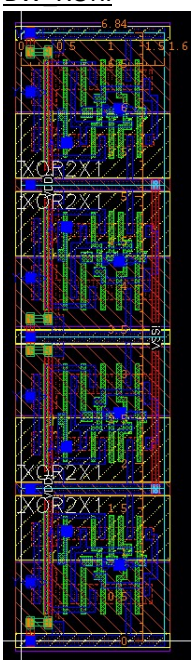
Neg:



Size: $1.6\mu m \times 8.55\mu m$

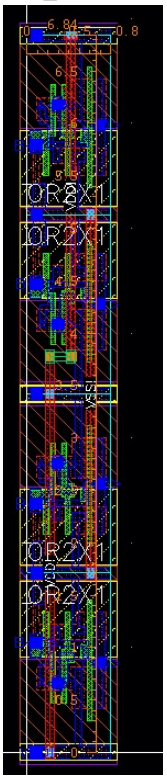
In the planning state we estimated this block will be in the size of $1.6\mu m \times 6.84\mu m$. In the implementation we decided to add a buffer, as describe above so the height of this block increased. This change hasn't affected the size of the overall floor plan because we had space above this block that was empty.

BW_XOR:



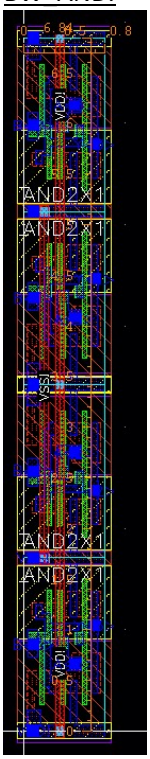
Size: $1.6\mu m \times 6.84\mu m$

BW_OR:



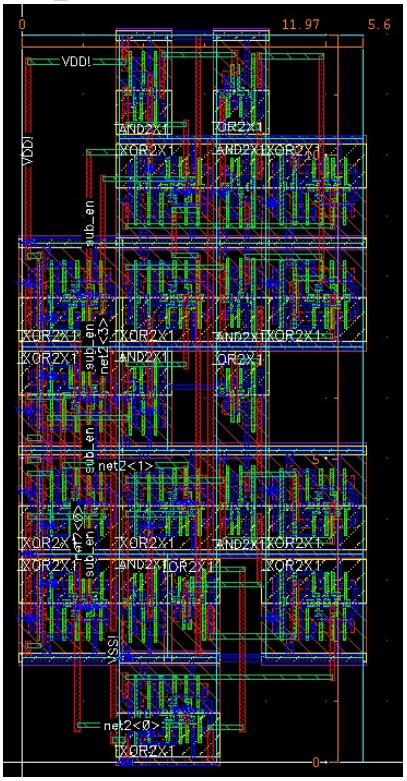
Size: $0.8\mu m \times 6.84\mu m$

BW_AND:



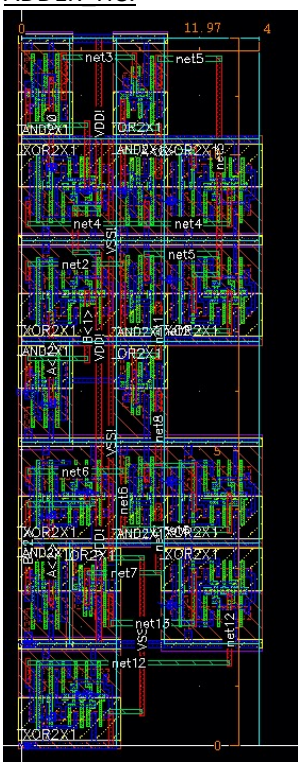
Size: $0.8\mu m \times 6.84\mu m$

ADD_SUB:



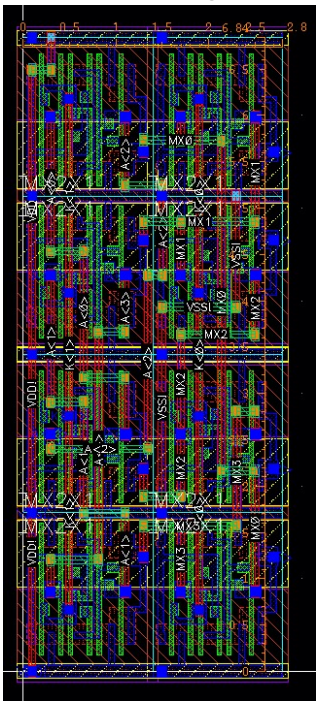
Size: $5.6\mu m \times 11.97\mu m$

ADDER_HC:



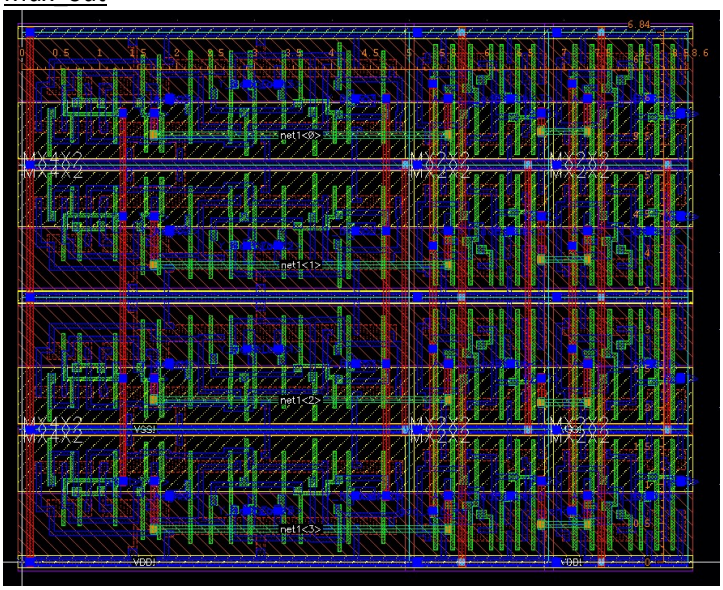
Size: $4\mu m \times 11.97\mu m$

BSR (barrel shift right):



Size: $2.8\mu m \times 6.84\mu m$

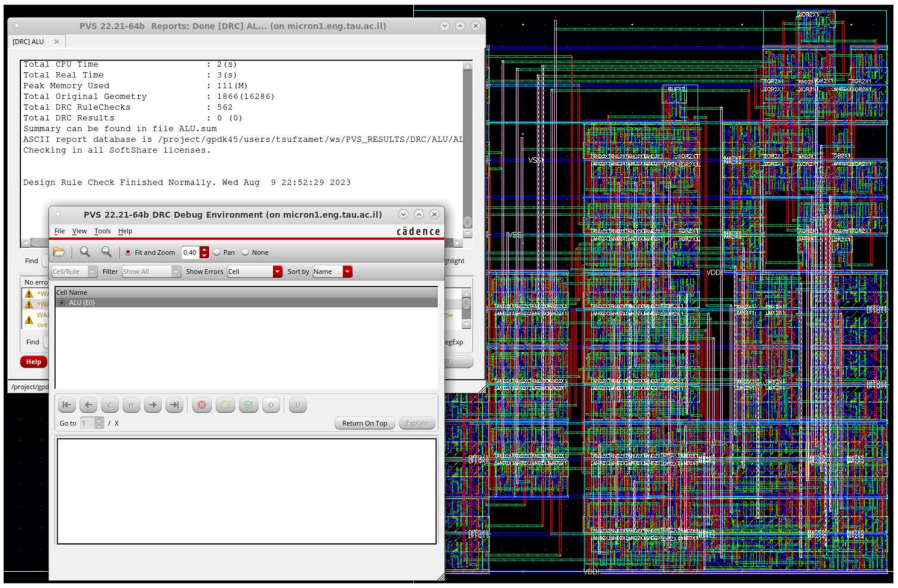
5. Mux_out



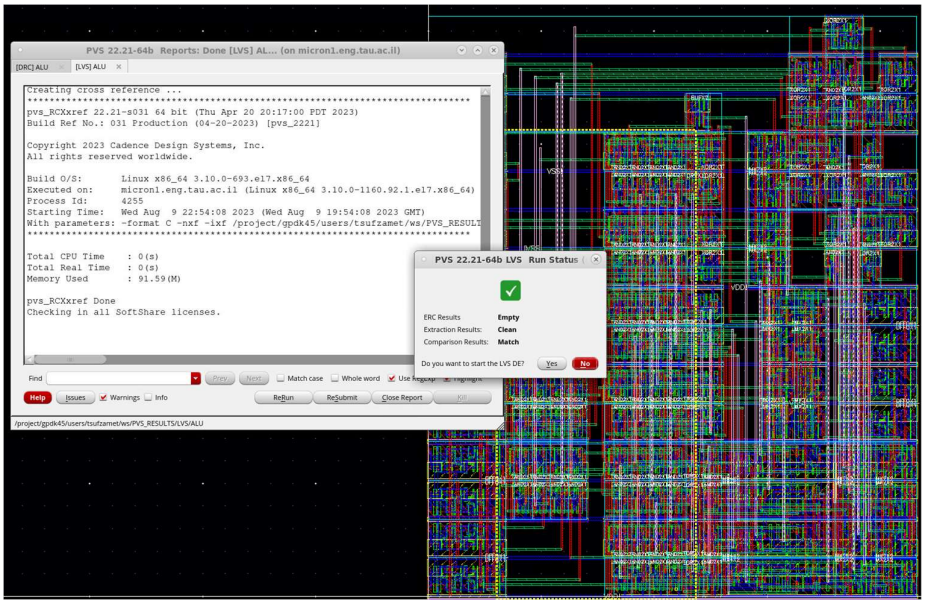
Size: $8.6\mu m \times 6.84\mu m$

6. Output register
   We used the REG_4bit for the output register (layout above in the input registers section).

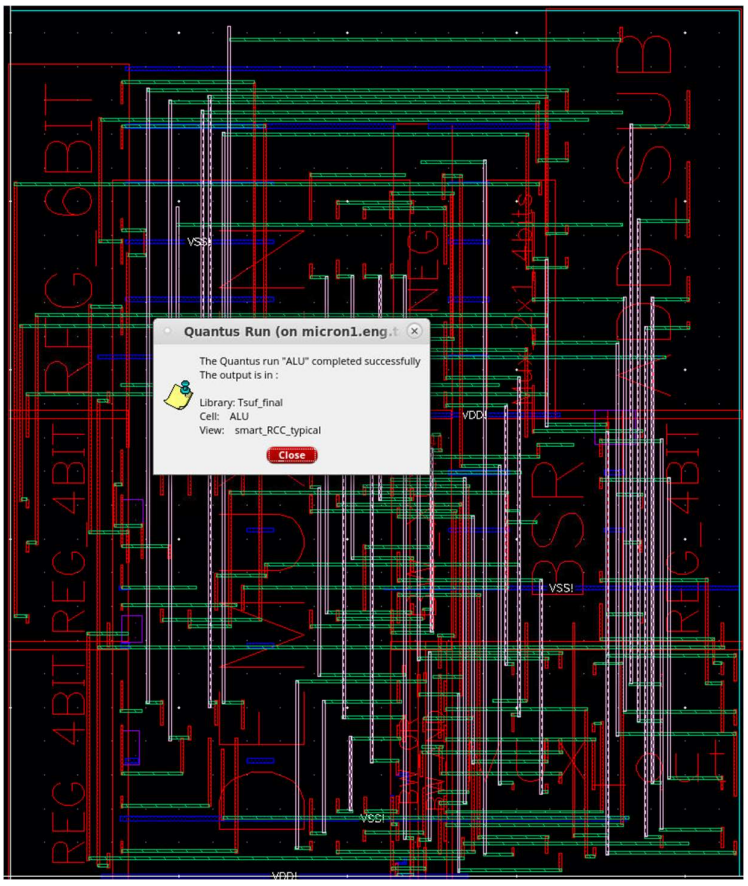
- DRC , LVS and QRC confirmations
  Our ALU passed DRC and LVS tests with no errors.
  DRC confirmation:



LVS confirmation:



QRC extraction confirmation:



Summary

During the planning and implementation stages of the project, our focus was mostly on making our design simple, and as small as possible (area-wise).
We divided our ALU to blocks, each one meant to preform different functionality.
We also chose to build some of the complex blocks from smaller sub-blocks for modularity.
One thing we think that could've improve our ALU's performance:

- Our Demux_IN block is relatively big, and its inside logic is quite slow (for example, data from input registers needs to go through 3 demux blocks before getting to some of the logic blocks).
  Improving the logic and arrangement of this block (for example by using pass transistor logic instead of logic gates) could help shorten the critical path delay time (and of course help increasing the maximal clock frequency).