LABORATORY DESIGN PROJECT

By Reuben De Souza

Introduction

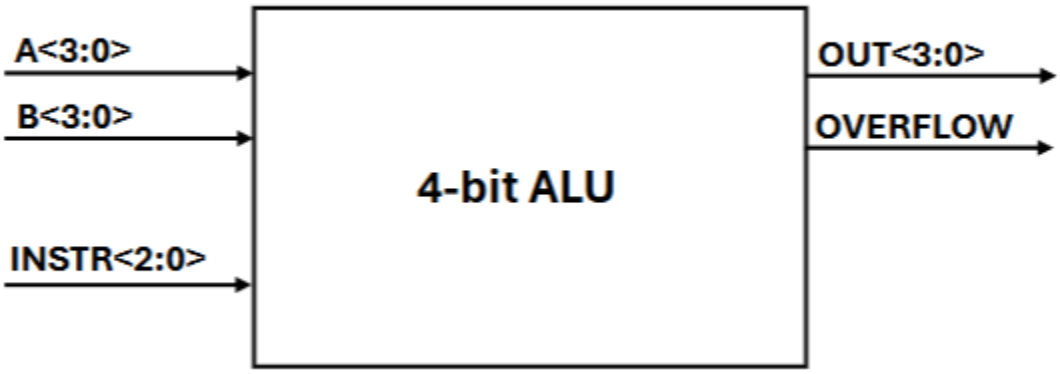

Figure 1: High Level Block Diagram

This project focused on the design, implementation, and testing of a 4 bit ALU using TSMC's 0.18 µm CMOS logic gates. The ALU takes in two 4 bit inputs **A** and **B**, as well as a 3 bit opcode, **INSTR**, indicating which operation is to be performed. The ALU has a 4 bit output, **OUT**, and an overflow flag to indicate additional overflow.

The following instructions were executed on the following inputs:

INSTR<2:0>	OUT<3:0>	Description
000	A	Pass A
001	B	Pass B
010	= (A AND B)	Bitwise AND
011	= (A OR B)	Bitwise OR
100	= (A XOR B)	Bitwise XOR
101	= A + B	Add
110	= A - B	Subtract (using two's complement)
111	= 1111	All high

Figure 2: Opcode set

Design

I followed the example structure provided in the assignment, with a few modifications to the ROM that served as the instruction decoder. Every logic gate in this schematic uses 2:1 PMOS:NMOS sizing for equivalent strength.

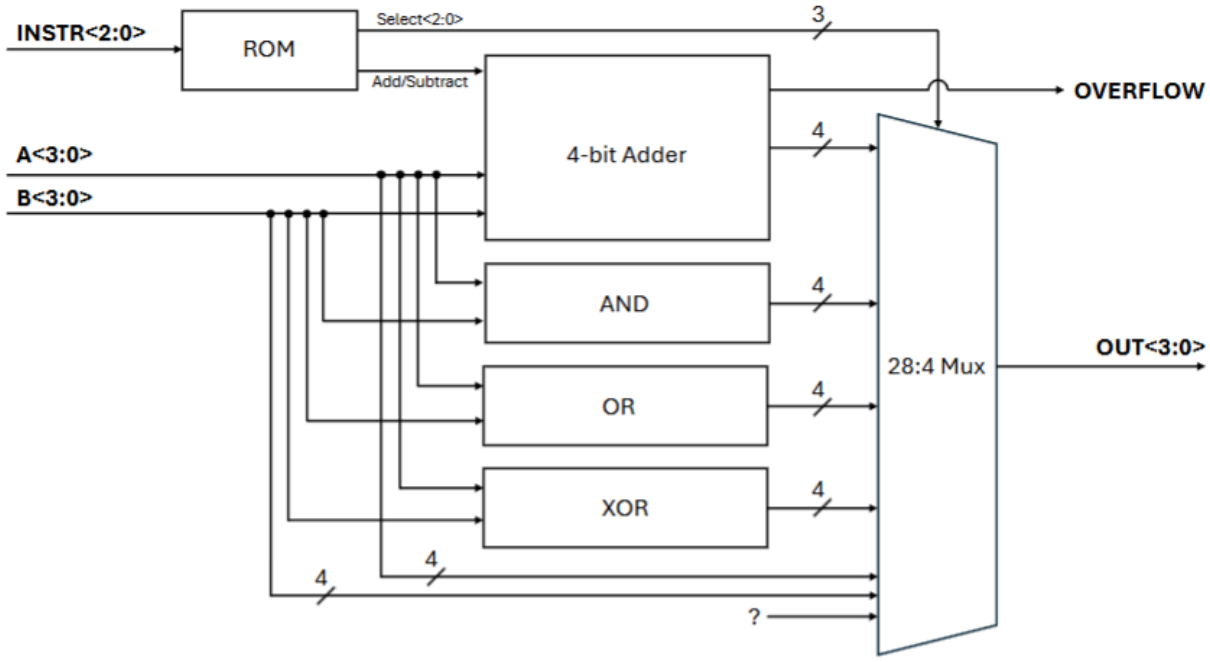

Figure 3: Instruction Level Diagram

In this architecture, both A and B are passed to all operation blocks, and based on the opcode, the appropriate output is multiplexed out. Another way to do this would be to multiplex the inputs A and B, then have some OR gates to combine the outputs into a single bus, but for simplicity I decided against this. Much of the design work was in creating the individual blocks so let's walk through them.

AND

This block performs bitwise A AND B, and outputs the 4 bit result. This operation is performed by passing each bit through a two input NAND gate and an inverter to match the functionality of AND. This NAND2 gate is the exact same block from a previous lab, and follows the standard CMOS NAND arrangement. Rather than use buses, I chose to copy paste and change net names to index A<3:0>, B<3:0> and out<3:0>.

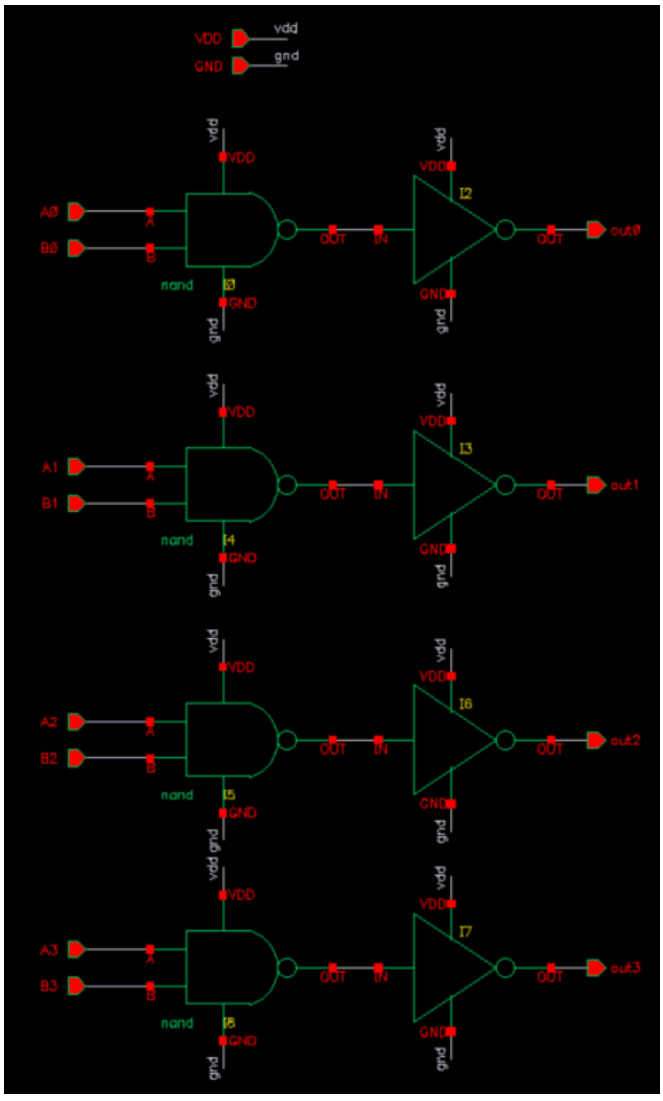

Figure 4: Bitwise AND

OR

This block performs bitwise A OR B, and outputs the 4 bit result. This operation is performed by passing each bit through a two input NOR gate and an inverter to match the functionality of OR. This NOR2 gate is the exact same block from a previous lab, and follows the standard CMOS NOR arrangement. Again, rather than use buses, I chose to copy paste and change net names to index A<3:0>, B<3:0> and out<3:0>.

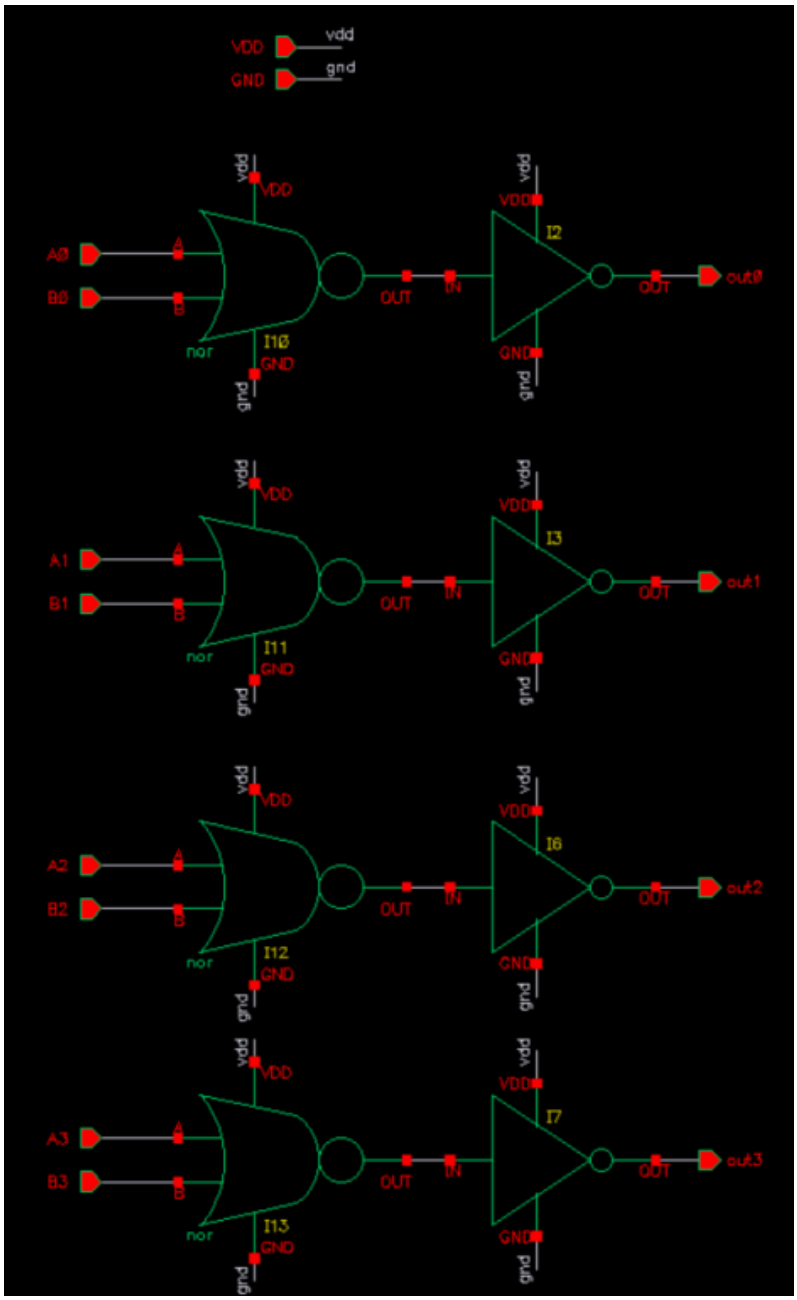

Figure 5: Bitwise OR

XOR

This block performs bitwise Exclusive OR (A XOR B), and outputs the 4 bit result. This operation is performed by passing each bit through a two input XOR gate. This XOR gate is the exact same block from a previous lab, and utilizes NAND2 gates in the arrangement shown below.

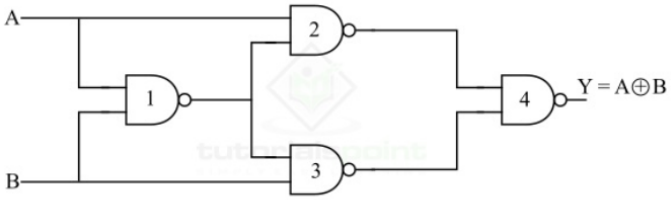

Figure 6: XOR using NAND Gates

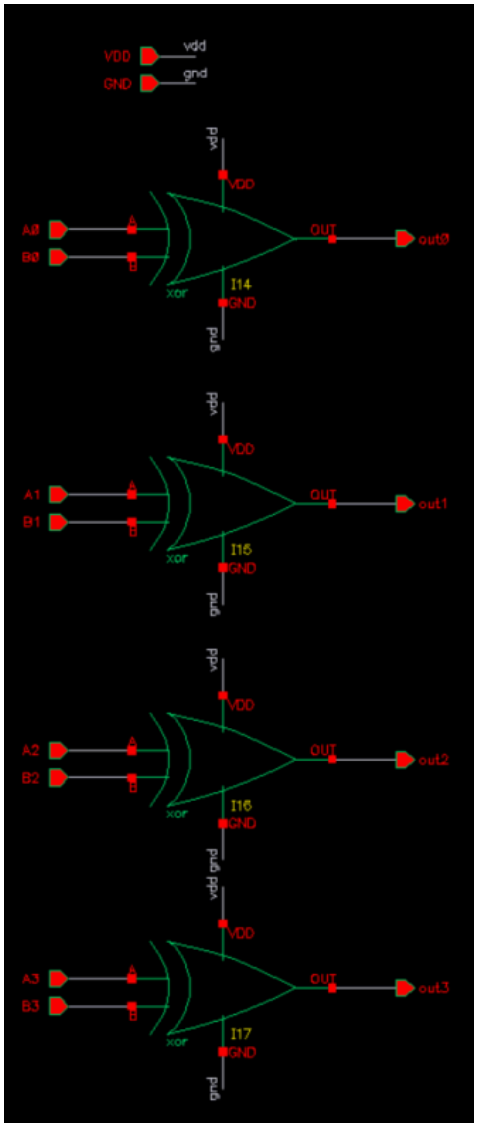

Figure 7: Bitwise XOR

4 Bit Adder/Subtractor

This block performs A + B or A - B. The fundamental block is a full adder, which takes inputs A, B, and a Carry In (Cin), and outputs the sum (S), as well as a Carry out bit (Cout). I have chosen to use a mirror adder, which decreases the number of transistors allowing for faster logic.

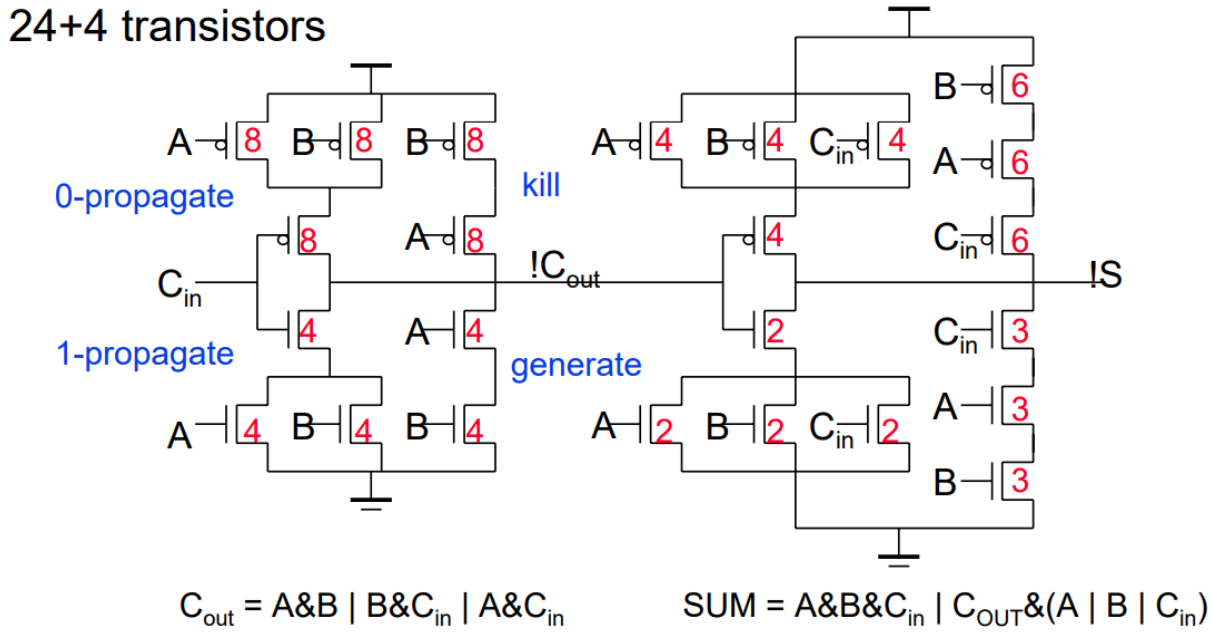

Figure 8: Mirror Implementation of Full Adder [1]

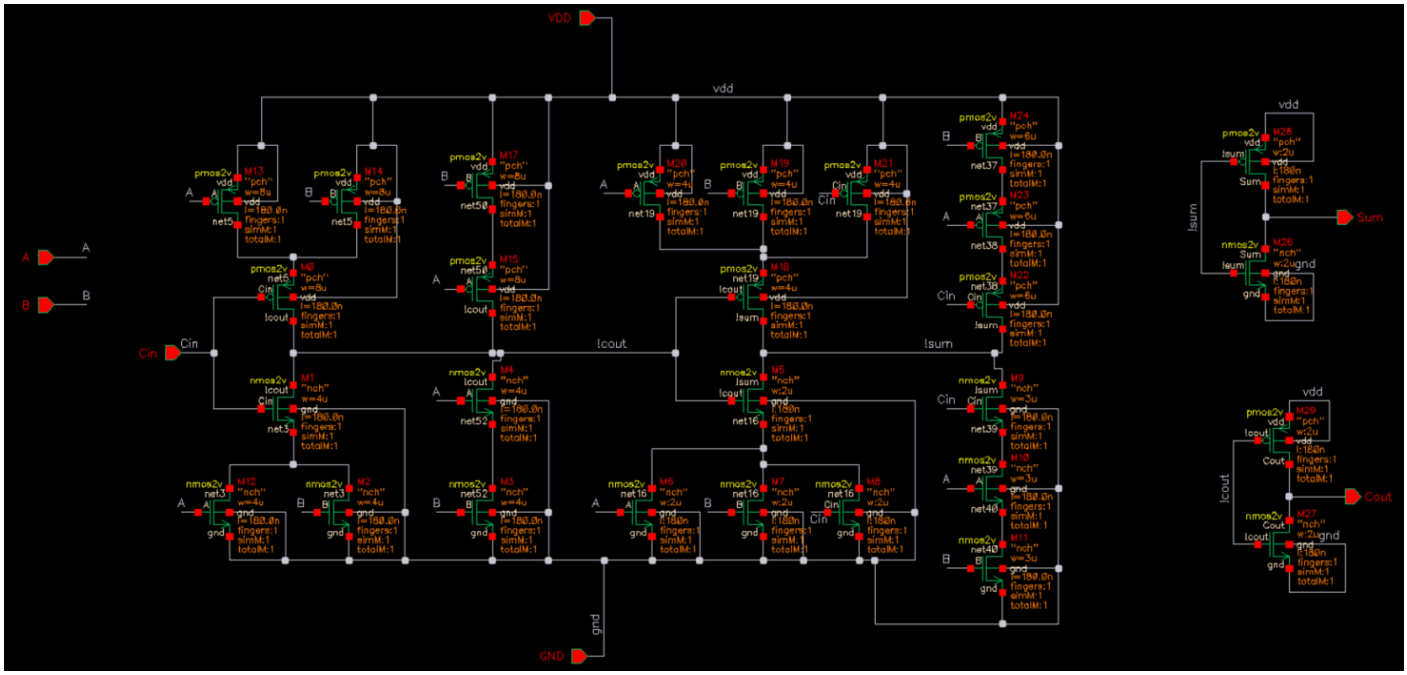

Figure 9: Mirror Adder Schematic in Cadence

The extra 4 transistors are necessary because the mirrored logic outputs !Cout and !Sum, so inverters are required to obtain Cout and Sum. The sizing of the transistors is also shown here to maintain a PMOS/NMOS ratio of 2. You will notice how the transistors that drive !Cout are double the size, effectively a 4:2 width ratio. This is because !Cout drives the !sum stage and the inverter to create Cout, so to maintain the optimal fan-out of 2 (based on the fact that each input has a logical effort of 2) we must make them larger. [1] Some other features of the mirror adder are illustrated in Figure 10.

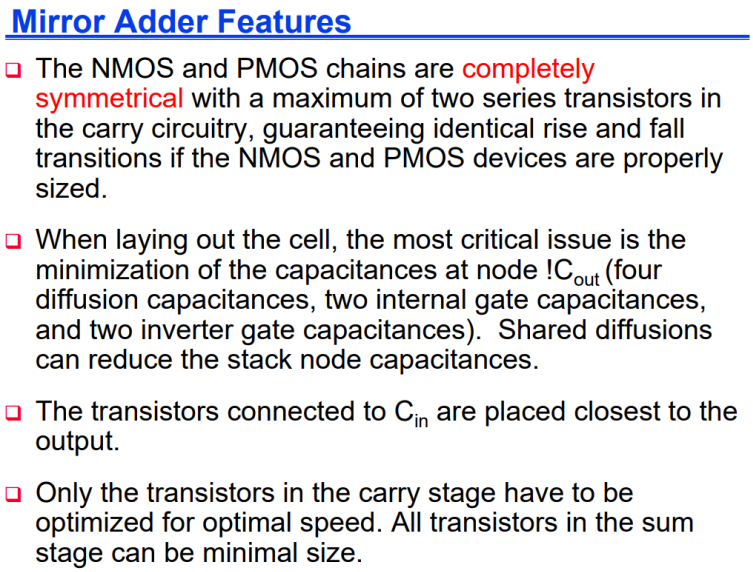

Figure 10: Benefits of a Mirror Adder [1]

For this portion of the block it was required to create a layout. I roughly followed the stick diagram in Figure 11; however, our PDK doesn't allow for different sized fingers so I created three different blocks. Due to the mirrored nature, I really only had to implement the Pull-up network and then copy it for the Pull-down.

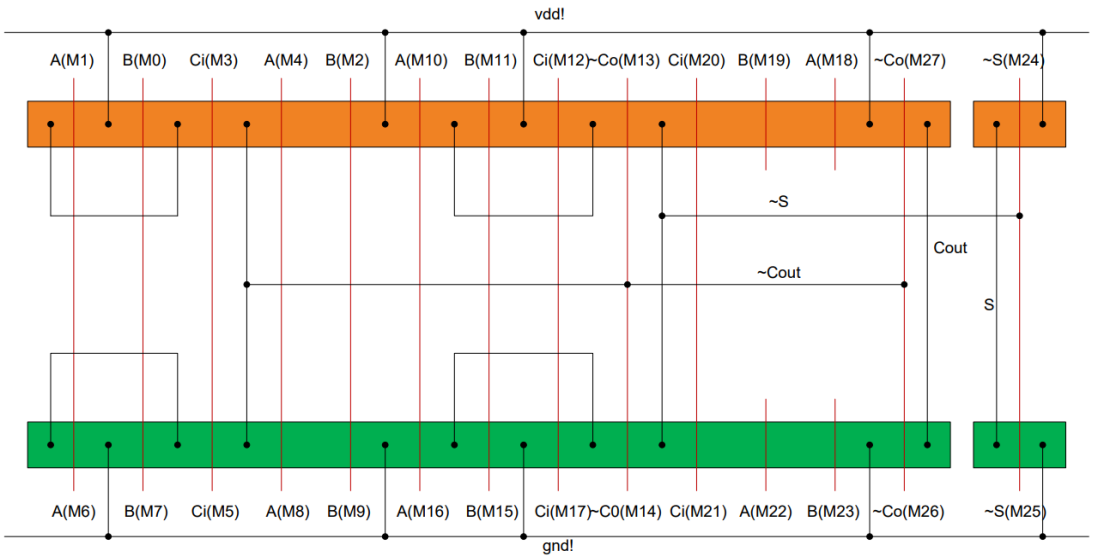

Figure 11: Mirror Adder Stick Layout [2]

Using the techniques learned in class I created a layout like this. In a revision, I could definitely trim the spaces between the PMOS and NMOS networks, and simplify the substrate taps.

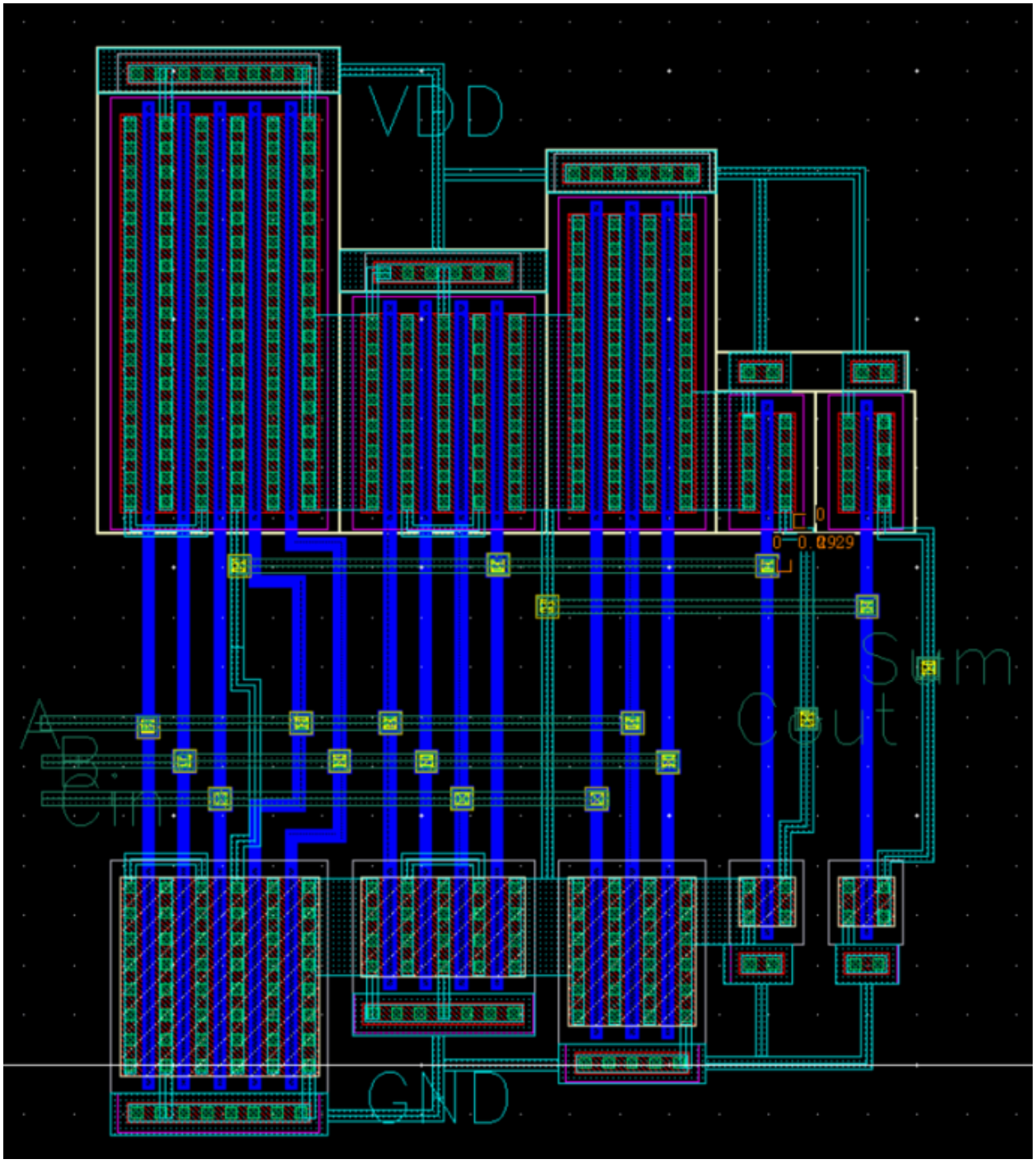

Figure 12: Layout in Cadence

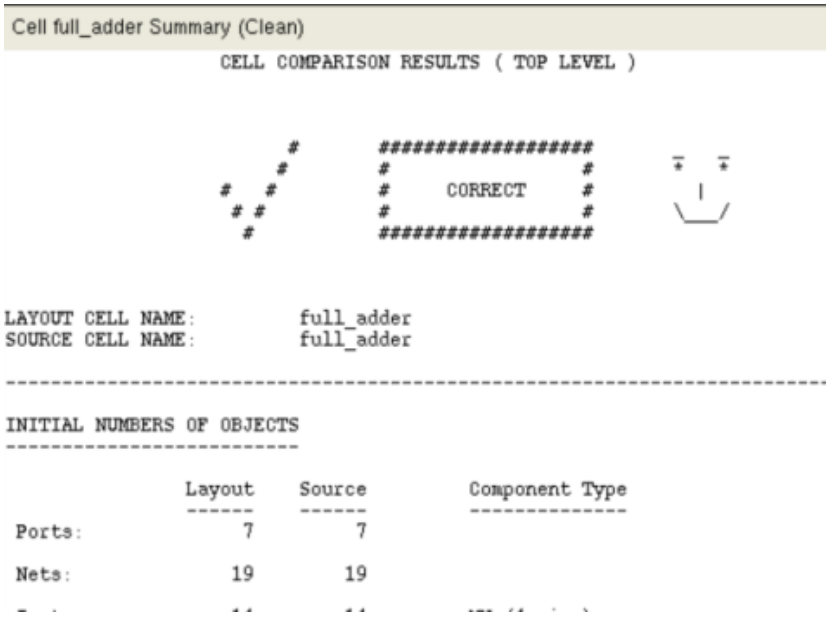

Figure 13: LVS

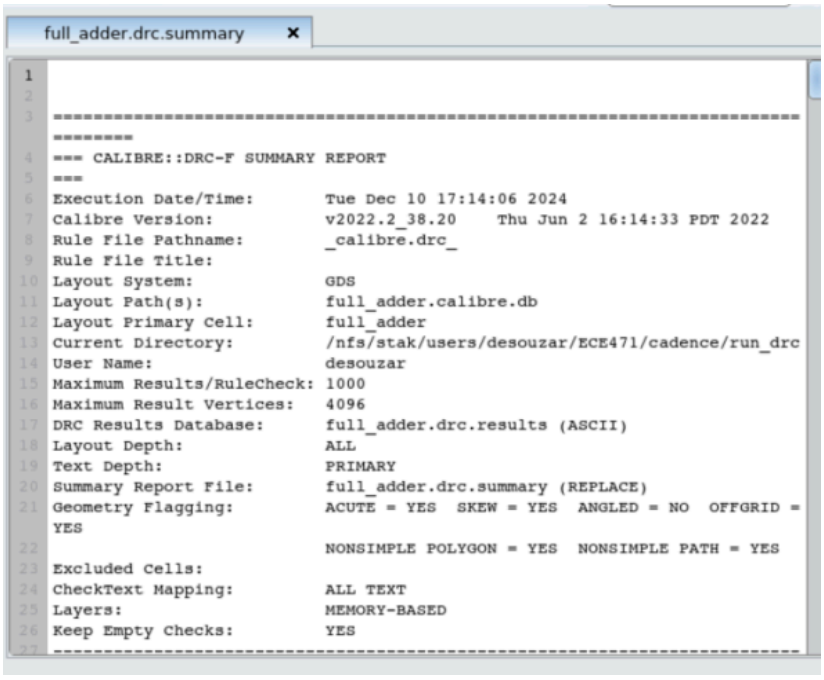

Figure 14: DRC Log

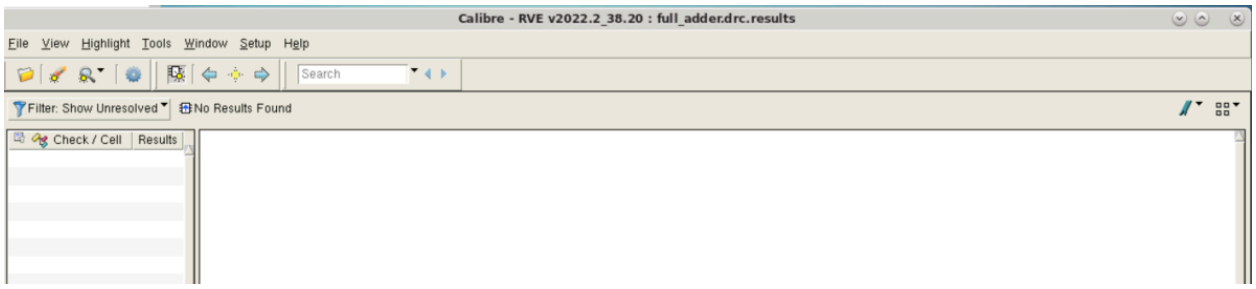

Figure 15: No Results Found!

To achieve an adder that is 4 bits long we chain together the Cout of one full adder to the Cin of the next adder.

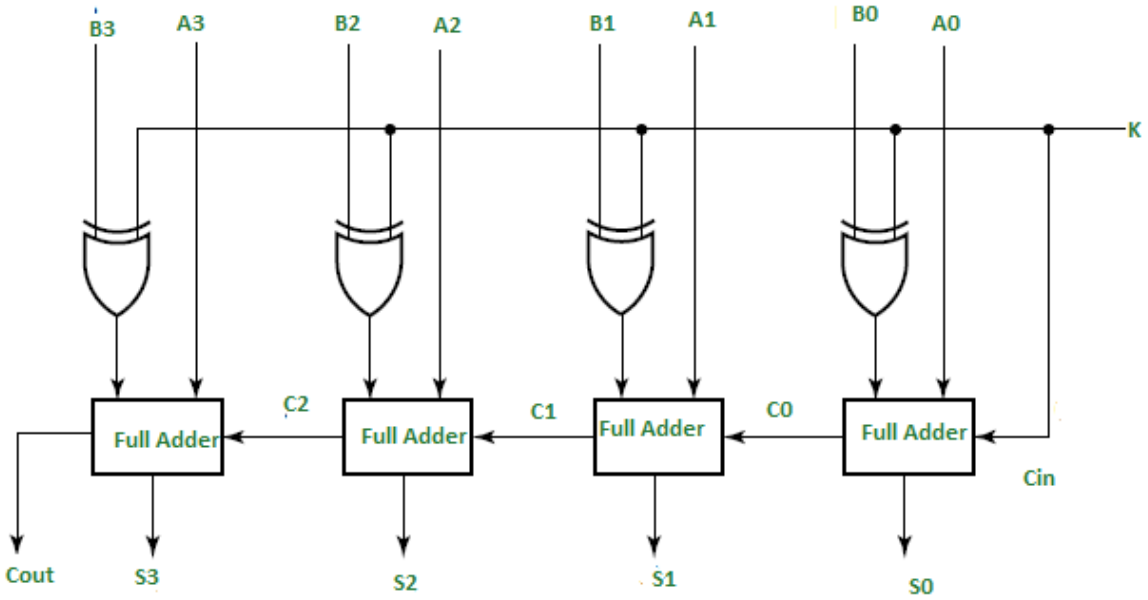

Figure 16: 4-bit Adder-Subtractor [3]

To allow our adder to perform subtraction using two's complement, input B can be XOR'd with the first Cin bit. If Cin is set, all bits of input B will be flipped, and an extra 1 will be added, mimicking taking the two's complement. From there normal addition can be performed, and our output will indicate a two's complement value.

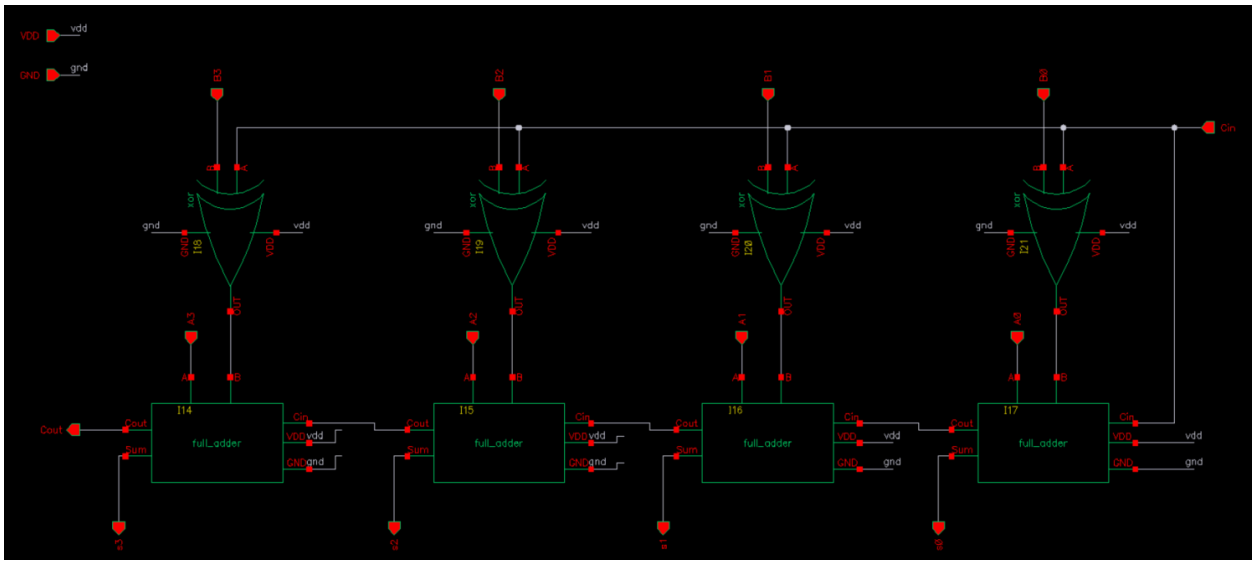

Figure 17: 4-bit Adder-Subtractor in Cadence Schematic

Multiplexer

This block determines which input should be passed through to the output using selection bits. If we are counting every signal signal, this is technically a 28:4 multiplexer; however, since the signals are grouped in 4's, we really have a 7:1 multiplexer where each signal is a 4 wide bus.
To accommodate all 7 inputs (remember the addition and subtraction inputs come from the same stage) we have to use 3 selection bits (2^3 = 8), where one of the combinations is unused. This implementation I chose to use is based on the design in Figure 18.

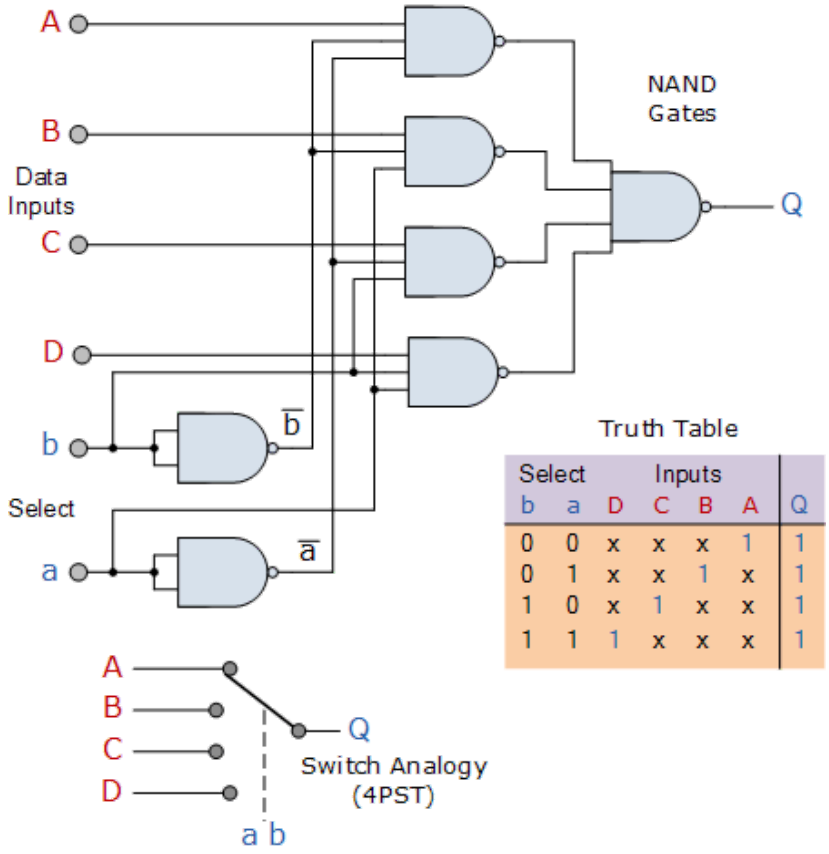

Figure 18: NAND-based Multiplexer [4]

Turning this into a 7:1 bus multiplexer, several changes are required. First, to get the NOT of the selection bits, I just used an inverter, due to its increased speed; not super important critical to a schematic but still simpler. Second, to accommodate 3 selection bits the first row of NAND gates requires 4 inputs: 3 for the selection bits and 1 for the data input to pass through. Then the final NAND gate is required to have 7 inputs, one for each possible state. Both of these gates had to be created, but were quite straightforward.

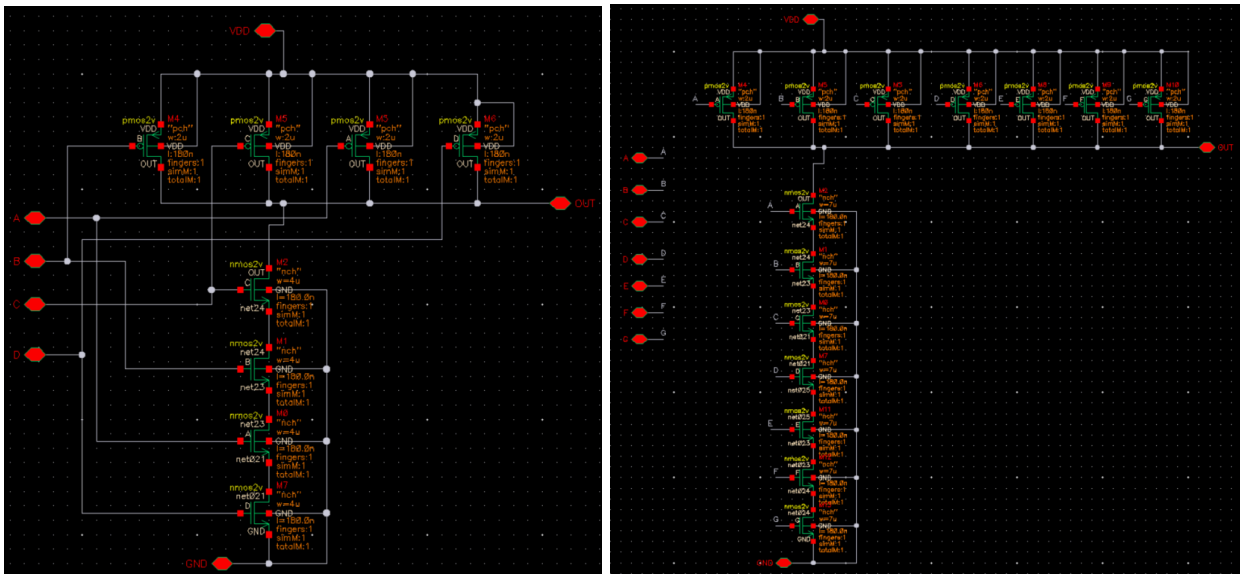

Figure 19: 4 Input and 7 Input NAND Respectively

The truth table for the multiplexer looks very similar to the overall instruction mapping. This is done on purpose, in fact the only change is that the INSTR that normally indicates subtraction (110), does nothing. This combination will never be passed to the MUX block due to the logic in the instruction decoder.

SEL2	SEL1	SEL0	Selected Output
0	0	0	Pass A
0	0	1	Pass B
0	1	0	AND
0	1	1	OR
1	0	0	XOR
1	0	1	ADD
1	1	0	N/A
1	1	1	High

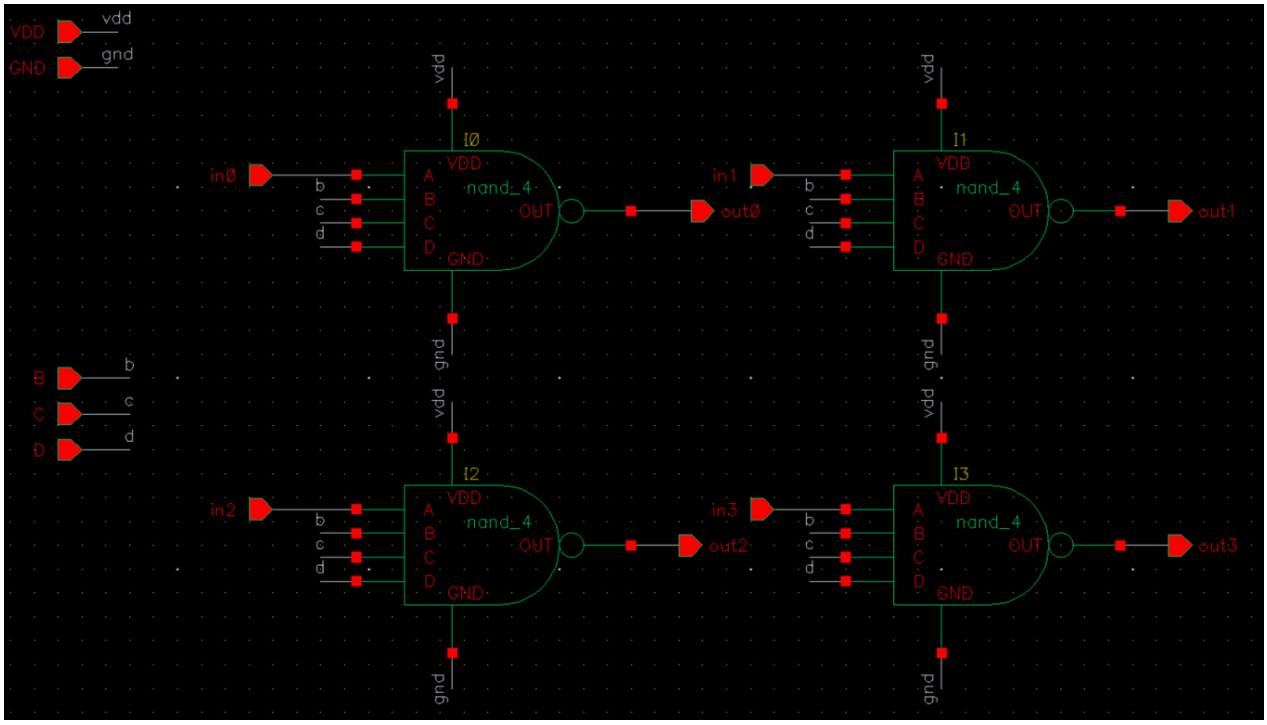

Figure 20: Block with Four 4 Input NAND Gates

To simplify the implementation I created a block that encompasses the 4 four input NAND gates. Each of these NAND gates receives the same selection inputs but differ in the degree of bits, passing through all or none of "bus" bits. Therefore, for each possible selection value I used one of these blocks, adjusting the selection signals that make it active. An example of the block passing XOR is shown in Figure 21.

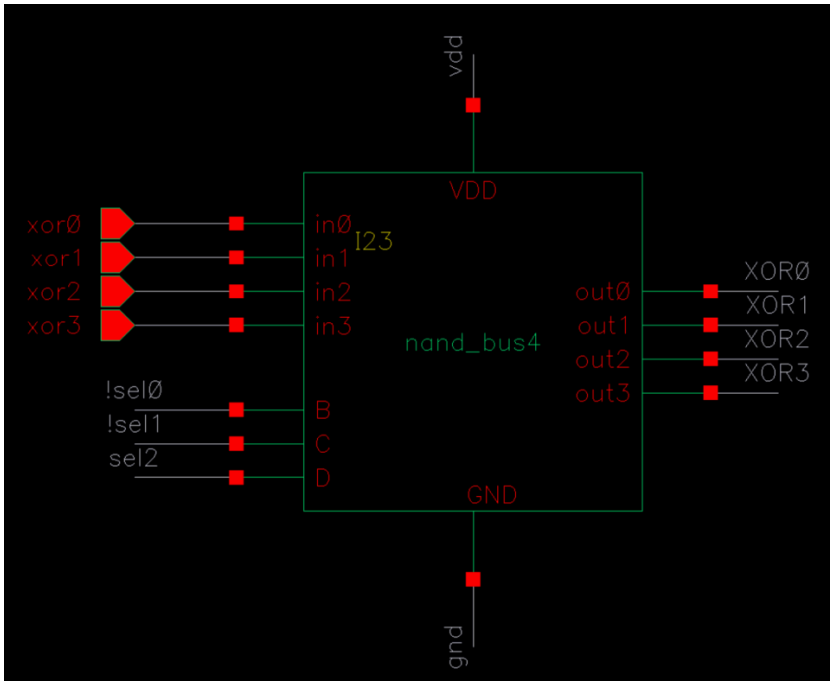

Figure 21: First Stage of Multiplexer Logic for XOR

For the XOR, signals if the selection bits are set to 100, the signals will be allowed through. This is how the selection is controlled, the other NAND gate is for combining the outputs for each bit place for the block. The NAND_7 block is shown in Figure 22.

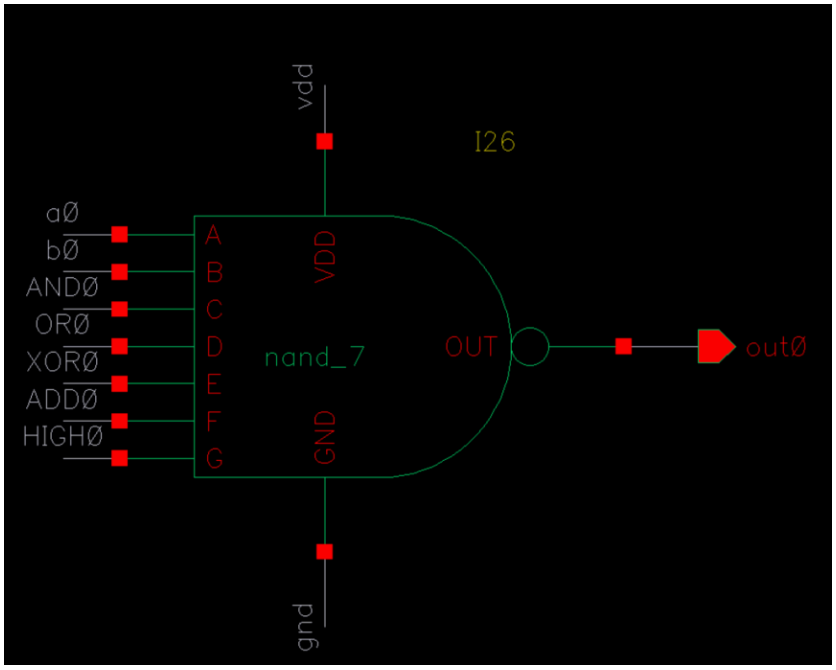

Figure 22: 7 Input NAND Gates

Due to the first stage of NANDs, only one of these bits can be low at a time; therefore, only one signal can pass 1 through to the output bit. Here is the full schematic for the block:

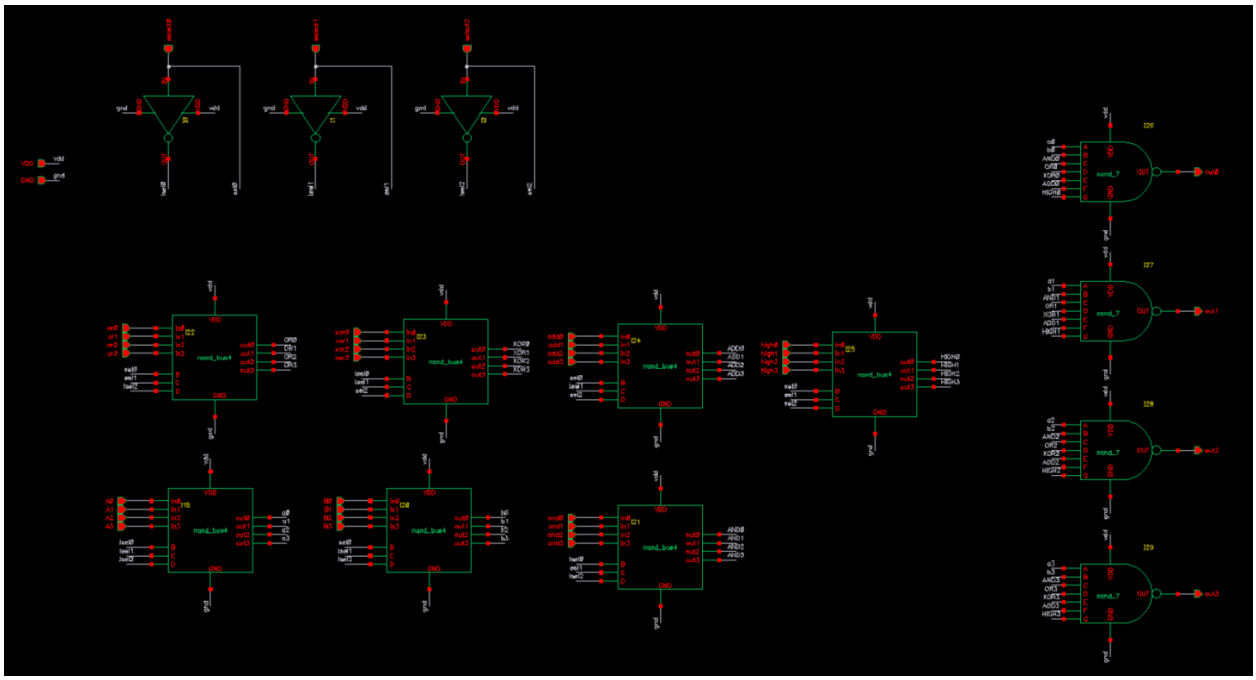

Figure 23: Full 28:4 Multiplexer Implementation

You can generally see how each input “bus” has a 4 NAND block, and those outputs get combined together in the final NAND_7 to create the 4 output bits.

Instruction Decoder

The instruction decoder has quite a simple job. For most inputs just pass the bits through to the output. For the case that INSTR<3:0> is 110, switch the output to 101 and set the subtraction flag. This pairs with the mux logic to select the correct output and to perform two’s complement subtraction with the adder block. The truth table necessary is the following:

INSTR2	INSTR1	INSTR0	Select <3:0>	flag
0	0	0	000	0
0	0	1	001	0
0	1	0	010	0
0	1	1	011	0
1	0	0	100	0
1	0	1	101	0
1	1	0	101	1
1	1	1	111	0

Notice the one special case (110) where logic is required. From this truth table we can come up with the following logic equations for each bit of the selection output.

1. select2 = INSTR2
2. select1 = (INSTR1) · !(INSTR2 · INSTR1 · !INSTR0)
3. select0 = INSTR0 + INSTR2 · INSTR1 · !INSTR0
4. sub_flag = INSTR2 · INSTR1 · !INSTR0

For the select2 bit, the output will always be the input. For the select1 bit the output will be the input as long as the bus is NOT 110. For select0 the output will be the input or 1 when the bus is 110. For the sub_flag, the output will only be 1 when the bus is 110. This logic changes our special case to pass 101 when the input bus is 110, and it sets the sub_flag. All other cases it just passed the input through to the output. The implementation in logic gates is shown in Figure 24.

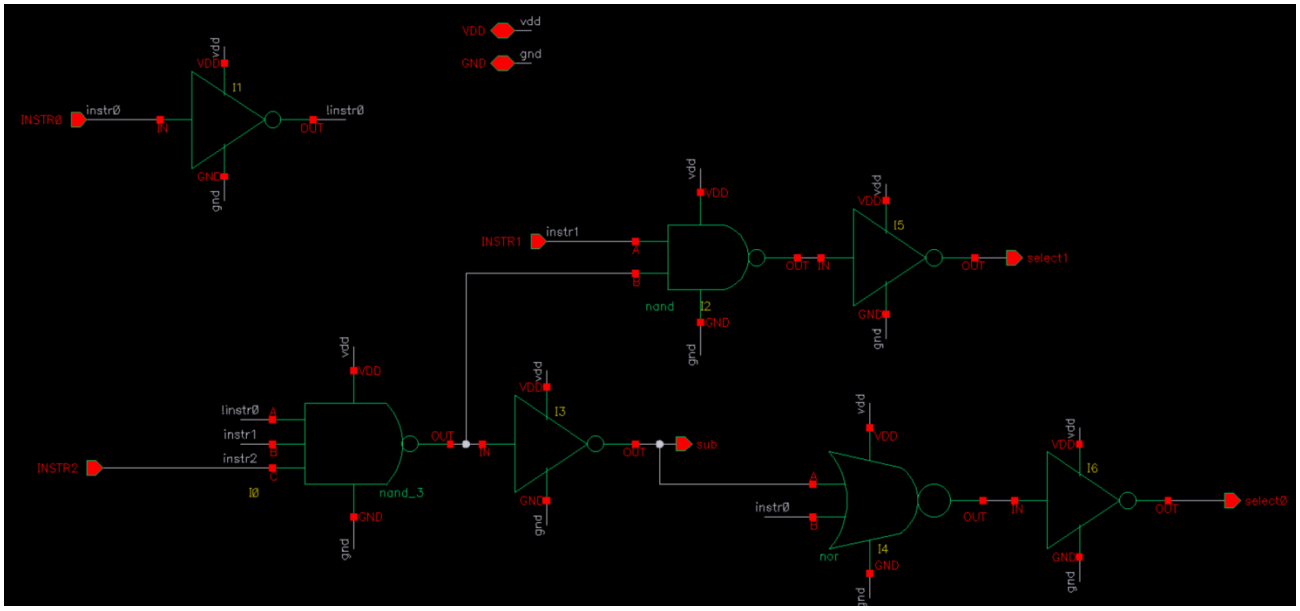

Figure 24: Instruction Decoder Schematic

Note, this isn't the simplest boolean logic but it plays nicely with NAND, NOT and NOR based logic.

Full System

All these blocks come together to form the system described in the introduction.

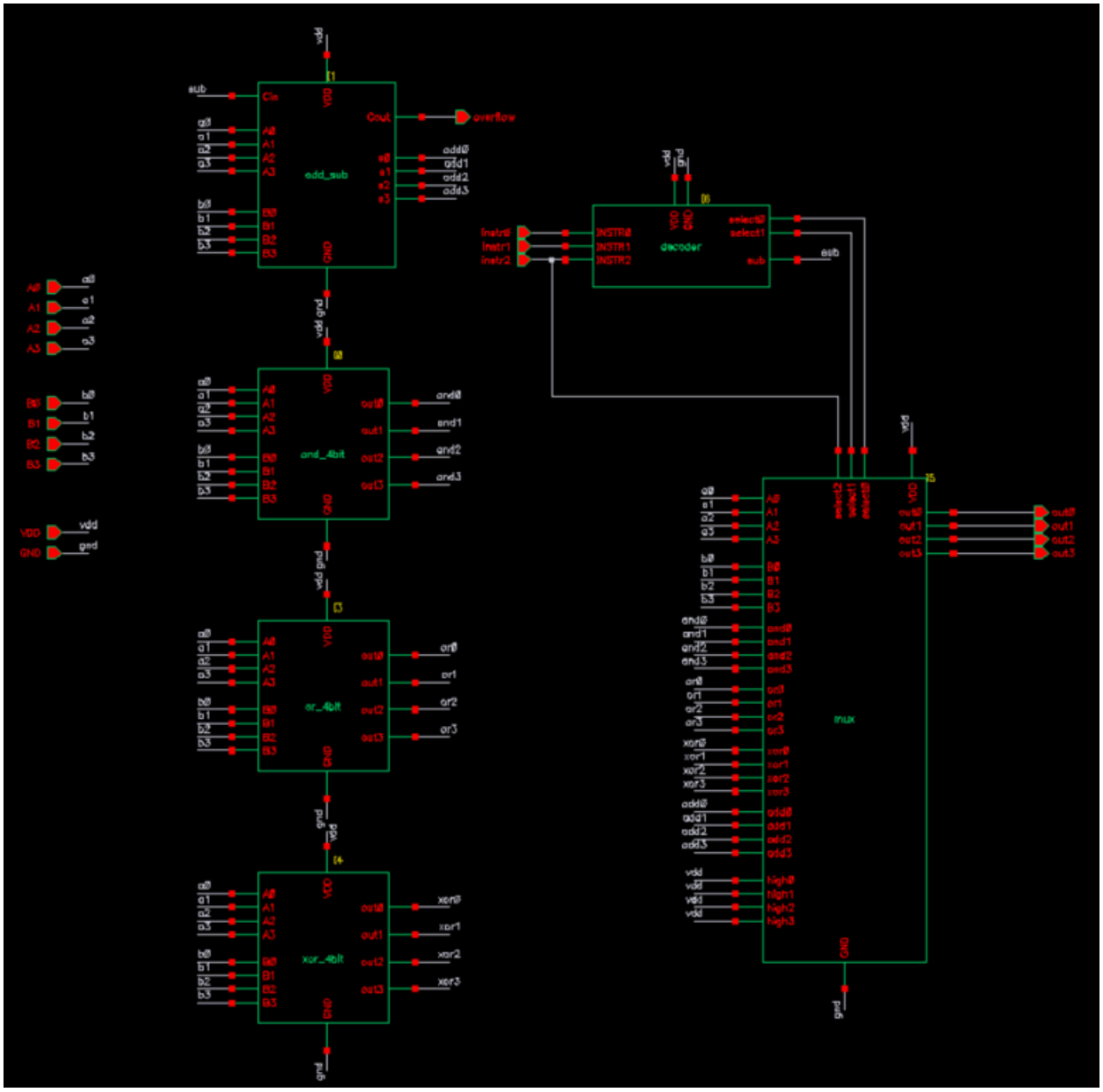

Figure 25: 4 Bit ALU Schematic

The flow of the logic is pretty intuitive to follow. All logic operations are performed on the A and B and their outputs arrive at the multiplexer. The instruction decoder decides which operation needs to be outputted, and if that operation is subtraction, it passes Cin to the 4-bit adder. The instruction decoder runs in parallel to each operation so the Cin bit will arrive in time for the addition. To achieve the HIGH instruction, those bits on the mux are simply connected to VDD.

Testing

Before testing the full system I decided to test each individual block. Testing all exhaustive inputs for every block would require too much simulation with the current tools. Therefore, I have decided to perform simple simulations that demonstrate basic functionality. All blocks are loaded with 100 fF capacitors at the outputs.

Bitwise AND

For the bitwise AND block, I have 4 pulse waves each of a period 2x the previous pulse. For inputs A<3:0> the period is in increasing order, that is A0 represents the top wave (red) and A3 represents the fourth wave from the top (neon green). For input B<3:0> the period is in decreasing order, that is B3 represents the top wave (red) and B0 represents the fourth wave from the top (neon green). The outputs (out<3:0>) are the bottom four waves, where out0 is the bottom most wave.

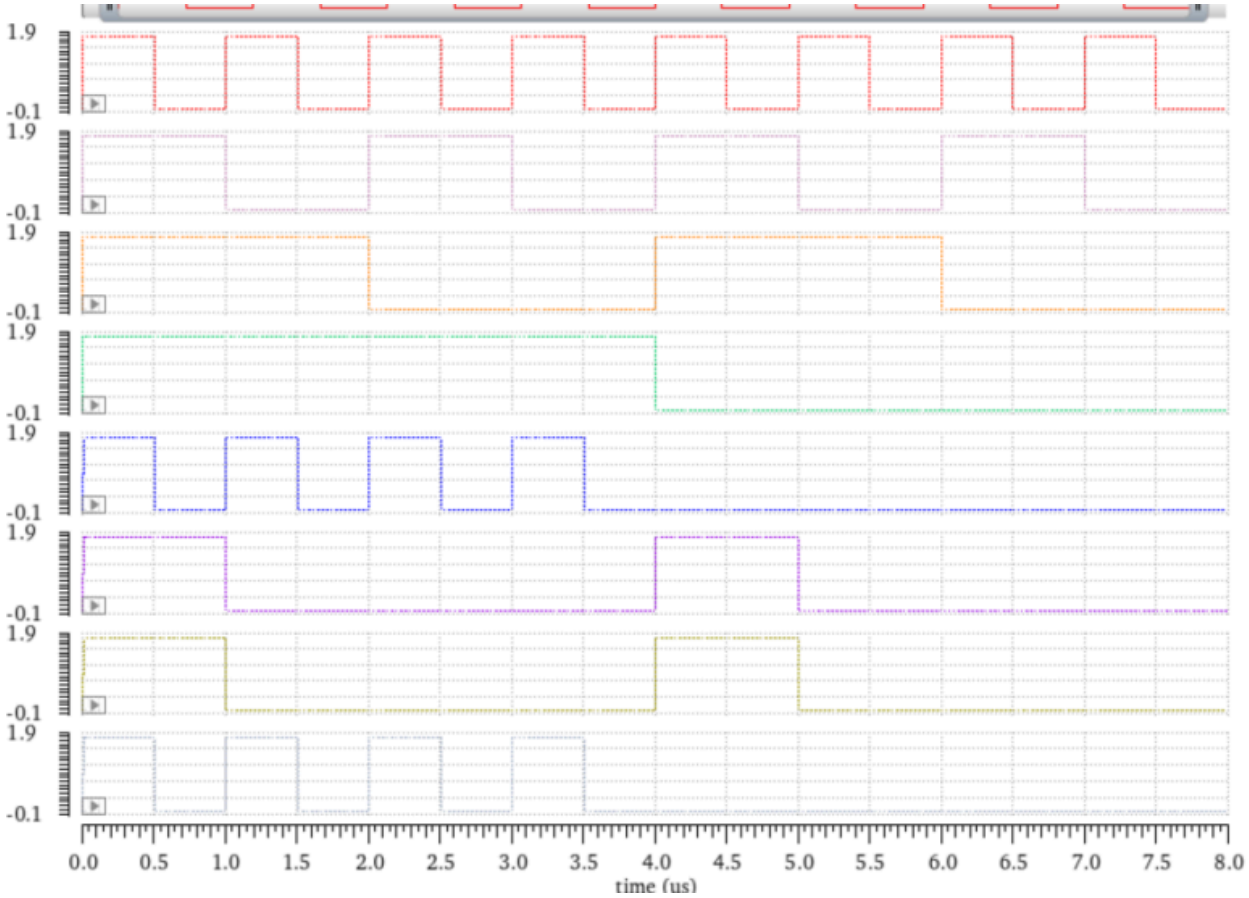

Figure 26: Bitwise AND Test

It can be a little tricky to read but this simulation indicates a correct bitwise AND operation. Let's look at the time period 0.5 - 1.0 µs. Here we see that A0 = 0 while B0 = 1, so out0 should be 0, which we can see in the blue wave. At the same time, A1 = 1 and B1 = 1, so out1 should be

1, which we see in the purple wave. This result is mirrored for out2 and out3, and if we keep this process up we'll see that the output is only 1 when both inputs are 1.

Bitwise OR

Following the exact same process as the AND simulation, I tested the bitwise OR block. Recall the input pulses are as follows: A0-A3 goes from top to bottom (only first four waves, B3-0 goes top to bottom (only first four waves).

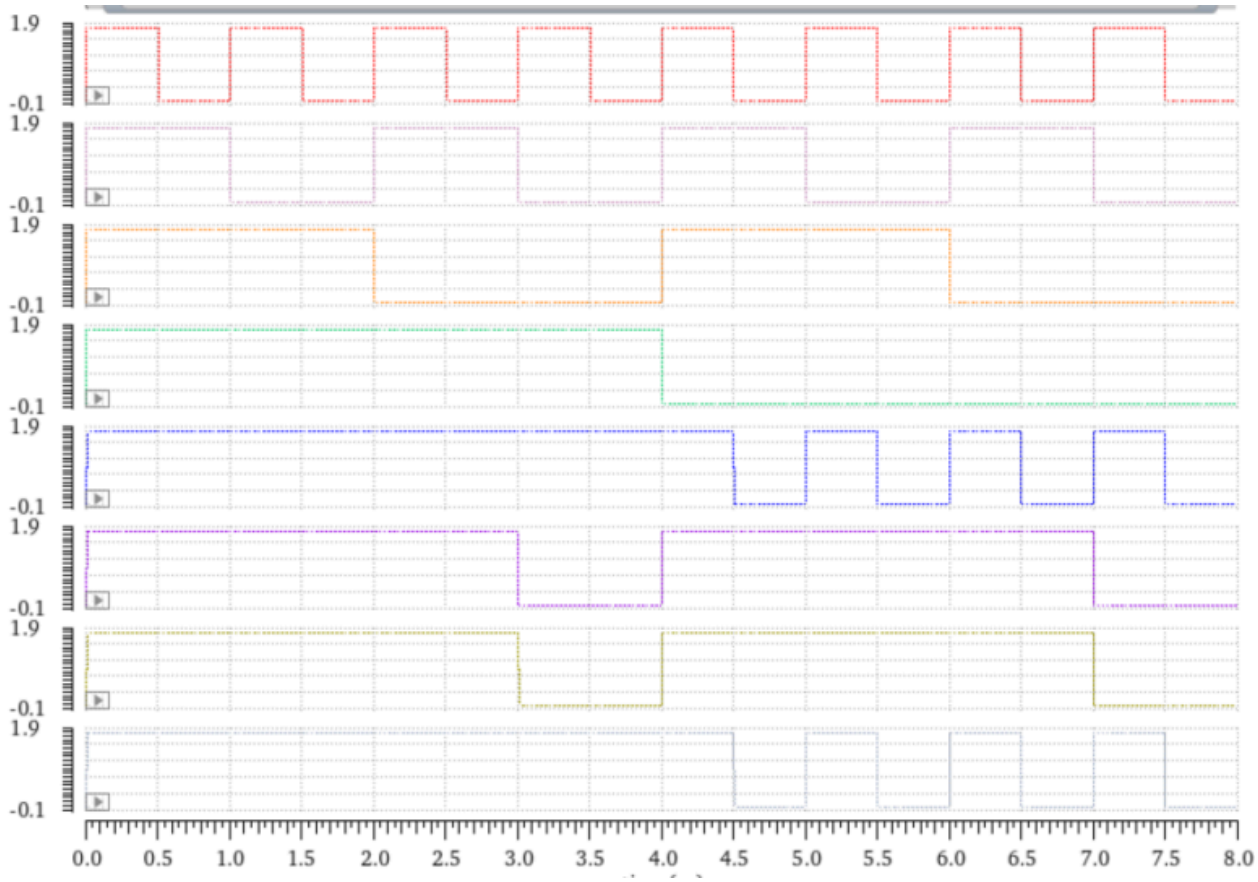

Figure 27: Bitwise OR Test

It's not difficult to see that the output for each bit only goes low when both inputs are low. Example: B2, A2 between 3.0 - 3.5 µs.

Bitwise XOR

Again the same process was used to test bitwise XOR. Recall the input pulses are as follows: A0-A3 goes from top to bottom (only first four waves, B3-0 goes top to bottom (only first four waves).

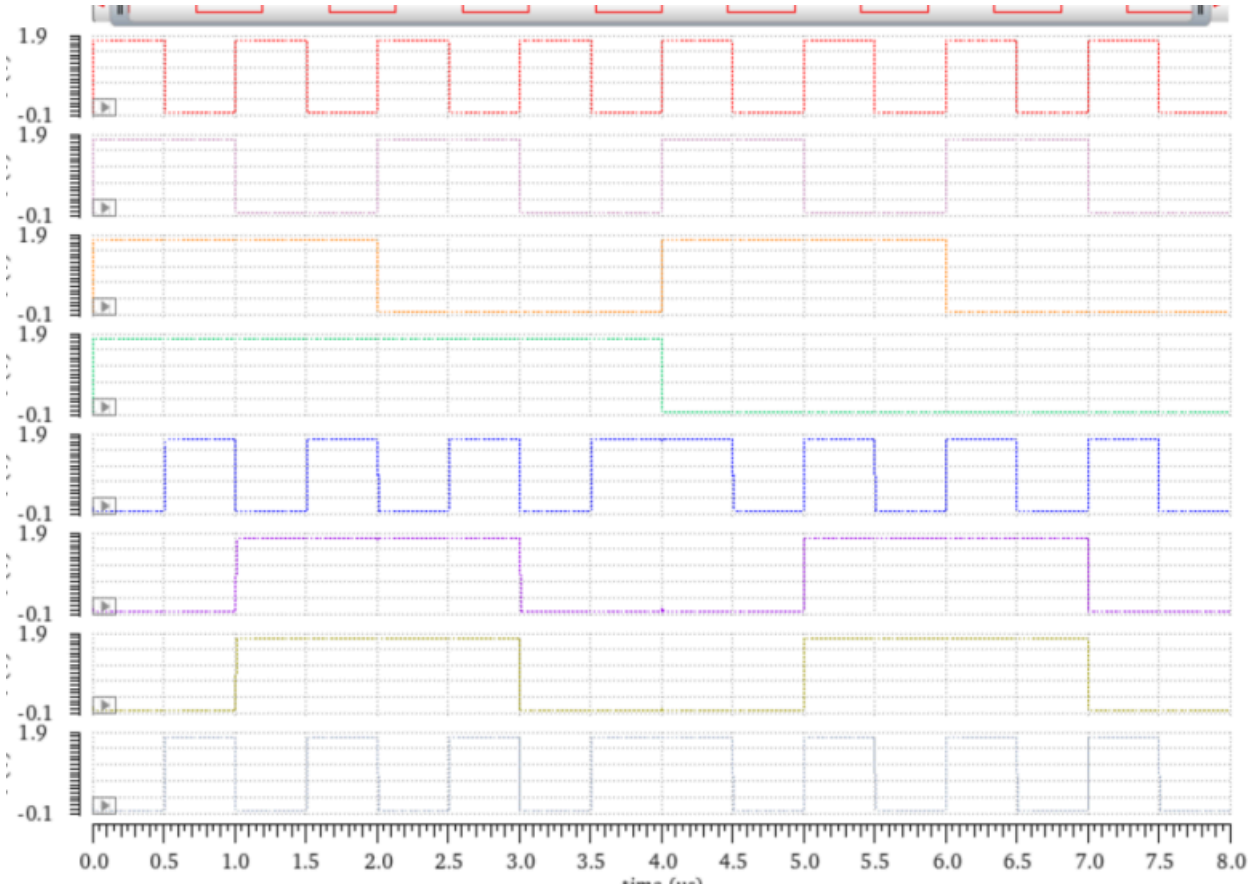

Figure 28: Bitwise XOR

This plot proves the correct bitwise XOR operation. The best example of this is from 4.5 - 5.0 µs, where both input bits are either high or both low, and you can see that all outputs are low.

Full Adder

Though this isn’t a dedicated function it is critical it works. For this block, it's pretty easy to work through all input options. The order of signal from top to bottom is: A, B, Cin, Cout, S.

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

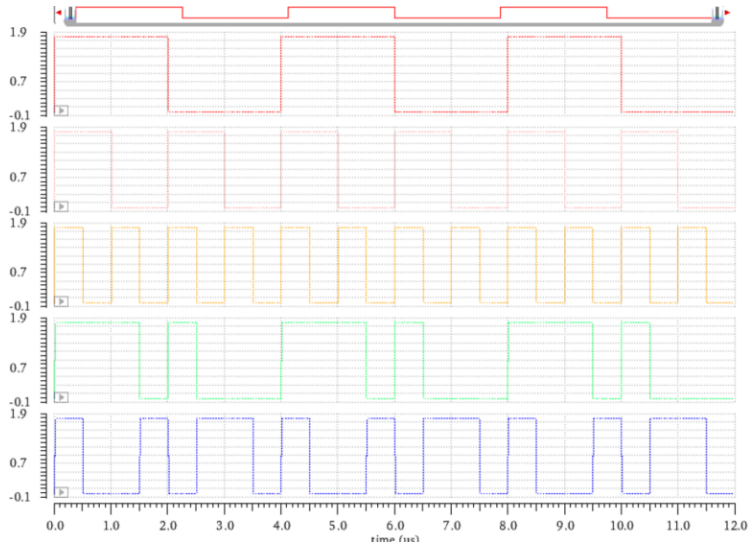

Figure 29: Full Adder Truth Table and Simulation [1]

4 Bit Adder

For we will perform one example of addition without overflow, one with overflow, and one with subtraction. Later, when testing the full system these three tests will be repeated with different values. Furthermore, instead of using pulse waves, the value will be hard coded, so we can visualize the addition clearer.

<u>Addition without Overflow:</u>

A = 0110, B = 1000, Cin = 0; signal order (top to bottom): s0, s1, s2, s3, Cout

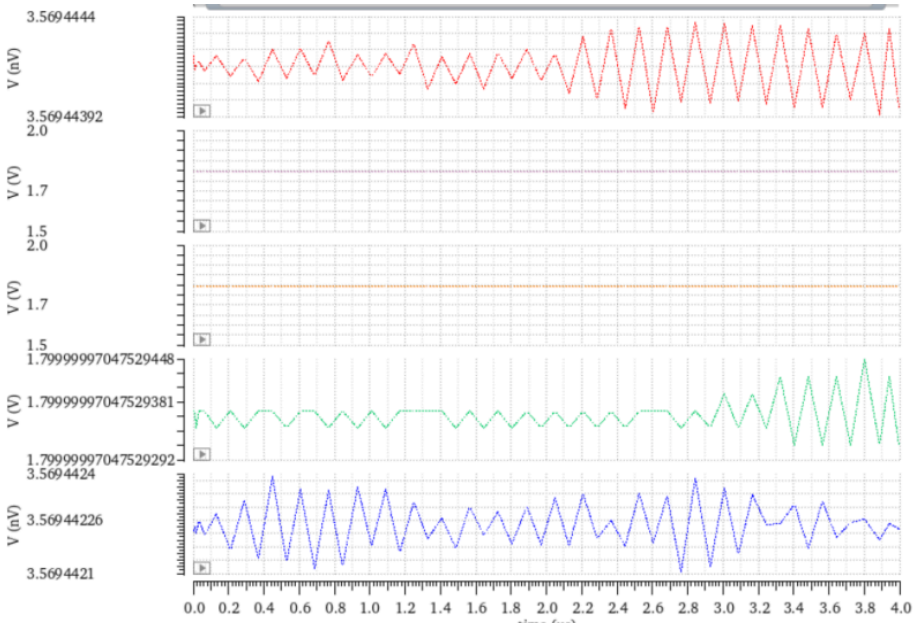

Figure 30: 4 Bit Adder Test, without Overflow

Output = 1110, Cout = 0; As expected!

Addition with Overflow:

A = 1100, B = 0110, Cin = 0; signal order (top to bottom): s0, s1, s2, s3, Cout

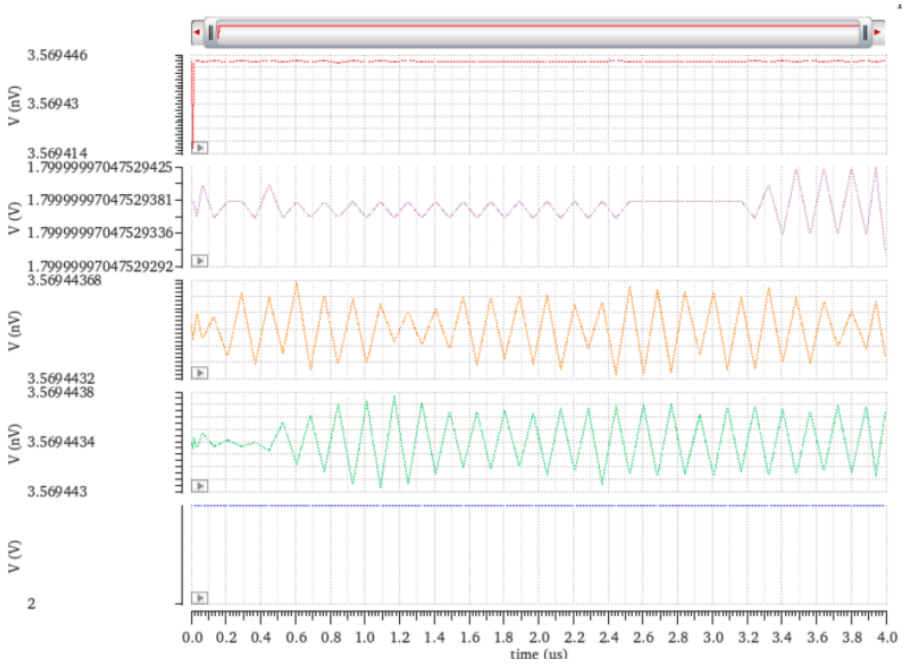

Figure 31: 4 Bit Adder Test, withOverflow

Output = 0010, Cout = 1; As expected!

Subtraction:

A = 1110, B = 1010, Cin = 1; signal order (top to bottom): s0, s1, s2, s3, Cout

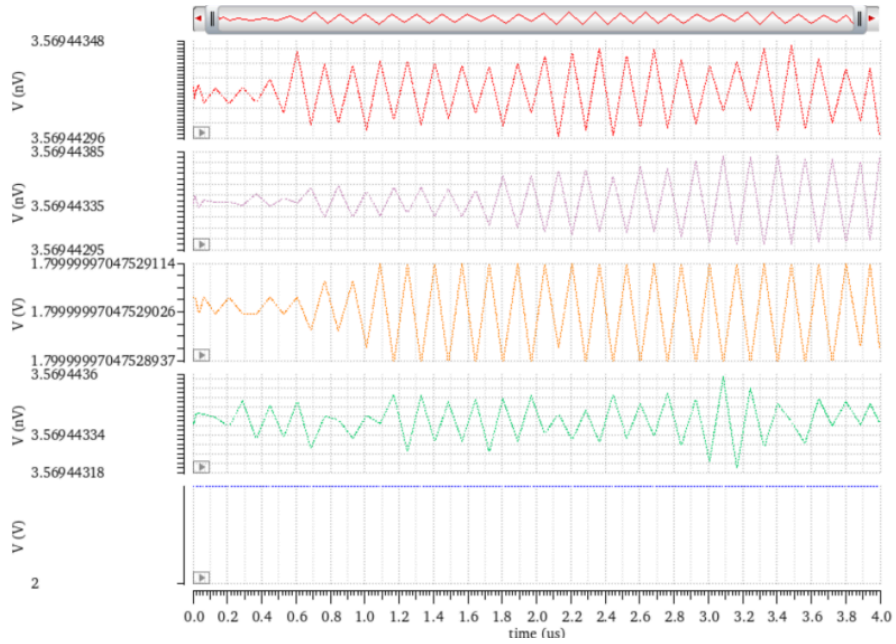

Figure 32: 4 Bit Subtraction Test, Two's Complement

Output = 0100, Cout = 1 (here the Cout is set but will be discarded); As expected!

Multiplexer

For this block I got a bit creative. I set each input to a specific value, and walked through all possible selection inputs. Here is a truth table indicating the parameters:

SEL0	SEL1	SEL1	Expected Value	Out<3:0>
0	0	0	0101 (Pass A)	1010
0	0	1	1010 (Pass B)	0101
0	1	0	0011 (And)	1100
0	1	1	1100 (Or)	0011
1	0	0	0110 (Xor)	0110
1	0	1	1001 (Add)	1001
1	1	0	N/A	N/A
1	1	1	1111 (High)	1111

The signal order is: select0, select1, select2, out0, out1, out2, out3

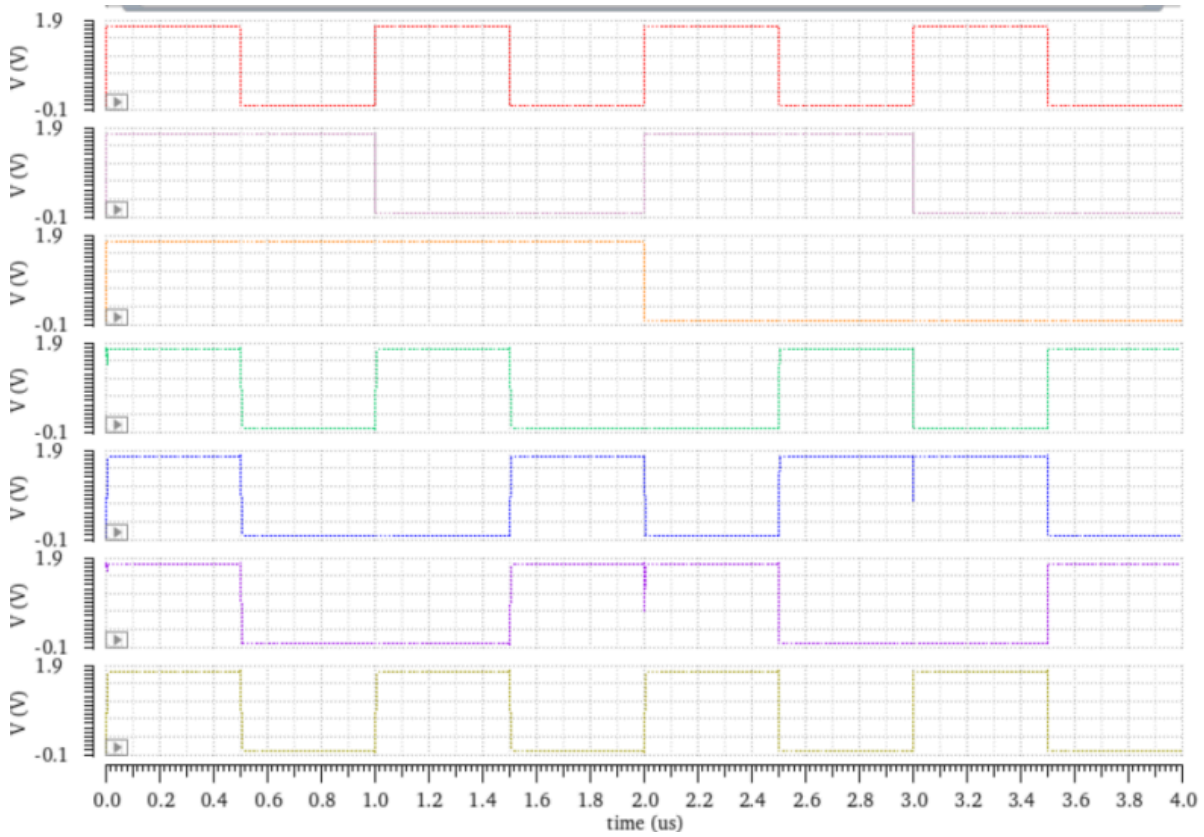

Figure 33: Multiplexer Test

To check each state, we have to read the graph, bottom to top. For example, between 3.5-4.0 µs, we can see the output is 0101 and the selection is 000, matching what I hard coded. Remember this is just testing the multiplexer so the values are just being passed through.

Instruction Decoder

This block only needs to have some special logic on a specific input. Here is the table of expected inputs/output.

INSTR2	INSTR1	INSTR0	Select <2:0>	flag
0	0	0	00	0
0	0	1	01	0
0	1	0	10	0
0	1	1	11	0
1	0	0	00	0
1	0	1	01	0
1	1	0	01	1
1	1	1	11	0

The order signal order is (top to bottom): INSTR0, INSTR1, INSTR2, select1, select0, sub_flag

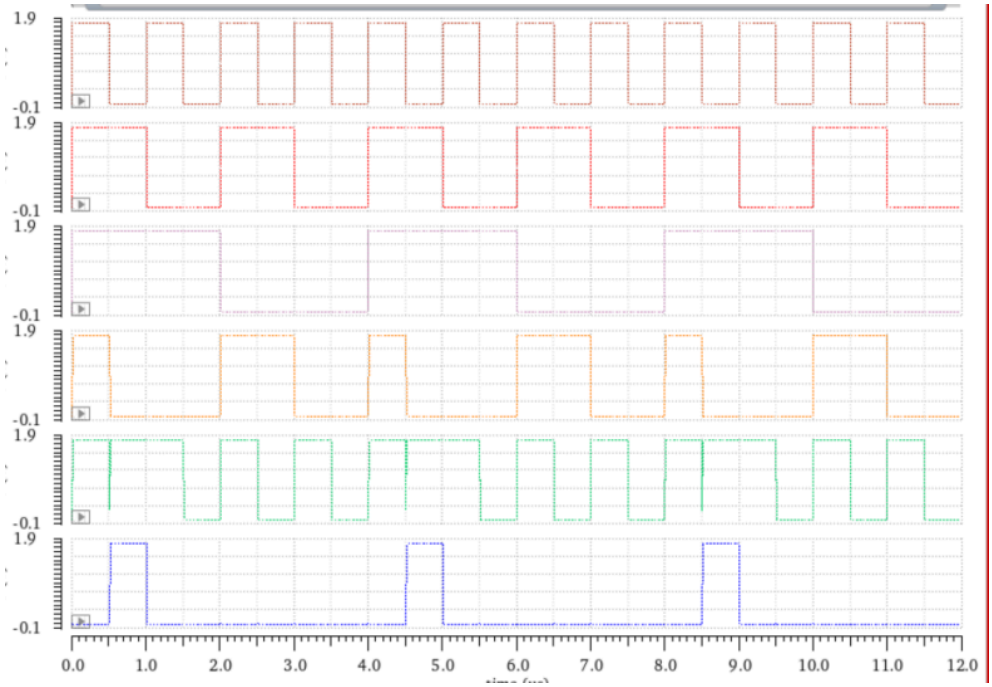

Figure 34: Instruction Decoder Test

Between 0.5 - 1.0 µs, we can see the input is 110. We can also see that the flag is active, and the select<1:0> are 01 (remember select2 is passed directly to mux). This matches the highlighted case in the table. One thing to note is that the signal that tells select0 to stay on in the special case is delayed because it requires logic gates. This means when the select0 signal turns off like it should for the case 110, there is a slight delay before the signal rises up to normal again. You can see this at every rising edge of the flag, the signal right above (neon green) has a dip. I could have mitigated this dip by adding delay to the other signal but for a project of this scale, it was not worth the effort.

Full System

Finally we are ready to test the full system. Again, full simulation of all inputs with the current tools is impractical, so one test will be shown for each instruction, one set of inputs will be used, except for the adder, where one addition without overflow, one addition with overflow, and one subtraction will be simulated. The inputs will be hard-coded, rather than set off by a pulse wave to make the outputs simple to decipher.

Pass A

A = 0101, B = 1010, INSTR = 000; and the output shown below matches A, 0101 (signal order on left side)

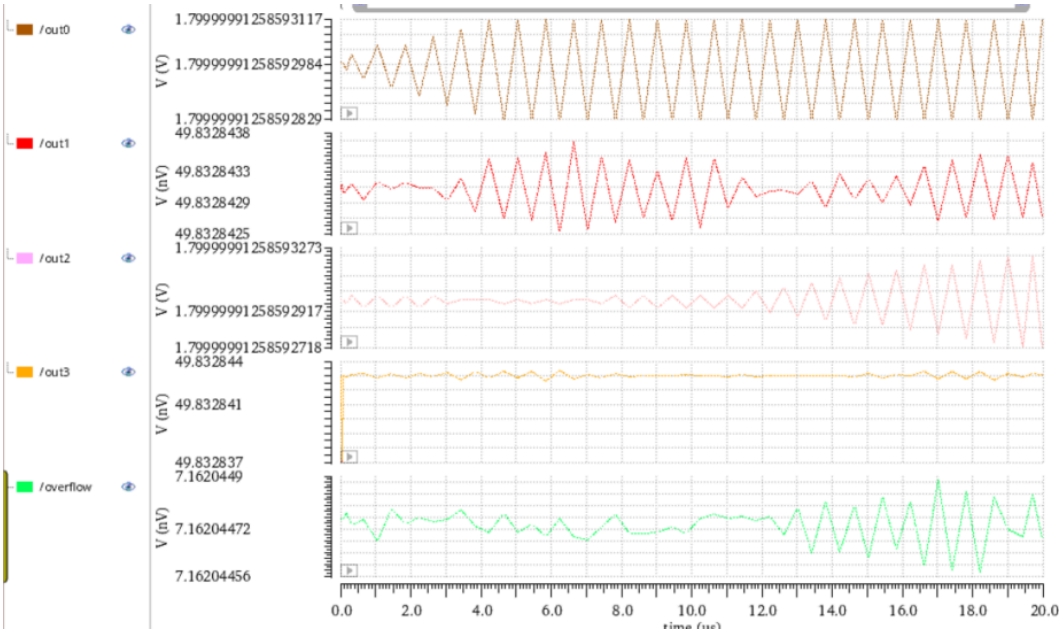

Figure 35: Pass A Test

Pass B

A = 0101, B = 1010, INSTR = 001; and the output shown below matches A, 1010. (signal order on left side)

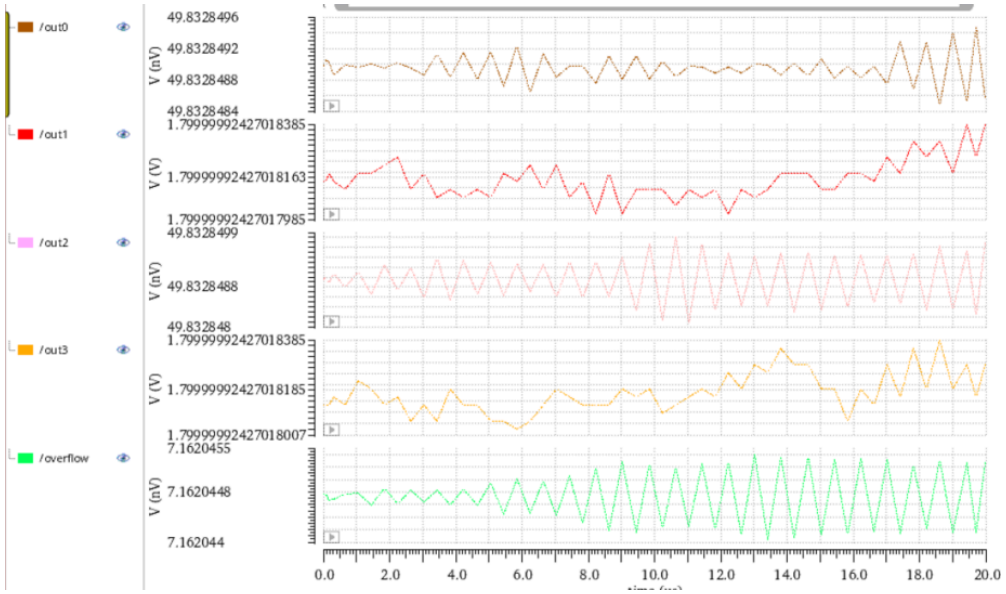

Figure 36: Pass B Test

AND

A = 0101, B = 1010, INSTR = 010; output = 0000, since A AND B = 0000 (signal order on left side)

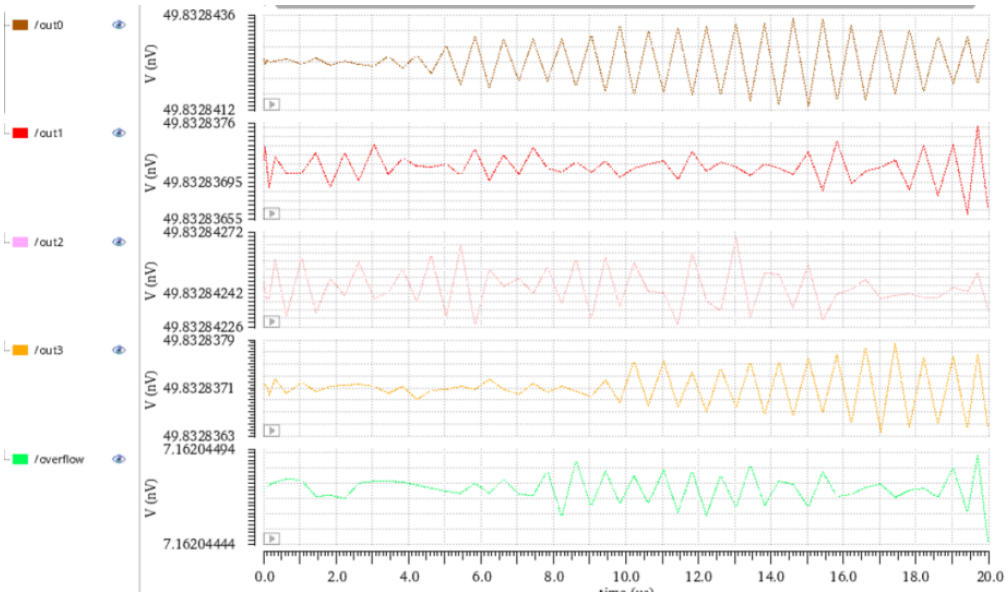

Figure 36: AND Test

OR

A = 0101, B = 1011, INSTR = 011; output = 1111, overflow = 0 (signal order on left side)

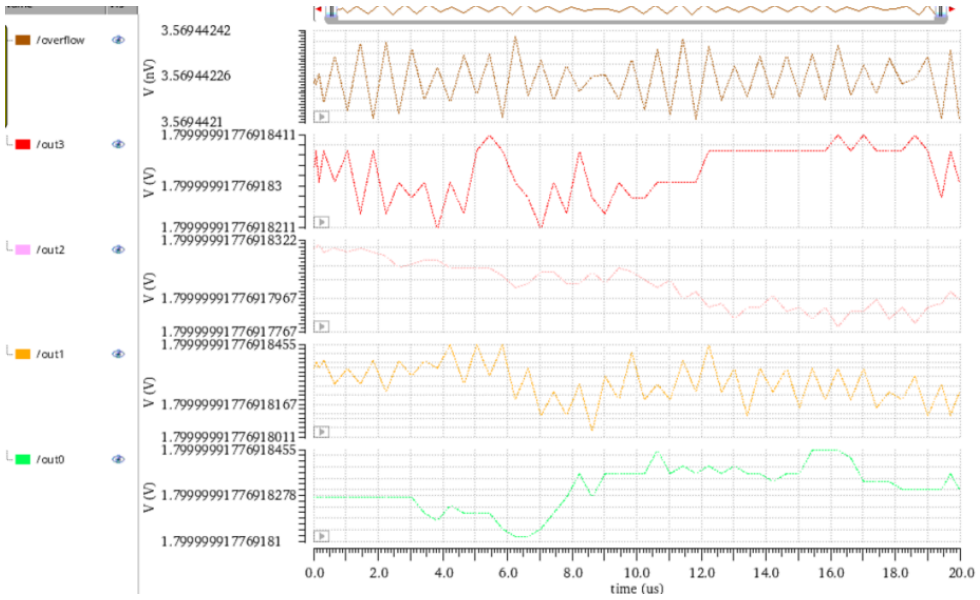

Figure 37: OR Test

XOR

A = 0101, B = 0011, INSTR = 100; output = 0110, overflow = 0 (signal order on left side)

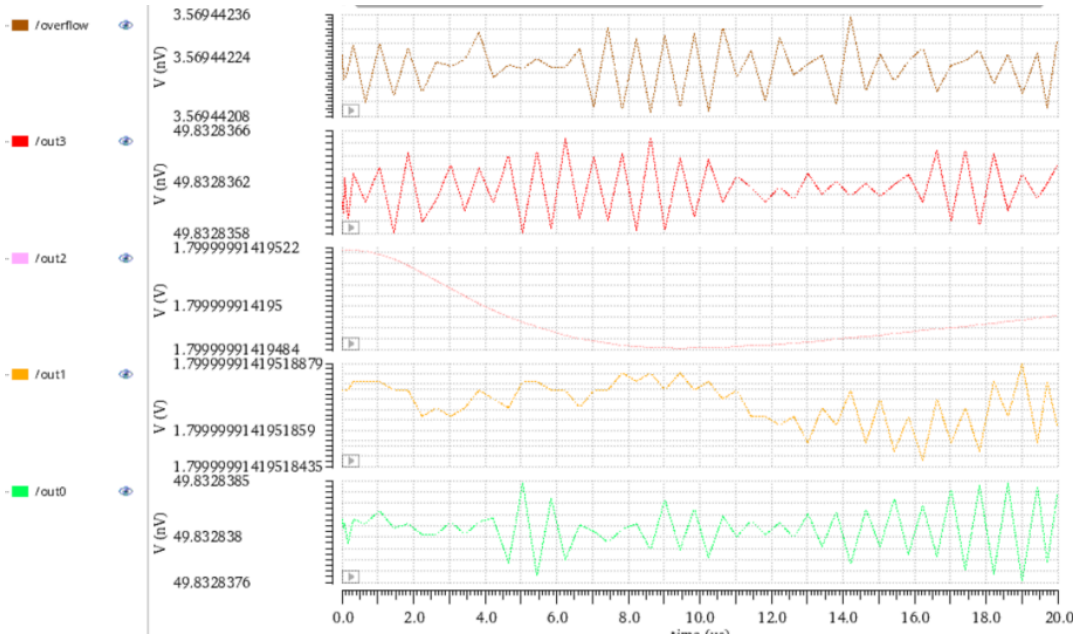

Figure 38: XOR Test

ADD without overflow

A = 1010, B = 0101, INSTR = 101; output = 1111, overflow = 0 (signal order on left side)

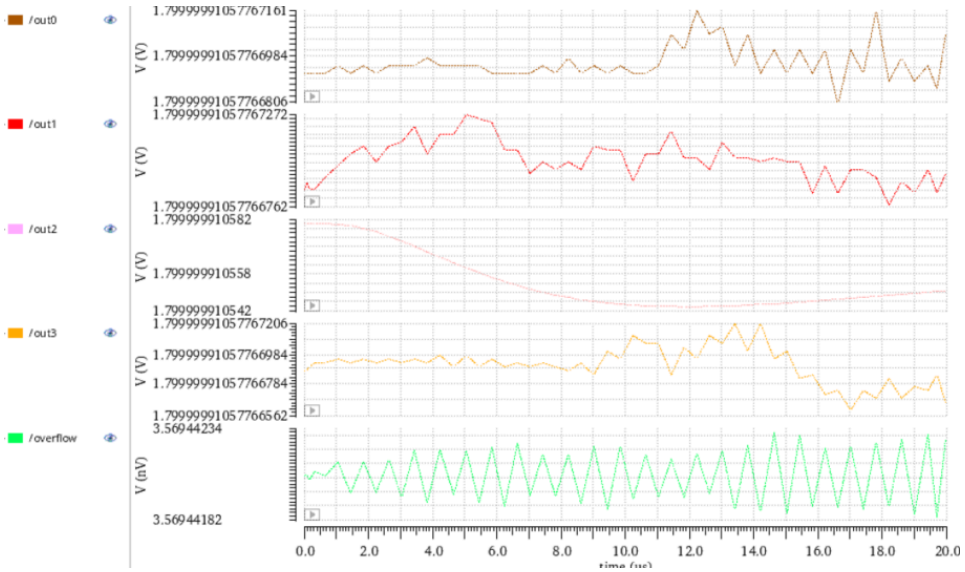

Figure 39: Add without Overflow Test

ADD without overflow

A = 1010, B = 0111, INSTR = 110; output = 0001, overflow = 1 (signal order on left side)

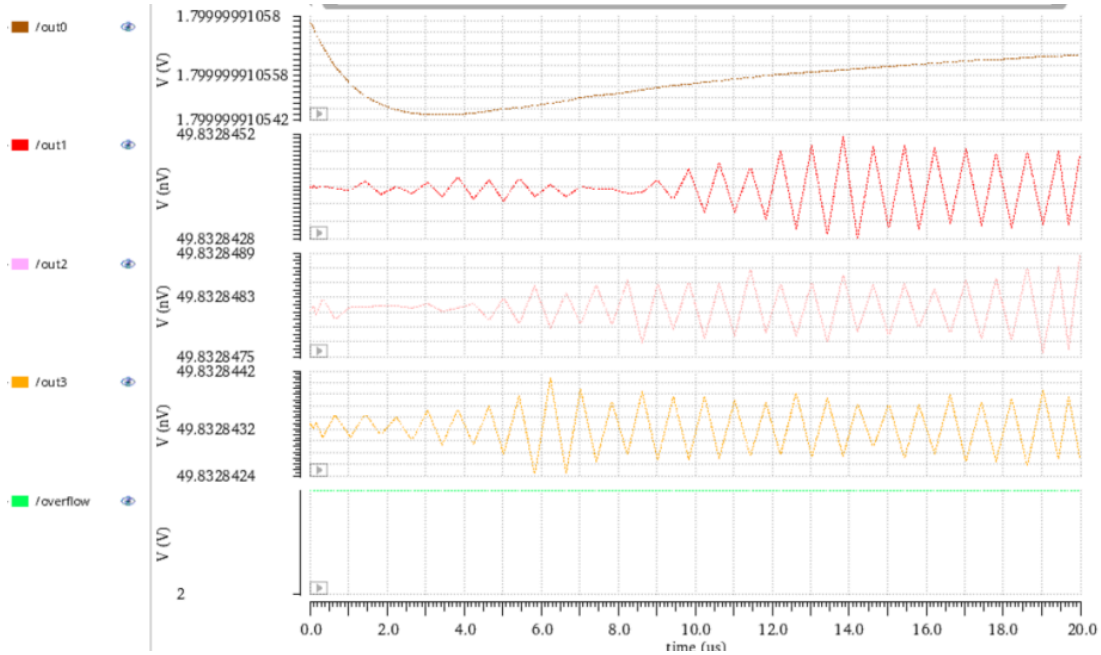

Figure 40: Add with Overflow Test

SUB

A = 0110, B = 0111, INSTR = 110; output = 1111, overflow = 0 (signal order on left side)

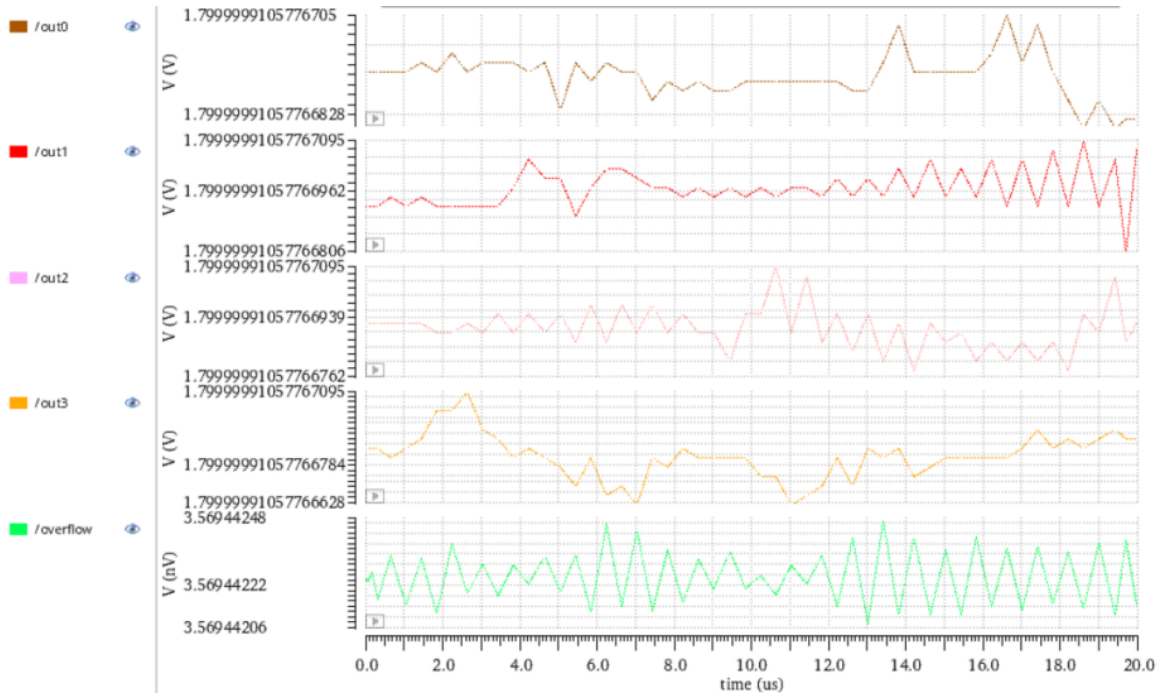

Figure 41: Subtraction Test

Pass HIGH

A = 0110, B = 0111, INSTR = 111; output = 1111, overflow = 0 (signal order on left side)

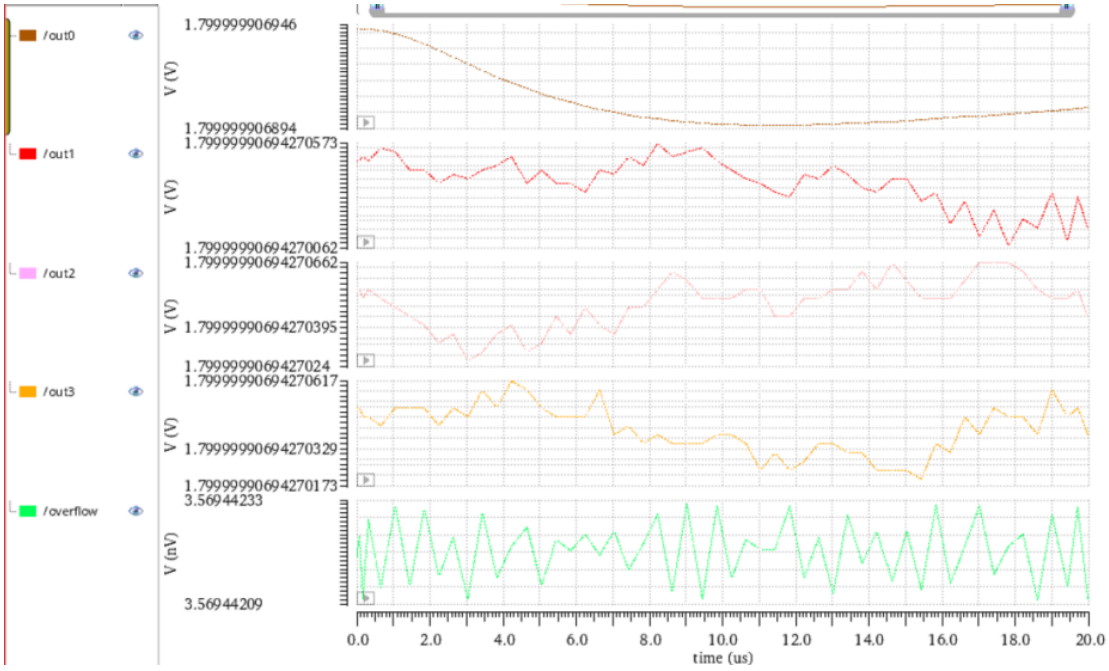

Figure 42: Pass High

Again, this test plan was far from exhaustive, but hopefully it demonstrated sufficient functionality for the instruction set of the ALU.

Closing Thoughts

Overall this project was challenging due to the scale of implementation and the testing required to provide significant evidence of the functionality. While this lab did not introduce too many new concepts, coming up with designs that are both efficient and feasible to complete in the time was tough. One big design choice I made is to use individual bits instead of Cadence's bus feature. I chose this to make it easier to visualize every signal, since the bus system was new and I didn't know how it would work for the mux or adder. Also, I needed to simulate every signal in my testing, so I thought it was easier to just stick to what I know (though I learned later there is a nice way to do this in Cadence). Testing this project was also quite time consuming. It makes me wonder how I could automate this process, that would make it easier to set up and to illustrate successful (or unsuccessful) results.

References

[1] "VLSI Digital Circuits Adder Design." Accessed: Dec. 12, 2024. [Online]. Available: https://uweb.engr.arizona.edu/~ece507/lecture_12.pdf

[2] Z. Pritchett and C. Hogan, "Adder Design and Analysis," 2012. Accessed: Dec. 12, 2024. [Online]. Available: https://zpritchett.wordpress.com/wp-content/uploads/2012/03/technical-lab-report.pdf

[3] "4-bit binary Adder-Subtractor," *GeeksforGeeks*, Aug. 27, 2019. https://www.geeksforgeeks.org/4-bit-binary-adder-subtractor/

[4] Electronics Tutorials, "Multiplexer (MUX) and Multiplexing Tutorial," *Basic Electronics Tutorials*, Aug. 2013. https://www.electronics-tutorials.ws/combination/comb_2.html