CompArch Lab 1 Writeup

Logan Sweet & Maggie Jakus

October 12, 2017

1 Implementation

We went into this lab with a tenuous grasp of how an ALU worked. Once we reviewed the in-class materials and several internet sources, we decided that we would want to create an ALU of n bitslices. We wanted one bitslice to consist of an AND/NAND module, an OR/NOR/XOR module, and an ADD/SUB/SLT module, and a series of switches would determine the output of each individual bitslice. We would then connect these 32 bitslices in series to form a 32-bit ALU. However, when we tried to combine our three modules to create one bitslice, nothing worked. We tried to debug for a while, but eventually decided to try what ended up being our final configuration.

We created an n-bit AND/NAND module, an n-bit OR/NOR/XOR module, an n-bit ADD/SUB module, and an n-bit SLT module, and then used multiplexers to determine which to use as the final output. We only calculated carryout, zero, and overflow in the n-bit ADD/SUB and SLT modules. We did not calculate carryout, zero, and overflow in the 1-bit ADD/SUB/SLT, since we wanted to combine our 1-bit modules to form an n-bit module, and only the $n-1$th values would matter to calculating carryout and overflow. Zero depended on the entire value of the output, so we calculate that at the end as well.

We recognize that our second attempt is really no different from our initial attempt and therefore we didn't need to switch tactics, but this ended up being less effort on our part. An overview of our final schematic can be seen in figure 1.

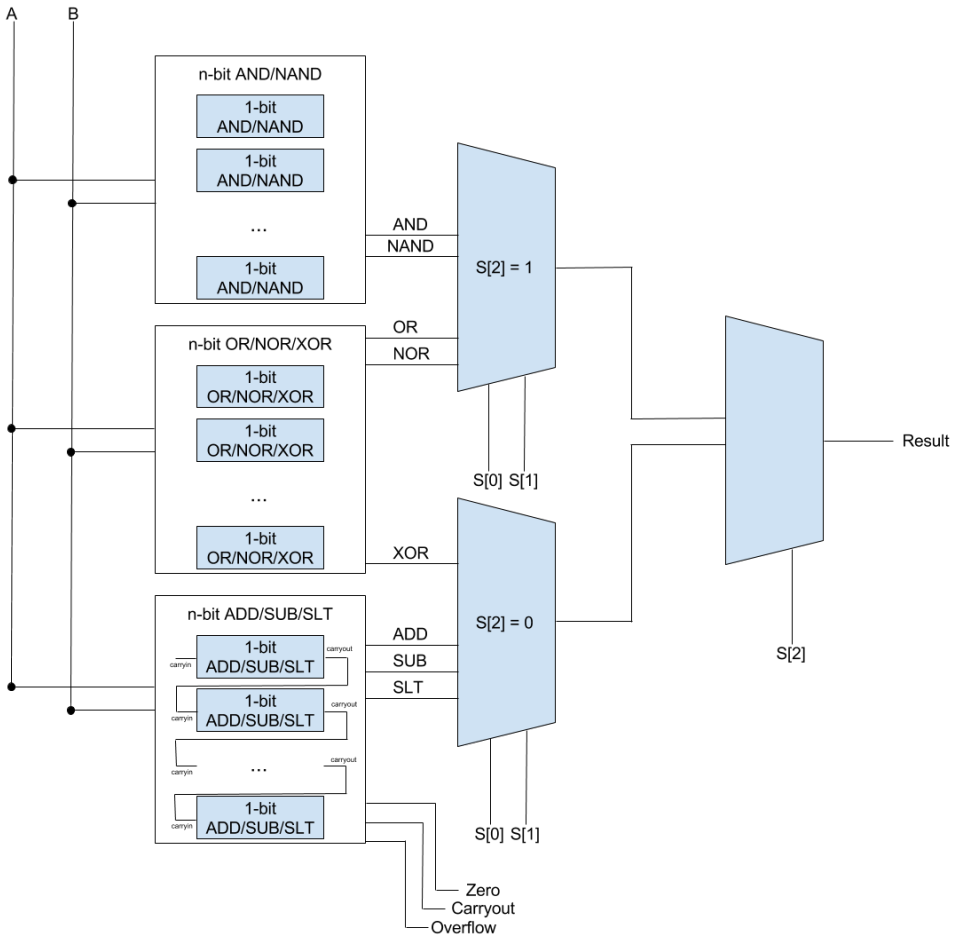

Figure 1: An overview of the n-bit ALU. We ended up separating ADD/SUB from SLT, but this change was made after the graphic was done. Imagine that there is another n-bit SLT module that looks almost identical beneath the n-bit ADD/SUB module. The n-bit SLT module is still made of the 1-bit ADD/SUB/SLT modules, but the n-bit SLT module does additional operations on its final result.

We created our AND/NAND module exactly as we had done in previous homework and lab assignments, and we will not show it here. Similarly, our OR/NOR/XOR module was fairly simplistic, so we will not include a schematic. Our ADD/SUB/SLT module was more challenging for us - we were not familiar with the SLT functionality or the Zero flag, and figuring out how to trigger these took some time.

A schematic showing our 1-bit ADD/SUB/SLT module can be seen in figure 2. In our final n-bit ADD/SUB and SLT modules, we determine overflow based on the carryin and carryout values (if they are not equal, overflow has occurred). We calculate SLT in the final n-bit module as well; SLT only occurs when $A - B < 0$, which means that either the most significant bit of the result is 1 and there is no overflow, or the most significant bit is 0 and there is overflow. We ended up separating SLT from ADD and SUB because our code was being problematic.

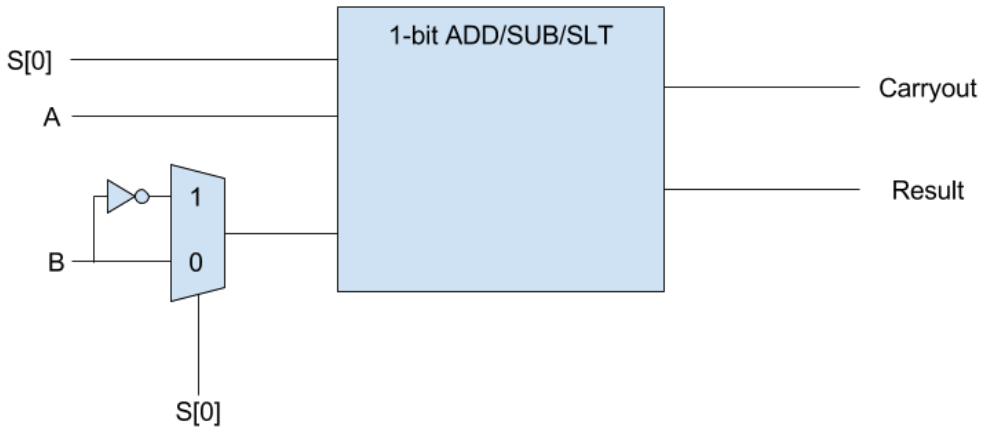

Figure 2: The 1-bit adder/subtractor. The internal structure is the same as it was in lab 0.

2 Test Results

We structured our test benches similarly to how we structured our ALU. Since the functionality of our ALU relied on our multiplexers working, we began by testing those. We then tested the four input multiplexer (the "FourInMux" module). We wanted to confirm that it would choose the correct value based on the switches. Because there are many possible inputs, we used the variable x for the input that should be selected. Our tests were successful from the beginning, which was not surprising since we made a multiplexer in a previous assignment.

We began by writing and testing the one-bit AND/NAND module, as this was the easiest module for us to understand. We tested all possible scenarios (4 for each), and this gave us confidence in our AND/NAND module. We did notice incorrect values when we first tested the module. We went back to our module code and realized that, when we set the output value ("AndNandOut"), we were choosing between "A" and "B" rather than "AandB" and "AnandB".

We then wrote and tested the one-bit OR/NOR/XOR module. This was also something with which we were already familiar, so writing and testing went smoothly. We tested these exhaustively as well, since there were still only four different scenarios for each function.

We then only had one basic module left to test - the ADD/SUB/SLT module. We started by just testing ADD, since this is the simplest of the three operations. We began by exhaustively testing 1-bit addition, which went easily. Subtraction was also easy; all we had to change was the command and the carryin. When subtracting using an adder/subtractor, the initial carryin is 1. While we included that in the code for our 32-bit adder/subtractor, it didn't feel worth it to add it to the 1-bit. We didn't really test 1-bit SLT.

We then created and tested the 32-bit adder/subtractor. We initially struggled with using generate to combine 32 individual 1-bit modules, since there is very little documentation online. We also were initially confused about how we would use generate while also having specific cases for the first and last bitslices. We decided to separate the 0th case from the generate loop, and we calculated end-specific variables after we had generated the whole value. For addition, we tested the same cases we had tested in Lab 0. We tested addition with and without overflow with positive and negative numbers. We tested all different types of scenarios twice, which convinced us that our adder was working. We tested subtraction with SLT, as it is part of SLT. For SLT, we tested $A < B$, $A = B$, and $A > B$ for A and B positive and negative. We made sure to include cases with and without overflow.

For the n-bit AND/NAND and OR/NOR/XOR, we tested a few different examples each. This was less important to test rigorously, since in these, the bits are unrelated to one another, while in the adder/subtractor, they are connected by carryin/carryout.

Finally, we tested the n-bit ALU as a whole. Since we believed the functionality of each individual unit, all we needed to do was confirm that the ALU could select the proper action. We did one example for each action, and they all worked as expected. We then confirmed that each could flag zero if the answer was zero.

We never tested more than a four-bit ALU, since anything larger would be annoying to type in. However, this isn't an issue, and the length can be changed as needed. We were intentional in structuring our code so that the parameter "size" can be changed to any length, and then all variables of size "size" will update to the new value.

We very rarely found that our test benches caught an error in functionality; we rarely got a 1 instead of a 0 or vice versa. We did find that running the test bench instead alerted us to errors simply running the main .v file. This does a pretty good job of reflecting that most of our struggles in this lab were around syntax rather than understanding the functionality of our ALU.

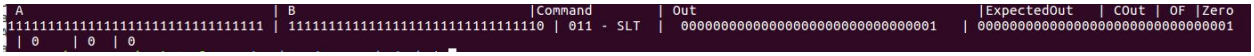

Figure 3: We tested the 32-bit ALU module with 32 bits and it worked!

3 Timing Analysis

After struggling with GTK Wave for a while, we decided to manually calculate the timing delays. We already had a solid understanding of the ALU as a whole, but using paper (instead of GTK Wave) made us double check our work and further deepen our understanding of the ALU.

We started by calculating the delay of a two-input multiplexer, which is the simplest module in our system. The delay on the two-input multiplexer is 50 units. We then calculated the delay for a four-input multiplexer, which we found to be 70. We then calculated the delay for a 3-input and 4-input NAND gate. The delays are both 30.

Moving up in complexity, we calculated the worst-case delay of the 1-bit AND/NAND module, which we found to be 70. In an n-bit AND/NAND module, each bit is separate from those before and after it, so all n bits can be operated on simultaneously. That means that the worst-case delay for an n-bit AND/NAND module is still just 70 units. Likewise, we calculated that the worst-case delay for a 1-bit OR/NOR/XOR module is 40 units (10 for NOR, 20 for OR, and 40 for XOR), and since each bit is independent of all others, the worst-case delay for an n-bit OR/NOR/XOR module is 40 units.

The ADD/SUB and SLT modules were harder to calculate delay for. For a one-bit adder/subtractor (which is also used in calculating SLT), the delay on the sum is 100 and the carryout delay is 120. However, the delay for calculating the sum delay for an n-bit addition/subtraction is $120 \times (n-1) + 100$, since no bit can add until the carryout from the one previous has been calculated. Calculating overflow requires waiting until the final carryout has been calculated (so $120 \times n$ units), and then adding an additional delay of 60. Calculating whether the zero flag should be set requires waiting for the final sum, then an additional delay of 40. Calculating the delay for SLT is a little different, due to the nature of how we had to structure our code. Because we have two additional multiplexers in calculating each sum, the one-bit sum delay is 200, and the carryout is still 120. Therefore, the sum delay is $200 \times n$, and the carryout delay is $120 \times n$. Additionally, there are zeros and overflow delays (40 additional after sum and 60 after carryout, respectively), and there is the final SLT value (all zeros or all zeros with a final 1), which adds an additional delay of 180 after the final sum value. These can be summarized in table 1. Photos of our calculations are in figures 4 to 7.

Table 1: The calculated delays.

	OutDelay				
2-Input Mux	50				
4-Input Mux	70				
3 and 4-Input NAND	30				
1-Bit AND/NAND	70				
n-Bit AND/NAND	70				
1-Bit OR/NOR/XOR	40				
n-Bit OR/NOR/XOR	40				
	SumDelay	**COutDelay**	**Zero Delay**	**OFlow Delay**	**FinalSLTSumDelay**
1-Bit ADD/SUB	100	120			
n-Bit ADD/SUB	$120 \times (n-1) + 100$	120 x n	SumDelay + 40	COutDelay+60	
1-Bit SLT	200	120			
n-Bit SLT	200 x n	120 x n	SumDelay + 40	COutDelay+60	SumDelay+180

4 Work Plan Reflection

We definitely missed the mark on our work plan. We knew that we would be able to figure out the logic/ structure of the ALU in the time we planned, but didn't realize that we would have so many syntax errors and issues along the way.

Task	**Time alotted**	**Due Date**	**Actual Time**
Decide on ALU structure & confirm understanding	3 hours	Oct 5	3 hours
Decide on test benches	2 hours	Oct 6	1 hours
Write verilog for each module	3 hours	Oct 8	12 hours
Write test benches with self-checking	3 hours	Oct 8	∞ hours
Check test benches work	1 hour	Oct 9	1 hours
Possible debugging time	2 hours	Oct 10	included in Verilog time
Timing analysis	1-2 hours	Oct 10	1 hour
Write report	3 hours	Oct 12	3 hours
Total	18 hours	-	23 hours

Table 2: We never added self-checking to our test benches, which is why that part took ∞ hours. In reality, writing the test benches took around 2 hours. Writing the code for the ADD/SUB/SLT modules was much more difficult to understand than we had anticipated. We're also not sure how accurate this is, as we often did not keep track of how long we had been working on this, and we found it helpful to take breaks to clear our minds rather than try to struggle through problems.

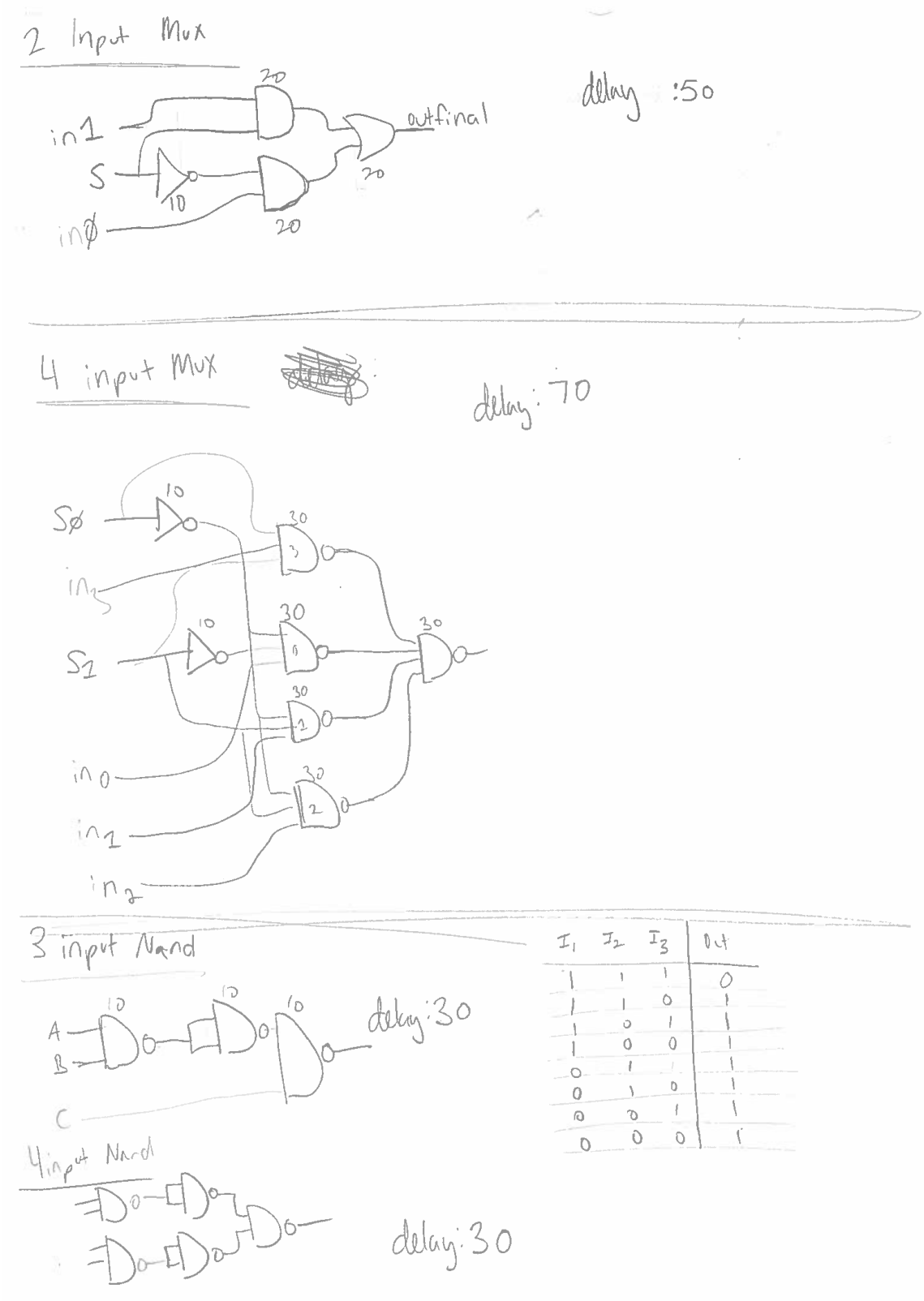

Figure 4: Calculating delay for multiplexers and 3 and 4 input NAND gate.

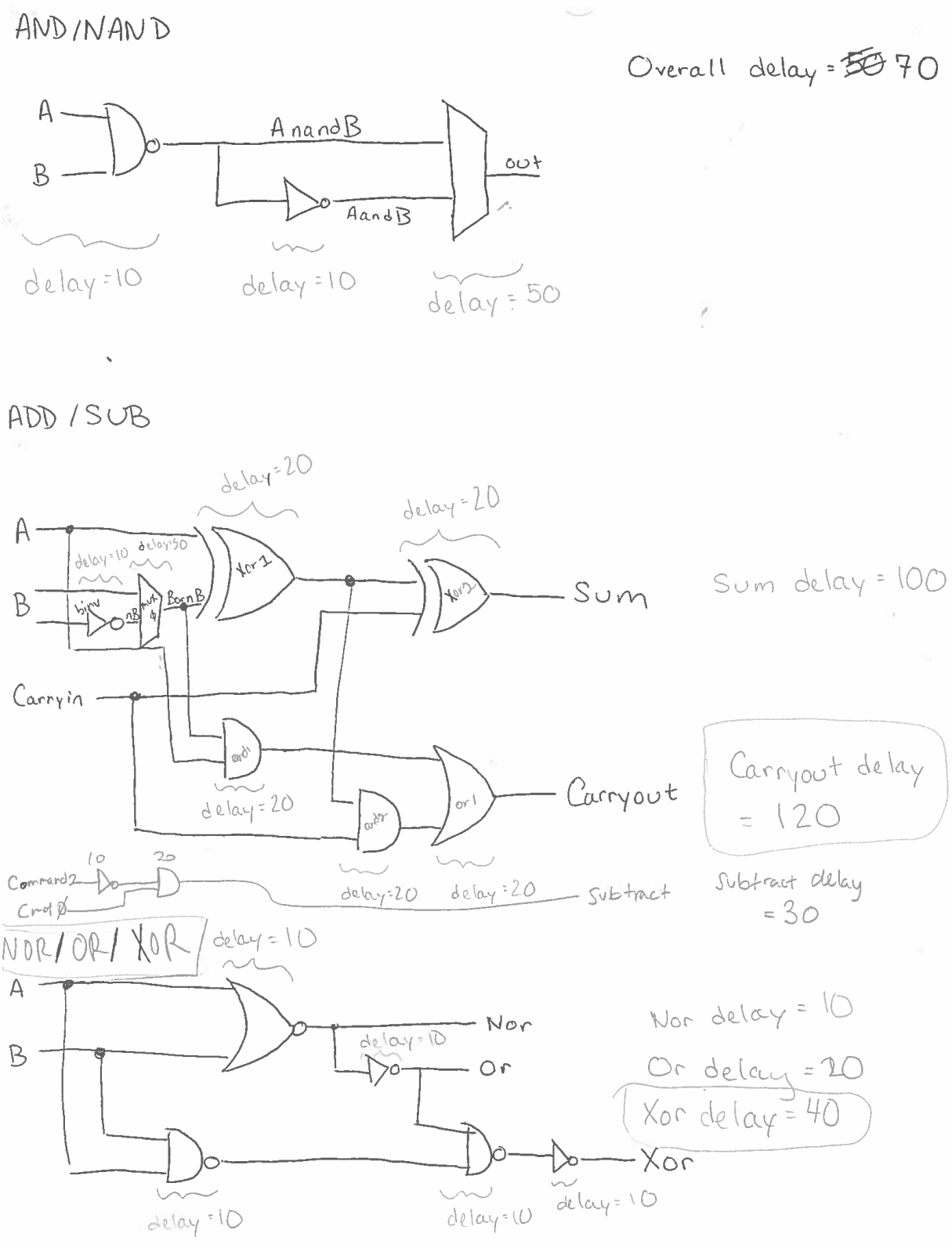

Figure 5: Calculating delay for 1-bit AND/NAND, ADD/SUB, and OR/NOR/XOR.

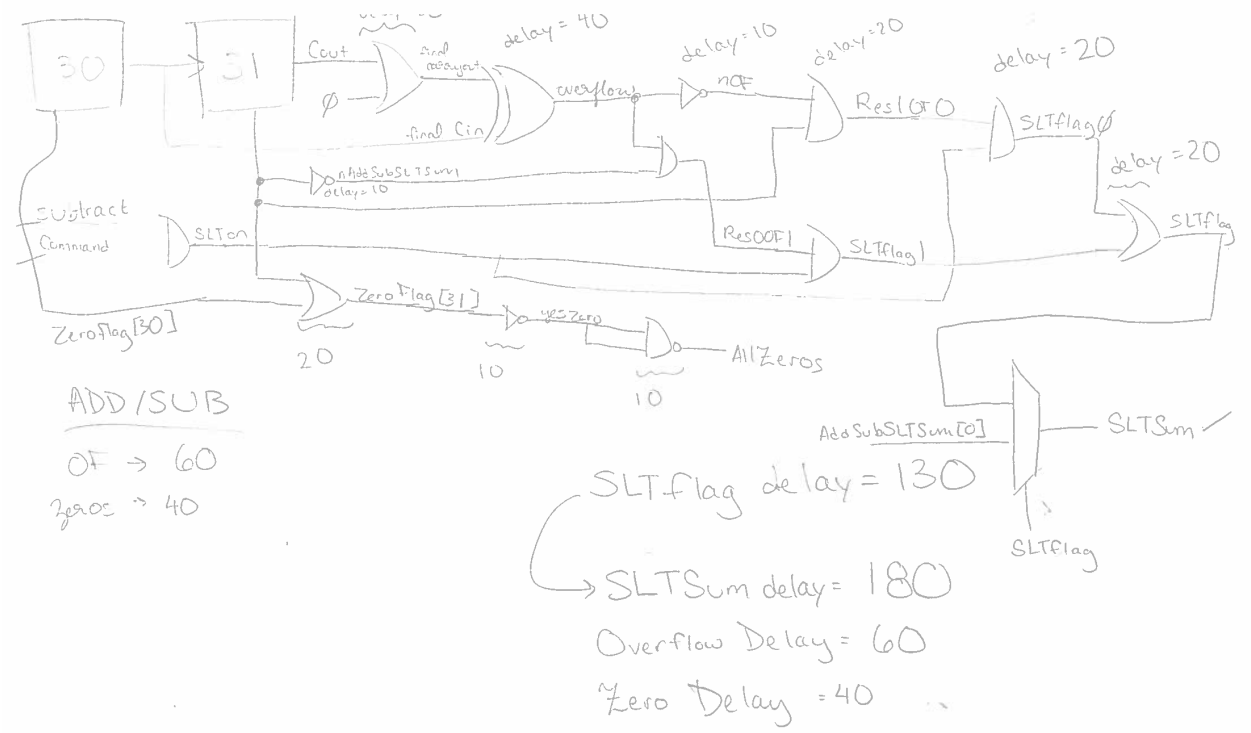

Figure 6: Calculating ADD/SUB/SLT delay.

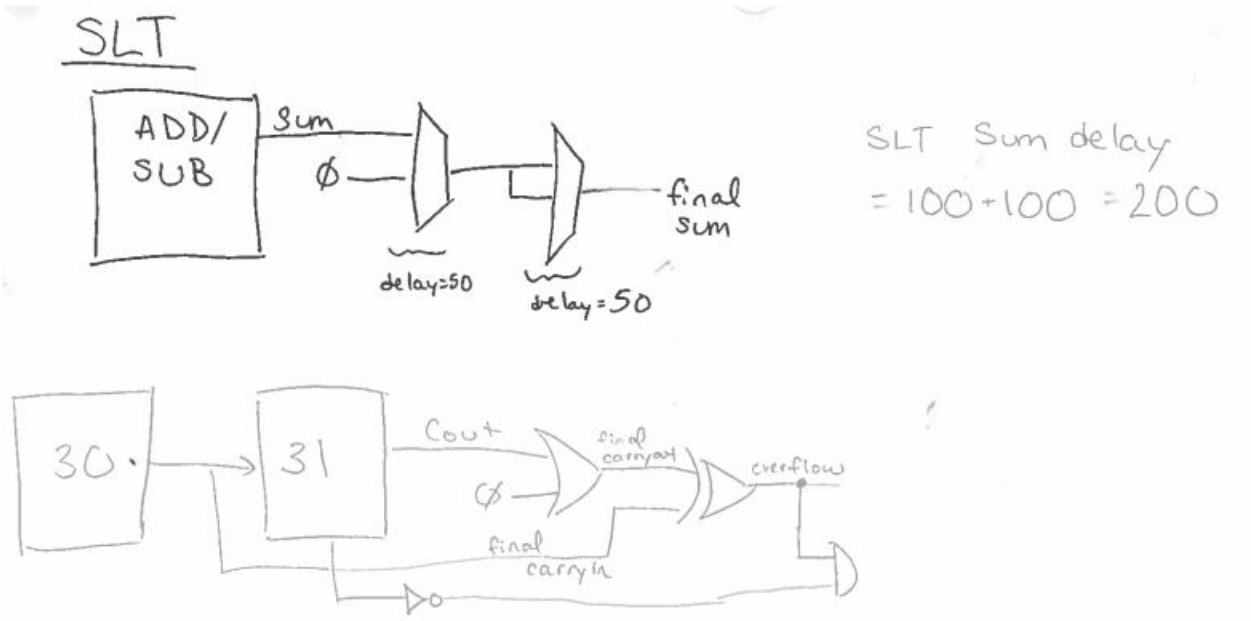

Figure 7: Calculating 1-bit SLT delay.