# **CprE 381, Computer Organization and**

# **Assembly Level Programming**

# **Lab 1 Report**

Student Name: Jackson Hafele

**[Part 1.c] Think of three more cases and record them in your lab report.**

The following cases were used to test TPU\_MV\_Element in the file *tb\_TPU\_MV\_Element.vhd*. The goal was to create test cases with varying weights iW, activation input iX, and partial sums iY.

| **iW** | **iX** | **iY** | **oX** | **oY** |
| --- | --- | --- | --- | --- |
| 1000 | 5 | 25 | 5 | 5020 |
| 0 | 5 | 25 | 5 | 20 |
| 1 | 3 | 2 | 3 | 5 |

*Table 1. tb\_TPU\_MV\_Element Test Cases*

**[Part 1.e] For labels 1, 7, 22, and 28, specify where (VHDL file and line number) these values are located – some will be found in more than one place. Also attempt to explain the functionality of each label as it occurs in the code**

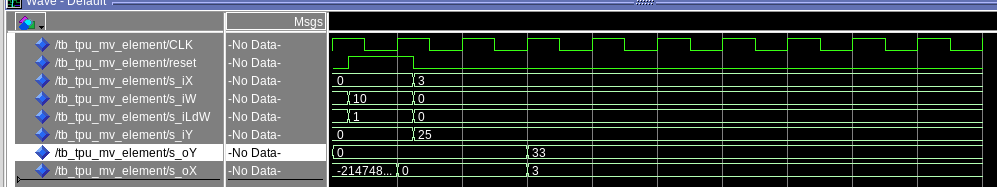
In the attached diagram area, (1) is the top level TPU\_MV\_Element Module, which is defined in the TPU\_MV\_Element.vhd file. TPU\_MV\_Element is defined as an entity on line 19, and the components of TPU\_MV\_Element are defined starting at line 32.

In the attached diagram area, (7) is the g\_Add1 module that is an Adder component. The Adder component is referenced on line 36 in the structure of TPU\_MV\_Element.vhd, and g\_Add1 is instantiated on line 117, including the inputs iCLK, s\_WxX, and s\_Y1, and the output oY.

In the attached diagram area, (22) is the oQ output of the RegLd module, which is defined as an integer on Line 64 of TPU\_MV\_Element.vhd For this specific instance of the RegLd module, the output oQ is connected to a signal called s\_W on line 86.

In the attached diagram area, (28) is the signal s\_X1 wire in the TPU\_MV\_Element Module. s\_X1 is defined as an integer on line 69. s\_X1 is assigned as the output of oQ in the Module g\_Delay1 on line 94 and as the input iD in the Module g\_Delay3 on line 114.

**[Part 1.g.v] In your lab report, include a screenshot of the waveform. Describe, in plain English, any differences between what you expected and what the simulation showed.**

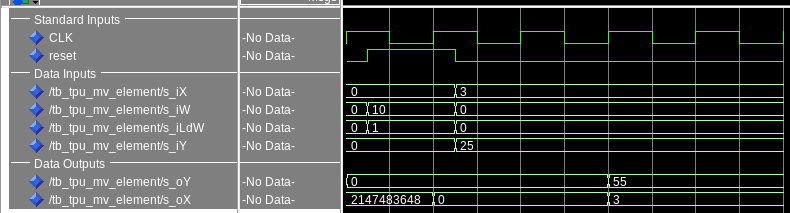


*Figure 1. tb\_TPU\_MV\_Element Waveform with bugs*

Case 1 works as expected, but in case 2, after the activation input iX is loaded as 3 and the partial sum input iY is loaded as 25, the output should be 55 = 3 \* 10 + 25. Since the weight input iW is 10, it gets multiplied with iX and added to iY, for the partial sum output oY.

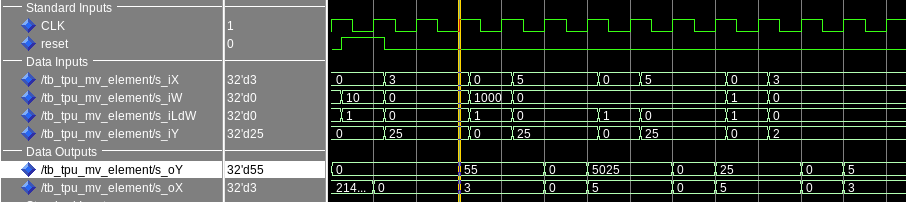
**[Part 1.h] In your lab report, include a screenshot of the waveform. Describe, in plain English, how your waveform matches the expected result (e.g., reference the specific cycles and times). In your submission zip file, provide the completed *TPU\_MV\_Element.vhd* file in a folder called ‘MAC’.**

The error above was due to iX being assigned as the data input iD in the Reg module g\_Delay2 instead of iY. Now, after two clock cycles, oY outputs as 55 as expected, and oX outputs the expected output 3.



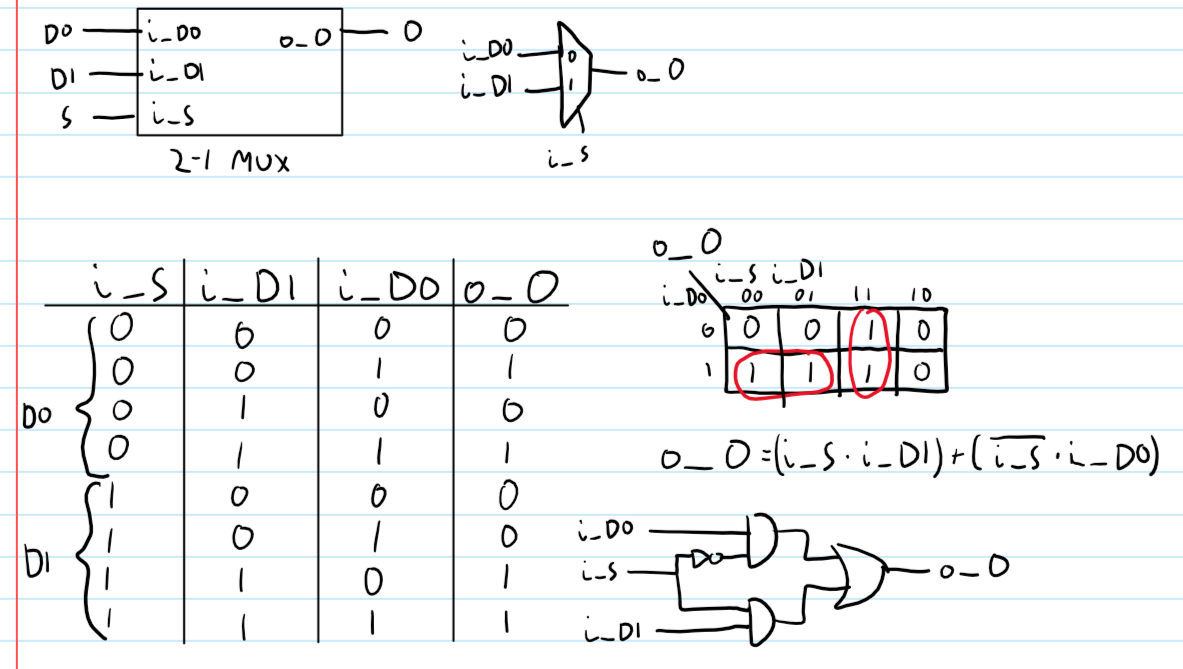
*Figure 2. tb\_TPU\_MV\_Element Waveform After Debug*

I also updated tb\_TPU\_MV\_Elements to include the three extra test cases listed in part 1c. After recompiling the testbench and running the .do file again, I verified that the expected results were met for oX and oY.



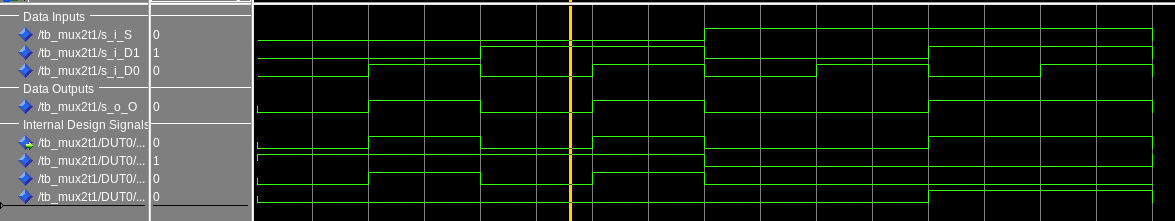
*Figure 3. tb\_TPU\_MV\_Element Waveform With Additional Testcases*

**[Part 3.a] Draw the truth table, Boolean equation, and Boolean circuit equivalent (using only two-input gates) that implements a 2:1 mux. Include this in your lab report.**

*Figure 4. 2-1 MUX Block Diagram, Truth Table, K-Map, and Boolean Circuit*

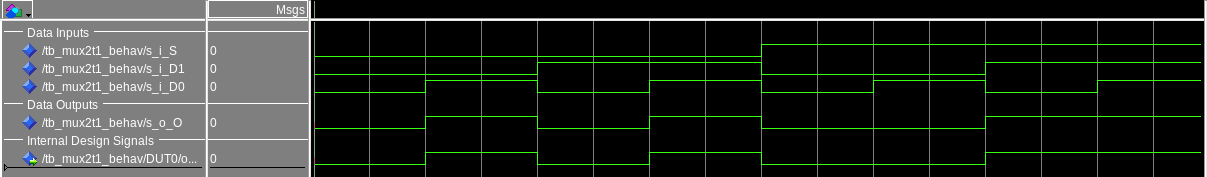
**[Part 3.d] In your lab report, include a screenshot of the waveform. Make sure to label the screenshot with which module it is testing.**

The testbench file I used to generate the waveform below was **tb\_mux2t1**.

*Figure 5. 2-1 mux2t1 Structual Waveform*

**[Part 3.e] Again, in your lab report, include a labeled screenshot of the waveform showing the dataflow mux implementation working.**

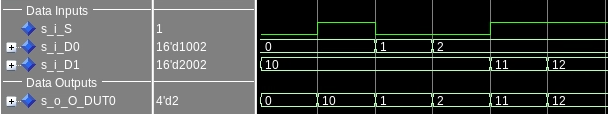
The testbench file I used to generate the waveform below was **tb\_mux2t1\_behav**.

*Figure 6. 2-1 mux2t1 Behavioral Waveform*

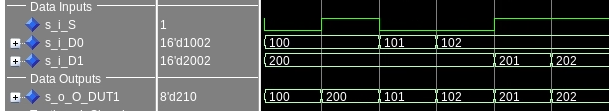
By using the IEEE library for the 2 input AND and OR gates, along with the 1 input NOT gate, I was able to condense my code a ton. This is especially true since I did not have to include the component definitions of each gate before the begin statement in the architecture block.

**[Part 4] Include a waveform screenshot and corresponding description demonstrating it is working correctly.**

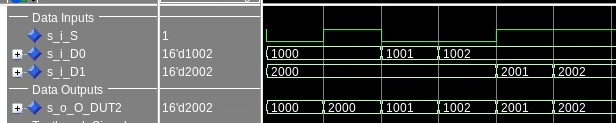
For this waveform design, I instantiated three different modules of mux2t1\_n, named DUT0, DUT1, and DUT2, which contained N muxes for the generic values of 4, 8, and 16 respectively. I wanted to instantiate different sized modules so I could verify I had the correct syntax for the generic mapping. So, when DUT\_select is 0, the outputs for DUT0 are checked, and so on for the other two DUT1/2 modules. I used one set of input pins and only used N bits needed for each DUT, to try and condense my waveforms. Because of this, the outputs for the other DUT’s should not be used besides the one indicated by DUT\_select for a given section. The testbench code used was in the file **tb\_mux2t1\_N**.



*Figure 7. 2-1 mux2t1\_N, DUT0 N=4 Waveform*



*Figure 8. 2-1 mux2t1\_N, DUT1 N=8 Waveform*

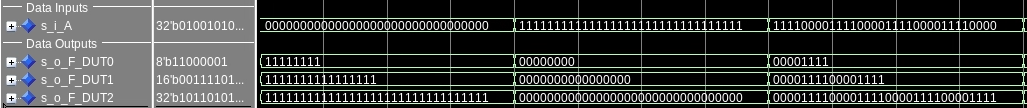


*Figure 9. 2-1 mux2t1\_N, DUT2 N=16 Waveform*

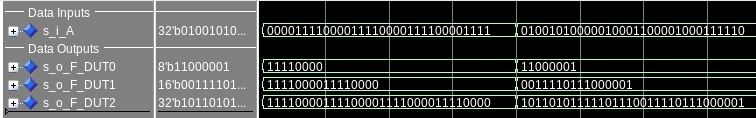
As you can see in each waveform, the output o\_O displays every bit of i\_D0 when the select line i\_S is 0, and outputs every bit of i\_D1 when the select line i\_S is 1.

**[Part 5.b] Include a waveform screenshot and description in your lab report.**

In this section, I created a testbench that was very similar to the mux2t1\_N testing. s\_i\_A was my testbench signal to act as an N bit input for DUT0, DUT1, and DUT2, which were instances of the ones\_comp module. The outputs s\_o\_F\_DUT0, s\_o\_F\_DUT1, and s\_o\_F\_DUT2 were the outputs of the ones\_comp modules, with respective integer inputs of 8, 16, and 32 bits. Since each DUT shared the same input s\_i\_A, I used the first 8 bits of that signal for DUT0, the first 16 bits for DUT1, and all 32 bits for DUT2. This helped me create a more concise testbench and compare results for the bits that were shared between each of the DUT modules.



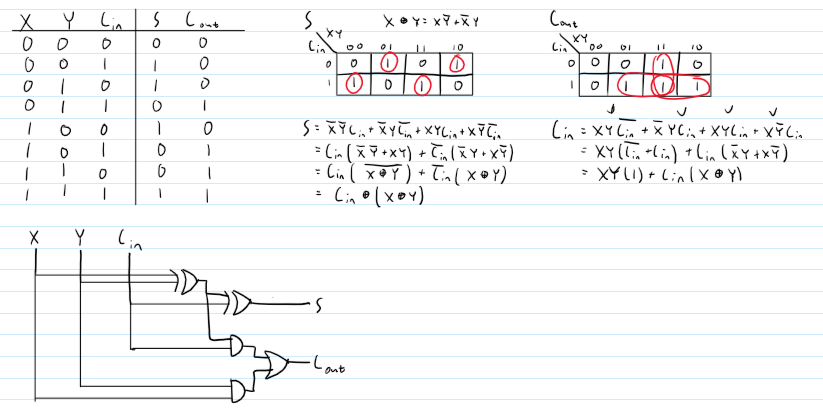
*Figure 10. ones\_comp Waveform 1*



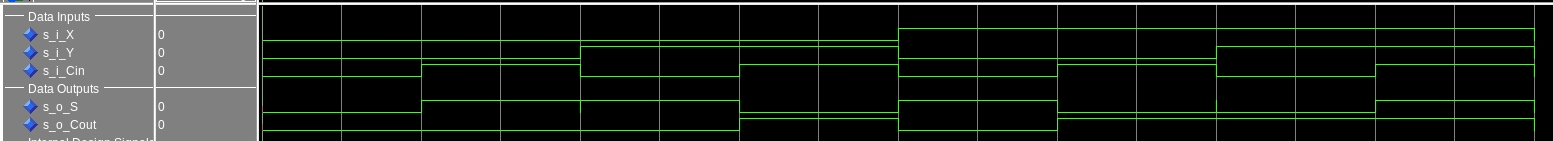
*Figure 11. ones\_comp Waveform 2*

**[Part 6.a] A full adder takes three single-bit inputs and produces two single-bit outputs – a sum and carry for the addition of the three input bits. Draw the truth table, Boolean equation, and Boolean circuit equivalent (using only two-input gates) that implements a 1-bit full adder. Include this in your report.**

Below, I derived the truth table, boolean equation, and boolean circuit equivalent for a 1 bit full adder circuit. I decided to take advantage of the fact that we were provided with a 2-input XOR gate to condense the structural design and circuit of the full adder



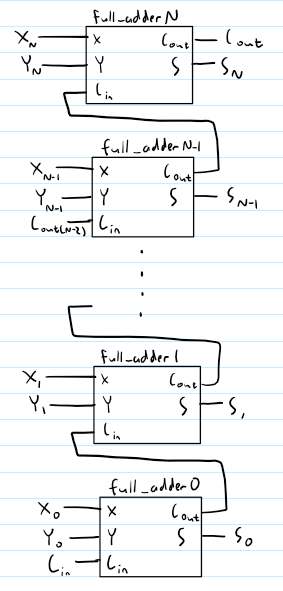
*Figure 12. Full Adder Truth Table, K-Maps, Boolean Circuit*



*Figure 13. Full Adder Waveform*

**[Part 6.c] Then draw a schematic of the intended design, including inputs and outputs and at least the 0, 1, N-2, and N-1 stages. Include this in your report.**

In this design, I routed the individual bit inputs for i\_X and i\_Y into the A and B inputs of each full\_adder module. For each N bit ripple carry adder design, I planned to use N full\_adder modules. The carryout bit of each full adder would get routed to the carry in bit of the next full adder.



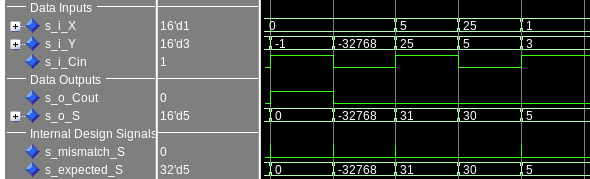
*Figure 14. N-bit Ripple Carry Adder Block Diagram*

**[Part 6.d] Include an annotated waveform screenshot in your write-up.**

Based on my waveforms below, I ran multiple different test cases to test my N bit ripple carry adder with a 16 bit component instantiated. I first decided to add together all three inputs, i\_X, i\_Y, and i\_Cin as zero to verify both outputs o\_S and o\_Cout would return 0. After that, I tested adding just one input and the other two set to zero for all three inputs. After that, I tried together 0 and -1 to set o\_Cout to 1. Then, I tried adding a very large negative signed value, along with some test cases similar to the TPU testbench in question 1c.



*Figure 15. N-bit Ripple Carry Adder Waveform 1*

**

*Figure 16. N-bit Ripple Carry Adder Waveform 2*

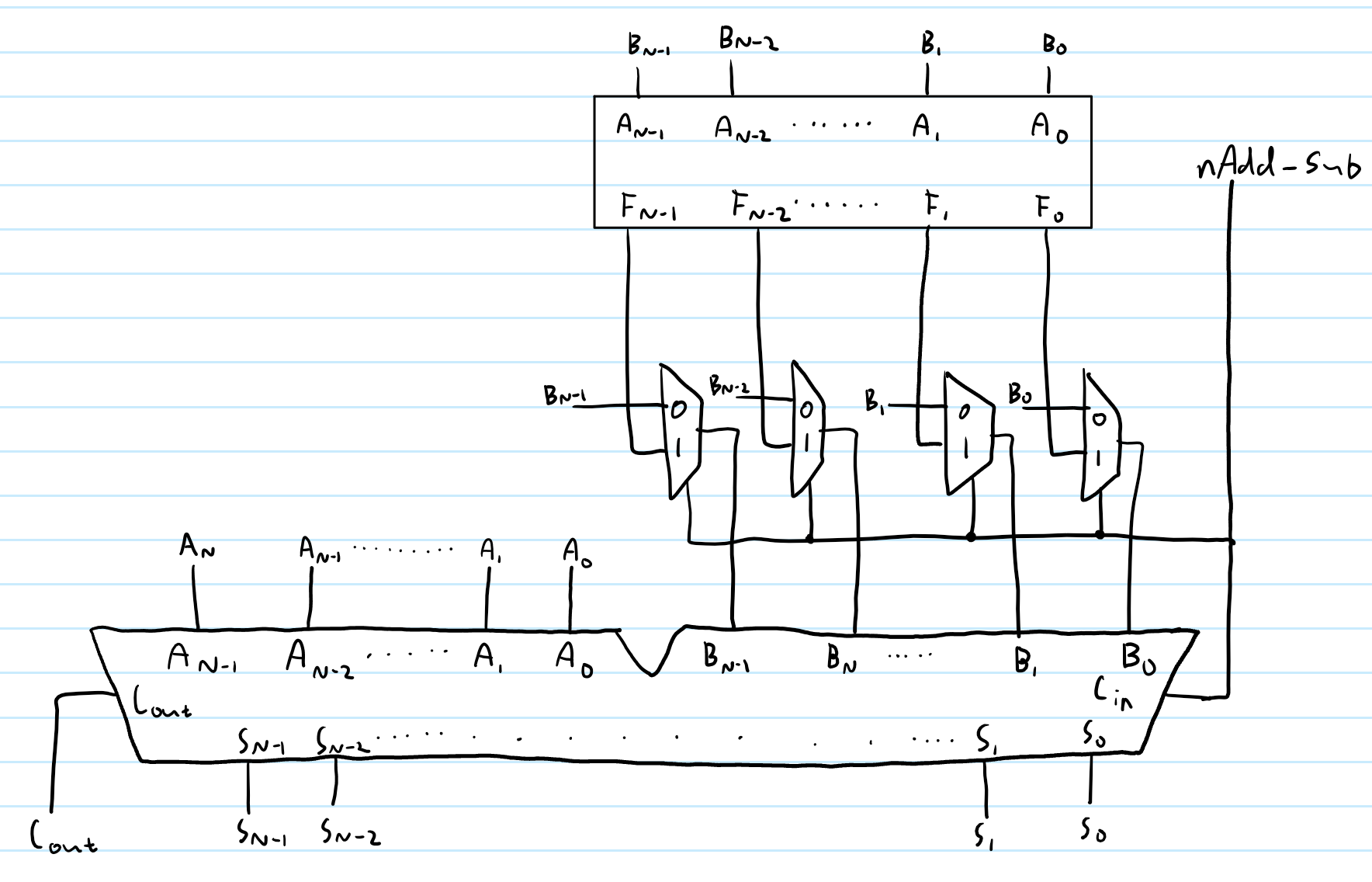
**Table 2** below shows the given test cases and expected results for the waveforms in **Figure 15** and **Figure 16** above.

| **i\_X** | **i\_Y** | **i\_Cin** | **o\_Cout** | **o\_S** | **expected\_S** | **expected\_Cout** |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 2 | 0 | 0 | 0 | 2 | 2 | 0 |
| 0 | 3 | 0 | 0 | 3 | 3 | 0 |
| 0 | -1 | 1 | 1 | 0 | 0 | 1 |
| 0 | -32768 | 0 | 0 | -32768 | -32768 | 0 |
| 5 | 31 | 1 | 0 | 31 | 31 | 0 |
| 25 | 30 | 0 | 0 | 30 | 30 | 0 |
| 1 | 5 | 1 | 0 | 5 | 5 | 0 |

*Table 2. N-bit Ripple Carry Adder Test Cases*

**[Part 7.a] Draw a schematic (don’t use a schematic capture tool) showing how an N-bit adder/subtractor with control can be implemented using only the three main components designed in earlier parts of this lab (i.e., the N-bit inverter, N-bit 2:1 mux, and N-bit adder). How is the ‘nAdd\_Sub’ bit used? Include this in your report.**

The design below incorporates the N-bit inverter, N-bit 2:1 mux, and the N-bit adder. Normally for an adder subtractor circuit, you use XOR gates with the B input and the nAdd\_Sub control. Since we have to use the three modules we designed, we instead inverted all bits B with the N-bit inverter and used a multiplexer with nAdd\_Sub as the select bit. If nAdd\_Sub is 0, we are adding so the input B can be selected. If nAdd\_Sub is 1, then B needs to be converted to twos complement, so the inverted B will be selected and the nAdd\_Sub will be added as 1 for a carry in for the N-bit ripple carry adder module. Since we are controlling adding or subtracting B, we can just wire the input A to the ripple carry adder module. The output sum S is given for N bits after the ripple carry adder, along with a carryout.

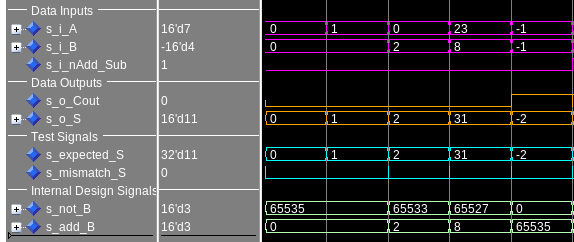


*Figure 17. N-bit Adder Subtractor Block Diagram*

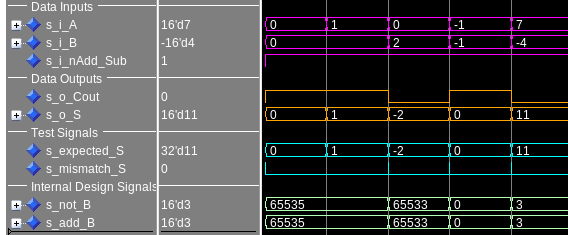
**[Part 7.c] Provide multiple waveform screenshots in your write-up to confirm that this component is working correctly. What test-cases did you include and why?**

Below are the waveforms for the N-bit Adder Subtractor module. Waveform 1 describes when i\_nAdd\_Sub is 0, so the module is performing A + B. In Figure 19, Waveform 2 describes when i\_nAdd\_Sub is 1, where the module is performing the operation A - B.

My test cases were very similar for both the adding and subtracting cases. For adding, I first tried adding 0 for both inputs, to confirm my output would be 0. I then tried adding an individual value for both A and B to confirm they would be routed to the output. After that, I tried adding together two positive and two negative signed numbers. These operations were very similar for the second Waveform, except that the operation A - B was being performed instead of A + B.

****

*Figure 18. N-bit Adder Subtractor Waveform 1, s\_nAdd\_Sub = 0*



*Figure 19. N-bit Adder Subtractor Waveform 2, s\_nAdd\_Sub = 1*

**Table 3** below shows the given test cases and expected results for the waveforms in **Figure 18** and **Figure 19** above.

| **i\_X** | **i\_Y** | **i\_nAdd\_Sub** | **o\_Cout** | **o\_S** | **expected\_S** | **expected\_Cout** |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 2 | 0 | 0 | 2 | 2 | 0 |
| 23 | 8 | 0 | 0 | 31 | 31 | 0 |
| -1 | -1 | 0 | 1 | -2 | -2 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 2 | 1 | 0 | -2 | -2 | 0 |
| -1 | -1 | 1 | 1 | 0 | 0 | 1 |
| 7 | -4 | 1 | 0 | 11 | 11 | 0 |

*Table 3. N-bit Adder Subtractor Test Cases*