CprE 381, Computer Organization and Assembly Level Programming

Lab 1 Report

Student Name: **CONNOR LINK**

Submit a typeset pdf version of this on Canvas by the due date. Refer to the highlighted language in the lab document for the context of the following questions.

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

Test Case 1: Reset weight to 0

- Inputs: Set the *iW* signal to 0 and *iLdW* to 1 prior to the positive clock edge and hold until after the clock edge
- Outputs: I expect the *s_W* signal to fall to zero at the clock's rising edge and the partial sum output to change to be equal to the value of *iY* after two positive clock edges (once the partial sum has updated, *iX * s_W* will be zero).

Test Case 2: Perform single MAC operation

- If case 1 passes successfully, proceed to execute case 2
- Inputs: Set the iW signal to 1 and iLdW signal to 1 prior to the positive edge of the clock and hold until after it. Then set the iLdW signal to 0. Set the iX signal to 5 and the iY signal to 5.
- Outputs: I expect the s_W signal to become and stay 1 following the first rising edge. After two positive clock edges, I expect the oX signal to be 5 and the oY signal to be 10 (1 * 5 + 5).

Test Case 3: Perform birthday MAC operation

- If case 2 passes successfully, proceed to execute case 3
- Inputs: Set the iW signal to 11 and the iLdW signal to 1 prior to the positive edge of the clock and hold until after it. Then set the iLdW signal to 0. Set the iX signal to 100 and the iY signal to 4. Then wait for two positive clock edges to elapse.
- Outputs: I expect oX to be 100 and oY to be 1104 (11 * 100 + 4)—my birthday 11/04—after waiting two positive clock edges. The s_W intermediate signal should be 11 after the first positive edge.

[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

[Please format your responses similar to the following example: in the attached diagram area, (17) is the oC output of the multiplier module, which is defined as an integer on Line 31 of Multiplier.vhd and set on Line 41. For this specific instance of the multiplier module, the output oC is connected to a signal called s_WxX on line 105 of TPU_MV_Element.vhd.]

1.) In the diagram, label "1" references the overall TPU_MV_Element block, the interface to which is defined on line 24 of TPU_MV_Element.vhd, and the architecture of which is outlined—in structural form—following line 38 of TPU_MV_Element.vhd. After the later, the internal components, including gates and other references entities are re-declared, and then modules instantiated to make up the internal structure of this block. This particular module is instanced as *DUT0* on line 65 of

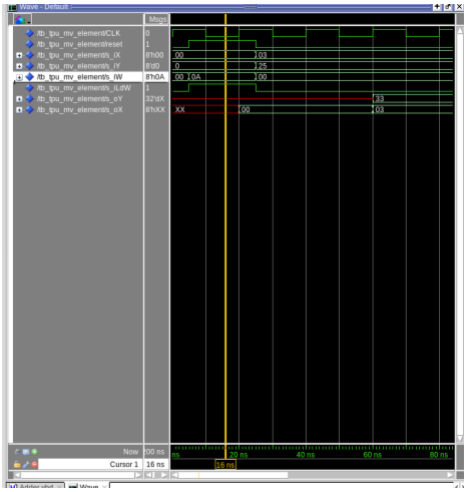
tb_TPU_MV_Element.vhd, and all inputs and outputs—iCLK, iX, iW, iLdW, iY, oY, oX—are connected to testbench stimuli defined near line 52 of the same file.

7.) In the attached diagram, region "7" points to an instance of an adder module *g_Add1*, which was instantiated on line 131 of TPU_MV_Element.vhd; the clock *iCLK* is passed directly, while the *s_WxX* signal (defined line 83 of same file) is connected to one input and *s_Y1* (defined line 81 of same file). The output signal is wired directly to the module output *oY* defined on line 33 of the same file. The adder entity originates on line 26 of Adder.vhd, and its architecture is modeled behaviorally under line 39 in this file.

22.) In the provided graphic, "22" references the signal *s_W*, which is first defined on line 77 of TPU_MV_Element.vhd and driven by the delay circuit to load a new weight on line 96 therein. *s_W* then goes on to power an input to the *g_Mult1* multiplier instance on line 118.

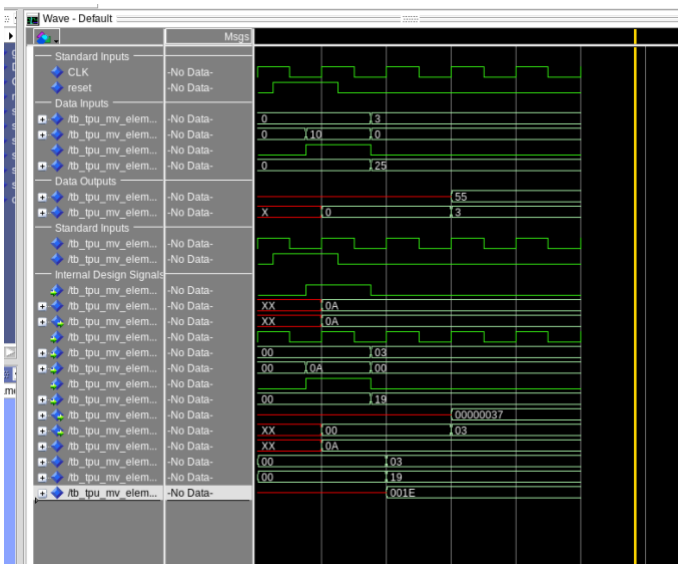
28.) I assume "28" to refer to the "*iD*" input to the *g_Delay3* delay circuit instance in the provided figure. This instance is made on line 125 of file TPU_MV_Element.vhd, and the *iD* signal is tied to the signal *s_X1* on line 128. *s_X1* originates from another delay circuit that can be traced back to being driven by the overall module input *iX*.

[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.



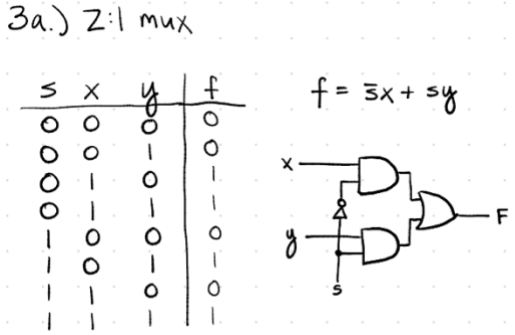
Unmodified TPU: The DUT is functioning according to the timings expected; *however*, there is a miscalculation at the end, for we are getting 33 and not 55; thus, the simulation does not match the expected behavior.

[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'.

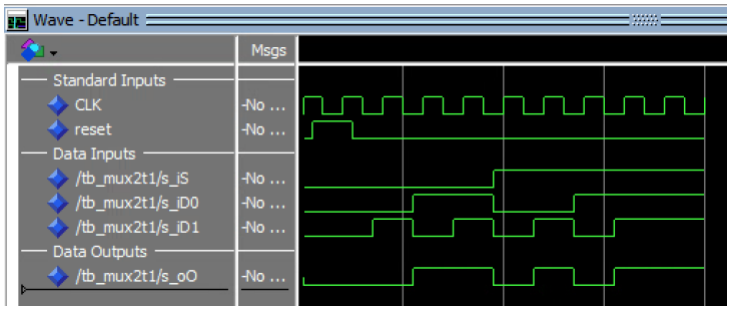


Final TPU: Even after fixing the compile and load errors, the problem was that the end result was being miscalculated. I traced the problem back to being that the s_Y1 signal was the same as s_X1, which was because iX was mistakenly connected to the Y delay circuit. I corrected this issue, and the circuit is now producing a value of 55 as expected.

[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.



[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.

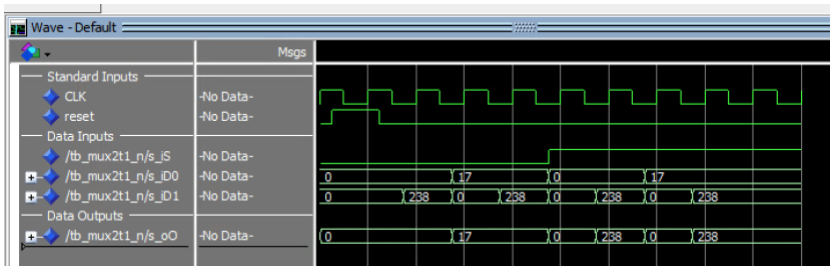


Module Under Test: mux2t1 (Mux/mux2t1.vhd)
Testbench File: tb_mux2t1 (Mux/tb_mux2t1.vhd)

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

Module Under Test: mux2t1d (Mux/mux2t1d.vhd)
Testbench File: tb_mux2t1d (Mux/tb_mux2t1d.vhd)

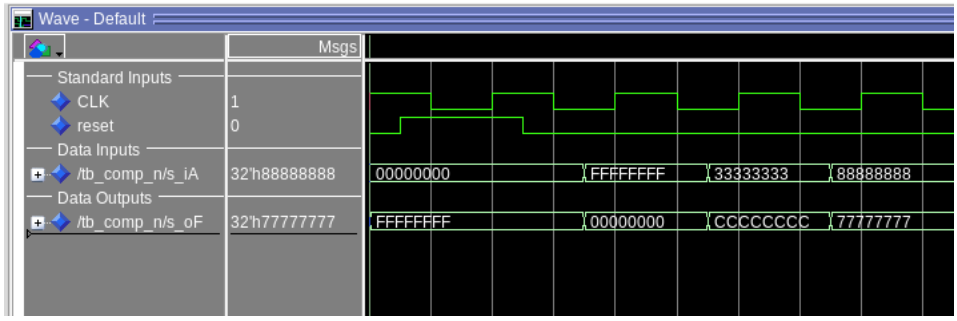
[Part 4] Include a waveform screenshot and corresponding description demonstrating it is working correctly.



Module Under Test: mux2t1_N (mux2t1_N.vhd)
Testbench File: tb_mux2t1_N (tb_mux2t1_N.vhd)

Each of the 8 truth table cases are exhaustively covered here. First, *iS*, *iD0*, and *iD1* are set to 0 and the result is zero as expected. Since the selector is 0, it follows that the output is equal to *iD0*—regardless of the value of *iD1*, *oO* remains equal to *iD0*. Then, when the selector *iS* is set to 1, the output then becomes the value of *iD1* irrespective of *iD0*: 0, 283, 0, then 238 again.

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

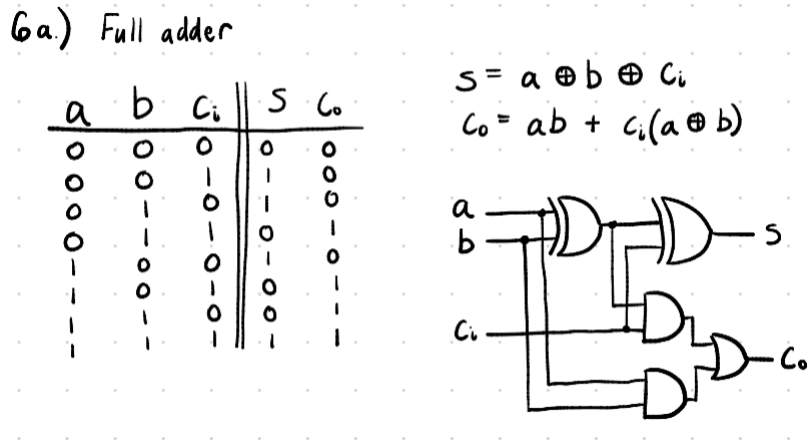


Module Under Test: comp_N (OnesComp/comp_N.vhd)
Testbench File: tb_comp_N (OnesComp/tb_comp_N.vhd)

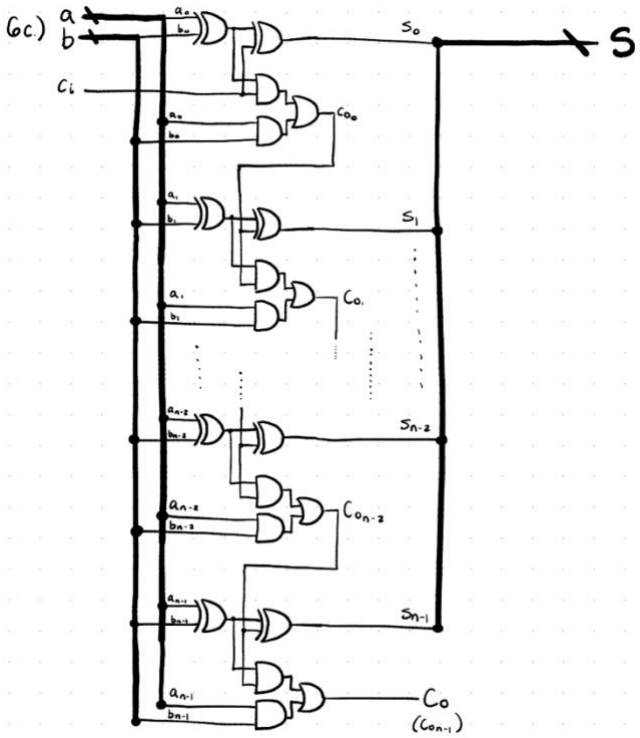
The testbench created for comp_N involved 4 primary test cases. First, setting *iA* to $00000000 gives $FFFFFFFF as expected—inverting all zeroes will give all ones. The same follows but vice-versa for test

case 2. For test case 3, the *iA* input of $33333333 is a bit pattern of 0011… meaning the result should be 1100… or $CCCCCCCC, which is what is seen on the waveform output. Finally, $88888888 (1000…) is inverted to correctly get $77777777 (0111…).

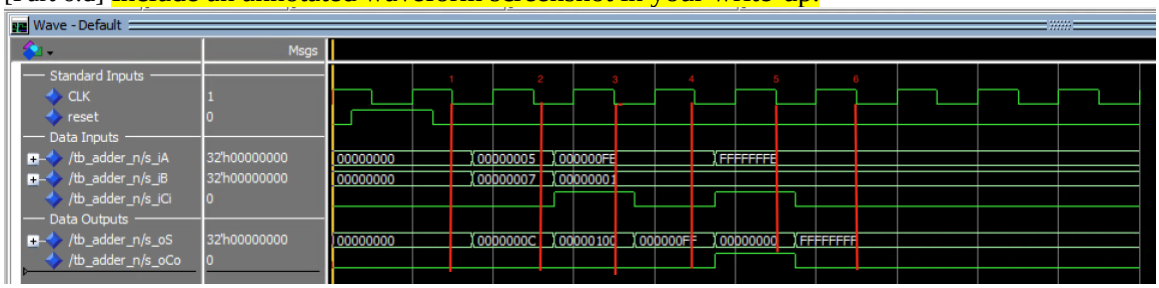
[Part 6.a] A full adder takes three single-bit inputs and produces two single-bit outputs – a sum and carry for the addition of the three input bits. Draw the truth table, Boolean equation, and Boolean circuit equivalent (using only two-input gates) that implements a 1-bit full adder. Include this in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



[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.



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



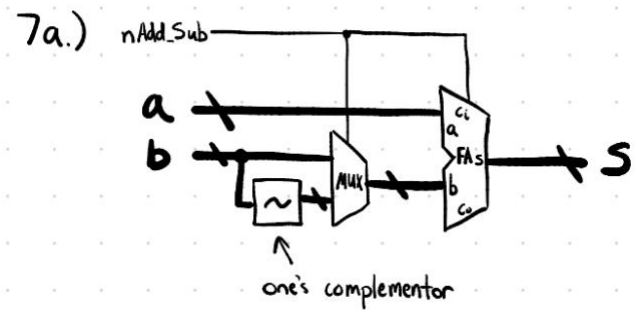
Module Under Test: adder_N (Adder/adder_N.vhd)
Testbench File: tb_adder_N (Adder/tb_adder_N.vhd)

Annotation Table:

Point #	s_iA	s_iB	s_iCi	Expected s_oS	Actual s_oS	Expected s_oCo	Actual s_oCo
1	0x0	0x0	0	0x0	0x0	0	0
2	0x5	0x7	0	0xC	0xC	0	0
3	0xFE	0x1	1	0x100	0x100	0	0
4	0xFE	0x1	0	0xFF	0xFF	0	0
5	0xFFFFFFFE	0x1	1	0x0	0x0	1	1
6	0xFFFFFFFE	0x1	0	0xFFFFFFFF	0xFFFFFFFF	0	0

Annotation Description:
Starting off with the trivial 0 case, the output is 0 as expected. Then, setting up the equation 5+7+0 on the inputs *iA*, *iB*, and *iCi* respectively nets $0…0C on the output—12—as anticipated. Then, adding 254+1+1 should be 256 (which doesn't fit in 2 hexadecimal digits) so the output *oS* is $0…0100. No outgoing carry *oCo* is generated for either case as they fit well within the constraints of the N=32-bit configuration. Next, the *iCi* signal is set to 0, meaning the sum *oS* should decrease by 1—indeed it drops from $100 to $FF as expected. For the last test case, I test the overflow and carry out signals by setting *iA* to $FFFFFFFE (1 less than UINT_MAX), keep *iB* at $0…01 and set the *iCi* to 1. By adding a total of two to the A value, I expect the result to overflow. Indeed, the generated *oS* is $00 and the *oCo* signal goes high to indicate that the value has overflowed.

[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.

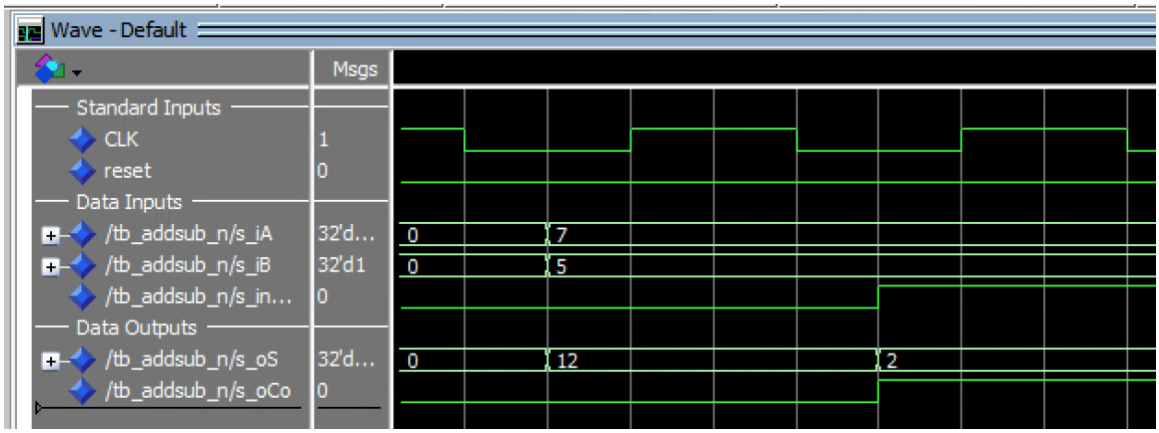


Since we are dealing with two's complement signed integers, subtraction can be accomplished by taking A+(-B). To negate B, we can take its logical NOT—the one's complement—and add 1 to this total.

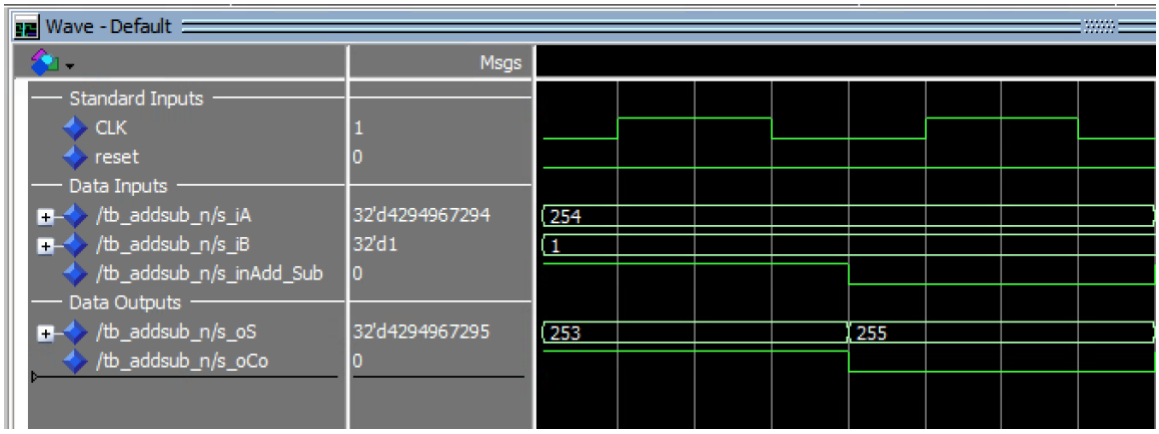
Using the *nAdd_Sub* input, we use it as a carry into the adder: when it is 0 (and we desire to add), nothing is being added to the total, whereas when it is 1 (and we desire to subtract), we are adding 1 to the running total. Coupled with using *nAdd_Sub* as the selector to a multiplexer wired to the unmodified and inverted B signals, the two's complement negation can be computed as desired. When *nAdd_Sub* is one, ~B is sent to adder to then compute A+(~B)+1, which is logically equivalent to A+(-B)=A-B.

[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?

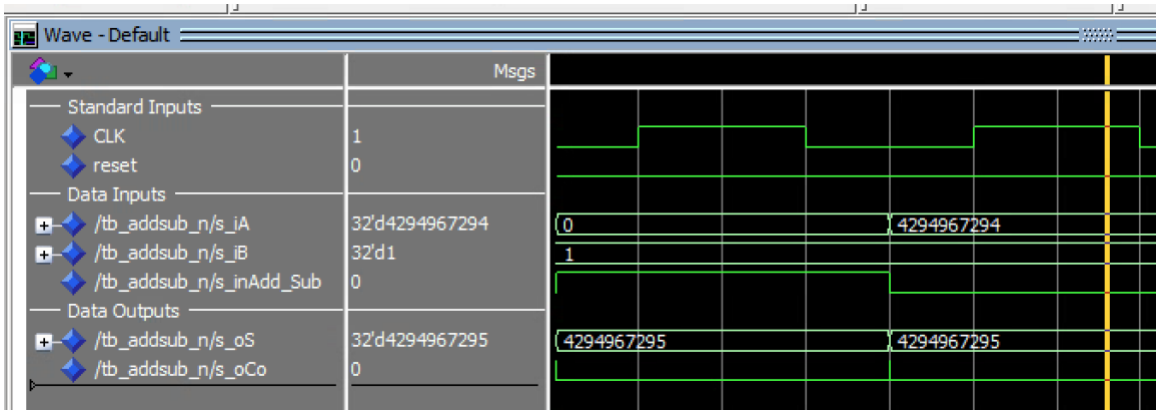
Module Under Test: addsub_N (AddSub/addsub_N.vhd)
Testbench File: tb_addsub_N (AddSub/tb_addsub_N.vhd)



I included this test case as a simple entry-level verification to demonstrate functionality of both the addition and subtraction cases. To start, the waveform shows that 0+0=0, then shows that 7+5=12, exactly as expected. Then, I drive the *nAdd_Sub* input high to switch to subtraction, where the waveform correctly shows that 7-5=2. In this case, the carry out—or rather, borrow flag—goes high for subtraction (or no borrow), which logically makes sense in this case as I have designed an arithmetic unit that will produce a complemented signal for borrow as carry.



I included this test case as another simple test to prove that the addition and subtraction functionality is working as expected. Starting off, *iA* is 254 and *iB* is 1, with *nAdd_Sub* set to 1 (subtract). The result is 254-1=253, exactly as predicted. And again, the *oCo* signal (borrow) being 1 indicates that no borrow was required. Then, *nAdd_Sub* is set to 0 (add), where the waveform shows that 254+1=255, again as anticipated.



I included the last as a bit more of a stressor for an underflow condition and to better test the borrow functionality. Starting out I set *iA* to 0 and *iB* to 1 and enable *nAdd_Sub* to enable subtraction. This primes the equation 0-1=-1 which is correctly indicated by the result *oS* of 4,294,967,295 ($FFFFFFFF = -1) and will have required a borrow to complete which is also corrected indicated by the complemented borrow *oCo* signal of 0. Then I change *nAdd_Sub* to again ensure that the unit will change over to addition correctly, where it shows that $FFFFFFFE+$1=$FFFFFFFF.