CprE 381, Computer Organization and Assembly Level Programming

Lab 1 Report

Student Name _____Owen Jewell_____

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.

Case 3: Overriding an existing weight value

- The purpose of this case is to ensure that new a new Weight Data Input Value can be successfully loaded. This requires a new value for *iW* and for the Weight Load Control Signal, *iLdW*, to be set to 1.
- To begin I initialize a weight value register by setting *iW* to 4 and *iLdW* to 1 before a positive edge of the clock and holding them for that positive edge in order to load the new value.
- Once the weight initialization of 4 is complete, the activation input, *iX*, should be set to 2 while the partial sum input, *iY*, is set to 3.
- I expect that after two clock cycles, the activation output, oX, will be 2 and the partial sum output, oY, will be 11. This is because (weight * activation input) + partial sum input = partial sum output => (4 * 2) + 3 = 11.
- I will then set iW to 7 and iLdW to 1 for a whole positive clock edge. After the cycle the intermediate internal signal, *s_W*, should have taken on the new value of 7.

Case 4: New Value for Weight Data Input Value but Weight Load Control Signal is 0

- The purpose of this case is to ensure that new Weight Data Input Value is not accidentally loaded while the Load Control signal is set to 0.
- To begin I initialize a weight value register by setting *iW* to 20 and *iLdW* to 1 before a positive edge of the clock and holding them for that positive edge in order to load the new value.
- Once the weight initialization of 20 is complete, the activation input, *iX*, should be set to 4 while the partial sum input, *iY*, is set to 10. To test if the original weight value remains unchanged the value of *iLdW* should be set to 0 while the *iW* is set to 50.
- I expect that after two clock cycles, the activation output, oX, will be 4 and the partial sum output, oY, will be 90. This is because (weight * activation input) + partial sum input = partial sum output => (20 * 4) + 10 = 90.
- If the new weight value is incorrectly loaded then the partial sum output will not be equal to 90 because the weight vlaue of 50 would have been used instead.

Case 5: Initial Values Resulting in my Birthday (28)

- To begin I initialize a weight value register by setting *iW* to 5 and *iLdW* to 1 before a positive edge of the clock and holding them for that positive edge in order to load the new value.
- Once the weight initialization of 5 is complete, the activation input ,*iX*, should be set to 5 while the partial sum input, *iY*, is set to 3. To ensure that the original weight value remains unchanged the value of *iLdW* should be set to 0.
- I expect that after two clock cycles, the activation output, oX, will be 5 and the partial sum output, oY, will be 28. This is because (weight * activation input) + partial sum input = partial sum output => (5 * 5) + 3 = 28.

[Part 1.e] For labels 9, 20, 32, and 33, 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

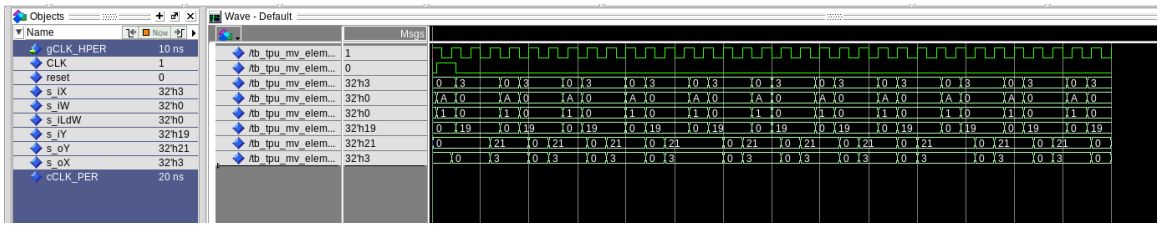
In the attached diagram area, (9) is the oQ output of the weight module, which is defined as an integer on Line 31 of RegLd.vhd and set on Line 51. For this specific instance of the multiplier module, the output oC is connected to a signal called s_W on line 86 of TPU_MV_Element.vhd.]

In the attached diagram area, (20) is the iD input of the delay3 module, which is defined as an integer on Line 29 of Reg.vhd and used to set the output on Line 40. For this specific instance of the multiplier module, the output iD is connected to a signal called s_X1 on line 114 of TPU_MV_Element.vhd.]

In the attached diagram area, (32) is the g_Add1 (adder) module, its entity declartion is on lines 26-33 and its behavior is on lines 35-45 of Adder.vhd. This adder has 3 inputs iA, iB, and iCLK as well as one output oC. For this adder module it is called g_Add1 with the port map on lines 117-121 of TPU_MV_Element.vhd.]

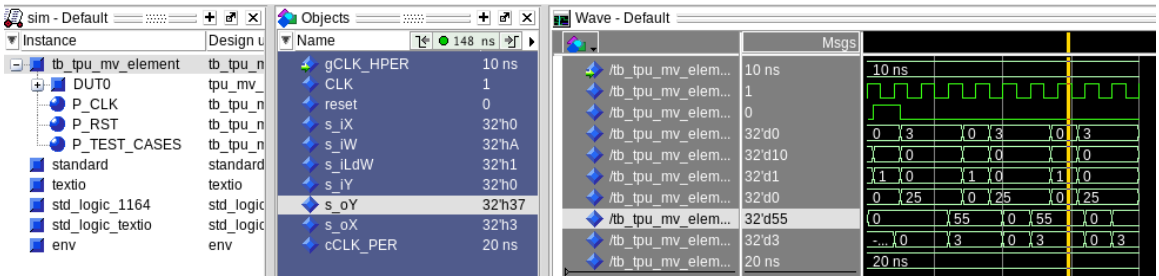
In the attached diagram area, (33) is the TPU_MV_Element module, which is defined in TPU_MV_Element.vhd The entity declaration is lines 23-33. The structure is lines 35-124. This is the top level module that encompasses all of the small modules within.

[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.



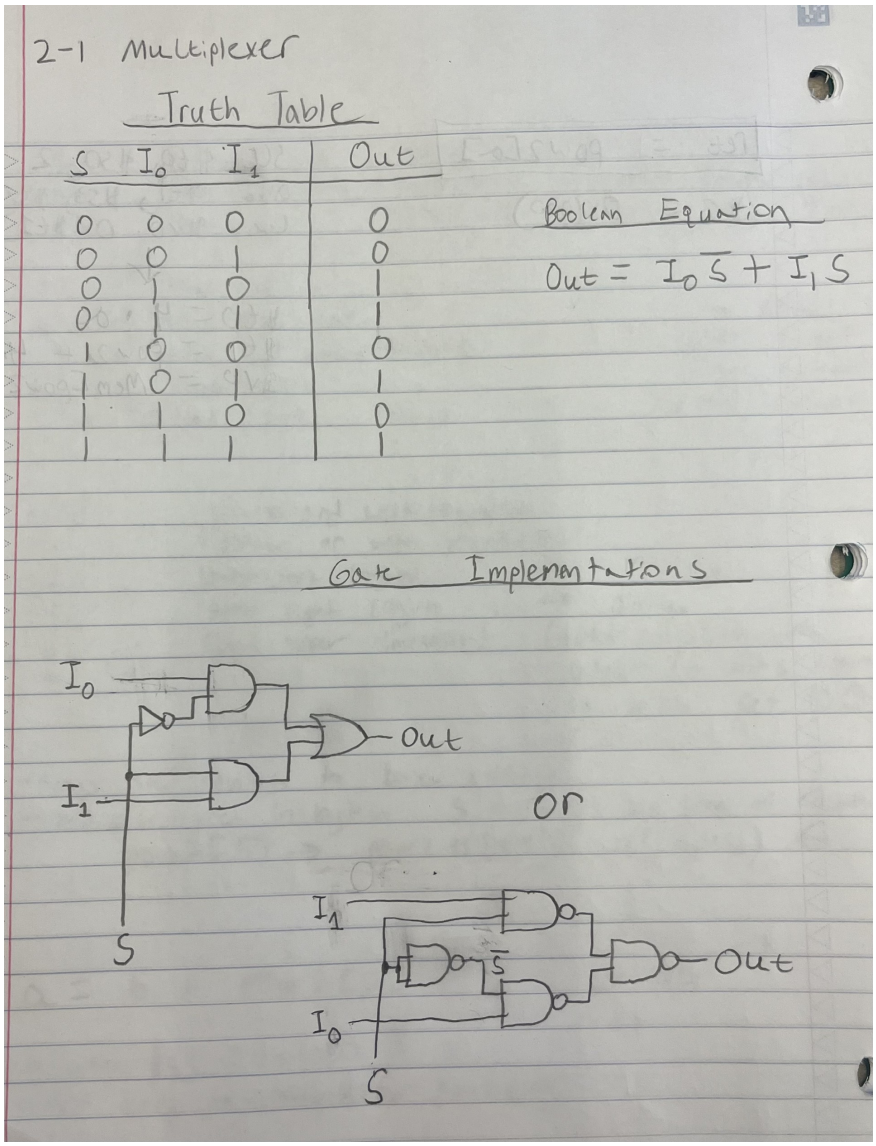
In this screenshot of the waveform there are a few things that do not match the expected result. First of all it is possible to tell something is wrong because the output oY is not 55 after the first two clock cycles. There is also an issue with the value of iY after the first two clock cycles because the value is 19 instead of the desired 25.

[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'.

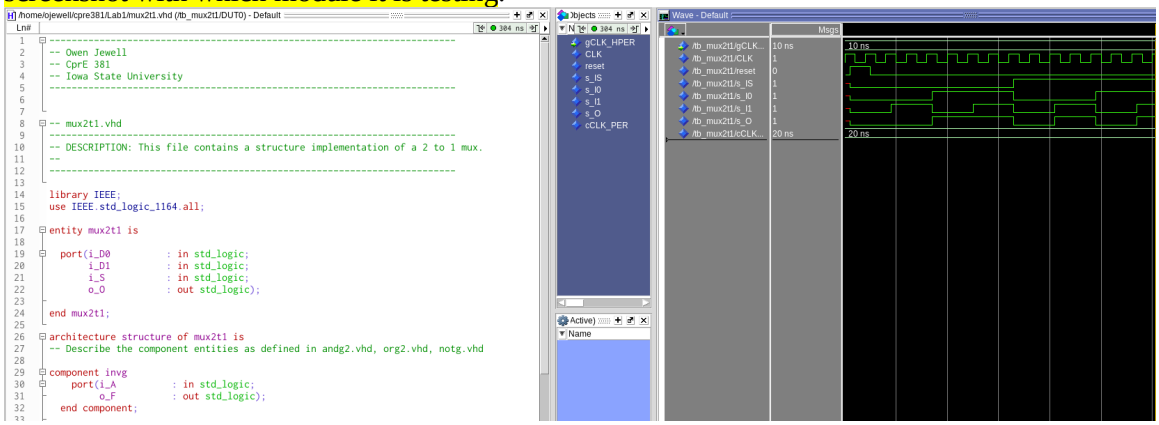


In this screenshot of the waveform all of the values do match the expected results. First we can see that after the two lock cycles the value of iX is 3. The value of the load weight is 1 to start and then the actual weight is seen to be 0xA or 10 in decimal. Then the oY value can be seen to be 55 which is the expected result after 10 * 3 + 25 = 55.

[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.

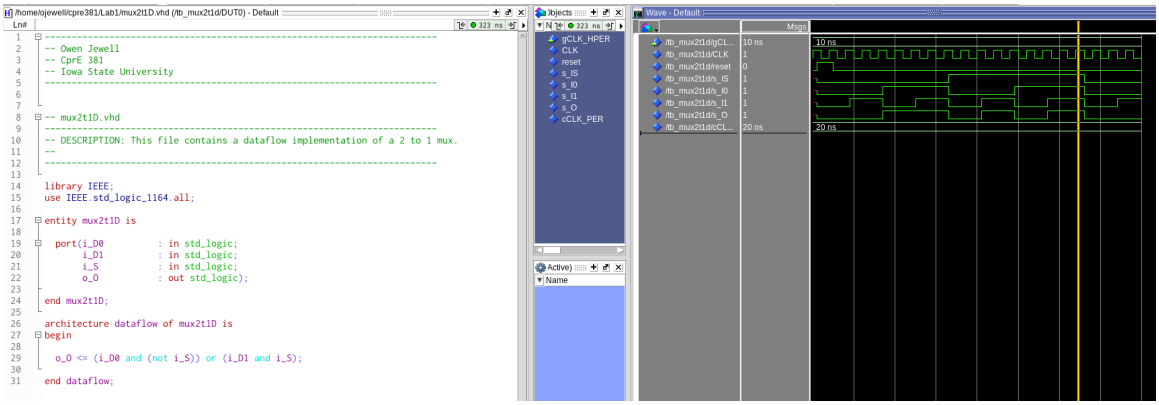


[Part 3.d] In your lab report, include a screenshot of the waveform. Make sure to label the screenshot with which module it is testing.



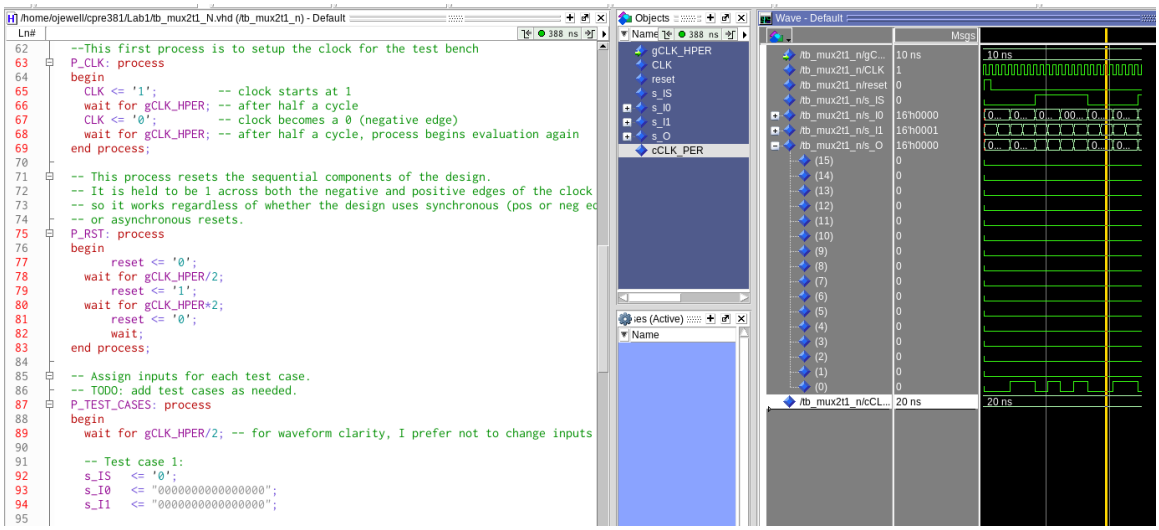
This waveform is for the structural two-to-one multiplexer. The output wave s_O follows the truth table that can be seen in the picture above.

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



This waveform is for the dataflow two-to-one multiplexer. The output wave s_O follows the wave from the structure implementation above.

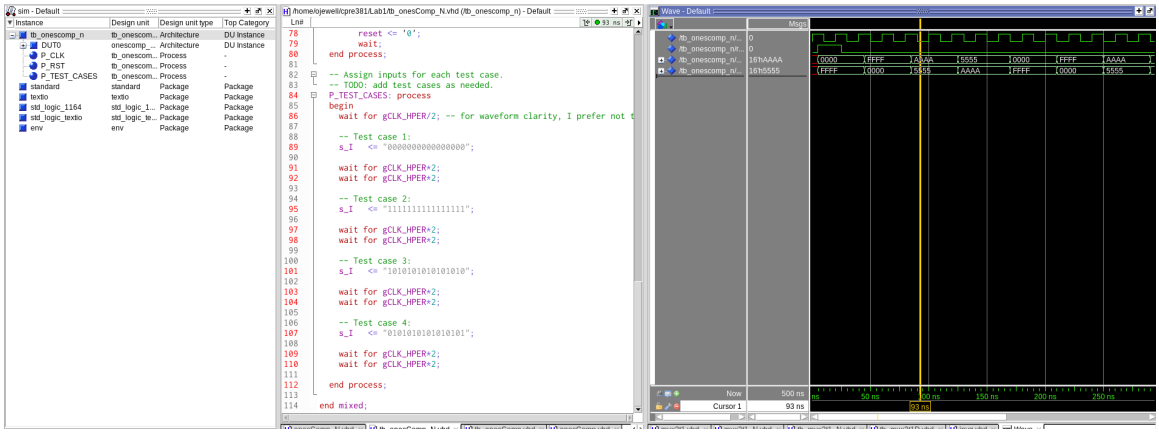
[Part 4] Include a waveform screenshot and corresponding description demonstrating it is working correctly.



```
--This first process is to setup the clock for the test bench
P_CLK: process
begin
  CLK <= '1';          -- clock starts at 1
  wait for gCLK_HPER; -- after half a cycle
  CLK <= '0';          -- clock becomes a 0 (negative edge)
  wait for gCLK_HPER; -- after half a cycle, process begins evaluation again
end process;

-- This process resets the sequential components of the design.
-- It is held to be 1 across both the negative and positive edges of the clock
-- so it works regardless of whether the design uses synchronous (pos or neg ec
-- or asynchronous resets.
P_RST: process
begin
      reset <= '0';
  wait for gCLK_HPER/2;
      reset <= '1';
  wait for gCLK_HPER*2;
      reset <= '0';
      wait;
end process;

-- Assign inputs for each test case.
-- TODO: add test cases as needed.
P_TEST_CASES: process
begin
  wait for gCLK_HPER/2; -- for waveform clarity, I prefer not to change inputs

  -- Test case 1:
  s_IS   <= '0';
  s_I0   <= "0000000000000000";
  s_I1   <= "0000000000000000";
```

Looking at the bottom waveform for the least significant bit of the output we can see that it matches the output wave for the previous mux. The test cases were the same as previous except manipulating the last bit of the 16 bit vector.

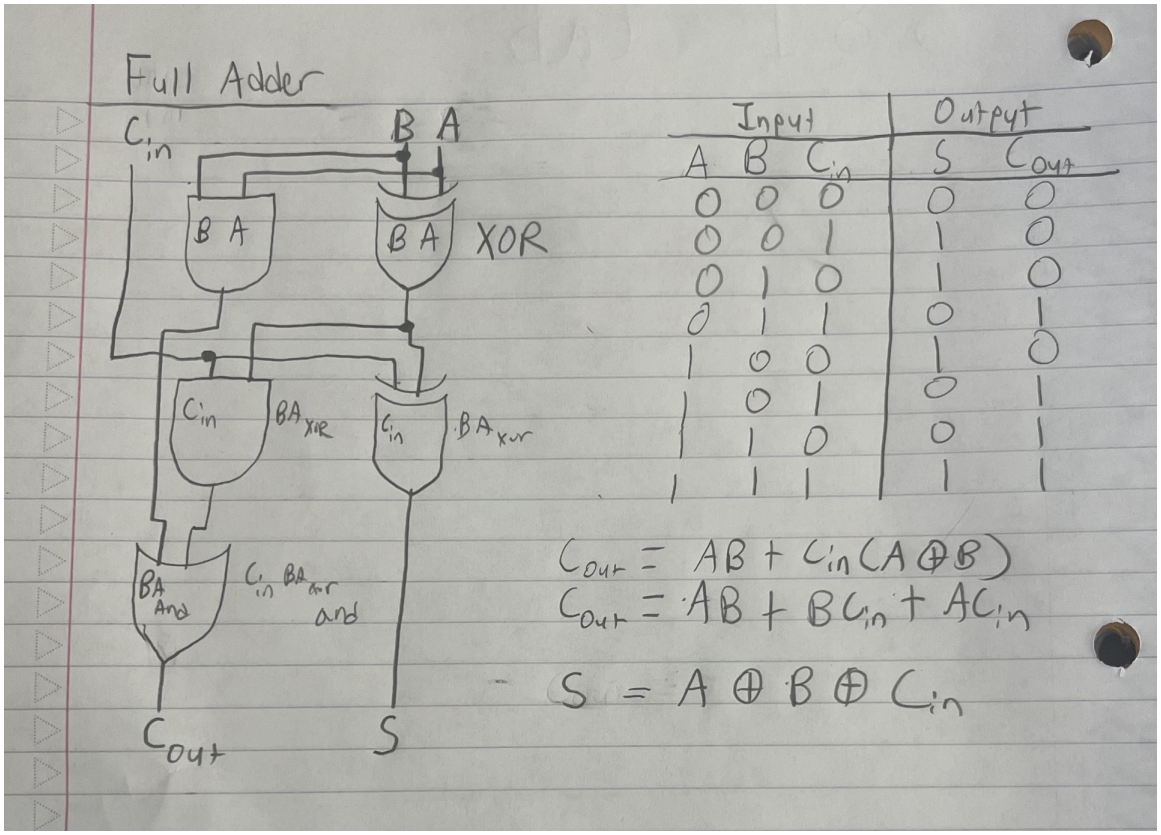
[Part 5.b] Include a waveform screenshot and description in your lab report.



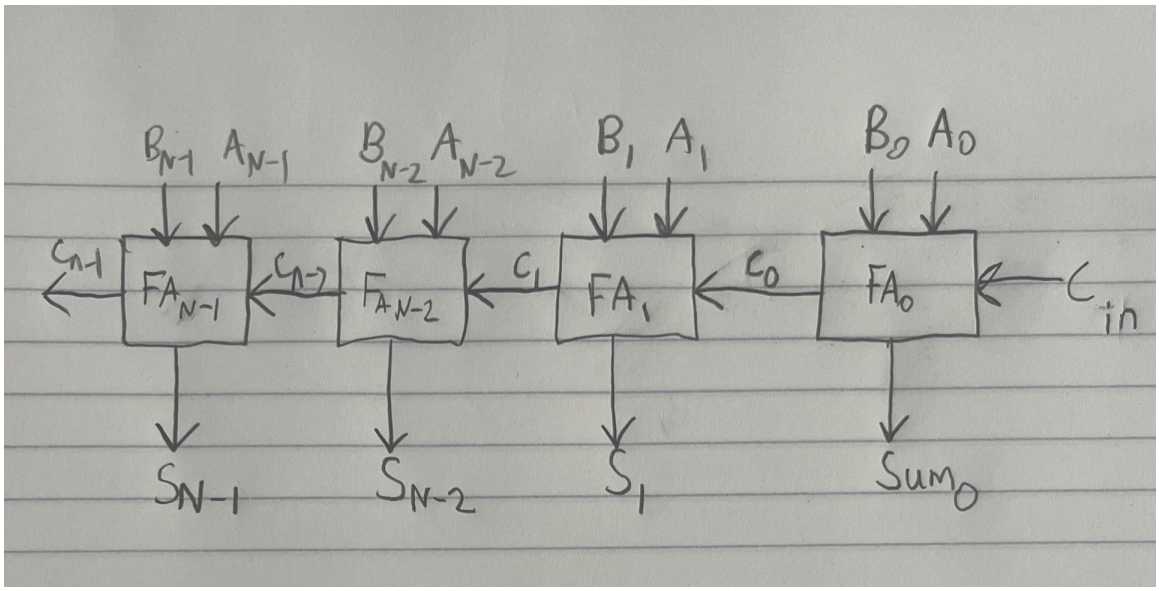
```
reset <= '0';
wait;
end process;

-- Assign inputs for each test case.
-- TODO: add test cases as needed.
P_TEST_CASES: process
begin
  wait for gCLK_HPER/2; -- for waveform clarity, I prefer not t

  -- Test case 1:
  s_I   <= "0000000000000000";

  wait for gCLK_HPER*2;
  wait for gCLK_HPER*2;

  -- Test case 2:
  s_I   <= "1111111111111111";

  wait for gCLK_HPER*2;
  wait for gCLK_HPER*2;

  -- Test case 3:
  s_I   <= "1010101010101010";

  wait for gCLK_HPER*2;
  wait for gCLK_HPER*2;

  -- Test case 4:
  s_I   <= "0101010101010101";

  wait for gCLK_HPER*2;
  wait for gCLK_HPER*2;

end process;

end mixed;
```

As can be seen in the waveform the input wave is always inverted in the output wave. All 0's goes to all 1's. AAAA goes to 5555 and vice versa because 1010 inverted is 0101.

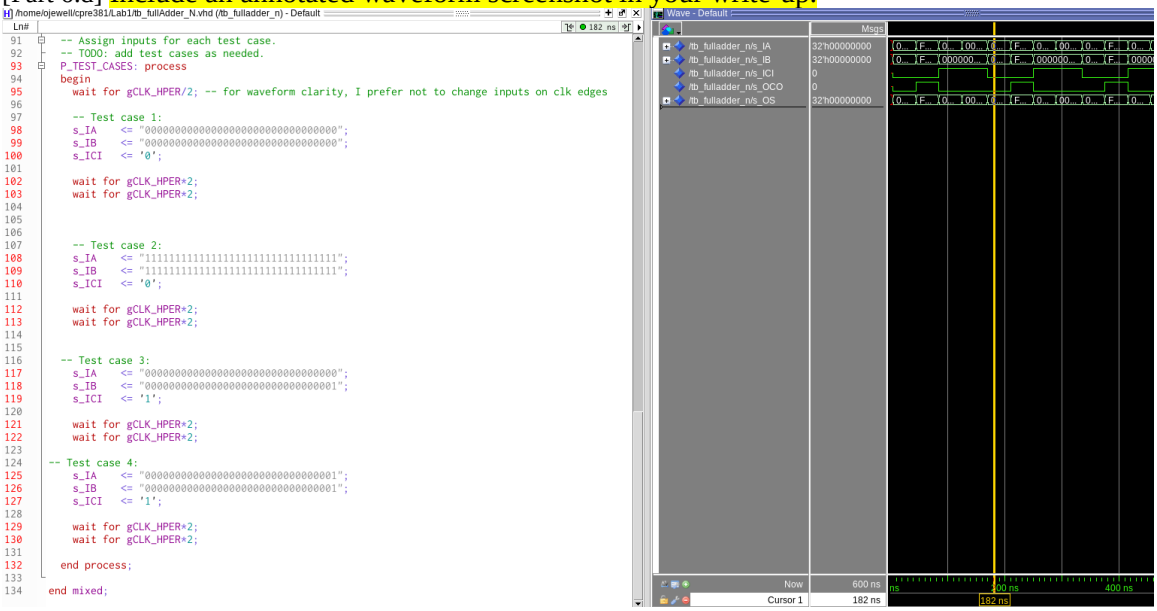
[Part 6.a] A full adder takes three single-bit inputs and produces two single-bit outputs – a sum and carry for the addition of the three input bits. Draw the truth table, Boolean equation, and Boolean circuit equivalent (using only two-input gates) that implements a 1-bit full adder. Include this in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

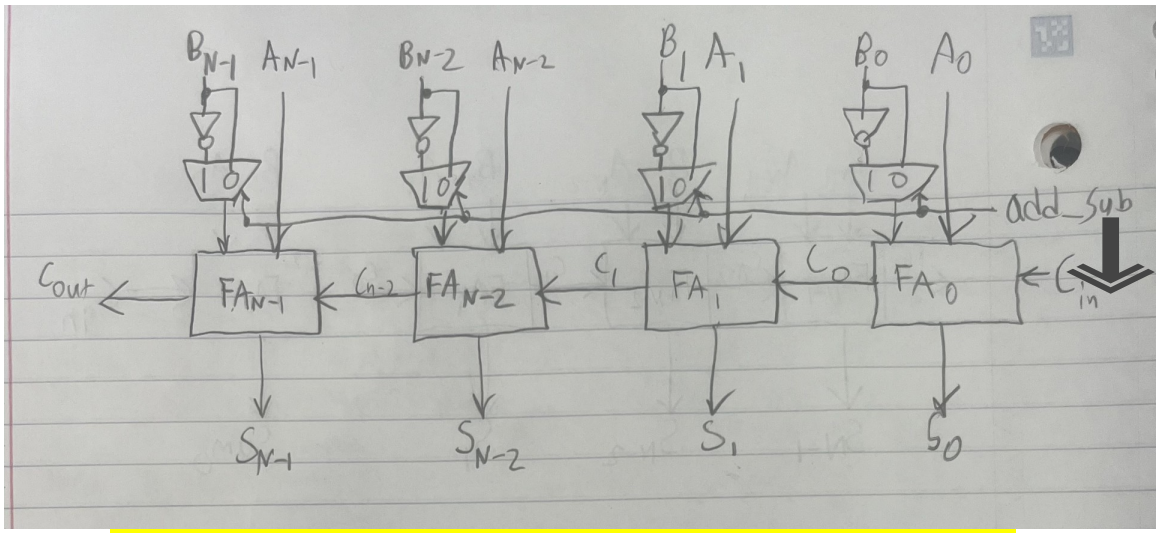


[Part 6.d] Include an annotated waveform screenshot in your write-up.



```
91    -- Assign inputs for each test case.
92    -- TODO: add test cases as needed.
93    P_TEST_CASES: process
94    begin
95      wait for gCLK_HPER/2; -- for waveform clarity, I prefer not to change inputs on clk edges
96
97      -- Test case 1:
98      s_IA    <= "00000000000000000000000000000000";
99      s_IB    <= "00000000000000000000000000000000";
100     s_ICI   <= '0';
101
102     wait for gCLK_HPER*2;
103     wait for gCLK_HPER*2;
104
105
106
107     -- Test case 2:
108     s_IA    <= "11111111111111111111111111111111";
109     s_IB    <= "11111111111111111111111111111111";
110     s_ICI   <= '0';
111
112     wait for gCLK_HPER*2;
113     wait for gCLK_HPER*2;
114
115
116   -- Test case 3:
117     s_IA    <= "00000000000000000000000000000000";
118     s_IB    <= "00000000000000000000000000000001";
119     s_ICI   <= '1';
120
121     wait for gCLK_HPER*2;
122     wait for gCLK_HPER*2;
123
124   -- Test case 4:
125     s_IA    <= "00000000000000000000000000000001";
126     s_IB    <= "00000000000000000000000000000001";
127     s_ICI   <= '1';
128
129     wait for gCLK_HPER*2;
130     wait for gCLK_HPER*2;
131
132   end process;
133
134 end mixed;
```

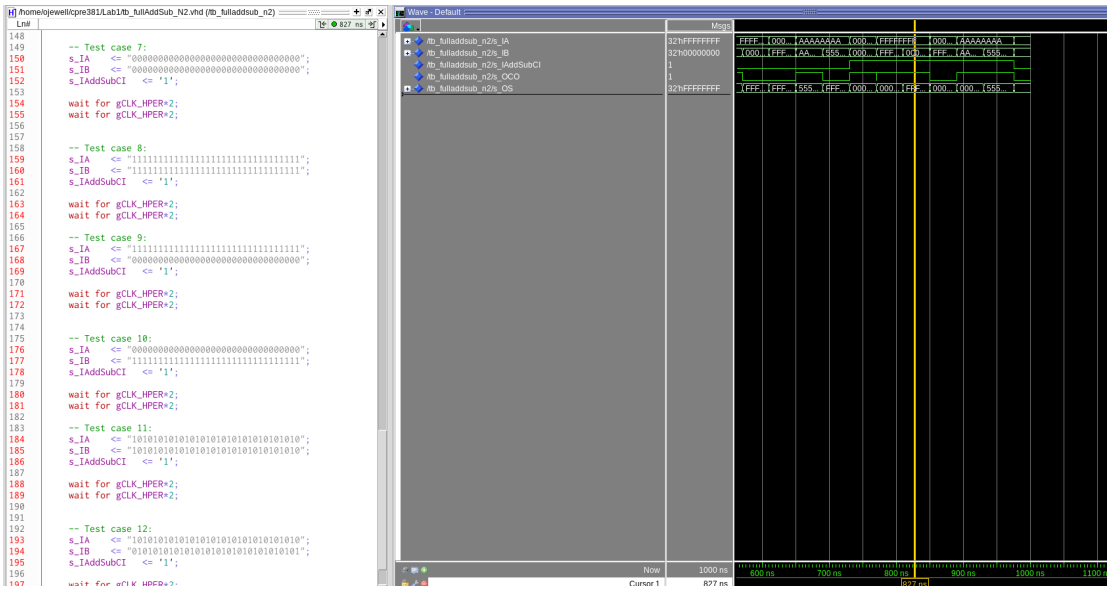
In this screenshot the 4 test cases I wrote can be seen to be successfully completed. The first is to add 0 with 0 with a 0 carry-in. This correctly outputs a sum and carry-out of 0. The second test is to add 32 bits of 1's with itself and a 0 carry-in. This correctly outputs a hex sum of FFFFFFFE with a carry-out of 1. The last letter is E because 1+1 would be 0 in the least significant bit and the 1 is carried forward. The third test case is 0 + 1 with a 1 carry-in. This correctly results in 2. The fourth test is 1+1 with a 1 carry-in. This correctly results in 3.

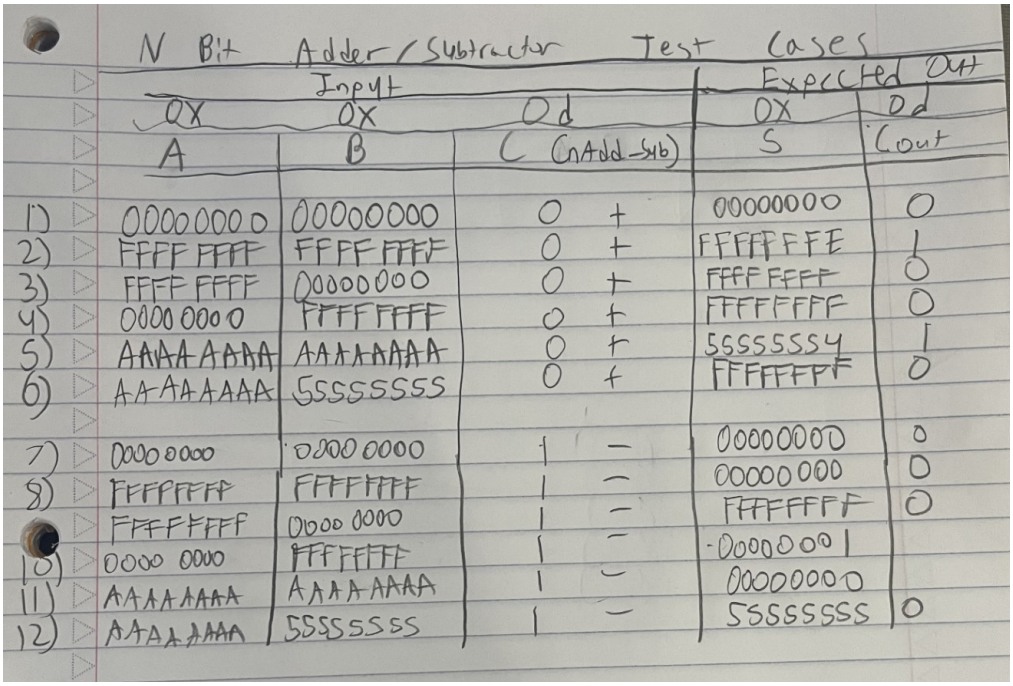


[Part 7.a] Draw a schematic (don't use a schematic capture tool) showing how an N-bit adder/subtractor with control can be implemented using only the three main components designed in earlier parts of this lab (i.e., the N-bit inverter, N-bit 2:1 mux, and N-bit adder). How is the 'nAdd_Sub' bit used?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

The difference with this diagram is the additional of the nAdd_Sub bit which is used as a select line for the muxes. If the bit is a 1 then all of input B should be inverted in order to do subtraction. Also the add_sub bit should connect to the carry in.

[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?





I decided to do 12 total test cases. I did the same 6 test cases for both addition and subtraction. As can be seen in the waveform all of my sums match the expected sums in the test case chart. The first screenshot is the addition test cases and the second screenshot is the negative test cases.