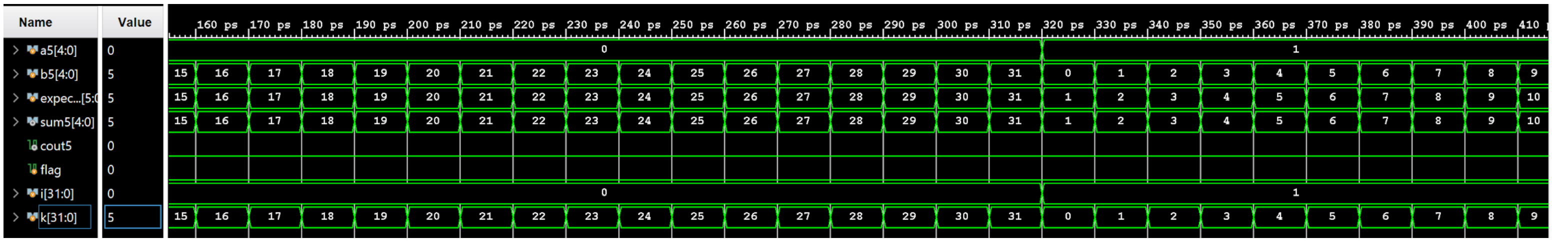
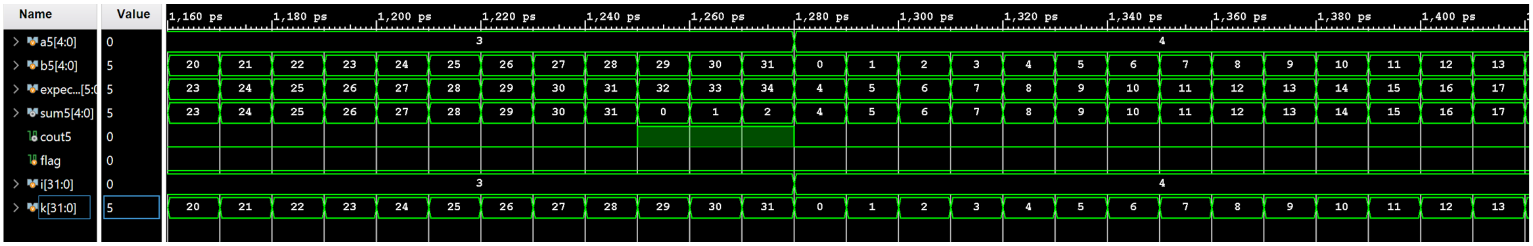
Lab 2 Simulation Outputs

Convince us that you know how to write appropriately thorough testbenches, you wrote such testbenches, and your implementation of each module functions as it should

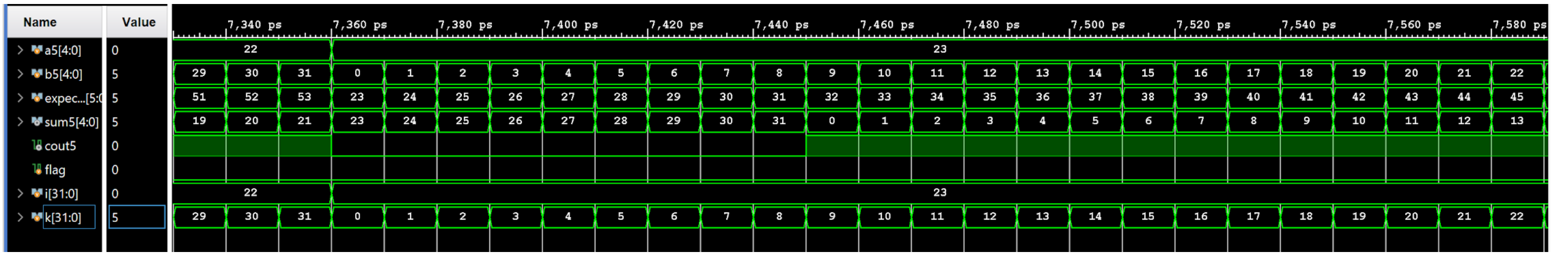
Adder


Name
Value
160 ps 170 ps 180 ps 190 ps 200 ps 210 ps 220 ps 230 ps 240 ps 250 ps 260 ps 270 ps 280 ps 290 ps 300 ps 310 ps 320 ps 330 ps 340 ps 350 ps 360 ps 370 ps 380 ps 390 ps 400 ps 410
a5[4:0] 0 0 1
b5[4:0] 5 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9
expec...[5:0] 5 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 1 2 3 4 5 6 7 8 9 10
sum5[4:0] 5 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 1 2 3 4 5 6 7 8 9 10
cout5 0
flag 0
i[31:0] 0 0 1
k[31:0] 5 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9

First image here, you can see values of a changing with i and values of b changing with k. i and k go up to a value max of 31 since a and b are 5-bit numbers with a max value of 31. So far, cout remains at 0 since the sum has not been greater than 31.


Name
Value
1,160 ps 1,180 ps 1,200 ps 1,220 ps 1,240 ps 1,260 ps 1,280 ps 1,300 ps 1,320 ps 1,340 ps 1,360 ps 1,380 ps 1,400 ps
a5[4:0] 0 3 4
b5[4:0] 5 20 21 22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13
expec...[5:0] 5 23 24 25 26 27 28 29 30 31 32 33 34 4 5 6 7 8 9 10 11 12 13 14 15 16 17
sum5[4:0] 5 23 24 25 26 27 28 29 30 31 0 1 2 4 5 6 7 8 9 10 11 12 13 14 15 16 17
cout5 0
flag 0
i[31:0] 0 3 4
k[31:0] 5 20 21 22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13

In this second image, there are 3 cases where cout is set to 1, and that is due to the sum of a and b being greater than 31. The variable sum may seem incorrect, but if you consider that the decimal value of cout when it is 1 is actually 32 and add the value of the sum var, then it returns the correct result.


Name
Value
7,340 ps 7,360 ps 7,380 ps 7,400 ps 7,420 ps 7,440 ps 7,460 ps 7,480 ps 7,500 ps 7,520 ps 7,540 ps 7,560 ps 7,580 ps
a5[4:0] 0 22 23
b5[4:0] 5 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
expec...[5:0] 5 51 52 53 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
sum5[4:0] 5 19 20 21 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13
cout5 0
flag 0
i[31:0] 0 22 23
k[31:0] 5 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

In this third image, you can see that there are more cases where cout is set to 1. Notice how the variable flag remains constantly at 0. Flag is only ever set to 1 if the expected value does not match {cout, sum}.

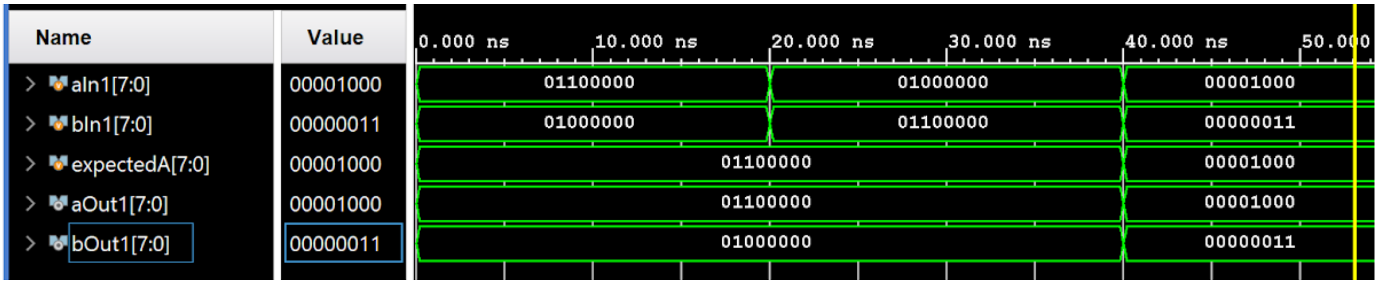
Testbench tcl output:

```
000 + 000 = 000 -- Expected 000
000 + 001 = 001 -- Expected 001
000 + 002 = 002 -- Expected 002
000 + 003 = 003 -- Expected 0
...
030 + 000 = 030 -- Expected 030
030 + 001 = 031 -- Expected 031
...
Passed 5-bit Addition Test :)
```

Testbench explanation:

We are convinced that our testbench is through and proves we have a properly working adder module because it runs through every possible test case and returns the correct result. This is done by using two nested for loops, one that corresponds to the value of input a and one that corresponds to the value of input b.

Big_Number_First



Name	Value
aIn1[7:0]	00001000
bIn1[7:0]	00000011
expectedA[7:0]	00001000
aOut1[7:0]	00001000
bOut1[7:0]	00000011

In this image, you can see three test cases being ran. In the first case, a and b have the same mantissa, but different exponents. We expect aIn to be returned as aOut due to it having the larger exponent, and it is.

In the second test case, the values of aIn and bIn are switched so bIn now has the larger exponent. We can see bIn returned as bOut as expected.

The final test case sets aIn and bIn to have the same exponent but different mantissa. bIn actually have a larger mantissa, and overall, is a larger value than aIn. It should still return aIn as aOut because the values of the mantissa are irrelevant in this module and we are only concerned with setting bIn as aOut *only if it has a larger exponent*. Again, our expected matches our output.

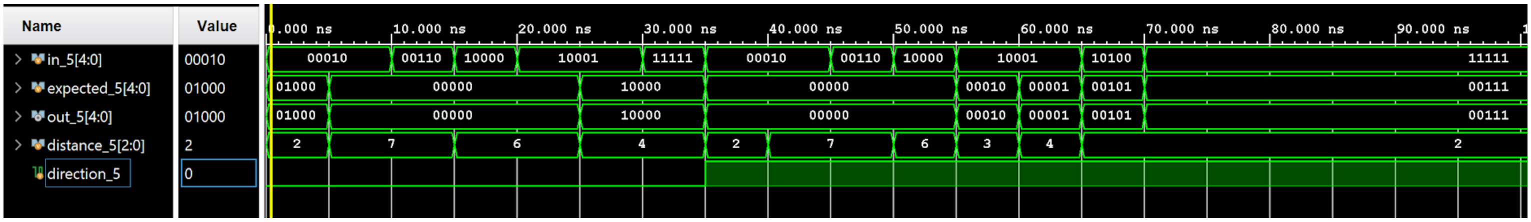
Testbench tcl output:

```
aIn: 01100000, bIn: 01000000, expected aOut: 01100000, actual aOut: 01100000
passed
aIn: 01000000, bIn: 01100000, expected aOut: 01100000, actual aOut: 01100000
passed
aIn: 00001000, bIn: 00000011, expected aOut: 00001000, actual aOut: 00001000
passed
```

Testbench Explanation:

Our testbench is through and provides confidence in our module because we tested for the several different cases we may encounter - aIn's exponent being larger than bIn's, bIn's exponent being larger than aIn's and both aIn and bIn having the same exponent, but bIn having a larger mantissa.

Shifter



The value of direction indicates whether to shift bits left (set to 0) or right (set to 1). Distance indicates how many places it should be shifted over. Once a value is asked to be shifted past the either edge of the binary number, 0s are put in and it is not seen to "wrap-around." This is the desired result as an appropriate shift of the mantissa is to be partnered with a different valued exponent in order to return an 8-bit float number that represents the same original value. About half of the cases are tested in the left directionand the other half in the right to cover all bases.

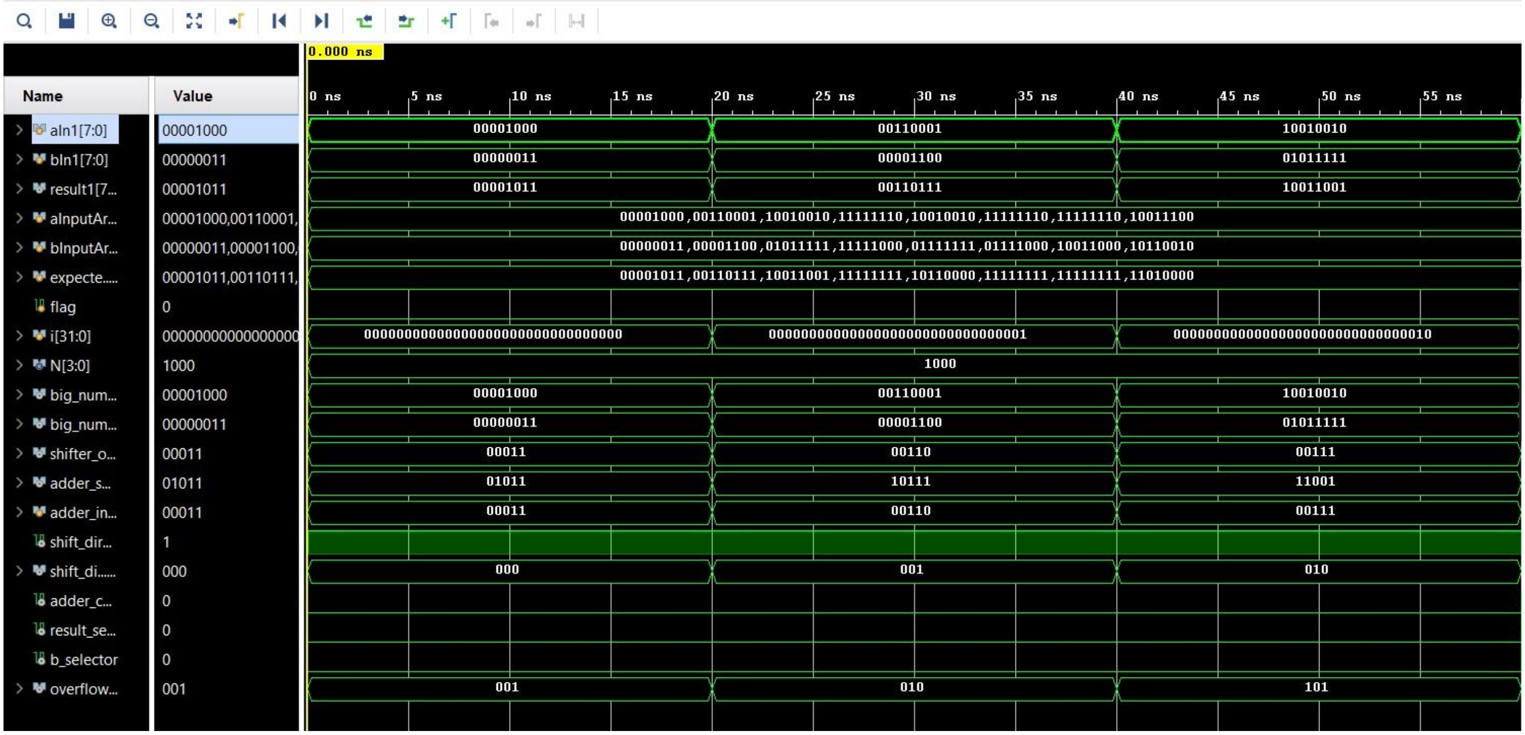
Testbench tcl output:

```
00010 -> 01000, expected 01000
00010 -> 00000, expected 00000
00110 -> 00000, expected 00000
10000 -> 00000, expected 00000
10001 -> 00000, expected 00000
10001 -> 10000, expected 10000
11111 -> 10000, expected 10000
00010 -> 00000, expected 00000
00010 -> 00000, expected 00000
00110 -> 00000, expected 00000
10000 -> 00000, expected 00000
10001 -> 00010, expected 00010
10001 -> 00001, expected 00001
10100 -> 00101, expected 00101
11111 -> 00111, expected 00111
```

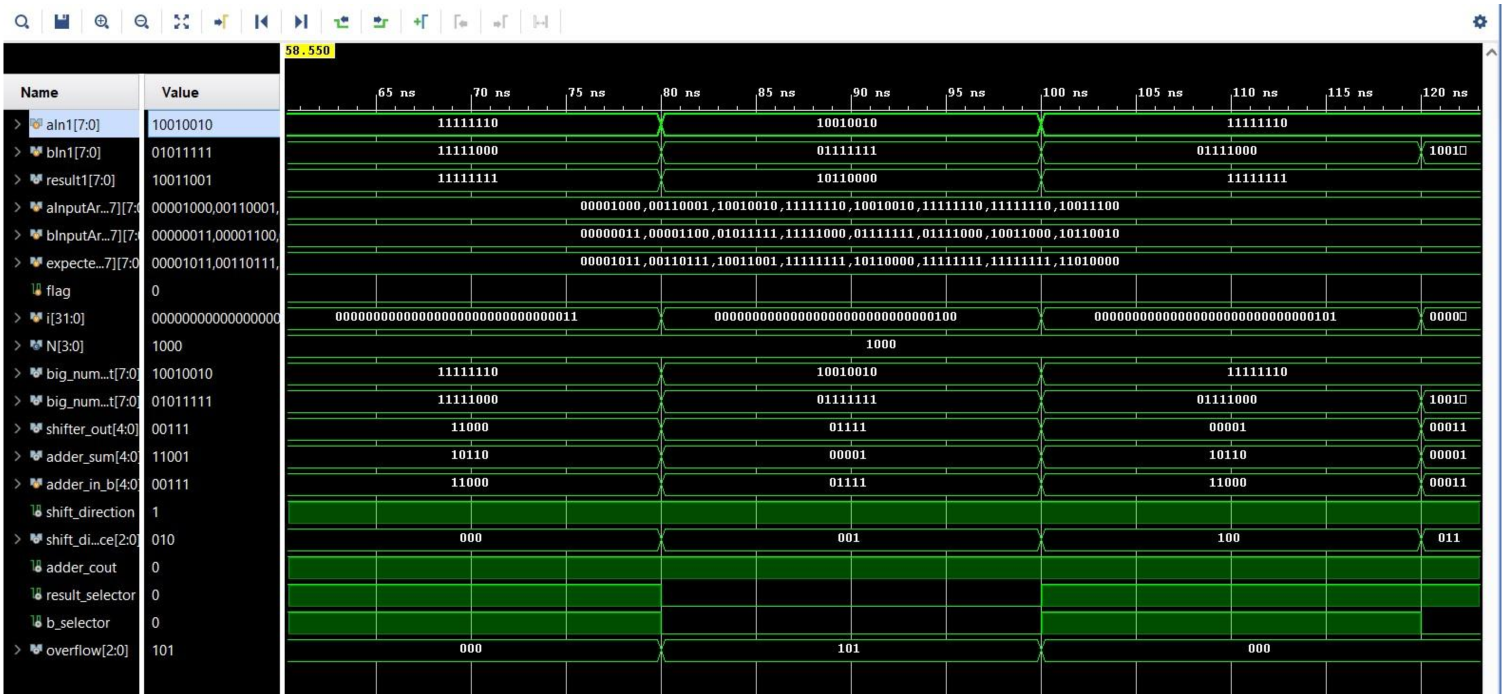
Testbench explanation:

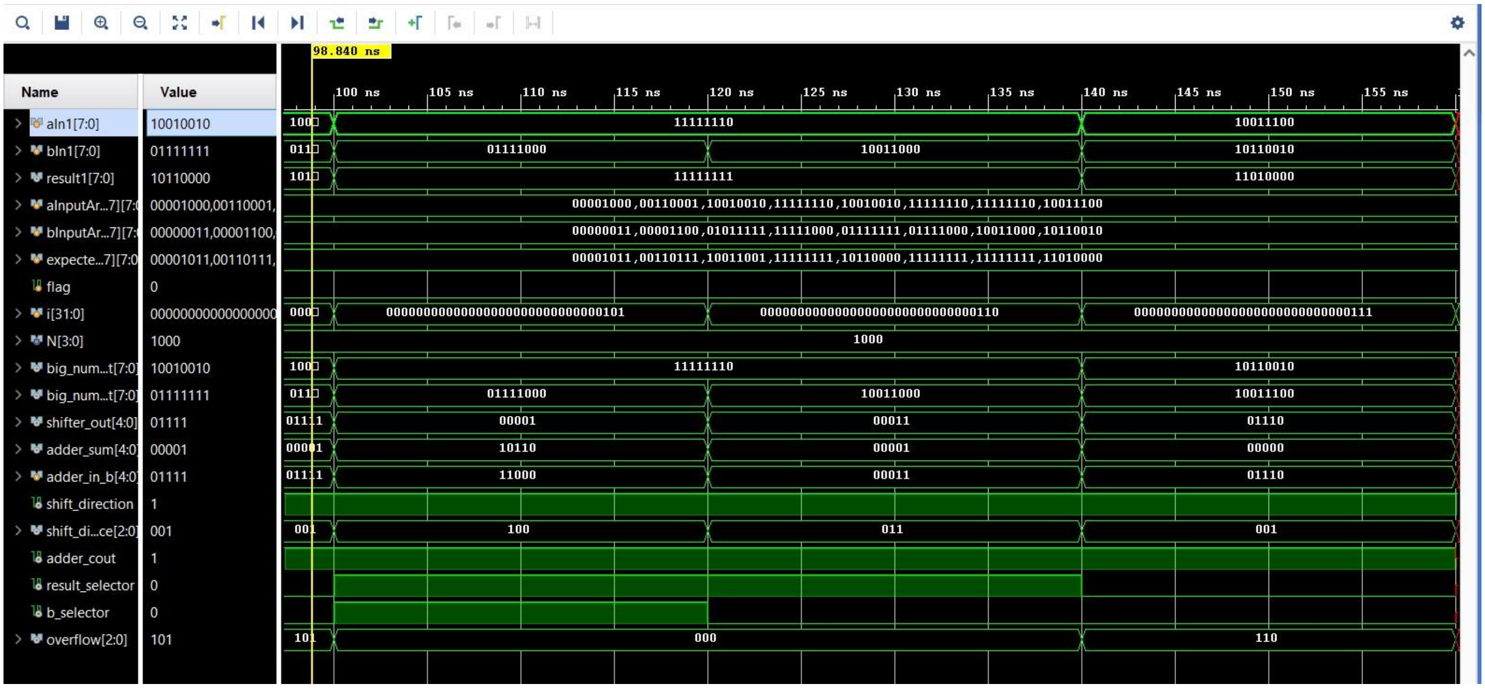
The testbench is considered to be through as we have tested the possible edge cases in multiple ways with different direction and distances, on top of just having regular different cases. The major key was ensuring proper output when we want to shift the number past the edge and that we don't just have the number "wrap-around."

Float_Add

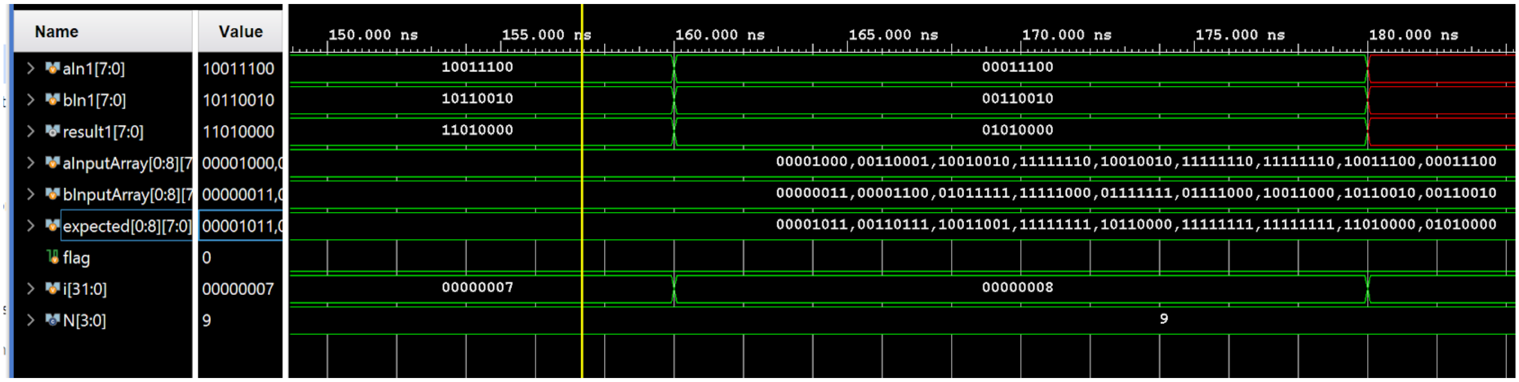

0.000 ns
Name
Value
0 ns
5 ns
10 ns
15 ns
20 ns
25 ns
30 ns
35 ns
40 ns
45 ns
50 ns
55 ns
aIn1[7:0] 00001000 | 00001000 | 00110001 | 10010010
bIn1[7:0] 00000011 | 00000011 | 00001100 | 01011111
result1[7... 00001011 | 00001011 | 00110111 | 10011001
aInputAr... 00001000,00110001, | 00001000,00110001,10010010,11111110,10010010,11111110,11111110,10011100
bInputAr... 00000011,00001100, | 00000011,00001100,01011111,11111000,01111111,01111000,10011000,10110010
expecte.... 00001011,00110111, | 00001011,00110111,10011001,11111111,10110000,11111111,11111111,11010000
flag 0
i[31:0] 00000000000000000 | 00000000000000000000000000000000 | 00000000000000000000000000000001 | 00000000000000000000000000000010
N[3:0] 1000 | 1000
big_num... 00001000 | 00001000 | 00110001 | 10010010
big_num... 00000011 | 00000011 | 00001100 | 01011111
shifter_o... 00011 | 00011 | 00110 | 00111
adder_s... 01011 | 01011 | 10111 | 11001
adder_in... 00011 | 00011 | 00110 | 00111
shift_dir... 1
shift_di..... 000 | 000 | 001 | 010
adder_c... 0
result_se... 0
b_selector 0
overflow... 001 | 001 | 010 | 101

The test cases in this image are the ones given in the handout. We can see that the output matches exactly what we were told to expect.


58.550
Name
Value
65 ns
70 ns
75 ns
80 ns
85 ns
90 ns
95 ns
100 ns
105 ns
110 ns
115 ns
120 ns
aIn1[7:0] 10010010 | 11111110 | 10010010 | 11111110
bIn1[7:0] 01011111 | 11111000 | 01111111 | 01111000 | 1001
result1[7:0] 10011001 | 11111111 | 10110000 | 11111111
aInputAr...7][7:0] 00001000,00110001, | 00001000,00110001,10010010,11111110,10010010,11111110,11111110,10011100
bInputAr...7][7:0] 00000011,00001100, | 00000011,00001100,01011111,11111000,01111111,01111000,10011000,10110010
expecte...7][7:0] 00001011,00110111, | 00001011,00110111,10011001,11111111,10110000,11111111,11111111,11010000
flag 0
i[31:0] 00000000000000000 | 00000000000000000000000000000011 | 00000000000000000000000000000100 | 00000000000000000000000000000101 | 0000
N[3:0] 1000 | 1000
big_num...t[7:0] 10010010 | 11111110 | 10010010 | 11111110
big_num...t[7:0] 01011111 | 11111000 | 01111111 | 01111000 | 1001
shifter_out[4:0] 00111 | 11000 | 01111 | 00001 | 00011
adder_sum[4:0] 11001 | 10110 | 00001 | 10110 | 00001
adder_in_b[4:0] 00111 | 11000 | 01111 | 11000 | 00011
shift_direction 1
shift_di...ce[2:0] 010 | 000 | 001 | 100 | 011
adder_cout 0
result_selector 0
b_selector 0
overflow[2:0] 101 | 000 | 101 | 000



If we take a look at the first two test cases in this image, we see aIn properly returned from big_num_first. With bIn being anything but 0, we should expect an output of 11111111, because if b is 000 00001 then it will correctly add to 111 11111 but if bIn is larger than it will trigger the saturation case which means the resultant is too large of a sum and shall just be represented as the max val possible.



This last case has bIn as larger aIn, and it still returns the correct result. The flag variable stays at a value of 0 and is never set to 1, because that only occurs when the result does not match the expected.

Testbench tcl output:

```
00001000 + 00000011 = 00001011, expected = 00001011
00110001 + 00001100 = 00110111, expected = 00110111
10010010 + 01011111 = 10011001, expected = 10011001
11111110 + 11111000 = 11111111, expected = 11111111
```

```
10010010 + 01111111 = 10110000, expected = 10110000
11111110 + 01111000 = 11111111, expected = 11111111
11111110 + 10011000 = 11111111, expected = 11111111
10011100 + 10110010 = 11010000, expected = 11010000
Passed float_add Test :)
```

Testbench explanation:
We believe our testbench is through as we have test cases that trigger all possible branches that the logic diagram will need to follow - bIn larger than aIn, input whose sum is larger than what can be possibly represented and will return a saturated output, inputs who after adding the mantissa will return a value that is too large to be held in the mantissa and require the exponent to be increased, and the normal case where the exponent does not have to be changed from what is considered to be the largest exponent and will just concatenate the result from adder with the exponent.