Lab 0: Full Adder on FPGA

Emma, March, Vivien

September 28, 2018

Test Cases

We selected a few simple cases at the top of our table to ensure we could quickly check the accuracy of the adder. We then introduced negative numbers, and ensured that the computation still worked as desired. We then moved on to individually test the carryout functionality.

We then tested overflow functionality and edge cases at the bottom of our table.

```
comparch@comparchVM:~/CompArch_HW/Lab0$ ./adder
VCD info: dumpfile adder.vcd opened for output.
A    B    | Cout Sum  Over | Ecout Esum  Eover
0000 0000 |  0   0000  0   | 0     0000  0
0010 0100 |  0   0110  0   | 0     0110  0
0010 1100 |  0   1110  0   | 0     1110  0
0000 0111 |  0   0111  0   | 0     0111  0
1110 1010 |  1   1000  0   | 1     1000  0
1110 1100 |  1   1010  0   | 1     1010  0
1110 0100 |  1   0010  0   | 1     0010  0
0101 0010 |  0   0111  0   | 0     0111  0
1101 1011 |  1   1000  0   | 1     1000  0
0101 0011 |  0   1000  1   | 0     1000  1
1001 1110 |  1   0111  1   | 1     0111  1
0001 1111 |  1   0000  0   | 1     0000  0
1111 0001 |  1   0000  0   | 1     0000  0
1111 1111 |  1   1110  0   | 1     1110  0
1101 1000 |  1   0101  1   | 1     0101  1
0101 0110 |  0   1011  1   | 0     1011  1
```

Figure 1: Output from testbench

Table 1: Test cases with results from testbench and FPGA

			Expected			Testbench	FPGA
Test #	A	B	Carryout	Sum	Overflow	Matched?	Matched?
1	0000	0000	0	0000	0	yes	yes
2	0010	0100	0	0110	0	yes	yes
3	0010	1100	0	1110	0	yes	yes
4	0000	0111	0	0111	0	yes	yes
5	1110	1010	1	1000	0	yes	yes
6	1110	1100	1	1010	0	yes	yes
7	1110	0100	1	0010	0	yes	yes
8	0101	0010	0	0111	0	yes	yes
9	1101	1011	1	1000	0	yes	yes
10	0101	0011	0	1000	1	yes	yes
11	1001	1110	1	0111	1	yes	yes
12	0001	1111	1	0000	0	yes	yes
13	1111	0001	1	0000	0	yes	yes
14	1111	1111	1	1110	0	yes	yes
15	1101	1000	1	0101	1	yes	yes
16	0101	0110	0	1011	1	yes	yes

The adder passed all of the tests we ran. We believe that the adder was able to pass all of the the test cases on the first try due to our process in constructing it. We first constructed a fully functional 2 bit adder using the same daisy-chaining approach to fully work out any kinks in the syntax or function. Thus, when we extrapolated up to a 4 bit adder, it was able to pass our tests.

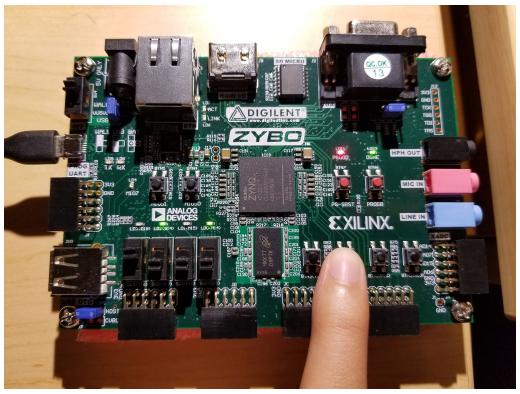
We faced some issues with syntax as we were testing the 2 bit adder. We tried to define the register/wire width by placing the array dimension after the individual register/wire, i.e. *reg a[1:0]*, instead of after reg/wire, i.e. *reg[1:0] a*. As soon as we fixed this error, our 2 bit adder test cases passed.

FPGA Testing

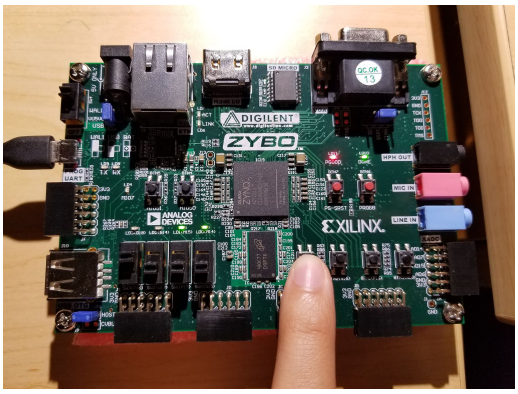
After working through the FPGA tutorial, we moved to test our 4-bit adder manually on the FPGA. We tested through the same test cases as in the testbench.

We faced a hitch when we ran synthesis of the wrapper in Vivado; there was an error opening include file adder.v. For a quick fix, we deleted the line in lab0_wrapper.v and manually loaded adder.v in Vivado. We ran synthesis again and faced yet another hitch because we had used the same gate names, e.g. xorgate, in multiple places. While it had worked in simulation to use the same gate names, this bad coding practice did not hold in practice. After renaming the gates, we were able to run synthesis and implementation, generate bitstream, and load our code onto the FPGA successfully.

The results of our tests are shown in Table 1. All of our test cases passed, with the resulting LED combination after loading A and B (flipping the switches and pressing button 0 and button 1, respectively) equal to the sum when button 2 was pressed and LED0 equal to carryout and LED1 equal to overflow when button 3 was pressed. Below is an image of the output the FPGA gave us for test case #15, with inputs A = 1101 and B = 1000.



(a) Sum = 0101; LED3 = 0,
LED2 = 1, LED1 = 0, LED0 = 1



(b) Carryout = 1, Overflow = 1;
LED1 = 1, LED0 = 1

Figure 2: Test #15: A = 1101, B = 1000

Waveforms

This is the structure of our adder:

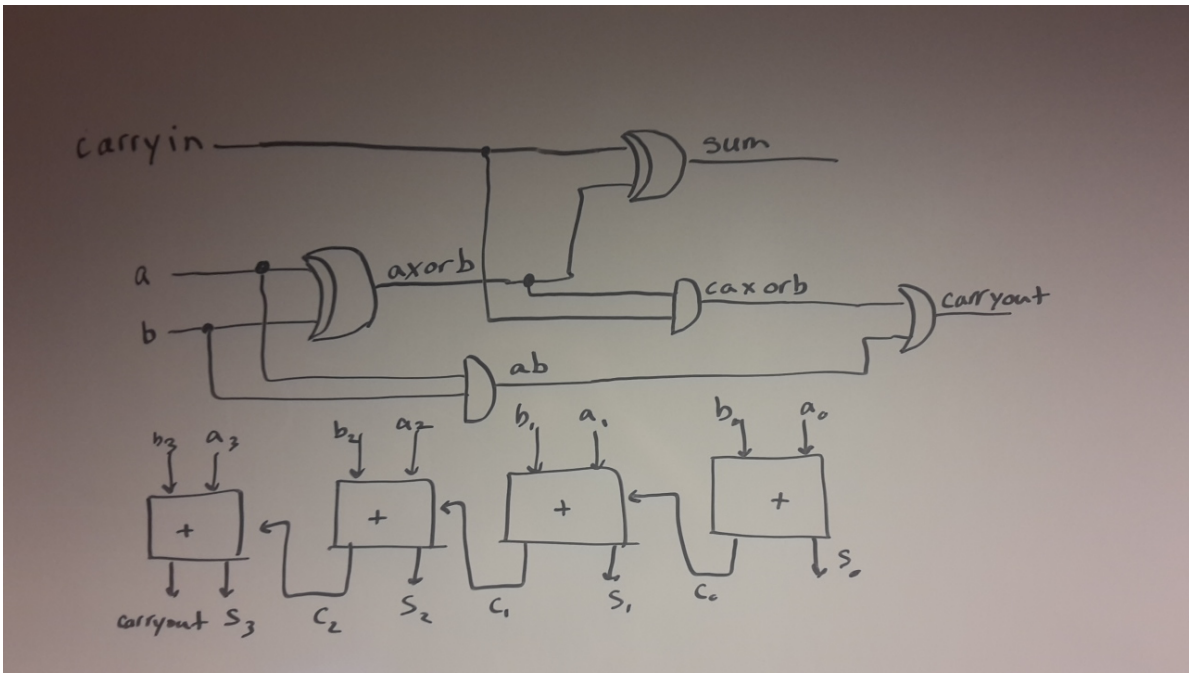


Figure 3: Full adder structure

This table presents the worst case gate delays for 1 bit:

	A	B	Cin
Sum	2	2	1
Cout	3	3	2

Note that the XOR gate that a and b enter to form axorb does not depend on receiving a carryin bit.

There is similar reasoning for the AND gate that forms ab. Therefore, the gate delay propagation depends only on when the carryin bit appears. This results in worst case gate delays of:

	S0	C0	S1	C1	S2	C2	S3	Carryout
A0/B0	2	3	4	5	6	7	8	9

These are the simulation waveforms showing the full adder stabilizing after the changing inputs of our 16 test cases.

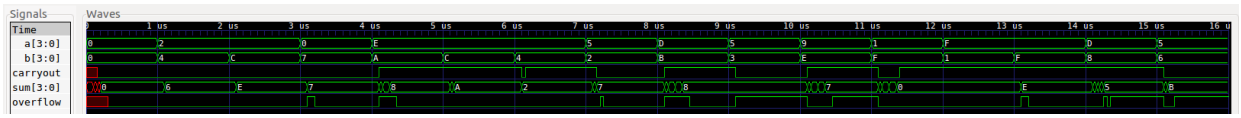


Figure 4: Full adder stabilization after input change

Our inputs, a and b, change instantaneously when we tell them to. Our outputs: carryout, sum, and overflow, all change accordingly after some delay.

The worst case delay occurs when the adder overflows, in which case the sum output is delayed about 380 ns. The worst case sum gate delay we calculated was 8 gate switches, which comes out to $8 * 50ns = 400ns$. We also calculated that the worst case carryout gate delay should be $9 * 50ns = 450ns$, but were unable to locate a test case that gave this result.

The overflow should only be high after 9, 10, 14, and 15 ns. However, we note lots of smaller jumps to high outside of these ranges. This occurs because of the propagation delay between the inputs and the other outputs, and should be ignored. In order to receive a proper output, we should add a debouncer component to the overflow output.

Summary Statistics

We were able to extract summary statistics from Vivado and by visually observed the output waveforms:

Propagation Delay	Value
Sum Propagation Delay (Best Case)	200 ns
Sum Propagation Delay (Worst Case)	380 ns
Carryout Propagation Delay (Best Case)	100 ns
Carryout Propagation Delay (Worst Case)	400 ns

Resource	Utilization	Available	Utilization %
LUT	6	17600	0.03
FF	9	35200	0.03
IO	13	100	13.00
BUFG	1	32	3.13

Figure 5: Utilization of resources