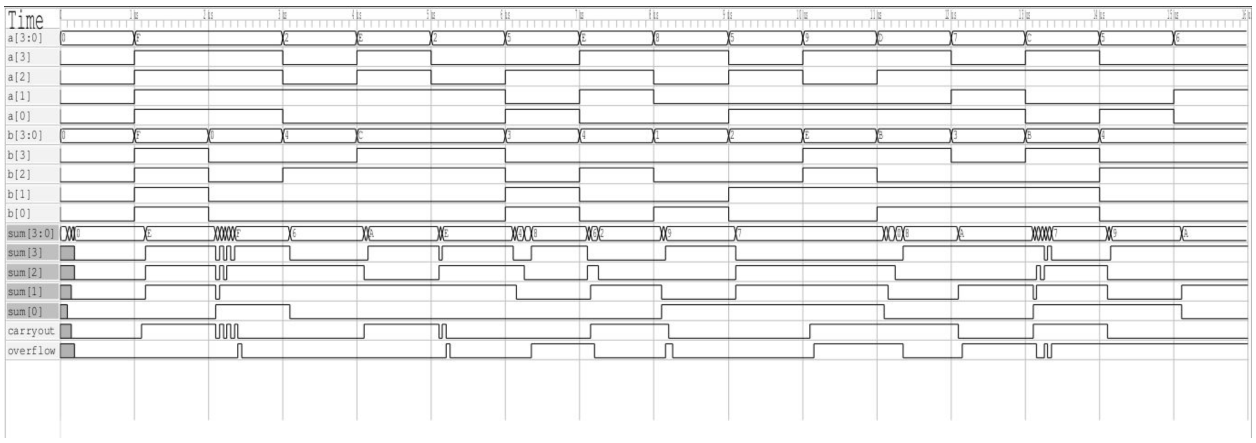
Lab0 Write-Up

Waveforms for full adder



Propagation Delay For 1-Bit Full Adder (Delay = 50 units of time per gate)

	A	B	Cin
Sum	2	2	1
Cout	3	3	2

Propagation Delay For 4-Bit Full Adder (Delay = 50 units of time per gate)

	A/B0	A/B1	A/B2	A/B3
S0	2	-	-	-
C1	3	-	-	-
S1	3 + 1 = 4	2	-	-
C2	3 + 2 = 5	3	-	-
S2	(3 + 2) + 1 = 6	3 + 1 = 4	2	-
C3	(3 + 2) + 2 = 7	3 + 2 = 5	3	-
S3	(3 + 2 + 2) + 1 = 8	(3 + 2) + 1 = 6	3 + 1 = 4	2
Cout	(3 + 2 + 2) + 2 = 9	(3 + 2) + 2 = 7	3 + 2 = 5	3

Propagation Delay For Overflow Detection (Delay = 50 units of time per gate)

	C3	Cout
Overflow	1	1

Above is shown the number of gates that an input must pass through to reach the output node. The time delay per gate is 50 units of time, so by multiplying the number of gates by 50, we can calculate the time delay from the table above. These delays are reflected in the waveform graph generated by our test cases, shown above. Because all 4 bits in A and B are input into the adder at the same time, there is a propagation delay in the carryout of each 1-bit adder, which will affect the sum bits. It is necessary to wait 9 * 50 units of time for the circuit to stabilize before measuring the output. The overflow indicator passes through an additional gate with a delay of 50 time units, so we must wait an additional unit of time for this output to stabilize for a total of 10 * 50 = 500 time units. Because we measured our test cases after 1000 units of time, we gave the circuit plenty of time to stabilize, and we received the expected output at each of our test cases.

Test case strategy

Inputs	Outputs	Reasoning

A = 0000
B = 0000
	Cout = 0
	Overflow = 0
	Sum = 0000
		This is a very basic test case but it is a good starting one to determine any obvious errors in our initialization of the 4-bit adder.

A = 1111
B = 1111
	Cout = 1
	Overflow = 0
	Sum = 1110
		This is another basic test case but it has more interesting results - with a Cout but no Overflow. We were trying out all the combinations of Cout and Overflow throughout our testing. This also has Cins and Couts of 1 between the individual adders, so we wanted to test out that functionality.

A = 1111
B = 0000
	Cout = 0
	Overflow = 0
	Sum = 1111
		This is another very basic test case with a non-0000 sum. This also has Cns and Couts of 0 between adders.

A = 0010
B = 0100
	Cout = 0
	Overflow = 0
	Sum = 0110
		Here is another basic test case - we have our fair share of Cout = 0 and Overflow = 0 combinations, but these tests really help make sure that the sum calculated is correct.

A = 1110
B = 1100
	Cout = 1
	Overflow = 0
	Sum = 1010
		This is another combination of Cout = 1 and Overflow = 0 but with ‘less basic’ binary numbers (as opposed to the A = 1111 and B = 0000 case). This also has Cins and Couts of 1 between the individual adders.

A = 0010
B = 1100
	Cout = 0
	Overflow = 0
	Sum = 1110
		Basic test case with a non-0000 sum and also ‘less basic’ binary numbers.

A = 0101
B = 0011
	Cout = 0
	Overflow = 1
	Sum = 1000
		This test case has an Overflow and not a Carryout. It was chosen because we hadn't tested this exact combination before.

A = 1110
B = 0100
	Cout = 1
	Overflow = 0
	Sum = 0010
		As mentioned, we wanted to have a fair share of 'all possible Cout and Overflow combinations,' especially when they are different values. This is because we actually found an error in our implementation/understanding of Cout and Overflow through one of these types of tests (See *Test case failures and design changes*) for more information)

A = 1000
B = 0001
	Cout = 0
	Overflow = 0
	Sum = 1001
		This case makes sure the actual 'summing' function of our 4-bit adder is behaving as expected.

A = 0101
B = 0010
	Cout = 0
	Overflow = 0
	Sum = 0111
		Another test chosen based on whether the summing part of our 4-bit adder works.

A = 1001
B = 1110
	Cout = 1
	Overflow = 1
	Sum = 0111
		This is a case where the Cout and the Overflow are both one, and was chosen for that reason.

A = 1101
B = 1011
	Cout = 1
	Overflow = 0
	Sum = 1000
		This is another case whether the Cout = 0 and Overflow = 0 with non-0 Cins and Couts between some of the individual adders in our 4-bit adder implementation.

A = 0111
B = 0011
	Cout = 0
	Overflow = 1
	Sum = 0101
		This is another case where Cout = 0 and Overflow = 1 just to make sure this type of output works with our implementation.

A = 1100
B = 1011
	Cout = 1
	Overflow = 1
	Sum = 0111
		Another case whether both Cout and Overflow are 1, just to make sure that our implementation works.

A = 0101
B = 0100
	Cout = 0
	Overflow = 1
	Sum = 1001
		At this point in choosing test cases, we wanted to have more cases with Overflows. We also wanted to make sure that Cout was 0, because this was when we realized our implementation was wrong. Also, this also has a point where there is a Cout on one of the adders initialized within the 4-bit adder.

A = 0110	Cout = 0	See above explanation.

B = 0100	Overflow = 1
	Sum = 1010	

Test case failures and design changes

Inputs	Test Outputs	Actual Outputs	Error/Design Change

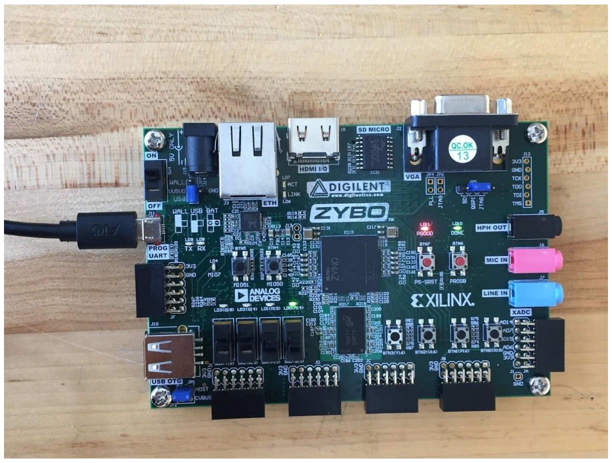
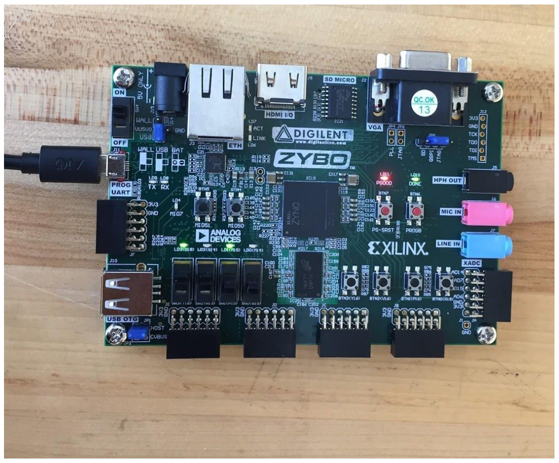
A = 1111
B = 1111
	Cout = 1
	Overflow = 1
	Sum = 1110
		Cout = 1
		Overflow = 0
		Sum = 1110
			We had initially thought that the final Cout was equal to the Overflow. So, in our implementation, we had used a buffer to set these outputs equal to each other. However, we realized that it was strange that the final Cout is *always* equal to Overflow (because then, what was the point of having both of those outputs?). Referring to the notes, we realized our understanding, and therefore our implementation, of the 4-bit adder was wrong. We then changed Overflow to equal whether the final Cout is equal to the Cin of the most significant bit. But . . .

A = 1111
B = 1111
	Cout = 1
	Overflow = 1
	Sum = 1110
		Cout = 1
		Overflow = 0
		Sum = 1110
			. . . The Overflow still showed up as 1! This is because we implemented overflow with an or. Or will also be 1 if Cout and Cin to the most significant bit are both 1 - which we don't actually want (in that case there is no overflow). We then changed our implementation to xor - which does what we want. We only want Overflow to be 1 when the final Cout is not equal to previous Cin. Xor is only true when input any sequence of 0, 1 (which will be input when Cout is not equal to the previous Cin).

A = 0101
B = 0011
	Cout = 0
	Overflow = 1
	Sum = 1000
		Cout = 0
		Overflow = 1
		Sum = 1000
			This might seem not seem like a problem - the test output is equal to the actual output! However, we had initially defined this combination of operands to be:
			Cout = 0
			Overflow = 0
			Sum = 1000
			In our expected results. So even though it wasn't an error in our initialization, it was an error in what we solved by hand.

---	---	---	Sometimes we wouldn't even get an output from our test bench - this was due to incorrect coding. The main three problems we had were: 1. Not initializing a, b, or sum as having 4-bits and not parsing through it correctly. We fixed this by defining a as a[3:0]. 2. Trying to initialize values at the end of our adder.v file (i.e. make a reg and input a0 for the first adder, and later set that equal to a[0]). We fixed this by removing all these extraneous variables and setting a[0] in our actual implementation. 3. Not specifying all numbers as binary with **'b** and not specifying the initial 0 we passed in was **1'b0**. The error message kept on saying we were passing in 32 bits but it only expected 4 bits. We remedied this by actually defining it as binary and then defining the 0 as 1 bit long only.

FPGA Board Testing

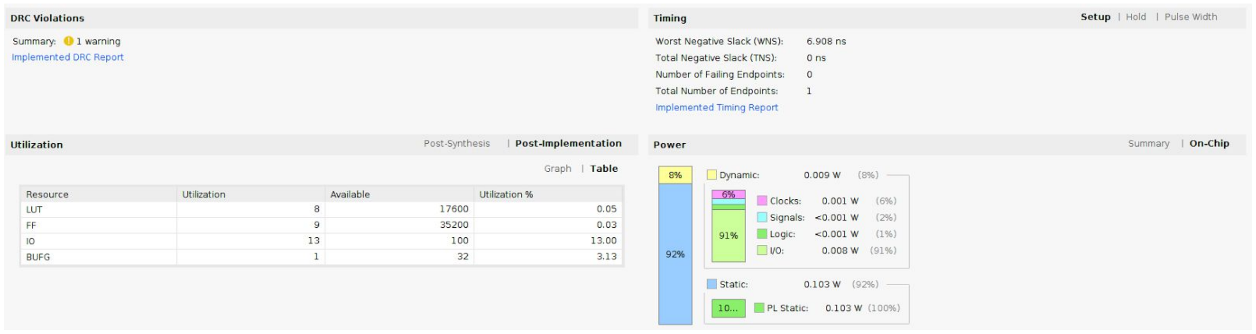



Using the FPGA to add A = 1110 and B = 1100. On the left is the board displaying the overflow and carryout (overflow = 0, carryout = 1). On the right, the board is displaying the sum (sum = 1010).

Input	Sum	Cout, Overflow	Pass

Through switches Flipped up → 1 Flipped down → 0	*Through sequential order (LD3 most significant bit, LD2, LD2, LD0 least significant bit) of LEDs on = 1, off = 0*	*LD0 on → Cout = 1 LD0 off → Cout = 0 LD1 on → Overflow = 1 LD1 off → Overflow = 0*	**Test?**
0000 + 0000	0000	0, 0	Yes
1111 + 1111	1110	1, 0	Yes
1111 + 0000	1111	0, 0	Yes
0010 + 0100	0110	0, 0	Yes
1110 + 1100	1010	1, 0	Yes
0010 + 1100	1110	0, 0	Yes
0101 + 0011	1000	0, 1	Yes
1110 + 0100	0010	1, 0	Yes
1000 + 0001	1001	0, 0	Yes
0101 + 0010	0111	0, 0	Yes
1001 + 1110	0111	1, 1	Yes
1101 + 1011	1000	1, 0	Yes
0111 + 0011	0101	0, 1	Yes
1100 + 1011	0111	1, 1	Yes
0101 + 0100	1001	0, 1	Yes
0110 + 0100	1010	0, 1	Yes

Summary Statistics



DRC Violations

Summary: ● 1 warning

Implemented DRC Report

Timing — **Setup** | Hold | Pulse Width

Worst Negative Slack (WNS):	6.908 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	1

Implemented Timing Report

Utilization — Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	8	17600	0.05
FF	9	35200	0.03
IO	13	100	13.00
BUFG	1	32	3.13

Power — Summary | **On-Chip**

Dynamic:	0.009 W	(8%)
Clocks:	0.001 W	(6%)
Signals:	<0.001 W	(2%)
Logic:	<0.001 W	(1%)
I/O:	0.008 W	(91%)
Static:	0.103 W	(92%)
PL Static:	0.103 W	(100%)

Above is a screenshot of summary statistics, namely DRC violations, Timing, Utilization (Resources) and Power. I/O has a larger utilization because the bulk of this lab was sending inputs (the operands), and discerning outputs (through lights). BUFG also has a ‘high’ utilization percentage because there is a global clock net being driven from the lab0_wrapper.v file.