CompArch Lab 0: Full Adder on FPGA

Annie Kroo, Nathan Yee

Development

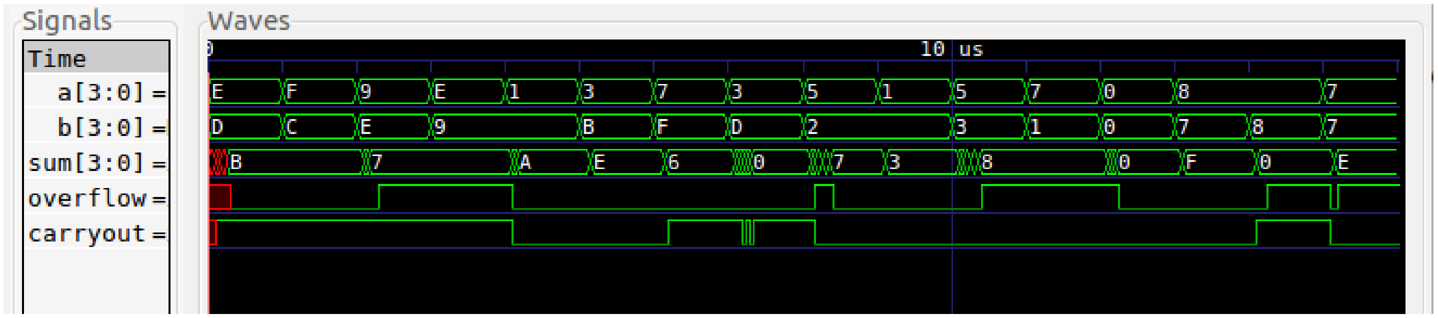
During the early stages of development we used a 4 bit unsigned number system instead of 2’s complement. As a result the overflow would always be true when carryout true. After switched to using the 2’s complement number system by adding additional gates, we properly handled overflow output.

Test bench failures

The only time the test bench caught an error in our circuit was when we added two negative numbers that shouldn’t overflow. Fr example, “-2 + -3 = -5” would overflow. We realized that we needed to invert the signing bit of the sum before we used it in overflow calculations and subsequently added an inverter to fix the error.

Timing Waveforms

Using GTKwave we viewed the delay of the sum, carryout and overflow. In our cases the worst case delay occurred in our overflow where there was a delay of 400ns or 8 gate delays. By tracing the signal through the gates, it is clear that the maximum possible delay is 9 gate delays or 450ns. The asymmetry in our design results in glitches to occur. These glitches are primarily the carry outs propagating through the bit spliced chain of full adders.


Signals
Waves
Time
a[3:0] =
b[3:0] =
sum[3:0] =
overflow =
carryout =
0
10 us
E F 9 E 1 3 7 3 5 1 5 7 0 8 7
D C E 9 B F D 2 3 1 0 7 8 7
B 7 A E 6 0 7 3 8 0 F 0 E

Testing

In selecting test cases we broke up our test cases into four distinct categories to cover a wide variety of inputs. We looked at the outcome of the sum of two negative numbers, one positive and one negative number, and two positive numbers. Within these cases we selected test cases that produced a different combinations of carry out and overflow values as well as testing other properties of our adder. The Full Adder successfully passed 100% of the test cases.

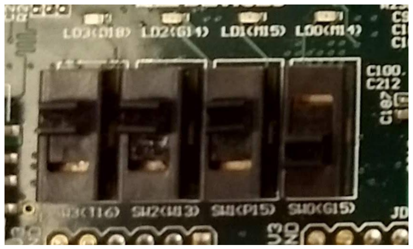
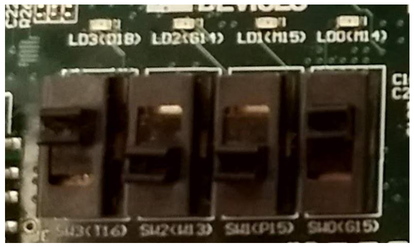
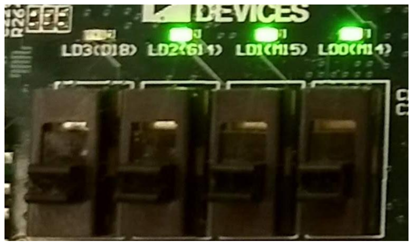
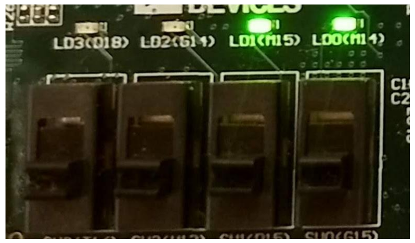
Test		Inputs		Theoretical Results			Simulated Results			FPGA Results		
Index	Equation	A	B	Sum	Carry Out	Overflow	Sum	Carry Out	Overflow	Sum	Carry Out	Overflow
0	-2 + -3 = -5	1110	1101	1011	1	0	1011	1	0	1011	1	0
1	-1 + -4 = -5	1111	1100	1011	1	0	1011	1	0	1011	1	0
2	7 + -2 = 8	1001	1110	0111	1	1	0111	1	1	0111	1	1
3	-2 + -7 = 8	1110	1001	0111	1	1	0111	1	1	0111	1	1
4	1 + -7 = -6	0001	1001	1010	0	0	1010	0	0	1010	0	0
5	3 + -5 = -2	0011	1011	1110	0	0	1110	0	0	1110	0	0
6	7 + -1 = 6	0111	1111	0110	1	0	0110	1	0	0110	1	0
7	3 + -3 = 0	0011	1101	0000	1	0	0000	1	0	0000	1	0
8	5 + 2 = 7	0101	0010	0111	0	0	0111	0	0	0111	0	0
9	1 + 2 = 3	0001	0010	0011	0	0	0011	0	0	0011	0	0
10	5 + 3 = -8	0101	0011	1000	0	1	1000	0	1	1000	0	1
11	7 + 1 = -8	0111	0001	1000	0	1	1000	0	1	1000	0	1
12	0 + 0 = 0	0000	0000	0000	0	0	0000	0	0	0000	0	0
13	-8 + 7 = -1	1000	0111	1111	0	0	1111	0	0	1111	0	0
14	-8 + -8 = 0	1000	1000	0000	1	1	0000	1	1	0000	1	1
15	7 + 7 = -2	0111	0111	1110	0	1	1110	0	1	1110	0	1

Index	Equation	Reason
0	-2 + -3 = -5	Adding two negative numbers close to zero to get carryover but no overflow.
1	-1 + -4 = -5	Adding two different negative numbers from the first test case to verify output consistency.
2	-7 + -2 = 8	Adding two more negative numbers such that there the sum overflows as well as has a carry out.
3	2 + -7 = 8	Adding the same two negative numbers as the previous example but in different orders to validate the commutative property of our adder.
4	1 + -7 = -6	Adding a positive and a negative number that sum to a negative number to get no overflow and no carry out.
5	3 + -5 = -2	Again adding a positive and negative number that sum to a negative number to show the generalization of the no overflow and no carry out property shown in the previous example.
6	7 + -1 = 6	Adding a positive number with a small negative to get a positive number to show carry out without overflow.
7	3 + -3 = 0	Adding a number to the negative of that number to show the zero sum case holds.
8	5 + 2 = 7	Adding two relatively small positive numbers that sum to the edge of the range of positive numbers to show no overflow and no carry out.
9	1 + 2 = 3	Adding two small positive numbers to comfortably get a number in the range.
10	5 + 3 = -8	Adding two relatively large positive numbers to get overflow but no carry out.
11	7 + 1 = -8	Adding two different relatively large positive numbers to get the same as the previous example with overflow and no carry out.
12	0 + 0 = 0	Adding two zeros to get zero sum case out with no overflow and no carryout.
13	-8 + 7 = -1	Adding the most negative value in our domain with the most positive to show the edge case of the carry out and no overflow positive plus negative sum.
14	-8 + -8 = 0	Adding the most negative number to the most negative number in our domain to show the edge case of our sum of two negative numbers overflowing and having a carry out.
15	7 + 7 = -2	Adding the highest positive number to the highest positive number in our domain to show the edge case of our sum of positive numbers overflowing with no carry out.

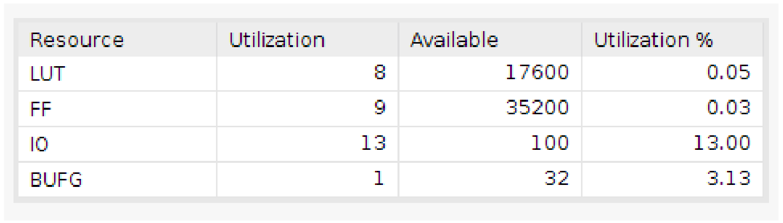
FPGA

We took the results from our test bench and moved forward to implement our design in hardware. Using the FPGA interpreter Vivado we synthesized, implemented, and generated a bitstream of our Verilog code. This code was then uploaded to the board and allowed us to visualize our adder. To do this we utilized a provided wrapper for our Verilog code that interacted with both our code and the Vivado software itself. Through this process we were able to use a set of built in switches on our Zybo FPGA board to set the two 4 bit signed binary strings that were to be added (A and B). Using built in buttons to set and hold the configuration for A and B, the selection of a third button resulted in the display of the sum with LEDs. The final built in button displayed on the LEDs whether the sum overflowed and whether there was a carry over out of the most significant digit's single bit adder in the chain.

The FPGA was programmed and showed that it followed the theoretical and simulated behavior. Shown below is test case index 3.

Input A	Input B	Output Sum	Output Carryout and Overflow
-2 = 1110	-7 = 1001	8 = 0111	Carryout = 1 | Overflow = 1
			

Vivado's synthesis tool used to generate the bitstream to upload to the FPGA gives additional information about the space utilization of the program relative to the boards capacity.



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

As this is a very light weight calculation and simple circuit, we notice that we do not take up very much space on the FPGA relative to its capacity. The downside of this format is an asymmetry in the design resulting in glitches in the sum, carryout, and overflow.