Computer Architecture: Lab b000

Lauren Anfenson
Anusha Datar
Will Fairman

September 2018

1 Background

A full adder takes two bits and a carryin bit and adds them together, outputting two bits that represent the sum and the carryout of the operation. To create a four bit adder we cascaded four full adders, taking the carryout of the first operation and making it the carryin of the second operation, continuing this through operations on all four bits. Our general architecture can be seen in the diagram in Figure 1 – this is formally known as a "ripple carry adder."

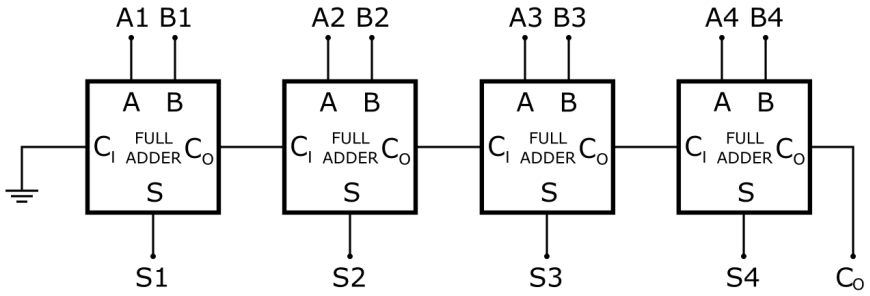

Figure 1: Simple schematic of a 4 bit ripple carry adder

Our overflow flag is triggered when the sum of two numbers is out of the range that can be expressed in four-bit 2's complement. This event can be observed when the carryin to the most significant bit is different than the carryout of the most significant bit, so we implemented it with a simple XOR gate between the last carryin and carryout to express the overflow status.

2 Test Cases

When selecting our test cases, we chose a variety of situations that would test both the functionality of the subcomponents of the four bit adder and the subcomponents' ability to interface with each other. We also wanted to ensure we included key edge cases and that our overflow flag functioned as expected.

More specifically, we included test cases that tested each carryout bit, each carryin bit, each permutation of addition on each bit, and the overflow flag, In simulation, we were able to build a layered framework with which to test our operations: we started by simply performing 0000 + 0000 and 1111 + 1111 to ensure that the addition functions worked. We then added 0000 and 1111 in both permutations to confirm that the results are the same and that our adder is commutative as expected. After we knew that basic addition functioned as expected, we could test carryin and carryout. The next step was adding 1111 with a one in each bit in bus B to make sure each individual bit added correctly with carryin. We also made sure that some of the situations triggered the overflow flag and/or the final carryout so that we could validate the

functionality of those as well. A full list of our test cases and results can be seen in Table 1 and in our testbench file.

Once our test cases had been validated in iverilog we uploaded our code to the FPGA using the Lab 0 wrapper. We then ran through our test cases using the switches and LEDs on the FPGA, recreating all the cases shown in the table below.

A	B	Sum	Carryout	Overflow
Testing Addition Operations/Bit				
0000	0000	0000	0	0
1111	1111	1110	1	0
1111	0000	1111	0	0
0000	1111	1111	0	0
Testing Carryout and Carryin/Bit and Overflow				
1111	0001	0000	1	0
1111	0010	0001	1	0
1111	0100	0011	1	0
1111	1000	0111	1	1
0001	1111	0000	0	0
0010	1111	0001	1	0
0100	1111	0011	1	0
1000	1111	0111	1	1
General Testing				
1000	0001	1001	0	0
1000	0011	1011	0	0
1000	0111	1111	0	0
1010	1111	1001	1	0

Table 1: Test Cases Performed

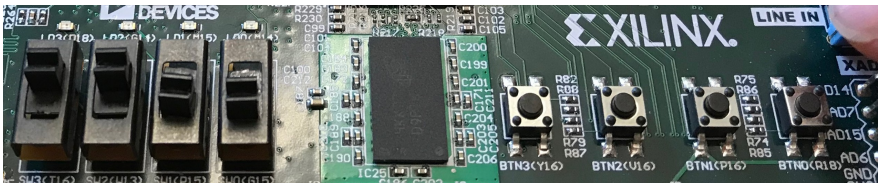

Figure 2: Selecting A as 1100

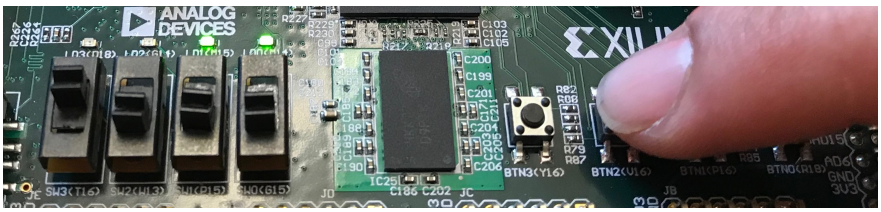

Figure 3: Selecting B as 1000

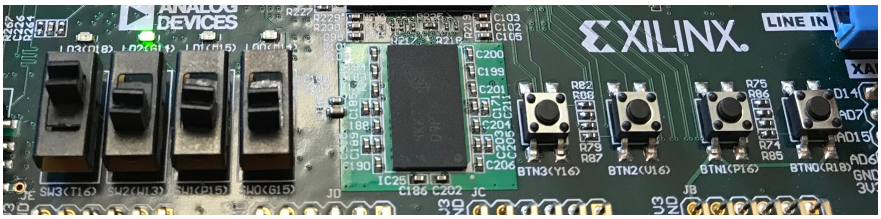

Figure 4: Sum

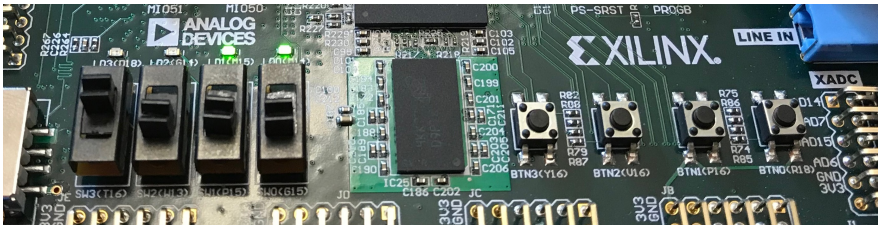

Figure 5: Carryout and Overflow

2.1 Complications

The first failure we noticed was that our sum nibble was flipped from our expected sum value. So, when adding 0010 and 0001, we would get a sum nibble of 1100 instead of our expected 0011. We realized that this was due to us defining our buses in iverilog using [0:3], which assumes that our bits are Least Significant Bit first. By changing our definitions to [3:0], so we were consistently working in Most Significant Bit, we were able to fix this issue.

Another failure that influenced the design of our four bit adder was noticing that the first adder in the chain does not need a carryin bit because there's no operation that comes before it. When using our test bench we got around this by setting our first carryin bit to 0 for all test cases. We remedied this when putting our code onto the FPGA by creating a new sub-module of a half adder, which is like a full adder but without a carryin bit. We then used this half adder as the first of our four cascaded adders.

3 Results

Our adder passed all of the test cases in simulation and on the FPGA. We created a .vcd file from our simulation test bench and loaded it into gtkwave to visualize our waveforms and propagation delays. The first interesting behavior we noticed was that our sum, carryout, and overflow are disconnected for a while after the A and B nibbles are set, as seen in the figure below.

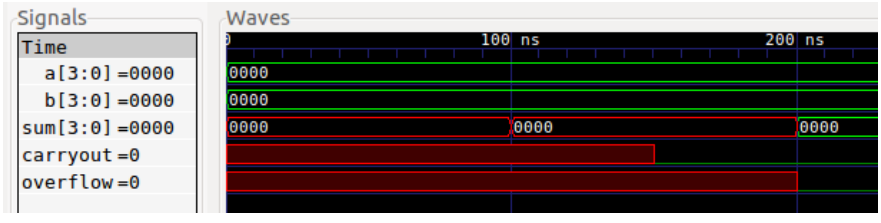

Figure 6: Disconnected outputs at the start of the simulation

We determined that this is because these outputs do not have a prior value when the simulation starts, so they are not driven until the inputs propagate through the circuit to drive them. We can see that the overflow is one time step behind the carryout, which is consistent with what we would expect because there is one XOR gate between carryout and overflow. It's more interesting that the sum is also one time step behind the carryout – we think that this is because the sum is not fully driven until the last carryin

bit has been propagated through the adders, and there are more gates between the carryin bit and the sum than there are between the carryin bit and the carryout bit.

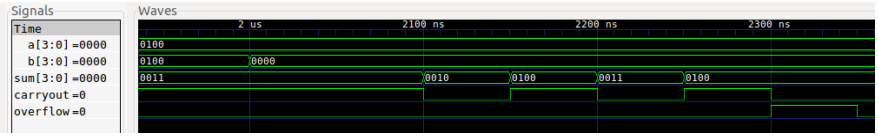

Figure 7: Propagation to the addition of b0010 and b0010.

The next behavior that we analyzed was the propagation of our results. Due to the small (2 timesteps) delay between the individual inputs for each bit and their respective sum output, the sum of the 4-bit adder is generated the quickest. However, this sum is not accurate until the carryin and carryout bits travel through each of the four full adders. This causes the sum output to flip flop until the carryin and carryout bits settle. However, because the overflow and final carryout pins are the last to change in the sequence of adders, they experience very little to no flipping until they output their final state.

Following confirming the accuracy of our computations, we looked at the summary statistics provided by Vivado to find the resources used by our program.

Figure 8: Summary Statistics from Vivado

4 Reflection

By completing this lab exercise, we learned several important verilog concepts such as buses and assigning values to buses, byte order and how to enforce it throughout the program, and general syntax. Most of this learning occurred by means of trial and error, which seems so far to be the best way to learn these concepts. In addition to how to write verilog for simulation, we also learned how to program the FPGA using Vivado and about the importance of verifying that code works in simulation before testing it on actual

hardware. When determining which cases to test on the FPGA, we gained experience with developing test suites that optimize for coverage and follow a logical order such that we were always building on the success of previous cases. Developing this list in advance allowed us systematically validate our design and implementation.