Computer Architecture: Lab 0

Will Derksen, Chris Aring

September 25, 2017

1 Introduction

In this lab we used our structural full adder design to create a ripple carry adder. A full adder is able to add three one-bit binary numbers "a", "b", and "carry in" to output two one-bit binary numbers "sum" and "carry out."
We created a ripple carry adder that could handle 4-bit binary numbers by combining multiple full adders. A ripple carry adder is able to add two n-bit numbers by combining multiple full adders where each full adder's "carry in" is the "carry out" of the previous adder. The first full adder can be replaced by a half adder if there is no initial "carry in" or it can be left as a full adder with a "carry in" of 0.
The ripple carry adder is relatively slow because each full adder has to wait for the finished calculation from the previous full adder.

2 Schematic

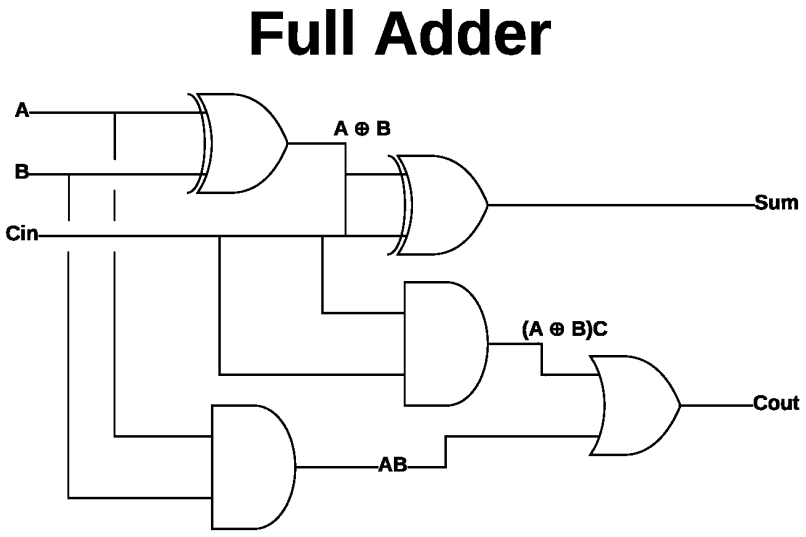

Figure 1: Our full adder schematic

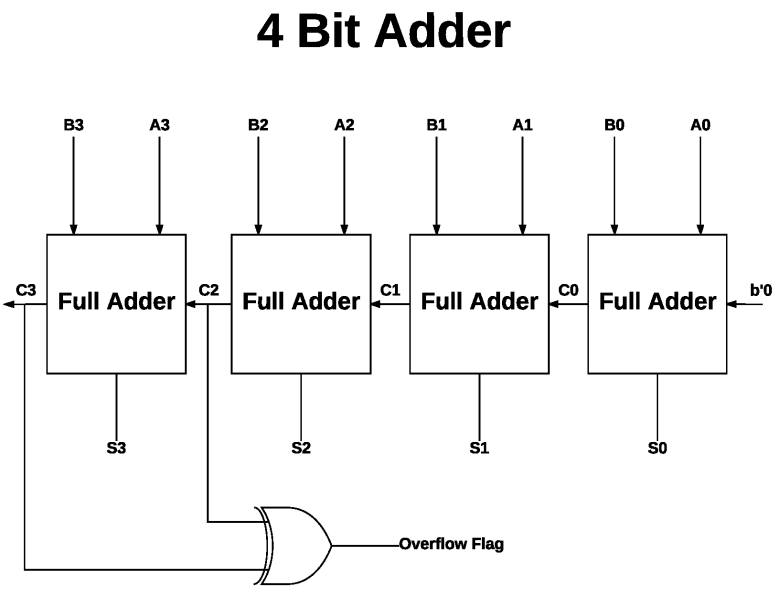

Figure 2: Fully implemented 4-bit adder with overflow flag

3 Wave Forms

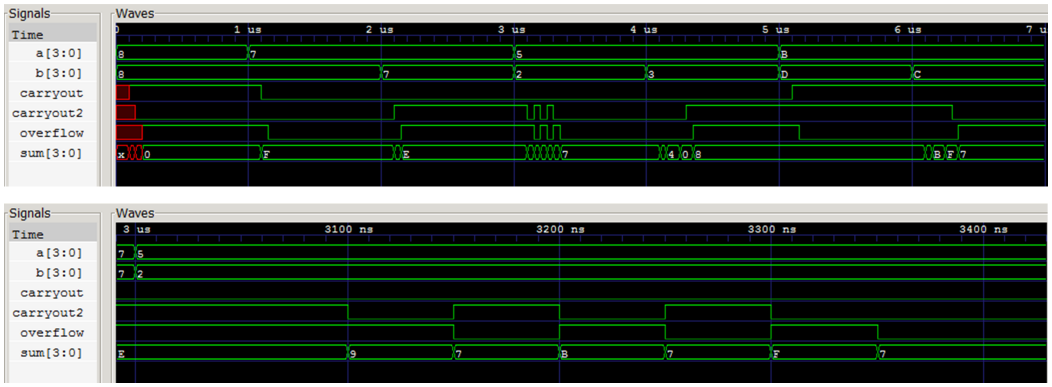

Figure 3: Full waveform and waveform showing the full adder stabilizing after changing inputs

Based on Figure 1, we developed a table of delays based from inputs to outputs.

	A	B	Cin
Sum	2	2	1
Cout	3	3	2

Table 1: Delay Table

From this table we can determine the maximum delay from the inputs A0 and B0 to the outputs Cout and Sum3. Since the time to Cout is generally longer than Sum3, Cout will be the last wire to update. The number of gate delays encountered is the initial time from A and B to Cout, and

then from Cin to Cout. This leads to the following equation for gate delay with this full adder setup.

$$D = 2 + 2(F_a - 1) \tag{1}$$

where D is the number of gate delays, and F_a is the number of full adders. Therefore, for our 4-bit adder system, there is a 8 gate delays in our system at most. In our test bench we approximate this delay to be about 50 nanoseconds per gate, therefore, the maximum gate delay of the system should be 400 nanoseconds.

4 Test Cases

The strategy we used for our test cases were to try the most extreme values we could in our operation and make sure they correctly add or overflow based on our predictions. Then we thought to add test cases that barely overflow or almost overflow, for example anything that adds to -9 or 8 for those that barely overflow, and anything that adds to -8 or 7 for those that almost overflow.

4.1 Chosen Test Cases

For the test cases we choose to

a	b	S	C2	COut	OverFlow	Sum	ECout	EOvrflow
0	0	0	0	0	0	0	0	0
-1	-1	-2	1	1	0	-2	1	0
1	1	2	0	0	0	2	0	0
7	-7	0	1	1	0	0	1	0
6	-2	4	1	1	0	4	1	0
6	5	-5	1	0	1	11(-5)	0	1
5	-7	-2	0	0	0	-2	0	0
7	3	-6	1	0	1	10(-6)	0	1
-5	-5	6	0	1	1	-10(6)	1	1
-8	-8	0	0	1	1	-16(0)	1	1
7	-8	-1	0	0	0	-1	0	0
7	7	-2	1	0	1	14(-2)	0	1
5	2	7	0	0	0	7	0	0
5	3	-8	1	0	1	8(-8)	0	1
-5	-3	-8	1	1	0	-8	1	0
-5	-4	7	0	1	1	-9(7)	1	1

Figure 4: Table of our test cases

4.2 Test Case Failures

One of the first test case failures was that we tried to set the 'a' and 'b' registers to base-10 integers which did not work. We fixed this by making them binary numbers.
Additionally, our original wiring for the overflow flag was the XOR between the Sum[3] and Carryout, aka the two most significant bits of the resulting sum number, one of which is outside of the bit range of our adder (the fifth bit). When we started testing this didn't work because it set the flag to true with the addition of 7 and -8. This is -1 and has a carry of 0 with a signed bit

or sum[3] being 1. From this we learned that the overflow flag would be set incorrectly using this method for any negative numbers added to positive numbers since they all result in the most significant bit (the signed bit) and the carryout being different and thus setting the overflow flag. After this we iterated a bit and discovered that we could use the carryout[2] xor carryout[3] in order to determine whether something overflows. The logic is as follows:
Given two positive numbers, a carryout[3] has to be 0, but if there is a carryout[2] when it is summed with the a[3] and b[3] which we know are zero since those are the signed bits, then the resulting number is negative. Meanwhile when there isnt a carry then there isnt overflow
Given one negative number and one positive number, a carryout[3] depends on carryout[2]. If carryout[2], then the resulting number is positive and the carryout[3] is also true meaning no overflow. If not carryout[2], then the resulting number is negative and the carryout[3] is false meaning no overflow.
Given two negative numbers, a carryout[3] will be true. If carryout[2] then the number is still negative and thus there isnt an overflow, logically and with this method. If not carryout[2] then the number is not negative and thus overflow happened, logically and with this method.

	C[3]	Ovflw	Exp. Ovflw
2 Positive; C[2]=1	0	1	1
2 Positive; C[2]=0	0	0	0
1 Each; C[2]=1	1	0	0
1 Each; C[2]=0	0	0	0
2 Negative; C[2]=1	1	0	0
1 Negative; C[2]=0	0	1	1

Table 2: Caption

5 FPGA board Testing

As a detailed process step, we decided to add b0001 and b1101 on the FPGA. Our expected Sum was 1110 with an expected carryout and overflow of 0. As seen in figure 3, the top left photo is before any numbers are set, The top right photo is setting the first number to b0001, and the bottom left photo is us setting the second number to b1101.

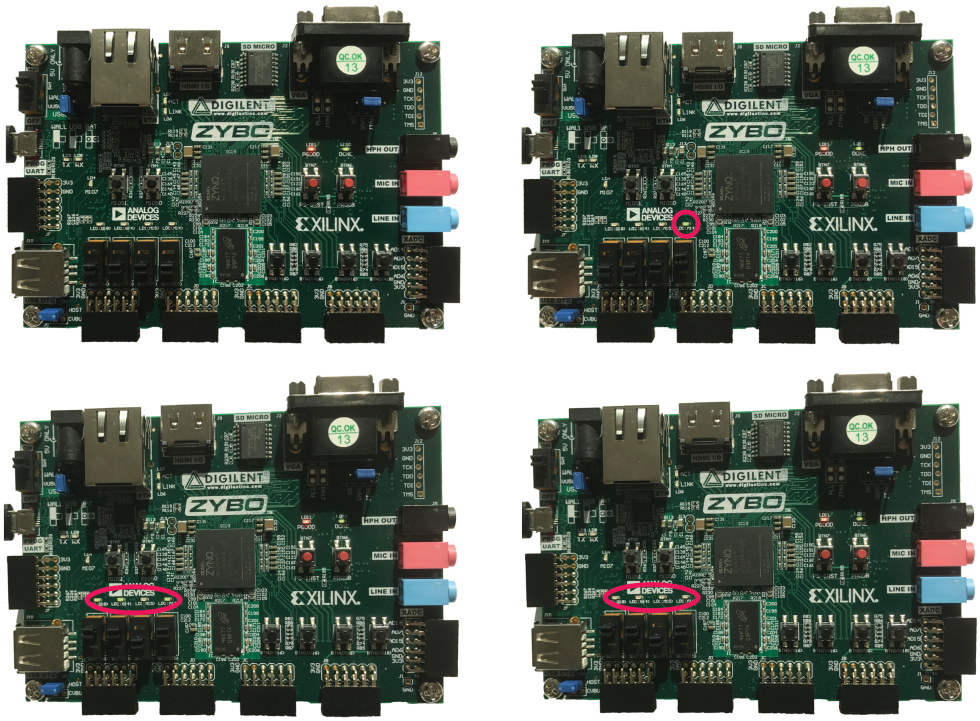

Figure 5: Calculating b0001 + b1101 on the FPGA board.

We proceeded to then test the rest of the numbers in our test bench on the FPGA. We found that they all worked correctly

6 Summary Statistics

We had some odd errors with finding the Summary Statistics for our experiment. Not only did we not get a timing value, but we also appear to have an error with the power. When comparing to other groups we noticed others had power coming from static, additionally the other tab of the power window had some red text about this dynamic power output being too high and a junction temperature being too high. We weren't sure what the issue is, we tried a couple different things to make the error stop, but it didn't cease. Either way, the FPGA worked correctly by our understanding. We are a worried about this difference though.

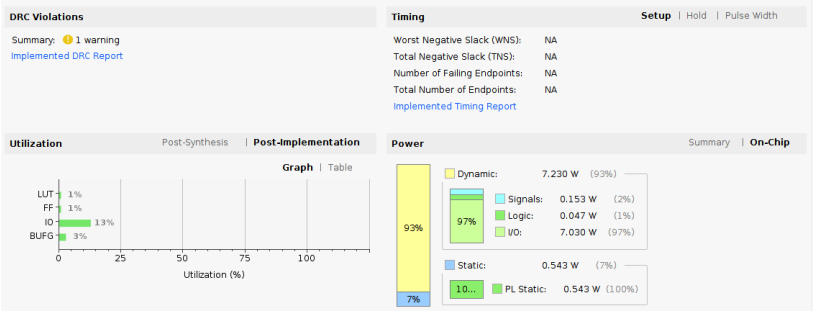

Figure 6: Summary Statistics