Implementing a 4-bit Adder

Anne Ku, Wilson Tang

September 2017

1 Introduction

We implemented a 4-bit adder in Verilog with some help from the 1-bit adder.v from Homework 2. With 4 bit vectors for a, b, and the sum, we had to make sure the right result from the first index influences the second, the right second index influences the third, etc.

2 Implementing the Adder in Verilog

```
// adder.v
include "fulladder.v"

module FullAdder4bit
(
  output[3:0] sum, // 2's complement sum of a and b
  output carryout, // Carry out of the summation of a and b
  output overflow, // True if the calculation resulted in an overflow
  input[3:0] a,   // First operand in 2's complement format
  input[3:0] b    // Second operand in 2's complement format
);

structuralFullAdder adder0 (sum[0], carryout0, a[0], b[0], 0);
structuralFullAdder adder1 (sum[1], carryout1, a[1], b[1], carryout0);
structuralFullAdder adder2 (sum[2], carryout2, a[2], b[2], carryout1);
structuralFullAdder adder3 (sum[3], carryout, a[3], b[3], carryout2);


// Overflow when carry in to the most significant column does not equal carry out
// carryout1 is the carryin of the 0th bit and carryout is the carryout of the 0th bit
xor getoverflow(overflow, carryout2, carryout);

endmodule
```

```
// adder.t.v, Adder Testbench
`timescale 1 ns / 1 ps
`include "adder.v"

module testFullAdder();
    reg[0:3] a;
    reg[0:3] b;
    wire[0:3] sum;
    wire overflow, carryout;

    FullAdder4bit adder1 (sum, carryout, overflow, a, b);
```

```
    initial begin
    // Your test code here
    $display("  Inputs   |   Output ");
    $display("a   b      | sum    carryout overflow ");
    a='b0000;b='b0000; #50
    $display("%b %b  | %b   %b       %b ", a, b, sum, carryout, overflow);
    \\ more testcases

    end
endmodule
```

3 Deciding on the Test Cases

We picked 18 different test cases to verify our adder.

Our first 16 cases rely on various combinations of the first two bits of each input a and b. With four different bits to change in a and b, we got a total of 16 combinations.

We decided to make both the 3rd and 2nd bits 0s in these 16 cases, because we represented each index of a and b the same way during the addition. Each index (in the addition) has an a bit, a b bit, and a carryin bit (for the rightmost bit the carryin would be 0). This abstraction allowed us to narrow the number of test cases to just 16.

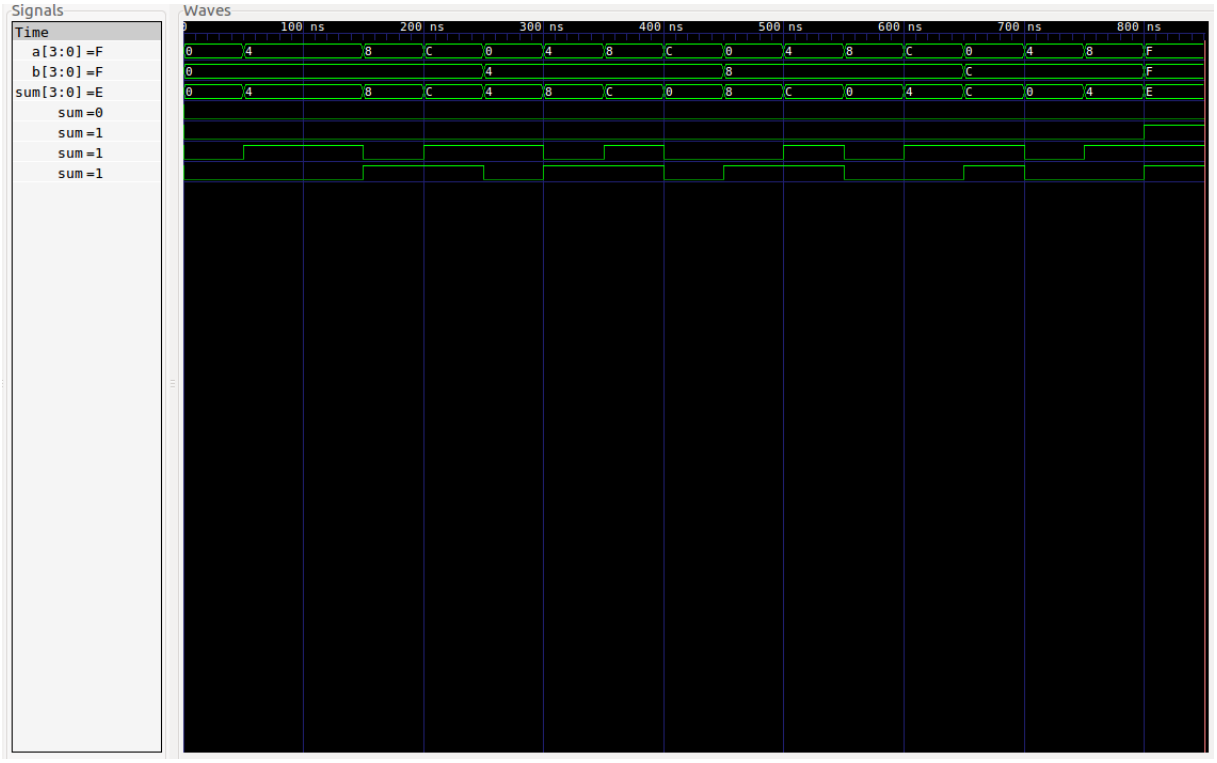
We added one more test case–when both a and b are b'1111. This case exemplifies the situation where we are guaranteed to have a carryout in the negative side. This case had the correct outcome (with carryout equal to 1 and overflow of 0).

This was our resulting truth table:

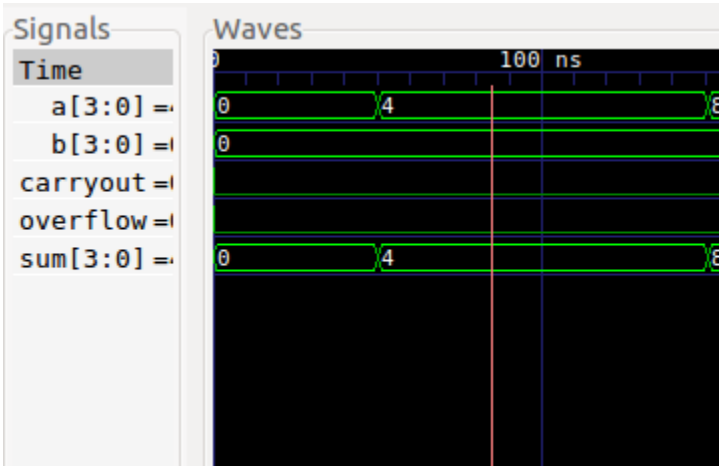
```
   Inputs   |   Output
a    b      | sum    carryout overflow
0000 0000   | 0000   0        0
    Testing Overflow
0100 0000   | 0100   0        0
0100 0000   | 0100   0        0
1000 0000   | 1000   0        0
1100 0000   | 1100   0        0

0000 0100   | 0100   0        0
0100 0100   | 1000   0        1
1000 0100   | 1100   0        0
1100 0100   | 0000   1        0

0000 1000   | 1000   0        0
0100 1000   | 1100   0        0
1000 1000   | 0000   1        1
1100 1000   | 0100   1        1

0000 1100   | 1100   0        0
0100 1100   | 0000   1        0
1000 1100   | 0100   1        1
1100 1100   | 1000   1        0
      Carryout Test
1111 1111   | 1110   1        0
```

4 Adder Waveforms

We took screenshots of our 4-bit adder waveforms to show that the outcome is reasonable. We selected 5 cases with combinations of various overflow, carryout, and sum outcomes. We did not find a worst case propagation delay. All of our 18 cases cases lined up appropriately as seen in the following image:

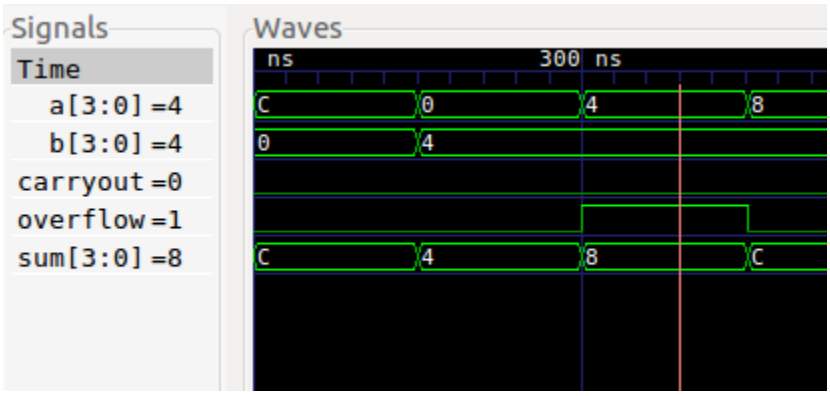


4.1 Result of 4+0



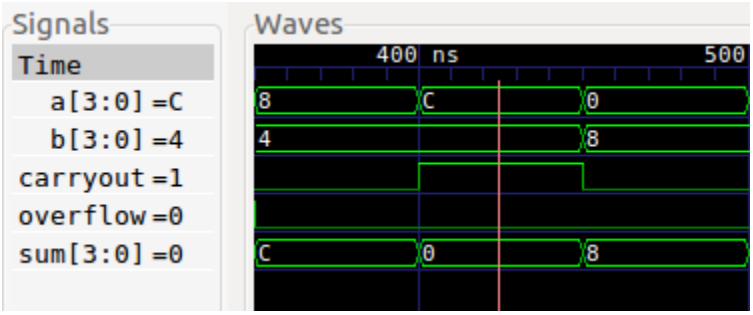
This snapshot shows the results of a = d'4 and b = d'0. The resulting sum is d'4 and there is no overflow and carryout.

4.2 Result of 4+4



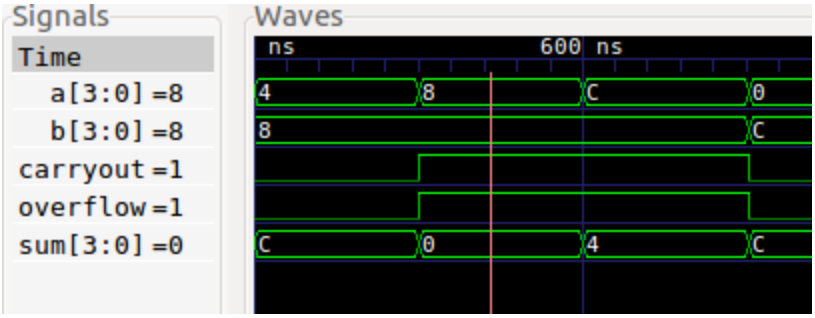
This snapshot shows the results of a = d'4 and b = d'4. The resulting sum is b'1000, or d'-8 and we have an overflow and no carryout. Although GTKWave got the output "right" (by showing +8), the overflow of 1 makes sense because the sum's signed bit (1) indicates a negative number from a sum of positive ones.

4.3 Result of C+4



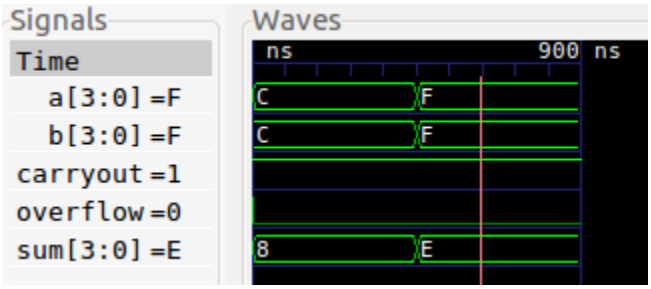
GTKWave represents b'1100 (d'-4) as C. The GTKWave-calculated sum as 0, which makes sense for this case because d'4 + d'-4 = 0. The extra carryout bit value makes sense because the ones at the most significant bit carry over, but there is no overflow because the carryout of the 3rd bit (1) and the carryout of the 4th (1) and the most significant bit (1) are the same. The equality of these bits mean the overflow is 0.

4.4 Result of 8+8



This snapshot shows the results of a = d'-8 or b'1000 and b = d'-8 or b'1000. The resulting sum is 0 but it should be b'10000. We also notice that overflow and carryout are set to 1. This sum is similar to section 4.3, but the overflow equals 1. This is right because the carryout value for b'1000 + b'1000 is 1, and the previous bit's carryout was 0. Because these bits have different values, the overflow is 1. If you look at the ideal sum (b'10000) you'll notice that the 4th bit is 0, whereas the 4th bit in b'1000 is 1.

4.5 Result of F+F



GTKWave represents b'1111 (d'-1) as F. The GTKWave-calculated sum as E (b'1110 or -2), which makes sense for this case because d'-1 + d'-1 = d'-2. Since the values b'1111 and b'1110 both have a 4th bit of 1, we know there is no overflow and the carryout is 1 because the carryout variable at the 4th bit is 1.

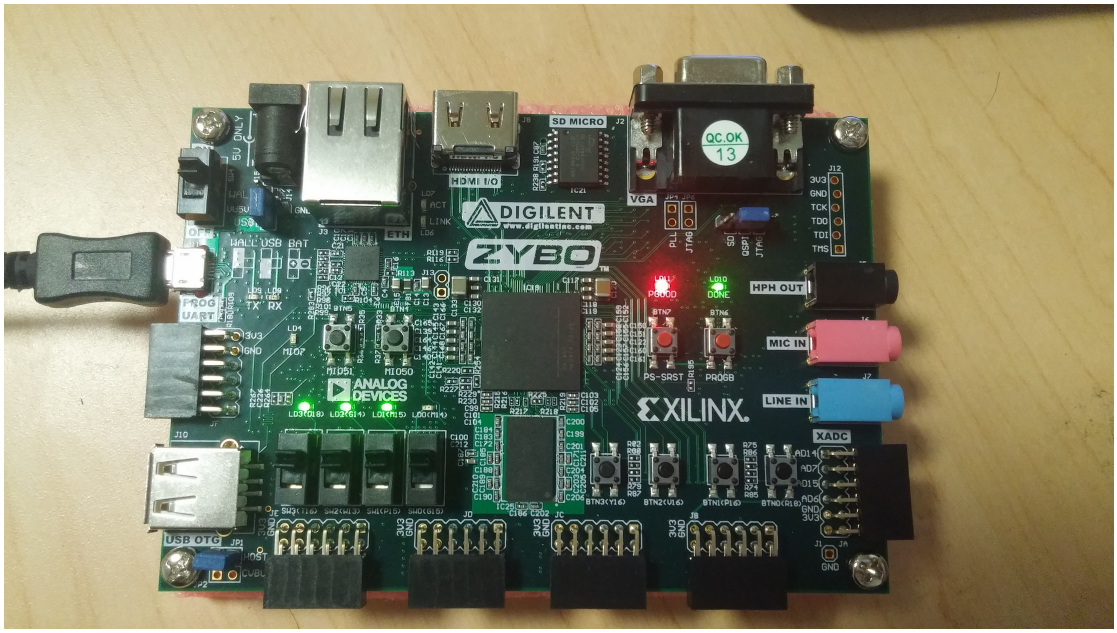
4.6 Test Bench Failures

We had no test bench failures that we found during our testing.

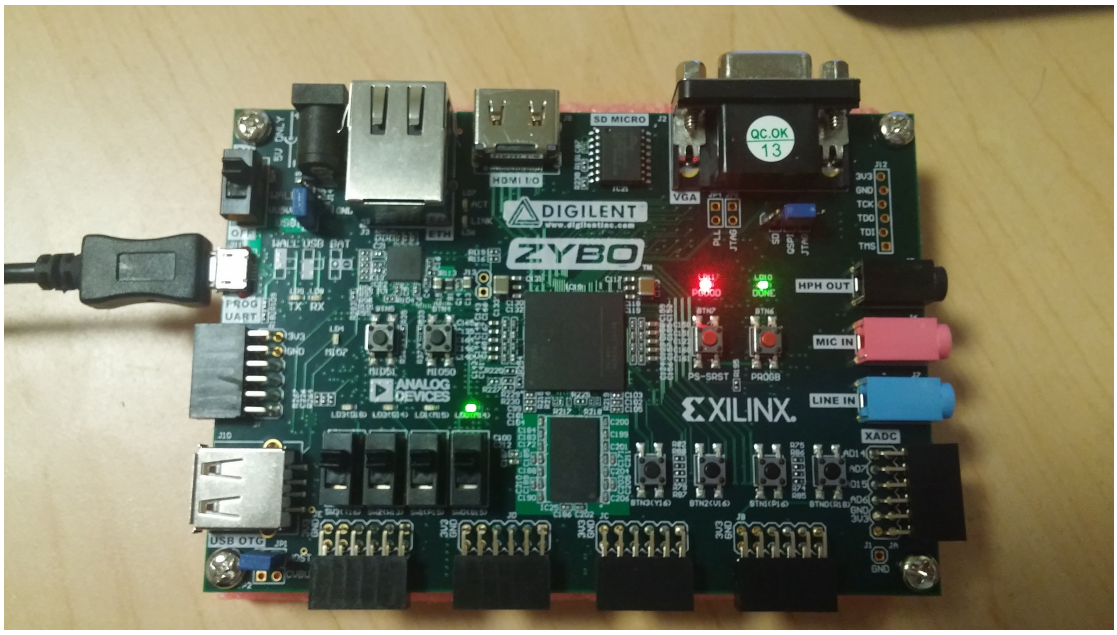
5 FPGA Testing

Finally, we implemented our verilog code using the FPGA and a wrapper provided by the teaching team. We tested the same 16 overflow cases and the carryout case as we used in iverilog because these cases utilize cover all of the overflow as well as show sum and carryout.

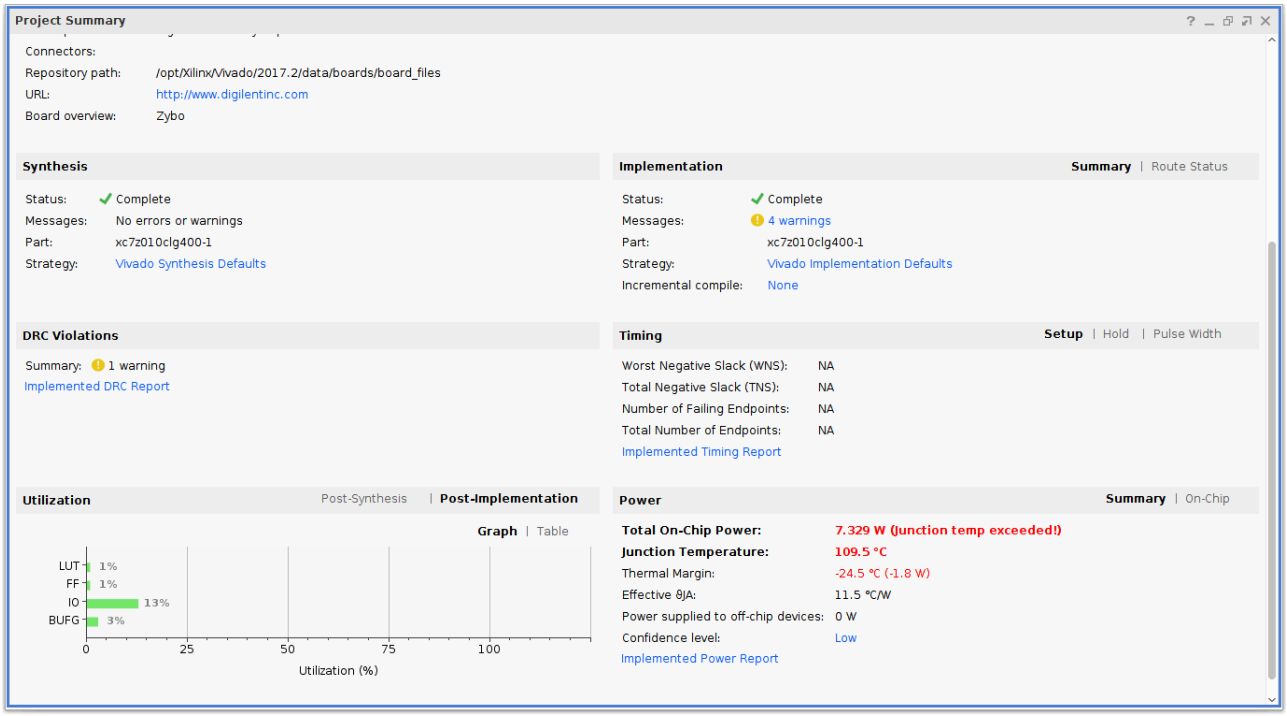
Example of a working test case in which we calculated the Carryout Test (1111 + 1111): Here we can see the sum which should be (1)1110



Here we can see that there is no overflow since led1 is off and there is a carryout since led0 is on:



6 Summary Statistics



7 Areas of Improvement

There were times when we couldn't display the values we wanted. Instead of printing the exact value like -4 or -1 or 0, we end up getting letter names like C, F, or E. If we were to study more about GTKWave or verilog, we may be able to display the right value for easy interpreting.

We did not find a worst-case propagation delay because everything seemed to line up perfectly.