# Comparch Lab 0 : Full Adder on FPGA

Lisa Hachmann & Anisha Nakagawa

September 28, 2016

## 1 Introduction

A four-bit full adder is made up of four individual adders. In this lab, we created a four-bit full adder in verilog, made from four connected adder modules. The four-bit adder takes two 4-bit inputs (numbers in 2's complement) and outputs a 4-bit sum, the one-bit carryout, and a one-bit value for overflow.

The schematic for the connections between the adders is shown in 1. Each of the adders computes the sum for one bit location: for example, the 0th adder calculates the sum of the 0th bit in input A and the 0th bit in input B, and returns the 0th bit in the overall sum. Each adder module has its carryout bit connected to carryin bit of the subsequent module. The first adder module has a carryin bit equal to zero, so that it has the behavior of a half-adder. The final sum is a bit string containing the results from each of the adder calculations.

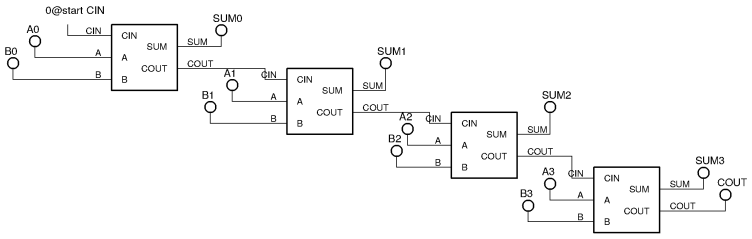



Figure 1: Circuit diagram for connecting 4 full adders to create a four-bit adder.

After calculating the sum, we then checked to see if there was overflow. The overflow has a value of 1 if there has been overflow according to 2's complement - when the sum is out of the range of values represented by 2's complement. This condition can only occur when adding two numbers of the same sign. Furthermore, overflow only occurs when the value of the left-most bit of the sum is different from the left-most bits of A and B. The conditions for overflow are described in Table 1. Based on the table, we wrote the equation for overflow to be: $Overflow = A_3 B_3 \; S_3 + \; A_3 \; B_3 S_3$, where the subscripts indicate the bit position. The circuit diagram for calculating the overflow is given in 2. Overflow can also be calculated by comparing the carryin and carryout of the most significant bit. If the carryin does not equal the carryout, then there is overflow. Both these methods of calculating overflow lead to the same result.

| A left-most bit | B left-most bit | Sum left-most bit | Overflow |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 |

Table 1: Truth table for overflow cases

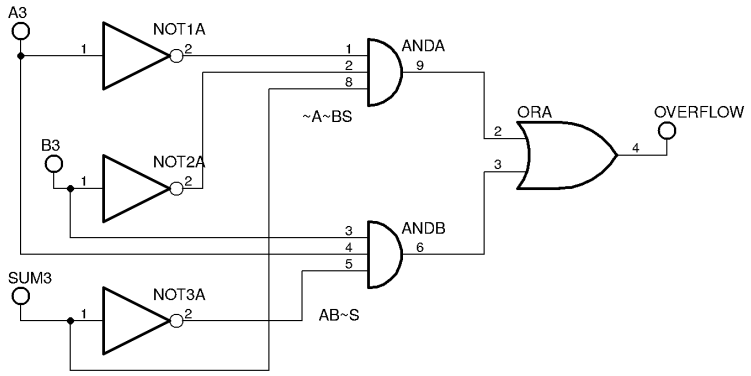



Figure 2: Circuit diagram for calculating whether there is overflow, based on the inputs and sum.

## 2 Waveform results

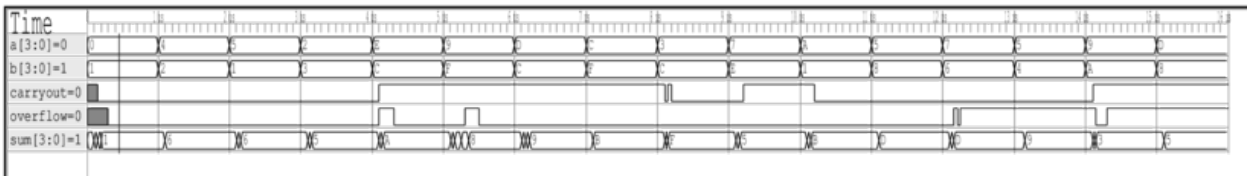



Figure 3: Waveform of 4 bit full adder

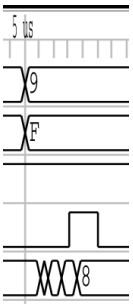



Figure 4: Zoom in at the 5-6 µsecond mark, to show the worst delay

The worst delay measured was circa 500 nS, as shown in figure 4. We compared this to the calculated worst case delay of 550 nS and found it to be very close. It makes sense that all the observed delays were less than

the calculated worst case delay. We calculated the worst case delay using figures 1 and 2 and the results can be seen in tables 2 and 3. Each gate had 50 nS delay, as programmed in the verilog test bench. We found that the general case for an N-bit adder of our design topology was $2N + 3$.Therefore, for our 4 bit adder, there were 11 delays at 50 nS each and totalling 550 nS (0.55 µsecond) propogation delay. For clarity, please note that the sum, a and b inputs of figure 2 are the most significant bit of each input, so the entire 4 bit adder must complete before the overflow circuit can set.
Please note that the waveform results of the 4 bit adder including all inside circuitry can be found in figure 7.

| | A | B | Cin |
|---|---|---|---|
| Sum | 2 | 2 | 2 |
| Cout | 2 | 2 | 2 |

Table 2: Propagation delay for the 4 bit adder circuitry

| | A | B | Sum |
|---|---|---|---|
| Overflow | 3 | 3 | 3 |

Table 3: Propagation delay for the overflow circuitry

# 3 Test cases

We chose test cases to cover the range of possible types of calculates. We made sure the test cases gave examples of the following four categories: adding two positive numbers, adding two negative numbers, adding one positive number and one negative number, and adding two numbers that specifically result in overflow. For the case where there should be overflow, we tested both adding two positive numbers and adding two negative numbers. The full set of test cases is included in Table 4.

| A | B | Cout | Sum | Overflow | Expected Output |
|---|---|---|---|---|---|
| 0000 | 0001 | 0 | 0001 | 0 | 0, 0001, 0 |
| 0100 | 0010 | 0 | 0110 | 0 | 0, 0110, 0 |
| 0101 | 0001 | 0 | 0110 | 0 | 0, 0110, 0 |
| 0010 | 0011 | 0 | 0101 | 0 | 0, 0101, 0 |
| 1110 | 1100 | 1 | 1010 | 0 | 1, 1010, 0 |
| 1001 | 1111 | 1 | 1000 | 0 | 1, 1000, 0 |
| 1101 | 1100 | 1 | 1001 | 0 | 1, 1001, 0 |
| 1100 | 1111 | 1 | 1011 | 0 | 1, 1011, 0 |
| 0011 | 1100 | 0 | 1111 | 0 | 0, 1111, 0 |
| 0111 | 1110 | 1 | 0101 | 0 | 1, 0101, 0 |
| 1010 | 0001 | 0 | 1011 | 0 | 0, 1011, 0 |
| 0101 | 1000 | 0 | 1101 | 0 | 0, 1101, 0 |
| 0111 | 0110 | 0 | 1101 | 1 | 0, 1101, 1 |
| 0101 | 0100 | 0 | 1001 | 1 | 0, 1001, 1 |
| 1001 | 1010 | 1 | 0011 | 1 | 1, 0011, 1 |
| 1101 | 1000 | 1 | 0101 | 1 | 1, 0101, 1 |

Table 4: Test cases table

### 3.1 Test case failures

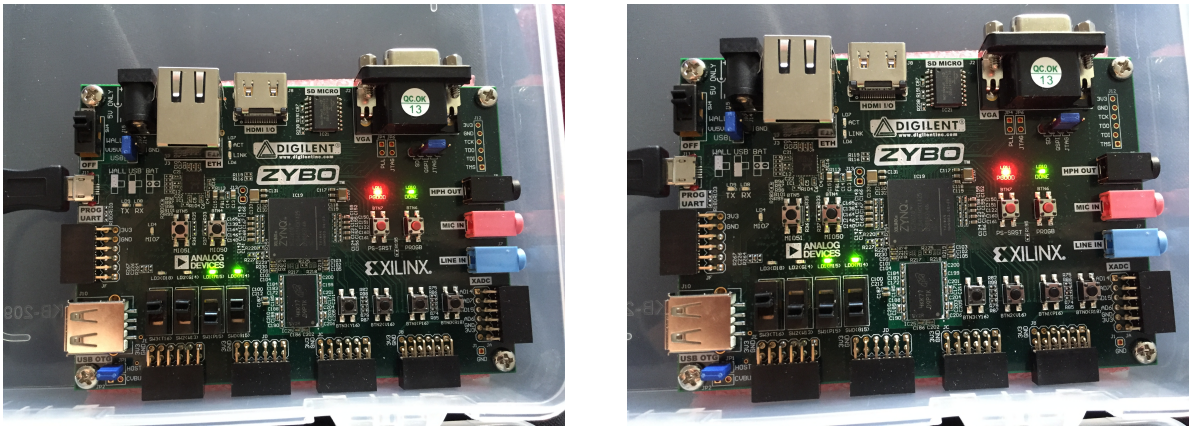
During the process of creating the adder, there were a couple ways that our adder failed the test cases, which led us to modify the design of the adder.

- The left sum bits did not change based on the carryin from the previous adders.
  We had been trying to use the same variable for each of the carryouts from the adders, and tried to re-write it each time. However, verilog does not work procedurally, so it does not work to override the value of a variable multiple times in the same module. Instead, we changed the design of the module to have a separate wire for each of the carryouts between the adders, which fixed the problem. It also means that each subsequent adder must wait until the value of the carryin wire is set. The biggest take-away is to remember that the variables in verilog are actually wires connecting components, so they behave differently than variables in other programming languages.

- The overflow was always 0, even when it should have been 1.
  Initially we had tried to set the value of the overflow using conditional if-statements to determine whether overflow would happen. However, we were unable to make this work, and the overflow always returned 0. We think it may not have worked because the values at the bit positions were not properly compared using standard operation of "==". Since we were unable to find sufficient documentation about if-statements with this use case, we instead decided to calculate overflow using logic gates instead of conditional statements. An explanation of the final overflow calculations is in the introduction.

### 3.2 Summary of Testing

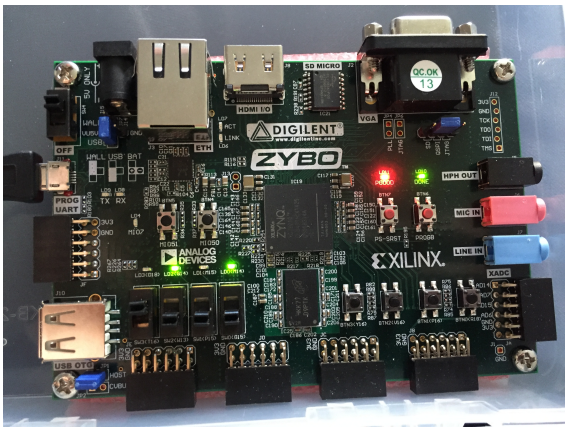
When testing the FPGA board, we used the same test cases as we used for the verilog tests (in Table 4). These test cases cover the four main categories of adding two positive numbers, two negative numbers, one positive and one negative, and cases with overflow. Our FPGA board correctly calculated each of these cases relative to the expected output, which was also the same as the verilog script output. Photographs of one of the test cases is shown in figures 5 and 6. Note that the switches in figure 5 denote the 4 bit binary number (up = 1, down = 0) while the leds in figure 6 a denote the output of the sum and the carryout and overflow.



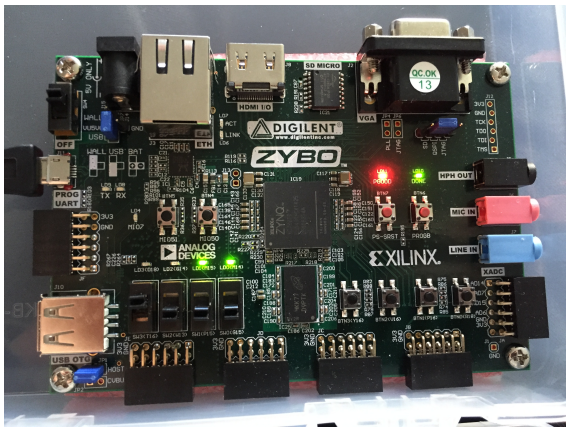

(a) Input A: 1101

(b) Input B: 1000

Figure 5: Inputs to the 4 bit full adder on the FPGA board, given by the positions of the switches.




(a) Resulting sum of inputs A and B: 0101




(b) Overflow and carryout for the sum, both high (=1)

Figure 6: Outputs to the 4 bit full adder on the FPGA board, shown with the LEDs.

### 3.3 Summary Statistics

After uploading the module to the FPGA board, we can analyze the summary statistics. A table containing information about the resources used is in Table 5. Because this is a relatively simple device, it only uses a few resources out of the number of available resources.

| Resource | Utilization | Available | Utilization% |
|---|---|---|---|
| LUT | 7 | 17600 | 0 |
| FF | 9 | 35200 | 0 |
| IO | 13 | 100 | 0 |
| BUFG | 1 | 32 | 0 |

Table 5: Resource use for the FPGA

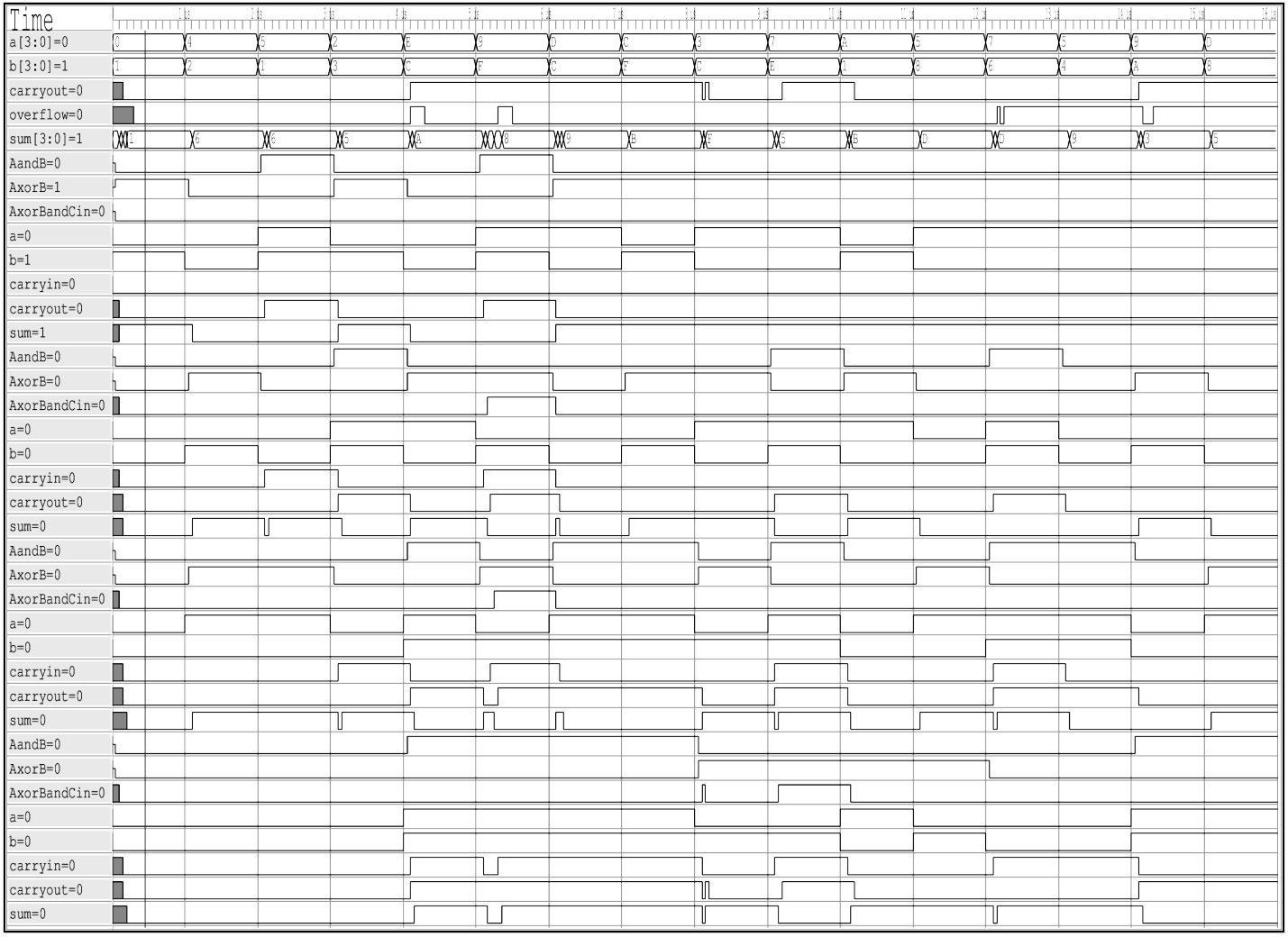



Figure 7: Waveform results of the 4-bit full adder, including signals of each adder