CSCE 22104

Lab Report

Brent Marcus Orlina

ID: 011019116

Lab Section 001

Lab 2

Introduction

This lab's goal was to create a full-adder component and a 4-bit ripple carry adder component. The full-adder should be able to add two bits along with an extra bit, `cin` in the case that there is an incoming carry bit. The output should be the sum of the input bits and another bit, `cin` to signal that there is a carry onto the next digit in the sum.

The 4-bit ripple carry adder should be able to add two 4-bit numbers. The output should be the sum of the two 4-bit numbers and another bit, `cout`, to signal that there was a carry in the sum of the last digit. The ripple carry adder also has an extra input bit, `cin` in the case that there is an incoming carry bit from some other adder.

Approach

In the addition of two bits A and B, all the possibilities of A and B can feasibly be considered. Its truth table looks like

A	B	$A+B$
0	0	0
0	1	1
1	0	1
1	1	0

The truth table is identical to the XOR operation in boolean algebra, therefore the XOR operation can model bit-addition. The XOR operation is also associative and thus is able to add three bits together without worrying about order. In the end, the sum of the three input bits of the full-adder can be expressed as

$$S = A \oplus B \oplus C_{in} \tag{1}$$

where A and B are the two input bits and cin is the extra incoming carry bit.

In addition, the concept of a "carry" is to signal that the sum of a digit is too large to fit in a single digit and thus must be "carried" and added onto the next digit. In bit addition, the sum of two 1's is too large for a single digit to represent and thus must be carried onto the next digit. Therefore, checking if there are at least two 1's being added is enough to know if there should be a carry onto the next digit. In boolean algebra, it can be expressed as

$$C_{out} = (A \wedge B) \vee (A \wedge C_{in}) \vee (B \wedge C_{in}) \tag{2}$$

The full-adder was built by following the two previous equations. A `GateXOR`, `GateAND`, and `GateOR` were made so that the full-adder was built structurally. The `GateXOR` and `GateOR` was made such that it had three inputs and did the respective binary operation of each input in order. E.g. the `GateOR` had the output of `A xor B xor C` where `A`, `B`, and `C` are the three inputs.

The 4-bit ripple carry adder was built structurally by using four full-adders. Each bit of the input bits is connected to a full-adder. E.g. the first bit of inputs `A` and `B` are connected to the first full-adder, the second bits are connected to the second full-adder, and so on. The first full-adder's `cout` is connected to the second full-adder's `cin`. The second full-adder's `cout` is connected to the third full-adder's `cout`, and so on for the third and fourth full-adder. The ripple carry adder's input bit `cin` is connected to the first full-adder's `cin` and the last full-adder's `cout` is connected to the ripple carry adder's output bit `cout`.

Experimentation

The components' behaviors were tested using hand-crafted testbenches. The testbench for the full-adder component tested all possible combinations of the input bits. The full-adder worked as expected and follows the equations (1) and (2).

The testbench for the 4-bit ripple carry adder did not test for all possibilities of the inputs. However, it is sufficient enough as it tests addition with `cin` active and inactive and overflowing addition with `cin` active and inactive. The 4-bit ripple carry adder worked as expected, displaying the correct sum of the input bits. Inputs in which the sum does not fit in four bits correctly activates the output bit `cout` and the resulting sum loops back around.

Results & Discussion

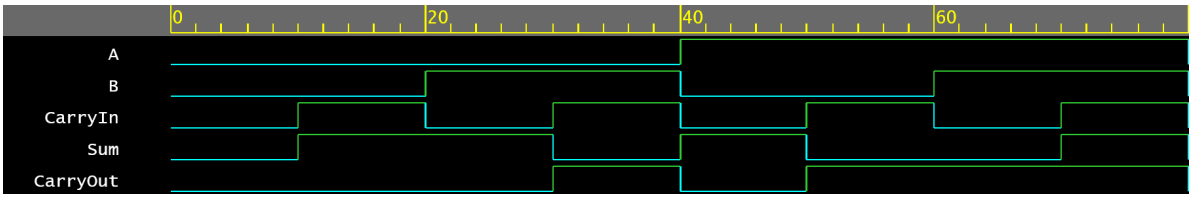

Figure 1: The waveform for the full-adder component implementation.

The full-adder component works as expected. In figure 1, it shows that when only one of the input bits are active, the `Sum` is active and `CarrOut` is inactive. When only two of the input bits are active, `CarryOut` is active and `Sum` is inactive. When all three input bits are active, both `Sum` and `CarryOut` are active. This follows both equations (1) and (2). The testbench for this component is sufficient since it has tested all possible combinations of the input bits.

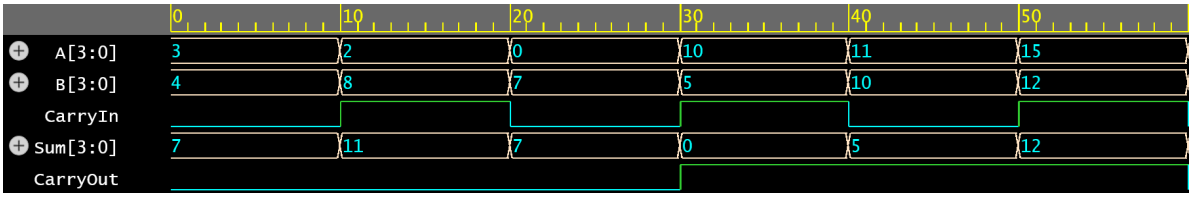

Figure 2: The waveform for the 4-bit ripple carry adder implementation.

The 4-bit ripple carry adder also works as expected. In figure 2, it shows the addition of `A` and `B` with the result in `Sum`. From 0ns to 10ns, it shows $3 + 4$ correctly showing the sum of 7 with the `CarryOut` inactive. From 10ns to 20ns, it shows the $2 + 8$ with the input bit `CarryIn` active. It correctly shows the sum of 11, with `CarryOut` inactive, and not 10 since the `CarryIn` input is active, thus adding an extra 1. The cases from 30ns to 60ns shows cases of overflow with each one correctly showing the results with `CarryOut` active. These cases overflow because there aren't enough bits to represent the sum of the two numbers. For example, the 5-bit representation of the case from 40ns to 50ns would be

$$\texttt{01011} + \texttt{01010} = \texttt{10101}$$

correctly, resulting in the binary form of 21. However, the adder only has four bits, so the leftmost bit is "thrown away", leaving only `0101`, or 5 in decimal form. This has the same effect of doing

modular arithmetic in mod 2^4, or mod 16.

$$\begin{aligned} 11 + 10 &\equiv 21 \pmod{16} \\ &\equiv 5 \pmod{16} \end{aligned}$$

Although the testbench for the 4-bit ripple carry adder did not test for all possible combinations of the inputs, it is sufficient enough since it tests for addition with `CarryIn` both active and inactive and overflowing addition with `CarryIn` both active and inactive.

Conclusions

Both the full-adder component and the 4-bit ripple carry adder displayed the correct behavior during testing. The knowledge learned from this lab was learning how a full-adder and 4-bit ripple carry adder works and how to construct them structurally, using the full-adder component to make the 4-bit ripple carry adder. The skills practiced in this lab was learning how to build components structurally and create sufficient testbenches.