ECE 272 Lab 2
Fall 2018

# Adders on an FPGA

Phi Luu

October 8th, 2018
Grading TA: Edgar Perez
Lab Partner: Benjamin Geyer

# 1 Introduction

This lab focuses on the logic level of computer arithmetic and a critical component of processors—called the *Arithmetic Logic Unit* (ALU)—that performs arithmetic operations. This lab shows how to implement a system of adders to perform and demonstrate additions of two 4-bit binary numbers using combinational logic design.

There are three main ways of implementing multi-bit adders from combinational logic: ripple-carry, carry-lookahead, and carry-save. There is a tradeoff between speed and size of the three methods. Ripple-carry adders are the slowest but smallest, and carry-save adders are the fastest but biggest. The logic designer can choose between the three depending on the situation. Because the ripple-carry adder is easy to understand and implement, it will be the main focus on this lab.

We will also learn how to use Quartus Prime to design a ripple-carry adder from scratch (logic gates and wires), make the adder itself as a logic symbol, and then use that symbol to perform 4-bit binary addition operations.

# 2 Design

A ripple-carry takes in three inputs—**operand1**, **operand2**, and the **carry-in**—and produces two outputs, the **sum** and the **carry-out**, as sketched in Figure 1 below:

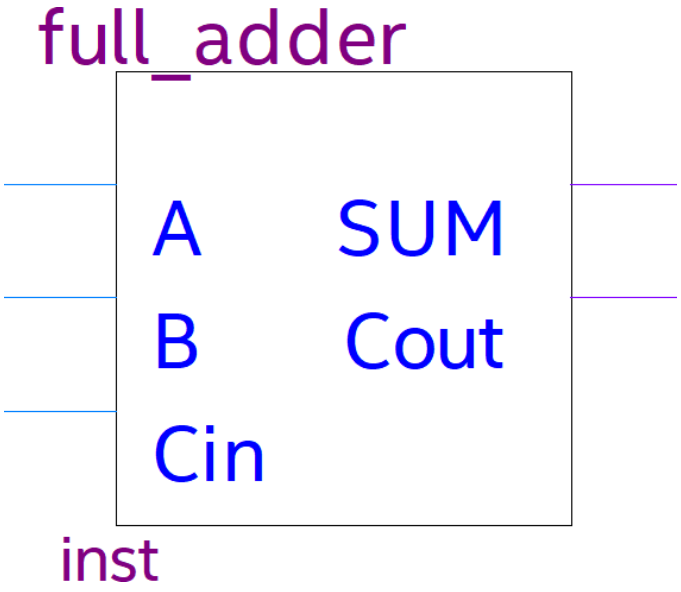


Figure 1: The pin layout of a full ripple-carry adder for $A + B$

The carry-in works similarly to when we carry the numbers when adding decimals together. If the value of the carry-in is 1, then the addition on the more significant column will increase by 1. If the next column's addition result is already 1 before adding the carry-in, then its value will circle back to 0 and continue carrying the 1 to the more significant column and so on.

The block diagram showing the structure of the implementation is as follows:

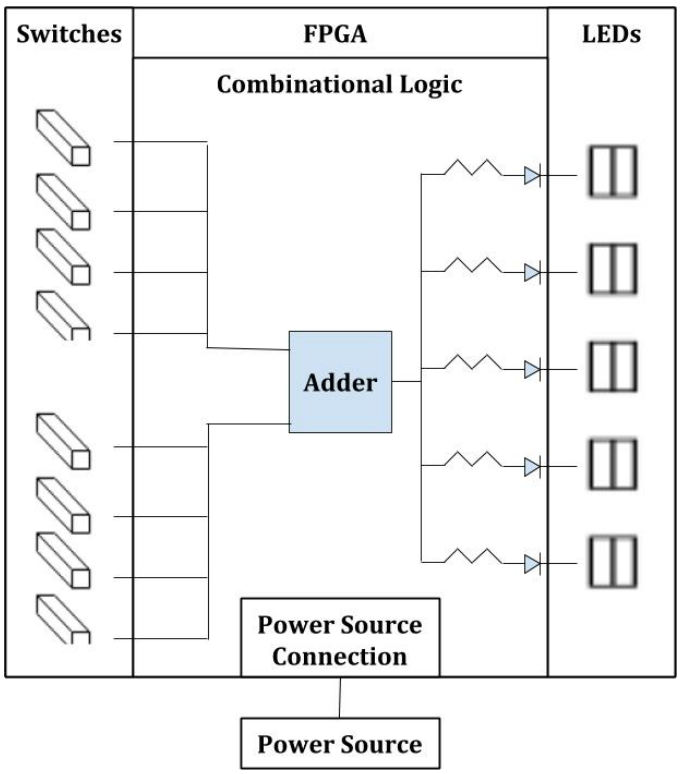


Figure 2: The block diagram showing the connections between the switches, the FPGA, and the LEDs in a 4-bit binary adder implementation

Since the circuit takes in two 4-bit binary numbers as operands, there are four pairs of 2-bit inputs—one bit from the first operand and the other from the second operand. Each ripple-carry adder will take in one of the pairs described earlier and return the carry-out with the sum. The adder that ties to the addition of the **least significant** bits of the two binary numbers will have **no carry-in value** ($C_{in0} = 0$). Similarly, the adder that ties to the addition of the **most significant** bits of the two binary numbers will have its **carry-out connected to the output**, as in some cases the result can be a 5-bit binary number. Finally, the remaining adders will connect to each other in the same way as follows: **the carry-out of the less significant adders connects to the carry-in of the more significant adders**.

A picture is worth a thousand words. Therefore, the schematic of the internal structure of a ripple-carry adder and the schematic of the 4-bit binary addition circuit are shown in Figures 3 and 4, respectively.

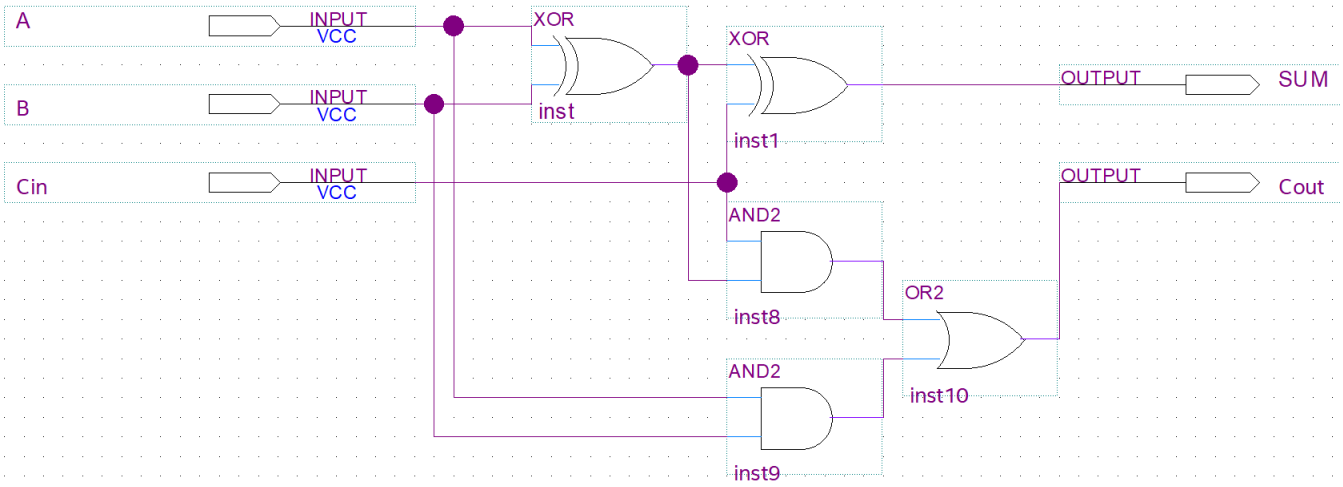


Figure 3: The internal logic-level structure of a full ripple-carry adder

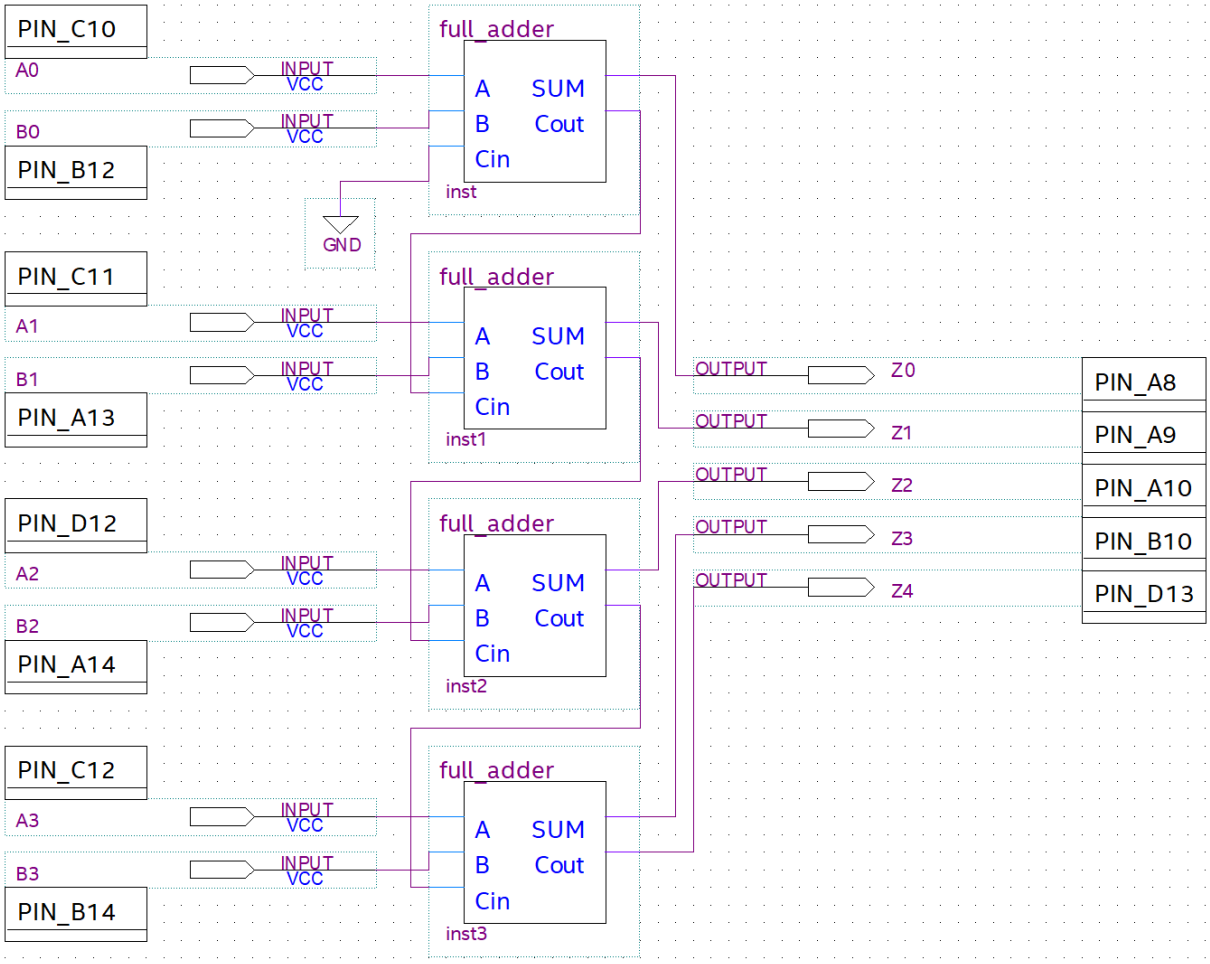


Figure 4: The schematic of the main circuit

Using Figure 3, the sum $SUM$ and the carry-out $C_{out}$ of a ripple-adder with operand $A$, operand $B$, and carry-in $C_{in}$ are expressed in the following sum-of-products forms in Equations 1 and 2:

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

$$C_{out} = AB + AC_{in} \oplus BC_{in} \tag{2}$$

A full-adder truth table used when adding two binary digit is shown in Table 1. Extending this table to construct a partial 4-bit adder truth table (Table 2), we obtain the following results:

| **Operand1** | + | **Operand2** | + | **Carry In** | = | **Value (2-bit Binary)** | **Value (Unsigned Decimal)** |
|---|---|---|---|---|---|---|---|
| 0b0 | + | 0b0 | + | 0b0 | = | 0b00 | 0 |
| 0b0 | + | 0b0 | + | 0b1 | = | 0b01 | 1 |
| 0b0 | + | 0b1 | + | 0b0 | = | 0b01 | 1 |
| 0b0 | + | 0b1 | + | 0b1 | = | 0b10 | 2 |
| 0b1 | + | 0b0 | + | 0b0 | = | 0b01 | 1 |
| 0b1 | + | 0b0 | + | 0b1 | = | 0b10 | 2 |
| 0b1 | + | 0b1 | + | 0b0 | = | 0b10 | 2 |
| 0b1 | + | 0b1 | + | 0b1 | = | 0b11 | 3 |

Table 1: A full-adder truth table

| **Operand1** | + | **Operand2** | = | **Value (5-bit Binary)** | **Value (Unsigned Decimal)** | **Value (Hexadecimal)** |
|---|---|---|---|---|---|---|
| 0b0000 | + | 0b0000 | = | 0b00000 | 0 | 0x00 |
| 0b0000 | + | 0b0001 | = | 0b00001 | 1 | 0x01 |
| 0b0000 | + | 0b0010 | = | 0b00010 | 2 | 0x02 |
| 0b0000 | + | 0b0011 | = | 0b00011 | 3 | 0x03 |
| 0b0000 | + | 0b0100 | = | 0b00100 | 4 | 0x04 |
| 0b0000 | + | 0b0101 | = | 0b00101 | 5 | 0x05 |
| 0b1000 | + | 0b0000 | = | 0b01000 | 8 | 0x08 |
| 0b1000 | + | 0b0001 | = | 0b01001 | 9 | 0x09 |
| 0b1000 | + | 0b0010 | = | 0b01010 | 10 | 0x0A |
| 0b1000 | + | 0b0011 | = | 0b01011 | 11 | 0x0B |
| 0b1111 | + | 0b0011 | = | 0b10010 | 18 | 0x12 |
| 0b1111 | + | 0b1000 | = | 0b10111 | 23 | 0x17 |
| 0b1111 | + | 0b1010 | = | 0b11001 | 25 | 0x19 |
| 0b1111 | + | 0b1011 | = | 0b11010 | 26 | 0x1A |

Table 2: A partial 4-bit adder truth table

# 3 Results

Before uploading the project to the real FPGA, we created a simulation waveform of the addition. Specifically, from Table 2 we picked **0b0000 + 0b0000** (first row), **0b0000 + 0b0100** (fifth row), **0b1000 + 0b0011** (tenth row), and **0b1111 + 0b1010** (thirteenth row) and tested them in the simulation. Figure 5 shows the simulation result:

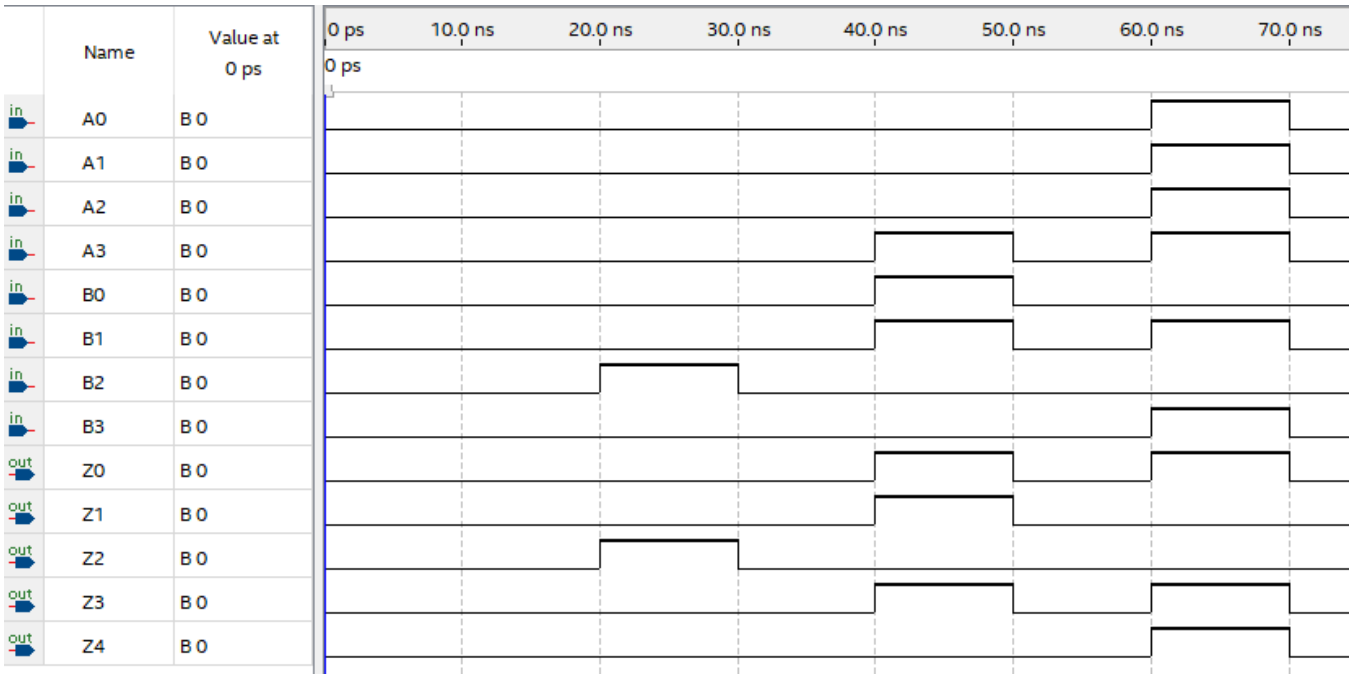


Figure 5: Simulation results after testing first (0 - 10ns), fifth (20ns - 30ns), tenth (40ns - 50ns), and thirteenth (60ns - 70ns) rows of Table 2

The simulation yielded **0b0000 + 0b0000 = 0b00000**, **0b0000 + 0b0100 = 0b00100**, **0b1000 + 0b0011 = 0b01011**, **0b1111 + 0b1010 = 0b11001**, which matched with the corresponding results calculated from Table 2. We then moved on to real-board testing and also got the same results. Based on the results, we hence believed that our implementations of the 4-bit binary adders were correct.

# 4 Experiment Notes

## Reflection

This lab brought us from simple schematic design of lab 1 to a more complex design, where we had to define our own ripple-carry adders from scratch (logic gates and wires) and then used the adders in our main circuit. This was quite a big change, but we were able to grasp the idea of reusing sub-circuits and successfully made the grand system work. We are looking forward to more challenging labs in the future.

## Study Questions

1. Explain how you would convert your 4-bit adder to a 4-bit adder/subtractor.

   Subtraction in decimal system is the same as addition with the opposite of the second operand: $A_{10} - B_{10} = A_{10} + (-B_{10})$. Similarly, we can subtract a binary number $B_2$ from $A_2$ by adding $A_2$ with the two's complement of $B_2$: $A_2 - B_2 = A_2 + (-B_2)$. Therefore, add an inverter to every bit of $B$ before it goes to a logic gate, and we will have a 4-bit subtractor.

2. In your own words, explain what pull resistors do in the FPGA.

   The pull-up and pull-down setups of a circuit should have a switch in it. For example, a pull-up resistor setup means a resistor ties the input pin of the FPGA to $V_{CC}$ and a switch connecting the circuit to GND. Similarly, a pull-down resistor setup means a resistor ties the input pin of the FPGA to GND and a switch connecting the circuit to $V_{CC}$. See Figure 6 below

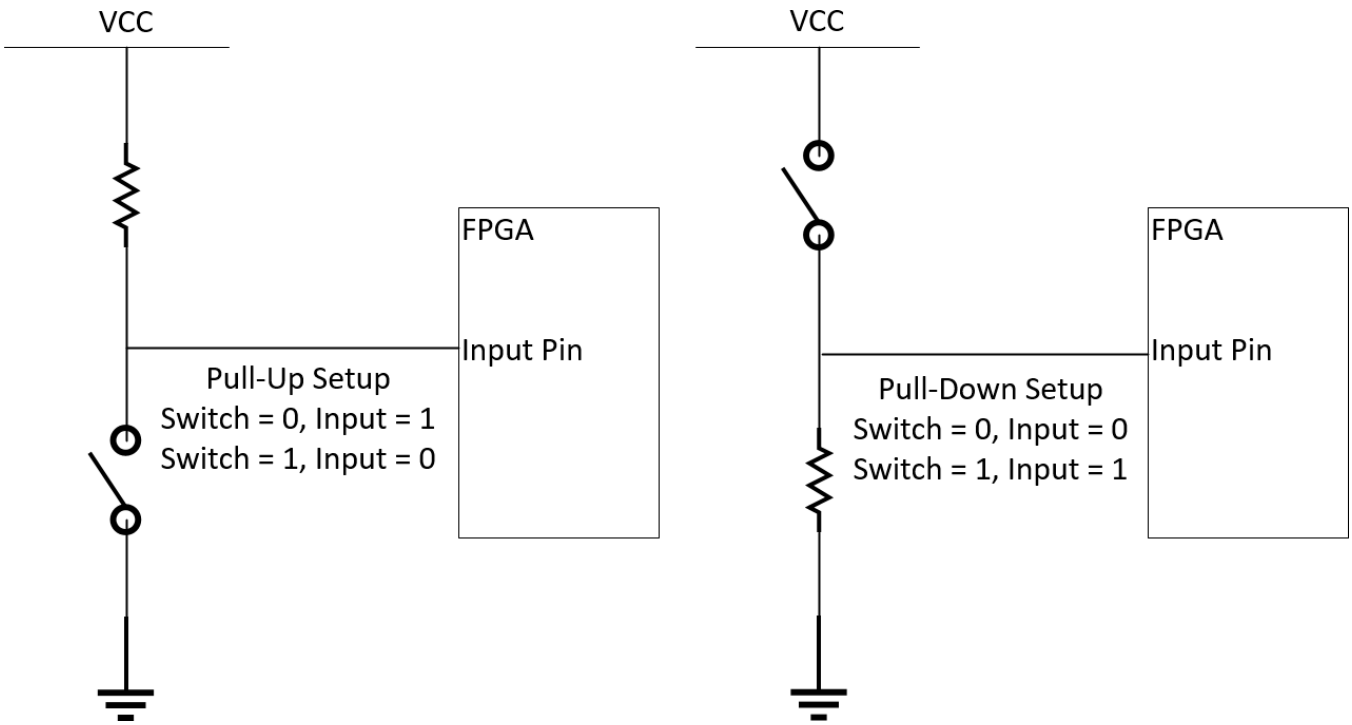


Figure 6: Pull-up and pull-down resistor setups and their truth tables

   The outcome is different between the two setups. However, a significant thing here is that by using pull resistors, we effectively make sure that the input *always* receives *either 0 or 1—nothing in between and thus no floating value*. The resistors significantly reduce the noise the input pin of the FPGA would take and guarantee the FPGA would never take a floating value as an input.

3. Explain your selection for the pullmode for the pin connected to the least-significant full-adder's carry-in.

   We pulled least-significant full-adder's carry-in to GND because when we add the very first bits of the two binary numbers together, there is always no initial carry-in (or $C_{in0} = 0$).

# Appendix

No appendix is available in this lab.