Computer Architecture Lab 0 : Full 4-Bit Adder

Yoonyoung Cho, Haozheng Du

September 29 2016

1 Introduction

In this lab, we constructed a signed 4-bit full adder (in 2's complement system) with verilog and ran simulations accordingly, verifying our implementation against test cases. Then, we converted our design with Vivado and loaded on the ZYBO FPGA Board to see the code in action.

2 Simulation

2.1 One-Bit Full Adder

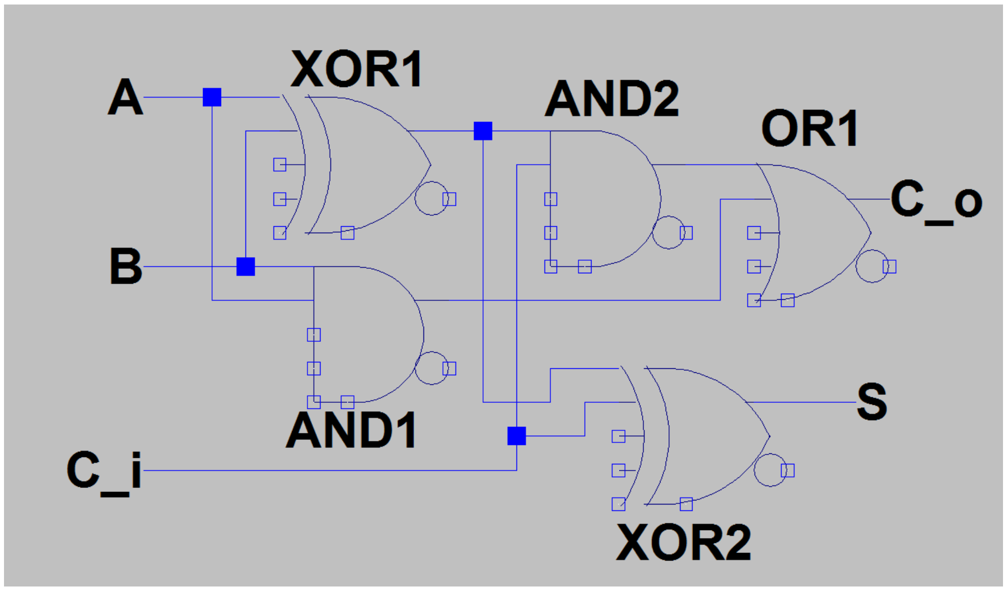


Figure 1: One-Bit Full Adder; LTSpice

The above circuit directly translates to the verilog code below:

Listing 1: "One-Bit Full Adder; Verilog"

```
module FullAdder1bit
(
    output sum,
    output carryout ,
    input a,
    input b,
    input carryin
);
    wire axorb;
    wire aandb;
    wire axorb_and_c;
    `XOR xorgate1(axorb,a,b);
    `AND andgate1(aandb,a,b);
    `AND andgate2(axorb_and_c,carryin,axorb);
    `XOR xorgate2(sum,carryin,axorb);
    `OR orgate(carryout,axorb_and_c,aandb);
endmodule
```

Table 1: Sequential Demonstration of the above circuit

A	**B**	C_i	$A \oplus B$	$A \oplus B \oplus C_i$	S	AB	$(A \oplus B)C_i$	AB + $(A \oplus B)C_i$	C_o
0	0	0	0	0	0	0	0	0	0
0	0	1	0	1	1	0	0	0	0
0	1	0	1	1	1	0	0	0	0
0	1	1	1	0	0	0	1	1	1
1	0	0	1	1	1	0	0	0	0
1	0	1	1	0	0	0	1	1	1
1	1	0	0	0	0	1	0	1	1
1	1	1	0	1	1	1	0	1	1

The one-bit adder acts as a fundamental building block for the final four-bit adder. Essentially, the circuit outputs the "sum" bit as the boolean representation of whether or not there exist odd numbers of ones; as for the carry-out, the circuit simply reuses the precomputed xor values and the carry-in to compute the cases in which AC_i or BC_i are true.

2.2 Four-Bit Full Adder

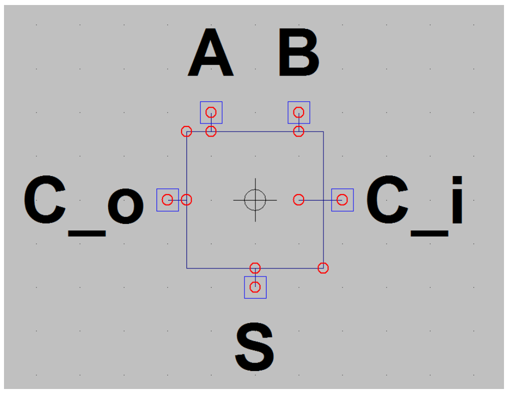


Figure 2: One Bit Full Adder (Block Representation); LTSpice

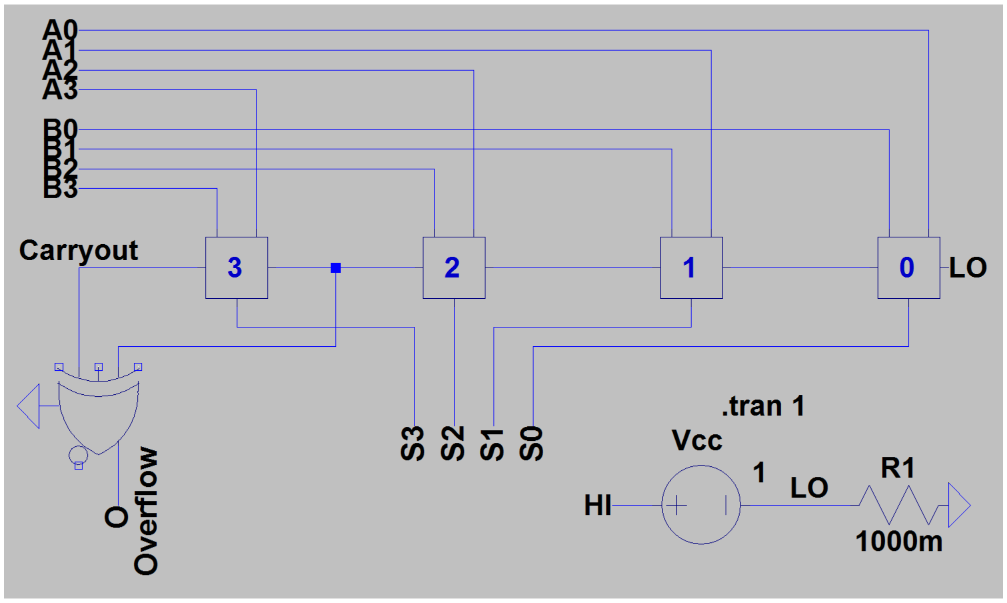


Figure 3: Four Bit Full Adder; LTSpice

Our final full 4-bit adder consisted of four full 1-bit adders coupled together, with an XOR circuit on the final carry-out and carry-in values to determine whether or not an overflow has occurred. The logic behind this is as follows: if the carry-in and the carry-out to the adder operating on the MSB(Most Significant Bit) are the same, the sign of the sum remains consistent with the two summands; otherwise, the sign would flip, resulting in an overflow[1].

Table 2: Semi-Exhaustive Proof; Red marks overflow.

A	B	C_i	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

As seen in the above table,

$$\neg A \neg B C_i + A B \neg C_i = C_i \oplus C_o \tag{1}$$

Which are the cases in which the sign bit of the resultant sum is different from the sign bit of the summands.

Listing 2: "Four Bit Full Adder; Verilog"

```
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
```

[1]Here, the term 'overflow' is loosely defined to include underflow as well: roughly put, 'a state in which the result is inconsistent with the operation'.

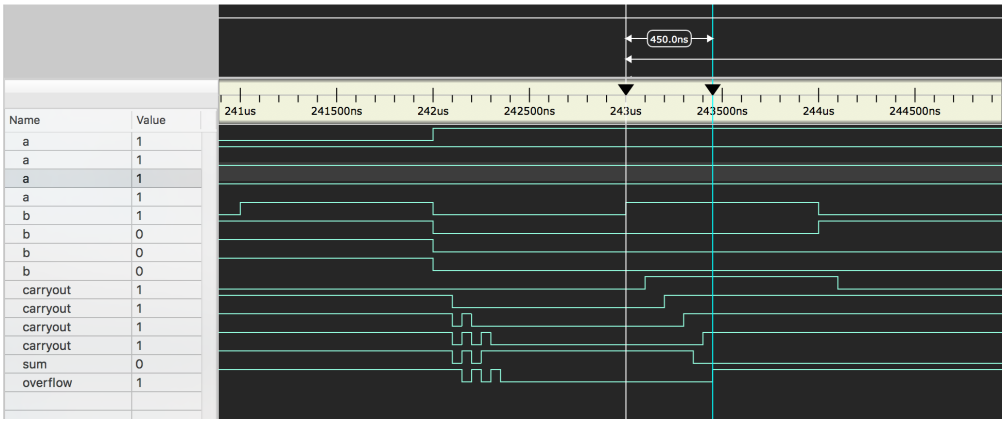
```
    input[3:0] b        // Second operand in 2's complement format
);

wire carryin = 0;

wire carryout_0;
wire carryout_1;
wire carryout_2;

FullAdder1bit fa1(sum[0],carryout_0,a[0],b[0],carryin);
FullAdder1bit fa2(sum[1],carryout_1,a[1],b[1],carryout_0);
FullAdder1bit fa3(sum[2],carryout_2,a[2],b[2],carryout_1);
FullAdder1bit fa4(sum[3],carryout,a[3],b[3],carryout_2);
`XOR xorgate(overflow, carryout, carryout_2);
endmodule
```

2.3 Delay



Name	Value
a	1
a	1
a	1
a	1
b	1
b	0
b	0
b	0
carryout	1
carryout	1
carryout	1
carryout	1
sum	0
overflow	1

Figure 4: Estimated Worst Propagation Delay, 450 ns.

To find the worst case delay, we searched for the time in the waveform when all the carryouts and the overflow flip. In this case, input A stays 1111, and input B switches from 0000 to 0001. All the carryouts switches from 0 to 1 and the overflow switches from 0 to 1. This case results in a delay of 450ns, which is 450/50 = 9 time units.

This is consistent with our design(Fig. 1); starting with the least significant bit, the first carry-out takes three time units for computation. Each route from carry-in to carry-out takes two time units; 3+2+2+2 = 9 time units. In general, for n-bit addition, our circuit will take $3 + 2 * (n - 1)$ units of time. If we're also interested in the overflow bits, the said time would be incremented by one.

2.4 Validation

To fully test our 4-bit full adder, we chose 16 test cases to provide a reasonable coverage:

Table 3: Comprehensive List of Test Cases

A	B	Sum	Overflow	In 2's Complement	Explanation
0	0	0	0	0000 + 0000 = 0000	Test the most basic 0+0 operation, everything should stay 0.
1	2	3	0	0001 + 0010 = 0011	2 positive numbers without overflow.
7	3	10	1	0111 + 0011 = 1010	2 positive numbers with overflow.
7	1	8	1	0111 + 0001 = 1000	2 positive numbers with overflow (at boundary 8).
-1	-3	-4	0	1111 + 1101 = (1)1100	2 negative numbers without overflow.
-5	-6	-11	1	1011 + 1010 = (1)0101	2 negative numbers with overflow.
-4	-4	-8	0	1100 + 1100 = (1)1000	2 negative numbers without overflow (at boundary -8).
-3	-6	-9	1	1101 + 1010 = (1)0111	2 negative numbers with overflow (at boundary -9).
7	7	14	1	0111 + 0111 = 1110	Extreme Case: 2 largest positive numbers
-8	-8	-16	1	1000 + 1000 = (1)0000	Extreme Case: 2 smallest negative numbers
2	-2	0	0	0010 + 1110 = (1)0000	Cancellation: opposite-sign numbers with same magnitude.
-1	-1	-2	0	1111 + 1111 = (1)1110	Signed vs unsigned: would overflow in the unsigned system.
2	1	3	0	0010 + 0001 = 0011	Symmetry: swap A and B in Case 2.
3	0	3	0	0011 + 0000 = 0011	Equality: add 0 to a positive number.
-4	0	-4	0	1100 + 0000 = 1100	Equality: add 0 to a negative number.
-3	7	4	0	1101 + 0111 = (1)0100	Sanity test: random sampling of 2 numbers for verification.

In our first iteration, we implemented a 4-bit full adder in the unsigned system. Naturally, the carryout and overflow were identical. When we tried to directly port this into the signed system, all the test cases in which carryout and overflow were not the same – naturally – failed. Soon, we realized that the adder in the most significant bit was most relevant to determining whether or not the overflow would occur; by examining the table, we observed that the XOR gate between the carryin and carryout of the final adder could equivalently represent the overflow logic (as demonstrated in Table 2). With this new design, our results matched all 16 of our tests consistently. The extensive list containing all 256 scenarios can be found in the attached document (results.txt).

3 FPGA

To perform the tests on the FPGA board, we loaded our tested 4-bit full adder design onto the board using the given wrapper. The four switches, SW0, SW1, SW2, SW3 represent the 4 bit input A and B in binary. By toggling switches and pushing the button BTN0, we gave the input A, and by pushing the button BTN1, we gave the input B. Then pressing on button BTN3 gave us the 4 bit output sum in binary which is represented by the 4 LEDs, LD0, LD1, LD2, LD3. Finally, we push down button BTN3 to check the last carryout according to LD0, and the overflow according to LD1. For all LEDs and switches, on stands for 1 and off stands for 0.

3.1 Testing

In this case, we performed a test on the FPGA for A = 1001 (-7), B = 1111 (-1). First we pulled up SW0 and SW3, and pressed BTN0 to assign 1001 to input A (Figure 1). Then we pulled up all 4 switches and pressed BTN1 to assign 1111 to input B (Figure 2). To show the output sum, we pushed BTN2 down and saw that only LD3 is ON (Figure 4), indicating that the sum is 1000 (-8). Finally, we pushed down BTN3 and found out that LD0 is ON but LD1 is OFF, which shows that the carryout is 1 but overflow is 0. This test case gave us 1001 + 1111 = (1)1000 without overflow.

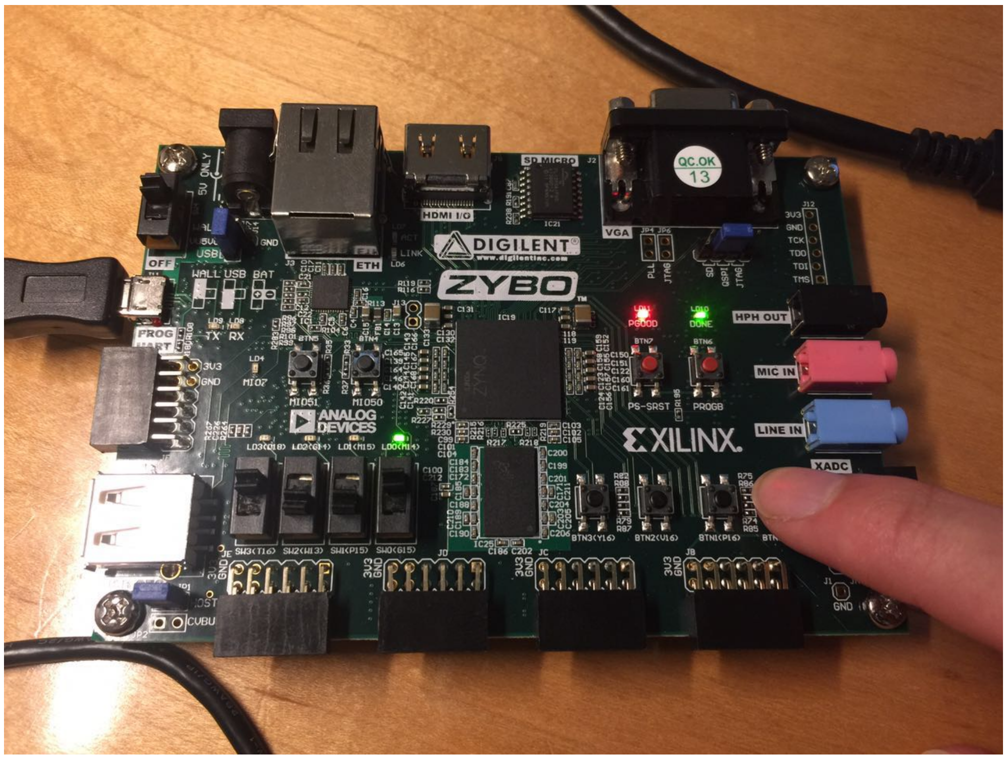


Figure 5: First Summand, 1001 (-7)

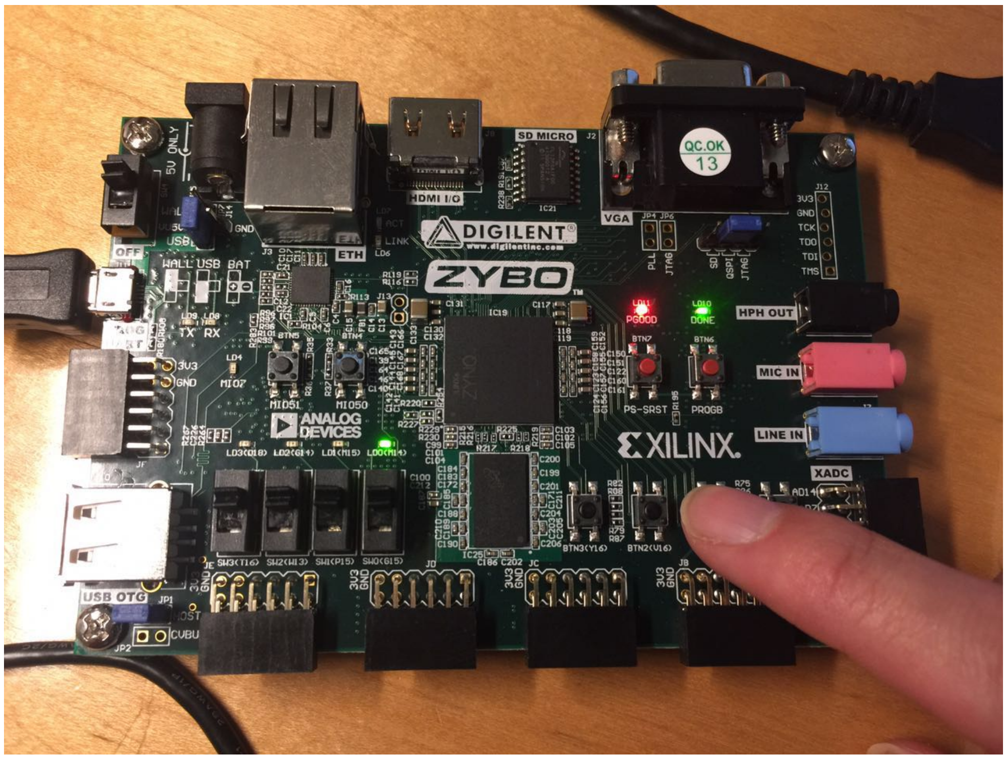


Figure 6: Second Summand, 1111 (-1)

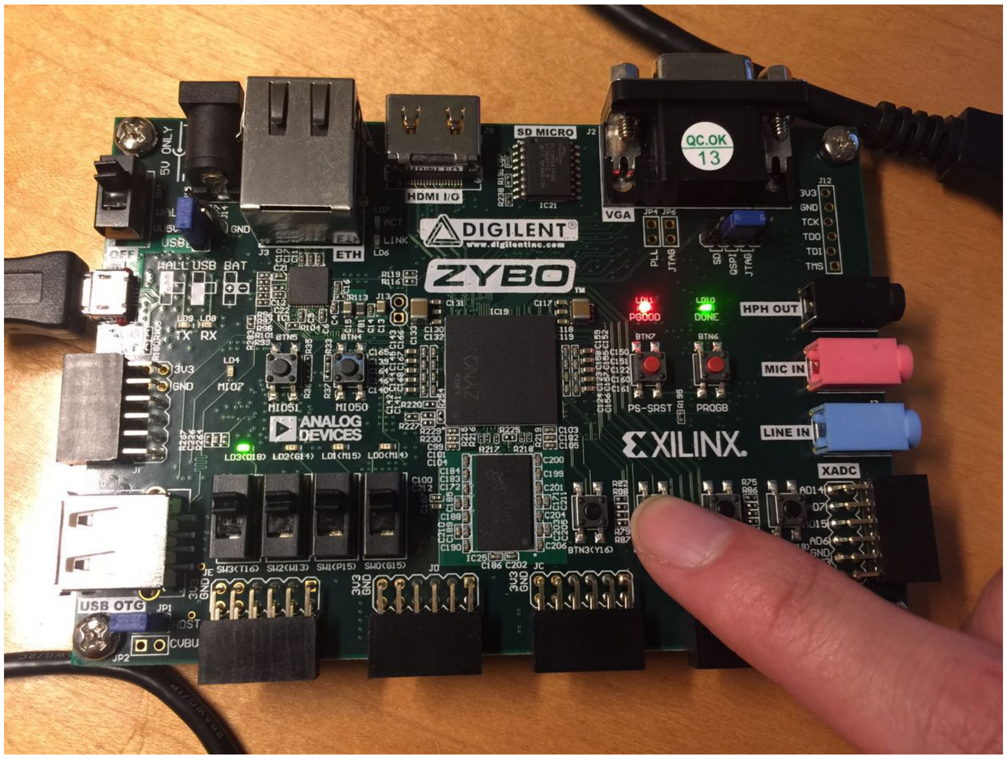


Figure 7: Sum, 1000 (-8)

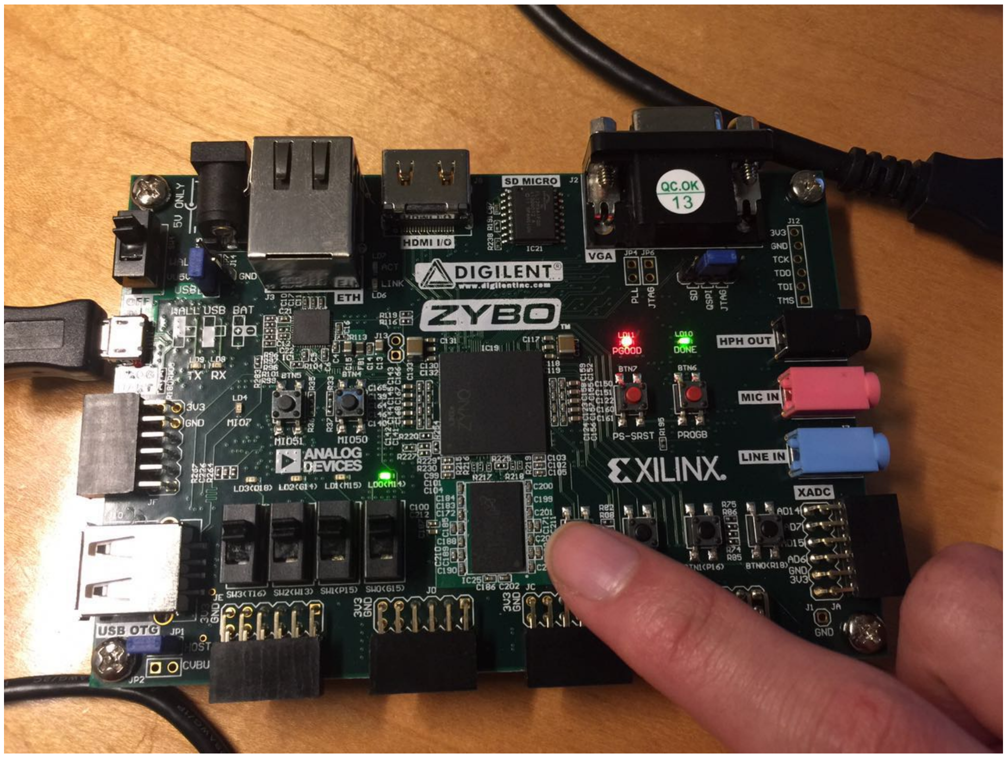


Figure 8: Result; carryout bit is set, but overflow did not occur.

3.2 Statistics

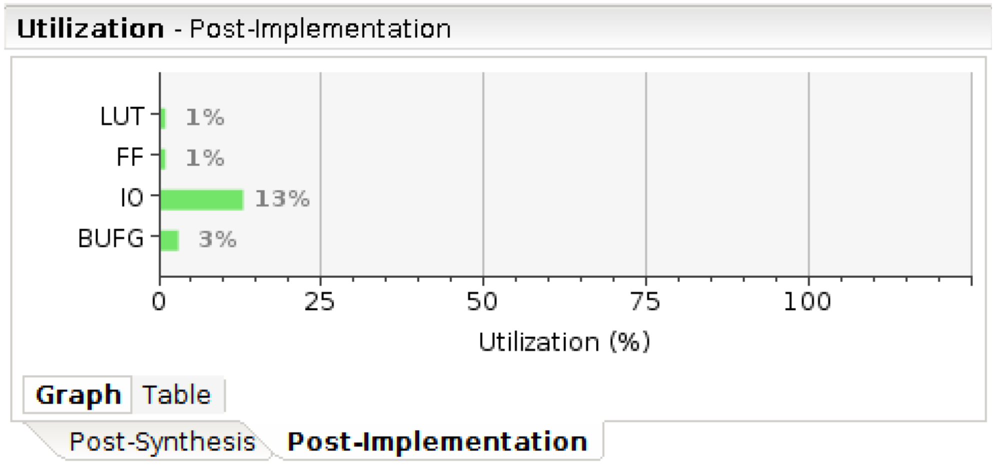


Figure 9: Resources Utilized

Considering the basic statistics analysis from Vivado, most of our resources (which we haven't used a lot) were committed to input and output. This is natural since our implementation involved few computation and concentrated more on interactivity (i.e. pressing switches and toggling LEDs, etc.). As the small percentage of LUT, FF, and BUFG suggests, our application was not very storage/data intensive; it also occupies a very small portion of the arrays.