ECE385
Spring 2019
Experiment #4

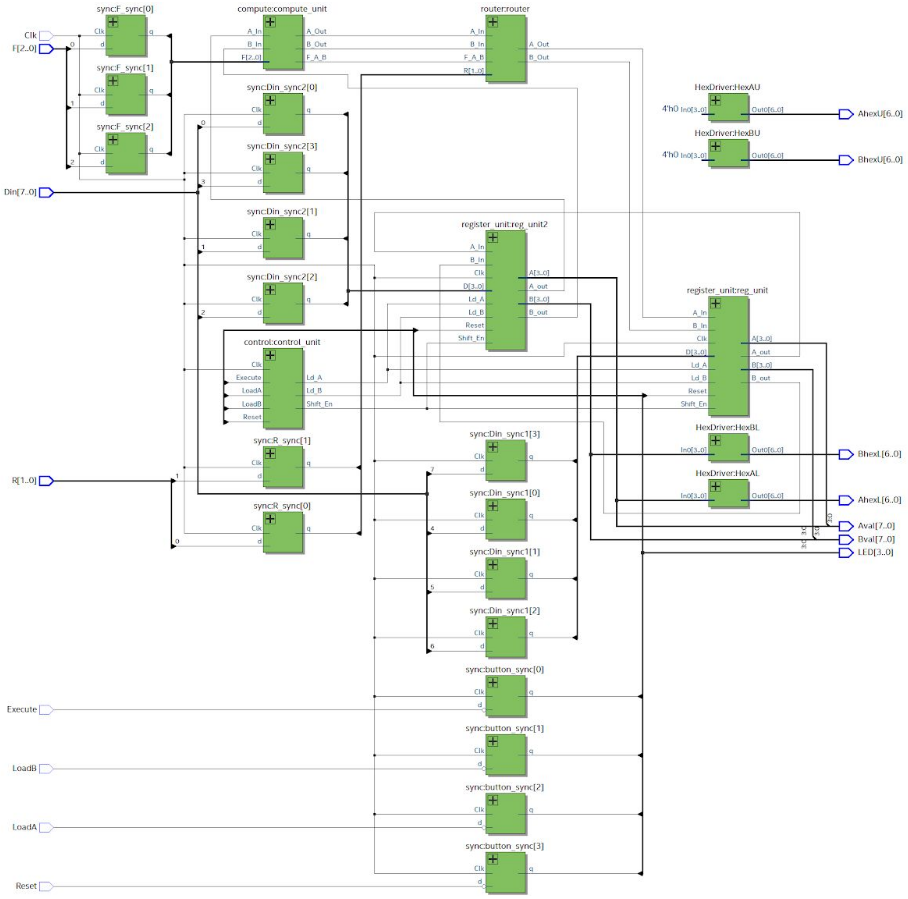
Introduction to System Verilog
&
16 Bit Adders

Austin Lee (sal3)
Zuling Chen (zulingc2)
Tuesday 3PM ABE
Patrick Wang, Gene Shiue

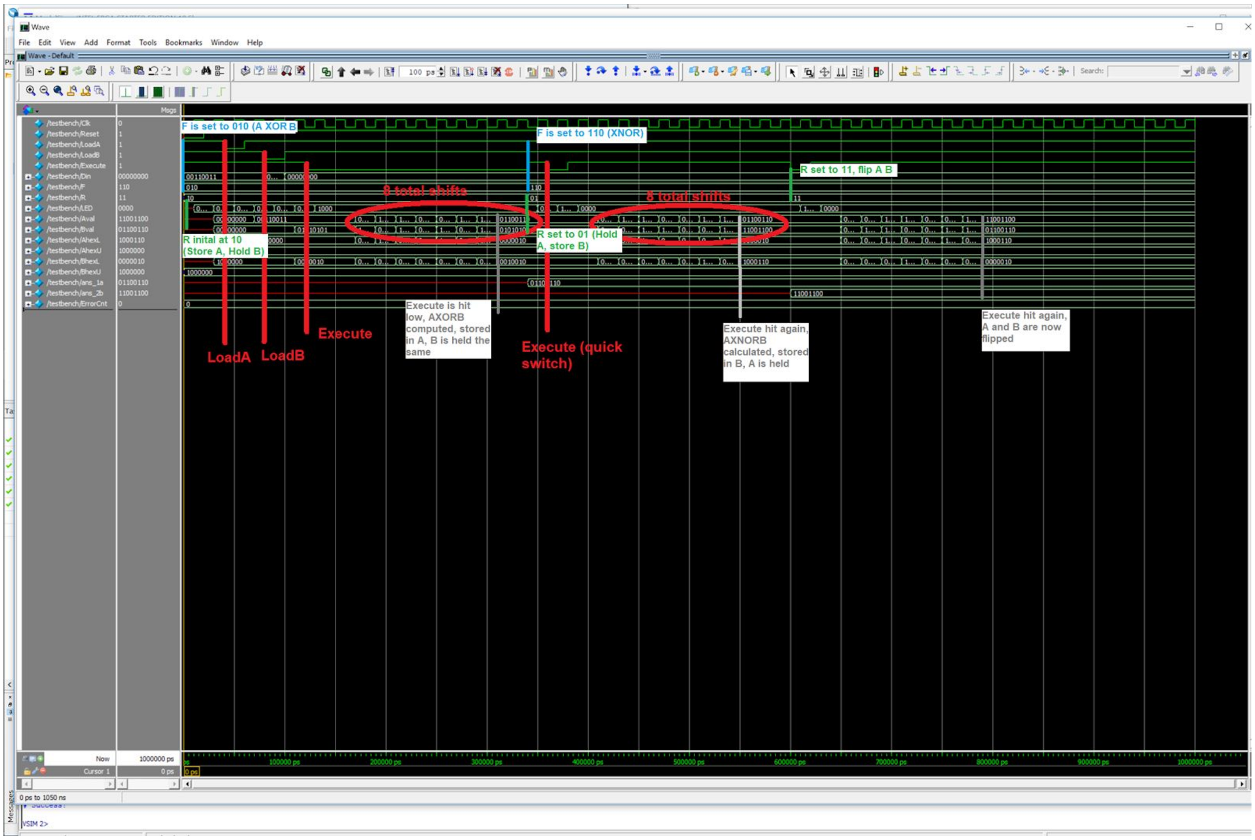
Introduction

After a few TTL labs dealing with physical circuitry, we have started to move into the software part of ECE385. This lab served as an introduction to SystemVerilog. Through use of the FPGA board and SystemVerilog, we were supposed to expand our last lab's 4 bit processor into an 8 bit processor as well as implementing three types of 16-bit adders: a ripple adder, a carry look ahead adder, and a carry select adder.

Processor Block Diagram



Processor Annotated Simulation



Written description

To use the adder circuit, we had to “activate” different sections of our top level code by toggling commenting for each different type of adder. This was because they all wired to the same inputs and outputs.

First, data was stored into the B register from the switches and on press of the LoadB signal (KEY1). The A register simply took whatever data was currently on the switches--it was loaded on every cycle-- and pressing RUN (KEY3) would add the two registers using the adder that was activated in the code. The output was displayed on a 16-bit LED display (LEDR0 through LEDR15). At any time, the RESET button (KEY0) could be used to erase the stored data in register B.

Module description(s)

In order to implement three different adders, we constructed three different “helper” modules. In the case of the look-ahead and carry select adders, the helper module allowed us to reduce the computation into four 4-bit chunks. Namely, we utilized a hierarchical design to decrease computation time.

For the ripple adder, we designed a simple 2-bit full adder first. The full adder takes two bit inputs, a carry in, and outputs a sum and carry out. We used sixteen full adders, inputting each next carry-in as the previous’ carry-out, to complete the ripple adder. Each full adder’s sum output was also stored into the respective bit in our ripple adder sum output.

For the look-ahead adder, our helper module took in two 4-bit inputs, the carry in, and outputted a sum and two 1-bits. Namely, the two output bits determined if a carryout is generated or potentially propagated. The helper module predetermined the carry in bits for each bit in the two 4-bit inputs and used said carry in bits in four full adders. The combinational logic for the carry ins and generate/propagate bits are:

```
c1 = (c_in & p[0]) | g[0];
c2 = (c_in & p[0] & p[1]) | (g[0] & p[1]) | g[1];
c3 = (c_in & p[0] & p[1] & p[2]) | (g[0] & p[1] & p[2]) | (g[1] & p[2]) | g[2];
po = p[0] & p[1] & p[2] & p[3];
go = g[3] | (g[2] & p[3]) | (g[1] & p[3] & p[2]) | (g[0] & p[3] & p[2] & p[1]);
```

By using four of the look-ahead helper module, we were able to complete the implementation of the look-ahead adder. Before that however, we also had to pretermine the carry in bits for each call to the helper module based on the generate/propagate outputs of the previous call. The logic used is as follows, where CO is the carry-out bit of the look-ahead adder :

```
c1 = (1'b0 & p0) | g0;
c2 = (1'b0 & p0 & p1) | (g0 & p1) | g1;
c3 = (1'b0 & p0 & p1 & p2) | (g0 & p1 & p2) | (g1 & p2) | g2;
CO = (1'b0 & p0 & p1 & p2 & p3) | (g0 & p1 & p2 & p3) | (g1 & p2 & p3) |
(g2 & p3) | g3;
```

For the carry-select adder, our helper module once again broke the 16-bit inputs into four 4-bit sections. Each instance of the helper module also received a carry in as an input and outputted a 4-bit sum and carry out. The helper module basically performed a small 4-bit ripple adder but did so twice, once with the initial carry in bit as a 1 and the other with the initial carry in bit as a 0. Then, based on what the actual carry in bit input was to that instance of the module, the sum output would either be the sum from the first ripple adder or the second. Our carry-select adder module used four instances of the carry-select helper module to complete its task.

Comparing Adders

The first adder was the carry ripple adder. This adder is the slowest of the three because it had to go through each bit, calculate the sum and carry out bit, then send that carry out bit to the next adder. However, the simplicity of this design meant that this could be implemented with the least amount of logic gates.

The use of a carry-select and look ahead adder makes for quicker calculation of sums. Rather than going bit by bit and waiting to see what the next carry in bit will be is, the other two adders make use of a 4x4 hierarchical design to calculate the potential values of a set of four bits simultaneously. This way, the adder doesn't have to wait to get to a specific bit in order to start adding, rather it can calculate the result for both carry in bit of 0 and 1 and have the result ready for when the program reaches that group of four. By calculating the result ahead of time either with calculating both sums for carry bits of 0 and 1 or by predicting the carryout bit, rather than waiting to calculate the answer, the program could select one of two options and move on.

Block Diagram(s)

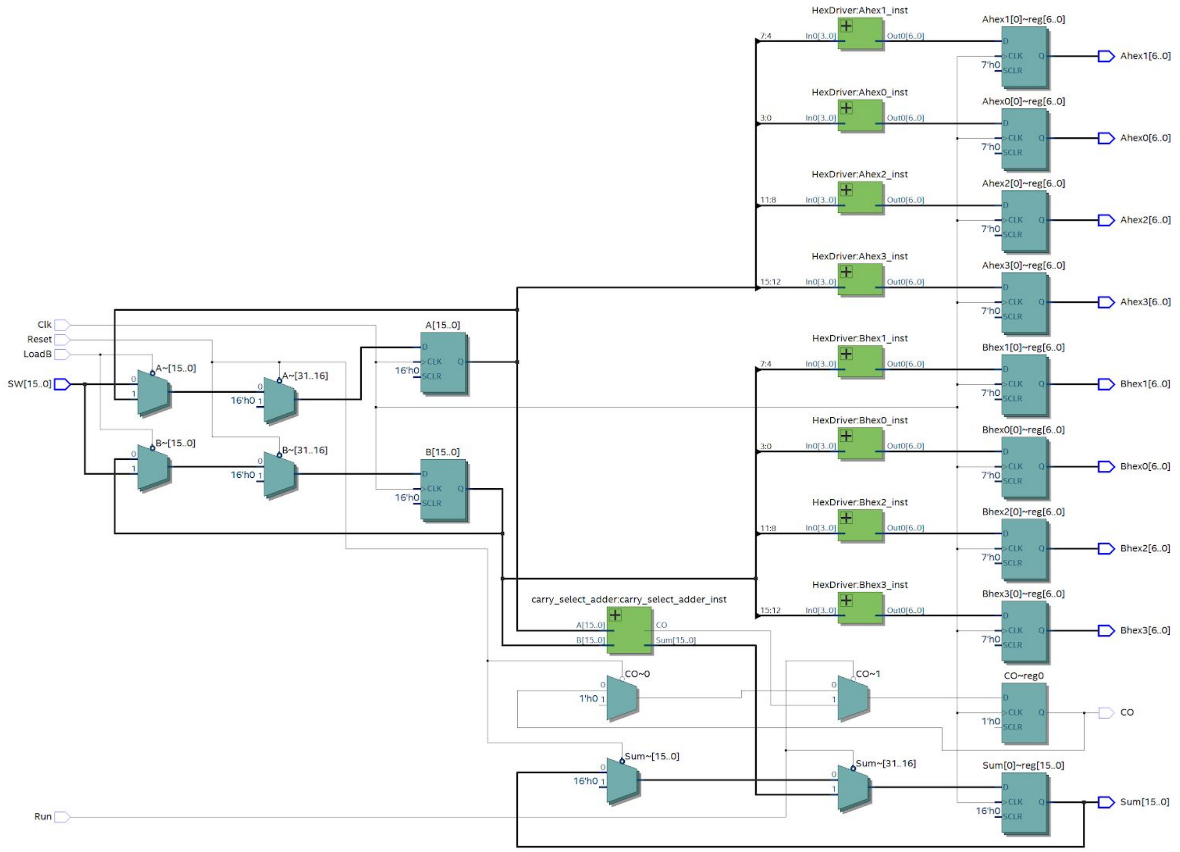


Figure 1: Top-level adder block diagram

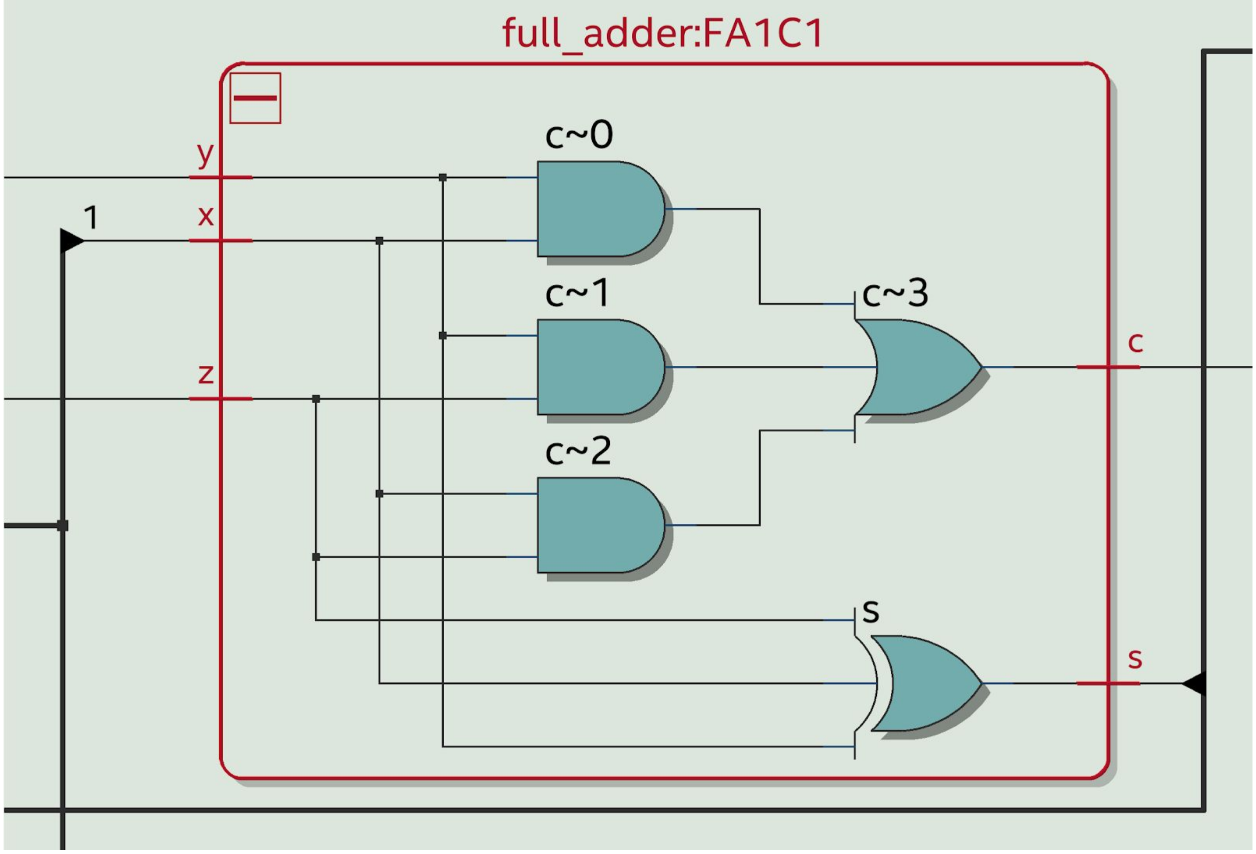


Figure 2: Full adder block diagram

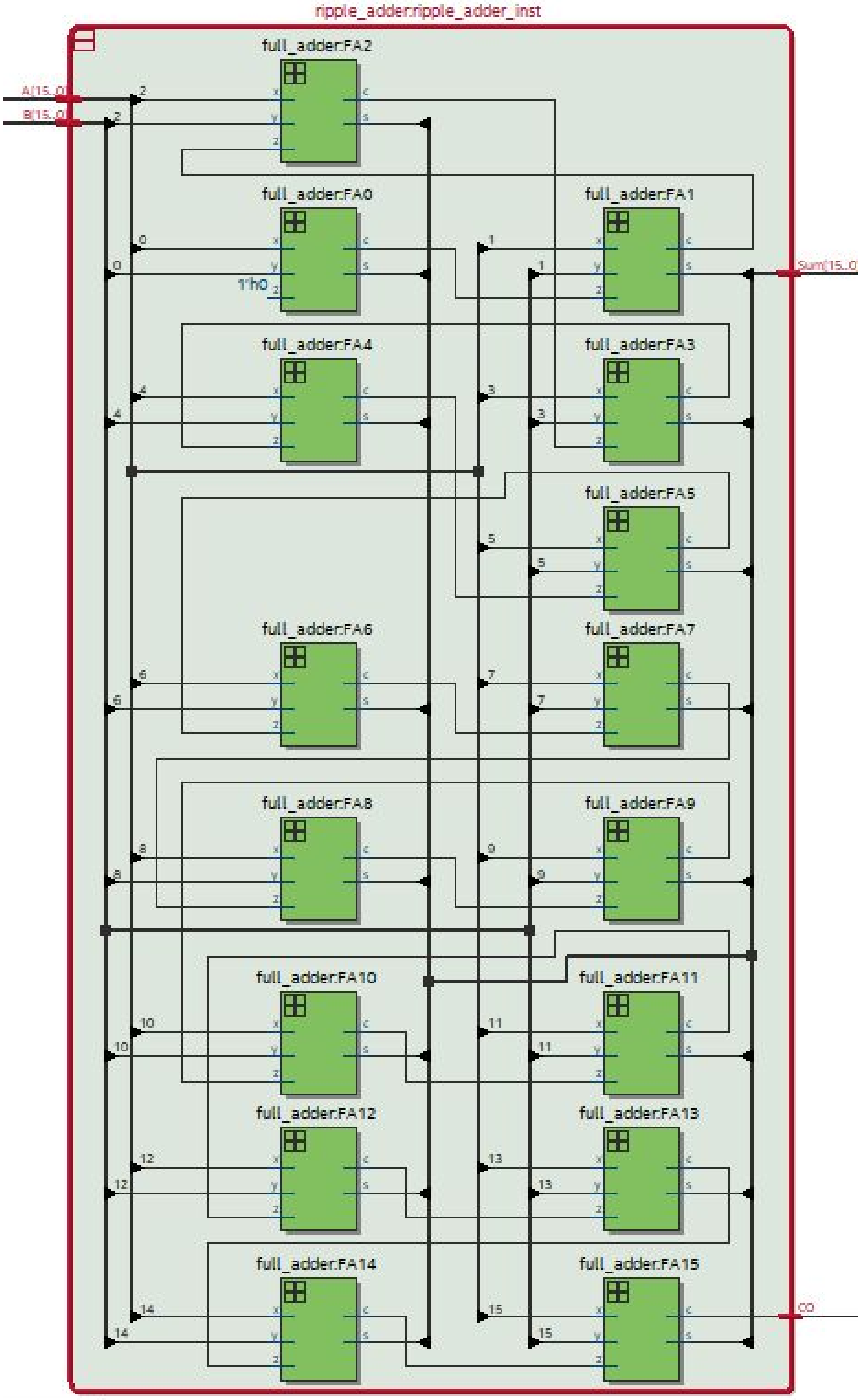


Figure 3: Ripple adder block diagram

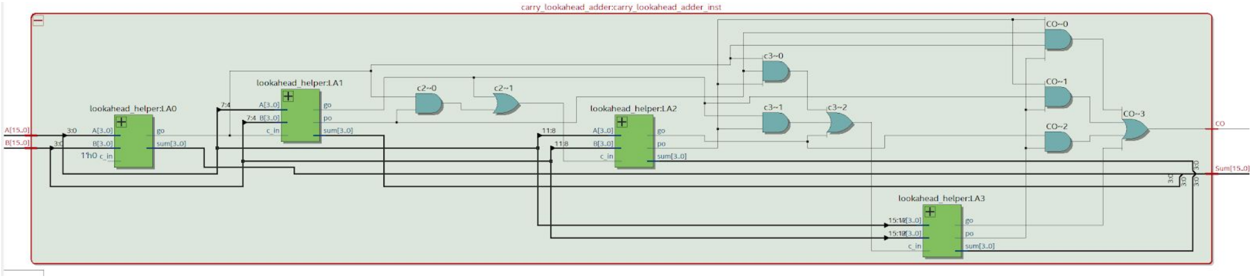


Figure 4: Look-ahead adder block diagram

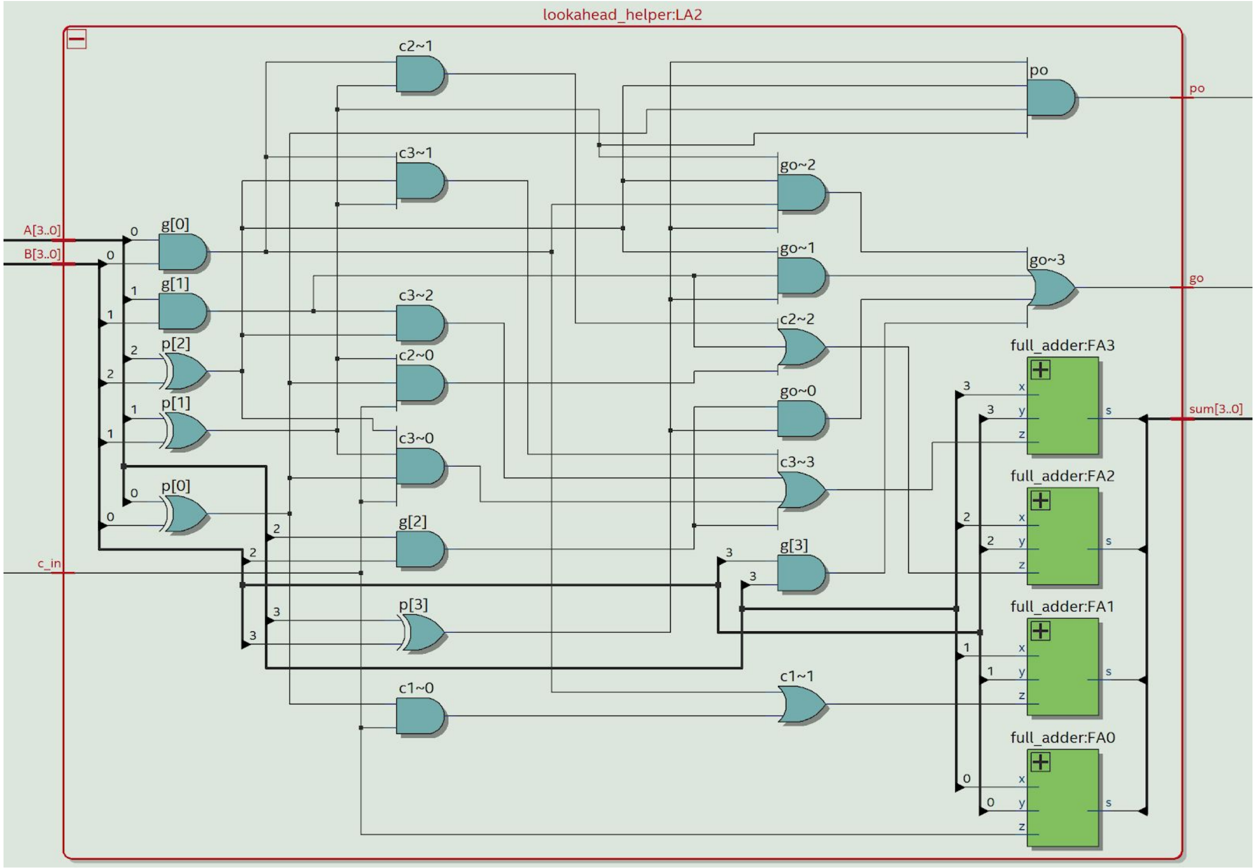


Figure 5: Look-ahead adder helper module

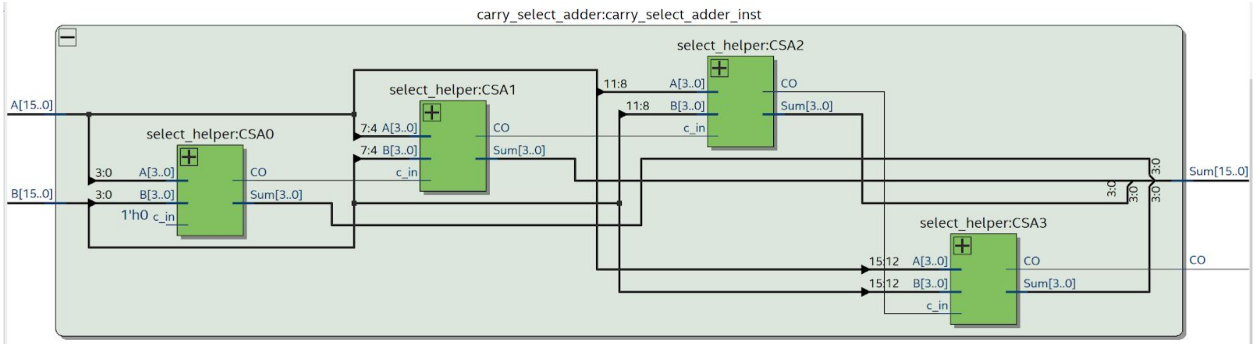


Figure 4: Carry select adder block diagram

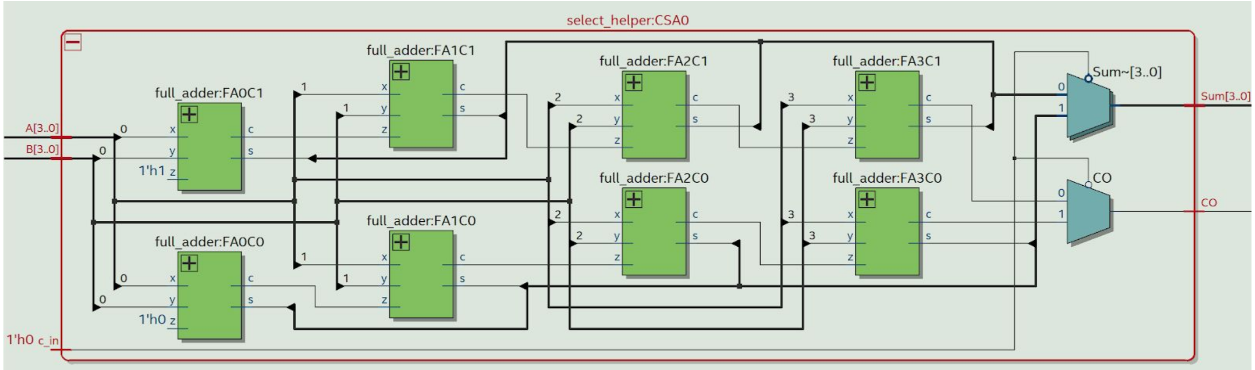
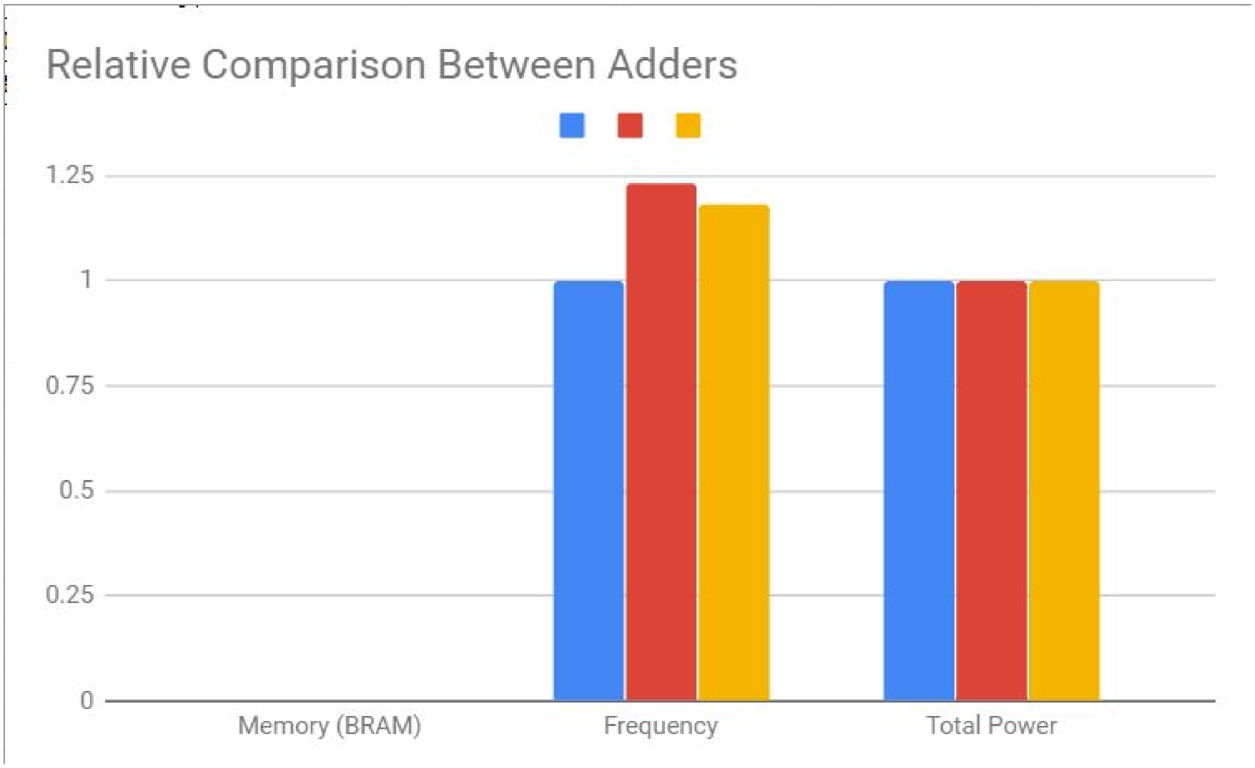


Figure : Carry select adder helper module

Adder Comparisons (**Blue** -Ripple Adder, **Red** - Carry Select, **Yellow-** Carry LookAhead)



	Carry-Ripple	Carry-Select	Carry-Lookahead
Memory (BRAM)	0	0	0
Frequency	62.10 Mhz	85.08 Mhz	86.41 Mhz
Total Power	101.8 mW	105.82 mW	105.82 mW

Post-lab questions

1. 8 Bit Serial Processor

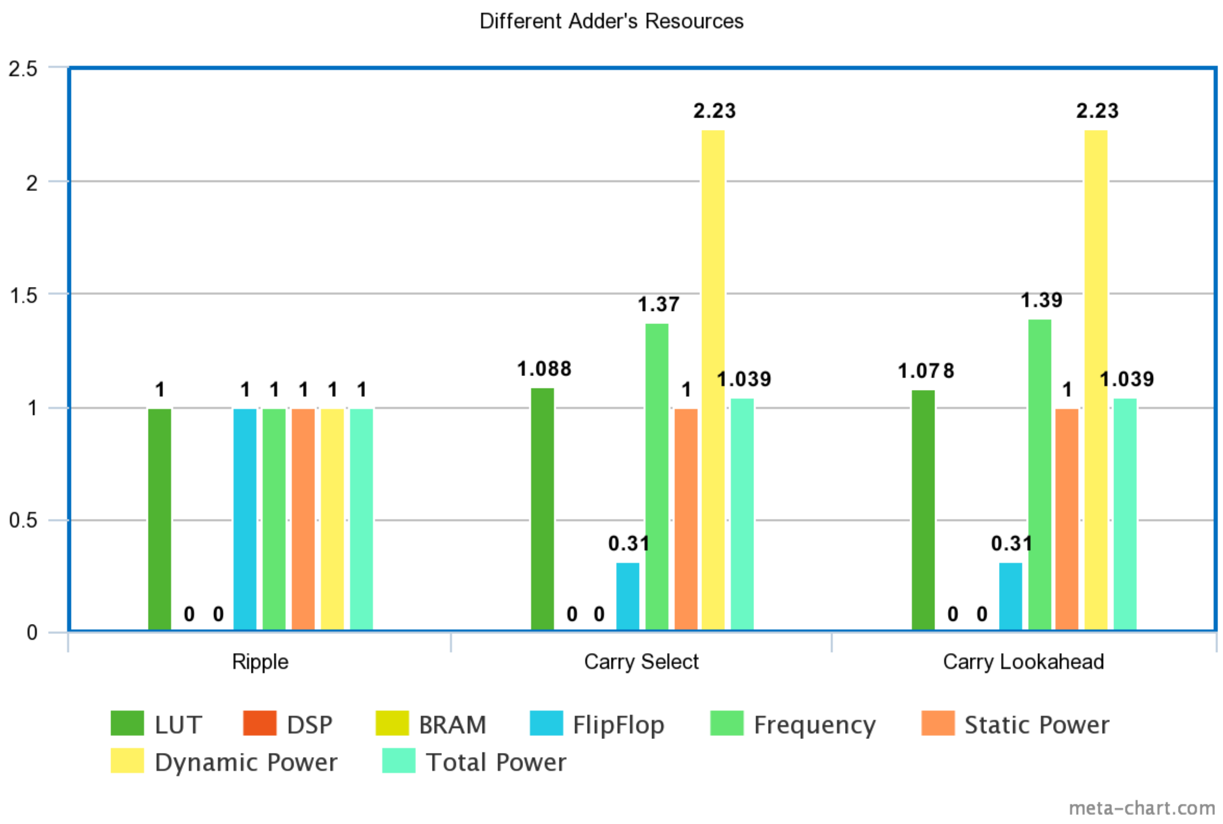
LUT	70
DSP	0

BRAM	0
Flip flop	43
Frequency	82.48 Mhz
Static Power	98.52 mW
Dynamic Power	2.45 mW
Total Power	100.97 mW

Although Lab 3 only had us create a 4-bit serial logic processor, we can extend the idea of it to 8 bits and then compare it with the one generated with SystemVerilog. From looking at the chart, it seems that this lab's used more resources to fully implement the 8 bit processor. We must also consider that in Lab 3, the chips used had a preset number of logical gates. Even if we only needed one NAND gate, we needed to use an entire chip containing multiple NAND gates. The FPGA takes out the need for this and only leaves behind the necessary elemental logic units to be used. In Lab 3, placement of chips on the breadboard was limited by human error and design whereas in the FPGA, this is not an issue which results in an optimization of the space and performance of the designs.

2.

	Ripple	Carry Select	Carry Lookahead
LUT	114	124	123
DSP	0	0	0
BRAM	0	0	0
FlipFlop	105	32	32
Frequency	62.10 Mhz	85.08 Mhz	86.41 Mhz
Static Power	98.55 mW	98.57 mW	98.57 mW
Dynamic Power	3.25 mW	7.25 mW	7.25 mW
Total Power	101.8 mW	105.82 mW	105.82 mW



Looking at the graph, the only thing that really seems to stand out was the fact that the carry-select and carry-lookahead adders both had the same dynamic power use and total power. Although the number of resources was very similar between the two, they still differ by one, which means that the power should vary at least a tiny bit. This could be just due to a glitch in the power analysis tool in SystemVerilog. The other data points however do match the expected results of the experiment. Because both the carry select and carry look ahead adders both have more logical units, they should have more power and operate at a higher frequency than the carry ripple adder. The power used by all three was about the same, but the only major difference was the dynamic power used. The frequency of the carry look ahead adder is indeed higher than the carry select adder, however, it was only by a small margin. Looking at the number of logical units, we should have expected that carry select adder to have the most power consumption. We did not get this result perhaps due to only small difference in the number of logical units used.

Conclusion

During the whole design process of the 8 bit processor in SystemVerilog, we thought that all we had to do was change the values of some registers to hold 8 bits instead of 4. Unfortunately, we had overlooked the fact that now because it was an eight bit processor, we needed more states in the FSM for the correct number of shifts to occur. After looking into this and adding more states, we finally got our processor working and our computations to be correct.

The biggest issue we had with designing the adders was just working with the syntax of SystemVerilog. While trying to print out the carry out bit, we had originally stored the answer into an array we created for carry out bits, but we were not using that last bit of data to send to our LEDs. We also encountered an issue when we tried to print out the carry out bit we named CO. CO was also the name we had used to call the carry out bit we passed along to the next function. This resulted in us not knowing how to send to the LEDs and eventually we had to set CO to CO, which was confusing to implement.

All in all, we were completely successful in achieving this lab's objectives.