**ECE 385**

Fall 2019

Experiment #4

Introduction to System Verilog, FPGA, EDA, and 16-bit Adders

Jacqueline Jiang and Ji Ho Han

AB6 / Wednesday 12:00 - 2:50pm

Nicholas Cebry and Yujie Zhou

# Introduction

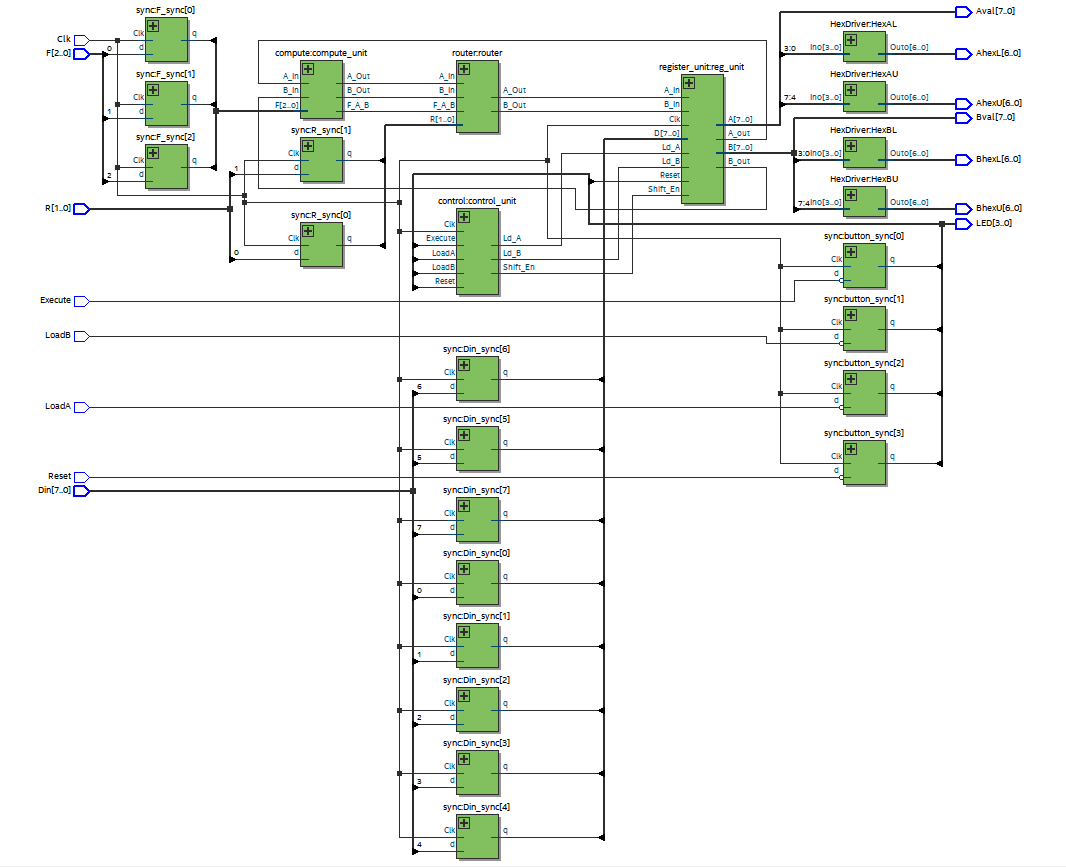
## Summary of the High-Level Function

There were two main purposes for this experiment. The first objective was to redesign a 4-bit serial logic processor to support 8 bits through SystemVerilog. This bit-serial logic processor will take two 8-bit inputs, perform a set calculation, and route the result back to the set register. The second goal of this experiment was to develop three different types of 16-bit adders and implement this design on an FPGA board using SystemVerilog. The adders implemented in this experiment will take in two 16-bit inputs and output a 16-bit sum using the Carry-Ripple method, the Carry-Lookahead method, and the Carry-Select method.

# 

# Part 1: Bit-Serial Logic Processor

## Block Diagram



*Figure 1: Block Diagram of Bit-Serial Logic Processor*

## 

## Short Description about the Extension from 4 Bits to 8 Bits

The bit serial logic processor originally takes in 4-bit operands and will perform one of 8 operations on it (AND, OR, XOR, SET, NAND, NOR, XNOR, CLEAR). It will then route the result to one of the four options: HOLD (the values in the registers), A, B, SWAP (the registers). This entire process will run when EXECUTE is toggled. We designed a TTL version of this processor in Lab 3, and our goal in this lab was to extend the registers to 8-bits using SystemVerilog.

In order to extend the bit serial logic processor from 4-bits to 8-bits, there were a few files that needed to be changed. First, we extended the registers to 8 bits by changing the number of bits accepted and accommodating accordingly in the Reset and Shift functionality of the registers. Afterwards, we edited any input or output logic that used these registers to support 8 bits. The input data (Din) and its relatives (Din\_S and D) were also adjusted to support 8 bits, and the HEX drivers were set to support the upper 4 bits of the modified registers. Lastly, we added an extra 4 states to the state machine so that the upper 4 bits are also processed. For this entire process, the only files modified are *Processor.sv*, *Reg\_4.sv*, *Register\_unit.sv*, and *Control.sv*. None of the other files need to be changed since *Router.sv* and *compute.sv* only did bitwise operations, and *HexDriver.sv* and *Synchronizers.sv* were modular units that took a specific set of bits to synchronize with the board.

## Simulation of the Processor

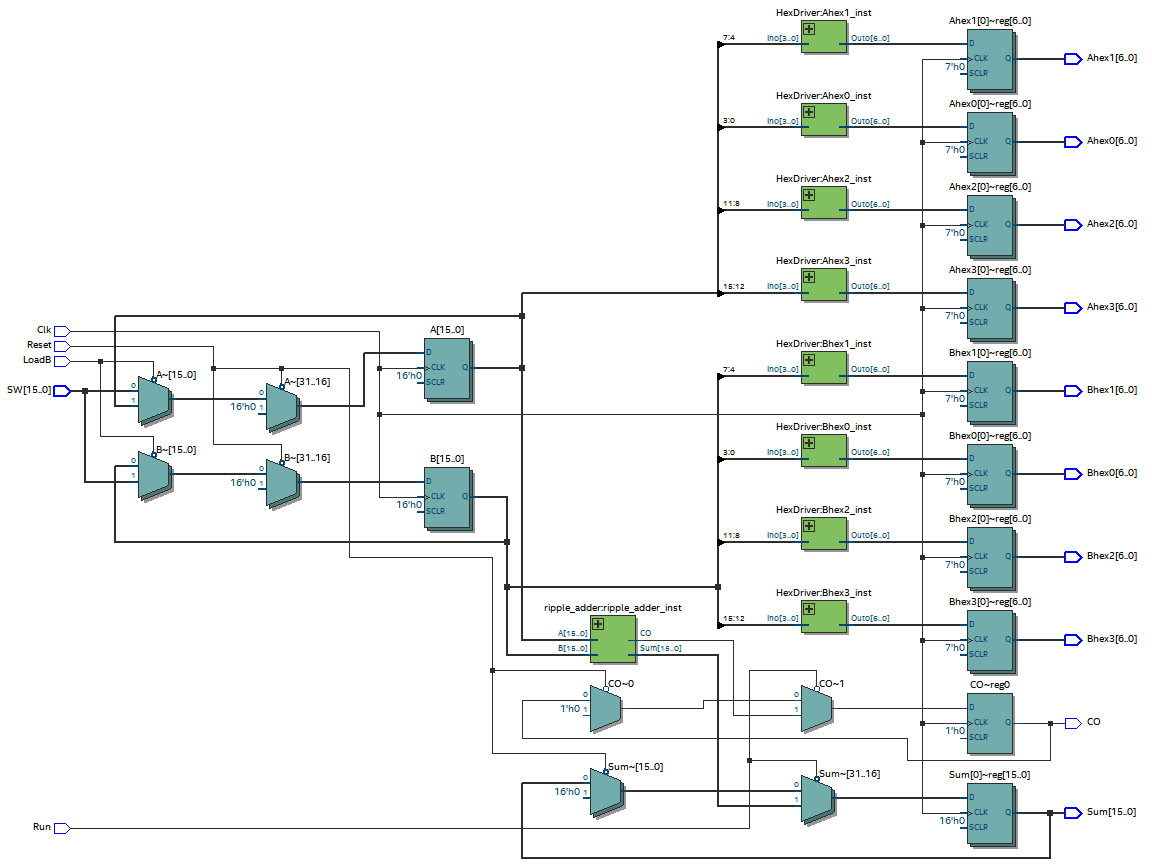
The simulation of the bit-serial logic processor is shown below (*Fig 2*). Within this simulation, the circuit was first reset, and then registers A and B are loaded with values obtained from the switches. Subsequently, three operations were performed. The first operation was A XOR B and its results were stored in A. The second operation is A XNOR B and store the result in B. The third operation is to swap the results in A and B.

## 

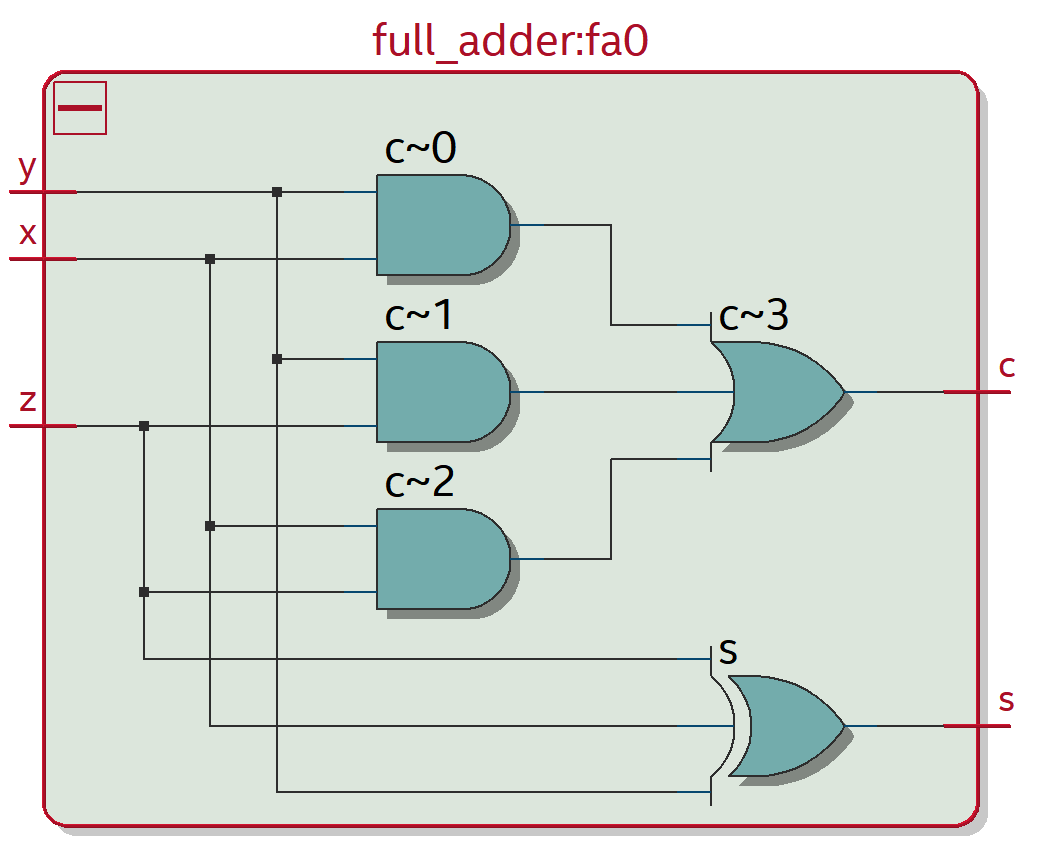
## *Figure 2: Waveforms for the Bit-Serial Logic Processor*

# Part 2: Adder Overview

For the second portion of the experiment, we created 16-bit adders using different algorithms. The overall design of the adder circuit can be seen in *Figure 3*. For all of the adder implementations, similar designs were used. The only difference between the designs was the adders used in the implementation. In addition, all of our adders within this experiment were implemented using full adders (*Fig. 4*), which takes in two 1-bit operands and a carry-in bit and outputs a 1-bit sum and carry-out bit. A waveform showing the functionality of our adder circuit is shown in *Figure 5*.



## *Figure 3: Block Diagram of the Overall Adder Circuit*



## *Figure 4: Block Diagram of a Full Adder*

# 

## *Figure 5: Waveforms for the Adder Circuit*

# Part 2.a: Carry-Ripple Adder

## Written Description of the Architecture of the RCA

The 16-bit Carry-Ripple Adder is the simplest adder implemented in this experiment. This adder was designed by cascading 16 full adders serially. This means that the carry-out of the full adder of the prior bit must be calculated before progressing to the next addition operation because the carry-out of the prior bit is directly fed to the full adder of the next bit as the carry-in. A block diagram of the Carry-Ripple Adder can be found in *Figure 6*.

## Block Diagram of Carry-Ripple Adder

# 

*Figure 6: Block Diagram of Carry-Ripple Adder*

# 

# 

# Part 2.b: Carry-Lookahead Adder

## Written Description of the Architecture of CLA - How P and G Logic is Used

The Carry-Lookahead Adder (CLA) utilizes, as the name itself implies, carry-lookahead logic, which predicts the value of the carry-out bit based on the carry-in bit. This adder determines the carry bits in parallel with the full adder calculations through the use of P and G logic. P, which stands for propagate, and G, which stands for generate, are the values that will determine the carry-out bits. Computed using XOR, P is only set when either one of the operands, but not both, is equal to 1 whilst G, implemented using AND, is only set when both operands are equal to 1. The equation for calculating any carry-out bit *i* is shown below.

The way the carry-lookahead logic works is quite simple—the idea starts from the fact that there will be 2 situations that will result in a carry-out bit. One of them is if both of the operands that are being added are equal to 1—which will, without a doubt, “generate” a carry-out bit. The other method that can result in a carry-out bit is if the carry-in bit is equal to 1 and one of the operands is also equal to one 1; this will “propagate” the carry-out bit.

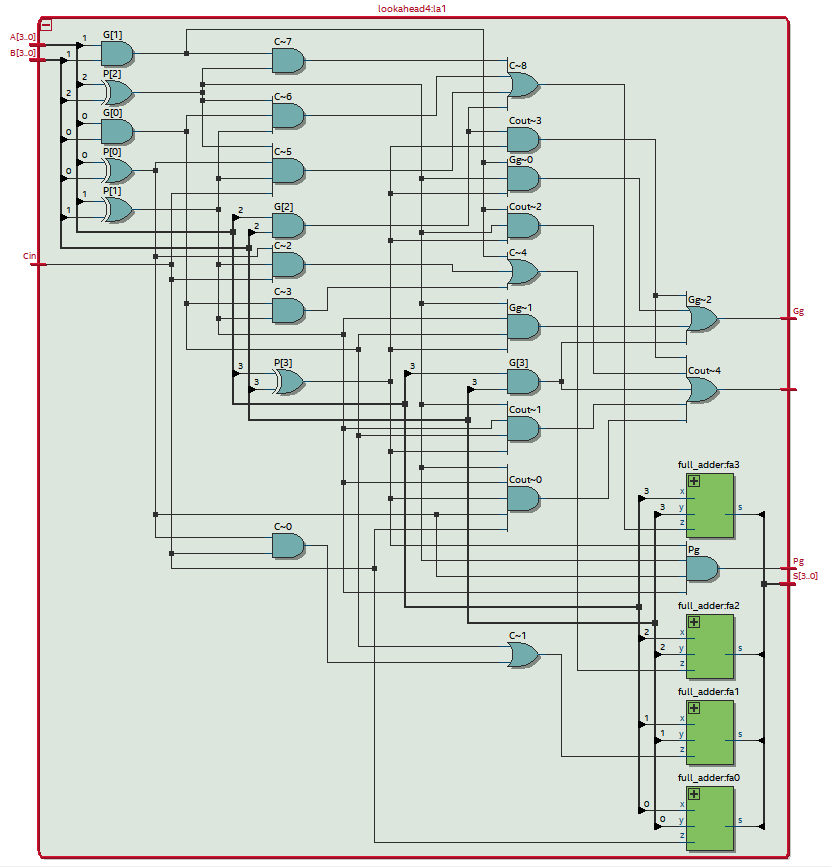
One of the limitations of using P and G logic is that the calculation is strictly serial. Because each carry-out bit depends on the carry-in bit, each carry bit calculated through the P and G method is dependent on the last calculation. This creates two issues: the number of logic gates between the first carry-in bit and the last carry-out bit grows exponentially with each bit added and the maximum frequency of this operation is limited due to this dependency.

## Written Description of the Architecture of CLA - How Hierarchical 4x4 Adder was Created

Because the Carry-Lookahead Adder becomes more inefficient as the number of bits increases, the hierarchical design was created. Instead of calculating all 16 carry bits at once, the 16-bit Carry-Lookahead Adder was divided up into four units. Each of these units will calculate a 4-bit sum and output PG and GG, a group P and G bit, asynchronously for the carry-in calculation of the next Carry-Lookahead Unit. The calculation for the next carry-in bit for the 4-bit adder is similar to that for the regular full adder except that PG and GG are used in place of P and G. This implementation is slightly faster than cascading the adder units together in a Carry-Ripple Adder fashion because the carry bits are still being calculated in parallel to the actual addition. A 4-bit Carry-Lookahead Adder unit can be found in *Figure 7*, and *Figure 8* shows how the 16-bit Carry-Lookahead Adder chained the four adder units together.

## 

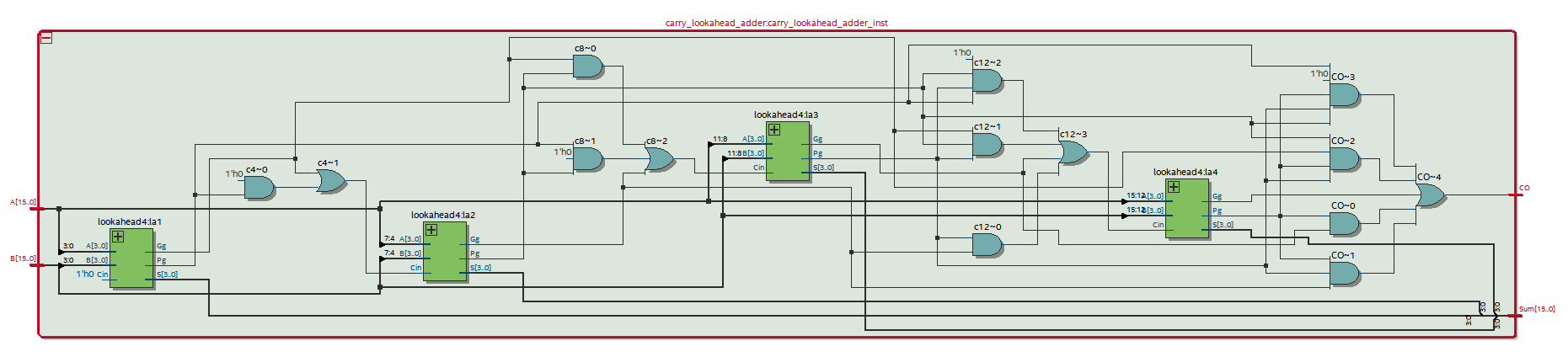
## Block Diagram - Inside a Single CLA (4 bits)



*Figure 7: Diagram of a 4-Bit Carry-Lookahead Adder Unit*

## 

## Block Diagram - How Each CLAs were Chained Together



*Figure 8: Block Diagram of a 16-Bit Carry-Lookahead Adder*

# Part 2.c: Carry-Select Adder

## Written Description of the Architecture of CSA - How it Chooses the Correct Sums

The Carry-Select Adder (CSA) is the third adder we implemented and utilizes parallel calculations to maximize the speed of computation. This adder approaches the carry bit issue by not caring about the carry-out bit until the very end. Instead, this adder will calculate the sum of each bit both with the carry bit as 0 and with the carry bit as 1, and it decides which sum to use once the real carry-out bit has been determined. The corresponding sum and carry-out bit used for the final output is determined with a 2-to-1 MUX with the correct carry-in bit as the select. The first full adder used already has a determined carry-in bit, which is 0; however, the calculation of every bit afterwards requires two full adders in order to precalculate the sums.

The 16-bit Carry-Select Adder was implemented in a 7x4-bit hierarchy design. This means that we have 4-bit Carry-Select Adder Units to calculate 4-bit sums. These adder units also have seven full adders each. The reason for this expands on the fact that the first carry-in bit is known, so it reduces the number of full adders used by one. This concept is then expanded upon with the overall 16-bit Carry Select Adder because the first adder unit has a known carry-in bit, but all of the following adder units have an undetermined carry-in bit, so both possibilities are calculated. The block diagram of the 4-bit Carry-Select Adder Unit is shown in *Figure 9*, and the overall 16-bit Carry-Select Adder is shown in *Figure 10*.

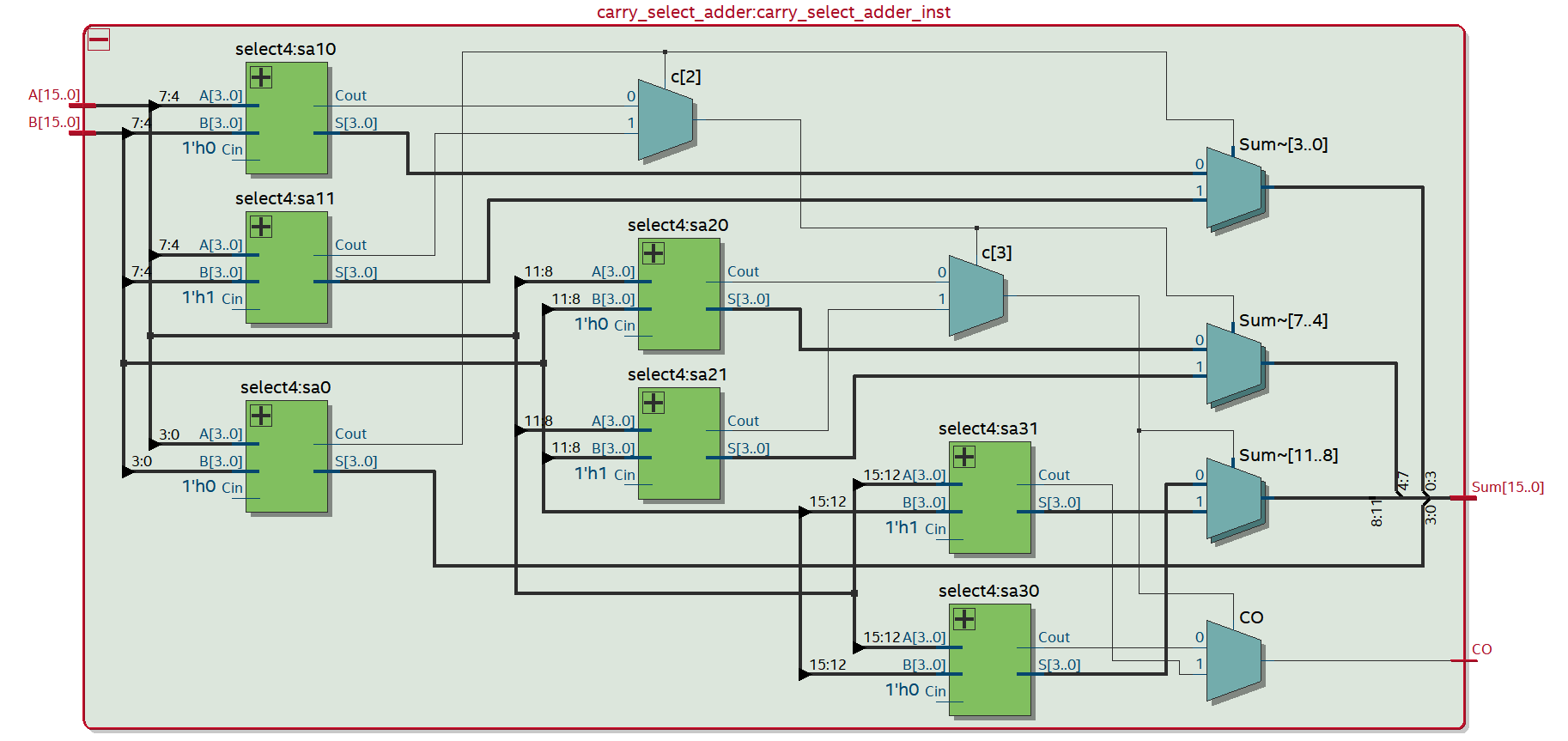
## 

## Block Diagram - Inside a Single CSA (4 bits)

## 

*Figure 9: Block Diagram of a 4-Bit Carry-Select Adder Unit*

## Block Diagram of the Carry-Select Adder



*Figure 10: Block Diagram of a 16-Bit Carry-Select Adder*

# Part 2.d: Written Description of all .sv Modules

Module: lab4\_adders\_toplevel from *lab4\_adders\_toplevel.sv*

Inputs: Clk, Reset, LoadB, Run, [7:0]SW

Outputs: CO, [15:0]Sum, [6:0]Ahex0, [6:0]Ahex1, [6:0]Ahex2, [6:0]Ahex3, [6:0]Bhex0, [6:0]Bhex1, [6:0]Bhex2, [6:0]Bhex3

Description: This is a top-level entity that connects the LEDs and buttons from the board to the adder circuit.

Purpose: This module is used as a top-level wrapper to connect the other modules to the inputs and outputs of the FPGA board.

Module: full\_adder from *full\_adder.sv*

Inputs: x, y, z

Outputs: s, c

Description: This is a single-bit adder that takes in two operands and a carry-in bit and outputs the sum and carry-out bit.

Purpose: This module is used to add two 1-bit values together.

Module: ripple\_adder from *ripple\_adder.sv*

Inputs: [15:0]A, [15:0]B

Outputs: [15:0]Sum, CO

Description: This is a 16-bit carry-ripple adder. It will add the bits together using full adders and links up each adder by feeding the carry-out bit to the carry-in bit of the next full adder.

Purpose: This module is used to add the 16-bit input registers (A and B) together using the carry-ripple method.

Module: lookahead4 from *lookahead4.sv*

Inputs: [3:0]A, [3:0]B, Cin

Outputs: [3:0]S, Cout, Pg, Gg

Description: This is a carry-lookahead unit that will add 4-bit inputs together. It will calculate the carry bit of each add operation separately from the actual add operation, and it will also output the overall propagate and generate bits of the input.

Purpose: This module is used to add 4-bit inputs together using the carry-lookahead method. It was created to help parallelize the load for the 16-bit carry-lookahead adder.

Module: carry\_lookahead\_adder from *carry\_lookahead\_adder.sv*

Inputs: [15:0]A, [15:0]B

Outputs: [15:0]Sum, CO

Description: This is a 16-bit carry-lookahead adder implemented in a 4x4-bit hierarchical design. It will calculate the carry bits of each add operation separately from the actual add operation.

Purpose: This module is used to add the 16-bit input registers (A and B) together using the carry-lookahead method.

Module: select4 from *select4.sv*

Inputs: [3:0]A, [3:0]B, Cin

Outputs: [3:0]S, Cout

Description: This is a carry-select unit that will add 4-bit inputs together. It will first calculate all possible carry-bit combinations for every bit after bit 0 and then select the correct carry bit and sum afterwards.

Purpose: This module is used to add 4-bit inputs together using the carry-select method. It was created to help parallelize the load for the 16-bit carry-select adder.

Module: carry\_select\_adder from *carry\_select\_adder.sv*

Inputs: [15:0]A, [15:0]B

Outputs: [15:0]Sum, CO

Description: This is a 16-bit carry-select adder implemented in a 4x4-bit hierarchical design. It will first calculate all possible carry-bit combinations for every bit after bit 0 and then select the correct carry bit and sum eventually.

Purpose: This module is used to add the 16-bit input registers (A and B) together using the carry-select method.

Module: HexDriver from *HexDriver.sv*

Inputs: [3:0]In0

Outputs: [6:0]Out0

Description: This is a converter that takes in a 4-bit hexadecimal number and transforms it into an on-board seven-segment display output.

Purpose: This module transforms a hexadecimal input into the corresponding output signal for the seven-segment display.

## 

# Part 2.e: The Trade-Offs Between the Adders

Every different adder has a different way of adding the values, and thus, they have distinctive strengths and weaknesses in their operations. Since addition is one of the indispensable operations for the digital design, it is crucial to know which adder suits for different situations.

First off, the main advantage of the Carry-Ripple Adder is in its simplicity. This simplicity moreover reduces the total power consumed by the adder. Additionally, because its design is so simple, it uses the least amount of area on the board. However, the main disadvantage of the Carry-Ripple Adder is in its delay of operation due to carry bits. Since the addition of a latter bit must wait for the calculation of the former to complete in order to get the carry bit, the delay of this adder, in the worst case, becomes linear to the number of bits we want to add. Therefore, it may not be optimal for designs that require faster operation or systems that puts emphasis on quality over simplicity.

The situation of Carry-Lookahead Adder is quite the opposite - it puts emphasis on the speed of the operation at the cost of simplicity. Noticeable advantage that Carry-Lookahead Adder provides is that it supplies a faster calculation, able to accomplish the entire addition operation in one clock cycle. The advantage is due to its ability to calculate the carry bits separate from the actual adding operation. The biggest drawback of the Carry-Lookahead Adder is in the exponential growth in complexity and area with each additional bit. This is because of the amount of logic necessary to calculate each carry bit is serial and depends on the past carry bits.

The characteristics and one of the strong points of Carry-Select Adders is in its parallel design. The Carry-Select Adder has the highest frequency out of all the adders we implemented in the experiment, which means that it was able to output the final sum the fastest. This is available due to its capability to precompute all possible sums in parallel. However, a major disadvantage that stems from this is the power consumption. Our implementation of the Carry-Select Adder requires over triple the amount of full adders to calculate the sum, and most of these calculations are either redundant or unused. Furthermore, the area necessary to implement this adder, though less than the Carry-Lookahead Adder, is still more than the Carry-Ripple Adder since we need to accommodate for all of the extra adders. The Carry-Select Adder is also less complex than the Carry-Lookahead Adder because it only requires full adders and multiplexers, but it is more complex than the Carry-Ripple Adder since it does need some additional logic to output the final sum.

# 

# Part 2.f: Documentation of the Performance of Each Adder

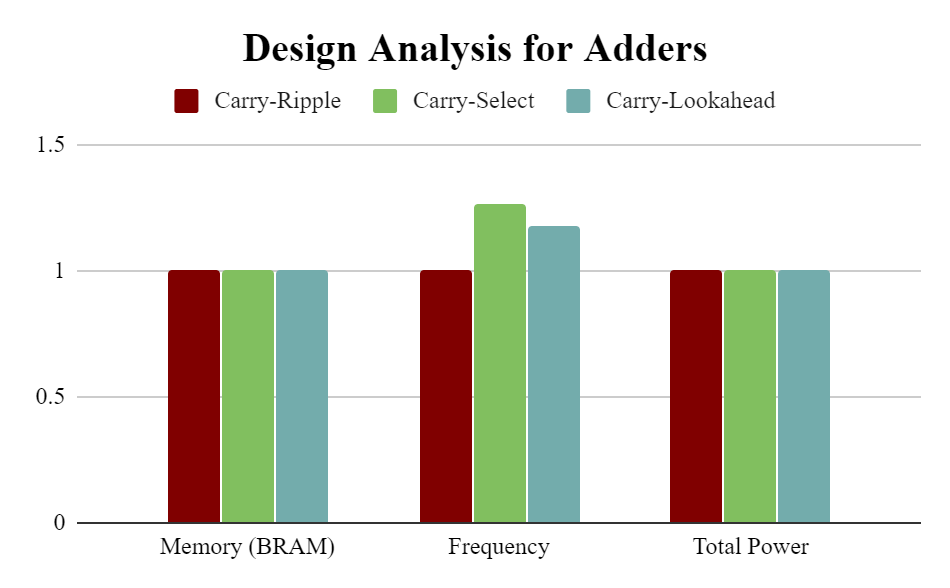
*Figure 11* below shows a simplified design analysis of the three adders implemented in this lab. Because only combinational logic was used, no BRAM was needed for any of the implementations. *Figure 12* shows the normalized performance of each adder, and *Figure 13* is a graph of the normalized data we obtained. Notable features from these figures include the fact that the Carry-Select Adder was both the fastest at adding and the biggest consumer of power in this experiment. Additionally, the Carry-Ripple Adder was the slowest adder but also used the least amount of power in this experiment, which was to be expected. One aspect that we noticed was that the total power did not vary as much between the different adders. We speculate that this is due to the fact that very little logic changes between the adders, so it is understandable that the total power consumption for each adder is similar across all adders.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Carry-Ripple** | **Carry-Select** | **Carry-Lookahead** |
| **Memory (BRAM)** | 0 | 0 | 0 |
| **Frequency (MHz)** | 178.86 | 225.68 | 209.82 |
| **Total Power (mW)** | 139.11 | 139.48 | 139.12 |

*Figure 11: Table of the Performance of Each Adder*

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Carry-Ripple** | **Carry-Select** | **Carry-Lookahead** |
| **Memory (BRAM)** | 1 | 1 | 1 |
| **Frequency** | 1 | 1.261768981 | 1.173096276 |
| **Total Power** | 1 | 1.002659766 | 1.000071886 |

*Figure 12: Table of the Performance of Each Adder, Normalized*



*Figure 13: Plot of the Performance of Each Adder*

# Answers to the Post-lab Questions

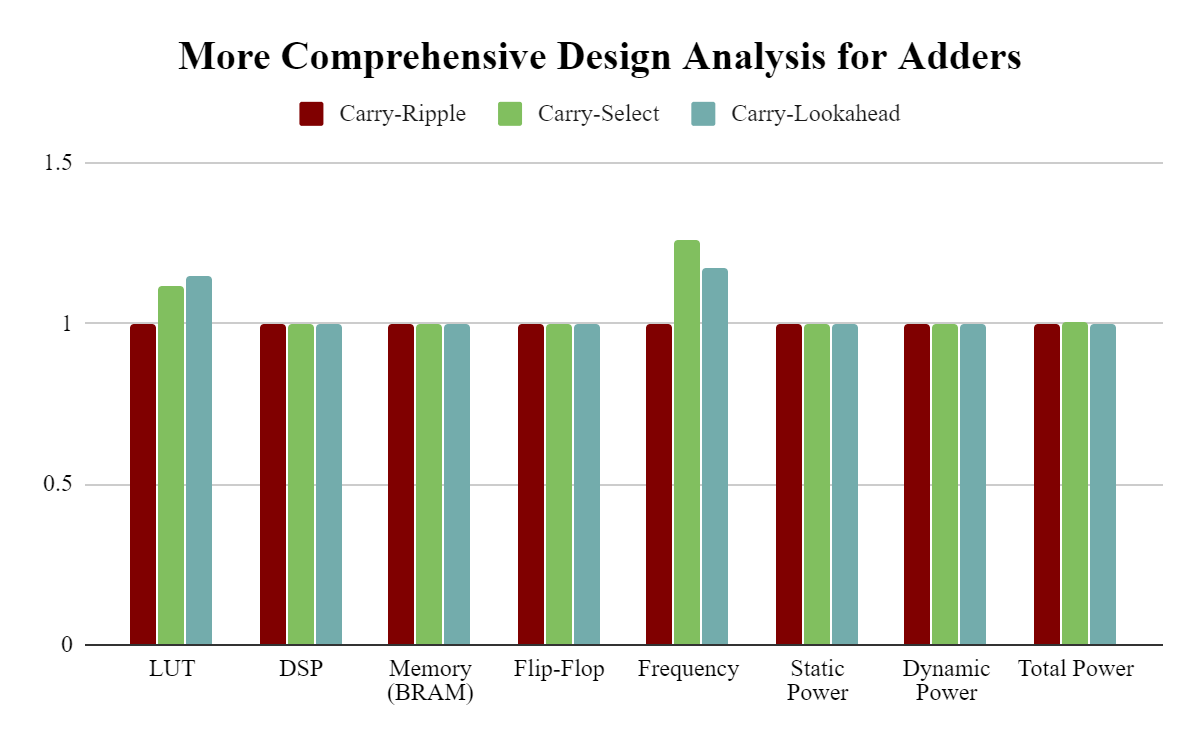
1. For the bit-serial logic processor from this lab, 72 LUTs, 0 bits of memory, and 43 flip-flops were used. In order to compare this data with our TTL design from Lab 3, we will first need to estimate the usage of resources for the TTL design if it were extended to 8-bits. A rough estimate can be made with the help of the data sheets by looking at the functional block diagram of each chip used in the circuit. As such, we likely would have used around 300 LUTs, 0 bits of memory, and 40 flip-flops in our TTL design. Based on these estimates, it is clear that the bit-serial logic processor implementation with the FPGA is superior because it uses much fewer resources than the TTL design. This makes sense because the FPGA is made to optimize the logic so that it uses the minimum resources necessary to achieve the set goal. Additionally, due to the minimal resources used, the implementation we had for this lab will likely run faster than our TTL design since there is less delay between gates.
2. No, we do not believe that the 4x4 hierarchy is the most ideal for the Carry-Select Adder. The reason behind this is in the way the Carry-Select Adder works. The first carry-in bit of the full adder for bit 0 is always known because it will always be zero if we are adding. However, the carry-in bit for any of the later bits is unknown. As such, both possible sums of the adder are calculated in parallel, and the final verdict of which sum to use is decided later. The reason why the 4x4 hierarchy may not be the most ideal design is the issue that arises when chaining the 4-bit Carry-Select Units together. By doing so, the number of adders used doubles in numbers without much increase in benefits since the sums are just getting recalculated in each Carry-Select Unit. To be exact, in a 16-bit adder, the 4x4 hierarchy requires 49 full adders in total while in the flat design, only 31 are needed. As one can see, a hierarchy design requires much more adders than ones without. However, the benefit of the hierarchy design is that it allows us to parallelize the computation of each unit. In order to design the ideal hierarchy on the FPGA, much more information is needed. The key information that we are interested in would be the time delay with each unit added. Due to the ability of FPGA to parallelize processes, the time it takes for adders to calculate the final sum would likely be less in the hierarchy design than in the flat design. We would conduct this experiment by creating Carry-Select Units that would calculate various powers of 2 bits and chaining different combinations of them together in a hierarchy design to find the implementation with the best time to resource ratio.
3. Yes, the resource breakdown comparison from the plot makes sense. The raw data from Quartus can be found in *Figure 14*. *Figure 15* shows the normalized data for the resources used, and this is plotted in *Figure 16*. Since no digital signal processing (DSP) blocks nor any block memory (BRAM) were used in any of the adders, all adders should have the same amount of these resources used, which is zero. Because there is only combinational logic in each adder implementation, it also makes sense that there is no dynamic power in any of the circuits and that the implementations dissipate the same amount of static power. Furthermore, all implementations have the same amount of flip-flops from the wrapper portion of the design. All of our resource breakdowns comply with the theoretical design expectations. The maximum operating frequency of Carry-Lookahead Adder is higher than the Carry-Ripple Adder, and the Carry-Select Adder has the highest frequency of them all. For the Look-Up Tables (LUTs), it also makes sense that the Carry-Ripple Adder has the least amount because there is little background logic in the adding operation. On the other hand, the Carry-Lookahead Adder has the most LUTs because it has to do a lot of extra logic to calculate the carry-in bits asynchronously to the adding operation. For the total power consumed, it makes sense that the Carry-Select Adder uses the most because it is trying to run all of its adders in parallel while the Carry-Ripple Adder uses the least because it has the least amount of operations to calculate the sum.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Carry-Ripple** | **Carry-Select** | **Carry-Lookahead** |
| **LUT** | 114 | 127 | 131 |
| **DSP** | 0 | 0 | 0 |
| **Memory (BRAM)** | 0 | 0 | 0 |
| **Flip-Flop** | 105 | 105 | 105 |
| **Frequency (MHz)** | 178.86 | 225.68 | 209.82 |
| **Static Power (mW)** | 98.5 | 98.5 | 98.5 |
| **Dynamic Power (mW)** | 0 | 0 | 0 |
| **Total Power (mW)** | 139.11 | 139.48 | 139.12 |

*Figure 14: Table of Comprehensive Design Analysis for Adders*

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Carry-Ripple** | **Carry-Select** | **Carry-Lookahead** |
| **LUT** | 1 | 1.114035088 | 1.149122807 |
| **DSP** | 1 | 1 | 1 |
| **Memory (BRAM)** | 1 | 1 | 1 |
| **Flip-Flop** | 1 | 1 | 1 |
| **Frequency** | 1 | 1.261768981 | 1.173096276 |
| **Static Power** | 1 | 1 | 1 |
| **AweDynamic Power** | 1 | 1 | 1 |
| **Total Power** | 1 | 1.002659766 | 1.000071886 |

*Figure 15: Table of Comprehensive Design Analysis for Adders, Normalized*



*Figure 16: Plot of the Comprehensive Design Analysis for Adders, Normalized*

# Conclusion

## Description of any Bugs and Countermeasures Taken

We had several bugs when designing our adders. The main bug we encountered was one with the compiler on Quartus. For some reason, our code was corrupted during compilation, so it caused some weird behavior when the code was transferred to the FPGA board. Most notably, one set of our adder units in our Carry-Select Adder always outputted a carry-out bit for all the bits in the unit despite having been coded correctly. Recompiling the adder and swapping between different adders did not solve this issue. This issue was finally resolved when we rewrote the instantiation of the adder units in question within *carry\_select\_adder.sv*. Although visually the code looked the same before and after we rewrote the lines, our Carry-Select Adder suddenly worked after this change. We are not entirely sure why it compiled correctly after this change, but we think it might have something to do with the difference in line endings between Linux and Windows because there were other groups that had issues with the line endings in this lab.

We also had a few bugs with our Carry-Lookahead Adder. This was coded incorrectly because we had the formulas for propagate and generate flipped. We also forgot to adjust a value when copying the formula over for one of the carry bits. Both of these bugs were fixed through careful examination of the code and cross-checking with the given documentation.

Although not truly a bug, we did have some issues with compiling on Quartus itself. Our virus scanners thought that Quartus was ransomware, so it prevented our code from fully compiling at first. This issue was resolved by ignoring the warning and hoping that our laptops do not die.

## Ambiguities and Difficulties in the Lab Manual

The main issue we found in the lab manual was the graph given on page 4.8. This graph has both misleading and incorrect information and stating that our resulting graph should resemble the given one can cause unnecessary anxiety among students. Because there was no memory used in any of the adders, there should not be any variance in the data for the memory used by each adder. Furthermore, the data used for the frequency is completely incorrect because it shows that the Carry-Ripple Adder has the highest frequency, which would defeat the whole purpose of this lab because we were trying to find other alternatives to combat the carry bit delay introduced by the Carry-Ripple Adder. In reality, the Carry-Ripple Adder has the lowest frequency among all of the adders implemented, which is to be expected. Lastly, the difference in total power is greatly exaggerated in the given graph, which can cause students to worry about whether or not their adders are implemented correctly.

## Additional Summary

Overall, we believe that we have achieved the goals of this experiment. Through this lab, we had the opportunity to explore SystemVerilog through our edits on the bit serial logic processor and through our implementation of the adders. We also learned about the Carry-Ripple, Carry-Lookahead, and Carry-Select implementations for adders and were able to discover the trade-offs and benefits of each. Furthermore, this lab has taught us a great deal about the advantage of using an FPGA board instead of just chips due to the advantages it provides to us in terms of optimization.