ECE 385 Experiment 3: Lab Report

Spring 2023
Eshaan Tibrewala (eshaant2) and Shivam Patel (shivamp6)
TA: Shitao Liu

Introduction

Experiment 3 was a one week experiment, and consisted of designing and implementing three different kinds of adders: carry ripple adder (CRA), carry-lookahead adder (CLA), and carry-select adder (CSA). Each one simply adds two different binary values, and can be modified for different lengths depending on the inputs. The purpose of this lab was to also transition from the the traditional breadboard TTL to RTL design on the FPGA. For this lab, we focused on implementing 16-bit adders. The CRA design is the most simple and straightforward, but has a long computational drawback. The CLA and CSA were designed in a 4x4 hierarchical manner. A full adder consists of inputs **A** and **B**, which are the bits being added, a C_{in}, carry in bit, a C_{out}, carry out bit, and the resulting **S**, which is the sum bit. Each adder in this lab uses a full adder on a basic level, but the functionality and path of the bits differ based on the adder, allowing adders such as the CLA and CSA to compute larger bits in a more efficient and effective manner.

Adders

Full Adder

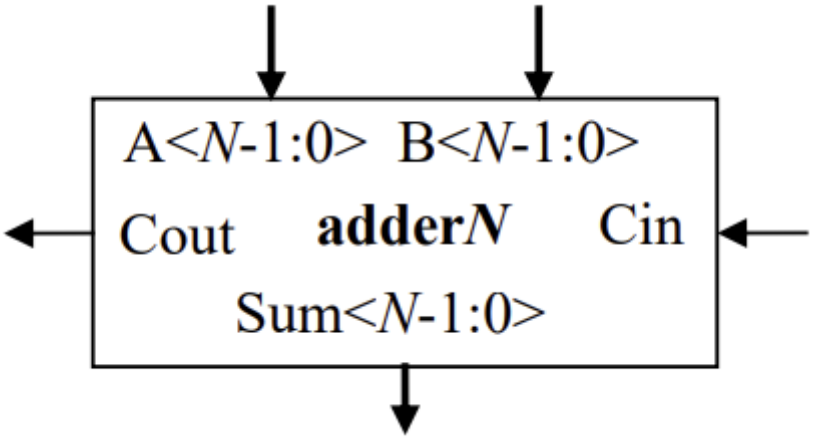

Figure 1: Full-Adder Block Diagram

A full adder is at the base of all three adders we built in this lab. Each one utilizes the unit differently, allowing it for tradeoffs in time and performance. So to understand how to implement a CRA, CLA, and a CSA, it is important to understand the functionality of a full adder. As seen in Table 1 and Figure 1, a full adder takes three inputs and has two outputs. The inputs are A and B, the bits being added, and C_{in}, the carry in bit, which comes from the previous full adder unit. The outputs of the full adder are S, which is the sum, and C_{out}, which is the carry out bit that will serve as the carry in for the next full adder.

Table 1: Inputs and Outputs of a Full Adder

INPUTS	OUTPUTS
A	C_{out}
B	S
C_{in}	

From basic decimal addition, we start at the ones place, and work our way out. If we get a sum grater than 9, we carry out the digit to the next position, and factor that into our addition for the next digit. An example of this addition is shown below. Notice how $3 + 8 = 11$, so we keep the least significant 1 and carry out the other 1 to the next digit, which then becomes $3 + 4 + 1 = 8$. Our final sum therefore is 81.

Math Block 1: Simple Decimal Addition

$$\begin{array}{r} 1 \\ 43 \\ +38 \\ \hline 81 \end{array}$$

A full adder works similar to the decimal addition example shown above. The adder takes 1 bit from A and 1 bit from B, and adds them together with the C_{in} bit. If at least two of the bits are high, the full adder produces a C_{out} bit with the value of 1, which serves as the carry in for the next adder. This process is repeated as the binary digits move from least significant bit to the most significant bit of A and B. Table 2 shows a truth table for a full adder, and how the output signals are produced.

Table 2: Truth Table for Full Adder

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$(1)\ \ S = A \oplus B \oplus C_{in}$$
$$(2)\ \ C_{out} = (A \wedge B) \vee (B \wedge C_{in}) \vee (A \wedge C_{in})$$

Equations 1 and 2 show the outputs of a full adder are derived from the inputs.

Carry Ripple Adder

A carry ripple adder is the most straightforward to implement. A N-bit CRA can be created by cascading N full adders put together where the C_{out} from one adder becomes the C_{in} for the next adder. Our design of a CRA was for 16-bits, so we simply called 16 full adders and discarded the carry out bit from the most significant bit to avoid overflow.

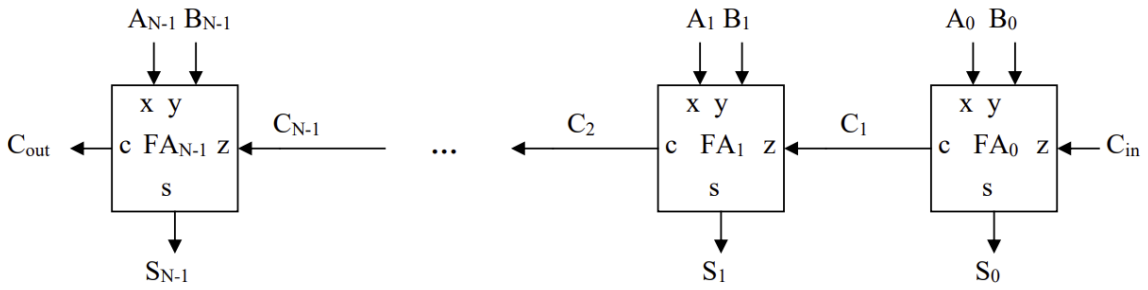

Figure 2: N-bit Carry Ripple Adder

Figure 2 shows an upper level block diagram for a CRA. As the figure reflects, a CRA consists of multiple full adders. The final sum of the adder is reflected by the S bits that are outputted from each adder. However, a design like this comes with certain drawbacks. One major drawback is that the S output for full adders is not accurate until the carry in bit is accurate. For this to happen, the CRA must wait for the carry signal to cascade down the bits until every S output reflects the correct value from a correct carry in. This adds time to the speed of the adder, so essentially a 32-bit adder would take twice as long as a 16-bit adder and so on. Therefore, the CRA scales linearly to the number of bits, and so the run time of a typical CRA is $O(n)$. So while the CRA is the easiest to implement, it is also the slowest adder available.

Carry Lookahead Adder

One way to reduce the computation time that comes with a CRA is to utilize a carry lookahead adder. While the 16-bit carry ripple adder was made up of 16 full adder in a cascading manner, the CLA was implemented in a 4x4 hierarchical manner, meaning that we first built a 4 bit carry lookahead adder, and then chained them together in order to reduce the performance time of the adder. Whereas the propagation delay of the CRA increases with the number of bits, the CLA eliminates this delay by computing the carry-out bits in a parallel manner to the rest of the computation. Instead of waiting for the actual carry in values to arrive from the lower bit neighbor carry out, the CLA uses

generation and propagation logic. The CLA does this by using its the inputs that are immediately available to predict the carry out bit of the adder. For a full adder, these immediate bits are A and B, so a carry out generated when both inputs are high. A generated bit is represented using G, and the equation for generation is therefore $G = A \wedge B$. The CLA has calculates propagation logic P, which represents the possibility of carry out bit. Using the immediate inputs, this happens when either A or B is high, but the adder still needs the C_{in} bit to determine the outputs. Propagation can therefore be represented as $P = A \oplus B$. The CLA then uses these P and G signals to compute C in a parallel fashion, making it faster then the carry ripple adder. So to create the final 4x4 hierarchical design, we first designed the single 4-bit CLA shown in Figure 3. This was achieved by taking four full adders and modifying them to output the P and G bit. Those bits are then fed into the carry look ahead unit, which generates $C_0, C_1, C_2,$ and C_3. These signals can be represented exclusively in terms of P and G, make the computation time of the CLA faster. These signals essentially serve as the carry in for each full adder unit in the 4-bit carry lookahead adder. The general equation for calculating these C bits is that for a given C_i, $C_{i+1} = G_i + (P_i \bullet C_i)$, where C_i can also be expressed in terms of P and G.

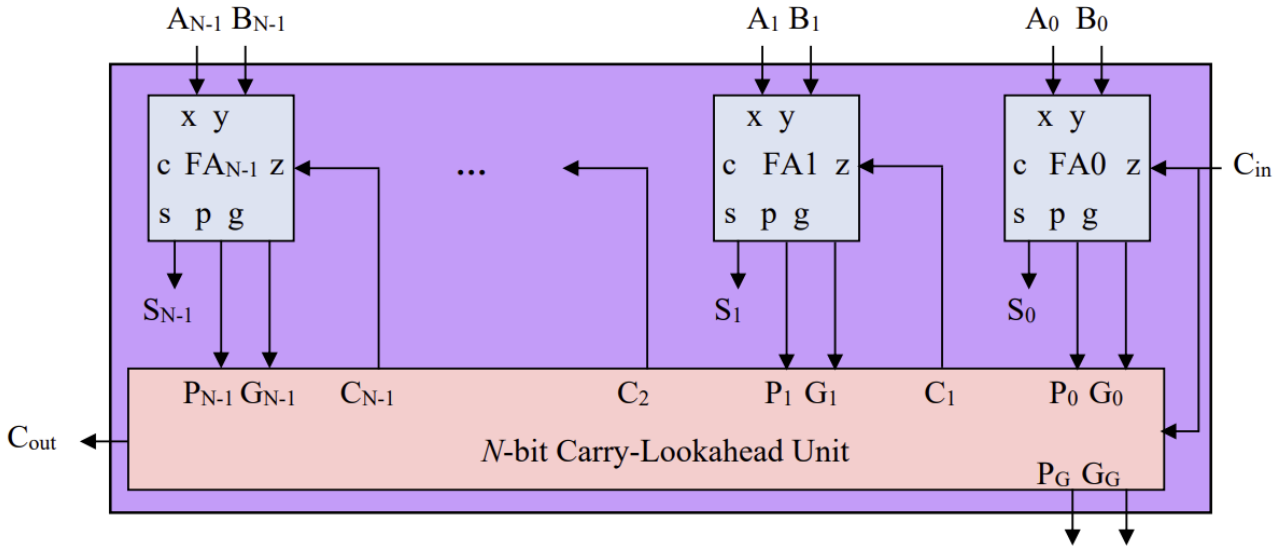

Figure 3: N-bit Carry-Lookahead Adder

Math Block 2: Equations for C_0, C_1, C_2, C_3

$$(1)\ C_0 = C_{in}$$
$$(2)\ C_1 = C_{in} \bullet P_0 + G_0$$
$$(3)\ C_2 = C_{in} \bullet P_0 \bullet P_1 + G_0 \bullet P_1 + G_1$$
$$(4)\ C_3 = C_{in} \bullet P_0 \bullet P_1 \bullet P_2 + G_0 \bullet P_1 \bullet P_2 + G_1 \bullet P_2 + G_2$$

Using the general block diagram shown in Figure 3 and using the pattern of the carry out equations, you can create an arbitrarily large N-bit carry lookahead adder that has a $O(1)$ run time. Theoretically, the CLA has two gate delays to generate thew C bits, however as you keep expanding the CLA, the number of necessary gates start to increase because gates not not infinitely wide. Therefore just using the standard CLA expansion can be expensive and impractical when operating on large numbers of bits. One way to solve this problem is to create a hierarchical model as shown in Figure 4. This allows you to create smaller CLA modules, and then combine multiple ones to compute larger number of bits. The two main ways to do this is by cascading the C_{in} to C_{out} as with ripple carry adders, or using another 4-bit generator to increase speed. For our lab, we implemented this second option.

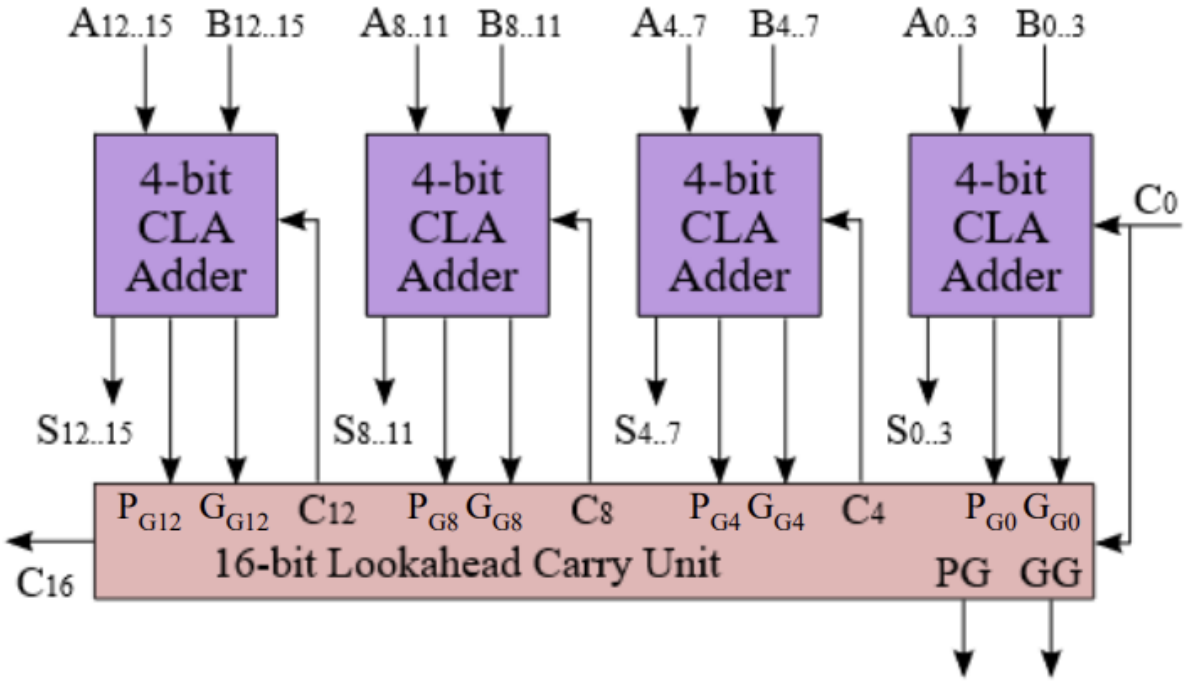

Figure 4: 4x4 Hierarchical Carry-Lookahead Adder

In the 4x4 design, the 16-bits of A and B are calculated in groups of 4 bits. In Figure 3, notice that the carry lookahead unit generates two extra bits: P_G and G_G. This additional outputs are referred to as group propagate and group generate. The equations for P_G and G_G are:

$$P_G = P_0 \bullet P_1 \bullet P_2 \bullet P_3$$
$$G_G = G_3 + G_2 \bullet P_3 + G_1 \bullet P_3 \bullet P_2 + G_0 \bullet P_3 \bullet P_2 \bullet P_1$$

When combined in the 4x4 hierarchical design, these signals are used to calculate the carry bits to next 4-bit CLA adder in the same manner as they were calculated in the individual 4-bit CLAs. Using this method instead of traditional cascading method allows the carry bits to be computed in parallel, saving even more performance time when implementing higher bit CLAs such as the 16-bit adder in the lab. This hierarchical design can be extended to compute even higher bits by adding another lookahead unit at the next level, and then using the G_G and P_G bits to compute the carry bits into the next CLA.

Carry Select Adder

The carry select adder provides another way to speed up the computation time of the carry bit. It consists of two full adders, and a 2-to-1 multiplexer. The CSA requires two full adders because it precomputes both possibilities for C_{in}, one where the bit is high and one where the bit is low. Both results are then directed to a 2-to-1 MUX which uses the C_{in} bit as the select bit to choose the correct output for the adder. However, the lowest group of the CSA only uses one adder because the carry in bit is assumed to be 0, and no additional computation or multiplexer is required.

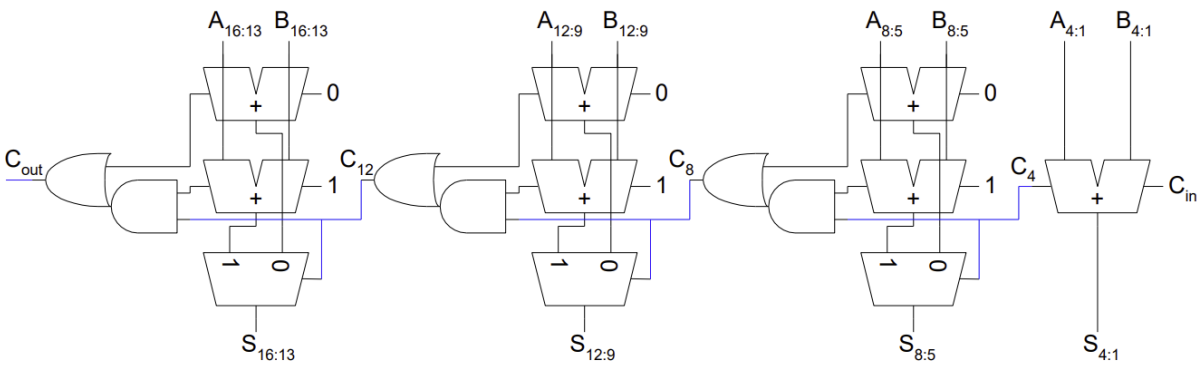

Figure 5: 16-bit Carry Select Adder Block Diagram

The CSA structure allows the adder to precompute the bits, and have both possible answers ready until the carry in bit arrives to the unit. However, because of this structure, the CSA requires twice as many full adder units and additional multiplexers in order to reduce the carry propagation delay. Since the carry in bits don't change, the CSA and calculate the sum in parallel. All of these sums are then ready to be selected based on the input to the multiplexer. As shown by the block diagram in Figure 5, the select bit for the multiplexer of the next CSA unit is calculated by $C_1 \bullet C_{i-1} + C_0$, where C_1 is the carry out generated by the full adder where carry in is 1, C_0 is the carry out generated by the full adder where carry in is tied to 0, and C_{i-1} is the carry out generated by the last CSA unit.

Summary of .SV Modules

Adder_toplevel.SV

Code Block 1: module adder_toplevel

```
module adder_toplevel  (input Clk, Reset_Clear, Run_Accumulate,
             input [9:0]     SW,
             output logic [9:0]  LED,
             output logic [6:0]  HEX0,
                       HEX1,
                       HEX2,
                       HEX3,
                       HEX4,
                       HEX5
             );
```

Code block 1 shows all the inputs and outputs of the top level module for this lab. `Adder_toplevel` drives all the adders and instantiates all the other modules and then outputs it to the FPGA. The module has all three adders instantiated, so we just had to uncomment the adder we want to test in order to use the adder. When `Reset_Clear` is high, the instantiated adder is reset and its inputs are cleared. When `Run_Accumulate` is

high the adder operation is executed. The purpose is to run the whole program and perform the addition.

Control.SV

Code Block 2: module control

```
module control (input Clk, Reset, Run,
                    output logic Run_O);
```

This is the control unit of the adders. The control unit is the same for all the adders, so it has the same inputs and outputs. `Reset` simply resets the FSM and returns the control unit to the start state. `Run` executes the FSM and tells the adder to complete its operation. The adder actually computes when `Run_O` is high.

Reg_17.SV

Code Block 3: module reg_17

```
module reg_17 ( input   Clk, Reset, Load,
          input         [16:0] D,
               output logic   [16:0] Data_Out);
```

The register module instantiates the 16-bit registers used by the adders. `Reset` sets the data in the register to 0, and `Load` allows the program to load in the input from the switches. 16-bit `D` is loaded into the register with the `Load` signal is high. The module outputs `Data_Out` which is the value stored in the register. The purpose of this module is to create the registers and to hold the accumulated sum.

Mux2_1_17.SV

Code Block 4: module mux2_1_17

```
module mux2_1_17    (    input                S,
                         input                   [15:0] A_In,
                         input                [16:0] B_In,
                         output logic        [16:0] Q_Out);
```

This module instantiates a 2-to-1 MUX. It takes one select bit S, which selects between the sum of A and B or B. The purpose is to send the data to the register. When `S` is high the module sends the sum of A and B to the register and when `S` is low the module stores B into the register.

HexDrive.SV

Code Block 5: module HexDriver

```
module HexDriver (input  logic [3:0]  In0,
                  output logic [6:0]  Out0);
```

This module takes a 4-bit input and then extends it to 7 bits so that the value can be displayed on the FPGA LEDs.

Testbench.SV

Since this is the testbench, there are no inputs or outputs. All the logic is local variables, and the purpose of the module is to call `adder_toplevel` module so that we could test our circuit with custom inputs.

Ripple_Adder.SV

Code Block 6: module ripple_adder, module full_adder

```
module ripple_adder
(
  input  [15:0] A, B,
  input         cin,
  output [15:0] S,
  output        cout
);

module full_adder (input A, B, cin, output S, cout);
```

This module creates the ripple carry adder. The `full_adder` module instantiates a single 1-bit full adder. `A` and `B` are the bits being added, `cin` is the carry in bit, `S` is the sum, and `cout` is the carry out bit. The purpose is to create a full adder that we could then use to implement our other adders.

The `ripple_adder` module essentially has the same inputs and outputs as the `full_adder`, but is extended to 16-bits. The purpose of the module is to essentially use the smaller `full_adder` module to create our 16-bit ripple carry adder by instantiating a full adder 16 times, once for each input bit.

Lookahead_Adder.SV

Code Block 7: module lookahead_adder, module carry_lookahead

```
module lookahead_adder (
  input  [15:0] A, B,
  input         cin,
  output [15:0] S,
  output        cout
);

module carry_lookahead(input[3:0] A, B, input cin, output PG, GG,
output[3:0] S);
```

This module creates our lookahead adder. Our lookahead adder was created in a 4x4 hierarchical manner, so we created a smaller 4-bit carry lookahead adder in this module. `carry_lookahead` takes a 4-bit inputs A and B, 1-bit carry in, and outputs `PG` , `GG`, and `S`, which are our group propagate, group generate, and sum signals. The purpose is to break the 16-bit lookahead adder into smaller groups so that we can improve performance and avoid a cascading design.

The `lookahead_adder` module instantiates the actual adder, and takes in 16-bit inputs A and B, and 1-bit carry in. A and B are the numbers being added by the adder. Outputs `S` and `cout` represent the 16-bit sum output, and the 1-bit carry out. The purpose of this module is to instantiate four 4-bit `carry_lookahead` modules so that the 4x4 hierarchical design can be implemented.

Select_Adder.SV

Code Block 8: module select_adder

```
module select_adder (
  input  [15:0] A, B,
  input         cin,
  output [15:0] S,
  output        cout
);

module carry_ripple(input [3:0] A, B,
                    input cin,
                     output [3:0] S,
                    output cout);
```

This is a 16-bit select adder module. It takes in 16-bit values A and B, which is the data to be computed, and outputs a 16-bit sum and a 1-bit carry out. There is also a `carry_ripple` module, which handles 4-bit inputs. The purpose of the `carry_ripple` module is to help create the overall design of the select adder. By making it a ripple adder, we can instantiate it based on tied carry in bits so that the outputs of our `carry_ripple` module ultimately function as the inputs to the MUX in the select adder. The purpose of the overall `select_adder` module is to instantiate the `carry_ripple` module multiple times with tied carry in bits, so that the multiplexers can select the correct values for the sum once the carry in bit arrives to the multiplexer.

Block Diagrams

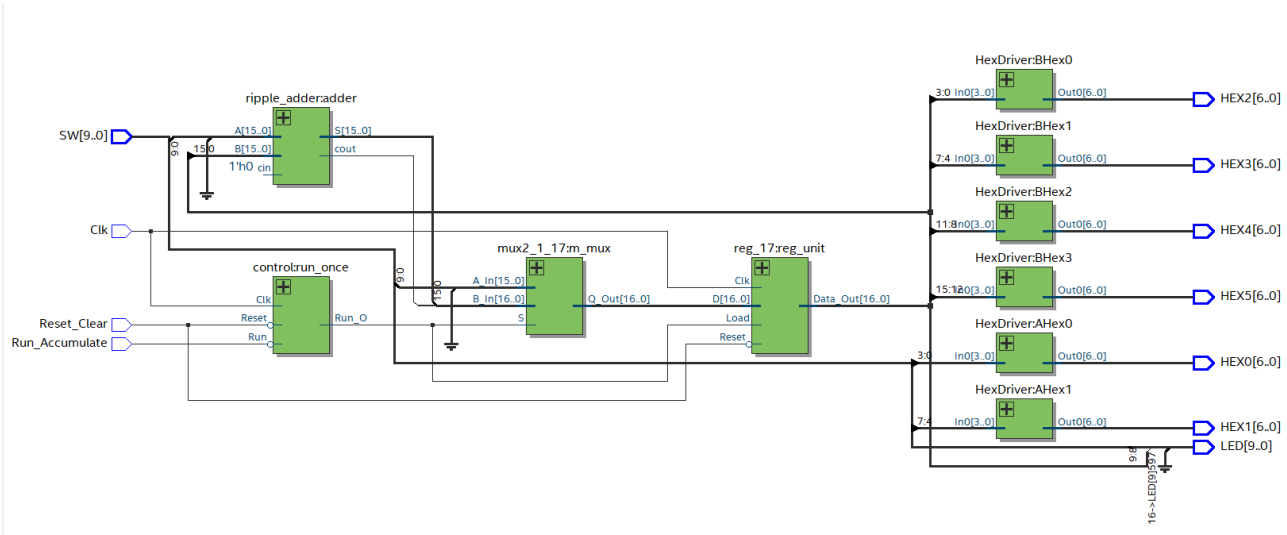

Figure 6: CRA Upper Level Block Diagram

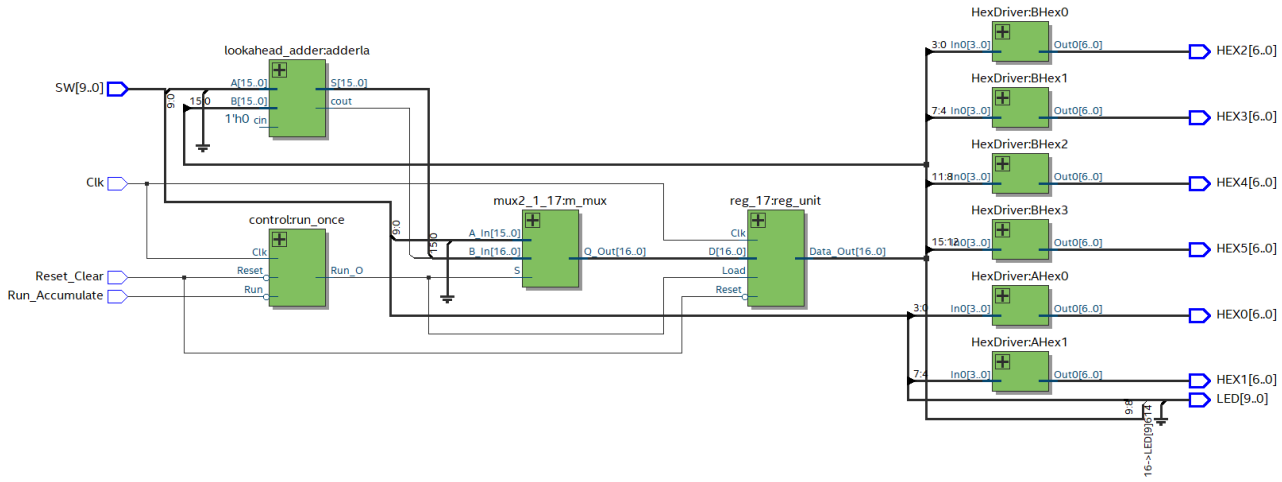

Figure 7: CLA Upper Level Block Diagram

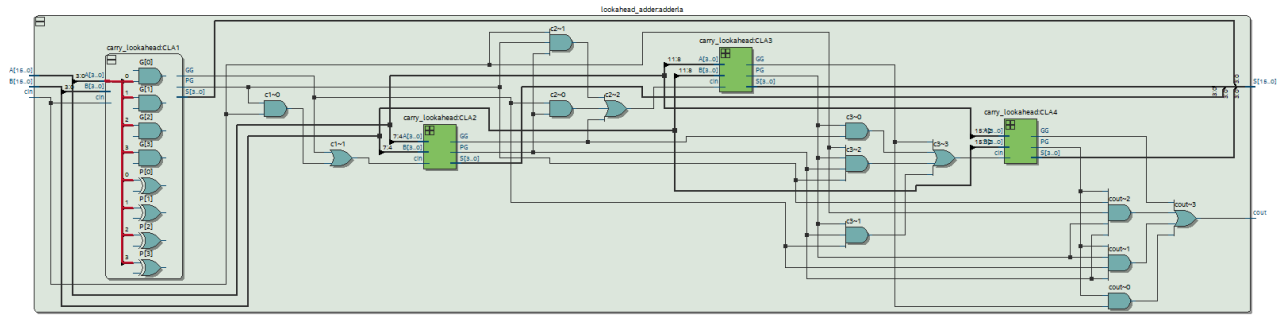

Figure 8: Single CLA Block Diagram Chained Together

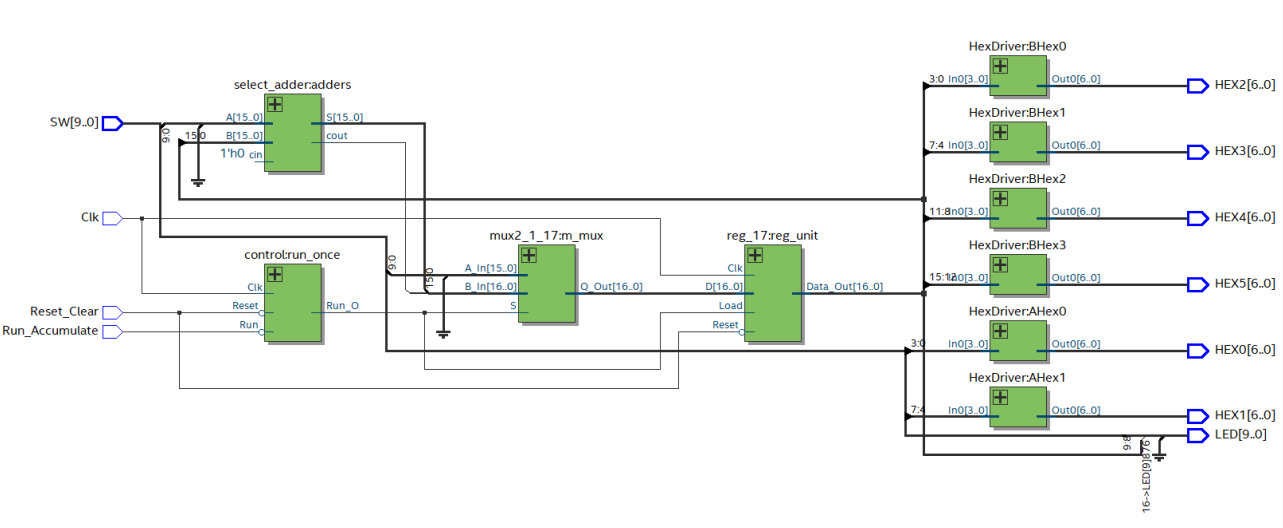

Figure 9: CSA Upper Level Block Diagram

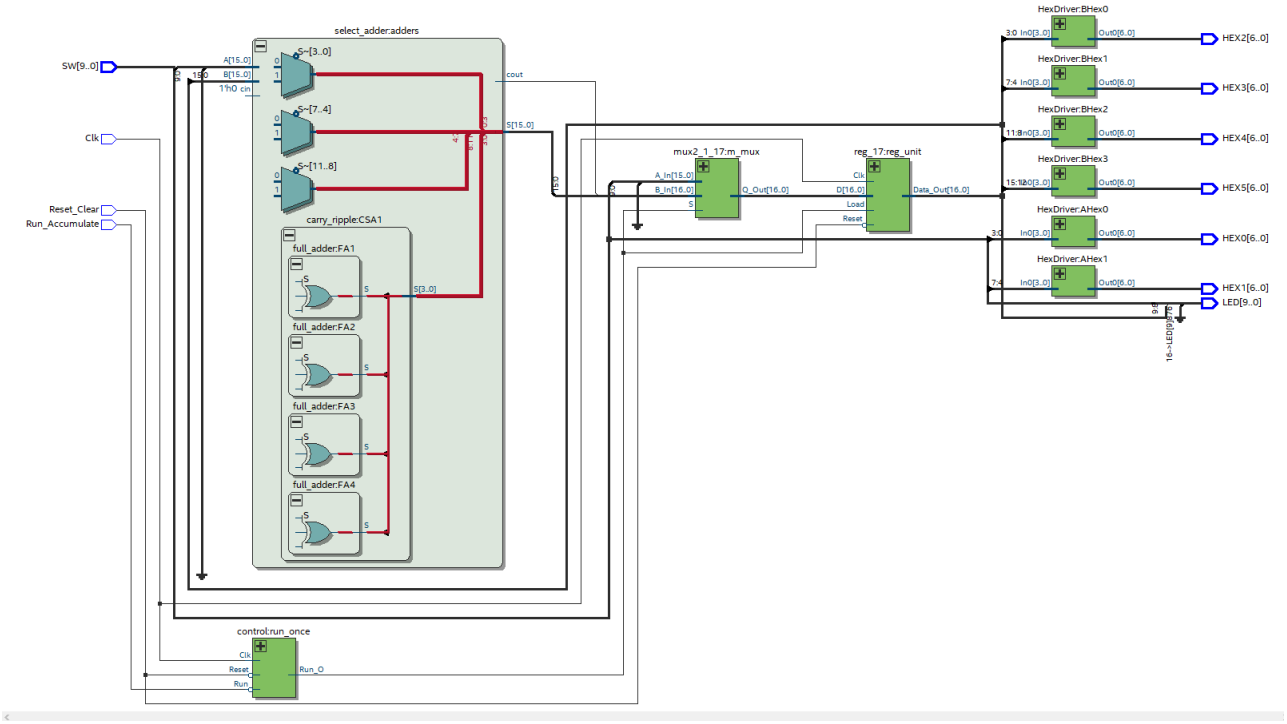

Figure 10: CSA with Single Carry Ripple Adder Block Diagram

RTL Simulation

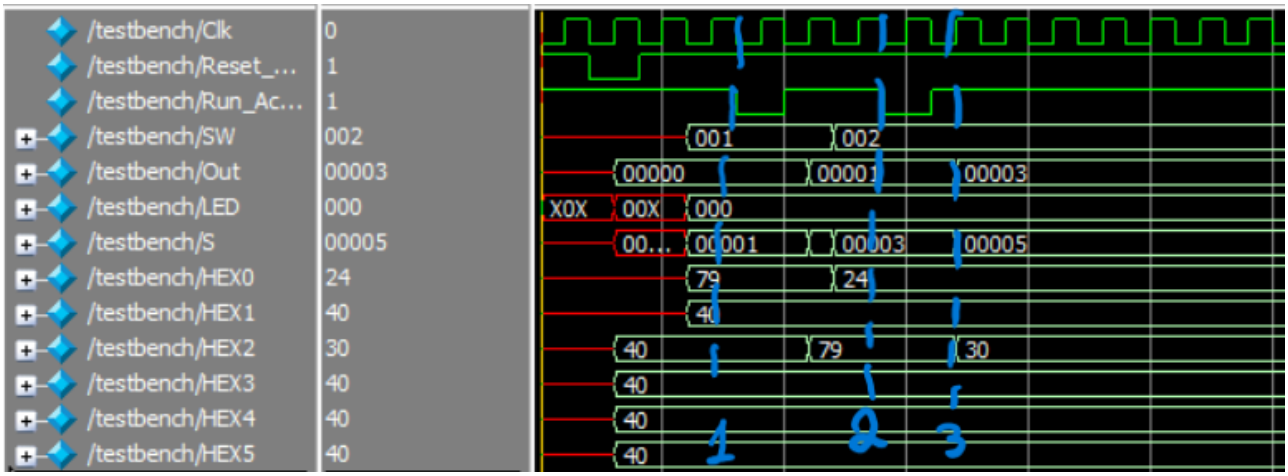

Figure 11: Annotated RTL Simulation of Adders

Figure 11 shows the annotated RTL diagram of the adder. For this simulation, we used the ripple carry adder. At Line 1, you can see that the switch holds `4'h0001` and `Run_Accumulate` is toggled. At the next positive edge, `4'h0001` is displayed as the `Out` value. At Line 2, the `SW` holds `4'h0002` and `Run_Accumulate` is toggled again. This time,

`Out` displays `4'h0003`, which is the correct value of the simulation, since $2 + 1 = 3$, which is exactly what the simulation shows.

Performance Documentations

Table 3: Design Analysis Table

	CARRY-RIPPLE	CARRY-SELECT	CARRY-LOOKAHEAD
Memory (BRAM)	1.000	1.000	1.000
Frequency	1.000	1.044	1.015
Total Power	1.000	1.001	1.004

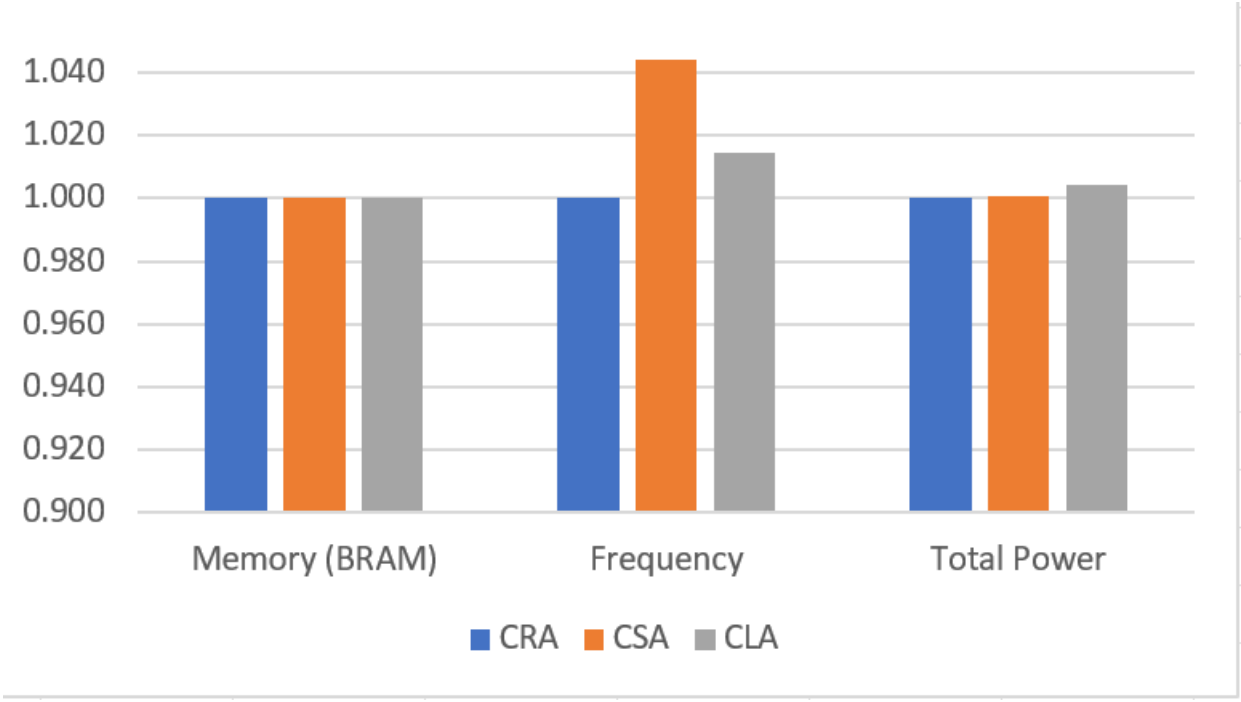

Figure 12: Design Analysis Plot

Complexity and Performance Tradeoffs

We can compare the areas from our three adders by looking at how many Look-up-tables (LUTs) they used. While the Carry Ripple adder used 79 LUTs, the Carry Select Adder and the Carry Lookahead adders used 90 and 83 LUTs respectively. This may not seem like a large difference. Still, it displays how our CLA and CSA use more gates and space, and depending on the number of bits being added (16 in our case), the difference in area between the adders at a higher level would significantly increase if we add more bits changing our 4x4 hierarchy to something larger. If we were to add 32 bits, for example, we could use a 4x8 hierarchy but in doing so the margin of difference in LUTs between the three adders becomes larger and this pattern continues if we were to add even more bits.

Out of the three adders, the CRA is the least complex of the three as it is conceptually the simplest with 16 full adders linked together in series. As we move on to the next adder-- the Carry Lookahead adder-- we can see that we are improving the implementation of adding 16 bits by first calculating the generating and propagating logic in each full adder and by chaining each group of four full adders together in a 4x4 hierarchy we only had to look at the Carry-in and Carry-out bits of each group significantly increasing the speed in comparison to the CRA where we had to wait for the output of the prior full adder to input into the next full adder. Finally, the CSA has a significantly higher complexity than both the CRA and CLA as it accounts for both possible carry in bits; however, where the Carry Select Adder makes up for its high complexity with its incredible efficiency in comparison to the other two adders which we can see by comparing the frequencies of the adders. The Carry-select has the highest frequency of 67.31 MHz whereas the CLA and CRA have frequencies of 65.41MHz and 64.47MHz respectively. After taking a look at the frequencies we can conclude that increasing the complexity of the adder significantly improved the speed of the computation. Similar to the area tradeoffs, if we were to add more bits and change our hierarchies accordingly, then the complexities will increase and thus the margin of difference between the frequencies of the adders will grow as we add more bits on a higher level since the more bits we add, the more we can see the difference in efficiency between our adders.

Critical Path Analysis (Extra Credit)

Critical Path Analysis Carry-Ripple Adder

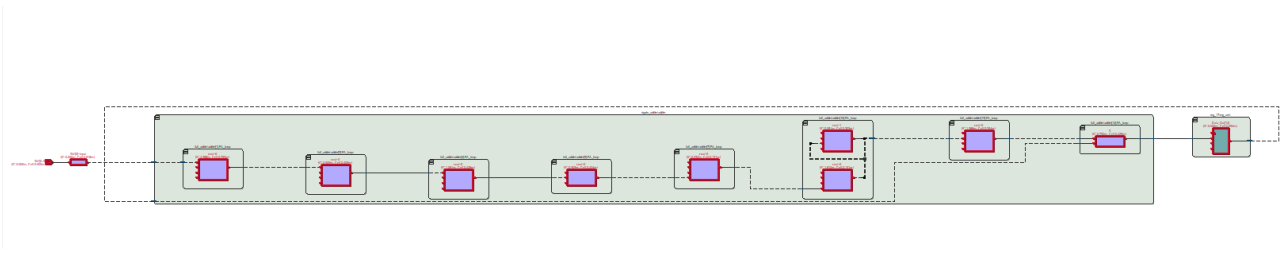

Figure 13: Critical Path Analysis of CRA

From this path analysis of the CRA, we can see that the inputs from the switches enter the `ripple_adder` module. Within the overall module, the signals travel through smaller instantiations of the `full_adder` submodules. This path is consistent with our theoretical understanding of the CRA because it shows that we made a cascading model where the carry out bit from one full adder module becomes the carry in bit for the next module. Ultimately, we used 16 full adders in our implementation to successfully create the 16-bit ripple carry adder.

Critical Path Analysis Carry-Lookahead Adder

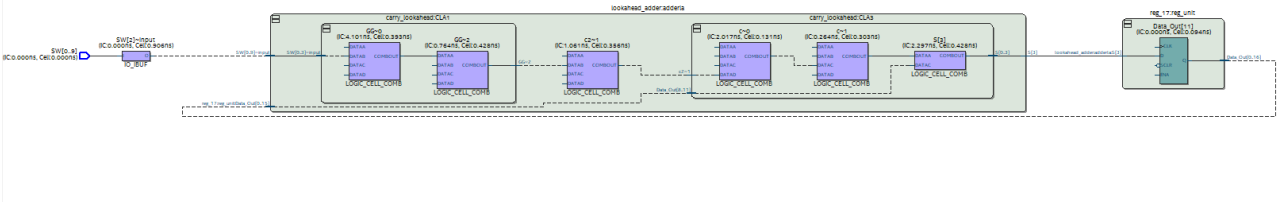

Figure 14: Critical Path Analysis of CLA

From this path analysis of the CLA, we can see that the inputs from the switches enter the `lookahead_adder` module. Within the overall module, the signals travel through smaller `carry_lookahead` submodules. This path is consistent with our theoretical understanding of the CLA because we created in a hierarchical manner, so we combined smaller, 4-bit look ahead modules to create the 16-bit module.

Critical Path Analysis Carry-Select Adder

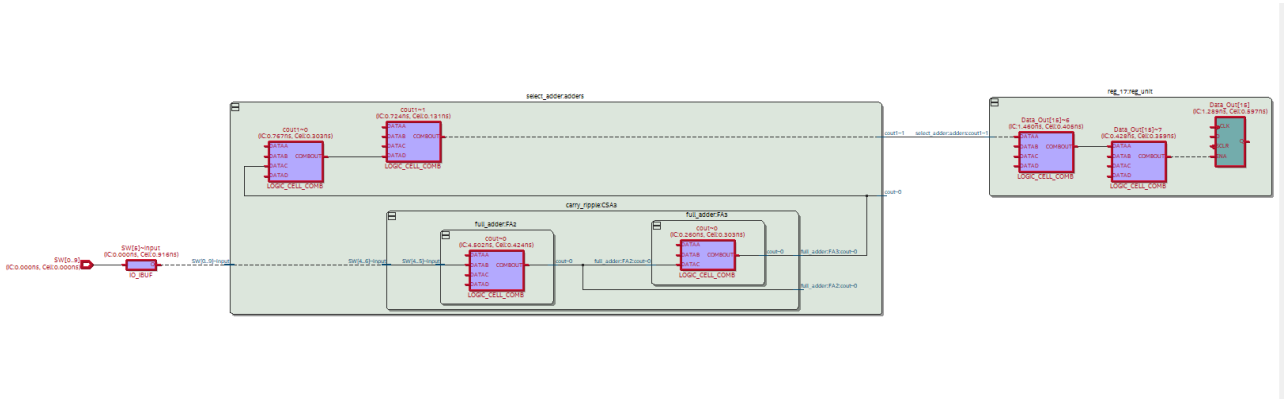

Figure 15: Critical Path Analysis of CSA

From this path analysis of the CSA, we can see that the inputs from the switches enter the `select_adder` module. Within the overall module, the signals travel through smaller `carry_ripple` submodules. The ripple adder module is made up of full adders, which as we know is the base level of any adder. We also see that there is separate logic blocks for the `cout` logic, which is consistent with our understanding of the CSA because we utilize MUXs to implement the adder and calculate the carry bits separately.

Post Lab Questions

Question 1

In the CSA for this lab, we asked you to create a 4x4 hierarchy. Is this ideal? If not, how would you go about designing the ideal hierarchy on the FPGA (what information would you need, what experiments would you do to figure out?)

In this lab, we were tasked with creating a Carry-Select-Adder to add two 16-bit inputs in a 4x4 hierarchy and for the task we were provided, a 4x4 hierarchy may not be ideal. In our logic, we connected seven Carry Ripple Adders with various 2-to-1 Multiplexers to create one 16-bit CSA but even this process can take quite some time due to our use of many Carry Ripple Adders and MUXs in a 4x4 hierarchy. A CSA processes each section of four bits in parallel and chooses which result to use based on the previous Carry-out bit but we do not know if separating 16 bits into four modules each calculating four bits is ideal. The information we would need is quite an important one: time. We would need to know how long it takes for bits to ripple through one ripple adder by seeing the amount of time it takes for one full adder to compute a value. Also, we would need to see the amount of time required for a MUX to select and output the correct result once it receives a select bit. With this information, we can create an ideal hierarchy by experimenting with which combination of ripple adders comprised of full adders and multiplexers would generate the best single module which we can then chain together to create a hierarchy for adding our desired number of bits together.

Question 2

For the adders, refer to the Design Resources and Statistics in IQT.16-18 and complete the following design statistics table for each adder. This is more comprehensive than the above design analysis and is required for every SystemVerilog circuit. Observe the data plot and provide explanation to the data, i.e., does each resource breakdown comparison from the plot makes sense? Are they complying with the theoretical design expectations, e.g., the maximum operating frequency of the carry-lookahead adder is higher than the carry-ripple adder? Which design consumes more power than the other as you expected, why?

Table 4: Design Resources and Statistics

	CRA	CSA	CLA
LUT	79	90	83
DSP	0	0	0
Memory (BRAM)	0	0	0
Flip - Flop	20	20	20
Frequency (MHz)	64.47	67.31	65.41
Static Power (mW)	89.97	89.97	89.97
Dynamic Power (mW)	1.4	1.44	1.72
Total Power (mW)	105.09	105.16	105.53

When we take a look at the LUT (Lookup table) section of the design statistics table, we can observe that the Carry Select Adder has the most Lookup tables and the Carry Ripple adder has the least amount of lookup tables. This is because the CSA is much more complex than the CRA requiring more gates and area to implement and thus more Lookup tables. As we go from the CRA to the CLA and CSA, the complexity increases so that is why we can see the increase in Lookup tables accordingly. In the same respect, if we look at the frequency section we can notice that with an increase in LUT comes an increase in frequency. We are increasing the complexity as we develop Carry lookahead and Carry Select adders so we can add faster and this is reflected in the frequency being higher. The CRA has the lowest frequency since it is the slowest of the three as each full adder needs the carry-out of the one before it so it can complete its computation. Since the CLA and CSA compute the addition in parallel, they have higher frequencies. The frequencies

comply with the theoretical design expectations as well as they are all less than the maximum operating frequencies. For example, the maximum operating frequency of a CLA is 1.85 GHz and our operating frequency is significantly lower than the maximum.

All of the adders have the same values for DSP, BRAM, Flip-flop, and Static power but different values for dynamic power which we can compare by taking a look at the total power consumption of the three adders. The total power of the CSA and CLA is higher than the CRA because they both require more computation than the CRA does as the CLA must generate P and G values and the CSA utilizes many CRAs and MUXs to compute its addition. All of the design statistics shown in the table thus make sense and comply with the theoretical design expectations.

Conclusion

Troubleshooting Errors and Bugs

As we were developing our logic for each of the three adders, we had known that we were still quite new to SystemVerilog and could run into a significant amount of bugs. As a countermeasure, we decided to verify all of our logic on paper and to have an understanding of all of the provided code so we understand how to implement our logic into SystemVerilog code with minimal errors. Of course, this would not remove all possible errors and we had two bugs in our code and a mistake in pin assignments that needed to be resolved for our code to work on the FPGA as intended. The first bug that we had is that we had accidentally switched the logic expressions for P and G which led to each of our adders failing to function properly. We were able to quickly resolve this by switching “assign P” to “assign G” and vice-versa. Another bug we ran into was that we did not have the proper logic expression for our CSA’s carry-out bit. We were able to find out that we needed to look at the previous two carry-out bits along with the initial one and were able to fix the expression so our CSA could function properly. Finally, we had trouble getting our buttons to work on the FPGA to Run-accumulate and reset-clear and after several hours of debugging, we found that there was no error in the code but rather in our pin assignments which were not named properly. Once we resolved this bug and the two mentioned above, we were able to successfully implement all three adders onto the FPGA.

Lab Manual Feedback

Any doubts we may have had after reading over the manual were resolved by attending lectures and watching past Q/A video streams as we were able to grasp a better visual understanding of the lab and how we can start implementing it. The manual provided us with knowledge of what was required of the design in the lab and what it must accomplish whereas the lectures and Q/A videos provided us with an understanding of how we may actually get started and with hints on designing our program. We were slightly rusty on the adders which we had learned about nearly 3 semesters ago and the document did a great job of providing information on what each adder does and how we can manipulate the signals with a logic expression so we may link full adders together to create a hierarchy and not only simplify our code but also simplify our adder. We believe that the lab manual was quite direct and does not have to be changed for future semesters.

Summary

In this lab experiment, we were tasked with implementing a carry-ripple adder, carry-lookahead adder, and carry-select adder in SystemVerilog and analyze their design statistics to see if their performance matches the theoretical design expectations. To create a carry-ripple adder we implemented full-adders in series with one another to compute addition with each bit of inputs A and B and return a carry-out bit if necessary. We designed our carry-lookahead adder by grouping four full-adders together and changing four of those modules together in a 4x4 hierarchy. We were able to compute each section of four bits in parallel by creating propagate and generate signals for each full-adder and then group them together to create “propagate group” and “generate group” signals for each individual module. We could then implement those signals in logic expressions so we can add our two 16-bit inputs much faster than the CRA implementation. Finally, we created a Carry-select adder by using seven CRAs and several 2-to-1 multiplexers. We had four modules, three containing two CRAs and one MUX. Since we know that the initial Carry-in bit is 0, the first module is just one CRA and the Carry-out bit is the select bit for the next module. Since we do not want to wait for the previous module’s carry bit to do the adding, we have two CRAs, one with a Carry-in value of 0 and one with a Carry-in value of 1 with the outputs put into a multiplexer. We then select the ripple adder we want by using the carry-out bit from the previous module as a select-bit. This implementation allowed us to significantly speed up the computation as we now did not need the previous carry bits to compute the addition but instead just to select one of two computed additions. Although we ran into some silly bugs, we were able to quickly understand the provided code and the logic behind each adder so we could successfully implement the lab on the FPGA DE10 to add and display 16-bit input values as intended.