ECE 385

Spring 2023

Lab 3

Introduction to SystemVerilog, FPGA, CAD, and 16-bit Adders

Gustavo Fonseca, Hunter Baisden
TA: Hongshuo Zhang

Introduction

The purpose of this lab was to explore the different design decisions and efficiency of three 16-bit adders. The three adders in question are: Carry Ripple Adder, Lookahead Adder, and Carry Select Adder. Each has their own run time and board space required, but all serve the same purpose of adding two 16-bit numbers together.

Adders

1. Carry Ripple Adder

   The Carry Ripple Adder is the simplest design out of the three adders. To understand this adder, one must understand the Full-Adder first. A Full-Adder takes 3 inputs: x, y, and z, and outputs carry-out bit, c, and the sum, S. The carry-out bit is defined as c = (x & y) | (x & z) | (y & z). S is defined as the XOR of x, y, and z, resulting in S = (x ^ y ^ z).

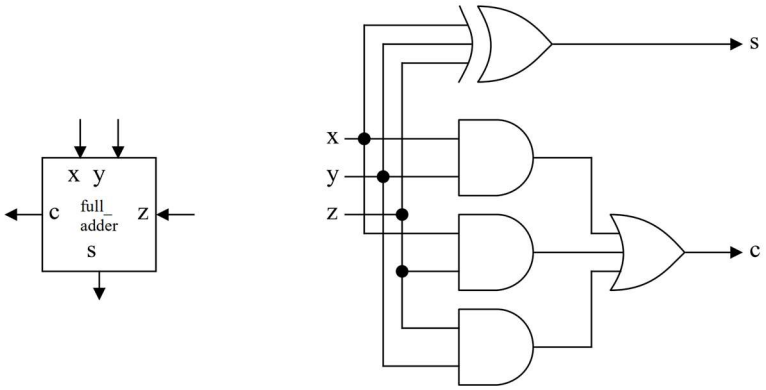

Figure 1: Full-Adder Block Diagram

The Carry Ripple Adder uses this design to create a N-bit adder by connecting N number of Full-Adders together, connecting every carry-out bit to the next Adder's carry-in bit, z.

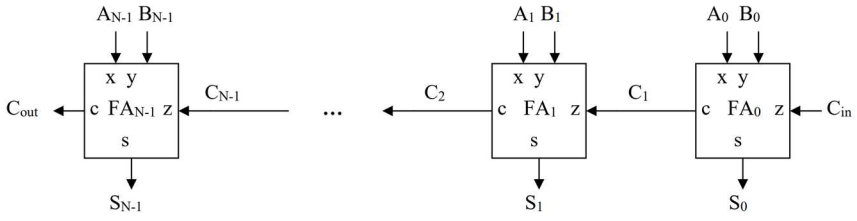

Figure 2: N-Bit Carry-Ripple Adder Block Diagram

The resulting Block Diagram from our 4-bit and 16-bit Carry-Ripple Adder we implemented on our FPGA can be seen below:

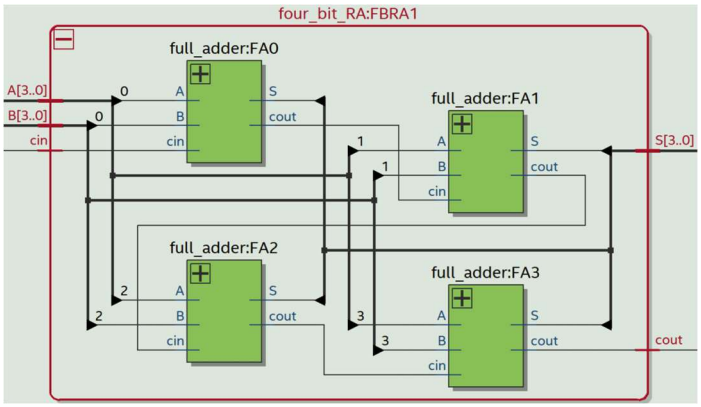

Figure 3: FPGA 4-Bit Carry-Ripple Adder Block Diagram

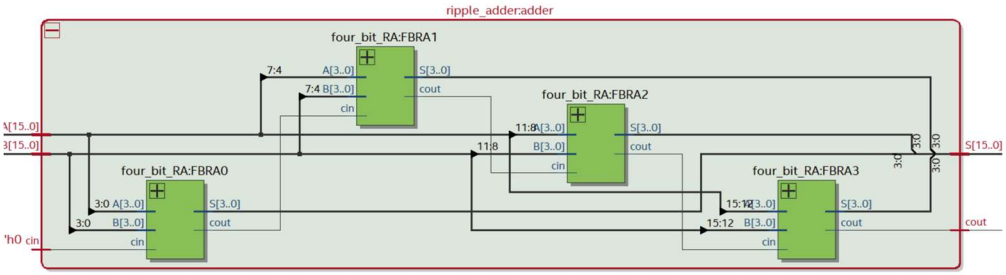

Figure 4: FPGA 16-Bit Carry-Ripple Adder Block Diagram

2. Lookahead Adder

Instead of waiting for the carry-out bit from the previous Full-Adder, the Lookahead Adder predicts the value of the carry-in bit by the previous inputs into the Adder. The Adder uses two variables to predict this: generating (G) logic and propagating (P) logic. Generating logic relies on the fact that if both inputs, A and B, are high then, regardless of the carry-in bit, the carry-out bit is a logic high, resulting in the equation: G = A & B. Propagating logic refers to how if either input is a logic high then if the carry-in bit is also a logic high then the carry-out bit will be high, resulting in the equation: P = A ^ B. Using these ideas together, we can predict the carry-out bit with the following equation: $C_{i+1} = G_i \mid (P_i \,\&\, C_i)$, where C_i refers to the current Full-Adder's carry-in bit, and C_{i+1} refers to the carry-out bit. If we use this logic from the beginning of the design and propagating it throughout, we get:

$$C_0 = C_{in}$$
$$C_1 = C_{in} \cdot P_0 + G_0$$
$$C_2 = C_{in} \cdot P_0 \cdot P_1 + G_0 \cdot P_1 + G_1$$
$$C_3 = C_{in} \cdot P_0 \cdot P_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + G_1 \cdot P_2 + G_2$$
$$\ldots$$

With this logic, we do not need the carry-out bit of any Full-Adder. Instead, we can use combinational logic based solely on inputs.

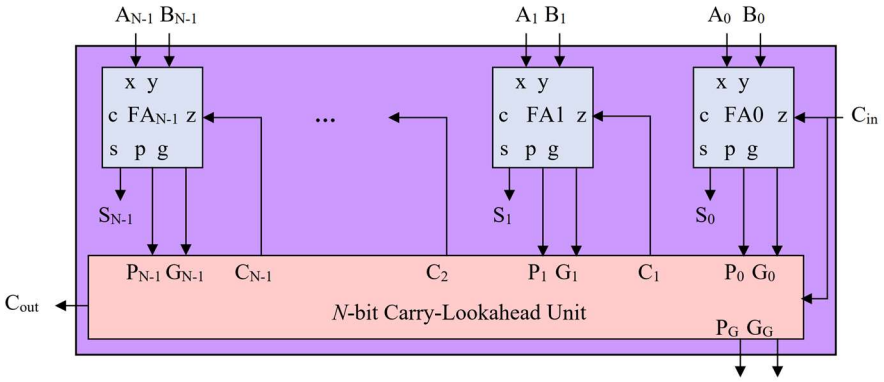

Figure 5: N-bit Carry-Lookahead Adder Block Diagram

This method would work for an arbitrary number of bits; however, after a certain number of bits, the number of logic gates needed increases the number of gate delays significantly. To prevent this, we can create a hierarchical design where we can essentially create a Lookahead Adder out of smaller Lookahead Adders. In our design we used a 4x4-Bit hierarchical design, meaning there were four 4-Bit Lookahead Adders within a Lookahead Adder. Since this method follows the logic as a normal Lookahead Adder then each Adder will need to output Group Generating and Group Propagating logic, G_G and P_G:

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

We can treat this Group logic exactly the same as in the normal Lookahead Adder. We can then use the same method of predicting, using the Carry-In bit and propagating that logic throughout:

$$C_4 = G_{G0} + C_0 \cdot P_{G0}$$
$$C_8 = G_{G4} + G_{G0} \cdot P_{G4} + C_0 \cdot P_{G0} \cdot P_{G4}$$
$$C_{12} = G_{G8} + G_{G4} \cdot P_{G8} + G_{G0} \cdot P_{G8} \cdot P_{G4} + C_0 \cdot P_{G8} \cdot P_{G4} \cdot P_{G0}$$
$$\ldots$$

This essentially creates a NxN-Bit Lookahead Adder with N number of N-bit Lookahead Adders. Thus, for a 4x4-bit Lookahead Adder, we get the following block diagram:

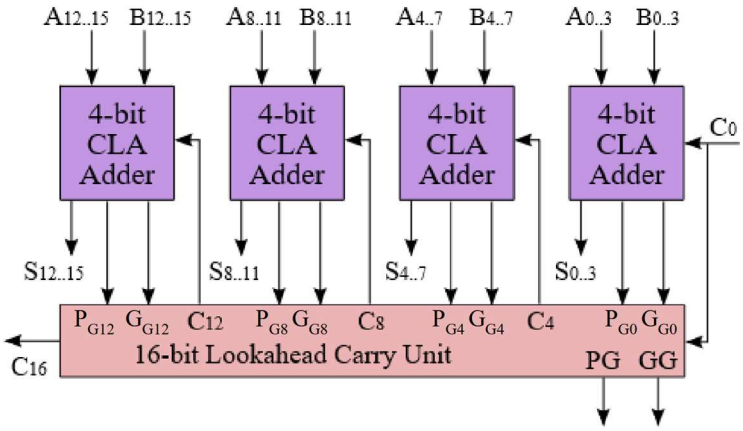

Figure 6: 4x4-bit Hierarchical Carry-Lookahead Adder Block Diagram

When implementing the Lookahead Adder in our FPGA, we generated the following block diagrams:

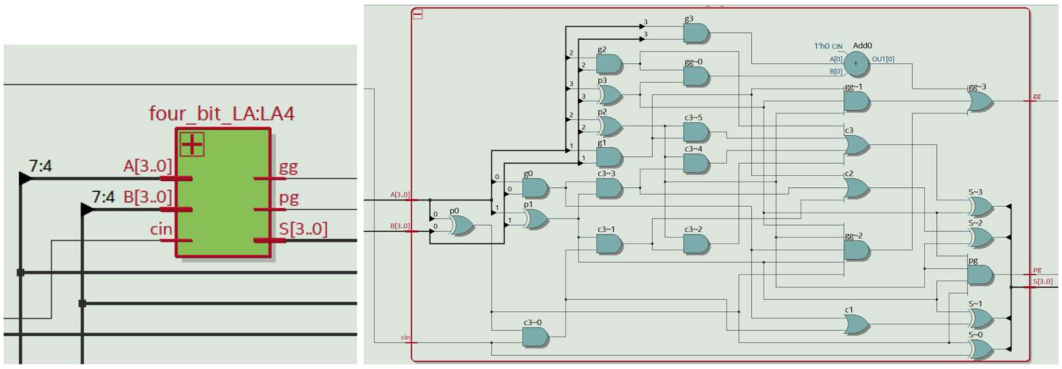

Figure 7: FPGA 4-Bit Lookahead Adder Block Diagram

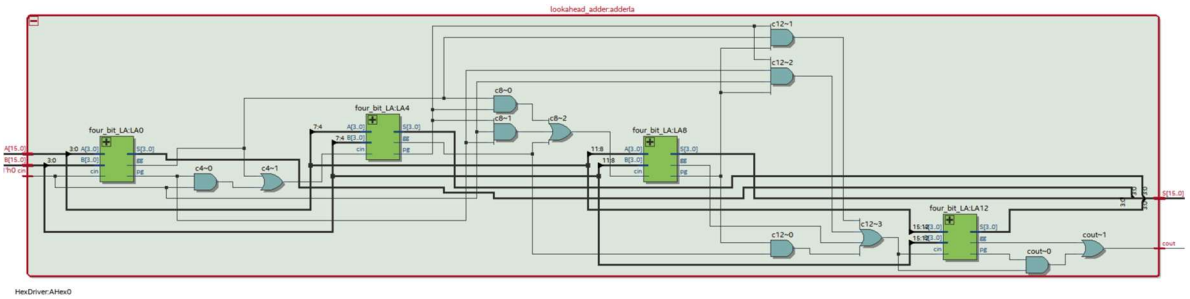

Figure 8: FPGA 4x4-bit Hierarchical Carry-Lookahead Adder Block Diagram

3. Carry Select Adder

The Carry Select Adder uses parallel addition to perform the addition before receiving the carry-in bit and then select the correct result. In this design, there are two 4-Bit Carry Ripple Adders that take the same inputs A and B. The first Adder has a carry-in bit of 0

where the second has a carry-in bit of 1. Once the actual carry-in bit is given then we use a 2-to-1 Mux to choose the correct output.

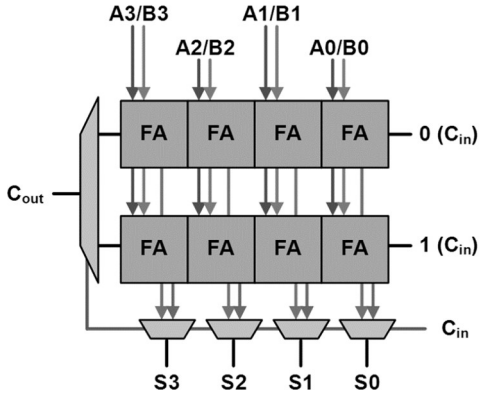

Figure 9: 4-Bit Select Adder Block Diagram

This by itself is actually slower than a normal Carry Ripple Adder. However, the real benefit is when we use a hierarchical design. If we use a 4x4-Bit Hierarchical Carry Select Adder, then all four 4-Bit Carry Select Adders can begin adding in parallel. Once the first Adder finishes, the carry-out bit can immediately be used to select the correct output for the second adder and so on. With this design, we essentially only need to wait for the gate delays of the Muxes plus the delay of a 4-Bit Carry Ripple Adder.

Implementing this into our FPGA, we get the resulting Block Diagrams:

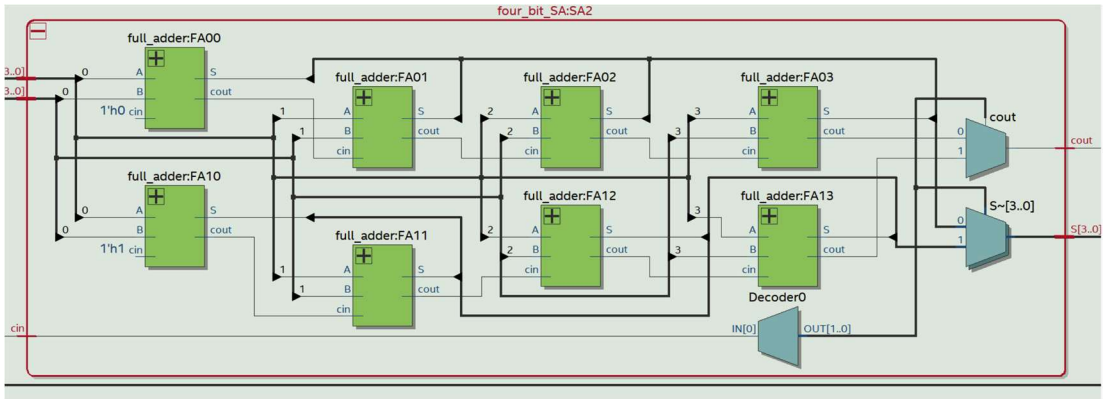

Figure 10: FPGA 4-Bit Select Adder Block Diagram

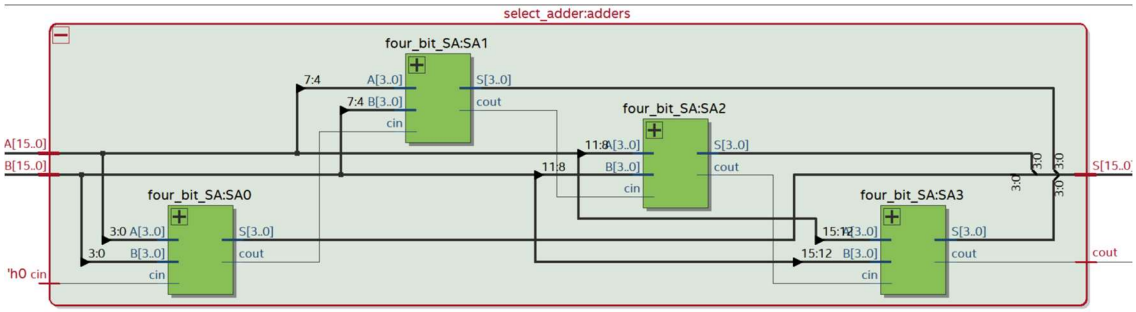

Figure 11: 4x4-Bit Hierarchical Carry Select Adder Block Diagram

When assessing this design, we realized that we could have simplified this design by using a 4-Bit Ripple Adder for the first adder as the first Carry-In Bit is given as an input. This would lower the complexity, space, and would improve performance.

SystemVerilog Modules

Module: adder_toplevel.sv

Inputs: Clk, Reset_Clear, Run_Accumulate, [9:0] SW

Outputs: [9:0] LED, [6:0] HEX(0-6), [16:0] S

Description: This module is the top level of this circuit and is used to instantiate all the modules within it. This includes the control unit, mux, register unit, the three adders, and the hex drivers.

Purpose: This module takes the input signals from the FPGA, directs the signals to perform the necessary operations, and displays the outputting signals on the hex displays.

Module: control.sv

Inputs: Clk, Reset, Run

Outputs: Run_O

Description: This module is the control unit for the design. By steeping through the states when called upon, it performs the correct operations in the right order.

Purpose: This module starts running when the Run signal turns active low. It then steps through 3 states to control the adders.

Module: full_adder.sv

Inputs: A, B, cin

Outputs: S, cout

Description: This is the baseline one-bit full adder. It is composed of 3 inputs, A, B and cin that are all XORed (*A ^ B ^ cin*) together to obtain S. The operation *(A & B) | (A & cin) | (B & cin)* is used to obtain cout.

Purpose: This module is the basis of all the adders above it. To scale the adders to 16 bits, a 4x4 hierarchy of the full adder is used.

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This module pairs every unique case of 4-bits (16 in total) into a 7-bit value used to illuminate the hex displays.

Purpose: This was used to output the values we were working with to the hex displays, both for debugging and for the demo itself.

Module: four_bit_RA.sv (in ripple_adder.sv)

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

Outputs: [3:0] S, cout

Description: This module takes in two 4-bit inputs (A and B) as well as a carry in (cin) and adds them together using 4 full adder modules and outputs the 4-bit answer to S and the carry out to cout.

Purpose: To create the 4x4 hierarchy of the ripple adder, this module is used to scale the full adder up from 1-bit to 4-bits.

Module: ripple_adder.sv

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

Outputs: [15:0] S, cout

Description: This module instantiates 4 four_bit_RA modules, each of 4-bits, to create a full 16-bit ripple adder. Each four_bit_RA module takes 4 bits of the 16-bit inputs (A and B) and adds them together, eventually outputting the answer to S.

Purpose: This is the final implementation of a 16-bit ripple adder, which is instantiated in the top-level module.

Module: four_bit_LA.sv (in lookahead_adder.sv)

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

Outputs: [3:0] S, pg, gg

Description: This module takes in 2 4-bit inputs (A and B) and a carry in, and outputs a 4-bit sum along with pg and gg. pg and gg are propagating and generating logic respectively and serve as a substitute to the carry out, speeding up the addition.

Purpose: This module is used as a helper for the lookahead module. By instantiating a 4-bit adder 4 times, we can make a 16-bit 4x4 hierarchy design.

Module: lookahead_adder.sv

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

Outputs: [15:0] S, cout

Description: This module instantiates 4 four_bit_LA modules, each of 4-bits, to create a full 16-bit carry lookahead adder.

Purpose: This is the final implementation of a 16-bit carry lookahead adder, which is instantiated in the top-level module.

Module: four_bit_SA.sv (in select_adder.sv)

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

Outputs: [3:0] S, cout

Description: This module takes in two 4-bit inputs (A and B) as well as a carry in (cin) and adds them together using 4 full adder modules and outputs the 4-bit answer to S and the carry out to cout. It assumes the carry in is both 0 and 1, and calculates the respective sum in both cases. It then uses simple logic to output the correct sum to S and cout.

Purpose: This module is used as a helper for the carry select module. By instantiating a 4-bit adder 4 times, we make a 16-bit 4x4 hierarchy design.

Module: select_adder.sv

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

Outputs: [15:0] S, cout

Description: This module instantiates 4 four_bit_SA modules, each of 4-bits, to create a full 16-bit carry lookahead adder.

Purpose: This is the final implementation of a 16-bit carry select adder, which is instantiated in the top-level module.

Module: reg_17.sv

Inputs: Clk, Reset, Load, [16:0] D

Outputs: [16:0] Data_Out

Description: This module instantiates a 17-bit register using the 17-bit D signal to load the register when Load is high.

Purpose: This register is instantiated once in the top-level module and is used to store the running total.

Module: mux2_1_17.sv

Inputs: S, [15:0] A_In, [16:0] B_In

Outputs: [16:0] Q_Out

Description: This module instantiates a Mux with two inputs: a 16-bit value (A_In) and a 17-bit value (B_In). Either input is sent to the output value, Q_Out, based on the value of the input S.

Purpose: This mux is used to select between a 16-bit value and a 17-bit value in the top level.

Area, Complexity, and Performance

	Carry Ripple	Lookahead	Carry Select
LUTs	81	93	84
DSP	0	0	0
Memory (BRAM)	0	0	0
Flip-Flop	20	20	20
Frequency	95.35 MHz	88.59MHz	93.25MHz
Static Power	89.95mW	89.95mW	90mW
Dynamic Power	1.39mW	1.35mW	1.4mW
Total Power	100.33mW	100.28mW	100.33mW

Table 1: Various Area, Complexity, and Performance values for the 3 adders

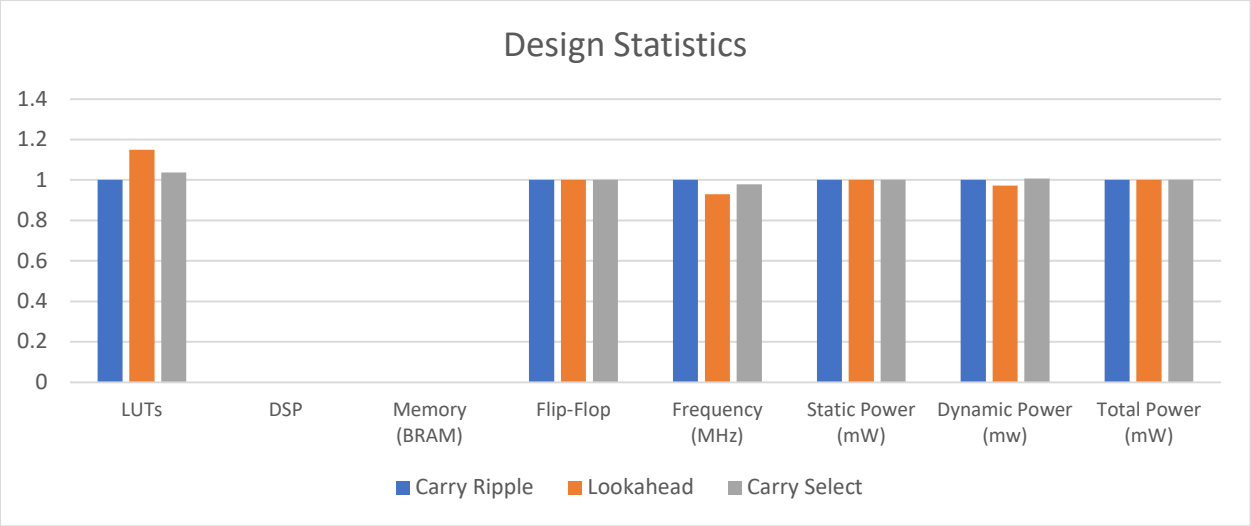

Figure 12: Design Statistics Graph Based on Table 1

1. **Area**
   The carry ripple adder has the least amount of Look-Up Tables (LUTs) required, followed by the carry select adder and then the lookahead adder. The more LUTs required for the design the less space we have for other modules in the FPGA. The more logic we add to the circuit the more LUTs we will need. When selecting which adder to use, we should consider how much space will be allocated to the adder. If we have limited space then it might be worth it to use the carry ripple adder as it uses the least amount of LUTs, even though it is typically the most inefficient.

2. **Complexity**
   The carry ripple adder is both the easiest to design and the easiest to implement as it consists of the simplest connection path. The overall structure of the RTL is much simpler versus that of the carry select adder and the lookahead adder. The carry select adder is more complex as it would require the implementation of Muxes in its design. The lookahead adder is the most complex as it relies on stringing several logic gates together to predict the carry-out bit. The complexity again comes into play when considering available space. If we want to limit the amount of IC chips or FPGA space used, then we could use a less complicated design like carry ripple adder.

3. **Performance**
   The carry ripple adder must wait for every previous operation before it can continue to the next operation. The lookahead adder and the carry select adder both get around this problem. The carry select adder avoids this problem by performing four 4-Bit ripple adder addition in parallel and then outputs the correct output once the carry-in bit is ready. The lookahead adder uses logic gates to predict every carry-out bit based solely on inputs. In theory, the carry ripple adder is expected to be the most inefficient, however, its maximum operating frequency was the largest, followed by the carry select adder, and then the lookahead adder. This means that we can perform

more 16-Bit addition operations with the ripple adder than the other designs in the same time span. While this may seem strange at first, the reason for this is due to the relatively small number of bits that we are operating on. The numerous gate delays of the lookahead adder make this design slower than the carry ripple adder for 16-Bits. The Muxes within the carry select adder increase the delay more than the delay reductions of parallel additions. We should however note that we could remove one Mux for our carry select adder design, as we added an unnecessary Mux for the first 4-Bit addition. This reduction will decrease the delay for the carry select adder, leading to the increase of the adder's frequency. Without a new implementation, it is unclear if this would lead to a more efficient 16-Bit than the carry ripple adder.
It is also important to mention that, as the number of bits within the addition increases, the carry ripple adder will become less efficient than the other adders. This is because the delay of waiting for each carry-in bit increases linearly with the number of bits within the addition. In contrast, the delay from logic gates and Muxes from the other adders would increase slower than linear.
The tradeoff for performance would come down to the number of bits we need to add. In the 16-Bit case, the carry ripple adder would be the best choice for this implementation as it has the least amount of LUTs and the highest frequency. However, as we use these designs for larger bit amounts; e.g. 64 bits, then the lookahead adder and the carry select adder would have a higher frequency and better performance. At this point, the designer would need to either value the amount of LUTs used by the adder or the performance and frequency.

Annotated Simulation Trace

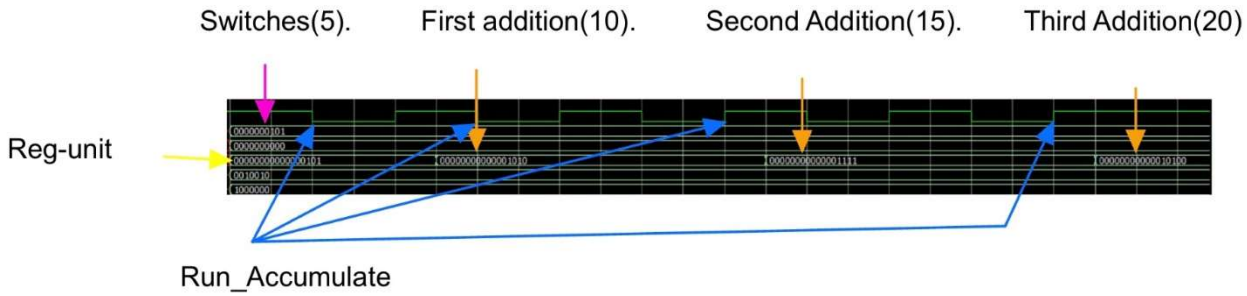

Figure 13: Annotation of simulation waveform of adding 5

This is the simulation of the Carry Ripple Adder. All the adders will produce the same output, as the RTL simulation does not consider gate delays. This shows the number 5 being loaded into the register from the switches and then Run_Accumulate being pressed. Each time, the number 5 is added to the running total. We do this 3 times. The simulation then goes from 5 -> 10 -> 15 -> 20.

Critical Path Analysis

Performing a critical path analysis on a circuit is finding the weakest link, or the part of the circuit that is the slowest. This is measured in “slack” which is “the margin by which a timing requirement is met or not met. Positive slack indicates the margin by which a requirement is met; negative slack indicates the margin by which a requirement is not met.” This means that the lower the slack value, the worse the timing of the circuit.

We performed a critical path analysis on the three adders individually. Starting with the ripple adder.

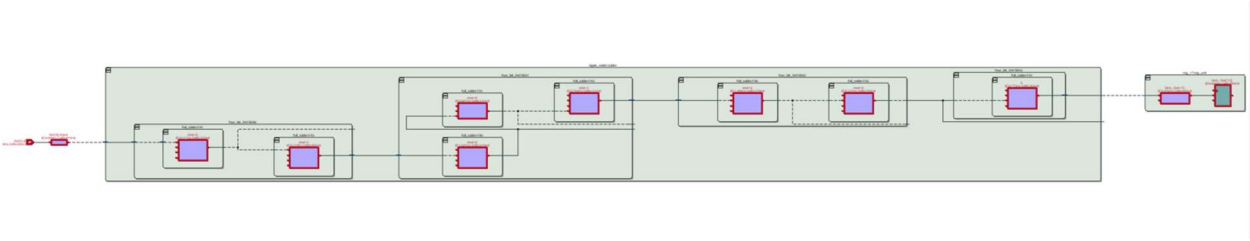

Figure 14: Ripple Adder Critical Path

Counter intuitively, the ripple adder had the highest slack value at 9.914. This means that it was closest to the set timing requirements of any of the adders.

Next up is the lookahead adder.

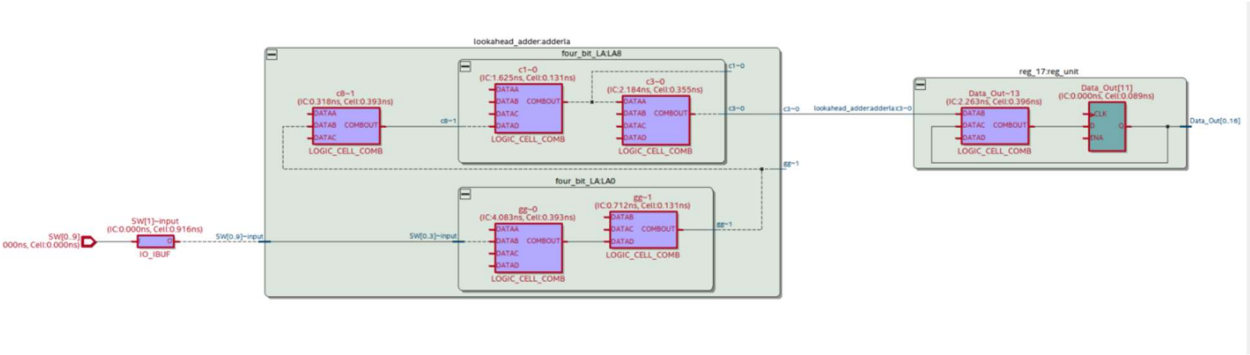

Figure 15: Lookahead Adder Critical Path

This adder had a slack value of 9.412. This means it actually is farther away from hitting the timing margin than the ripple adder.

As shown in Figure 16 for the carry select adder, the slack value turned out to be 8.62, which is worse than both the previous circuits.

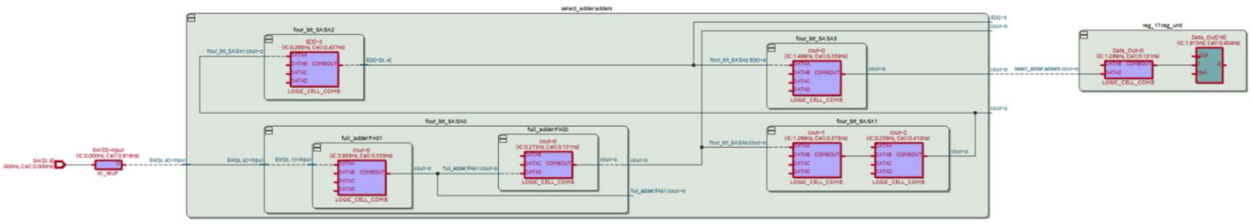

Figure 16: Carry Select Adder Critical Path

This is counter intuitive because the lookahead adder and the carry select adder are designed to be faster, to have a higher slack value.

However, we believe we got these results because the addition that we were doing was not with a wide array of bits, only 16 in a 4x4 array. If we were doing addition on 64 bits in a different array setup, we would see the opposite order of slack numbers appear on the circuit path analysis. Compared to the ripple adder, The lookahead and carry select adders both get much more efficient the more bits are being added together.

Post-Lab Questions

1. **Ideal Hierarchical FPGA Design**
   There are three considerations that are ideal for a 16-Bit Carry Select Adder, the 4x4-Bit design, the 8x2-Bit design, and the 2x8-Bit design. The main advantage of the using the 8x2-Bit design is that there is only one Mux being used. The main disadvantage of the 8x2-Bit design would be that there would only be two 8-Bit Ripple Adders working in parallel at once instead of 4. The main advantage of using the 2x8-Bit design is that there would be eight 2-Bit Ripple Adders working in parallel instead of four 4-Bit Ripple Adders. The main issue with the 2x8-Bit design is that we would need to more than double the amount of Muxes, from 3 to 7, increasing the space and delay needed for this design.
   For us to assess which design is ideal, we would need to know a variety of information. Firstly, we would need to know how much space we have for an adder on our chip. The experiment we must perform for this is to create all 3 designs and test the size of all 3 designs on the FPGA. We can then assess on a per-design basis if it the increase in size is worth the performance increase. Another question is to consider how the performance differs among each design. Is the time saved by performing a 2-Bit Ripple Adder instead of a 4-Bit design worth the 7 Mux gate delay rather than three? Would the one Mux gate delay cause the 8x2-Bit design to be faster? The experiment we must perform for this is to test the frequencies of all 3 designs to see the difference in performance. We can then again assess if there is an increase in performance. Is it worth the increased size of the

Adder?
Knowing that a Carry Ripple Adder is more efficient than a Carry Select Adder for 16-Bit addition, we can assert that the 8x2-Bit design may be the ideal design. The reason being is that it seems that increasing the number of Mux gate delays causes more delays than adding more bits into a single Carry Ripple Adder. Thus, the 4x4-Bit design having 3 Mux gate delays with 4-Bit adders could be slower than the 8x2-Bit design with 1 Mux gate delay with 8-Bit adders.

2. **Explanation of Data**
    a. **LUT**
    The Lookahead Adder had the most LUTs followed by the Carry Select Adder and the Carry Ripple Adder. This makes sense as the Carry Ripple Adder has the least amount of logic within its design. The Carry Ripple Adder sacrifices performance for being the simplest design. The additional logic gates used to predict the Carry-Out Bits added more needed LUTs in the design. The Carry Select Adder implements the use of Muxes to allow for the parallel addition of bits. These Muxes increases the amount of LUTs in the design. Thus, this results directly follows the theoretical design size of each design.
    b. **DSP**
    All 3 designs do not use our Digital Signal Processor (DSP) in their design. Thus, it follows theoretical design expectations that the DSP values would be zero for all designs.
    c. **Memory (BRAM)**
    All 3 designs do not use BRAM memory in their design. This follows the theoretical design expectations as we are not storing anything into memory only into registers in all designs.
    d. **Flip-Flop**
    All 3 designs have the same inputs and outputs. Thus, this follows the theoretical design expectations for all 3 designs as they would have the same amount of flip flops to store the values within the registers.
    e. **Frequency**
    This result does not follow the theoretical design expectations. The Carry Ripple Adder has a higher maximum operating frequency than the Lookahead Adder and the Carry Select Adder. While this does not follow the theoretical design exception, the main reasoning for this is that, with this low number of bits being added, the gate delays of the Lookahead Adder and the Carry Select Adder are larger than that of the Ripple Adder. More details on discrepancy are explained in the “Complexity” section of the Report.
    f. **Static Power**
    All 3 designs have relatively the same Static Power Consumption. This makes sense as they are all doing the same operation, the scale of this addition is not enough to make the circuit’s Power Consumption radically different.

g. **Dynamic Power**
   All 3 designs have relatively the same Dynamic Power Consumption. For the same reasoning as the Static Power Consumption, this follows the theoretical design expectation.
h. **Total Power**
   All 3 designs have relatively the same Total Power Consumption. For the same reasoning as the Static Power Consumption, this follows the theoretical design expectation.

Conclusion

1. **Bugs and Countermeasures**
   One of the only bugs we encountered on this lab was on the Lookahead Adder. When testing, we noticed there was an issue with adding 1 to any number ending in an F, such as 2F, resulting in 2F being outputted instead of 30. We resolved this issue by fixing our final Carry-Out Bit logic that was inconsistent with the rest of the design. After fixing this inconsistency, the bug was fixed.
2. **Recommendations**
   The main ambiguous part of this lab was the inconsistent design of the Carry-Select Adder. In the lab manual there was a different design than the one described in the lecture and Q&A session. This caused us confusion as we were unsure which design to implement. This also caused confusion, and almost a loss of points, during the demo. We believe that mentioning both on the lab manual would help with this confusion. We believe that the breakdown of each adder was done very well. Especially in the case of the Lookahead Adder, the description in the lab manual did a wonderful job explaining the hierarchical design.