ECE 385 Lab Report 3

Experiment #3: Introduction to SystemVerilog, FPGA, CAD, and 16-bit Adders

Spring 2022

Trusha Vernekar & Soham Manjrekar

Dohun Jeong (DJ)

tnv2 & sohammm2

Introduction

In this lab, we had to implement 3 different types of adders: Ripple Carry Adder, Carry Lookahead Adder, and Carry Select Adder. All the 3 adders performed the same operations. However, the main difference between each was the handling of the carry bit for each bit. Due to this difference, the time taken to process the addition of each bit was optimized. After the addition was carried out, the answer was displayed on the FPGA in Hexadecimal.

Ripple Carry Adder

Written description of the architecture of the adder

This adder was the simplest to implement out of all of them because the code to make a 4-bit Ripple Carry adder was provided to us. For the 4-bit adder, there were 4 Full Adders that were used one after the other. We simply had to modify this code to make it work for a 16 bit Ripple Carry Adder. To do this, we had to add 12 more Full Adders.

Our Ripple Carry Adder uses 16 Full adders (Number of input bits = Number of full adders required) to compute the sum between 2 inputs. Each bit from the input is passed into one full adder, with the least significant bit computed first. An initial carry bit, known as C_{in} is passed in the first full adder (C_{in} = 0 in our model). The sum is calculated for each bit. Once the sum is calculated in each full adder, the carry bit due to the addition is passed out to the current full adder as C_{out} and goes into the next consecutive full adder as C_{in}. The Carry bit and the Sum for each bit are calculated using the logic below:

$Sum/S_i = A \hat{} B \hat{} C_{in}$

$C_{out} = (A \,\&\, B) \mid (A \,\&\, C_{in}) \mid (B \,\&\, C_{in})$

The C_{in} bit used in the calculation above comes from the C_{out} from the previous adder (except for the first adder in which case the C_{in} is set to 0). The A and B are a single bit from each input. The two calculations are simultaneously in each full adder. Since the next full adder depends on the C_{out} calculation in the previous full adder, the time taken to compute the result is extremely slow. To make this computation faster, a Carry Lookahead Adder or a Carry Select Adder can be implemented instead.

Block Diagram

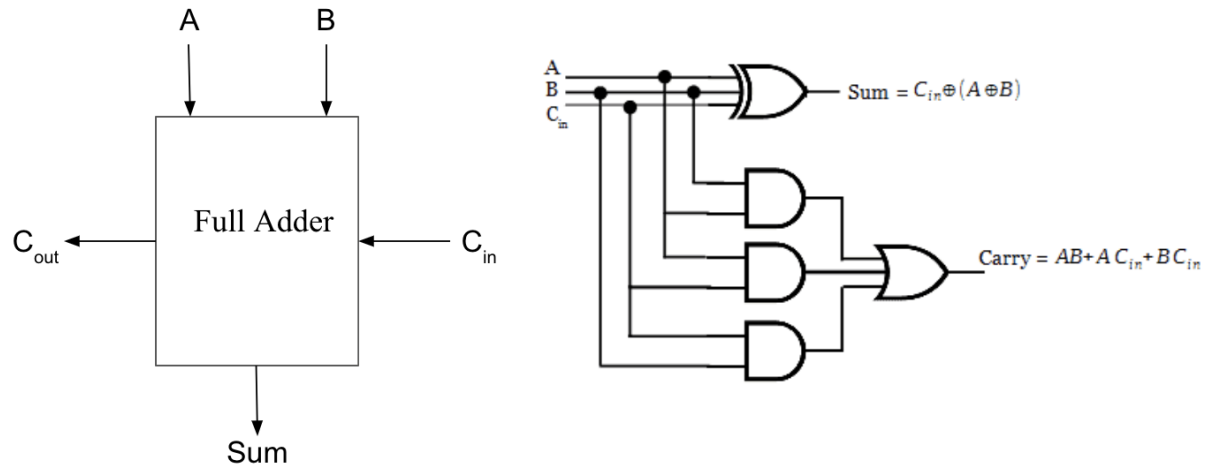


Figure 1: Full Adder Diagram and Schematic

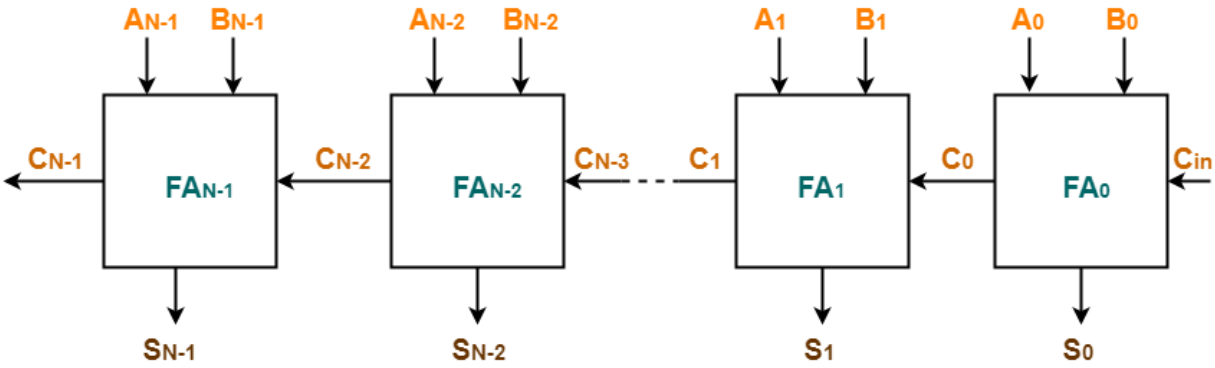


Figure 2: N-Bit Carry Ripple Adder Block diagram

Carry Lookahead Adder

Written description of the architecture and operation of the adder

The Carry Lookahead Adder was one of the two adders that we implemented which was more time-efficient than the Carry Ripple Adder. This was because there is no need to wait for the carry-in bits to be computed for the previous bit slice. The way the adder does this is by using two additional signals known as Propagate and Generate. Since we had to build a 16 bit Carry Lookahead Adder, we had to divide our unit into 4 CLA units, each carrying out the computation for 4 bits. These 4 CLA units were then connected to another 16 bit CLA unit (as shown in figure 5 below). One 4 bit CLA unit would compute the sum like any other adder using the input bits A and B, and the C_{in}. In addition to this, the Propogate and Generate signals were computed for each bit in the CLA unit. These signals were then further processed to calculate the carry bits for

the next CLA unit. The Propagate, Generate and the Sum for each bit is calculated using the logic below:

Sum/S_i = A_i^B_i^C_{in}

Propagate/P_i = A_i^B_i

Generate/G_i = A_i&B_i

What these Propagate and Generate signals mean is that a C_{out} is **Generated** when both available inputs (A and B) are 1, regardless of the C_{in}. On the other hand, a C_{out} has the possibility of being **Propagated** if either A or B is 1. The Propagate and Generate signals, as mentioned above, will be used to calculate the Carry bits for the next 4 bit CLA unit. The purpose of calculating these signals for each bit is so that the carry for any bit can be calculated without having to wait for the carry from the previous bit. The carries for each of the 4 bits, c_0 c_1 c_2, and c_3, are calculated as follows:

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

As mentioned earlier, we had to create a 4x4 hierarchical design for our Carry Lookahead Adder. We had four 4-bit CLA units (each calculating the carries c_0 c_1 c_2 and c_3 along with the P and G signals) which were attached to a bigger Carry Lookahead unit. This bigger CLA unit would compute the C_{in} values for each of the 4-bit CLA units, using the Propagate Group (PG) and Generate Group (GG) signals, computed for each 4-bit CLA called PG_0, PG_4, PG_8, PG_{12}, and GG_0, GG_4, GG_8, GG_{12}. These PG and GG signals were calculated as follows:

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

The PG and GG were calculated for each 4-bit CLA unit using the logic below:

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

As seen in figure 5 below, the C_4 C_8 and C_{12} are the C_{in} for each of the 4-bit CLA units. These PG and GG signals are also used to calculate the final C_{out} for the bigger CLA unit, using the logic below:

$C_{out} = GG_{12} + GG_8.PG_{12} + GG_4.PG_{12}.PG_8 + GG_0.PG_{12}.PG_8.PG_4 + C_0.PG_{12}.PG_8.PG_4.PG_0$

Hence, using all signals above, the delay due to the Carry getting rippled is reduced significantly by the use of a Carry Lookahead Adder.

Block Diagram

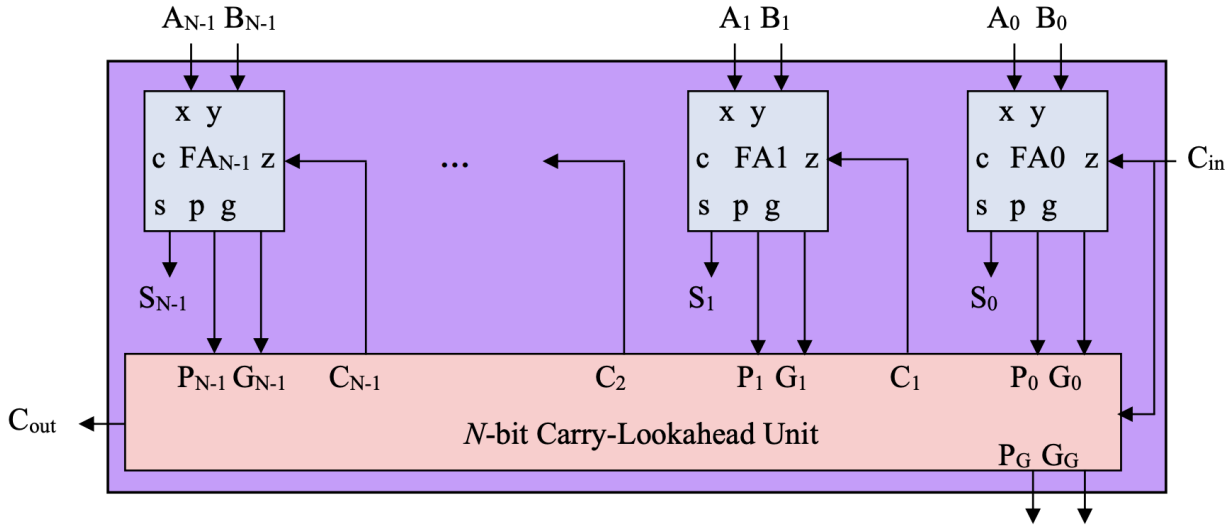


Figure 4: Inside a Single 4-bit Carry Lookahead Adder Block diagram

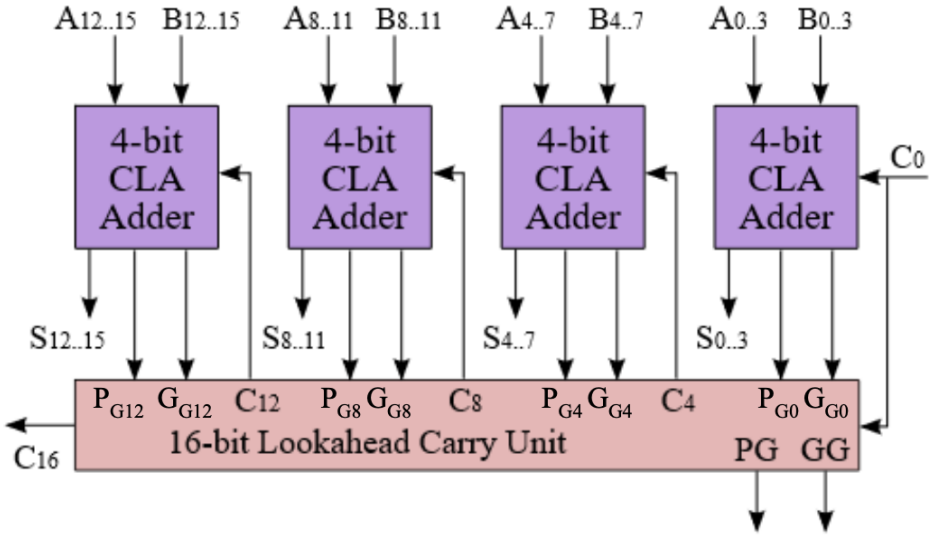


Figure 5: A 4x4-bit Hierarchical Carry-Lookahead Adder Block Diagram

Carry Select Adder

Written description of the architecture and operation of the adder

The Carry Select Adder was the last adder we implemented which was more time-efficient than the Carry Ripple Adder. This was done by reducing the delay due to the ripple of the carry bit in a Ripple Carry Adder. The way a Carry Select Adder does this is by having 2 Full Adders and a Multiplexer for each bit. It computes 2 sums: one assuming that the carry bit is 0 and the other assuming that the carry bit is 1. Since we were building a 16 bit Carry Select Adder, we needed to group 4 bits together. This created a 4x4 hierarchical design for our Carry Select Adder. Each of the 4 units had two 4-bit Ripple Carry Adder units except for the one computing the 4 least significant bits of the input. The reason for this is due to the main feature of the Carry Select Adder as mentioned earlier. Hence, we used 7 Ripple Carry adders and a Multiplexer for each of the 4-bit units.

In our 4x4 hierarchical design, we have two 4-bit Ripple Carry adders. For each of the 4 bits, the two sums are pre-computed. One adder computes the Sum and C_{out} based on the assumption that the C_{in} is 0, and the other assumes that the C_{in} is 1. Then the unit must wait for the real C_{in} to arrive from the previous 4-bit unit. Once this C_{in} is known, the corresponding Sum and C_{out} for the next 4-bit unit is selected by the use of Multiplexers. The reason why the least significant 4-bit unit requires only one 4-bit adder and no multiplexer is because the C_{in} for that unit is already known (The first C_{in} in our implementation is always 0). Since there is no need to choose between two sums as C_{in} for that unit is always 0, one Ripple adder is used. This is why our unit contains 7 CRA units instead of 8.

As mentioned above, the two sums are pre-computed by assuming that the C_{in} for one unit is 0 and for the other is 1. This computation is done for every 4 bits of our two 16-bit inputs A and B. All of the computations in the 4 CSA units are done in parallel. The only thing that each CSA unit has to wait for is the C_{out} from the previous CSA unit. This does not involve the first CSA unit as the C_{in} for this unit is always known. Once each of the CSA units receives its corresponding C_{in} value from the previous CSA unit's C_{out}, the sum and the C_{out} for the next CSA unit are rapidly chosen because all the values are pre-computed. This is done by the use of a 2:1 MUX. As we noticed, there is some kind of a Ripple of the Carry bit happening between each CSA unit. However, since the computation is already done, the Carry Select Adder is rapid and more efficient as compared to a regular Ripple Carry Adder.

Block Diagram

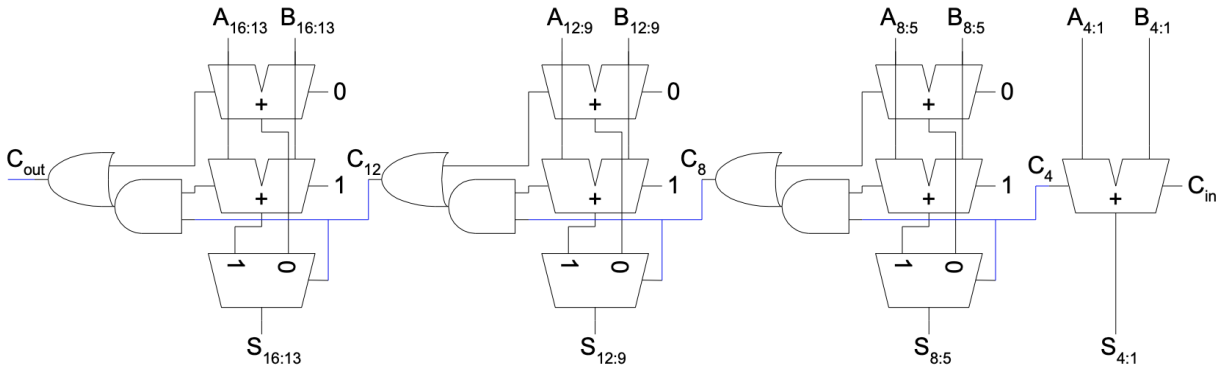


Figure 6: 16-bit Carry-Select Adder Block Diagram

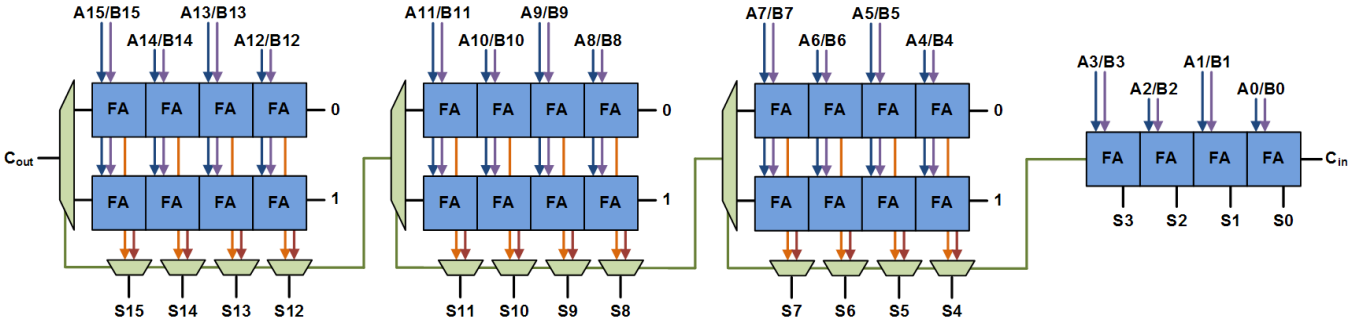


Figure 7: 4x4 Hierarchical Carry Select Adder Block Diagram

Describe at a high level the area, complexity, and performance tradeoffs between the adders.

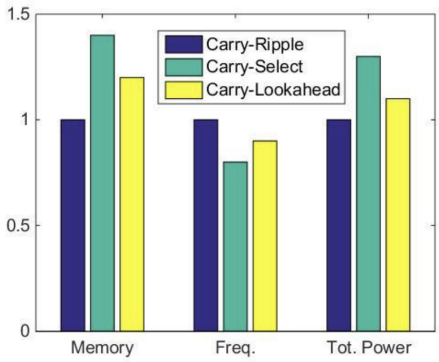
In the Ripple Carry Adder, the sum and C_{out} are calculated for each of the 16 bits of the input. Due to this, there are 16 Full Adders that are used in this adder. Also, as mentioned earlier, there is a Ripple of the carry bit between each Full Adder, causing a huge delay in computation, making this adder the slowest out of the 3 implemented in this lab. Although this adder was the easiest to implement, it uses the most number of gates, and hence, its performance can be considered as the worst out of the 3.

In the Carry Select Adder, since the sum and C_{out} are pre-computed for every 4 bits of the input, there is no ripple of the carry bit, except between each CSA unit. However, this does not cause a significant delay in time. Due to this, this adder is considerably faster than a Ripple Carry Adder. Even though this adder was easy to implement as well, it uses double the number of adders as compared to a Carry Lookahead Adder and uses multiplexers as well which takes up more area.

In the Carry Lookahead adder, since the carries are directly calculated using each bit from the inputs using the Propogate and Generate signals, none of the CLA units have to wait for the carry from the previous unit. Due to this, this adder can be considered the fastest amongst all of the adders implemented in this lab. Although this adder was complicated to implement, it uses less number of gates as compared to a Ripple adder and a CSA, making its performance the best out of the 3.

Prelab Question:

	Carry-Ripple	Carry-Select	Carry-Lookahead
Memory (BRAM)	0	0	0
Frequency	78.15 MHz	93.57 MHz	99.21 MHz
Total Power	105.36 mW	105.34 mW	105.41 mW



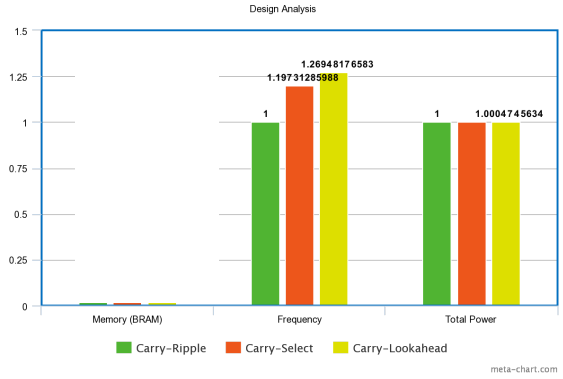


Figure 8: Evaluated Multi Bar Graph

Annotated simulation trace

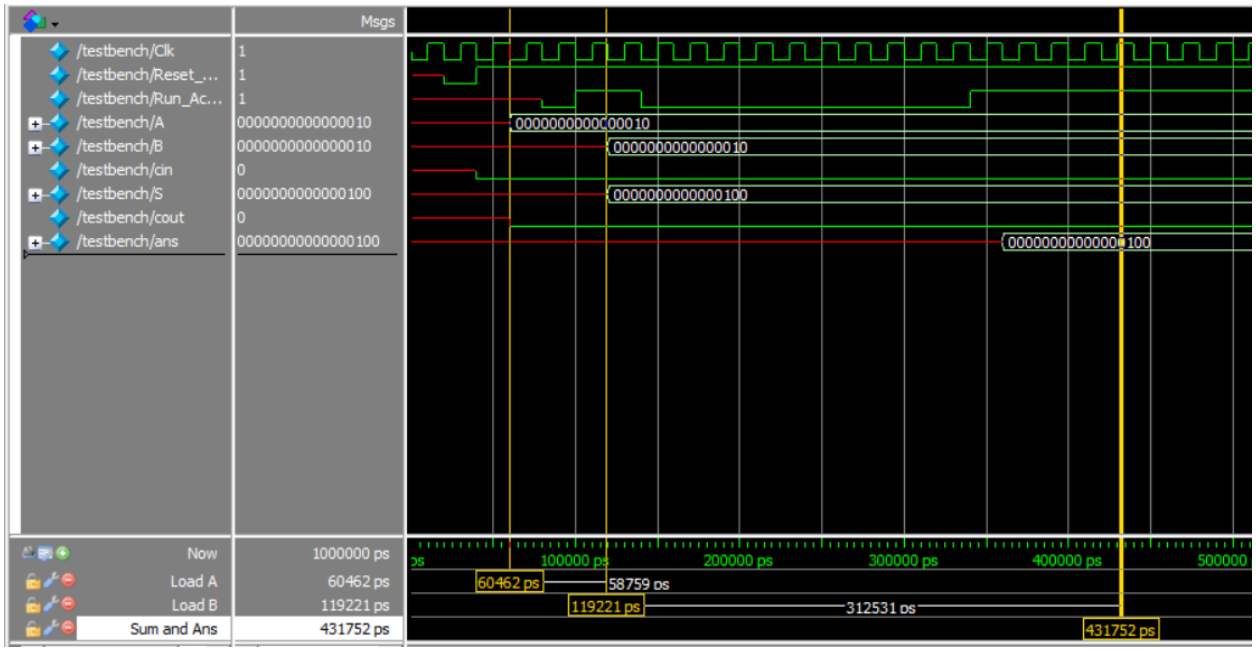


Figure 9: Annotated Trace

- In the simulation above we are showing how one of our modules performs an addition.
- You can see that A and B are loaded at the first two cursors positions
- And without any delay, the Sum is then calculated and displayed in the simulation, later when compared to the numerical operation of 2 + 2 done for “ans” the two match up.

Extra Credit: Critical Path Analysis

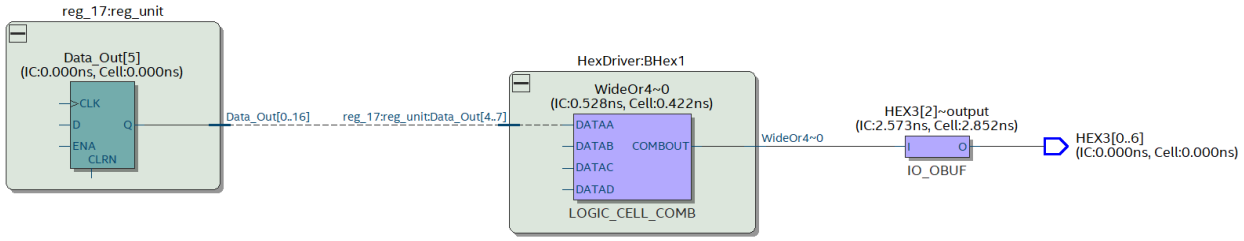


Figure 10: Critical Path for the Carry Lookahead Adder

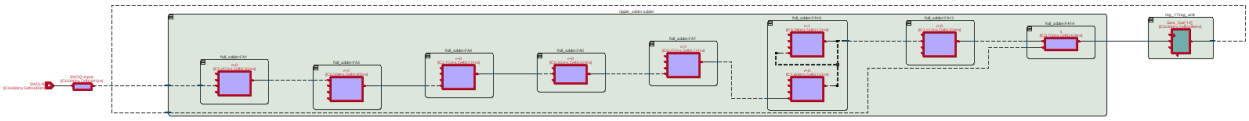


Figure 11: Critical Path for the Ripple Carry Adder

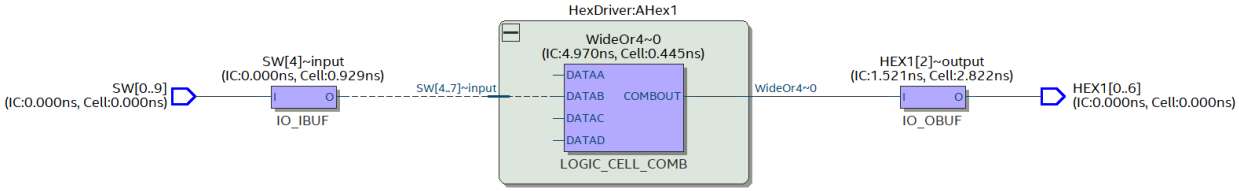


Figure 12: Critical Path for the Carry Select Adder

In the three figures above we see the critical paths for the three different adder modules. We see that for ripple carry there are many elements between input and output when compared to lookahead and select, which is expected. We also see that for the select adder between the switches at the input and the combinational logic block which outputs to the HEX driver there is the MUX that cuts down on the total number of elements on the path. For the lookahead adder, we see that the logic is purely combinational and thus goes straight from the input at the registers to the combinational logic block.

Post-Lab Questions

Q1) 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?)

Answer: The 4x4 hierarchy is not ideal for Carry-Select Adder the gate delays for this implementation are the delays from 4 full adders and 3 MUXes, there are only 3 MUX delays since the first adder has no carry in. To find the ideal configuration we need to configure the adder such that the delay from the MUX and the delay from calculating the addition of your inputs are equivalent so that the carry out is calculated as quickly as possible. To test we should vary the number of adders in each block and vary the number of MUXes used to pass the carry bit. We found that by increasing the total number of MUX delays we could decrease the number of full adder delays. We can do 8 two-bit adders which would have 7 MUX delays. If we extend the last adder to 6 bits, we have 1 six-bit adder, 5 two-bit adders, and 5 MUXes. By continuing to experiment you can figure out which division of the 16-bit adder is ideal. Some pieces of information you would need are how to calculate the delay from the total number of bits you are adding, and what the delays of an extended number of full adders are.

Q2) 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.

	Ripple Carry	Carry Lookahead	Carry Select
LUT	79	91	82
DSP	0	0	0
Memory (BRAM)	0	0	0
Flip-Flop	20	20	20
Frequency	78.15 MHz	99.21 MHz	93.57 MHz
Static Power	89.97 mW	89.97 mW	89.97 mW
Dynamic Power	1.6 mW	1.64 mW	1.61 mW
Total Power	105.36 mW	105.41 mW	105.34 mW

Observe the data plot and provide an explanation to the data, i.e., does each resource breakdown comparison from the plot make 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?

Answer: According to the data collected from the Fitter Resource Usage and the Power Analyzer, we can see that the Carry Lookahead Adder has the highest frequency out of all of the adders with a value of 99.21 MHz. A higher frequency suggests that the computations in the adder are done faster. This complies with our expectation because as mentioned earlier, the Carry Lookahead Adder must be the fastest amongst the 3 adders due to the Propagate and Generate signals. Along with this, we also mentioned that the Carry Lookahead Adder takes more area because it has more gates. The data collected above comply with this expectation as well as the LUT for the CLA is higher, suggesting more logic elements. The Carry Lookahead Adder consumes the maximum power. This is because there are more computations performed in this adder (Such as Propagate and Generate), causing it to use more power as compared to the other adders.

Description of all bugs encountered, and corrective measures taken

Bugs	Corrections
First Carry bit calculation in the Carry Lookahead Adder was incorrect- As mentioned earlier, the C_1 bit in a single 4 bit Carry Lookahead adder is calculated by: $C_{in}.P_0 + G_0$. However, while implementing this in our code, we grouped the terms as $C_{in}.(P_0 + G_0)$ instead of $(C_{in}.P_0) + G_0$. This affected the carry calculated for the first bit in each 4-bit carry-lookahead adder, causing it to output the wrong value at that bit of the sum.	We simply had to change the bracket positions in our code in order to fix this issue.
Implementing the MUX for the Carry Select Adder incorrectly in the higher level of the 4x4 hierarchy instead of the lowest level	Constructing the MUX module within select adder was fairly straightforward; however, figuring out where in the hierarchy we need to call the MUX so that the correct full adders output and carryout are used and passed was

	slightly difficult. When initially implementing the MUX we incorrectly used it in the highest level of the hierarchy but we actually needed to use it in the lower levels where we called the 4-bit full adders so that we could pass the correct values out to the higher level of the hierarchy.

Table 1: Bugs and corrections for the same

Conclusion

While doing this lab, we realized that the Carry Lookahead adder was the most difficult to implement out of the 3 adders simply because there were a lot of different values to be computed and to be kept track of apart from just the sum. Due to this, we ended up having a lot of local variables in each of our functions. Apart from this setback, in our opinion, the lab manual was extremely helpful in explaining each adder in detail including how each signal (propagate and generate for example) in the Carry Lookahead adder is calculated. The presence of multiple diagrams further helped us to understand and construct each adder.

This lab expanded on the adders we were introduced to in ECE 120. We learned how each adder performs its bitwise addition in detail and we were able to understand the reasons and cases in which we should use one adder over the other. Overall, the purpose of the lab, to create the 3 different adders, Carry Ripple Adder, Carry Lookahead Adder, and Carry Select Adder was successfully completed.

Appendix A:

Module: select_adder.sv

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

Outputs: [15:0] S, cout

Description: Carry Select module, 4 full adders whose carry-outs are passed through using 3 MUXes, each full adder has 2 sets of adders but the sum and carry out is chosen by the previous adder groups carry out with the use of a MUX. The MUX is a simple 2:1 MUX that chooses the sum based on whether the previous carry out is 1 or 0.

Purpose: To perform a 16-bit addition using carry-select, we should expect to see exactly what the outputs of our carry-lookahead and ripple carry adders are.

Module: lookahead_adder.sv

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

Outputs: [15:0] S, cout

Description: Carry lookahead module, 4x4 hierarchy where the carry bits we should expect to see exactly what the outputs of our carry select and ripple carry adders are, this adder should be faster/more efficient than the others. We calculated the carry outs using propagate bits p, and generate bits g. P is given by the logical XOR of the inputs, and G is given by the logical AND of the inputs. Using P and G we can calculate the carry bits not only within the 4 bits we are adding but for each of the 4 bits groups as well.

Purpose: To perform a 16-bit addition without rippling the carry bits, while we should expect to see exactly what the outputs of our carry select and ripple carry adders are, this adder should be faster/more efficient than them.

Module: ripple_adder.sv

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

Outputs: [15:0] S, cout

Description: Ripple Carry adder module, simply 16 full adders where the carry is passed through to each subsequent adder.

Purpose: To perform a 16-bit addition with seemingly the least amount of effort. We should expect to see exactly what the outputs of our carry select and carry lookahead adders.

Module: router.sv

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

Outputs: [16:0] Q_Out

Description: 17bit parallel MUX implemented using case statements. Assigns the outputs of A and B into the output sum depending on value and position

Purpose: ensures that when R is 0 A is the bit that enters into Q, and when R is 1 B is the bit that enters into Q.

Module: hexdriver.sv

Inputs: [3:0] In0,

Outputs: [6:0] Out0

Description: Takes a nibble or 4 bits and converts it to a 7 bit equivalent that is displayable on the HEX# output LEDs of the FPGA.

Purpose: To allow the LEDs to display what the input and output of our code are.

Module: adder2.sv

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

Outputs: [9:0] LED, [6:0] HEX (6 total)

Description: Top-level entity for the adder, calls the control unit to enable the register and allows you to choose what adder you would like to perform. Using logic variables the output from the adder is then used to drive the seven segment displays on the FPGA to show what the overall output of the sum is.

Purpose: To instantiate all modules and display the sum.

Module: control.sv

Inputs: Clk, Reset, Run,

Outputs: Run_O

Description: State Machine that designates the state of the route module. Drives the value of R to either 0 or 1 depending on the status of the execution and registers.

Purpose: To make sure that the adder performs one adder per execution.

Module: reg_17.sv

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

Outputs: [16:0] Data_Out

Description: when the reset load button is pressed the register is set to 0, otherwise the value of D is taken from the route module.

Purpose: to make sure that the value in register B when addition occurs is correct.