# ECE 385

Fall 2021

Experiment #3

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

Michael Grawe, Matthew Fang
ABE
Abigail Wezelis

# Index

**3.1 Introduction** **3**

**3.2 Adders** **3**
Carry Ripple Adder 3
Carry Lookahead Adder 4
Carry Select Adder 6
Written Description of .SV modules 7
Tradeoffs and Performances 11
Simulation Trace 12

**2.3 Post-Lab Questions** **11**

**2.4 Conclusion** **12**

## Introduction

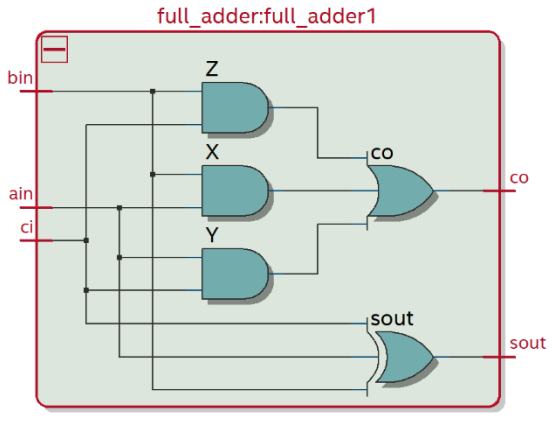
The focus in this lab was on the different types of adders that could be created and the differences in their implementation. All three adders take in two 16-bit values and a carry in and outputs a 16-bit sum and a carry out. One adder can be implemented at a time, and that adder is controlled by a Reset signal, which clears the register responsible for performing the addition, and the Run_Accumulate signal which sets in the motion the desired addition operation outputs the result to the accumulate register.

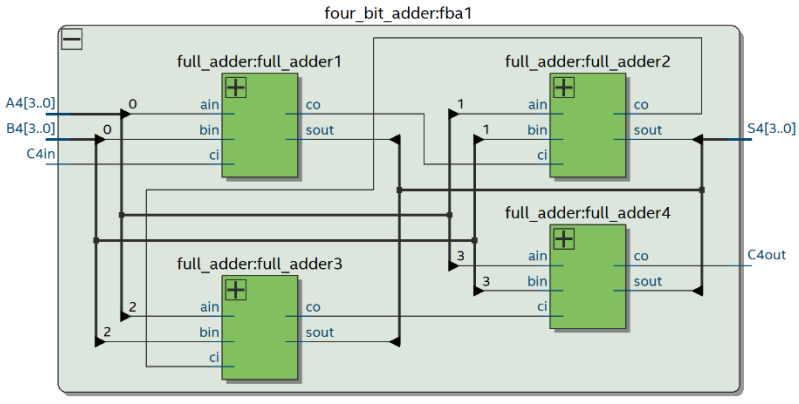
## Carry Ripple Adder

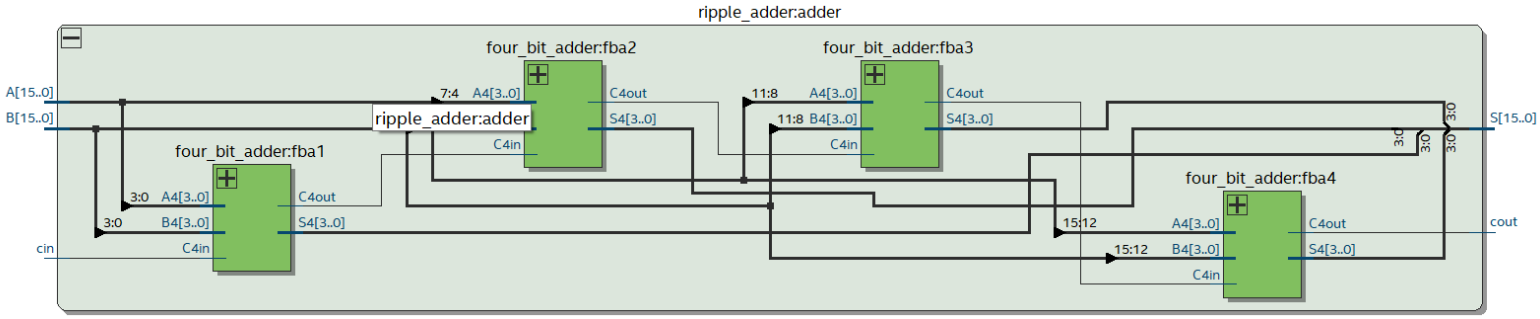
The first adder that we implemented was the carry-ripple adder. This adder was the simplest to construct. The basic building block for this adder was a full-adder module. This full-adder was meant to perform binary addition on 1 bit. Its inputs are two one-bit values (a,b) to add together, and a carry in (cin). Its outputs are the sum (s) of the carry in + a + b, and the carry out (cout) bit of that addition. The logic in this full-adder is:

$$c(out) = (ab)+(ac)+(bc)$$
$$sum = a \oplus b \oplus cin$$

This full adder is then chained together 4 times to create a 4-bit ripple adder. This is done by connecting the carry out bit of the previous adder to the carry in bit of the current adder. Finally, the 4-bit ripple adder is chained together 4 times to get a 16-bit ripple adder, the desired result. This is again done by connecting the previous carry out to the current carry in. This system is illustrated in the block diagrams below.






**Carry Lookahead Adder**

The carry-lookahead adder had a smallest building block of a full adder that we called the CLA full adder. This is different from our previous full adder since it has two different outputs. Our CLA full adder takes in two 1-bit values to add together (a,b), and a carry in (z). The output is a 1 bit sum (s), calculated the same as above, a propagate bit (p), and a generate bit (g). p and g are used to determine whether the carry-in bit will propagate through the current full adder (p), or whether the current adder will generate a propagate bit (g). These are calculated with the following formulas

$$p = a \oplus b$$
$$g = ab$$

Notice that the p and g values are not dependent on the carry in bit. Thus, the carry out for an individual adder can be determined by the a and b input values alone. We chained four of these CLA full adders together, not in ripple, but with glue logic that took advantage of this property.

Our 4-bit CLA consists of four CLA full adders and internal logic to glue them together. The carry in bit of each CLA full adder is determined by the following logic:

$$C0 = Cin$$
$$C1 = Cin \cdot P0 + G0$$
$$C2 = Cin \cdot P0 \cdot P1 + G0 \cdot P1 + G1$$
$$C3 = Cin \cdot P0 \cdot P1 \cdot P2 + G0 \cdot P1 \cdot P2 + G1 \cdot P2 + G2$$

Here, C0 is the carry in for the first CLA full adder, C1 is the carry in for the second, etc. P0,G0 are the propagate and generate bits from the first CLA full adder, P1,G1 are the propagate and generate bits for the second, etc. By using this logic, the carry in for all of the CLA full adders can be determined simultaneously, which means the sum for each full adder

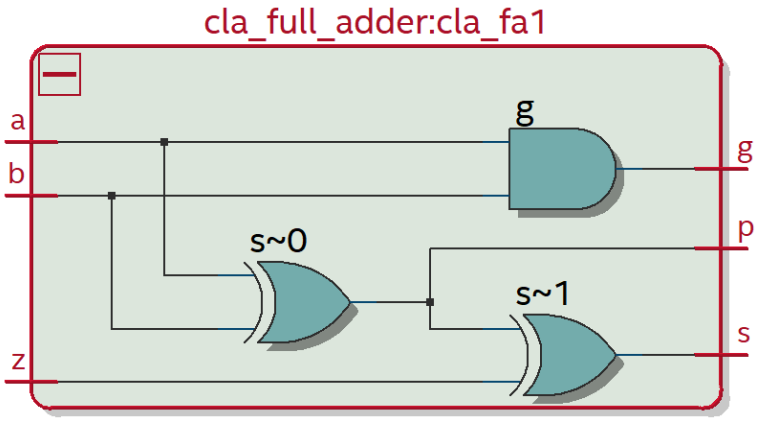
can be computed at the same time. Our 4-bit CLA has an additional set of outputs Pg and Gg, the group propagate and generate bits. They signify whether a carry-in bit propagates through the whole 4-bit CLA (Pg) or if the 4-bit CLA generates a carry bit (Gg). They are calculated with the following:

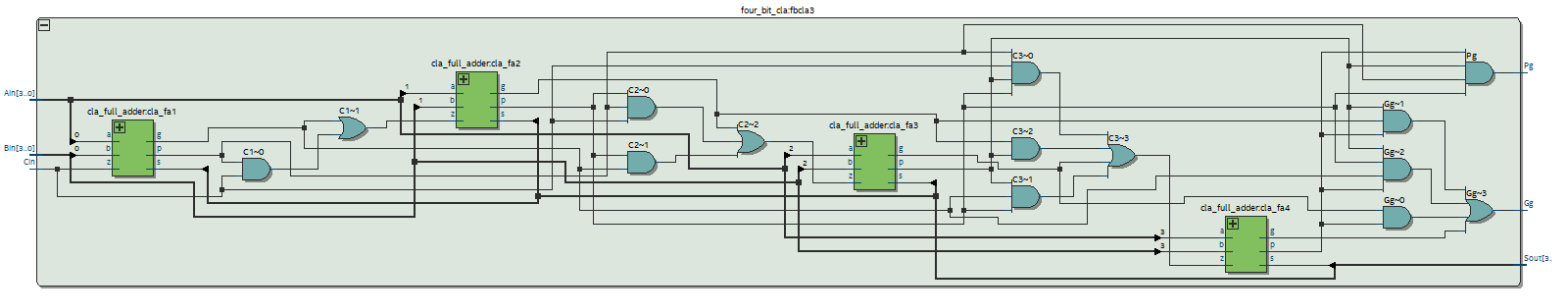
$$PG = P0 \cdot P1 \cdot P2 \cdot P3$$
$$GG = G3 + G2 \cdot P3 + G1 \cdot P3 \cdot P2 + G0 \cdot P3 \cdot P2 \cdot P1$$

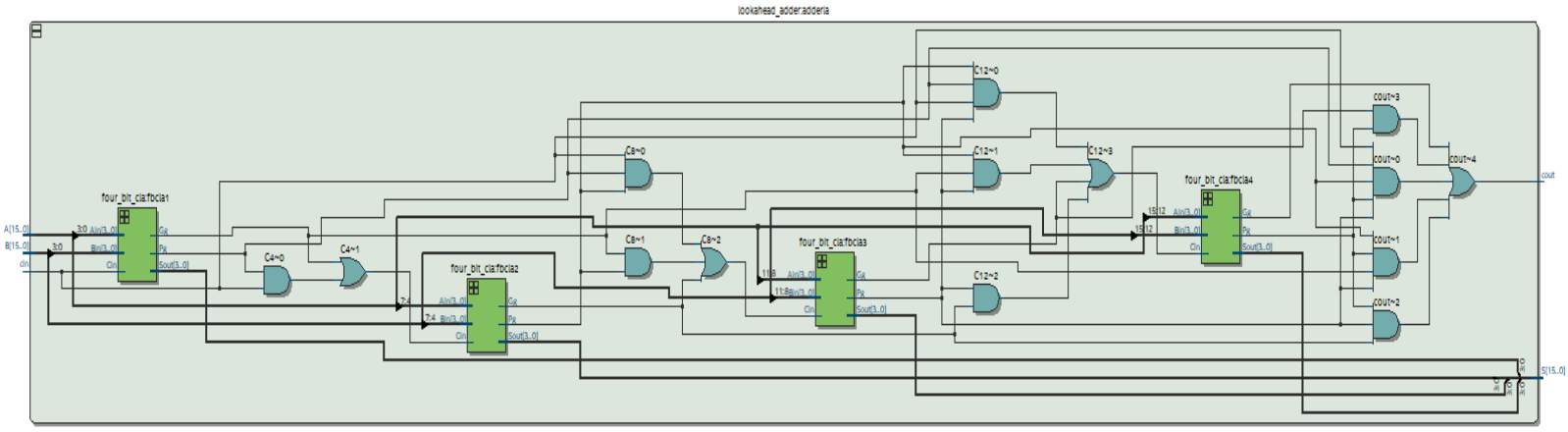
To form our complete 16-bit CLA, four of the 4-bit CLA's are connected together, much in the same way that the four CLA full adders were connected to create the 4-bit CLA. The difference is that the P and G bits used to determine the carry ins for all of the full adders simultaneously is replaced by using the Pg and Gg bits to determine the carry ins for all of the 4-bit CLA's simultaneously. The logic is the same as above, but has been shown below for convenience

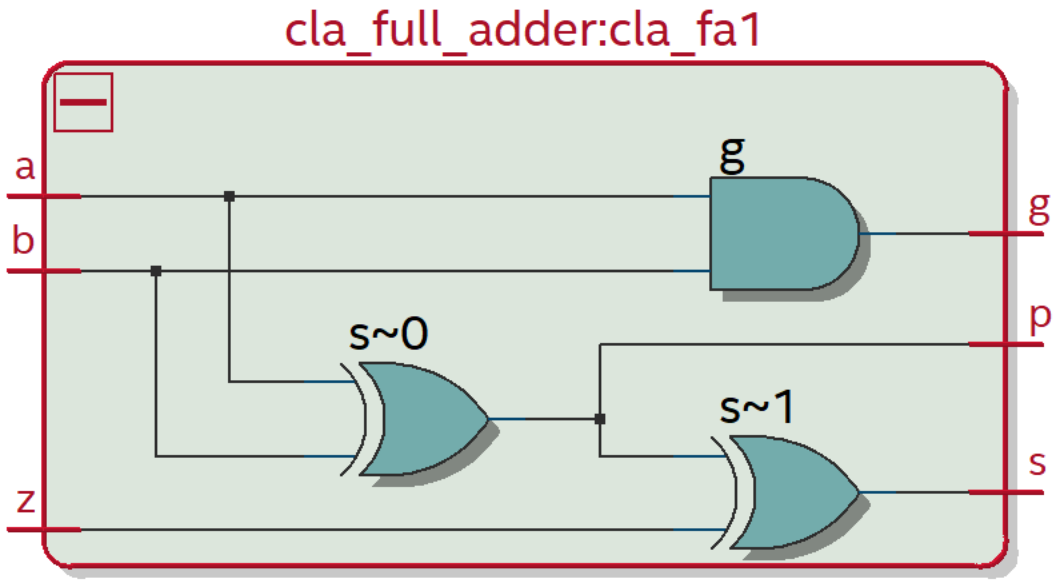
$$C0 = Cin$$
$$C4 = Cin \cdot Pg0 + Gg0$$
$$C8 = Cin \cdot Pg0 \cdot Pg1 + Gg0 \cdot Pg1 + Gg1$$
$$C12 = Cin \cdot Pg0 \cdot Pg1 \cdot Pg2 + Gg0 \cdot Pg1 \cdot Pg2 + Gg1 \cdot Pg2 + Gg2$$
$$C16 = Cin \cdot Pg0 \cdot Pg1 \cdot Pg2 \cdot Pg3 + Gg0 \cdot Pg1 \cdot Pg2 \cdot Pg3 + Gg1 \cdot Pg2 \cdot Pg3 + Gg2 \cdot Pg3 + Gg3$$

Here C0, Pg0, Gg0 is the carry in, propagate, and generate bits for the first 4-bit CLA, C1, Pg1, Gg1 is for the second, and so on. C16 is the carry out bit of our whole lookahead adder. The block diagrams for the Carry Lookahead adder are shown below.



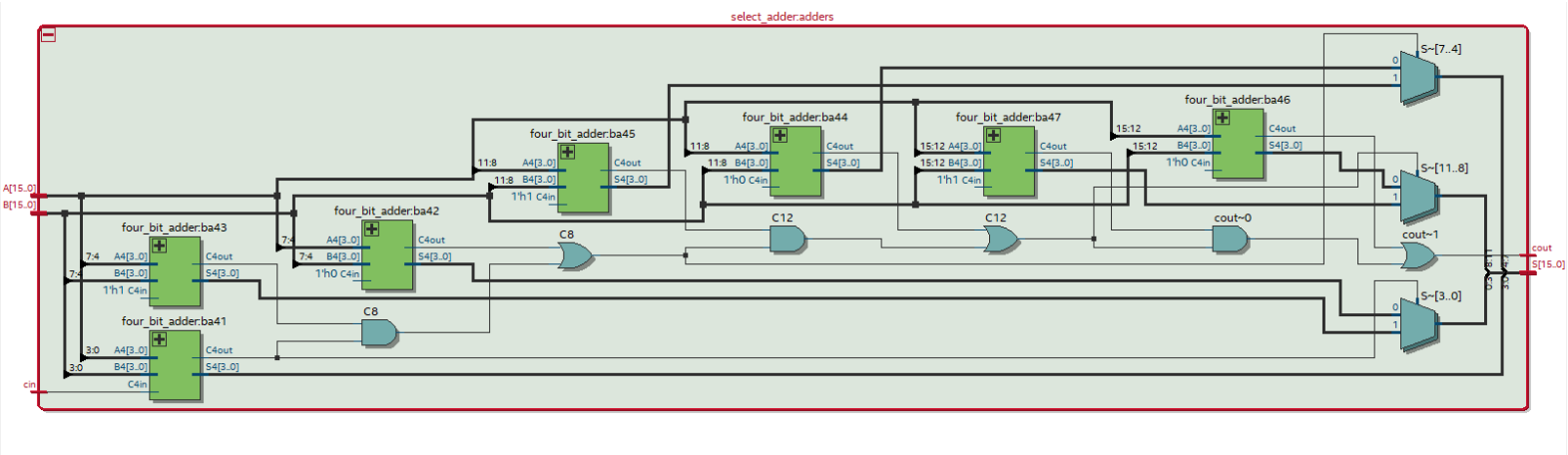



**Carry Select Adder**

The smallest building block of the Carry Select Adder is the 4-bit ripple adder discussed in the *Carry Ripple Adder*. The general design of this adder is divided into 4 segments, each 4 bits long. Within each section, there are two 4-bit ripple adders. One of the ripple adders operates assuming the carry in bit to that section is a 1, and the other operates assuming that the carry in bit is a zero. The sum outputs of both these 4-bit ripple adders are fed through a 2:1 MUX, and the select bit for the MUX is the carry-in for the given four-bit section. In this way, the correct sum is outputted since the carry-in will choose the output to be the correct version. They way that the carry bit is passed between four-bit sections is that the carry out from the previous section is ANDed with the carry out from the 4-bit adder that assumes carry in is 1, then this result is ORed with the carry out from the 4-bit adder that assumes carry in is 0.

This implantation of a 16-bit adder gets its efficiency from the parallel computation of sums that it can compute. Each four-bit section computes two possible sums for the given section, all at the same time. Then, the late sections only must wait for the carry-in to propagate to that section to choose the correct version through a MUX. The key is that the addition in each section does not have to wait to be computed until the carry in arrives. Instead, the carry in must only propagate through a 2:1 mux, instead of the whole 4-bit adder logic. Attached below are the block diagrams for the Carry Select Adder.




**Description of All Modules**

**Module:** full_adder.sv
Inputs: ain, bin, ci
Outputs: co, sout

Description: This module takes in two 1-bit values (ain and bin), as well as a carry in bit (ci), and implements logic to output the binary sum of the three through the sum and carry out (sout and co respectively.

Purpose: This module acts as a building block for the larger adders. This is done by rippling multiple full adders together.

**Module:** cla_full_adder.sv
Inputs: a, b, z
Outputs: s, p, g

Description: This module takes in two 1-bit values (a and b), as well as a carry in bit (z), and implements logic to output the binary sum of the three (s) and also the p and g outputs which signal whether the carry in bit is propagated through or generated by the full adder.

Purpose: This module acts as a building block for the larger adders in the 16-bit lookahead adder.

**Module:** four_bit_adder.sv
Inputs: [3:0] A4, [3:0] B4, c4in
Outputs: [3:0] S4, c4out

Description: This module takes in two 4-bit values (A4 and B4), as well as a carry in bit (c4in), and uses the ripple connection of four full adders to output the 4-bit sum (S4) and carry out (c4out).

Purpose: This module acts as a building block for the larger adders. This is done by rippling multiple four-bit adders together for the ripple adder, and selecting between two four-bit adders for the carry select adder.

**Module:** four_bit_cla.sv
Inputs: [3:0] Ain, [3:0] Bin, Cin
Outputs: [3:0] Sout, Pg, Gg

Description: This module takes in two 4-bit values (Ain and Bin), as well as a carry in bit (Cin), and uses the connection of four CLA full adders described above to output their four bit sum (S[3:0]) and group propagate and generate signals (Pg, Gg), which describe whether a carry in bit has been propagated through the 4-bit CLA or if a carry out bit has been generated.

Purpose: This module acts as a building block for the 16-bit lookahead adder. This is done by connecting four of these together in the method described above.

**Module:** ripple_adder.sv
Inputs: [15:0] A, [15:0] B, cin
Outputs: [15:0] S, cout

Description: This module takes in two 16-bit values (A and B), as well as a carry in bit (cin), and uses the ripple connection of four 4-bit adders to output their 16 bit sum (S[15:0]) and their carry out (cout).

Purpose: This module implements the ripple addition of the two user generated inputs A and B.

**Module:** select_adder.sv
Inputs: [15:0] A, [15:0] B, cin
Outputs: [15:0] S, cout

Description: This module takes in two 16-bit values (A and B), as well as a carry in bit (cin), and uses seven 4-bit adders as well as a series of 2:1 MUX-es to perform 16-bit addition in parallel, and then select the correct sum (S) and carry out (cout) based on the propagation of the carry in through the MUX-es.

Purpose: This module implements the select addition of the two user generated inputs A and B.

**Module:** lookahead_adder.sv
Inputs: [15:0] A, [15:0] B, cin
Outputs: [15:0] S, cout

Description: This module takes in two 16-bit values (A and B), as well as a carry in bit (cin). Using four 4-bit CLAs and some glue logic, the 16-bit sum (S) is calculated simultaneously and the carry out (cout) is generated.

Purpose: This module implements the lookahead addition of the two user generated inputs A and B.

**Module:** router.sv
Inputs: [15:0] A_In, [16:0] B_In, R
Outputs: [16:0] Q_out

Description: This module uses a select input R to select whether the output Q_out is routed to B_In or A_In.

Purpose: This module is important to route either the A signal or the B signal to the output based on the Load signal.

**Module:** reg17.sv
Inputs: Clk, Reset, Load, [16:0] D
Outputs: [16:0] Data_Out

Description: This module sets the output (Data_Out) of the register unit responsible for storing the intermediate and final values of the addition equal to the input D if Load is high. Alternatively, if Reset it high, it will set the output equal to 17 bit HEX 000.

Purpose: This module performs the function of storing the intermediate values of the addition as well as the result.

**Module:** HexDriver.sv
Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This module takes the 4-bit input In0 and converts it to the corresponding 7 bit representation of its HEX value in Out0.

Purpose: This module is used to map the contents of the registers into Hex values so that they can be displayed on the Hex display on the DE-10.

**Module:** control.sv
Inputs: Clk, Reset, Run
Outputs: [16:0] Run_O

Description: This module is a finite state machine with three states (A,B,C). The first state is A and Run_O is 0 in this state. If Run is 1, then the machine transitions to the B state. In the B state the Run_O output is 1 and it will unconditionally transition to state C. In state C, Run_O is 0. It will transition back to state A if Run goes back low.

Purpose: This module is important for executing only one addition operation. The HALT state (state C) and the B state where Run is high for only one clock cycle will start the addition process but will stop it before multiple operations are performed. This makes the Lab perform as designed.

**Module:** adder2.sv
Inputs: Clk, Reset_Clear, Run_Accumulate, [9:0] SW
Outputs: [9:0] LED, [6:0] HEX0 HEX1 HEX2 HEX3 HEX4 HEX5

Description: This module is the top level for this lab. It takes in all the user inputs and calls/connects all the other modules to correctly implement the desired behavior.

Purpose: This module acts as the driver for the lab. It connects all the user inputs to the desired location modules (register unit, adder, etc.). It also chooses which adder is going to be implemented.

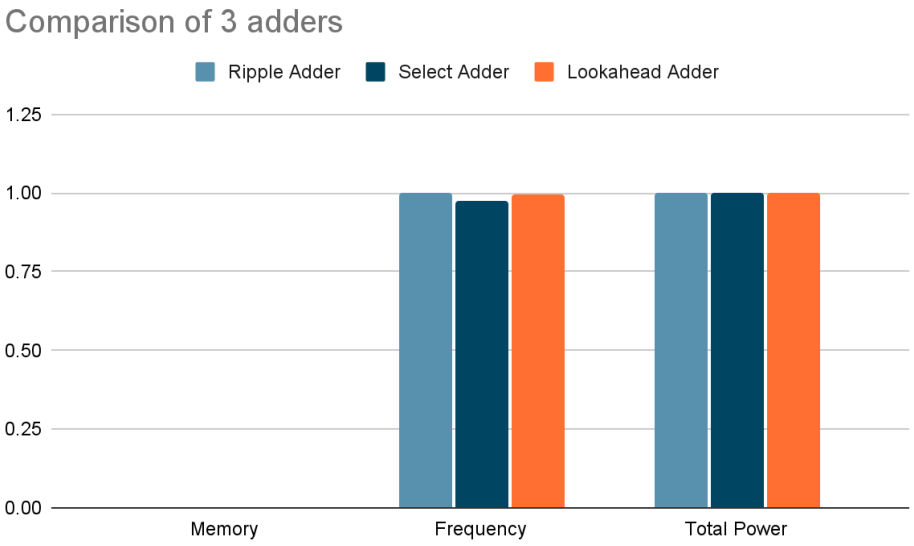
**Trade-offs Between Adders**

The different adders each have their own strengths and weaknesses in the relevant design parameters of area, performance, and complexity. The ripple adder has the simplest implementation of a 16-bit adder. It also takes up the least amount of area. These are both very desirable qualities. However, the other two designs definitely improve upon the performance parameter. The lookahead adder only uses slightly more area than the ripple adder, but has quite a bit more logic (the p and g signals for each CLA building block adder). This makes the implementation a good deal more complex. However, it also greatly improves the efficiency of the adder. Because the P and G signals are all calculated at the same time, the sum for larger bits can be described as O(log(n)) while O(n) for the ripple adder. The select adder takes up significantly more space, almost double that of a ripple adder. The trade-off again comes in performance. Because two versions of each 4-bit sum are calculated in parallel for the select adder, the carry-in only has to propagate through a MUX to choose the correct pre-computed sum.

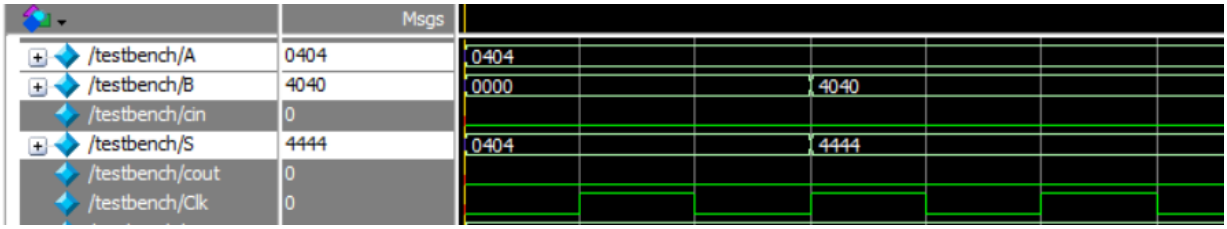
**Prelab Performance Documentation**

value//normalized value with respect to carry-ripple value

| | Carry-Ripple | Carry-Select | Carry-Lookahead |
|---|---|---|---|
| Memory (BRAM) | 0 | 0 | 0 |
| Frequency | 68.45 MHz//1 | 66.84 MHz//1.006 | 68.26 MHz//0.997 |
| Total Power | 134.65 mW//1 | 134.73 mW//1.0006 | 134.48 mW//0.9987 |




**Simulation Trace**




The Simulation trace for this lab was very simple. Register A was initialized as 16'h0404 while register B was initialized as 16'h0000 so the resulting sum was 16'h0404 with cout as 0. Two clock cycles later, register B was changed to 16'4040 and the S changed to reflect the new sum 16'h4444. Because the simulation does not include gate delays, all adders resulted in the same simulation trace.

**Post-Lab Questions**

*Ideal Hierarchy for a CSA:* In this lab, we went with a 4x4 hierarchy but, time wise, this may not be the most ideal. The CSA parallel processes each set of four bits and then chooses which result based off of the previous Cout. Because of this, you would want to minimize the time it takes for the first Cout to be calculated. However, each adder cannot only calculate one bit because then the adder will just become a ripple adder and it will not take advantage of the parallel calculations. The most efficient design would require information on the time it takes to ripple across various amounts of bits in a singular ripple adder and the time it takes for a carry out bit to go through each 'and' and 'or' gate to calculate the new carry out bit. To obtain this information you can sum the maximum delay of each gate. To calculate the sum of two one bit

registers, a xor gate is used. To calculate the carry out, one nand gate and one or gate is needed, as well as the carry in bit. Equipped with this information, the most efficient 16 bit CSA can be chosen.

*Design resources and Statistics:*

| | Carry-Ripple | Carry-Select | Carry-Lookahead |
|---|---|---|---|
| LUT | 78 | 82 | 87 |
| DSP | 0 | 0 | 0 |
| Memory(BRAM) | 0 | 0 | 0 |
| Flip-flop | 20 | 20 | 20 |
| Frequency | 68.45 MHz | 66.84 MHz | 68.26 MHz |
| Static Power | 90.11 mW | 90.11 mW | 90.11 mW |
| Dynamic Power | 1.37mW | 1.48 mW | 1.42 mW |
| Total Power | 134.65 mW | 134.73 mW | 134.48 mW |

The carry-ripple adder has the least amount of LUT because it is the most basic of adders. The carry-select and carry-lookahead adders have additional LUTS because they have additional processes to calculate additional values, like P and G values for the lookahead adders, in order to speed up the addition process. All three adders have the same DSP, memory, and flip-flops, which is as expected because each adder takes in the same amount of logic and outputs the same amount of logic. Frequency describes how quickly each design can run. Because the task we're performing is quite simple, there isn't much difference in the frequencies among the three but we can still see that the carry-select and carry-lookahead adders are faster than the ripple adder. Finally, each design consumes approximately the same amount of power with the lookahead taking the least and the carry-select adder taking the most.

**Conclusion**

The purpose of lab 3 was to look at different types of adders and explore their design and efficiency. All three adders take in 16 bits and a carry in and outputs a 16 bit sum and carry out. However, not every adder uses the same amount of resources or takes the same amount of time. The Carry-ripple is the simplest adder and takes the most time but uses the least amount of gates. The Carry-select and Carry-lookahead adders are both faster than the Carry-ripple adder but both require additional LUTs. Each adder has differences in resources, comparing the trade offs between speed, power, and number of transistors used as no one design is the best in every category.