ECE 385

Spring 2023

Lab 3

Experiment #3

Aleena Majeed & Ali Chaudry
17:50 - 17:55

1. **Introduction**

2. **Adders**
    a. **Ripple Carry Adder**
        i. **Written description of the architecture**
        ii. **Block diagram**
    b. **Carry Lookahead Adder**
        i. **Written description of the architecture**
        ii. **Describe how the P and G are used**
        iii. **Describe how you created the hierarchical 4x4 adder**
        iv. **Block diagram**
            1. **Block diagram inside a single CLA (4-bits)**
            2. **Block diagram of how each CLA was chained together**
    c. **Carry Select Adder**
        i. **Written description of the architecture of adder**
        ii. **Describe at a high level how the CSA speculative computes multiple sums in parallel and rapidly chooses the correct one later.**
    d. **Written description of all .SV modules**
    e. **Describe at a high level the area, complexity, and performance tradeoffs between the adders**
    f. **Document the performance of each adder by creating a graph as specified in Prelab part C**
    g. **Annotated simulation trace**
    h. **Optional for extra credits, perform a critical path analysis and compare this to the theoretical understanding of each adder**
3. **Post-lab questions**
4. **Conclusion**
    a. **Describe any bugs and countermeasures taken during this lab.**
    b. **Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given material which can be improved for next semester?**
    c. **Any additional summary**

INTRODUCTION

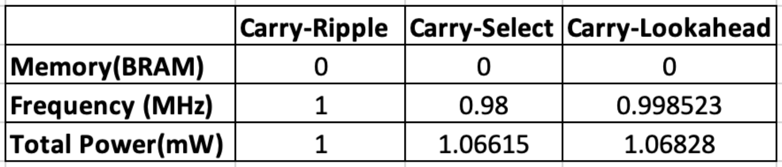
Summarize the high-level function performed by the three adders

Carry-Ripple-Adder - A one-bit full adder consists of three inputs, two bits to add (a and b) and a carry-in, cin. The two outputs, cout and the sum, s. To construct a ripple carry adder, multiple full adders are cascaded with the carry output from each full adder connected to the carry input of the next full adder in the chain. The sum is not computed until the carry in is accounted for so there is a propagation delay.

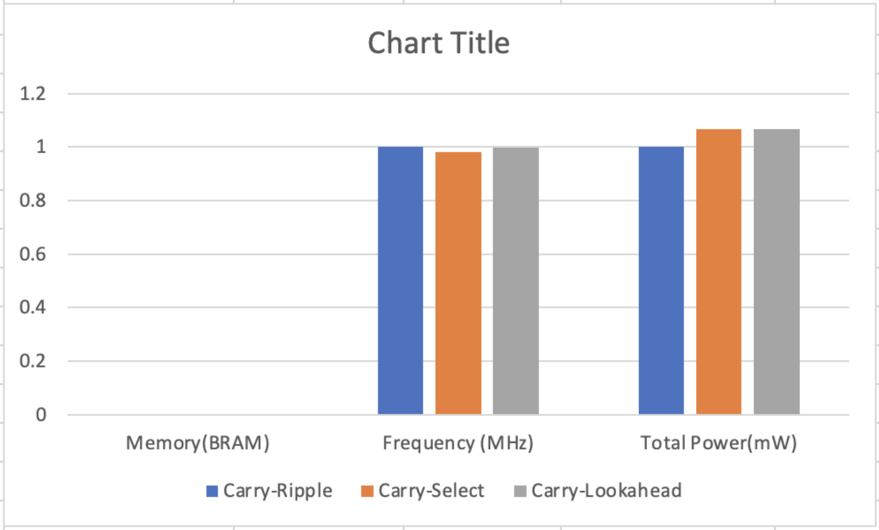
Carry-Look-Ahead-Adder - There is less delay in the CLA since the carry signals are calculated in advance based on the input signals. The carry signals are generated based on two cases. First, when both a and b bits are 1 or when one of the bits is 1 and the carry in is 1. Using this logic, you can derive equations to get G and P which are used to represent carry and propagate terms. The carry propagation is propagated to the next level whereas the carry generator is used to generate the output carry, regardless of input carry.

Carry-Select-adder - This adder consists of 7 full adders, 3 2-to-1 muxes, 3 AND gates and 3 OR gates. Each adder has 3 inputs, A, B, and a Cin value. However, for this design, the adders have a hardcoded Cin value of either 1 or 0, and compute as normal. Each mux takes the sums of the the ripple adders (one with a carry in of 1 and the other with carry in of 0), and uses the output of combination logic consisting of the Couts from the previous adders as the select bit for the mux, deciding which sum we want to select.

Prelab*:*



	Carry-Ripple	Carry-Select	Carry-Lookahead
Memory(BRAM)	0	0	0
Frequency (MHz)	1	0.98	0.998523
Total Power(mW)	1	1.06615	1.06828



Above is a graph of the normalized values of the BRAM, Frequency, and Total Power of each ripple adder

ADDERS

Carry-Ripple-Adder:

- **written description of architecture**
  Among the various designs of binary adders, the CarryRipple Adder (CRA) is probably the easiest to implement. This adder is composed of N full-adders, which are single-bit versions of binary adders. Each full-adder takes three 1-bit bits (A, B, and Cin) and utilizes logic gates to produce a single-bit sum (S) and a single-bit carry-out (Cout). The N full-adders are connected in series by connecting the Cout of the previous adder to the Cin of the next to create a N-bit carry ripple adder. When binary inputs are provided, the full-adder of the least significant bit (LSB) generates a sum (S0) and a carry-out (C1). This carry-out is then fed into the carry-in of the second full-adder, which produces a second sum (S1) and a second carry-out (C2). This process continues through all N bits of the adder until the full-adder of the most significant bit (MSB) generates its sum (SN-1) and carry-out (Cout). While the CRA is easy to design and implement, it suffers from a lengthy computation time. Each full-adder must wait for the previous adder to generate a carry-out before the next adder can generate the correct sum and Cout. This results in an increase in the propagation delay of the CRA as N increases. To reduce the computation time, parallelization of the computation of carry-out bits is required, which is precisely how a carry-lookahead adder operates.

- **Block Diagram**

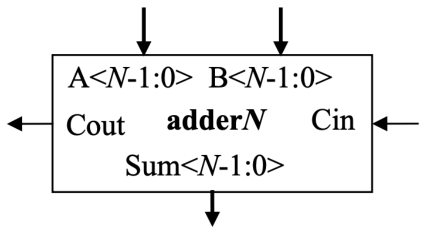


Figure 1: *N*-bit Binary Adder Block Diagram

Above is the block diagram of a single adder, which has 3 inputs: A, B, and Cin, and two outputs: Cout and Sum. This adder will be constructed in multiple ways to create different types of adders.

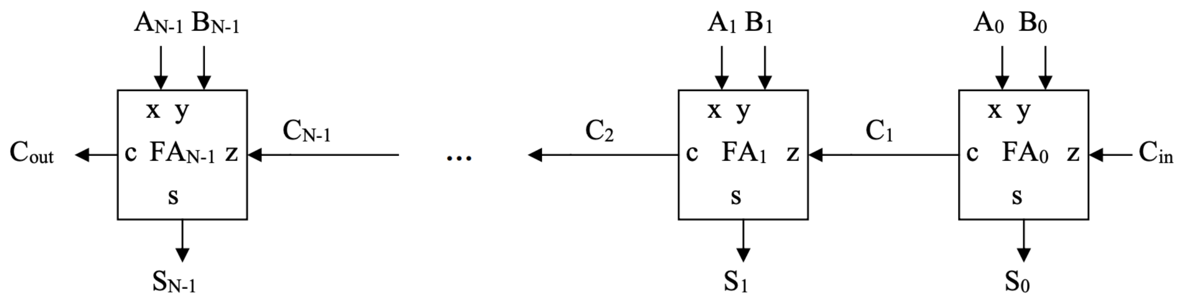
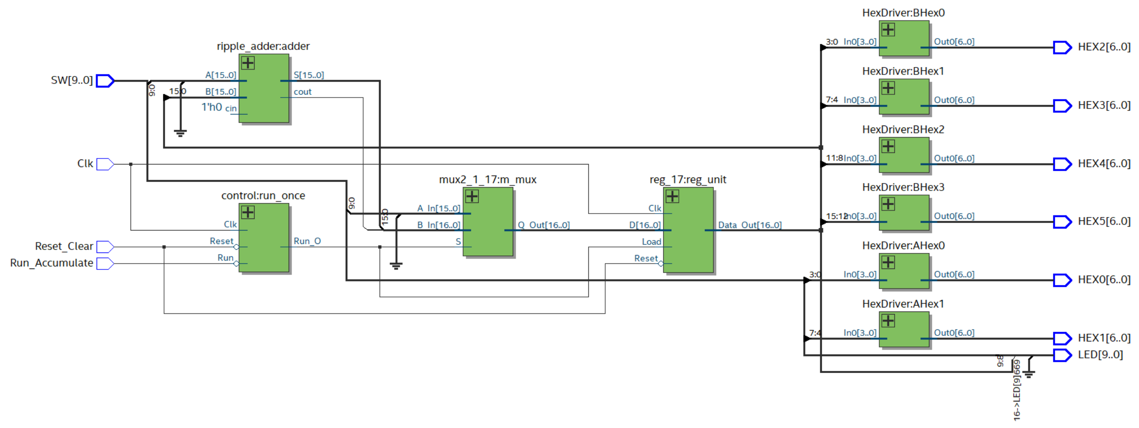


Figure 3: *N*-bit Carry-Ripple Adder Block Diagram

Above is the block diagram for an N-bit carry ripple adder. There is an A and B input (the least significant bit of each number that's being added), Cin (the carry out of the previous adder), Cout (the carry out of the current adder) and S (the sum).



Above is the RTL block diagram for the ripple-carry adder generated from Quartus

Carry-Look-Ahead-Adder:

- **Written description of architecture**

- **Description 4x4 hierarchical design**
  To construct an N-bit Carry-Lookahead Adder (CLA), initially, we thought to design it similarly to a ripple adder and make a more "flat" design. However, as our Cin as explained earlier, the larger our adder gets, the longer it takes for the adder to compute the final sum and Cout because the carry in of each adder is dependent on the previous one. As a solution, we can create the Cin values for the adders by using some sort of combination logic between variables, to then construct 4-bit CLAs, to then hierarchically combine them to form a larger CLA. In

this lab, a 4x4-bit hierarchical CLA will be implemented, in which the 16-bit inputs A and B are divided into groups of 4 bits. Each group of 4 bits then undergoes a 4-bit CLA, and each 4-bit CLA consists of 4, 1-bit adders. Most importantly, we must add that the 4-bit CLA generates two additional signals, the group propagate (PG) and group generate (GG). As stated above, we do not want to mimic the carry ripple adder, as the execution time will be too large from connecting Cin's to Couts, instead, the Cins of the 4-bit CLAs should be generated using the PGs and GGs. This is similar to how carry bits are generated within a 4-bit CLA. Therefore, a copy of the 4-bit Carry-Lookahead Unit in the 4-bit CLA can be directly utilized, with the inputs being the PGs and GGs from the 4-bit CLAs at the upper level. The use of P and G will be described in more detail below.

Observe that this is the same as how we generated the carry bits within a 4-bit CLA. Therefore, we can directly take a copy of the 4-bit Carry-Lookahead Unit in the 4-bit CLA, but instead of the inputs coming from full adders, this time the inputs are the PGs and GGs from the 4-bit CLAs at the upper level.

- **Description of P and G**
  Rather than using the carry-in values from the previous adder, the Carry-Lookahead Adder (CLA) created a carry-in value by using combinational logic involving the generating (G) and propagating (P) variables. Each bit of the CLA creates these P and G values based on the immediate available inputs (A and B) and predicts its carry-out for any value of its carry-in. A carry-out is generated (G) only when both available inputs (A and B) are 1, regardless of the carry-in. This idea can be logically implements by $G(A, B) = A \cdot B$. In contrast, a carry-out can be propagated (P) if either A or B is 1, which is expressed as $P(A, B) = A \oplus B$. By defining P and G, we can then construct a Cin, Ci, to then devise a possible Cout value: $Ci+1 = Gi + (Pi \cdot Ci)$. As shown, the Cout value can still depend on the Cin value, but it also must rely on the P and G values as well in order to prevent this from becoming a ripple carry adder.

- **Block Diagram**

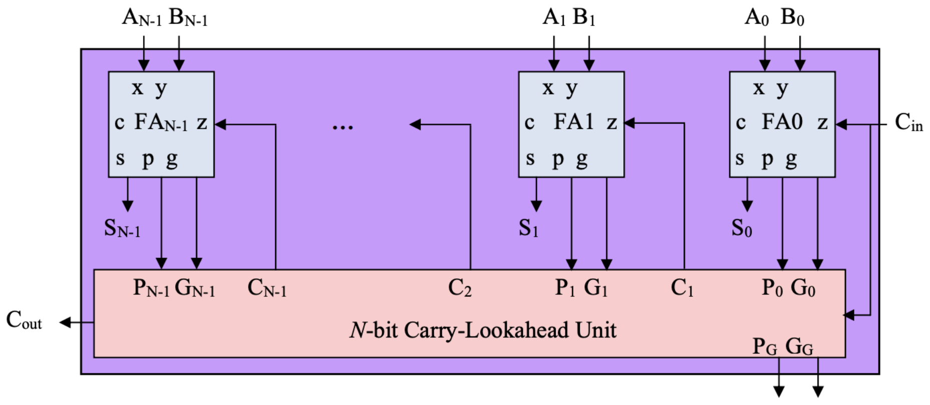
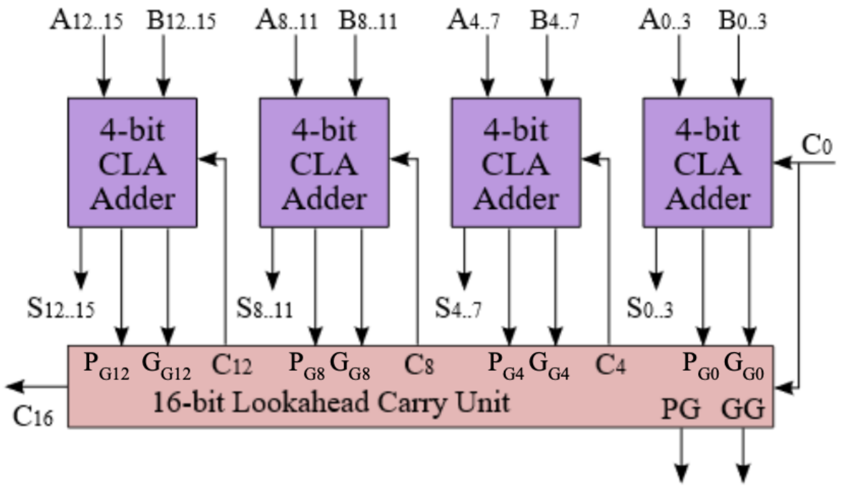


Figure 4: *N*-bit Carry-Lookahead Adder Block Diagram

Figure 4

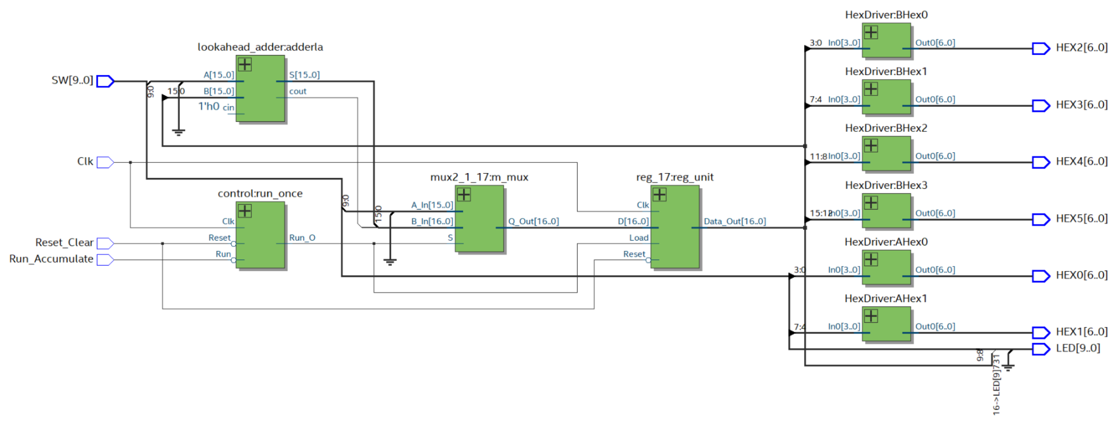
Above is the Carry Look Ahead adder block diagram on the 1-bit level. Each gray block is a single adder that computes the sum of the single bit input x and the single bit input y



$A_{12..15}$ $B_{12..15}$
$A_{8..11}$ $B_{8..11}$
$A_{4..7}$ $B_{4..7}$
$A_{0..3}$ $B_{0..3}$
4-bit CLA Adder
C_0
$S_{12..15}$
$S_{8..11}$
$S_{4..7}$
$S_{0..3}$
P_{G12} G_{G12} C_{12} P_{G8} G_{G8} C_8 P_{G4} G_{G4} C_4 P_{G0} G_{G0}
C_{16}
16-bit Lookahead Carry Unit
PG GG

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

Above is the block diagram for the Carry Look Ahead adder on the 4x4 level. This CLA consists of 4, 4 bit adders (the adders shown in figure 4 above). Each 4-bit adder no longer has a cin value connecting it to the previous adder, but instead it's Cin value depends on P and G.



Above is the RTL block diagram for lookahead adder which was generated from Quartus

Carry-Select-Adder:

- **Written Description**
  A carry select adder is a type of arithmetic combinational logic circuit that is utilized to add two N-bit binary numbers and produce both their N-bit binary sum and a 1-bit carry. The adder functions in a similar manner to a ripple carry adder; however, the carry select adder diverges in design by circumventing the propagation of the carry through as many full adders as the ripple carry adder does. This reduction in carry propagation signifies that the time required to add two numbers can be shorter.

- **High Level Description of CSA**
  The fundamental concept behind an N-bit carry select adder is to avoid sequentially propagating the carry from bit to bit. To accomplish this, two adders run in parallel: one with a carry input of 0, and the other with a carry input of 1. Subsequently, the actual carry input created is used to select between the two parallel adders' outputs. This configuration enables all adders to work simultaneously. Although having two adders for each result bit is wasteful, the N-bit adder can be arranged to use 2*N/M-1 M-bit ripple carry adders in parallel. It should be noted that the adder for the least significant bits will always have a carry input of 0, rendering parallel addition unnecessary in this instance.

- **Block Diagram**

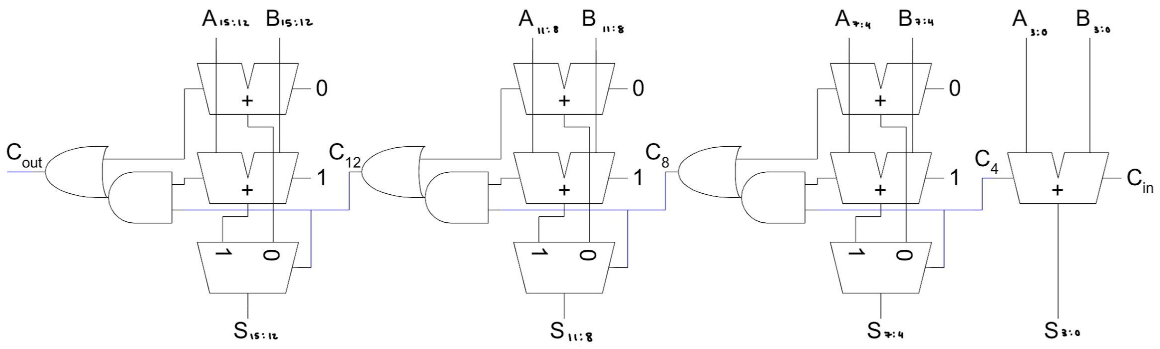
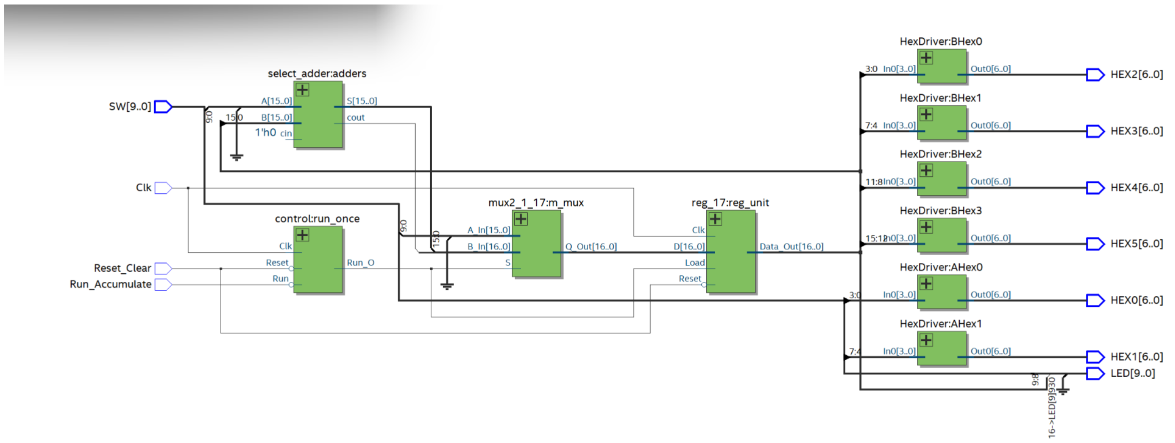


Figure 5: 16-bit Carry-Select Adder Block Diagram

Above is the block diagram for the Carry Select Adder. This adder contains 7 adders, 4 muxes, 3 AND gates and 3 OR gates. This adder computes the sum of A and B, one with a Cin of 0 and the other of 1. Then, a mux is used to determine which sum to use depending on the Cout of the previous adder block which is determined by simple combination logic of the Cout's of each adder.



Above is the RTL block diagram for the carry select adder which was generated by Quartus

- **Written Description of all .SV modules**
  - **select_adder.sv:** For our select adder, we have 4 different modules:
    - muxUnit: which holds the input and outputs to the 3 different muxes in the adder
      - ***input logic S,***
        input logic [3:0] A_In,
        input logic [3:0] B_In,
        output logic [3:0] Q_Out);

        - full_adder: which holds the inputs and outputs for each 1-bit adder as well as the logic for how the sum and Cout values are to be computed
            - ***input x,***
              input y,
              input cin,
              output logic s,
              output logic cout

        - four_bit_ra: creates our 4x4 adders that computes the addition of all 16 bits
            - ***input [3:0] x,***
              input [3:0] y,
              input cin,
              output logic [3:0] s,
              output logic cout

        - select_adder: The top module is the one that contains the inputs and outputs to the whole adder, and most importantly for the CSA, this module computes the Cout values that are to be feeded into each mux as the select bit to determine which sum is desired.
            - ***input [15:0] A, B,***
              input   cin,
              output [15:0] S,
              output   cout
    - **ripple_adder.sv:** Next is our ripple adder. This file has many of the same modules at the select adder. This file contains 3 different modules:
        - full_adder (same as select adder): module which is responsible for creating the variables for a 1-bit adder
            - ***input x,***
              input y,
              input cin,
              output logic s,
              output logic cout

        - four_bit_ra (same as the select): which is responsible for combining each 1-bit adder to create a 4-bit adder
            - ***input [3:0] x,***

input [3:0] y,
input cin,
output logic [3:0] s,
output logic cout

    - ripple adder (the top level): module which is responsible for connecting each 4-bit adder that was created in the previous module to create a 16-bit ripple adder by connecting the Cin's to the Cout's of the previous adder.
        - ***input [15:0] A, B,***
          input cin,
          output [15:0] S,
          output cout

- **reg_17.sv:** The purpose of this file is to reset and load the values that are input in the switches when the load button is pressed, or to clear the values when the reset button is pressed
- **mux_1_17.sv:** This module implements a 17-bit multiplexer
- **lookahead_adder.sv:** This purpose of this file is to make a 16-bit lookahead adder that does not depend on the Cout of the previous adder, but instead to depend on variables titled P and G which are assigned through logic containing the inputs. The file contains 3 different modules:
    - full_adder2 (same as the previous two adders): module which is responsible for creating the variables for a 1-bit adder. However, this adder uses two new variables called P and G to and uses logic containing the inputs to create these variables so they can be used as the Cin for each 1 bit adder without having to directly connect the previous Cout of and adder to the Cin of the next.
        - ***input x,***
          input y,
          input cin,
          output logic s,
          output logic cout

    - four_bit_ra (same as the previous two adders): which is responsible for combining each 1-bit adder to create a 4-bit adder. This module also uses new variables PG and GG, which are created through combinational logic with the other variables, and does a series of logic with them to create a Cin to the next adder.

            - ***input [3:0] x,***
              input [3:0] y,
              input cin,
              output logic [3:0] s,
              output logic cout

        - lookahead_adder (top module): This module combines all adders to create a 16-bit adder. The main difference of this adder compared to the other is once again that the Cin of the current adder does not depend on the Cout of the previous.
            - ***input [15:0] A, B,***
              input   cin,
              output [15:0] S,
              output   cout

    - **HexDriver.sv:** This module simply contains if statements to apply the correct input/output logic so that the hex drivers of the FPGA get the correct values depending on which switches are high
    - **control.sv:** This module holds a simple state machine that creates a one clock cycle long event to convert the switches
    - **adder_toplevel:** This is the top level module file. This file essentially connects each different file in this lab through the instantiation between the ports and variables created in the module. This module also assigns the hex display signals by taking the output signals and pairing them to the LEDs and hex drivers.
    - **testbench.sv:** This last module is the testbench file that allows us to test our code in the ModelSim software. This is done by hardcoding values for the inputs and the cin values of the adders, so that we are able to see the outputs of the addition on the simulator.
- ***<u>Describe at a high level the area, complexity, and performance tradeoffs between the adders</u>***

  <u>CARRY-RIPPLE ADDER</u>

  For this adder, it is the most straightforward to implement, as you can simply just connect the adders through their Cin and Cout values, thus having the ripple effect. However, despite the simplicity of the design, this adder is the slowest out of the 3 adders because in order for the adder to be able to compute the correct sum and Cout value, it depends on the Cin of the previous adder. The behavior of the time complexity of the ripple adder can be shown as O(n). This relationship is determined by the number of gates needed to implement this adder and the fact that it depends on n, means that the time complexity of this adder depends on the number of bits. This adder may have the least amount of speed but it consumes the least amount of power as well, as the circuit complexity is fairly simple. This adder uses the least amount of look up tables.

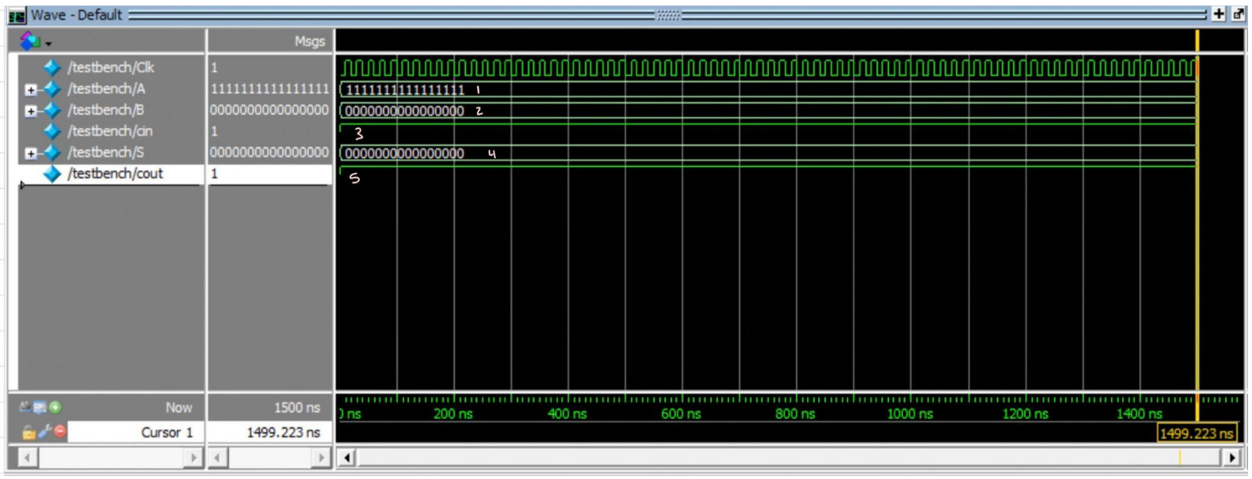
CARRY-LOOKAHEAD-ADDER

Compared to the carry ripple adder, the carry lookahead adder is slightly more complex due to the fact that you are not using Cout values of previous adders as the Cins for the next adder. Instead, we construct generating (G) and propagating (P) values and use logic containing these two values in order to create a carry out to be used for the next adder. But it is important to note that this Cout is logically created, not directly connected between adders. However, despite the more complex design, the lookahead adder is faster than the ripple adder for the exact reason that each adder does not directly depend on the Cin of the other adders, but instead also uses the P and G values. The time complexity of this adder is independent of the number of bits. The relationship described is O(1). This adder takes up the most area as it contains the most gates and uses the most boolean logical expressions for its operations, but it is also the fastest out of the three. This adder uses the most amount of look up tables.

CARRY-SELECT-ADDER

Finally, the carry select adder is possibly the most complex out of all the adders. For the select adder, we connect the Cout values for each of the adders (similar to the ripple adder) and put these Cout values through logic before reaching our final and overall Cout value (similar to the purpose of P and G in the lookahead adder). This combines some of the aspects of the first two adders, and in addition to the muxes, which are used to determine which sum we want to select based on the carry in. The time complexity of this adder is represented by O(log n), so it depends on the number of bits, which means that it is slower than the carry lookahead but faster than the ripple adder. It also consumes less power than the CLA. This adder uses more look up tables than the CRA, but less than the CLA.

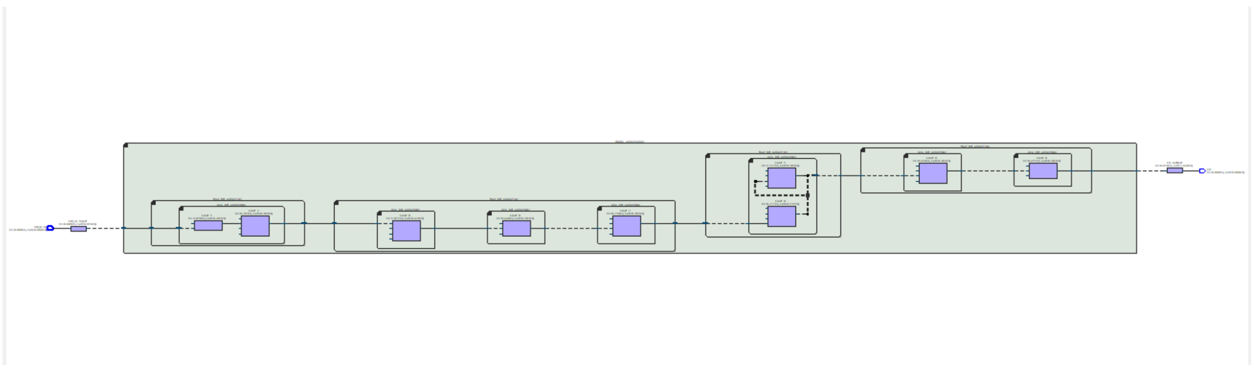
- ***Document the performance of each adder by creating a graph as specified in Prelab part C***
- **Annotated simulation trace**



Above is the ModelSim waveform where the input A = -1 and input B = 0, with a carry in of 1.

1. Number 1 shows the first input to the adder, it currently contains the value 1111111111111111 (-1). This value was hardcoded in our testbench file, or loaded in using the switches on our FPGA.
2. Number two shows the second input to the adder, in the simulation above it currently contains the value 0000000000000000 (0). This value was hardcoded in our testbench file, or can be loaded in using the switches on our FPGA
3. Number 3 is the value of the Cin, this is the initial Cin value to the very first 1-bit adder. In this case we've hardcoded the value to be 1.
4. Number 4 is the value of the sum (the sum coming out of the final adder), which in this case is 0000000000000000 (0). This is the value we expected from adding the inputs -1 + 0 + 1 (the carry in).
5. Number 5 is the value of the Cout for the final adder. In this case, the Cout value is 1 in this case which is what we expected with the values of the input.

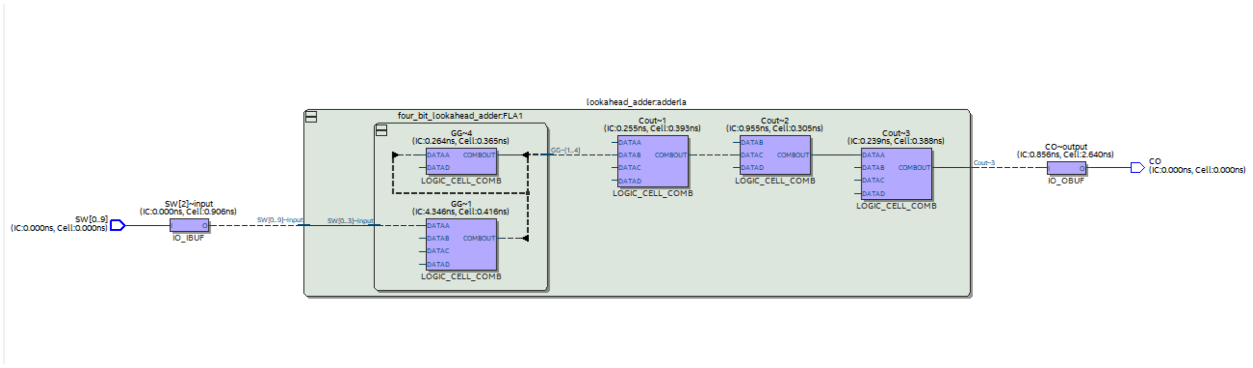
- **Extra Credit**
    - CARRY RIPPLE ADDER



Above is the critical path diagram for the Carry Ripple Adder generated by Quartus

For the carry ripple adder, we expect the adder to have the longest critical path analysis due to the fact that in order to compute the sums, the carry ripple adder depends on the output of the previous adder. As stated in the sections above, in order for the ripple adder to compute the correct final sum and cout (the sum and cout of the MSB), the adder must wait for all of the adders before it correctly computes their output values as well. Despite the timing issues, the CRA is the easiest to implement and requires the least amount of area. Therefore, the CRA has the largest critical path.

- CARRY LOOKAHEAD ADDER



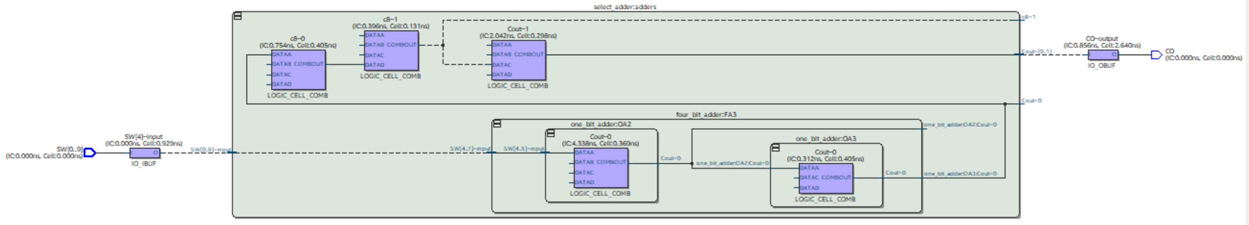
lookahead_adder:adderla
four_bit_lookahead_adder:FLA1
SW[0..9] (IC:0.000ns, Cell:0.000ns)
SW[2]~input (IC:0.000ns, Cell:0.906ns)
IO_IBUF
SW[0..9]~input
SW[0..3]~input
GG~4 (IC:0.264ns, Cell:0.365ns)
GG~1 (IC:4.346ns, Cell:0.416ns)
GG~[1..4]
Cout~1 (IC:0.255ns, Cell:0.393ns)
Cout~2 (IC:0.955ns, Cell:0.305ns)
Cout~3 (IC:0.239ns, Cell:0.388ns)
DATAA
DATAB
DATAC
DATAD
COMBOUT
LOGIC_CELL_COMB
Cout~3
CO~output (IC:0.856ns, Cell:2.640ns)
IO_OBUF
CO (IC:0.000ns, Cell:0.000ns)

Above is the critical path diagram for the Carry Lookahead Adder generated by Quartus

Now, observing the carry lookahead adder, we expect this adder to have a shorter critical path than the carry ripple adder, as the CLA does not depend on the outputs of the previous adder in order to compute the outputs of the current adder. This is once again because the CLA is able to predict these values using P and G.

- <u>CARRY SELECT ADDER</u>

  Finally, we have the carry select adder. This adder utilizes the aspect of predicting the output using a P and G variable rather than using the Cout of the previous adder, as well as using the implementation of muxes. Each adder has a hard coded carry in bit of either 1 or 0, and the mux determines which sum to choose based on the carry bit. This means that the adder is slower than the CLA, and there are additional gates with the muxes, but it is still faster than the ripple adder because of the lack of dependency on the previous adders. Therefore, the critical path of the CSA is the second largest.



<u>Above is the critical path diagram for the Carry Select Adder generated by Quartus</u>

Overall, were able to conclude that the CRA has the highest computation time, this computation time will increase as N grows, despite the simplicity of the design. Therefore, leading the CRA to have the largest critical path. Next, we have the CSA due to the fact that it utilizes the same method of looking-ahead as the CLA, but also needs muxes to implement, causing extra propagation delay. Finally, the CLA has the smallest critical path diagram, despite being more difficult to design and implement, because it takes the least amount of time to compute accurate outputs because it is designed to not rely on previous outputs and has a smaller propagation delay.

POST-LAB

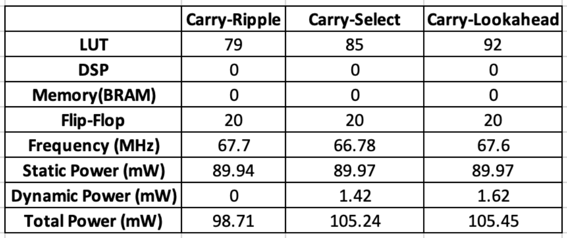
- *In the CSA for this lab, we asked you to create a 4x4 hierarchy. Is this ideal? If not, how would you go about designing the ideal hierarchy on the FPGA (what information would you need, what experiments would you do to figure out?)*

When designing a CSA utilizing a 4x4 hierarchy, there are many factors that determine efficiency such as complexity, area, power consumption, and speed. A 4x4 design is efficient for scalability of the CSA design. Additional 4-bit blocks can be added to the circuit to increase its size without requiring major changes to the underlying design and can reduce the number of logic gates required.

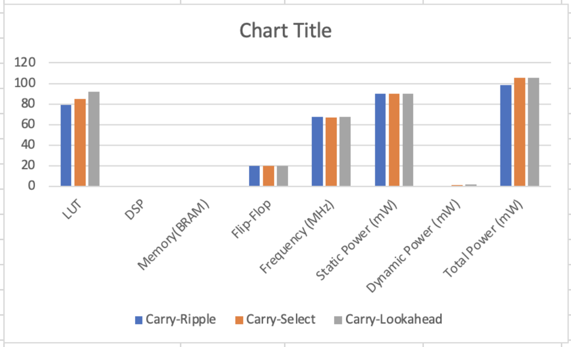
Overall the efficiency depends on the speed and power consumption of the circuit and increasing the efficiency of one comes at the cost of the other. For example, if we were to use a 2x8 hierarchy, then there would be 2 propagation delays compared to the 4 propagation delays of the 4x4, so it would be faster but the 4x4 would use less power so there is a tradeoff.

To design the ideal hierarchy, it would be helpful to have access to the area, power consumption and time analysis of each of the adders so that this way we can see which is the most efficient (i.e., which one has the largest area, which one takes the longest, which one uses up the most power) depending on the structure of our hierarchy. We could just do some trial and error experiments and simply compare the information we obtain from the experiments for each adder to finally determine the most ideal design.

- **Design statistics table for each adder**



	Carry-Ripple	Carry-Select	Carry-Lookahead
LUT	79	85	92
DSP	0	0	0
Memory(BRAM)	0	0	0
Flip-Flop	20	20	20
Frequency (MHz)	67.7	66.78	67.6
Static Power (mW)	89.94	89.97	89.97
Dynamic Power (mW)	0	1.42	1.62
Total Power (mW)	98.71	105.24	105.45



<u>Observe the data plot and provide an explanation to the data i.e. does each resource breakdown comparison from the plot make sense?</u>

It is reasonable to assume that the Carry-Select-Adder (CSA) and Carry-Look-Ahead Adder (CLA) would utilize more lookup tables and have a higher total power consumption compared to the Carry-Ripple Adder (CRA) due to the former designs' increased circuit complexity and number of logic operations required to process input and output data. Specifically, the CSA and CLA designs require more Boolean logical expressions than the CRA, which results in a greater number of lookup tables.

The frequency of the CSA and CLA is expected to surpass that of the CRA since they employ a parallel carry mechanism, which reduces the carry propagation delay and improves performance. However, the experiment results show that the CRA has the highest frequency, which suggests that the CRA design may have been optimized for speed and has a smaller overall area than the CSA and CLA.

As the number of bits being added increases, the size of the design increases as well, and the performance and power consumption differences between the adder designs become more evident. In this scenario, the frequency data becomes more accurate, and the CSA and CLA designs are anticipated to exhibit superior performance compared to the CRA.
The reason that the DSP and memory consumption are zero for all adders is that none of the adder operations requires memory access. Additionally, since the number of flip-flops is directly

related to the number of bits being added and all adders have the same-sized inputs, it is expected that they would have the same number of flip-flops.

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?

The data is not complying with the theoretical design expectations since the CRA's frequency is higher than that of the CSA's and CLA's. However, this could be attributed to the fact that the CRA was designed more for speed optimization and it uses less area. The disparity of the frequency between the adders becomes more apparent as more bits are being used to perform arithmetic operations. For example, if we performed operations on 64-bit numbers, then the frequency of the CSA and CLA would be a lot higher than that of the CRA.

Which design consumes more power than the other as you expected, why?

The design that consumed the most power was the CLA, followed by the CSA, and then the CRA. This makes sense as the CLA requires the most complex circuitry followed by the CLA, and CRA. The CLA requires the most power as it takes up the most area and uses the most gates. The CSA and CRA have the least amount of gates because they require less logic gates to generate the carry signals and as a result take up less area which means they consume less power.

CONCLUSION

- *Describe any bugs and countermeasures taken during this lab*

  Some bugs that we encountered were how to implement the 4x4 level hierarchy for the lookahead adder, as we had to understand the different levels of the adders and how each one played a role in creating the look ahead adder. We had to understand that we needed to create a module to create a 1-bit adder, and then a module to group 4 of these 1-bit adders to make a 4x1 module. Then finally, we needed to make the top level module that combines each 4x1 module to create the full 16x16 adder.

  We also had issues with our S when running our ModelSim for our carry select adder. This was because we accidentally assigned our bits to go to go from [16:1] instead of [15:0]. This bug caused us to have floating values because the connections were incorrect as it was the wrong number of bits being instantiated.

- *Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester?*

  For the carry select adder, the block diagram was labeled with the inputs A and B and the output sum S, and the bit numbers for each were labeled. However, these bit numbers were incorrect, as the packed array ranged from [16:1] when it should've been [15:0].
- *Any additional summary you want to include?*

  N/A