ECE 385
Fall 2021
Experiment #3

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

Michael Stoens (mstoens2)
Ralph Balita (rbalita2)
Lab Section: AB3

1) Introduction: Summary of high level junction performed by the three adders.

The purpose of experiment #3 is to design and implement three different types of 16-bit adders. The three types of adders implemented in the experiment are a Ripple Adder, a Carry Lookahead Adder and a Carry Select Adder. Each of these adders accomplishes the same task of adding two 16-bit values together and displaying the results in hexadecimal on the seven segment displays of the DE-10 development board. By implementing three different types of adders, we were able to see the benefits and drawbacks of each design first-hand.

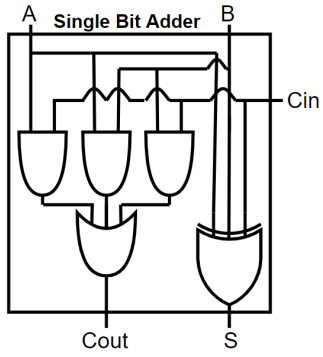
2) Adders

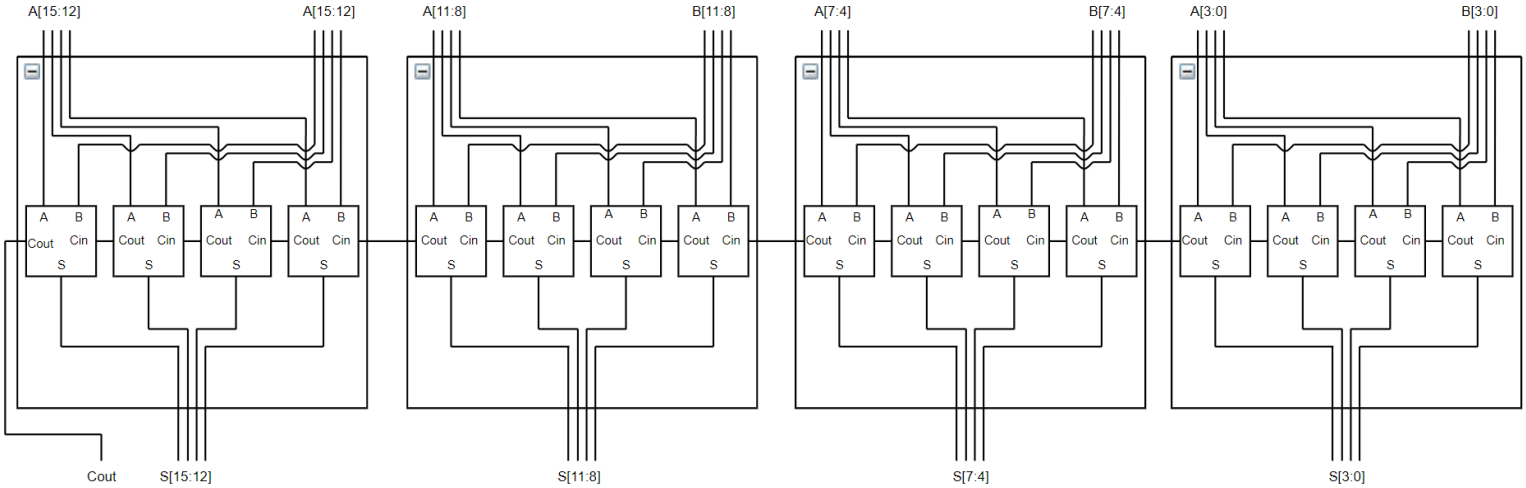
Carry Ripple Adder

i) *Written description of the architecture of the adder*

The Carry Ripple Adder is the simplest of the three adders implemented in this experiment. We implemented our Carry Ripple Adder with a 4x4 Hierarchical Design of single bit adders. Each of the single bit adders are connected in series with the Cin connected to the Cout of the previous adder. This means that the carry bits must “Ripple” through all 16 single bit adders before the result is computed. This ripple of carry bits propagating through the adders causes the Ripple Carry adder to be slow in relation to other designs.

ii) *Block Diagram*





Carry Lookahead Adder

i) ***Written description of the architecture of the adder***

The Carry Lookahead was the second adder we implemented in this experiment. Like the Carry Ripple Adder, it is implemented using 4x4 Hierarchical Design of single bit adders, however, instead of each single bit adder outputting a carry out bit, we output two signals: Generate and Propagate. These signals are used to determine carry value to the next single bit adder. The Generate and Propagate signals eliminate the rippling effect of the carry bits seen in the Carry Ripple Adder, allowing this design to be faster at the cost of extra logic.

ii) ***Describe how the P and G logic are used***

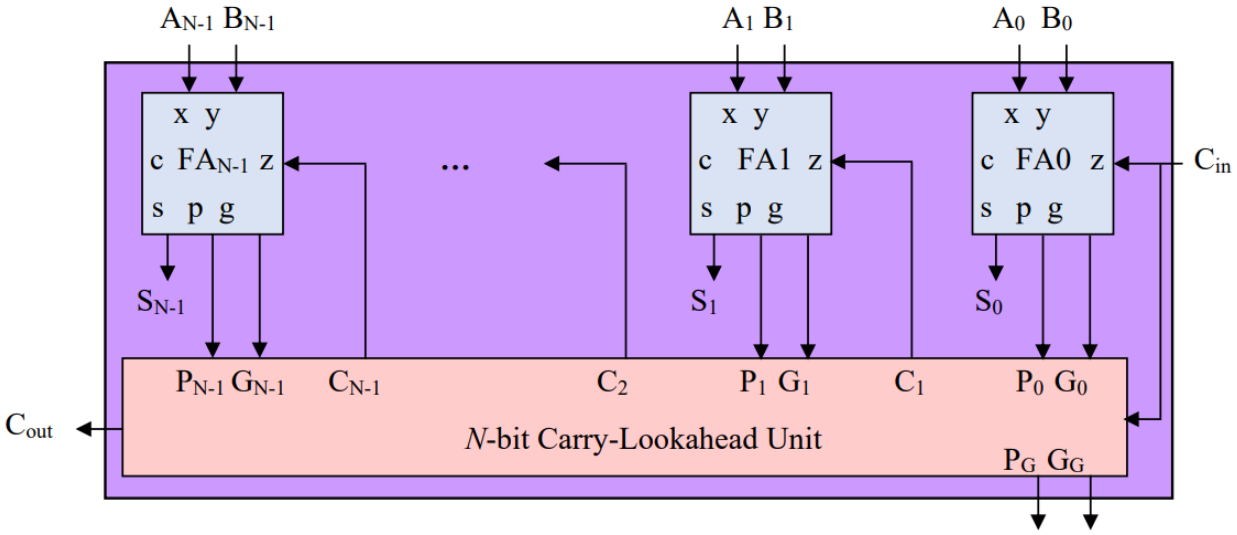
The Generate signal (G) is a logical AND of the A and B (A&B) signals to each single bit adder. If both the A and B signals to a single bit adder are a logical 1, there will need to be a carry to the next adder regardless of the carry in value. Similarly, the Propagate signal (P) is a logical OR of the A and B followed by a logical AND with the Cin signal (Cin(A+B)). If either the A or B signal is a logical 1 and the Cin signal is also a logical 1, there will be a carry to the next single bit adder. All of the P and G signals are output from each of the single bit adders and sent to a Carry Lookahead Unit (CLU). The CLU determines the carry in value to each of the four registers in the group, and also outputs two signals of its own: Propagate Group (PG) and Generate Group (GG). Within the CLU, the Cin value of each single bit adder is determined by if the previous adder generated a carry bit, OR if the previous adder propagated a carry bit. This logic is extended for each of the four adders in the group.

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

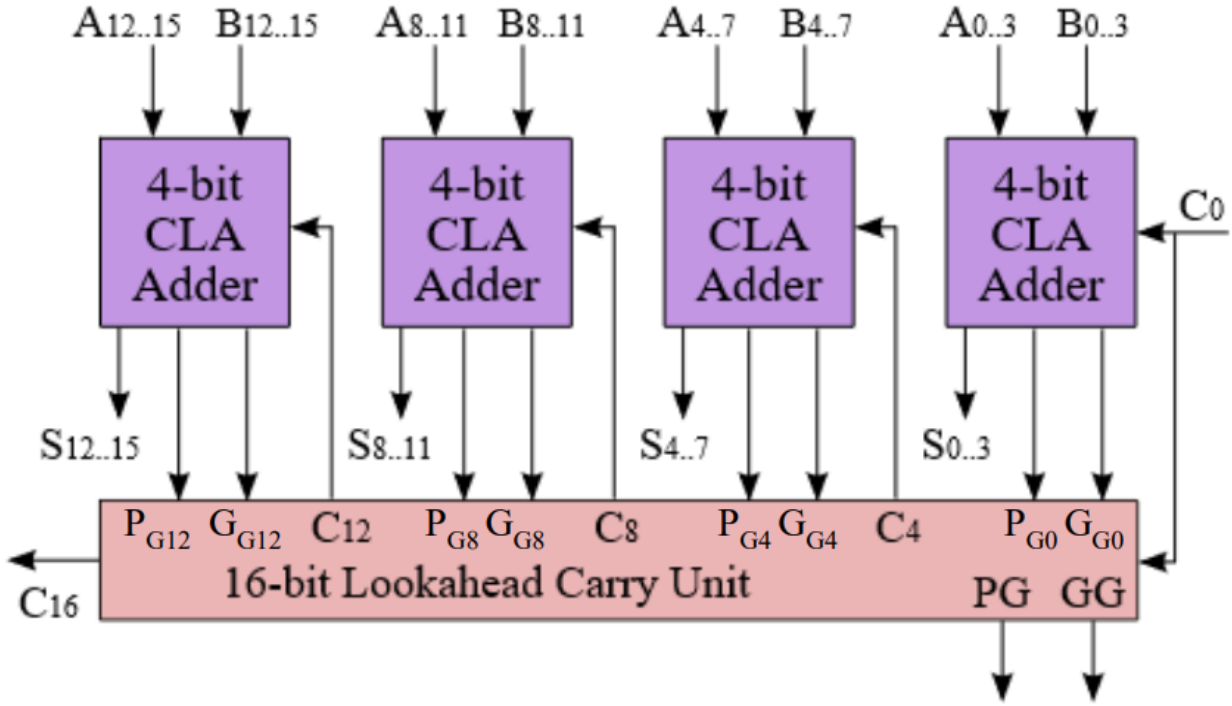
iii) ***Describe how you created the hierarchical 4x4 adder***

The hierarchical 4x4 bit adder is created by implementing four groups of four single bit adders. These four groups of single adders function in the same fashion as the four single bit adders within them. The carry bit to each group is determined by utilizing another CLU with the PG and GG signals from each group.

iv) ***Block Diagram: Inside a single CLA (4-bit internal adder)***



v) ***Block Diagram: How each CLA is chained together (4x4 bit external adder)***



Carry Select Adder

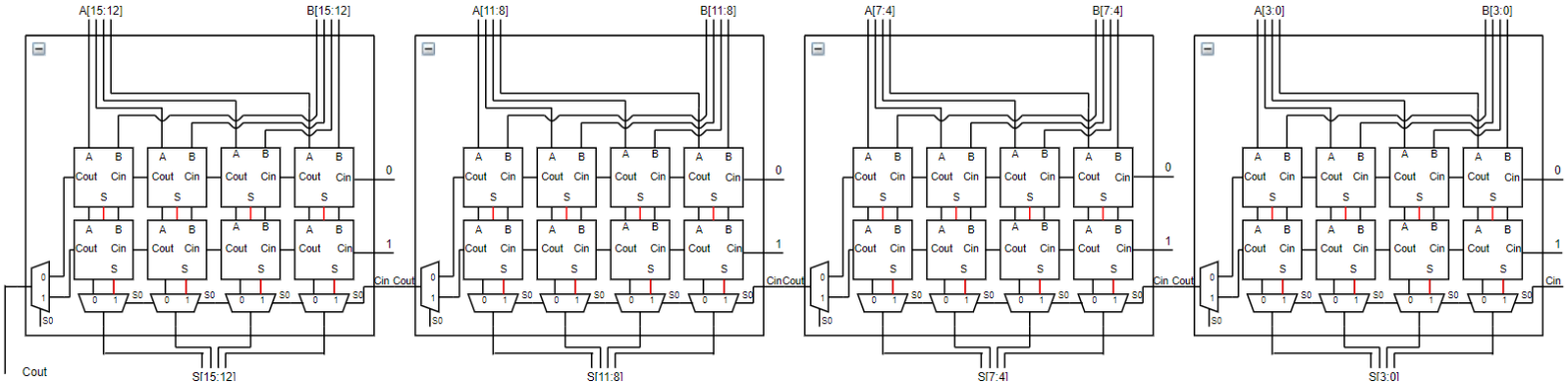
i) Written description of the architecture of the adder

The Carry Select Adder implements a different method of speeding up the computation without needing the complex logic of the Carry Lookahead Adder. This is possible at the cost of the extra area required. Once again it is implemented with a 4x4 Hierarchical design. The Carry Select Adder requires more area because each of the four groups contain two sets of four single bit adders. These two sets of single bit adders are used to make two separate calculations: one with a carry-in value of 0, and one with a carry-in value of 1.

ii) Describe at a high level how the CSA speculatively computes multiple sums in parallel and rapidly chooses the correct one later.

Each of the four groups that composes the 16-bit Carry Select Adder contains eight single bit adders and five 2:1 muxes. By using two sets of four single bit carry ripple adders in parallel, we make two calculations simultaneously: One calculation with a carry-in of 0, and the other with a carry-in of 1. The true carry-in is then used as the select bit for muxes to choose the correct output. While there is still propagation delay due to the carry-bits needing to ripple through the first group of single bit adders, this implementation is much faster than a typical Carry Ripple Adder because the other three groups are performing their calculation simultaneously. Once the four carry bits have rippled through the first group, the muxes are sequentially set to their correct values and the result is computed.

v) Block Diagram of the whole CSA circuit containing adders, multiplexers, and glue logic.



vi) *Describe how you created the hierarchical 4x4 adder*

Similarly to the Carry Ripple and Carry Lookahead implementations, the hierarchical 4x4 bit adder for the Carry Select Adder is created by implementing four groups of four single bit adders. In this case however, there are two sets of four single bit adders within each group. Each of these sets of four single bit adders function in the same manner as a Carry Ripple Adder..

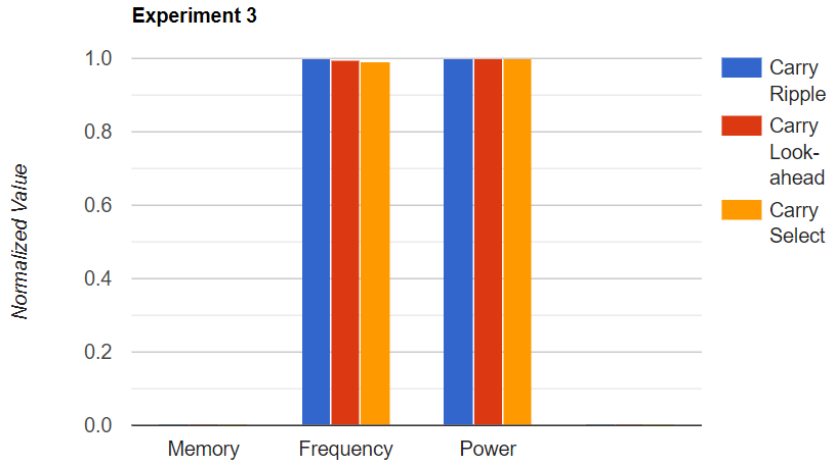
Written description of all .SV modules				
Module	**Inputs**	**Outputs**	**Description**	**Purpose**
adder2.sv	Clk, Reset_Clear, Run_accumulate SW [9:0]	[9:0] LED [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX 5	Top-level entity of the experiment, which allows us to select which adder to use and extend the input value to 16 bits before calculation.	This module is used to instantiate our adders, extend our input value to 16 bits and send the results to the hex drivers.
control.sv	Clk, Reset_Run	Run_0	This module implements a three state finite state machine to control the adders.	Controls the next state logic and signals the adders on when to begin executing a computation
HexDriver.sv	[3:0] In0	[6:0] Out0	This is used to control the HEX display. Each Input has a hard coded output to display a number on the HEX Display	Translates binary values to signals needed to display those values on the HEX displays of the DE-10
reg_17.sv	Clk, Reset, Load, [16:0]D	[16:0] DataOut	17 bit register. Automatically resets when Reset signal = 1. Enters load mode when Load = 1	Used to hold the value of the one operand followed by the result of the sum
router.sv	R, [15:0] A_in, B_in	[16:0] Q_out	17 bit parallel multiplexer used to route data into the register unit.	This module is used to place the sum of the two values into the register unit.
ripple_adder.sv	[15:0] A, B cin	[15:0] S cout	Implements a 4x4 bit hierarchical design for traditional ripple adder.t	Performs a 16-bit addition of two binary values.
four_bit_adder	[3:0] a, b cin	[3:0] S cout	Four bit adder used to implement the Ripple Carry Design. Calculates the S and Cout for a series of 4 bits.	Four of these modules are used to implement the 4x4 hierarchical design of the RIpple Carry Adder.
single_bit_adder	A, b, cin	S, cout	Single bit adder used to implement the four bit adders in the Ripple Carry Design. Outputs the Sum and Carry out of a one bit addition.	Four of these modules are used to implement a four bit ripple carry adder.
lookahead_adder.sv	[15:0] A, B cin	[15:0] S cout	Implements a 4x4 bit hierarchical design for a Carry Lookahead adder using single bit adders. Utilizes a CarryLookaheadUnit in order to output a GG and PG signals to determine the carry-in value for the following groups of adders	Performs 16-bit addition of two binary values. Implements P, G, PP and GG signals along with a Carry Lookahead unit to make calculation more efficient than a traditional Ripple Carry adder
four_bit_CLA_adder	[3:0] a, b cin	[3:0] S cout	Four bit adder used to implement the Carry Lookahead design. Calculate sum and outputs PP and GG signals for the External CLU.	Four of these modules are used to implement the 4x4 hierarchical design of the Carry Lookahead Adder
single_bit_adder_LA	A, b, cin	S, cout	Single bit adder used to implements the four bit adders in the Carry Lookahead Design. Calculates the sum and outputs P and G signals for the internal CLU of each group.	Four of these modules are used to implement a four bit CLA adder.
select_adder.sv	[15:0] A, B cin	[15:0] S cout	Implements a 4x4 hierarchical design for a Carry Select Adder using single bit adders. Utilizes two sets of four Ripple Carry adders and five muxes in each group.	Performs a 16-bit addition of two binary values. Simultaneously makes two separate calculations and utilizes the carry-in bit to select the correct result.
four_bit_adder_SA	[3:0] a, b cin	[3:0] S cout	Four bit adder used to implement the Select Adder Design. Performs two calculations the S and Cout for a series of 4 bits and chooses which result to output based on the carry-in	Four of these modules are used to implement the 4x4 hierarchical design of the Carry Select Adder.
single_bit_adder_SA	A, b, cin	S, cout	This is a single internal 1 bit full adder. It is used to calculate the S and Cout for a single bit	The module is used in series in the 4x4 bit external 4 bit adder

Area, Complexity, and Performance Tradeoffs Between Adders Types			
	Ripple Carry Adder	**Carry Lookahead Adder**	**Carry Select Adder**
Area	The Ripple Carry adder utilizes the fewest adders and least logic, meaning it also has the smallest area of the three designs.	The extra logic required by the P and G signals of the Carry Lookahead Adder cause the design to use more area than the Ripple Carry adder but less area then the Carry Select adder.	The Carry Select Adder requires the most area since the design requires two full sets of 16 single bit adders along with muxes in order to select the correct result.
Complexity	The Ripple Carry adder is also the least complex. Each single bit adder calculates its own sum and carry bit using a total of four gates.	The Carry Lookahead is the most complex of the three adder designs due to the significant amount of logic needed to implement the P, G, PG and GG signals between the individual adders and groups.	The Carry Select Adder is more complex than the Ripple Carry adder simpler than the Carry Lookahead. The logic behind the Carry Select adder is intuitive to understand which makes the design straightforward to implement.
Performance Pros	Simplest adder to implement and occupies the least amount of area.	The Carry Ripple adder is the fastest of the three adders. The P, G, PG and GG signals provide the fastest method for the carry bits to be determined, resulting in the fastest computation time.	The Carry Select adder is faster than the Ripple Carry adder since the design only needs to wait for four carry bits to ripple though instead of 16. However the Carry Select adder is still slower than the Carry Lookahead adder.
Performance Cons	Slow due to the carry bits needing to ripple through all 16 individual adders before the result is computed.	Complicated logic makes this adder somewhat difficult to understand. Requires the most number of individual gates to implement.	Requires the most amount of area to implement while being slower than the Carry Lookahead adder.

Normalized Performance of Each Adder Type

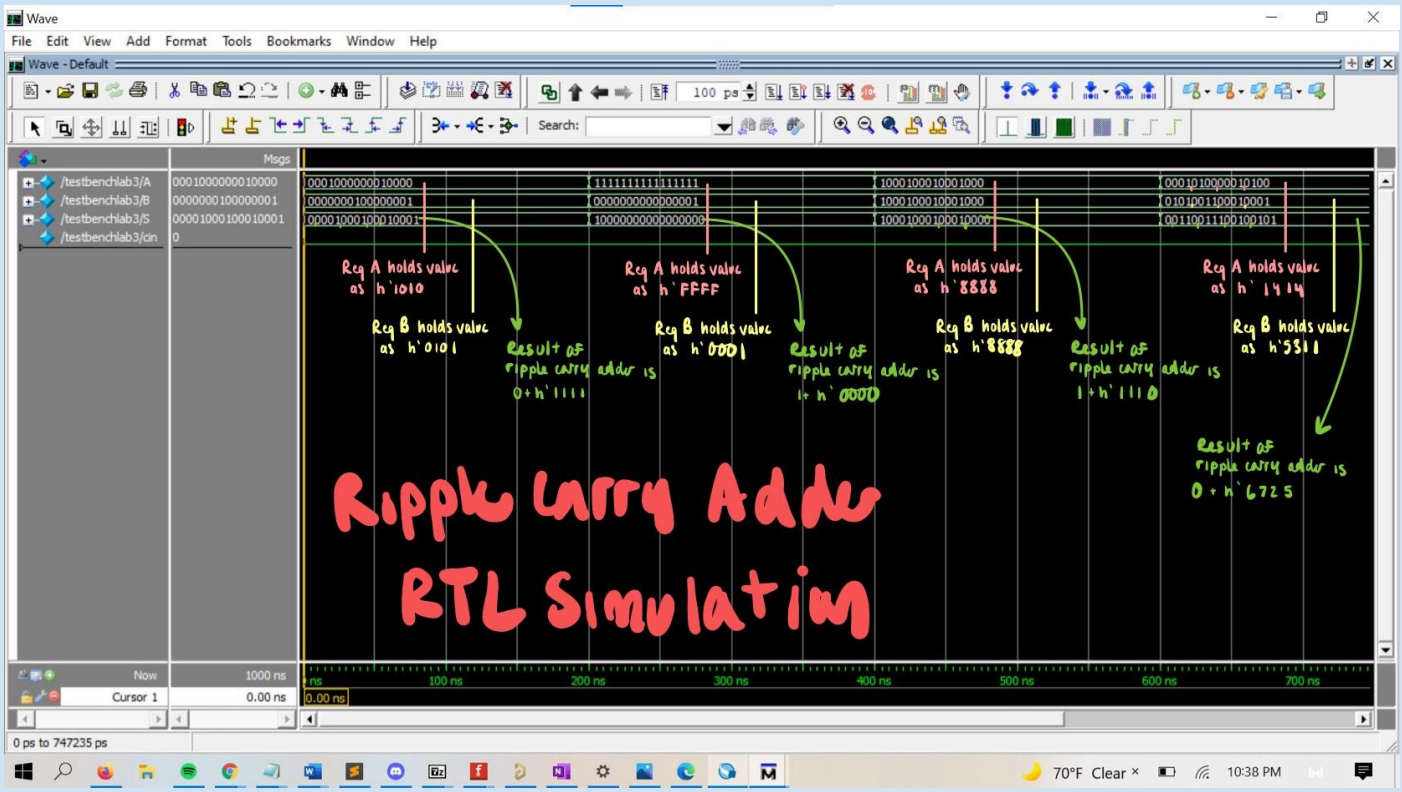
	Ripple Carry	**Carry Lookahead**	**Carry Select**
Memory (BRAM)*	N/A	N/A	N/A
Frequency	1	0.994	0.991
Total Power	1	1	1

*All memory values were 0 due to only using combinational logic. Cannot normalize.

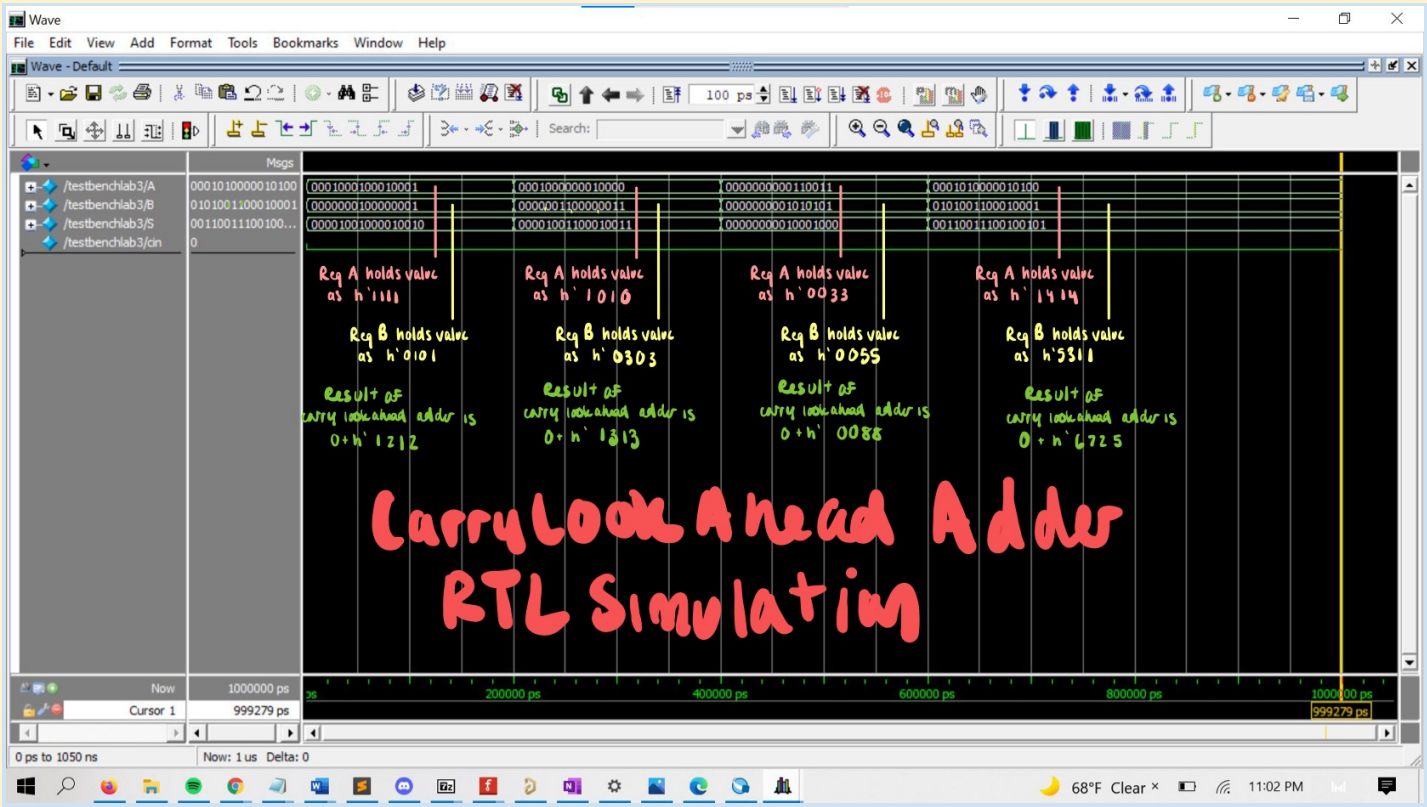


Annotated Signal Traces

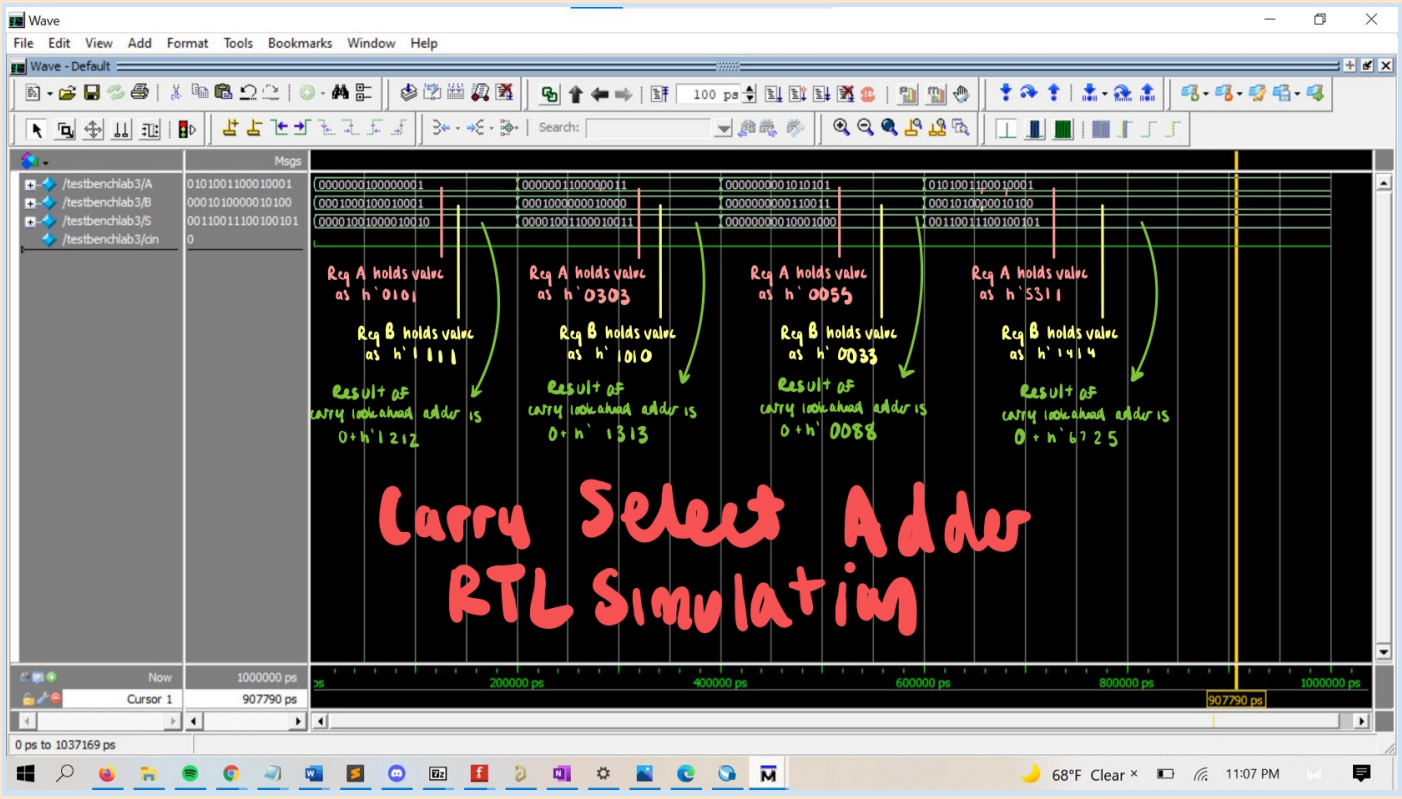
RTL Simulation Trace: Ripple Carry Adder



RTL Simulation Trace: Carry Lookahead Adder

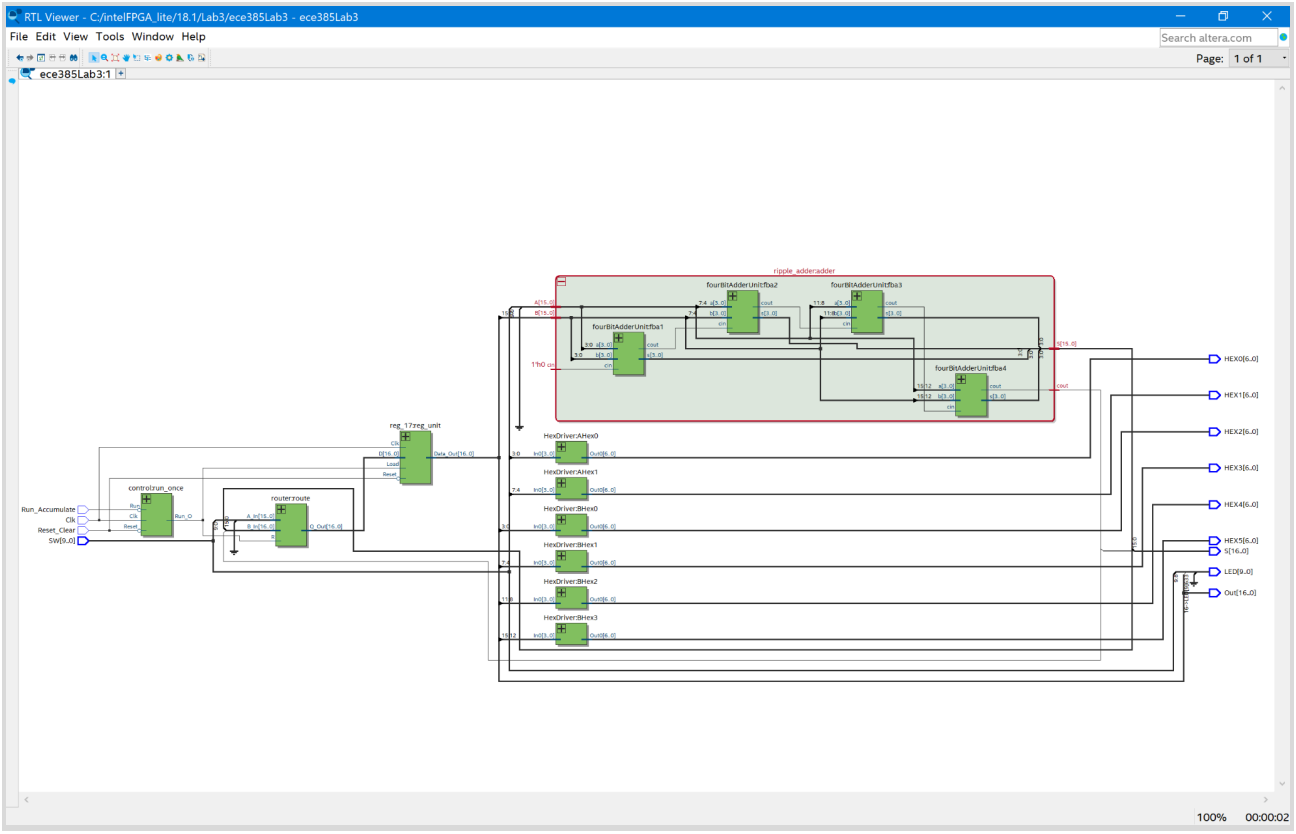


RTL Simulation Trace: Carry Select Adder



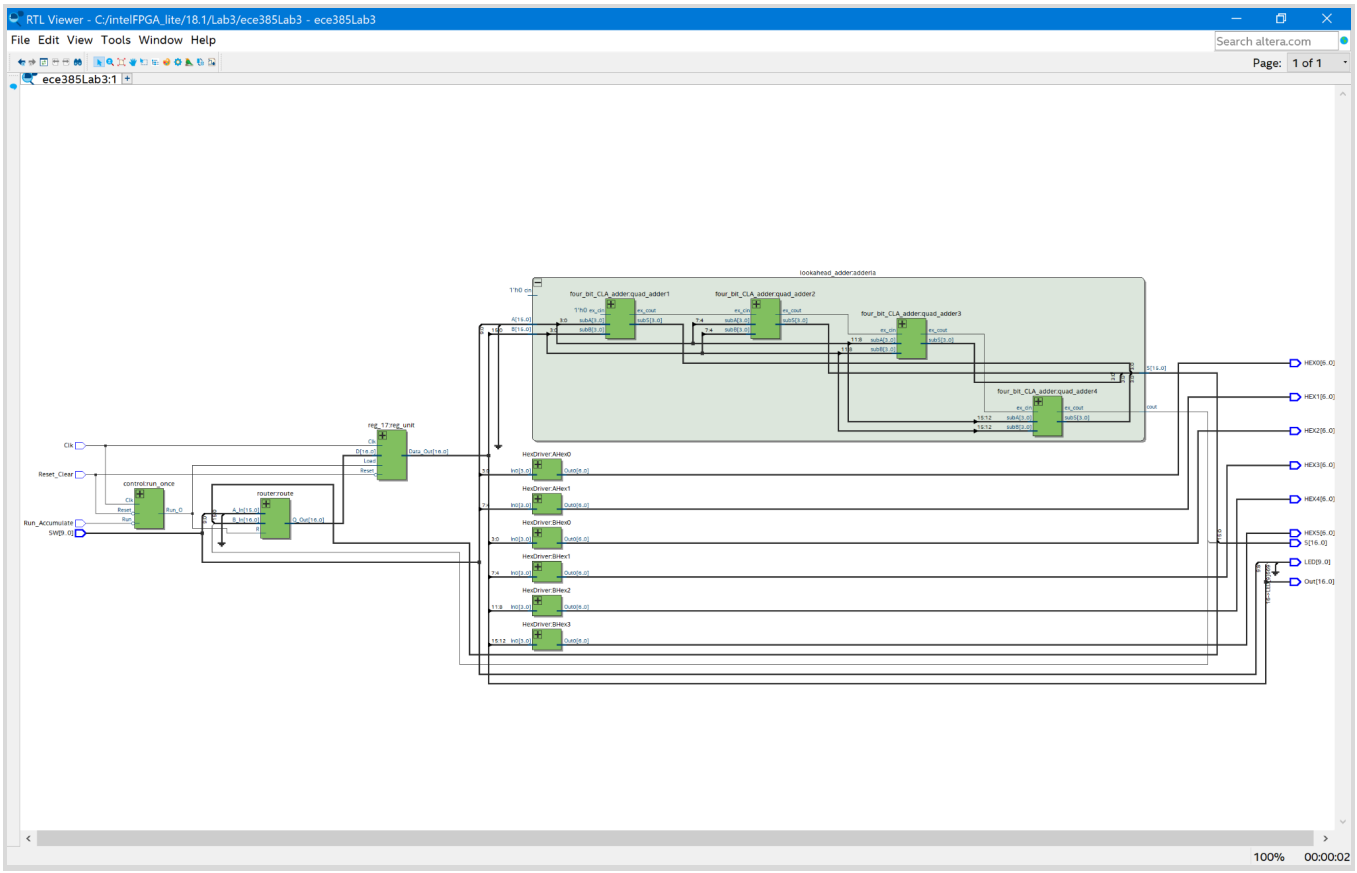
RTL Viewer and Modules of Each Design

i) Ripple Carry




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


     logic ex_cin0, ex_cin1, ex_cin2;
     four_bit_adder quad_adder1(.subA(A[3:0]),   .subB(B[3:0]),  . ex_cin(0),       .subS(S[3:0]),   .ex_cout(ex_cin0));
     four_bit_adder quad_adder2(.subA(A[7:4]),   .subB(B[7:4]),   .ex_cin(ex_cin0), .subS(S[7:4]),   .ex_cout(ex_cin1));
     four_bit_adder quad_adder3(.subA(A[11:8]),  .subB(B[11:8]),  .ex_cin(ex_cin1), .subS(S[11:8]),  .ex_cout(ex_cin2));
     four_bit_adder quad_adder4(.subA(A[15:12]), .subB(B[15:12]), .ex_cin(ex_cin2), .subS(S[15:12]), .ex_cout(cout));
endmodule

   module four_bit_adder (
      input [3:0] subA,subB,
      input ex_cin,
      output [3:0] subS,
      output ex_cout
      );

      //Declaring first level adders
      logic c0, c1, c2;
      single_bit_adder bit_adder1(.x(subA[0]), .y(subB[0]), .cin(ex_cin), .s(subS[0]), .cout(c0));
      single_bit_adder bit_adder2(.x(subA[1]), .y(subB[1]), .cin(c0),     .s(subS[1]), .cout(c1));
      single_bit_adder bit_adder3(.x(subA[2]), .y(subB[2]), .cin(c1),     .s(subS[2]), .cout(c2));
      single_bit_adder bit_adder4(.x(subA[3]), .y(subB[3]), .cin(c2),     .s(subS[3]), .cout(ex_cout));
   endmodule

   module single_bit_adder (
      input x, y, cin,
      output s, cout
      );

   //logic to generate s and cout
      assign s = x ^ y ^ cin ;
      assign cout = (x&y) | (x&cin) | (y&cin) ;
   endmodule
```

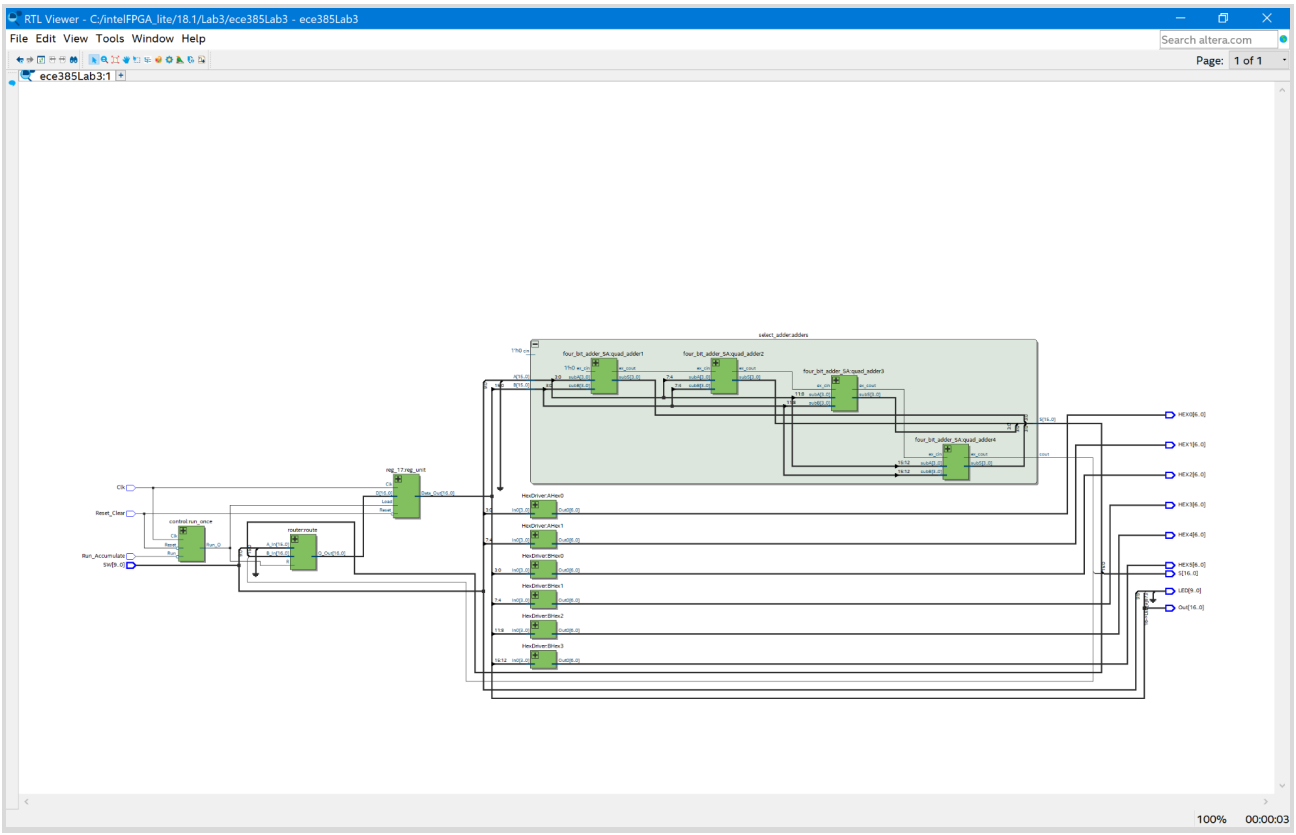
ii) Carry Lookahead




```
module lookahead_adder (
   input  [15:0] A, B,
   input         cin,
   output [15:0] S,
   output        cout );
     //Declaring second level adders
     logic ex_cin0, ex_cin1, ex_cin2;
     four_bit_CLA_adder quad_adder1(.subA(A[3:0]),   .subB(B[3:0]),  . ex_cin(0),  .subS(S[3:0]),   .ex_cout(c0));
     four_bit_CLA_adder quad_adder2(.subA(A[7:4]),   .subB(B[7:4]),   .ex_cin(c0), .subS(S[7:4]),   .ex_cout(c1));
     four_bit_CLA_adder quad_adder3(.subA(A[11:8]),  .subB(B[11:8]),  .ex_cin(c1), .subS(S[11:8]),  .ex_cout(c2));
     four_bit_CLA_adder quad_adder4(.subA(A[15:12]), .subB(B[15:12]), .ex_cin(c2), .subS(S[15:12]), .ex_cout(cout));
endmodule

   module four_bit_CLA_adder (
      input [3:0] subA,subB,
      input ex_cin,
      output [3:0] subS,
      output ex_cout );
      logic p0, p1, p2, p3, g0, g1, g2, g3, c1, c2, c3;

      assign p0 = subA[0] ^ subB [0];
      assign p1 = subA[1] ^ subB [1];
      assign p2 = subA[2] ^ subB [2];
      assign p3 = subA[3] ^ subB [3];
      assign g0 = subA[0] & subB [0];
      assign g1 = subA[1] & subB [1];
      assign g2 = subA[2] & subB [2];
      assign g3 = subA[3] & subB [3];

      assign c1 = g0 + (p0 & ex_cin);
      assign c2 = g1 + (g0 & p1) + (ex_cin & p0 & p1);
      assign c3 = g2 + (g1 & p2) + (g0 & p1 & p2) + (ex_cin & p0 & p1 & p2);
      assign ex_cout = g3 + (g2 & p3) + (g1 & p2 & p3) + (g0 & p1 & p2 & p3) + (ex_cin & p0 & p1 & p2 & p3);

      single_bit_adder_LA bit_adder1(.x(subA[0]), .y(subB[0]), .cin(ex_cin), .s(subS[0]));
      single_bit_adder_LA bit_adder2(.x(subA[1]), .y(subB[1]), .cin(c1),     .s(subS[1]));
      single_bit_adder_LA bit_adder3(.x(subA[2]), .y(subB[2]), .cin(c2),     .s(subS[2]));
      single_bit_adder_LA bit_adder4(.x(subA[3]), .y(subB[3]), .cin(c3),     .s(subS[3]));
   endmodule
```

```
   module single_bit_adder_LA (
      input x, y, cin,
      output s );
      //logic to generate s
      assign s = x ^ y ^ cin ;
   endmodule
```

iii) Carry Select




```
module select_adder (
   input  [15:0] A, B,
   input         cin,
   output [15:0] S,
   output        cout );
     //Declaring second level adders
     logic add0_ex_cin0, add0_ex_cin1, add0_ex_cin2, add0_out;
     four_bit_adder_SA quad_adder1(.subA(A[3:0]),   .subB(B[3:0]),   .ex_cin(0),            .subS(S[3:0]),   .ex_cout(add0_ex_cin0));
     four_bit_adder_SA quad_adder2(.subA(A[7:4]),   .subB(B[7:4]),   .ex_cin(add0_ex_cin0), .subS(S[7:4]),   .ex_cout(add0_ex_cin1));
     four_bit_adder_SA quad_adder3(.subA(A[11:8]),  .subB(B[11:8]),  .ex_cin(add0_ex_cin1), .subS(S[11:8]),  .ex_cout(add0_ex_cin2));
     four_bit_adder_SA quad_adder4(.subA(A[15:12]), .subB(B[15:12]), .ex_cin(add0_ex_cin2), .subS(S[15:12]), .ex_cout(cout));

endmodule

   module four_bit_adder_SA (
      input [3:0] subA,subB,
      input ex_cin,
      output [3:0] subS,
      output ex_cout);
      //Declaring first level adders
      logic add0_c0, add0_c1, add0_c2, add0_cout;
      logic sum0_0,sum0_1, sum0_2, sum0_3;
      single_bit_adder_SA bit_adder0_1(.x(subA[0]), .y(subB[0]), .cin(ex_cin),  .s(sum0_0), .cout(add0_c0));
      single_bit_adder_SA bit_adder0_2(.x(subA[1]), .y(subB[1]), .cin(add0_c0), .s(sum0_1), .cout(add0_c1));
      single_bit_adder_SA bit_adder0_3(.x(subA[2]), .y(subB[2]), .cin(add0_c1), .s(sum0_2), .cout(add0_c2));
      single_bit_adder_SA bit_adder0_4(.x(subA[3]), .y(subB[3]), .cin(add0_c2), .s(sum0_3), .cout(add0_cout));

      logic add1_c0, add1_c1, add1_c2, add1_out;
      logic sum1_0,sum1_1, sum1_2, sum1_3;
      single_bit_adder_SA bit_adder1_1(.x(subA[0]), .y(subB[0]), .cin(ex_cin),  .s(sum1_0), .cout(add1_c0));
      single_bit_adder_SA bit_adder1_2(.x(subA[1]), .y(subB[1]), .cin(add1_c0), .s(sum1_1), .cout(add1_c1));
      single_bit_adder_SA bit_adder1_3(.x(subA[2]), .y(subB[2]), .cin(add1_c1), .s(sum1_2), .cout(add1_c2));
      single_bit_adder_SA bit_adder1_4(.x(subA[3]), .y(subB[3]), .cin(add1_c2), .s(sum1_3), .cout(add1_cout));

      assign subS[0] = (sum0_0 & ~(ex_cin)) + (sum1_0 & ex_cin);
      assign subS[1] = (sum0_1 & ~(ex_cin)) + (sum1_1 & ex_cin);
      assign subS[2] = (sum0_2 & ~(ex_cin)) + (sum1_2 & ex_cin);
      assign subS[3] = (sum0_3 & ~(ex_cin)) + (sum1_3 & ex_cin);
      assign ex_cout = (add0_cout & ~(ex_cin)) + (add1_cout & ex_cin);
   endmodule
```

```
   module single_bit_adder_SA (
      input x, y, cin,
      output s, cout
      );

   //logic to generate s and cout
      assign s = x ^ y ^ cin ;
      assign cout = (x&y) | (x&cin) | (y&cin) ;
   endmodule
```

3) Post Lab Questions

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

In order to determine the ideal hierarchy for the16-bit Carry Select design we would have to perform analysis and testing of different adder configurations. When we were implementing our 4x4 design for this experiment, we discussed how different configurations would affect the performance of our machine. A couple designs that crossed our minds were a 8x2 configuration and a 2x8 configuration. We believe that each of these designs would have different advantages. For example, the 8x2 configuration would be faster than a 4x4 design, but would require more processing power since there are more calculations happening simultaneously. Similarly, a 2x8 design would be slower to calculate the result since there are less simultaneous calculations, but the design would require less processing power. Ultimately we do believe that the 4x4 implementation is the best balance of speed and efficiency. To prove this to be true, we would need to design each of these adders within Quartus to compare the amount of area used by designs, as well as their respective frequencies and power usage.

b) *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. 3.11 LUT DSP Memory (BRAM) Flip-Flop Frequency Static Power Dynamic Power Total Power.*

Design Statistics Table for Each Adder			
	Ripple Carry	**Carry Lookahead**	**Carry Select**
LUT (look up tables)	78	91	78
DSP (Digital signal processors	0	0	0
Memory(BRAM)	0	0	0
Flip Flop	20	20	20
Frequency	67.73MHz	67.37MHz	67.15MHz
Static Power	63.47mW	63.47mW	63.47mW
Dynamic Power	1.21mW	1.11mW	1.13mW
Total Power	81.43mW	81.18mW	81.35mW

This resource breakdown data from Quartus does align with our theoretical design expectations. Because the power usage by each of our adders is so low, it is difficult to get an accurate reading. This

is why the static power of each of our adders is the same value. The maximum operating frequency of our Carry Lookahead adder is higher than the maximum frequency of the Carry Select adder as we expected it to be. As for the power consumption, although the total power values are very similar, we were surprised to see that the Ripple Carry adder has the highest power consumption. We believed that the Ripple Carry adder would have the lowest power consumption because it is the simplest design.

4) Conclusion

a) Summary

Lab 3 consisted of 3 parts, each of which implemented a 16-bit adder using a 4x4 hierarchical design. These adders include a Ripple Carry Adder, a Carry Lookahead Adder and a Carry Select Adder. By implementing three different designs that each have the same functionality, we were able to develop a deeper understanding of how each design functions. Additionally, we were able to compare the performance of each design with respect to one another. When designing and implementing our adders, we made sure to take the time to fully understand each adder and how it functioned. After we felt that we had a good understanding, we moved on creating each module within Quartus. This strategy made it easier for us to write the code needed to implement our design, and also enabled us to not have any bugs or issues within our design. After sorting out some compilation and syntax errors, we were able to upload each of our designs to the FPGA where they worked immediately.

5) Feedback

a) Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it does not get changed.

When first implementing the Carry Lookahead adder, our understanding owas to use the Propogate and Generate logic on the internal 4-bit adders. It is the same logic that we would use for the external 4x4 bit adder, but it still worked out in the end. In a sense, it is the same logic that we would have used for the external 4x4 bit adders, but it was instead used on the internal 4 bit adders. One suggestion for future semesters is to include a better diagram of when each calculation should be made. Take the CarryLookahead adder for example, we were unsure when the G groups and P groups should be implemented. Our understanding of the logic was sufficient, but we were left confused about knowing where to implement the P and G group calculations (in the internal 4 bit adders or the external 4x4 bit adders).