ECE 385

Spring 2023

Experiment #3

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

Dylan Bautista, Matthew Wei

Lab Section NL/ Jan 27 11:00 AM

Nick Lu

Experiment #3

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

I. Introduction/Purpose of Circuit:

Each of the three adders created during this lab operate on the high-level the same. Every adder takes in two sets of 16 bits of input and adds those two 16-bit numbers together to output a 16-bit sum with carry-out. In physical demonstration with FPGA board, 10 bits can be loaded as one of the addends, with the other 6 being hard coded in software. Each press of Run_Accumulate adds the previous sum result in the updated Out register and the Switch register values, displaying the result in hexadecimal LED's. The Reset_Clear button should clear all registers and display accordingly on LED's.

II. Written description of Adders.

Ripple Carry Adder:

The Ripple Carry Adder is created through multiple instances of a full adder, one module for each bit pair being added. Each of these full adders takes in two input bits and a carry-in and uses combinational logic to output a carry-out and Sum. In our case, there are 16 of these modules connected in sequence, with the Cout of each one connected to the Cin of the following, and the dual 16 bits of input making up A and B values. The total sum of A and B are formed from the 16 output bits, with the final Cout of the last adder representing the Cout.

Carry Lookahead Adder

The 16-bit Carry Lookahead Adder is created from four, 4-bit Carry Lookahead Adder modules. Each module takes in 4 bits of the addends, A and B, and a Cin. There are 4 full adders for each pair which calculate the four sums, however instead of a Cout, each full adder generates a propagate ($A \oplus B$) and generate ($A \cdot B$) signal. The Cin for the last three adders is generated by an expanded form of $Ci = Gi + (Pi \cdot Ci)$. This way, all these Cins can be loaded in parallel after gate delays instead of having to iterate through each adder. Each 4-bit module generates a group propagate and generate signal from combinational gate logic and a Cout from the final full adder's Cout. Next, the full 16-bit hierarchal design is formed by generating the Cin for the last three 4-bit modules much like how we generated the carry bits within the 4-bit CLA, using the group propagates, group generates and initial Cin. The final sum is generated from combining the returned 4-bit sums from all four modules, and the total Cout from the final module's Cout.

Carry Select Adder

The 16-bit Carry Select Adder is created using a hierarchal structure of three 4-bit carry select adders, along with an initial, singular 4-bit ripple adder that takes in the first four bits of A and B along with the Cin. Each of the three 4-bit select adders is made up of two 4-bit carry ripple adders which calculate the Cout and sum of those 4 bits of A and B with a hardcoded Cin of 0 and 1 respectively. The Cout of the previous 4-bit select adder is used as the select for a multiplexer to choose between the outputs of the two carry ripple adders. The output of this multiplexer is used as that portion of the Sum. For the last three carry select adders, combinational gate logic between the outputs of the two ripple adders and the previous Cout is used to generate the next Cout. The final sum is generated from combining the returned 4-bit sums from all four modules, and the total Cout from the final module's Cout.

III. Design Steps for each Adder:

Ripple Carry Adder

The Ripple Carry Adder was designed by first creating a single-bit full adder module with the combinational logic to produce Cout and S. Then, in the ripple adder file, we created 16 instances of the module, connecting the Cout of one with the Cin of the following using logic variables. Each sum of the modules was assigned to a single bit of the 16-bit total sum.

Carry Lookahead Adder

The Carry Lookahead Adder file has four instances of a created four-bit CLA module. The four-bit CLA module has logic variables for each bit pair, assigning one propagate (A XOR B) and one generate (A AND B). Then, logic variables were created for the three Carry-in values as well as the group propagate and generate, using the combinational logic of the P, G, and original Cin values. Four instances of a full adder were created, taking in the four input bits and the created Cin's, and assigning the Sum bits and Cout. Lastly, to create a hierarchal 4x4 adder, we instantiated this module four times, splitting the input bits into groups of four among the adders. In between these instances, logic variables for the three carry-ins were assigned using the group values of P, G, and Cout. The total Sum was assigned through all four instances' sums and the Cout was the final instance's Cout.

Carry Select Adder

The Carry Select adder required that we use 4-bit ripple adders, so we created a module by instantiating a full adder 4 times and linking the Cins and Couts of adjacent adders. In the select adder module, we created the first instance of the 4-bit adder for the first 4 input bits and the original Cin. Then, we instantiated the 4-bit adder 2 more times for the following 4-bits, one with

Cin 1 and the other with 0. Then, an instance of a provided mux module was used to output one of the two sums, using the first adder's Cout as select. The output of the mux was assigned to the total Sum, while the current two Cout along with the previous Cout was used to assign a logic variable for the following mux select. In this, way, we linked all three 4-bit CSA parts, until we have assigned all 16 bits of the total Sum and the final Cout is generated. Because all 6 of the ripple adders can generate their sums in parallel, once the initial Cout is generated, only gate and mux delays factor into the timing since all available sum options for each 4-bit set are already available. Therefore, the efficiency effectively increases from iterating on all 16 bits to 4 bits plus logic delays.

<u>IV. Block Diagrams:</u>

Ripple Carry Adder, 16-Bit

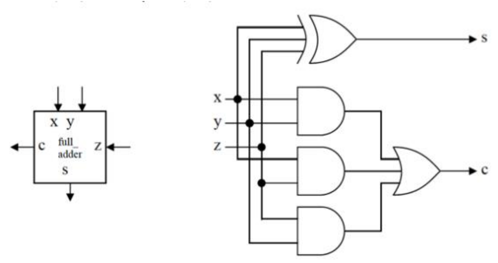
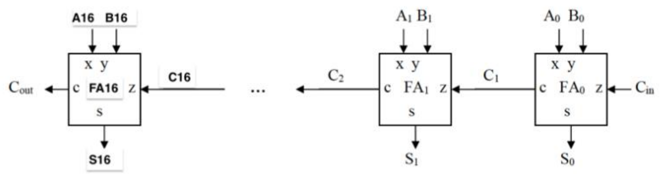
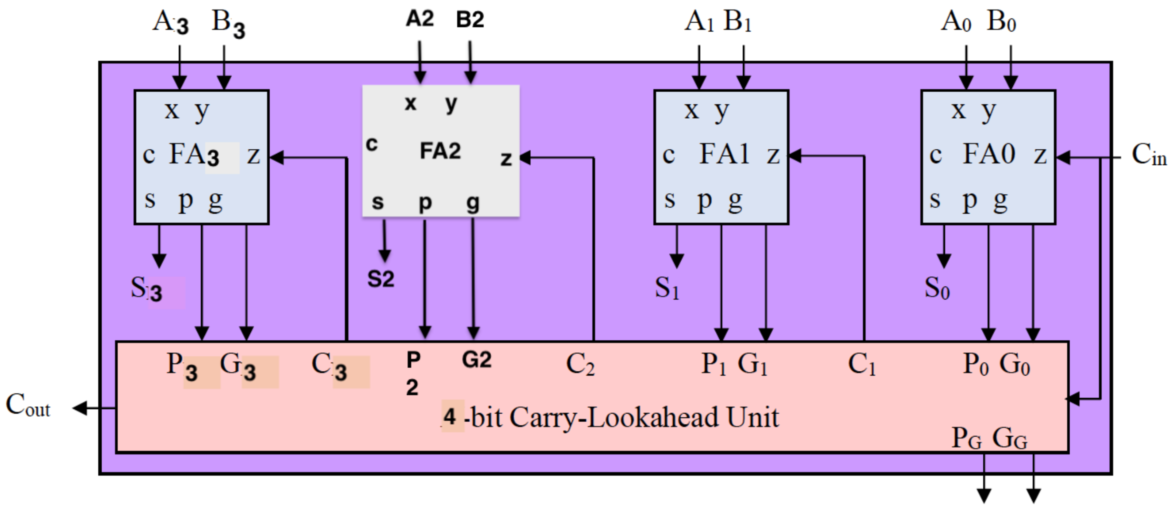
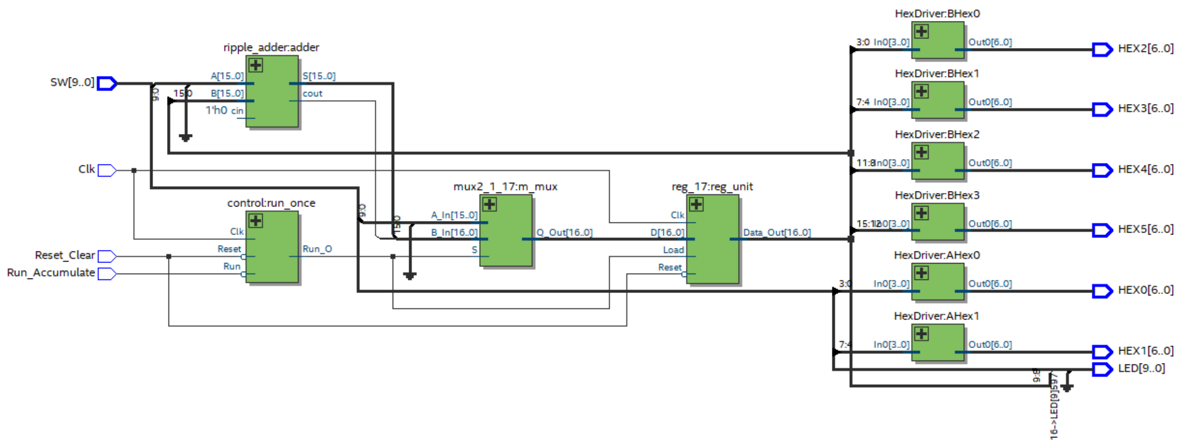


Figure 2: Full-Adder Block Diagram

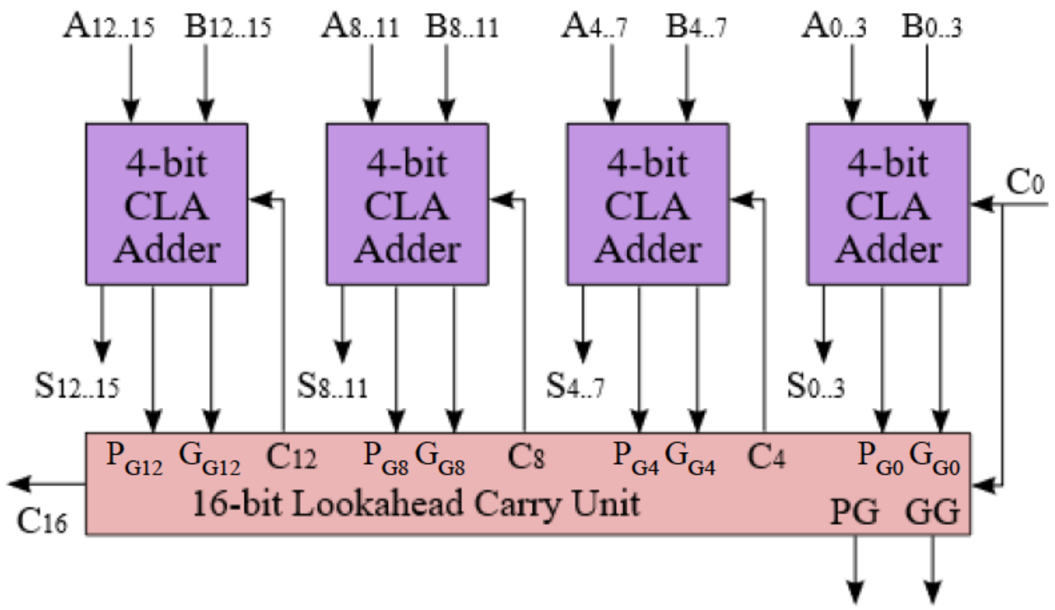


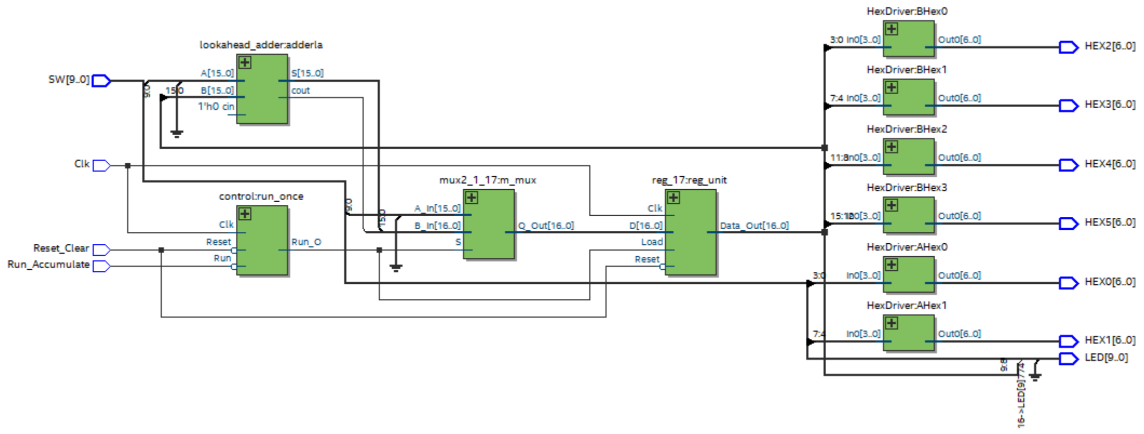
Carry Lookahead Adder, 4-Bit



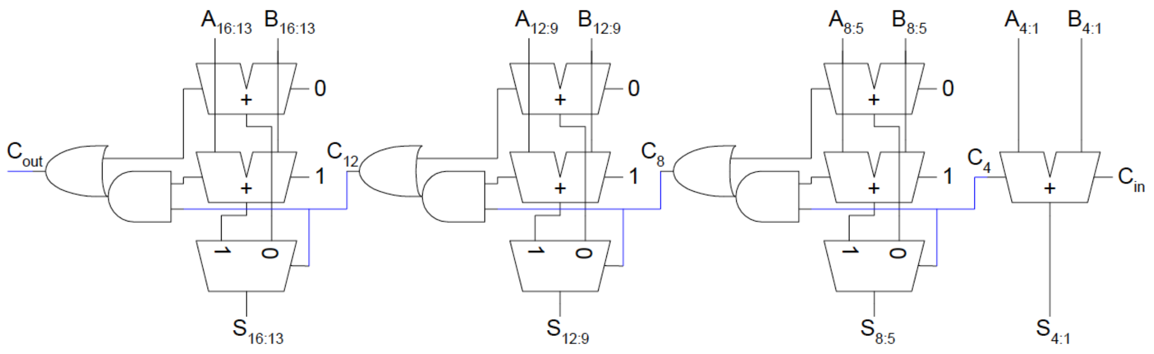


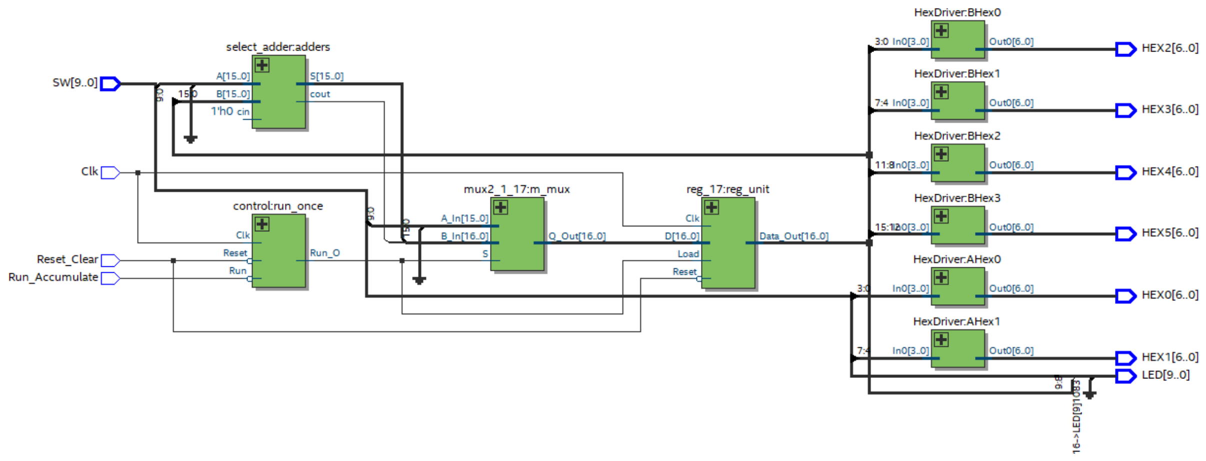
Carry Lookahead Adder, 16-Bit


A12..15
B12..15
A8..11
B8..11
A4..7
B4..7
A0..3
B0..3
4-bit CLA Adder
4-bit CLA Adder
4-bit CLA Adder
4-bit CLA Adder
C0
S12..15
S8..11
S4..7
S0..3
PG12 GG12 C12 PG8 GG8 C8 PG4 GG4 C4 PG0 GG0
16-bit Lookahead Carry Unit
C16
PG GG


lookahead_adder:adderla
A[15..0]
B[15..0]
1'h0 cin
S[15..0]
cout
SW[9..0]
Clk
Reset_Clear
Run_Accumulate
control:run_once
Clk
Reset
Run
Run_O
mux2_1_17:m_mux
A_In[15..0]
B_In[16..0]
S
Q_Out[16..0]
reg_17:reg_unit
Clk
D[16..0]
Load
Reset
Data_Out[16..0]
HexDriver:BHex0
HexDriver:BHex1
HexDriver:BHex2
HexDriver:BHex3
HexDriver:AHex0
HexDriver:AHex1
In0[3..0]
Out0[6..0]
HEX2[6..0]
HEX3[6..0]
HEX4[6..0]
HEX5[6..0]
HEX0[6..0]
HEX1[6..0]
LED[9..0]
16->LED[9]

Carry Select Adder, 16-Bit





V. Modules and Performance Analysis:

Modules

adder_toplevel.sv

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

Outputs: [9:0] LED, [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5

Description: The adder_toplevel module is used as the main unit where everything is put together. This is where it takes in inputs from the FPGA board and directs it to the control unit and whichever adder is currently being used.

Purpose: Combines all of the individual components together to make the whole design work.

control.sv

Inputs: Clk, Reset, Run

Outputs: Run_O

Description: The control unit tells the design how to respond after different input signals have been given to us. It uses state machines to determine which states to move onto after run or clear has been pressed.

Purpose: Controls how the design will respond to inputs.

full_adder.sv

Inputs: A, B, C_in

Outputs: S, C_out

Description: Computes S and C_out from A, B, and C_in with their given equations.

Purpose: Basic unit adder used for all the other more complex adders.

HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: Reads the input data and outputs the right hex value to be displayed on the LED display.

Purpose: Tells HexDriver what hex value to display on the LED.

four_bit_adder.sv

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

Outputs: [3:0] S, P, G, cout

Description: Has four full adders using the generated values of c as c_in. Values are assigned based on the propagate and generate equations for each individual bit.

Purpose: To be used for the lookahead adder as the units that compute the sum, propagate, and generate bits.

lookahead_adder.sv

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

Outputs: [15:0] S, cout

Description: Consists of four four-bit adder instances as well as logic expressions to calculate the carry value. Acts as the carry lookahead unit in the diagram.

Purpose: Adds 16-bits together and is our implementation of the 16-bit carry lookahead adder.

mux2_1_17.sv

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

Outputs: [16:0] Q_Out

Description: Selects an output based on the value of select S and whether it is high or low.

Purpose: Used as the select for the top-level design and as the select for the carry select adder.

reg_17.sv

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

Outputs: [16:0] Data_Out

Description: Either clears the register or loads a value into the register given by D and exported as Data_Out. Acts on the positive clock edge and checks the Reset signal.

Purpose: Stores the values of A and B for when the adder needs their values to calculate a sum.

ripple_adder.sv

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

Outputs: [15:0] S, cout

Description: 16 bit ripple adder made by combining 16 full adders in series with each other feeding one’s C_out bit the the next’s C_in bit.

Purpose: Adds 16-bits together and is our implementation of the 16-bit ripple carry adder.

four_bit_adder_select.sv

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

Outputs: [3:0] S, cout

Description: Basic 4-bit ripple adder created using 4 full adders and wiring them together.

Purpose: Used for the carry select adder so that 4 bits are grouped together at a time.

select_adder.sv

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

Outputs: [15:0] S, cout

Description: Four bit ripple adders used to compute each sum for whether Cout is high or low. Multiplexers are then used to select which sum to pick from. Module consists of ripple adders, multiplexers, and logic expressions for S and cout.

Purpose: Adds 16-bits together and is our implementation of the 16-bit carry select adder.

Area

The area (or number of gates for each adder) varies among all three of them. The ripple carry adder requires the least number of gates as its design only requires the gates needed for a regular full adder multiplied by however many bits are present in the design. The carry lookahead adder doesn't require the carry-out bit as the propagate and generate bits will determine it. This reduces the gates needed in a full adder. However, the propagation and generation bits add additional gates to it. Not only that, but the Lookahead carry unit also adds additional gates onto it. This makes the lookahead carry adder have a larger area than the ripple adder. Finally, the carry select adder has two adders for every bit besides the 0th bit. This means there are nearly twice as many gates in the carry select adder compared to the ripple carry adder. Not only that, but additional multiplexers are also needed to select the output of the summation.

In terms of size, carry lookahead has the highest area due to all the additional logic it requires, carry select has the second highest, and the ripple carry requires the least.

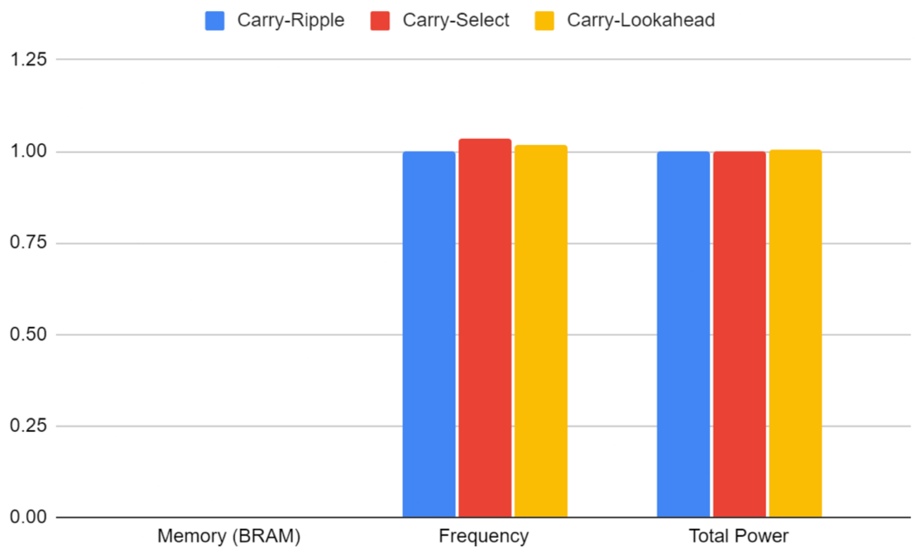
Complexity

The complexity (time complexity or Big-O) also is different for each adder. The ripple carry adder goes through each bit one by one, giving us O(n) time complexity. The carry lookahead gets rid of this need by generating propagate and generate values before-hand. This results in a O(logn) complexity. Finally, the carry select adder has a O(sqrt(n)) complexity as the gate delays depend on the multiplexers the total number of MUXs will be sqrt(n).

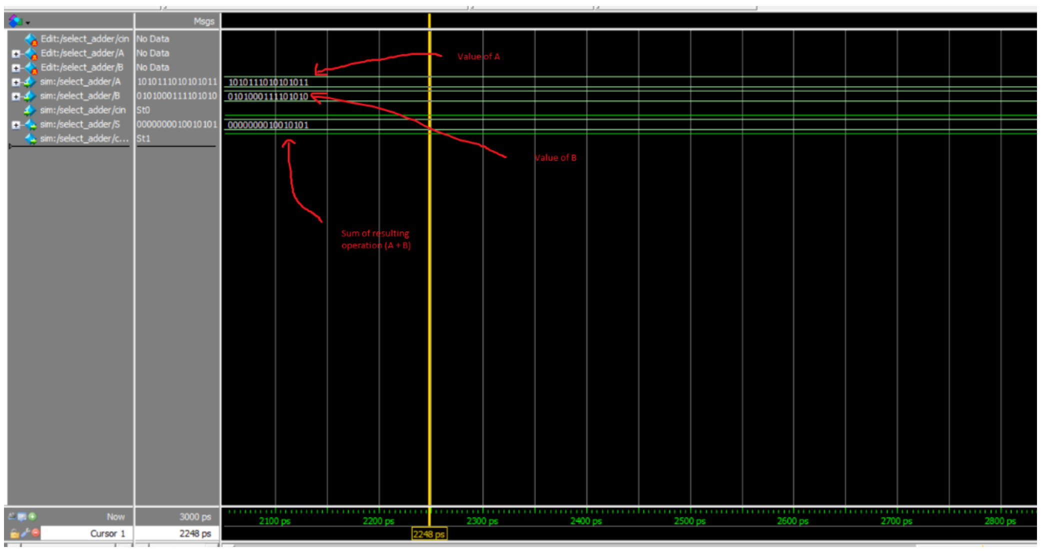
Performance

As shown from the area and the time complexities, all the final performance of each adder will be different. Although the simplest design, the ripple carry adder has the highest time complexity which makes sense since it goes through each adder one by one. At larger bit values, the time it takes will be more noticeable. The carry lookahead adder computes the sums in parallel without requiring the need to have each adder go through one another saving some time. This is why it results in it having the fastest time complexity. The carry select also computes it in parallel. However, the multiplexers that select the values at the end add an additional delay which makes it slower than the carry lookahead adder.

	Carry-Ripple	Carry-Select	Carry-Lookahead
Memory (BRAM)	0	0	0
Frequency	64.47	66.78	65.62
Total Power	105.09	105.24	105.52



Scaled chart of memory, frequency, and total power of the three adders



```
                                Msgs
Edit:/select_adder/cin     No Data
Edit:/select_adder/A       No Data
Edit:/select_adder/B       No Data
sim:/select_adder/A        1010111010101011     1010111010101011
sim:/select_adder/B        0101000111101010     0101000111101010
sim:/select_adder/cin      St0
sim:/select_adder/S        0000000010010101     0000000010010101
sim:/select_adder/c...     St1

Now                        3000 ps
Cursor 1                   2248 ps
```

Annotated simulation results of A + B in ModelSim

Timing Analysis

The carry ripple adder must go through 16 full adders before the result shows up. This is by far the slowest of the three. The carry lookahead adder must go through 4 full adders and some additional logic. It should be the fastest of all of the adders. The carry select adder must go through 1 full adder and 3 mux delays as well as some other additional logic. This makes it the second fastest.

VI. Bugs:

Throughout this lab, no major bugs were encountered. However, we did have one bug that stands out the most. When we were coding our carry lookahead adder, one of our propagate bits had the wrong Boolean expression assigned to it, giving us one wrong bit value in our final answer. We were able to pinpoint this by looking at which bit was incorrect in the final answer and checking the values of P and G. Once found, we simply corrected it and got the adder simulating properly.

VII. Conclusion:

Overall, this lab was a great introduction to creating our own modules and grouping them together. In the previous lab, all we had to do was change a couple values to make the processor 8 bits. For this lab, we had to write SystemVerilog code and create our own modules. This lab was simple enough that it wasn't a hard jump into writing code and was a great way of showing how to write modules or combine other modules together. This also accompanied by the clear documentation and function explanations of the different adders in lecture made this lab not that bad.

Post-Lab Questions

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?)

The hierarchical design is ideal depending on the situation. Using hierarchical design simplifies our design, reducing the input and outputs required and simplifying the adder. Specifically, if the MUX gate delay is equal to the full adder's delay, then the design would be optimized. Therefore, to design an ideal hierarchical design on the FPGA, we would need to do experiments to determine the gate delays of the adders and MUXs. From there, a more ideal grouping and design could be made.

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

	Carry-Ripple	Carry-Select	Carry-Lookahead
LUT	79	85	95
DSP	0	0	0
Memory (BRAM)	0	0	0
Flip-Flip	20	20	20
Frequency (MHz)	64.47	66.78	65.62
Static Power (mW)	89.97	89.97	89.97
Dynamic Power (mW)	1.40	1.42	1.78
Total Power (mW)	105.09	105.24	105.52

Overall, the data makes sense to our theoretical assumptions. The data for the carry select and carry lookahead when compared to the carry ripple look accurate. Power consumption is larger for the select and lookahead adders when compared to the ripple. The operating frequencies are also higher and LUTs are also higher. The flip-flip values are also the same which makes sense because they use the same registers. However, there seems to be some unexpected data between the carry-select and carry-lookahead. We figured that the carry-lookahead adder should be quicker and therefore resulting in a higher frequency than the carry-select. This could do with how we coded our SystemVerilog leading to differences between the two. The LUTs are higher for the carry lookahead which makes sense since it requires a lot more combinational logic. Power consumption would be highest for whichever had the highest LUTs. Looking at the data, it matches our assumptions as the lookahead adder has the highest power. The carry ripple adder consumes more power than we initially thought due to how many less gates are present in its design.