ECE 385

Fall 2022

Experiment #3

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

Ravi Thakkar, Siraj Khogeer

TA: Gavin Wu

9-21-2022

Introduction

In this lab, we are asked to create three different types of adders using SystemVerilog. The first adder that we created was a Ripple Carry Adder. This adder is a chain of multiple full-adders, which have inputs and outputs for carry-in and carry-out that are connected to adjacent full adders. This means that the carry-out of one full-adder is connected to the carry-in of another full adder. The Carry Lookahead Adder uses predetermined logic expressions for propagating and generating bits (P, G, respectively) to be able to avoid the slow rippling of the carry bits. Based on the carry-in input, the CLA predicts what the carry-out will be using P and G. The Carry Select Adder uses two adders for each bitwise addition and uses a multiplexor to select between a carry-in value of either zero or one, so that both outputs are pre-computed.

Ripple Carry Adder

i. Written description of the architecture of the adder

The ripple adder consists of 16 full adders that have their cout's connected to their cin's. This basically means that the carry in of each bit is rippled through the 16 full adders until it reaches the end. There are three inputs, A[15:0], B[15:0], and cin. There are two outputs S[15:0], and cout. The architecture is basically serially connecting 16 1-bit full adders to produce 1 16-bit adder. The full adder works by the formula S = A^B^Cin, and c_out = (A&B) | (A&c_in) |(B&c_in). The ripple adder essentially calls the full adder 16 times with the previous c_out connected to the next c_in.

ii. Block diagram.

The Ripple Carry Adder is the simplest adder because it uses the full adder as a building block and chains multiple full adders to create an n-bit Adder. Each full adder takes in three one-bit inputs and two one-bit outputs: A, B, and C-in are the inputs and S and C-out are the outputs. These full adders are chained through the carry-out and carry-in bits, where the carry-out signal from a full adder corresponding to a less significant bit is connected to the carry-in signal of the full adder corresponding to the next significant bit. The sum (S) is calculated through gate-level logic and the carry-out signal from the full adder corresponding to the most-significant bit is the carry-out of the entire adder. Below are block diagrams of the full adder and the Ripple Carry Adder.

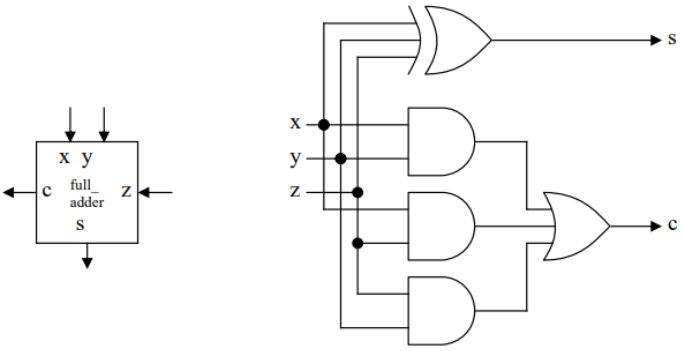


Figure 2: Full-Adder Block Diagram

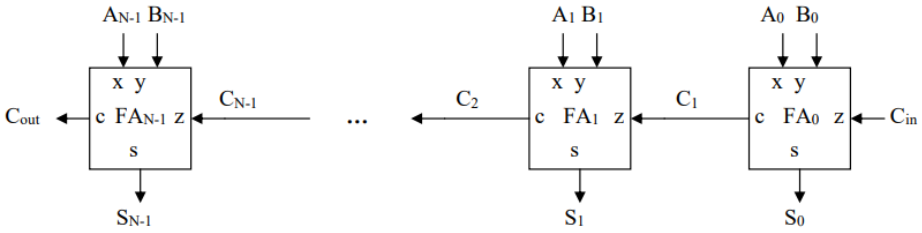


Figure 3: *N*-bit Carry-Ripple Adder Block Diagram

Carry Lookahead Adder

i. Written description of the architecture of the adder

The CLA works by utilizing a 4X4 hierarchical design, and calculating the c_in of each full adder using P and G signals that depend only on c_in. The architecture is composed of 4 4-bit CLA that each have a c_in, and a PG and a GG output. The input to the circuit is A[15:0], B[15:0], and c_in. The outputs are S[15:0] and c_out. Each 4-bit CLA is composed of 4 1-bit full adders. The c_in inputs are calculated using combinational logic that only depend on A, B and the original c_in. This reduces the need for rippling which reduces the time it takes to calculate the results. A 4X4 hierarchical design is used due to hardware limitations of the FPGA lookup tables. Each 1-bit CLA (full-adder) has 4 outputs. S, c_out, P, and G. C_out is left unconnected as it is unneeded. The adder is essentially a 4X4 full adder with an extra module that calculates the c_in of each 1-bit CLA, and the c_in of each 4-bit grouping to reduce the ripple time of the adder.

ii. Describe how the P and G logic are used.

P stands for propagate, and is P = A^B. This means that it will propagate the carry in signal that the previous adder provides. The G stands for Generate which is equal to G = A&B which means that a 1 will be generated for c_out, so the next adder knows what its c_in will be without needing to know c_in of previous adders. Each CLA provides a P and G signal which the carry

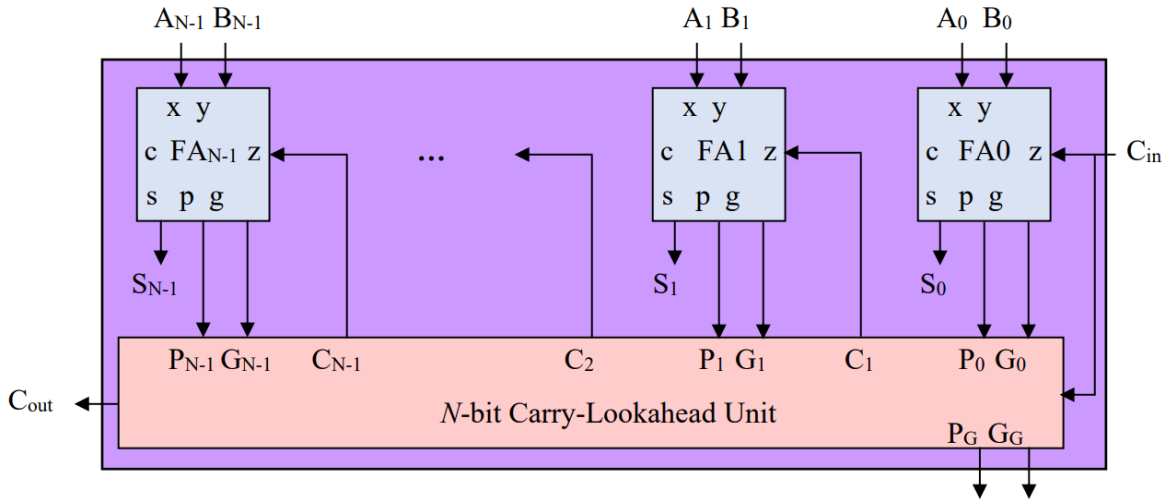
lookahead unit uses to generate a c_in for the next 1-bit adders. 4 CLA are grouped in this way, then a PG and GG signal are calculated using all the P and G signals of each adder. PG = p1&p2&p3&p0, GG = g0&p1&p2&p3 | g1&p2&p3 | g2&p3 | g3. These GG and PG signals are then used to calculate the c_in signal for the next 4-bit CLA using c_in as well. For example c3 = gg8 | gg4&pg8 | gg0&pg4&pg8 | cin&pg0&pg4&pg8.

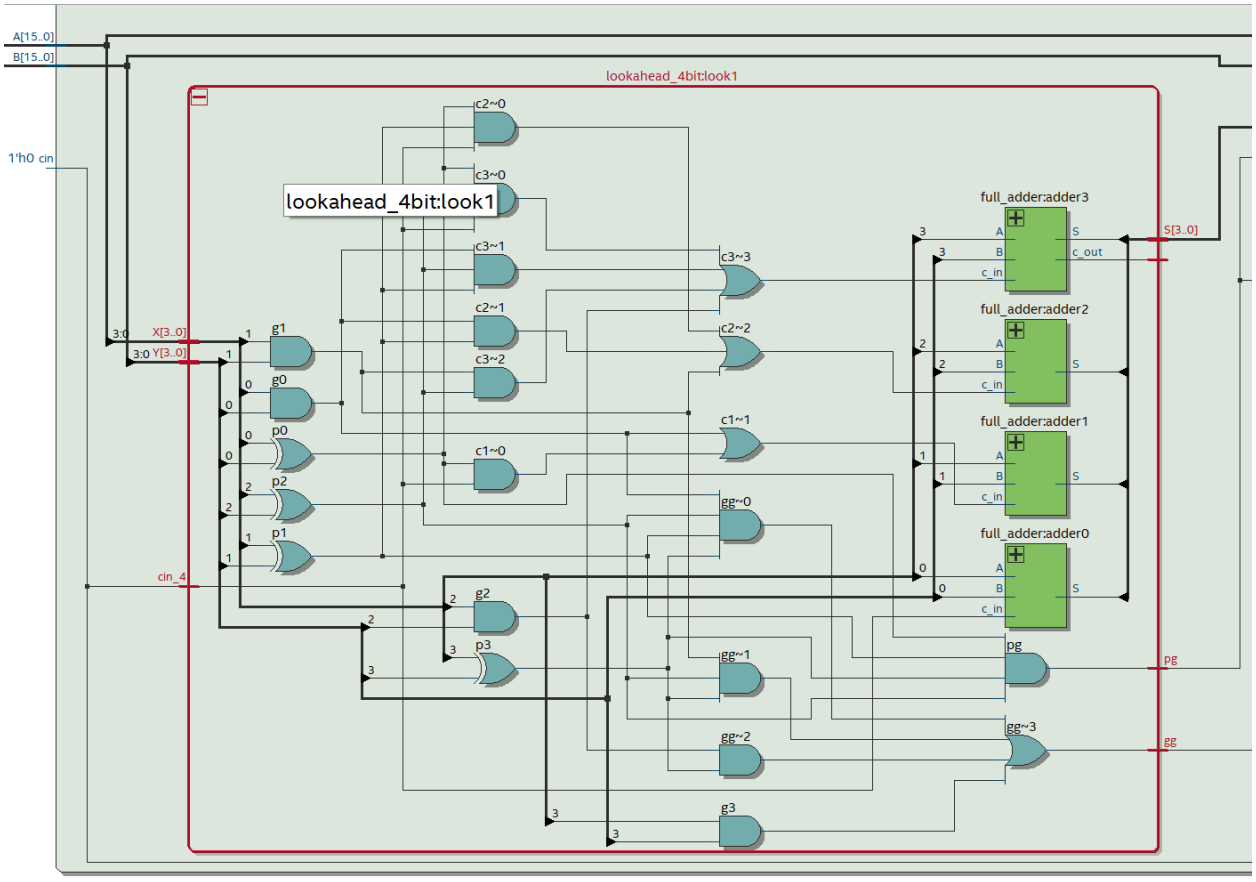
iii. Describe how you created the hierarchical 4x4 adder.

We created the 4X4 adder by following the procedure written in the experiment pdf. The CLA consisted of 4 4-bit CLA, so we began by first creating each group of 4-bits. We saw that the CLA were essentially full adders but with c_in calculated using P and G. We first began by calculating G for each CLA, which was equal to A&B. We then calculated P which is equal to A^B. We then went on to calculate the first c_in (C1). We found that C1 would equal to cin&p0 | g0. This meant that if G of the previous asdder was 1, then C1 = 1, else if P = 1, then cin would pass through to the second adder. We then moved to the second CLA which had C1 = C1&P1 | G1. We used this equation and substituted to get C2 = cin&p0&p1 | g0&p1 | g1. We got this by substituting C1 with the previous equation, this worked because the algebra is commutative. After reaching the 4rth CLA, we calculated PG and GG using the formula pg = p1&p2&p3&p0; gg = g0&p1&p2&p3 | g1&p2&p3 | g2&p3 | g3;. We got PG by thinking that if all Pi =1, then the cin would have to propagate to the end. We got GG by looking at how if g3 was 1, then GG should be 1, if G2 was 1 and P3 was 1 then GG should be 1, and continued with that logic until we reached the end. After getting the PG and GG signals, we used them as the output of the 4-bit CLA. We then used this to calculate the cin for the next 4-bit CLA using cg1 = gg0 | pg0&cin. This essentially follows the same logic as the 1-bit CLAs but uses PG and GG to simplify the equation due to FPGA hardware limitations. We continued this until we reached the end, where the output was complete. We create two modules; lookahead_adder and lookahead_adder_4. We used the lookahead_adder as the main module which instantiated the lookahead_adder_4 using PGs and GGs as the cin inputs to fulfill the hierarchical design.

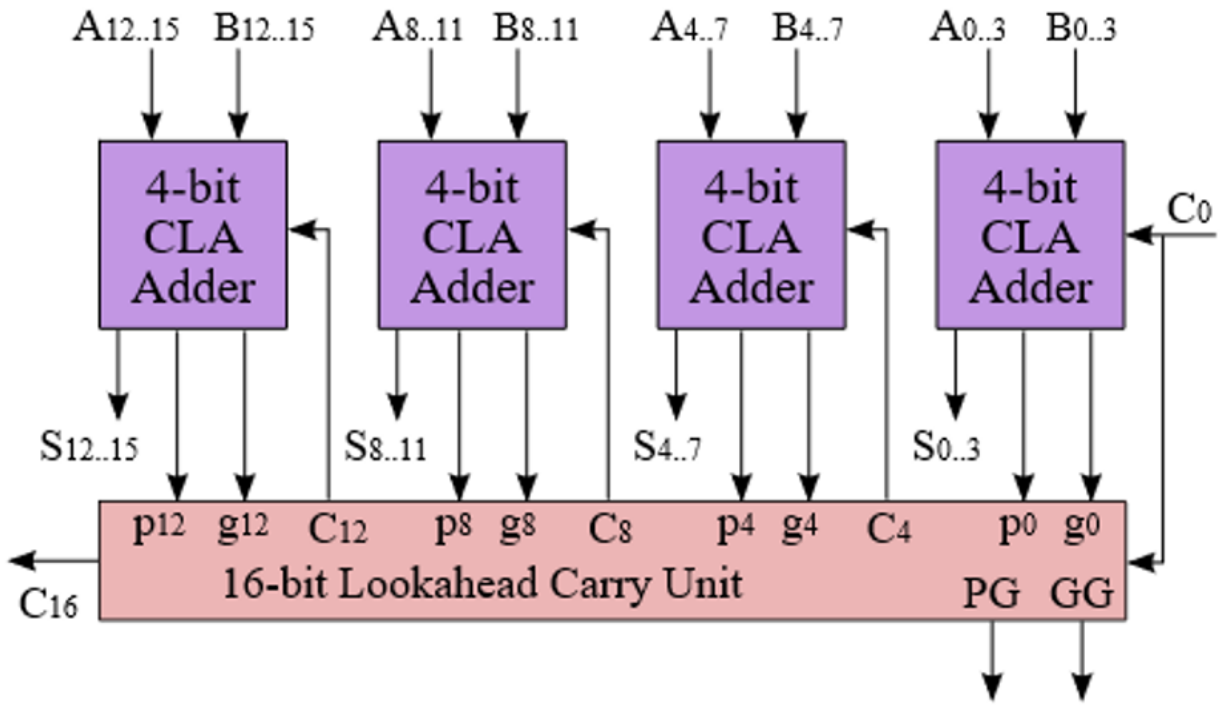
iv. Block diagram

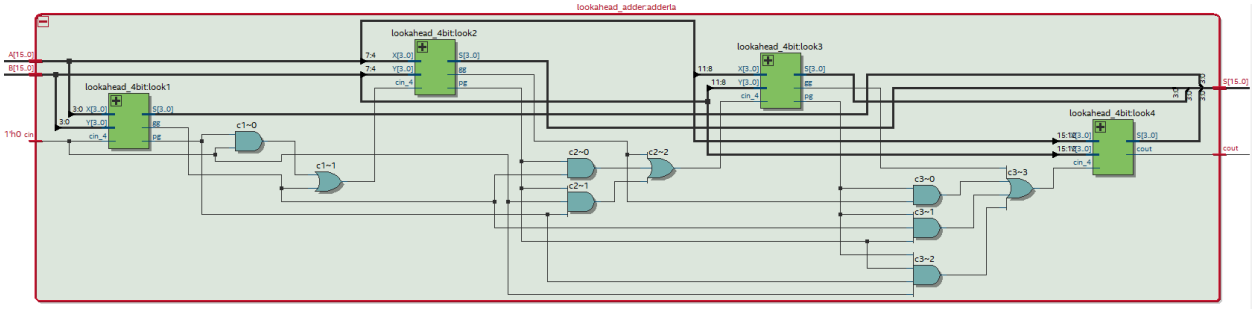
1. Block diagram inside a single CLA (4-bits)





2. Block diagram of how each CLA was chained together





Carry Select Adder

i. Written description of the architecture of the adder

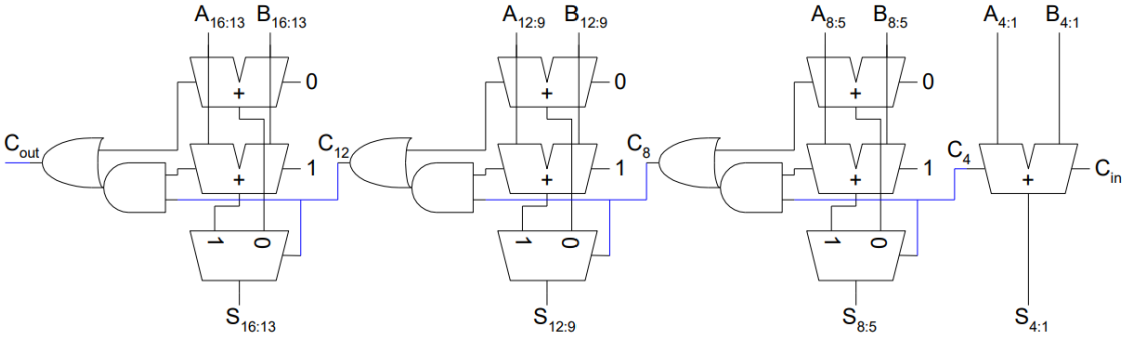
The carry select adder works by partitioning the 16-bits into a 4X4 hierarchical design. The CSA consists of 8 4-bit carry adders. The adder consists of 4 sets of two 4-bit ripple adders. Each ripple adder has either 1 or 0 as the c_in to the adder. This essentially computes both possible outputs at the same time. The output of each set of two ripple adders, the SUM and C_out goes into two MUXes, with the select bit as the C_in species by the previous set of two ripple adders. The first set only has c_in as the input, but the rest have their MUX selected as c_out(previous). The circuit essentially computes the two possible combinations of outputs (each 2 4-bit adder)

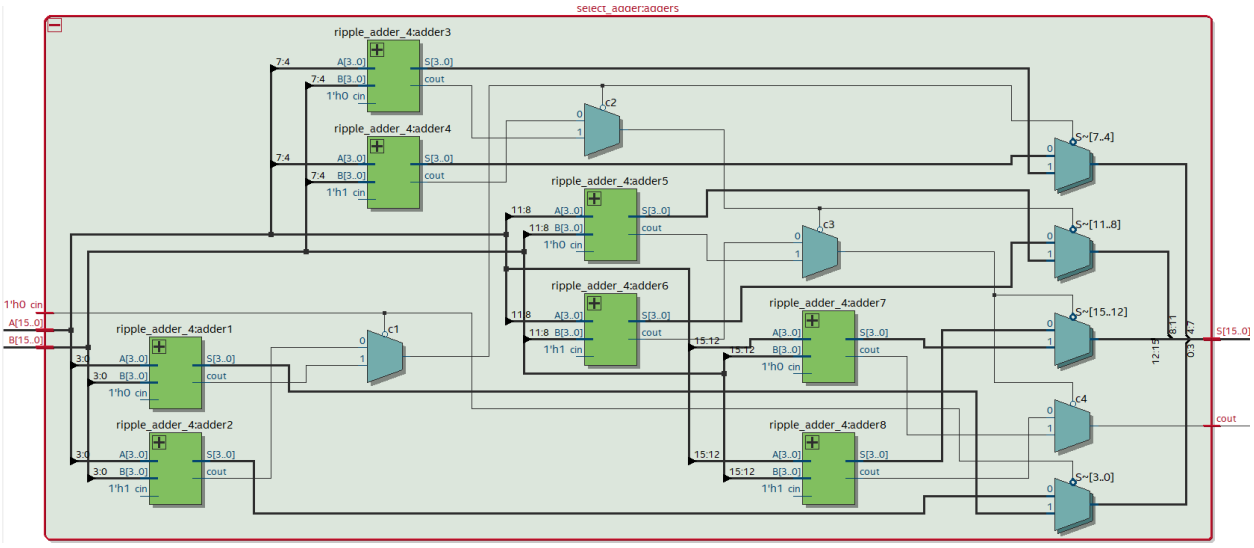
then connects to a MUX which dictates which output goes into the next set of adders or to the final output. The SUM MUXes feed into the output, while the c_out MUXes feed into the next 2 4-bit adders. The input to the adders is the same “There are three inputs, A[15:0], B[15:0], and cin. There are two outputs S[15:0], and cout.”.

ii. Describe at a high level how the CSA speculatively computes multiple sums in parallel and rapidly chooses the correct one later. Make sure you understand this!

The CSA computes the sums in parallel as it calculates each possible sum for each 4-bit block and then feeds it into the MUX. It calculates all the Sum and C_out values at the same time, so by the time the first 4-bit ripple adder calculates the Sum, all others have finished as well. Although there is an extra amount of delay which comes from the MUX’s. There are 8 MUXs in the implementation. The MUX delays can be thought to be negligible in comparison to the adder's delay. You need to wait for 1 MUX to get S{3:0}, 2 MUXes to get S[4:7], 3 MUX delays to get S[8:11], and all 4 MUXes to get S[12:15].

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





Written Description of .sv Modules

Module: **full_adder** within ripple_adder.sv

Inputs: a, b, c_in

Outputs: s, c_out

Description: This module is the full adder that takes in the inputs a, b, and c_in and adds them, generating the outputs s and c_out. This is done through the formulas s = a⊕b⊕c_in, and c_out = (a&b) | (a&c_in) | (b&c_in). Below is a truth table that displays how these functions are just adding these inputs.

a	b	c_in	s	c_out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Purpose: The purpose of this module is to be a building block for the ripple carry and the carry select adders. Since they work with only one bit they are easy to construct and comprehend.

Module: **ripple_adder_4** in ripple_adder.sv

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

Outputs: S[3:0], cout

Description: This module takes in two four-bit inputs (A and B) as well as a carry-in bit and returns a four bit sum (S) and a carry-out bit. Essentially, this module is a 4 bit ripple carry adder, a smaller version of the 16 bit ripple carry adder module that we created. It uses four full adders chained together through their carry-out and carry-in signals.

Purpose: This module is used when constructing the 4x4 hierarchical Carry Select Adder, since we are dividing the 16-bit CSA into four bit ones that each need two copies of module full_adder_4.

Module: **ripple_adder** in ripple_adder.sv

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

Outputs: S[15:0], cout

Description: This module takes in two sixteen-bit inputs (A and B) as well as a carry-in bit and returns a sixteen-bit sum (S) and a carry-out bit. This module is simply an extension of the ripple_adder_4 module, where instead of having four full adders chained together, we have sixteen full adders chained together. We did not use the full_adder_4 module to implement the ripple_adder module but instead decided to instantiate the full_adder module sixteen times, but either way would be a proper implementation of the ripple adder module for the lab demonstration.

Purpose: This module is the one used for the lab demonstration of a sixteen bit Ripple Carry Adder. Most basic form of adders.

Module: **router** in router.sv

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

Outputs: Q_Out[16:0]

Description: This module is a seventeen bit parallel multiplexer constructed through case statements. This module is used to load register B with the Sum of SW+B if R = 1, otherwise it passes through the SW values. The output of this module feeds into the Din of the register module.

Purpose: This module provides the output which goes into the Register input to be loaded into it. It is used to load Reg B with the switches, and to provide the D_in to load B with the Sum after the computation.

Module: **reg_17** in reg_17.sv

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

Outputs: Data_Out[16:0]

Description: This module is a positive edge clock-triggered seventeen-bit register. On the positive edge of the clock and when reset signal is high, the register is initialized with zeros. When the load signal is high, D[16:0] is loaded into the register.

Purpose: This module allows for the creation of seventeen bit registers to store the operations of A and B in the adder circuits. The module is used in the adder2 module.

Module: **control** in control.sv

Inputs: Clk, Reset, Run

Outputs: Run_O

Description: This module is the control unit of the circuit. The control unit is in the reset state, then if Run is pressed it moves onto the Run state. At this state the Run_O = 1 which loads the B register with the Sum of SW + B. This data comes from the adder which is connected to the router, which is connected to the register. When Run_O = 1 the register constantly loads the

Sum from the adder into the register. The control unit stays on the Run state for 1 clock cycle which it then moves to the halt state where it stays until Run goes to 0 again, then it goes to the rest/reset state.

Purpose: This module controls the behavior of the circuit. It essentially decides when to load the reg B and when to halt or rest. This ensures that the adder executes for only 1 clock cycle then stops. This also ensures that the loading is done properly.

Module: **testbench** in testbench.sv

Inputs: N/A

Outputs: N/A

Description: The testbench module interfaces with the three adder circuit designs and allows for the simulation of a clock cycle. There is a variable that checks if the value outputted by the selected adder is equivalent to the expected output of an addition, and is increased by one if there is an error.

Purpose: The purpose of this module is to allow for the simulation of the adder circuit designs through Modelsim. When these designs are simulated and tested through values set within the testbench file, it is easier to debug the SystemVerilog code and fix any errors.

Module: **lookahead_4bit** in lookahead_adder.sv

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

Outputs: S[3:0], cout, PG, GG

Description: This module takes in two four-bit inputs (A and B) as well as a carry-in bit and returns a four-bit sum (S) and a carry-out bit. The circuit uses the propagate and generate logic as described in the written description of the CLA and outputs and also generates the PG and GG outputs.

Purpose: The purpose of a four-bit lookahead adder is for us to implement the sixteen-bit lookahead adder using the 4x4 hierarchical design. This module is used four times to create the hierarchical design for the sixteen-bit lookahead adder that is used for the lab demo.

Module: **lookahead_adder** in lookahead_adder.sv

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

Outputs: S[15:0], cout

Description: This module takes in two sixteen-bit inputs (A and B) as well as a carry-in bit and returns a 16-bit sum (S) and a carry-out bit. The circuit uses the propagate and generate logic as described in the written description of the CLA and outputs.

Purpose: This adder is one of the three demonstrated in the lab demo and is used as the primary module for the Carry Lookahead Adder design. Used to decrease the ripple propagation delay as much as possible.

Module: **HexDriver** in HexDriver.sv

Inputs: In0[3:0]

Outputs: Out0[6:0]

Description: The module takes in four-bit values and converts the four-bit numbers into their corresponding hexadecimal number from 0-F. These values are hardcoded and are assigned through a case structure.

Purpose: This module converts the four-bit binary inputs into hexadecimal numbers to be able to program the LED display on the FPGA unit.

Module: **adder2** in adder2.sv

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

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

Description: This module is the top-level entity and is used to load the adder into the FPGA board and calls the adder that is wanted based on the addition and removal of the comments. It also calls on the HexDriver to convert four-bit binary numbers for the loads and the output into hexadecimal numbers, to be displayed on the LED board through the output LED[9:0].

Purpose: This module interfaces with the FPGA and the LEDs on the FPGA to provide the proper functionality to have I/O on the FPGA board, so that the loading of the registers and the addition operation can be solely performed through the FPGA board, and the actual outputs can be read through the LEDs that display the outputs in hexadecimal.

Module: **select_adder** in select_adder.sv

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

Outputs: S[15:0], cout

Description: This adder takes the two 16-bit inputs A and B and adds them into S. This module utilizes a 4x4 hierarchical design where it calls the ripple_adder_4 for cin = 1 and cin = 0 for each 4-bit block. The module consists of four if statements, depending on cin(c0), c1, c2, and c3. Each if statement takes the previous cin as the condition, and if it is 1 then S = the sum where the cin matches the condition, and cout = cout where cin matches the condition. The cout is then used as the condition for the next 4-bit block.

Purpose: This adder is one of the three demonstrated in the lab demo and is used as the primary module for the Carry SelectAdder design. It is used to decrease the time it takes for the ripple to propagate through the circuit.

Tradeoffs Between Adders

Since the Ripple Carry Adder is simply made up of a chain of full-adders without extra logic, it uses the least amount of gates and therefore it costs the least amount of area to be able to implement the adder. It is also the least complicated to create because of the same full-adder module being used n times without extra logic to scale up. However, the Ripple Carry Adder has the greatest computation time and therefore may perform the worst due to the fact that the full-adders depend on the carry-in input from the previous full-adder, so the propagation time increases linearly with an increase in the amount of full-adders in a chain.

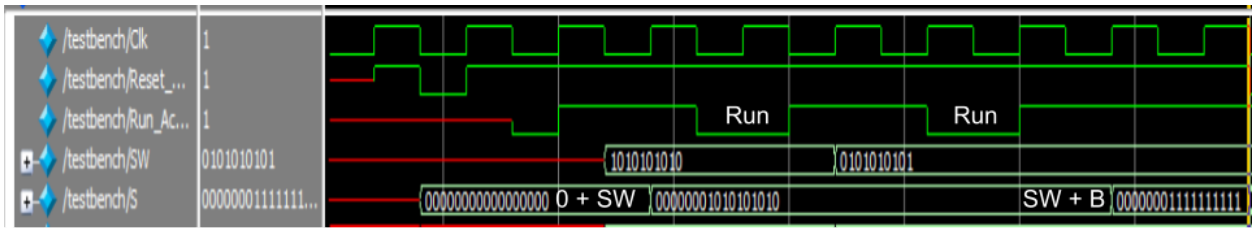
Contrary to the RCA, the Carry Lookahead Adder uses the greatest amount of gates out of the three adders. There is greater complexity in implementing the CLA. This is due to the extra logic required for the propagating and generating logic to be able to compute the carry-out bits in parallel. However, the problem of each adder having to wait for the lower-bit adders to propagate a signal for carry-in is solved and reduces the computation time, and therefore the performance is better in regards to the Ripple Carry Adder.

The Carry Select adder uses more gate level logic due to the mux and double the amount of full adders when compared to the RCA. While in theory the pre-computation should decrease the computation time, the performance results we got were actually lower than the ripple adder. This might have been due to the low resolution of the timing-analyser and maybe because the adder was not large enough to cause a significant difference. The double computation while in theory should be in parallel might not have worked as expected.

Annotated Simulation Trace

- You may just include just a single annotated simulation trace as all the RTL simulation does not include any gate delays.

Simulation of the adder2 program, the ripple_adder was used in this simulation. 288 + 155 = 3FF is computed. Note that the program executes at Run = 0.



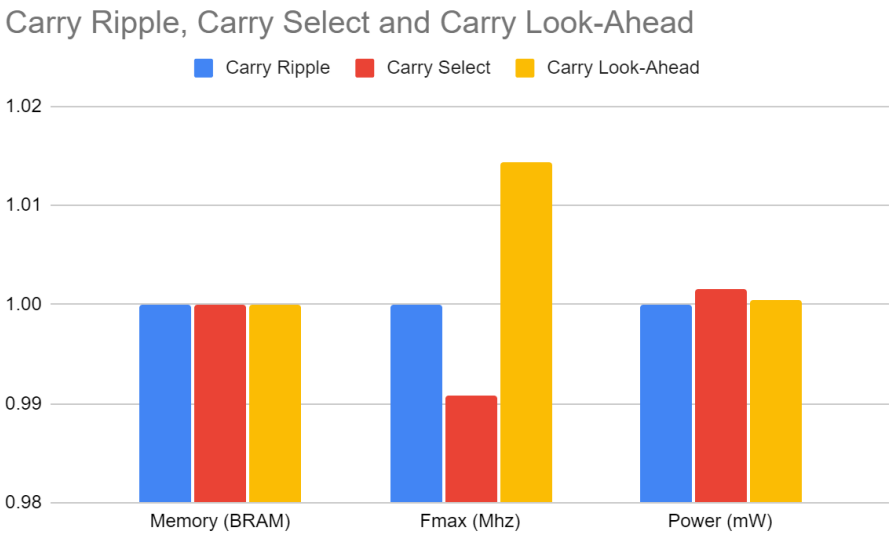
Performance Analysis:

Un-Normalized

	Carry Ripple	Carry Select	Carry Look-Ahead
Memory (BRAM)	0	0	0
Fmax (Mhz)	67.3	66.68	68.27
Power (mW)	107.39	107.56	107.44

Normalized

	Carry Ripple	Carry Select	Carry Look-Ahead
Memory (BRAM)	1	1	1
Fmax (Mhz)	1	0.9907875186	1.014413076
Power (mW)	1	1.001583015	1.000465593



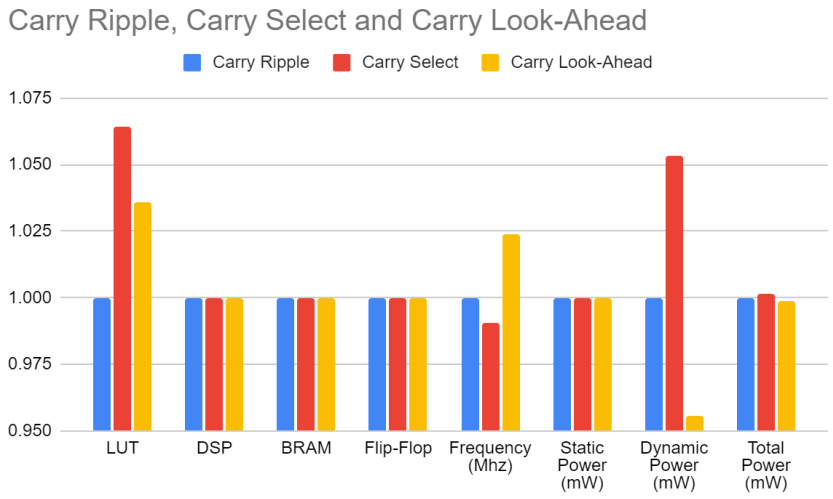
Postlab Questions

1. The 4x4 hierarchy design could be ideal, because this means you are simplifying the design of the overall adder and it also reduces the number of inputs that are used in the lookup tables. To truly understand which design would be ideal, it would be best to try different configurations such as a 2x8 hierarchy design and run a timing analysis on them to see which configuration gives us the best results in terms of performance and if there is a significant difference by changing up the configuration. We probably wouldn't use a hierarchical design if doing a small number of bits, or if not needed, this is to simplify design and debugging. The hierarchical design allows for more flexibility but consumes

more power and is more complex, so it might not be ideal in situations where efficiency is very important.

2. 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

	Carry Ripple	Carry Select	Carry LookAhead
LUT	78	83	86
DSP	0	0	0
BRAM	0	0	0
Flip-Flop	20	20	20
Frequency (Mhz)	67.3	66.68	68.27
Static Power (mW)	89.98	89.98	89.98
Dynamic Power (mW)	1.5	1.58	1.51
Total Power (mW)	107.39	107.56	107.44



Most of the results make sense. The LUT are the lowest in the ripple, second in the select, and highest in the lookahead. This is because the lookahead has the most formulas and extra

computations needed. The memory and flip-flops are the same as they all use the same input/output registers and don't have any sequential component in the adder. The frequency on the other hand is a bit perplexing, while it makes sense that the lookahead has the highest frequency, the select is actually lower than the ripple adder which is unexpected. While we are unsure why this occurs, this might be because the extra computations take time, and while in theory they should be in parallel, they aren't executed that way. In terms of power, they all consume the same static power, but the ripple suits the least dynamic power, and the select adder consumes the most. This is because the carry select essentially computes the sum twice which takes more power than one time. The lookahead adder takes more power because computing the P and G signals takes energy, but not a significant amount.

Conclusion

Overall, the lab was a great introduction for students to get their own experience writing SystemVerilog code from scratch, as well as getting to experience the applications of meaningful concepts explored in this class and ECE 120 through the practice of creating three different types of adders. It’s also a very important practice to be able to understand the advantages and disadvantages of using different implementations of the circuits with an identical or near-identical functionality. The ability to understand tradeoffs is an important intuition that is valuable when designing applications with constraints.

The lab manual was straightforward and not excessively hard to understand. An improvement that can be made is that the specific manuals should be more clearly named, and possibly linked, as it got confusing when we were working with multiple manuals and lab documents. We had some trouble trying to implement the carry lookahead adder when it came to computing the PG and GG signals for the four-bit lookahead adder module and then using these signals to create the sixteen-bit hierarchical structure, but that is due to us misinterpreting that part of the lab manual. That was definitely one of the more challenging aspects of this lab. Another challenging aspect was designing a test bench and debugging the circuit, which we solved by trying to isolate the issues.