University of Illinois at Urbana-Champaign

ECE 385

LAB 3

LAB REPORT

by Shubham Gupta & Devul Nahar (sg49 & danahar2)

2/20/2022

Introduction:

In this lab we design 3 different 16-bit adders which are hierarchical in design. The first adder is a ripple carry adder (RCA) which is made up of four 4-bit ripple carry full adders, its architecture depends on rippling of carry out bits from one-bit full adder to another to produce the sum of the two numbers. The second adder is also a 16-bit adder but is a carry look ahead adder (CLA). Its architecture is hierarchical in design and is made up of four 4-bit carry look ahead adder. It primarily generates the sum by computing propagate and generate bits. Lastly, we have a 16-bit carry select adder (CSA) which is also a hierarchical design being made up of seven 4-bit carry select adders. It primarily computes the sum by pre-computing answers for carry in bit as 1 and 0. Finally, based on carry in, we simply select our answer.

Adders:

a. **Ripple Carry Adder:**

   i. We made a 16-bit ripple carry adder by chaining four 4-bit ripple carry adders together. Though in case of ripple carry adder this does not make a difference in the performance, this was implemented just to maintain a consistent modular design throughout the lab. The underlying idea is to ripple the carry out bit between the 15 full adders, just like human addition. As soon as carry in bit is given to the 1st bit or the least significant bit, it calculates the 1st bit of the sum or SUM[0] and depending on the values of A[0], B[0], and carry in it produces a carry out bit. The carry out is then fed into 2nd full adder, which then takes inputs A[1], B[1], and carry out from FA1 to compute SUM[1] and carry out. Similarly, the rippling is carried on up to FA16 (16th 1-bit full adder) which then computes

SUM[15] and carry out for the 16-bit answer. Now let us see how 1-bit full adder computes its outputs.

$$S = A \oplus B \oplus c\_in$$

$$c_{out} = (A \,\&\, B) \mid (B \,\&\, c_{in}) \mid (A \,\&\, c_{in})$$

Hence, we can compute c_out and S based on our inputs in 1-bit full adder. The time complexity of this adder is O(n) as it must ripple through n bits before the final answer is computed.

ii. **Block diagram of Ripple Carry Adder:**

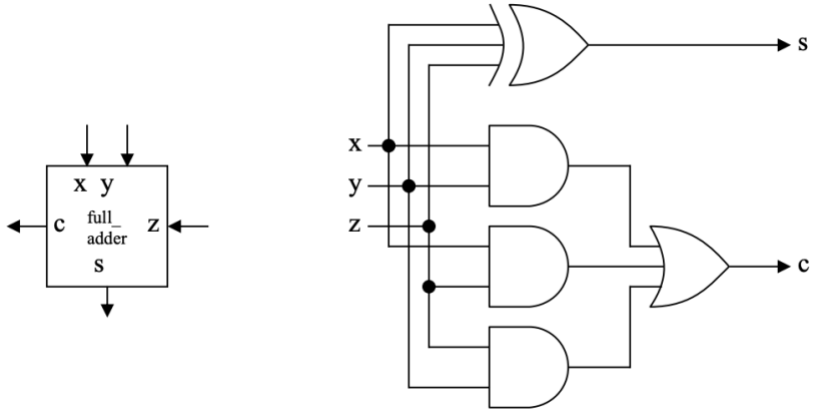

Figure 1 1-bit Full Adder

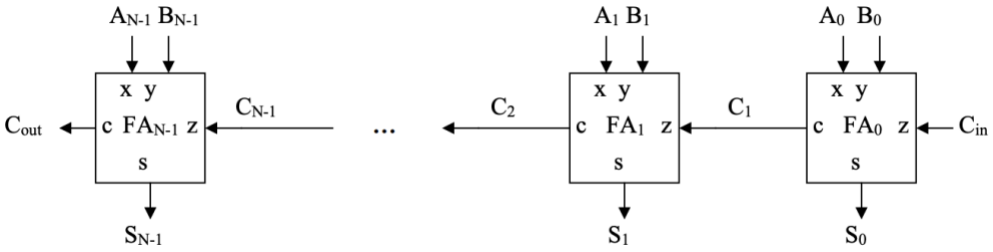

Figure 2 16-bit RCA Block Diagram

b. **Carry Look Ahead Adder:**

i. This is also a 16-bit carry adder, made up of four 4-bit carry look ahead adders. Each of the 4-bit carry adder modules are made up of 1-bit full adders. Now instead of waiting on the actual carry-in values, CLA uses the concept of generating (G) and propagating (P) logic. These 1-bit adders generate two more signals, propagate (P), and generate (G), in addition to SUM. With the help of P and G we can calculate our answer significantly faster.

ii. Propagate (P) is a 1-bit signal generated by every 1-bit full adder. If P is high, it means the carry out should simply be propagated to the next full adder from the previous full adder. Generate (G) is a 1-bit signal generated by every 1-bit full adder. If G is high, it means that carry out should be output as 1. It is calculated by,

$$G(A, B) = A \,\&\, B$$

Similarly based on immediate inputs A and B, P is generated by the equation,

$$P(A, B) = A \oplus B$$

With P and G defined, the expression for the carry-out C_{i+1} giving a potential C_i is then $C_{i+1} = Gi + (Pi \cdot Ci)$. Notice how the expression for ith carry out is made up of i-1 terms. Therefore, we can unroll our expressions for all the carry outs in terms of the carry in along with Pi and Gi.

$$C_0 = C_{in}$$
$$C_1 = C_{in} \cdot P_0 + G_0$$
$$C_2 = C_{in} \cdot P_0 \cdot P_1 + G_0 \cdot P_1 + G_1$$
$$C_3 = C_{in} \cdot P_0 \cdot P_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + G_1 \cdot P_2 + G_2$$

iii. We created the 4x4 design by creating 4 instances of 4-bit carry look ahead adders. Each of the 4-bit CLA just like 1-bit CLA generate the signals Propagate group (Pg) and Generate group (Gg). These are given by,

$$P_G = P_0 \cdot P_1 \cdot P_2 \cdot P_3$$
$$G_G = G_3 + G_2 \cdot P_3 + G_1 \cdot P_3 \cdot P_2 + G_0 \cdot P_3 \cdot P_2 \cdot P_1$$

Now based on Pg and Gg for each 4-bit CLA, we can generate the carry out bits called C4, C8, C12…

$$C_4 = G_{G0} + C_0 \cdot P_{G0}$$
$$C_8 = G_{G4} + G_{G0} \cdot P_{G4} + C_0 \cdot P_{G0} \cdot P_{G4}$$
$$C_{12} = G_{G8} + G_{G4} \cdot P_{G8} + G_{G0} \cdot P_{G8} \cdot P_{G4} + C_0 \cdot P_{G8} \cdot P_{G4} \cdot P_{G0}$$

Now we can simply chain the four 4-bit CLAs together to make a 4x4 hierarchical 16-bit CLA.

iv. **Block Diagrams**

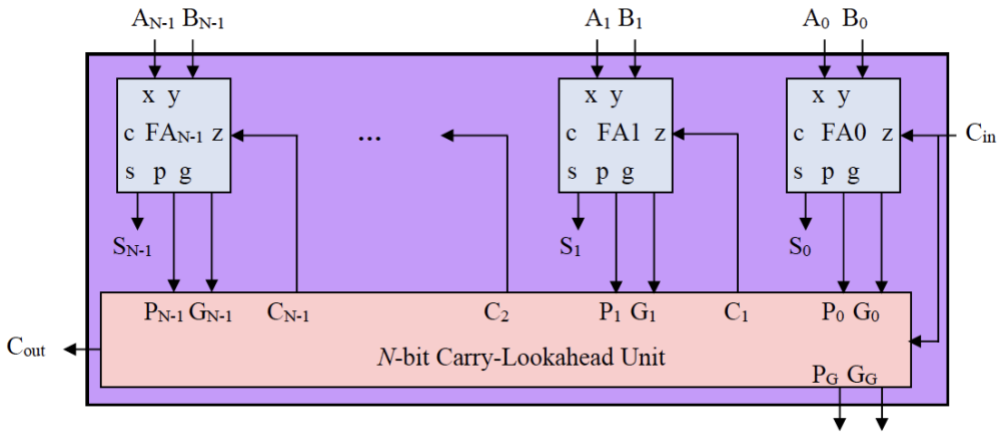

Figure 3 N-bit block diagram (4-bit)

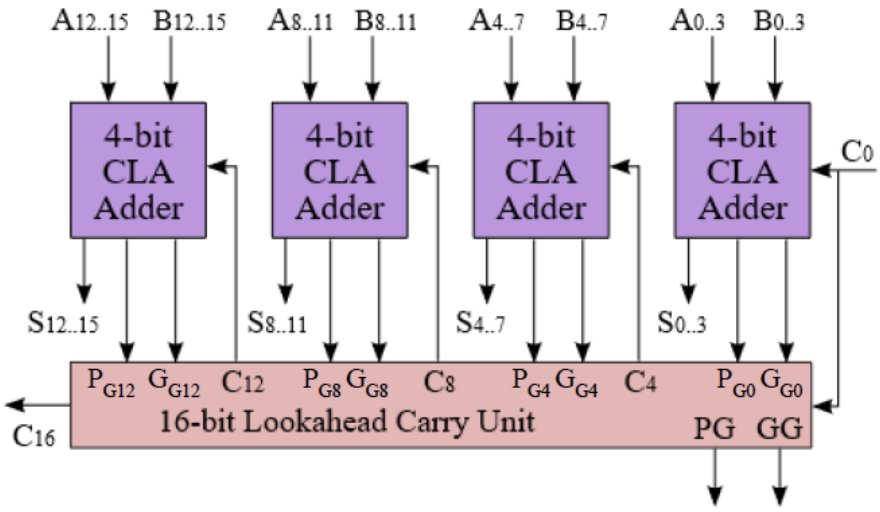

Figure 4 16-bit CLA Block Diagram

c. **Carry Select Adder:**

i. It is a 16-bit full adder which is made up of seven 4-bit ripple adders. Here, we have divided the 16 bits into parts of 4, 4-bit each. Each part's sum is calculated by two ripple carry adders (except for the lower 4-bits). With this implementation we can parallelly process our answers for each of the four parts without waiting for the rippling carry in bit. This is done by having two 4-bit ripple adders for each part, one that calculates with carry in as 1 and the other with a carry in as 0. This computation is started as soon as A and B can been set as carry-ins are set to 0s and 1s. We have then made use of 2 to 1 MUXes to select our desired output based on carry in bit, and the answer is calculated almost instantly.

ii. As we infer from our architecture, we have divided 16 bits into four parts, each part having 4-bits. As soon as the input bits of A and B are set, each of the four

parts parallelly starts to compute its answers, there is no waiting period for a carry in as each part contains two ripple carry adders, one with carry in as 1 and the other with 0. Hence all the four parts are ready with 2 variations of answers at the same time. Now, as soon as the carry in bit is set, with the help of 2 to 1 MUX and some combinational logic, we simply need to select one variation of the answer from each part based on the carry in. Since each variation is ready with its answer as well as its carry out, this generates our answer rapidly.

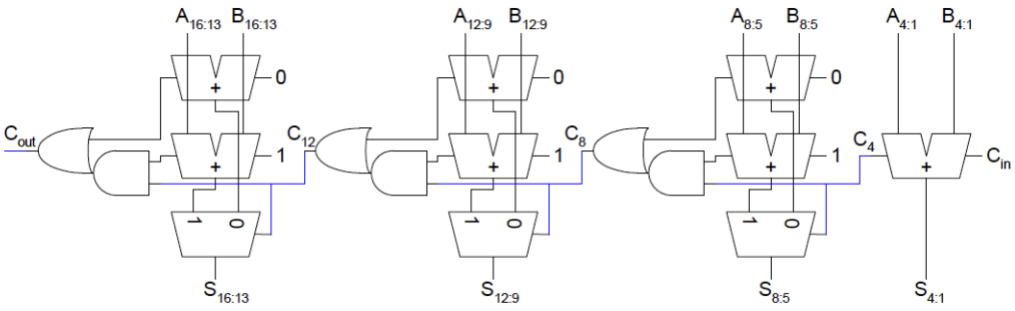

Figure 5 CSA Block Diagram

d. **Description of modules:**

i. **Module:** FA.sv

Inputs: A, B, c_in

Outputs: S, c_out

Description: It is an implementation of 1-bit full adder.

Purpose: It is used in many modules; it is a fundamental unit full adder.

ii. **Module:** FA_CLA.sv

Inputs: A, B, c_in

Outputs: S, c_out, P, G

Description: It is an implementation of 1-bit full adder. It also generates bits P and G mentioned in a CLA.

Purpose: It is used in the implementation of 4-bit CLA; it is a fundamental unit of Carry Lookahead Adder.

iii. **Module:** FA4.sv

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

Outputs: [3:0] S, c_out

Description: Creates four instances of 1-bit full adders to make a 4-bit ripple carry adder. It also links the required inputs and outputs.

Purpose: This module is further used in carry ripple adder and carry select adder implementations.

iv. **Module:** FA4_CLA.sv

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

Outputs: [3:0] S, c_out, Pg, Gg

Description: Creates four instances of 1-bit CLAs to make a 4-bit CLA. Further, it generates the output for bits Pg and Gg, to achieve hierarchical implementation of 16-bit 4x4 CLA. It also links the required inputs and outputs.

Purpose: This module is further used in 16-bit CLA implementation. This module implements the 4x4 hierarchical design in the 16-bit CLA.

v. **Module:** reg_17.sv

Inputs: [16:0] D, Clk, reset, load

Outputs: [16:0] Data_Out

Description: It is a 17-bit register. It resets to 0 when reset is high, else loads 17-bit data when load is high. It outputs the 17-bit data.

Purpose: It is used to store the values of A, B, and Sum.

vi. **Module:** Router.sv

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

Outputs: [16:0] Q_Out

Description: It is a 17-bit parallel multiplexer implemented using case statements.

Purpose: It is used to decide whether to load data from switches on the FPGA or not, also used to load values to A, B, and Sum.

vii. **Module:** HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: We use UNIQUE switch case to map the binary values to the hexadecimal display.

Purpose: It maps the binary values to the hexadecimal LED display. Used for displaying values stored in registers A, B, and the Sum. For user interface on the FPGA mainly.

viii. **Module:** Control.sv

Inputs: Clk, Reset, Run

Outputs: Run_0

Description: It the FSM implemented to run the add and load operations. It generates the required signals in every state.

Purpose: Used to control the Adder2 module, to implement the adder.

ix. **Module:** adder2.sv

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

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

Description: It initializes all the required inputs and outputs. It creates instances of registers, multiplexers, adders, and hex drivers. The control unit is also instantiated and connected to the components. It is the top level entity.

Purpose: Implements the complete I/O of our adder on the FPGA

x. **Module:** ripple_adder.sv

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

Outputs: [15:0] S, cout

Description: Instantiating four 4-bit ripple carry adder instances and connecting the input/outputs.

Purpose: Final implementation of the 16-bit ripple carry adder.

xi. **Module:** lookahead_adder.sv

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

Outputs: [15:0] S, cout

Description: Instantiating four 4-bit carry look ahead adder instances and connecting the input/outputs. Alongside, initializing the combinational logic to chain them together.

Purpose: Final 4x4 hierarchical implementation of the 16-bit carry lookahead adder.

xii. **Module:** select_adder.sv

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

Outputs: [15:0] S, cout

Description: Instantiating seven 4-bit carry ripple adder instances and connecting the input/outputs. Alongside, initializing the combinational logic to chain them together. Also initializing the 2 to 1 MUX for selecting the answer.

Purpose: Final hierarchical implementation of the 16-bit carry select adder.

e. **Area, complexity, and performance tradeoffs between the adders:**

i. Area:

1. Carry ripple adder takes the least area in terms of implementation, because it just simply chains sixteen 1-bit full adders. It is followed by carry select adder as it implements seven 4-bit ripple adders and a few MUXes. Lastly, we have the carry look ahead adder which takes the most space. This is because of the Pg and Gg expressions, also the expressions inside the 4-bit CLA for the last carry out bit, which is in terms of C0, C1, and C2. These combinational gates end up occupying more than the carry select adder.

2. Here the tradeoff is that the carry ripple adder occupies the least area but is the slowest among the three. While CLA is the fastest in terms of performance but occupies the most space.

By Area: CRA < CSA < CLA

ii. Complexity:

1. By design, carry ripple adder is the easiest to understand and design. Then we have the carry select adder since it is more intuitive as compared to carry look ahead adder, also unlike carry look ahead adder, carry select adder has easier combinational logic that is implemented. Carry look ahead adder is placed last as the complexity increases as we unroll the expression for carry out bit depending on previous P's and G's.

2. The tradeoff is that carry ripple adder is the simplest and the easiest to implement but is the slowest in terms of performance as it takes time to ripple through. CLA is the fastest in terms of performance but also the most complex to design.

By Complexity: CRA < CSA < CLA

iii. Performance:

1. The carry look ahead adder is the fastest with the time complexity of O(logn) because it generates the carry-in of each full adder simultaneously without causing any delay and does not wait for any ripple carry in. It implements the hierarchical design of 4x4 essentially reducing the problem into half at every step. Hence it is O(logn). The carry select adder has a time complexity of $O(n^{1/2})$. Lastly, we have the carry ripple adder with the time complexity of O(n) as it must ripple through N-1 bits.
2. Clearly the tradeoff is that the CLA is the fastest but occupies the most space. CRA is the slowest but the easiest to implement and takes the least space.

By Performance: CLA > CSA > CRA

f. Performance graphs:

	Carry-Ripple	**Carry-Select**	**Carry-Lookahead**
Memory (BRAM)	0	0	0
Frequency	64.47 MHz	67.8 MHz	66.57 Mhz
Total Power	105.11 mW	105.34 mW	105.29 mW

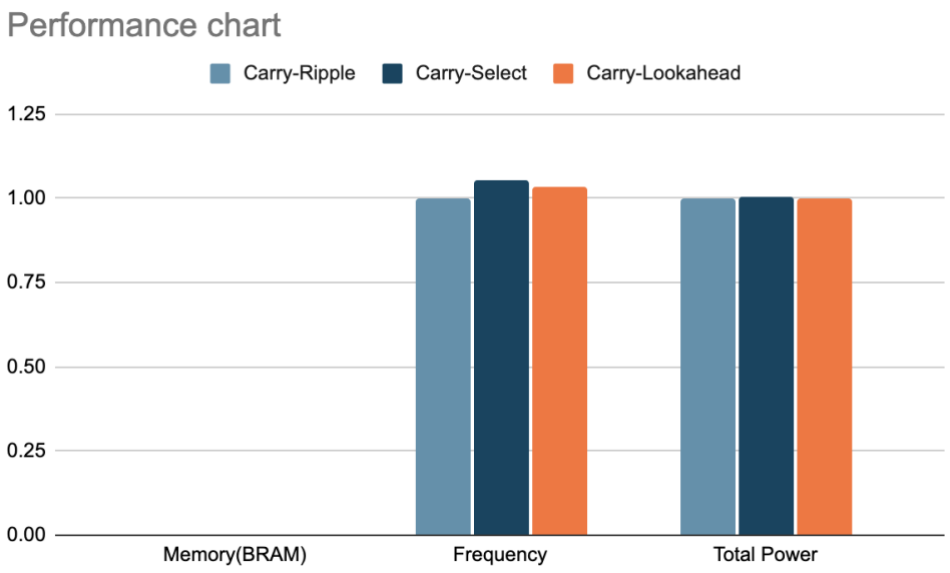

Figure 6 Performance chart

g. Annotated Simulation Trace:

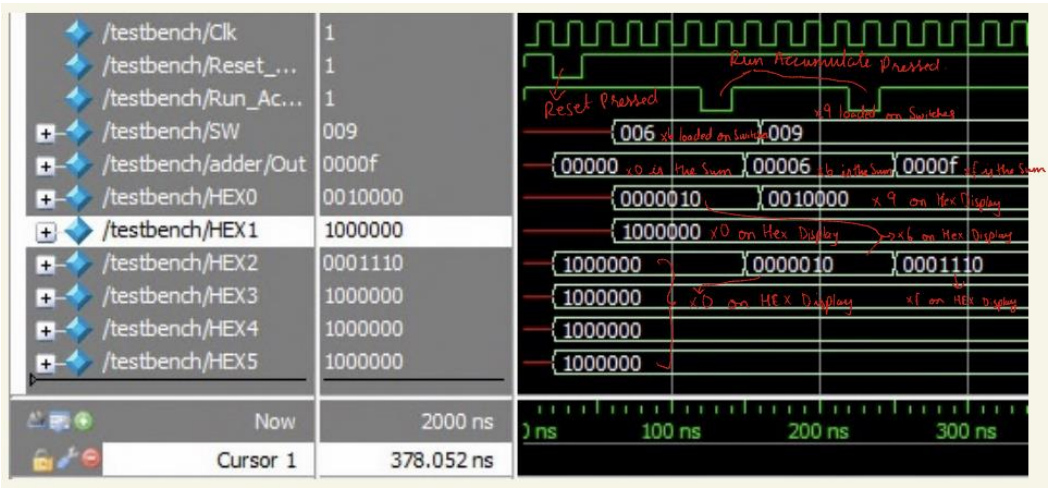

Figure 7 ModelSim Simulation

As one can see from the annotated modelsim diagram, we are doing the addition of x6 and x9, resulting in an output of xF. Firstly, the rest button is pressed to clear whatever is stored as sum. Now the value x6 is put in as input to the adder using switches. We then press run accumulate to add x6 to the overall sum. Therefore, we see the sum become x6 when run

accumulate is pressed for the first time. We now, input x9 into the switches, and press run accumulate again to see the overall output change to xF. We can see the hex displays 0-1 to see whatever we are adding to the overall sum. Finally, the output is clearly visible in the HEX 2-5. Note that the bit values shown in the annotated diagram, maps to each LED inside the HEX display mapped as on or off.

h. Extra Credit:

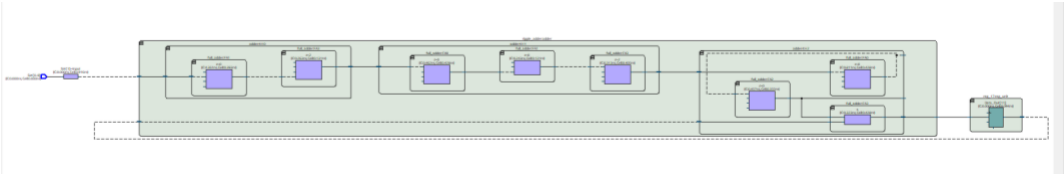

Figure 8 CRA

The above diagram represents the slowest path through the ripple carry adder. Here we have the 10-bit input coming from the switches going as input to the ripple carry adder module. Inside the ripple carry adder, we see c_in going through all 8 full adders and then the final output being stored in the 16-bit register unit.

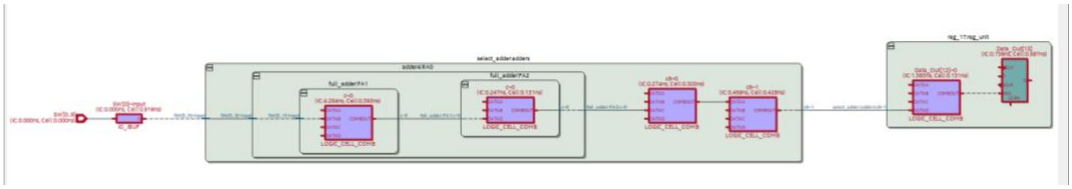

Figure 9 CSA

The above diagram represents the slowest path through the carry select adder. As you can see, we have the 10-bit input coming from the switches going as input to the carry select adder module. Inside the carry select adder module we have two 4-bit full adders that have different c_in bits as inputs (0 and 1). The CSA simultaneously then calculates two different sums, one with c_in equals 0, and one with c_in equals 1. Finally, there is a 2:1

MUX, selecting between these two outputs and then sending it to the Data_out of the register unit.

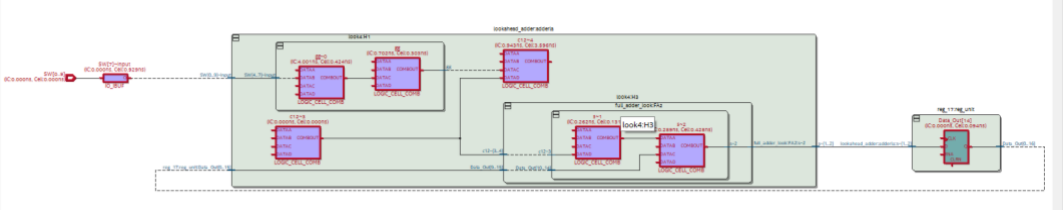

Figure 10 CLA

The above diagram represents the slowest path through the carry lookahead adder. As you can see, we have the 10-bit input coming from the switches going as input to the carry lookahead adder module. Inside the CLA module, we can clearly see that gg is being generated as well as c12. There are also two full lookahead adders used to calculate the final 16-bit sum which is then stored inside the register unit.

Post Lab Questions:

Q1) 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.

When designing a CSA, we must consider two things: the size of the CRA per unit and the total number of CSA units used. We are currently using 4 CSA units, each with two 4-bit CLA units (one with cin = 0, and cin = 1). Note that the execution time depends on the individual gate delays that exist within each CLA unit (the CLA unit computes the sum parallelly using combinational logic), MUXes, as well as the propagation time it takes to carry information between one CSA unit to the next. In retrospect, using a 4x4 hierarchy may not be the most ideal. The most ideal CSA unit will make a tradeoff between the number of MUXes between that it uses (to propagate information between each CSA unit) as well as the number of CRA

units we use in each CSA unit. We make this trade off by learning the propagation time for an n-bit CRA in each CSA unit as well as the time delay it takes for each mux in addition to any additional logic we use in the circuit. To test this out, we could run the timing analyzer to determine the delays and then choose the most ideal CSA design (1x16, 2x8, 8x2).

Q2) 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?

Our data plots are as follows:

Carry Ripple Adder	
LUT	78
DSP	0
Memory (BRAM)	0
Flip-Flop	20
Frequency	64.47 MHz
Static Power	89.97 mW
Dynamic Power	1.39 mW
Total Power	105.11 mW

Figure 11 Table 1

Carry Lookahead Adder	
LUT	91
DSP	0
Memory (BRAM)	0
Flip-Flop	20
Frequency	66.57 MHz
Static Power	89.97 mW
Dynamic Power	1.51 mW
Total Power	105.29 mW

Figure 12 Table 2

Carry Select Adder	
LUT	82
DSP	0
Memory (BRAM)	0
Flip-Flop	20
Frequency	67.8 MHz
Static Power	89.97 mW
Dynamic Power	1.58 mW
Total Power	105.34 mW

Figure 13 Table 3

As we can see the LUT is the highest for Carry Lookahead Adder and the least for Carry ripple adder. This clearly shows that the amount of combinational logic used in CLA was comparatively more than CRA and CSA. This also makes sense, because for CRA it was just 16 full adders chained together and used the minimal amount of combinational logic. On the other hand, CLA had multiple Boolean expressions for Ps and Gs along with Pg and Gg signals.

The maximum operating frequencies simply implies the performance of the circuit. As we can we ripple carry adder has slowest frequency while the carry lookahead adder has the highest operating frequency. This makes sense as CRA has the time complexity of O(n) and CLA has the time complexity of O(logn). Since, O(logn) is faster than O(n), CRA has a slower frequency. Note that CSA has a frequency in between, which makes sense as it has a running time of $O(n^{1/2})$.

The power consumption is directly proportional to the number of chips and logic gates used. Since CSA has the greatest number of adders and MUXes, it consumes the most power. But as we can we all the adders consume nearly the same amount of power. Ripple carry adder consumes the least power has it has the least amount of combinational logic in it.

Theoretically, CLA should have consumed the most power as it has the greatest amount of combinational logic. But this is different from the actual power consumed that is nearly the same for all three. The frequency and LUT parameters match up perfectly as per the theoretical value.

Conclusion:

We had a few bugs while coding the expressions for the Pg and Gg in the 4-bit carry look ahead adder. It was due to misplaced always_comb blocks. Similarly, we had a few compilation bugs while coding the MUX in the carry select adder, we had put if statements within an always_comb statement. To avoid such bugs in future we started unit testing each module before moving on to the next one by making it the top-level entity and running ModelSim on it.

We found the lab manual to be a great resource and was extremely helpful. The only addition which we feel should added is maybe a better explanation of the code that's already provided to us. The explanation of the three adders was very thorough in the manual.

Conclusively, we designed and implemented three 16-bit full adders on the FPGA. We saw that carry look ahead adder provides the best performance with time complexity of O(logn), followed by the carry select adder which has a time complexity of $O(n^{1/2})$, and then comes the carry ripple adder with time complexity of O(n). If we consider the space complexity, ripple carry adder takes the least space, followed by carry select adder and then finally carry look ahead adder. In terms of logic complexity, carry ripple adder is the simplest to implement. It is followed by carry select adder and finally carry look ahead adder. In this lab, we considered the different tradeoffs between the 3 adders and how each adder might be better than the other according to the use case.