ECE385

Fall 2021

Experiment #2

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

Haozhe Si

haozhes3

Section ABD

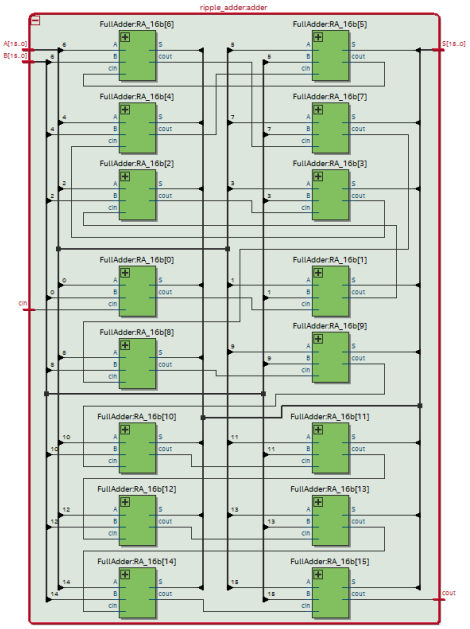
1. Introduction

In this lab, we designed and implemented three types of 16-bit adder: Ripple Carry Adder (RCA), Carry Lookahead Adder (CLA) and Carry Select Adder (CSA). Since the MAX 10 10M50DAF484C7G FPGA board only has 10 switches for input, we can only input 10 bits each time as input and the adder will extend it into 16 bits then perform as an accumulator—the inputs will accumulate once as we press the Run button and if we press the Reset button, the result will be cleared. The adders can handle 16 bits addition operation and if overflow happens and the output will be 17 bits. The lowest 8 bits of the input will be displayed on digit panels, the remaining 2 bits will be indicated by two LEDs. The lowest 16 bits of the result will be displayed on the digit panels and the highest bit will be shown by the LED.

2. Adders

a) Ripple Carry Adder (RCA)

The ripple carry adder is constructed by 16 1-bit full adders cascading in series. Each full adder will take the responsibility of calculating one bit of the 16-bit input. The full adder will take in A, B, and c_in as inputs, where A and B are the digits that we want to add and c_in is the carry in bit come from previous full adder; it has two outputs S and c_out, where S is the result of XORing A, B and c_in as the result of the addition and the c_out will be the majority of A, B and c_in, representing the carry of the addition which will go into the next full adder. The c_in for the first full adder will be 0. The carry will propagate through the series of full adders. Once it reaches the last full adder, we can have the valid result by reading the S of all the full adders and the c_out of the last full adder.



Block Diagram for the 16-bit RCA

b) Carry Lookahead Adder (CLA)

The 16-bit carry lookahead adder is constructed by four 4-bit carry lookahead adders and a carry lookahead units and a 4-bit carry lookahead adder consists of 4 full adders and a carry lookahead unit. The full adders we used in CLA is modified in the way that instead of calculating c_out, we calculate and output the P=A XOR B and G=A AND B signals, which stand for Propagation and Generation. All the P and G signals from a modified full adder in the 4-bit CLA will go into a CLU, where the following c_in would be calculated by all the previous P and G. For each 4-bit CLA, a PP and a GG signal will also be calculated by the CLU, and the four PP and GG signals from the four 4-bit CLAs will be fed into another CLU to calculated the final c_out for the 16-bit CLA.

P and G Signals: P and G signals are generated by each full adder according to their input A and B, deciding whether to propagate the carry in or generate a new carry: if both A and B are 1s (AND), Generate will be 1, meaning the adder will generate a carry; if either A or B is 1 but not both (XOR), Propagate will be 1, meaning the adder will propagate its carry. Given the P, G, and c_in, the carry can be calculated in one shot if we have wide enough chips. The c_out can be calculated by:

$$c\_out = (P \text{ AND } c\_in) \text{ OR } G$$

Thus, all the c_in a 4-bit CLA can be calculated by:

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

Hierarchical Structure: Although the CLA can calculate all the carry immediately as it sees the input theoretically, as the number of bits the input becomes, the number of gates increases in polynomial times. This becomes unpractical in implementation, for instance, we need about 800 gates for the 16-bit CLA. Therefore, instead of using the linear implementation, we used the 4x4 CLA hierarchical structure, which can reduce the number of gates needed.

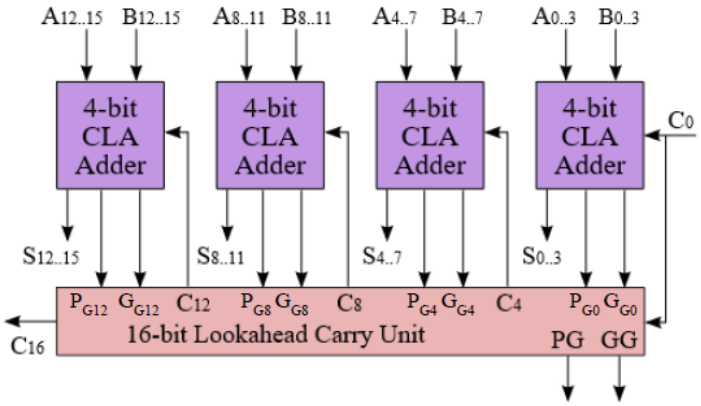
Since we are using four 4-bit CLAs, we need to generate PG and GG signals, which integrate the Propagation and Generation information, for the outmost combination. In each CLU of the 4-bit CLAs, PG signal represents the c_in of this module will be propagated to the next one and GG signal represents the module will generate a carry. They are given by:

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

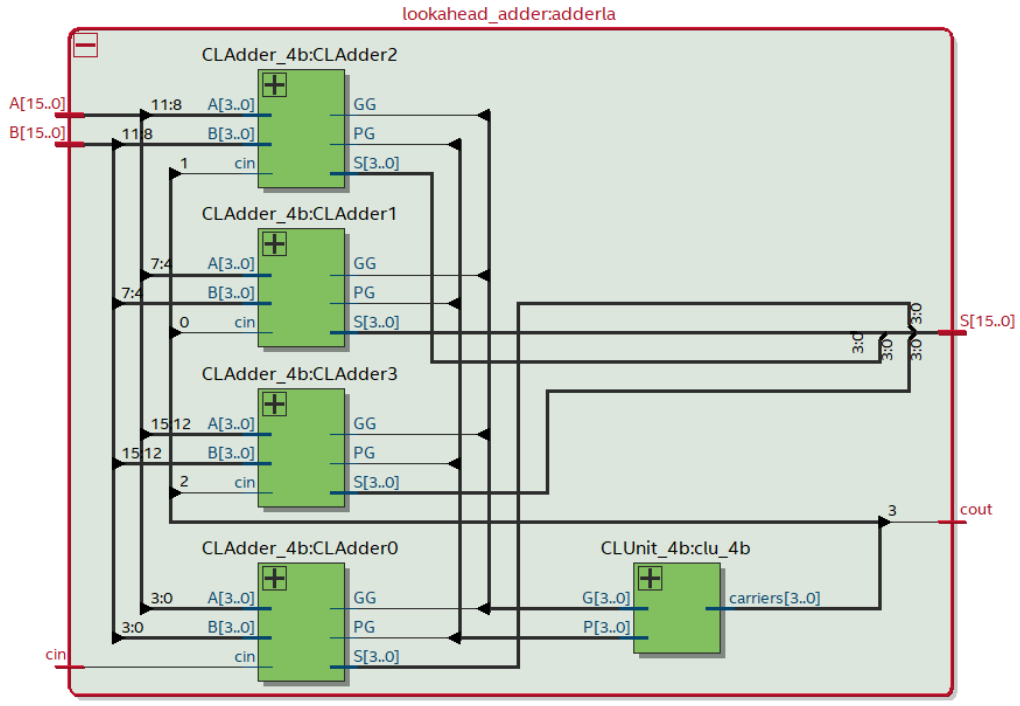
The PG and GG signals from the 4-bit CLAs will than feed into a CLU to calculate the c_in and c_out of each CLAs:

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

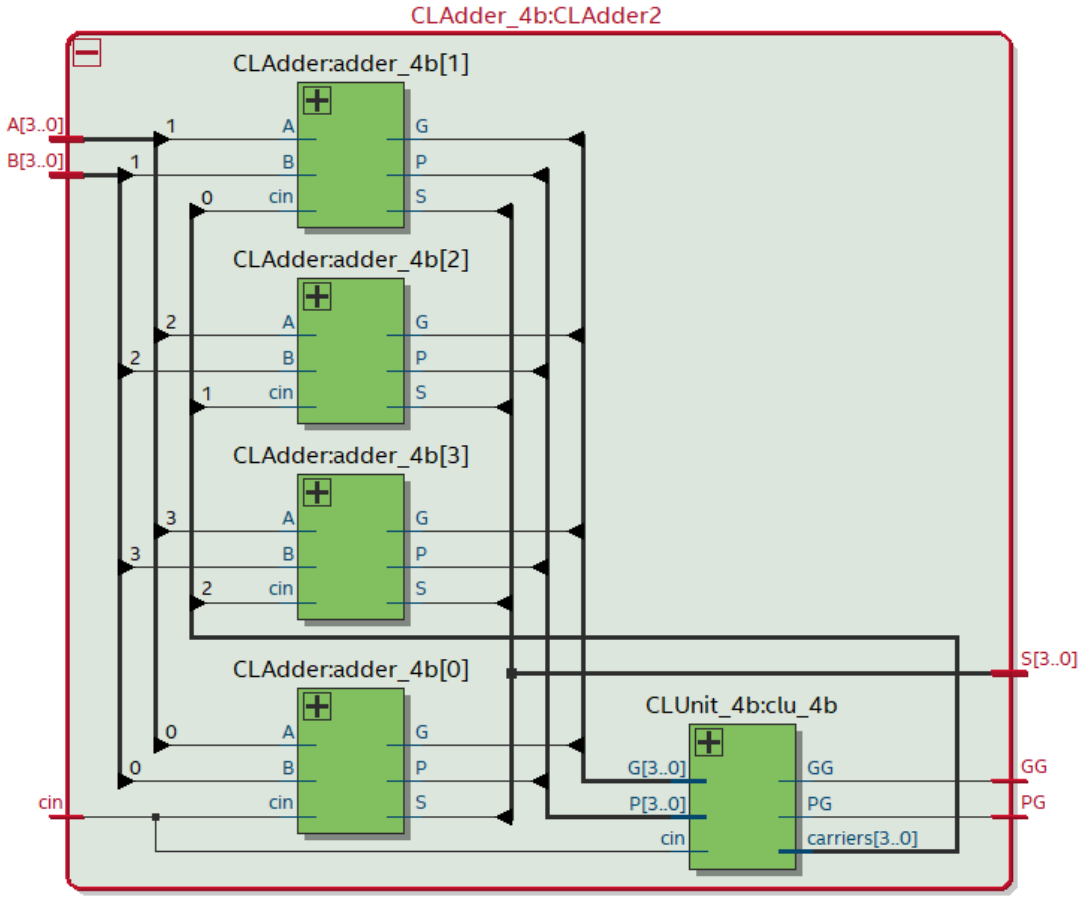
As mentioned above, each 4-bit CLA consists of four full adders that can give out P and G signals, and a CLU that can integrate the P and G signal while calculation carries. Similarly, the 16-bit CLA consists of four 4-bit CLAs that can give out PG and GG signals, and a CLU performing exactly the same task as the CLU in 4-bit CLAs does. Thus, we can reuse the CLU module implemented before. The overall structure is as follows:



Block Diagram:



Block Diagram for the Hierarchical Structure



Block Diagram for the 4-bit CLA

c) Carry Select Adder (CSA)

The 16-bit carry select adder is constructed by four 4-bit CSA modules, which each module consists two 4-bit RCAs and a MUX of output selection and a logical circuit for carryout selection. The high-level idea for CSA is that the two RCAs in the 4-bit CSA modules will calculate S and c_out signals with the input A, B, and c_in while c_in will be 0 and 1 for the two RCAs respectively. Two results will be calculated parallelly and given the c_in, the MUX and the c_out selection circuit will decide which result and carry to output from the CSA module. The carry will be fed into the following CSA module as the c_in input.

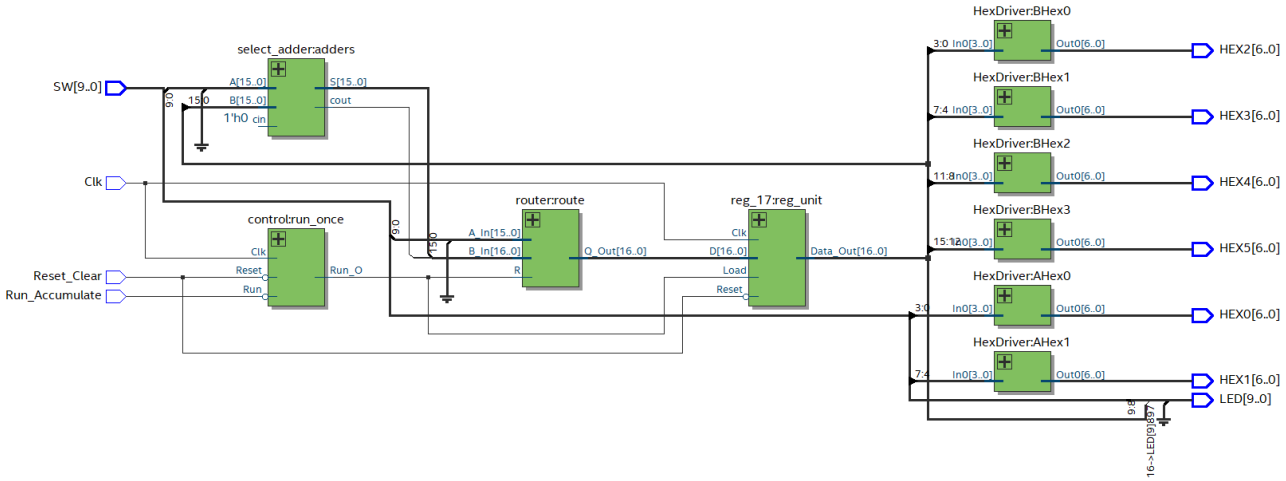
Parallel Computation: The CSA module can calculate the c_in-dependent S and c_out in advance and select the corresponding outputs depend on the actual input c_in to the module. To be specific, the two results of summation are calculated in parallel with different c_in values, which means all possible results are calculated once the inputs are fed into the CSA. All we need to do is to select the correct combination out according to the c_in signals. The four 4-bit CSA modules will be connected in series, and the c_out of one module

will be the c_in to the next. Once we fed in the first c_in, the correct answer will be select sequentially as the carry propagate through the modules. In actual implementation, since the first c_in is deterministic to be 0, we can replace the first CSA module to be a 4-bit RCA. In addition, the logical circuit for selecting c_out will be:

$$c\_out = (c\_in \text{ AND } c\_out\_1) \text{ OR } c\_out\_0$$

where c_in is the input to the module; c_out_1 and c_out_0 signals are the c_out from the RCAs assuming the c_in to be 1 and 0 respectively. Thus, once the summations are calculated together, the carry can be propagate to the next module by passing two gates, which makes the selection fast.

Block Diagram:



Block Diagram for the 4-bit CSA

3. SyetemVerilog Module Description

Module: FullAdder
Inputs: A, B, cin
Outputs: S, cout
Description: This module will perform combinational logic. It calculates *S* = *A* XOR *B* XOR *cin* and *cout* = the Majority of *A, B,* and *cin*.
Purpose: This module is used to performs one-bit full add and output the summation and carryout.

Module: CLAdder
Inputs: A, B, cin
Outputs: S, P, G
Description: This module will perform combinational logic. It calculates *S* = *A* XOR *B* XOR *cin, P* = *A* XOR *B,* and *G* = *A* AND *B*.
Purpose: This module is used to perform one-bit full add for CLA and output the summation, Propagation signal and Generation signal.

Module: CLUnit_4b
Inputs: [3:0] P, [3:0] G, cin
Outputs: [3:0] carriers, PG, GG
Description: This module will perform combinational logic. It computes four carryout and the *PG* and *GG* a given the initial *cin* and the P and G signals.
Purpose: This module is the CLU used in CSA. It can calculate the carries, *PG* and *GG* signals by taking in the initial *cin*, P and G from the four modified full adders inside 4-bit CSA, or it can calculate the carries and the final *cout* by taking in the initial *cin*, *PG* and *GG* signals from the four 4-bit CSAs.

Module: CLAdder_4b
Inputs: [3:0] A, [3:0] B, cin
Outputs: [3:0] S, PG, GG
Description: The module instantiates a CLU and four modified full adders to build the 4-bit CLA. It takes in two 4-bit inputs *A* and *B* and a carry *cin*. It will calculate the summation of the two inputs given the carry. It will also generate the *PG* and *GG* signals.
Purpose: The module is the building block for the 16-bit CLA. The four 1-bit modified full adders will calculate 1 bit *S*, *P* and *G*, and the CLU will combine the four *P* and *G* signals and outputs the *PG* and *GG* for the module.

Module: RAdder_4b
Inputs: [3:0] A, [3:0] B, cin
Outputs: [3:0] S, cout
Description: This module will perform combinational logic. It calculates the 4-bit summation between *A* and *B* given the carry *cin* by using four full adders to implement a 4-bit ripple carry adder. It will calculate the four-bit *S* and a carry out *cout*.
Purpose: The module is the building block for the 16-bit CSA that performs 4-bit addition.

Module: CSAdder_4b
Inputs: [3:0] A, [3:0] B, cin
Outputs: [3:0] S, cout
Description: This module will perform combinational logic. It instantiated two 4-bit RCAs to calculates the 4-bit summation between *A* and *B* given the *cin* being 1 and 0. It will then select the resulting *S* and *cout* according to the *cin*.
Purpose: The module is the building block for the 16-bit CSA that can perform 4-bit addition with parallel ripple adders and results selection.

Module: ripple_adder
Inputs: [15:0] A, [15:0] B, cin
Outputs: [15:0] S, cout
Description: This module will perform combinational logic. It calculates the 16-

bit summation between *A* and *B* given the carry *cin* by using 16 full adders to implement a 16-bit ripple carry adder. It will calculate the 16-bit *S* and a carry out *cout*.
Purpose: This is the 16-bit Ripple Carry Adder.

Module: lookahead_adder
Inputs: [15:0] A, [15:0] B, cin
Outputs: [15:0] S, cout
Description: This module will perform combinational logic. It calculates the 16-bit summation between *A* and *B* given the carry *cin*. The module instantiated four 4-bit CLA modules and a CLU module. Each of the four 4-bit CLAs will output 4 bits of *S* and one bit of *PG* and *GG* signals. The CLU will take in the *PG* and *GG* signals to calculate the carries for the cascaded 4-bit CLAs as well as the final *cout.*
Purpose: This is the 16-bit Carry Lookahead Adder.

Module: select_adder
Inputs: [15:0] A, [15:0] B, cin
Outputs: [15:0] S, cout
Description: This module will perform combinational logic. It calculates the 16-bit summation between *A* and *B* given the carry *cin*. The module instantiated three 4-bit CSA modules and a 4-bit RCA. Each of the three 4-bit CSAs and the RCA will output 4 bits of *S* and one bit of carryout signals. The carryout signals will propagate to next 4-bit CSA, selecting the output *S* and select the carryout for the next 4-bit CSA. The last 4-bit CSA will give the final *cout*
Purpose: This is the 16-bit Carry Select Adder.

Module: control
Inputs: Clk, Reset, Run
Outputs: Run_o
Description: This is a positive-edge triggered control module that has asynchronous *Reset* and synchronous *Run*. When *Run* is high, the control module will set output *Run_o* to high for a clock cycle when on the positive edge of *Clk*.
Purpose: The module generates the load signal to high for one clock cycle for the register to load result for addition.

Module: router
Inputs: R, [15:0] A_In, [16:0] B_In
Outputs: [16:0] Q_Out
Description: The router will route *A_In* to *Q_Out* with the most significant bit extension if *R* is 0, and will route *B_In* to *Q_Out* if *R* is 1.
Purpose: The module is the router to decide which data to be displayed on the HEX digit panel.

Module: reg_17
Inputs: [17:0] D, Clk, Reset, Load
Outputs: [17:0] Data_out
Description: This is a positive-edge triggered 16-bit register with asynchronous *Reset* and synchronous *Load*. When *Load* is high, data is loaded from *D* into the register on the positive edge of *Clk*.
Purpose: This module is used to create the registers that store operands *A* and *B* in the adder circuit.

Module: HexDriver
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: The module will output the code for seven-segment display depending on the input 4-bit data.
Purpose: This module is used to translate the binary data into the code for hex display on the board.

4. 16-bit Adder Comparison

Area: In terms of area, RCA used 16 full adders to implement the 16-bit addition. Given each full adder will use 5 gates, the RCA used 80 gates in all. The CLA also used 16 full adders. However, the modified full adder only needs 3 gates. Meanwhile, the CLA also have 5 CLU modules, which each takes 16 gates. Therefore, the CLA will need 128 gates. The CSA will use 28 full adders with 5 gates each, makes it already consuming 140 gates. In addition, it also has 12 2-1 MUXes and 6 gates for output selection. Therefore, RCA takes the less areas, following by the CLA, and the CSA takes the most areas.

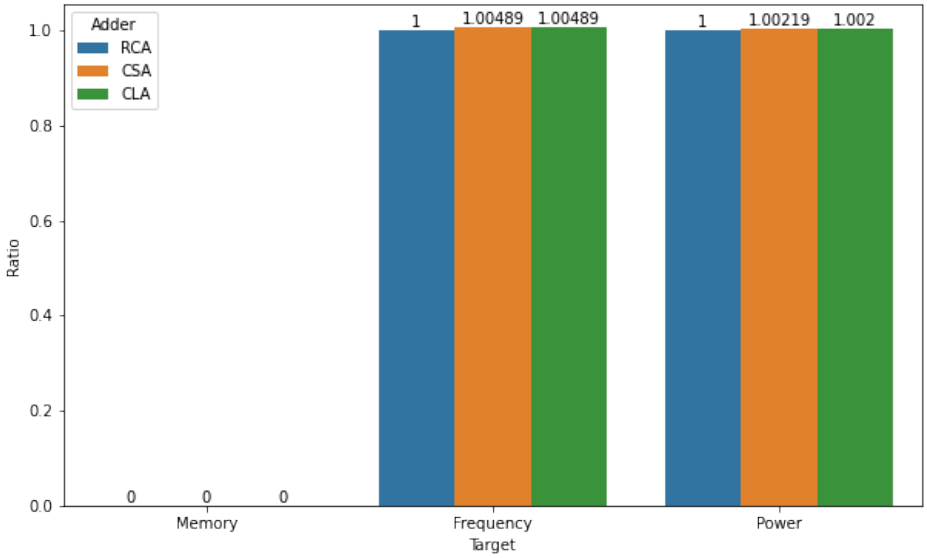
Complexity: The RCA is the most straightforward implementation since it only has 16 full adders and that is the least number of adders that needs to use. The CLA, by contrast, is the most complex one, and that is majorly because the CLU. CLU needs to calculate carries, *PG* and *GG* base on all the previous inputs, which involves a large amount of logic calculations, making it the most complex. The CSA is slightly more complex than the RCA since it just utilizes more adders and having them performing in parallel, and uses some logic circuits to select the outputs.

Performance: The RCA is implemented by cascading 16 full adders, and it would not finish until the carry propagate from the lowest bit all the way to the last bit. Therefore, given an N-bit adder, the complexity of RCA will be $O(N)$. For the CLA using the hierarchical structure, since all the carries can be calculated in one shot inside each submodule, we only need to wait for the carries propagate through the submodules. Such hierarchical will lead CLA to have $O(log\ N)$ complexity. Every summation inside the CSA can be done in parallel,

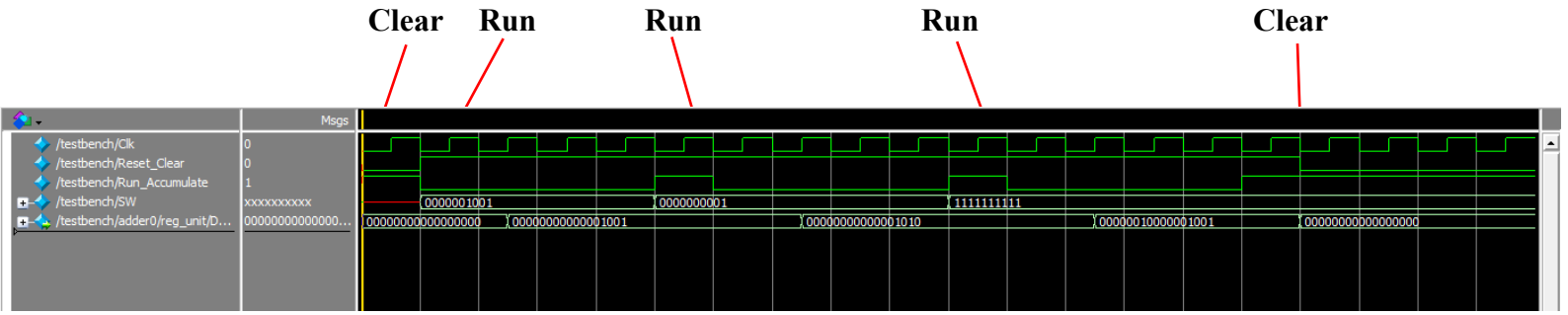
therefore the time complexity for calculation is $O(1)$. However, we cannot ignore the delay of the MUXes selecting the output. Thus, the time complexity for CSA is in O(MUX delay).

5. Adder Performance

	Ripple-Carry	Carry-Select	Carry -Lookahead
BRAM	0	0	0
Frequency	64.47MHz	67.8MHz	67.8MHz
Total Power	105.11mW	105.34mW	105.32mW



6. Simulation Trace



7. Post-Lab Questions

1) **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?**

It is not ideal. This is because although all the additions are done in parallel, which decreases the runtime, we still spend much time on waiting for the c_out to be propagate through the CSA modules. Since all the 4-bit summations are done together, they are waiting for the carry to be propagated to them. The final c_out will be generated after waiting for six gate delays (2 for each CSA module) and the last CSA module need to wait for four gate delays. We must notice that, although each summation is done in parallel, they still take time. Therefore, in order to exploit the use of time, we should not waste the time the later CSA modules waiting. We can design the structure such that latter CSA modules will calculate the addition for more bits. The ideal case is that the summations are done right before the c_out from the previous CSA module is selected and output. Then, no module is waiting and we can effectively reduce runtime for the overall module. To implement that, we need to know the gate delay for the AND gate and OR gate the module used to select c_out. We also need to figure out the time RCA needs for calculating addition with different number of bits. With the two kinds of time information, we will add corresponding number of bits of addition to each following CSA modules and compensate the time for waiting. So that, we can design a hierarchy so that the wasted time can be minimized.

2) **Design Analysis**

RCA:

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

CLA:

LUT	86
DSP	0
Memory (BRAM)	0
Flip-Flop	20
Frequency	67.8MHz
Static Power	89.97mW
Dynamic Power	1.53mW
Total Power	105.32mW

CSA:

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

The breakdown makes much sense: CLA and CSA are working at a higher frequency than RCA, which is the purpose of our design. As expected, the CSA consumes more power since it consists more logical computations in design. Meanwhile, the CLA has a larger LUT than CSA and RCA, which indicated that it is more complex. They all comply with our theoretical design.

8. Conclusion

a) Bug Log

One issue I met in the lab is when I implementing the testbench, I did not notice that the control signals *Run* and *Reset* are active low. This takes me some time to figure out and flip the control signals.

b) Comment on Lab Manual

Everything is good, but writing the description for the provided modules seems unnecessary in my opinion.

c) Conclusion

In this lab, in addition to further practice the skills and abilities of writing SystemVerilog, we learnt how to analysis our design, both from the data provided by the simulator and the from our design. Meanwhile, we also learnt how to write testbench. These skills will be helpful for us in further experiments.