ECE 385

Fall 2021

Experiment #4

Lab 4
Introduction to System Verilog, FPGA, CAD, and 16-bit Adders

Guanshujie Fu 3190110666

Xiaomin Qiu 3190110717

Lab Section D231 18:00 – 21:00

TA: Yuchuan Zhu

1. Introduction

In this experiment, we learned and used some basic knowledge of System Verilog to implement a serial logic processor and three adders with different design. Also, we acquire some basic skills to operate on Quartus Prime and FPGA. We extend the 4-bit serial logic processor to 8-bit and design Carry Ripple Adder, Carry Lookahead Adder and Carry Select Adder. We also compared the performance of different adder. Details will be presented later in other sections.

a. *High level function of Carry Ripple Adder:*

The Carry Ripple Adder utilizes a series of full adder. The function of full adder is shown below.

$$Sum = X \oplus Y \oplus Cin$$

$$Cout = (X \cdot Y) + (X \cdot Cin) + (Y \cdot Cin)$$

b. *High level function of Carry Lookahead Adder:*

The Carry Lookahead Adder utilizes *generated(G)* and *propagated(P)* to make the computation faster. The function is shown below.

$$G = A \cdot B$$

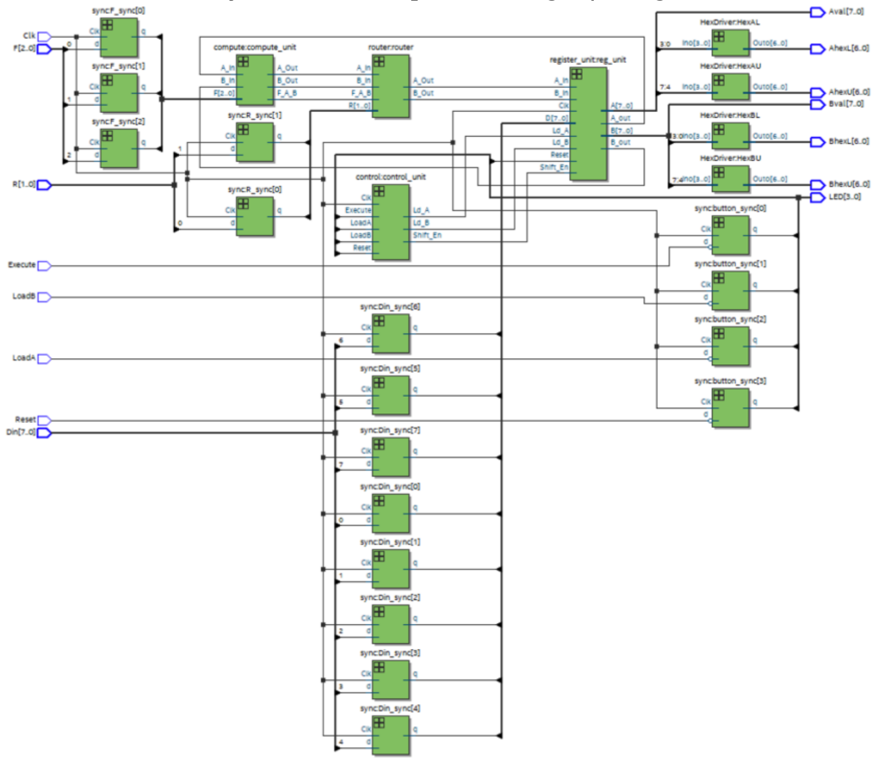
$$P = A \oplus B$$

c. *High level function of Carry Select Adder:*

The Carry Select Adder utilizes mux to speed up the computation while use more space. The basic computation uses full adder which is the same as CRA. The function has been shown above.

2. Part 1 - Serial Logic Processor

a. *Include a block diagram can be adapted from lab manual or the top-level schematic generated by the RTL viewer. Please only include the top-level design if using the RTL viewer.*



b. *Include a short description should include what was done with the provided code to extend it from 4 bits to 8 bits*

There are mainly two ways to extend the 4-bit code to 8-bit. We will discuss both two methods that can be applied to realize it.

i. ***The first method is direct modification to the register***.

Firstly, we change the bits of register from 4-bit to 8-bit by changing the variables *D*, *A*, *B and Data_Out* from [3:0] to [7:0].

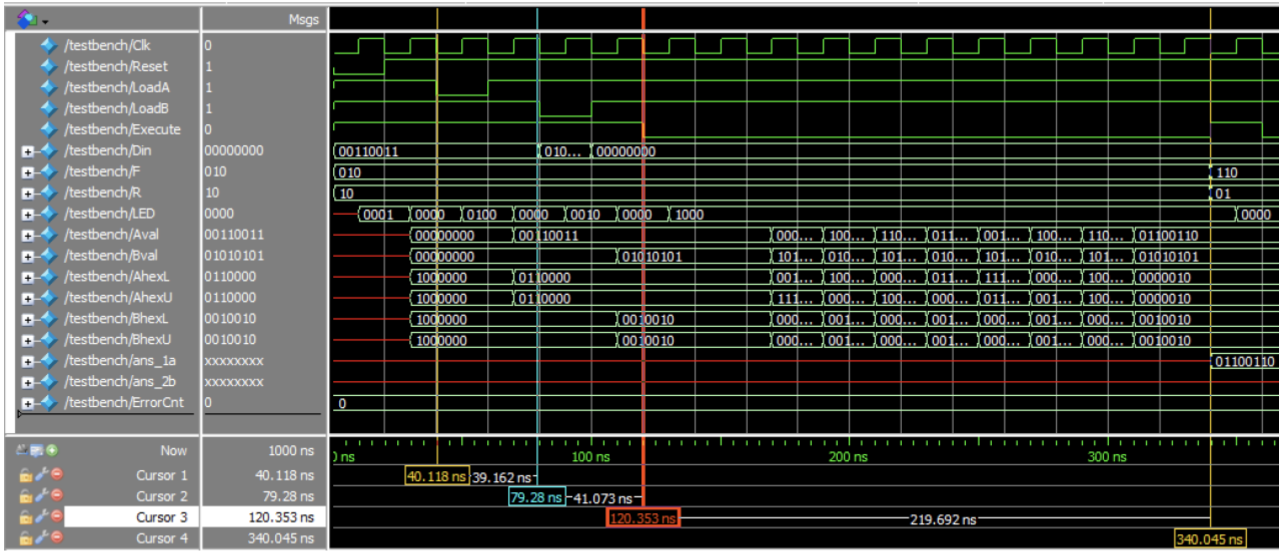
Secondly, we modify the control unit in control.sv to extend states from 6 to 10 by modifying the variable from *"enum logic [2:0] {A, B, C, D, E, F}"* to *"enum logic [3:0] {A, B, C, D, E, F, G, H, I, J}"*.

Finally, we change the number of Hexdriver needed. Two more drivers are needed and hence we connect two Hexdrivers *HexAU* and *HexBU*.

ii. ***The second method utilize two 4-bit registers to realize 8-bit register***.

The only difference between first and second method falls on how we get 8-bit register. Instead of changing variables from *[3:0]* to *[7:0]*, we utilize two 4-bit registers by connecting *A[3:0] and B[3:0]* to the first register and *A[7:4] and B[7:4]* to the second register.

c. *Include a simulation of the processor that has notes that give information such as what operation is being performed, where the result was stored, etc.*



Simulation Wave Form

Notes: As we can see from the simulation wave shown above, *F* is set to *010*, which should perform *XOR* function. *R* is set to 10, which will store the calculated result in register *A*. At 40ns and 80ns, the *LOADA* and *LOADB* are flipped. At 120ns, the *EXECUTE* is flipped. We can indicate that the correct result is calculated at 340ns, which is *00110011 XOR 01010101 = 01100110*. The result is stored at register *A* correctly.

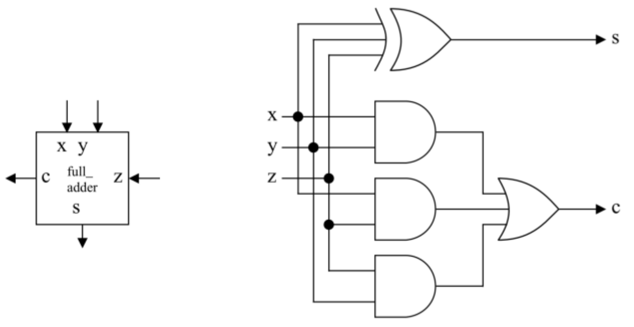
3. Part 2 - Adders

a. Ripple Carry Adder

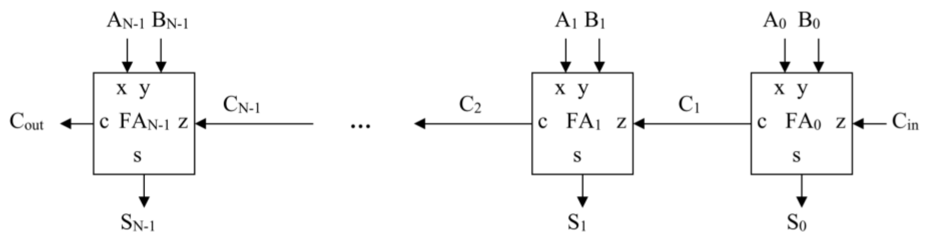
i. *Written description of the architecture of the adder*

CRA is constructed using N full adders. A full-adder is a single-bit version of the binary adder, where three binary bits (A, B and Cin) are inputted through a set of logic gates to produce a single-bit sum (S) and a single-bit carry-out (Cout). The N fulladders are then linked together in series through the carry bits, forming an N-bit binary adder. When the binary inputs are provided, the full-adder of the least significant bit (LSB) will produce a sum (S0) and a carry-out (C1). The carry-out is fed to the carry-in of the second full-adder, which then produces a second sum (S1) and a second carry-out (C2). The process ripples through all N bits of the adder as shown in Figure 3 and settles when the full-adder of the most significant bit (MSB) outputs its sum (SN-1) and carry-out (Cout).

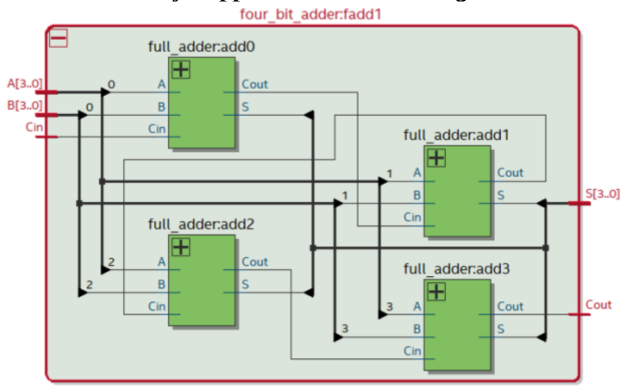
ii. *Block diagram*

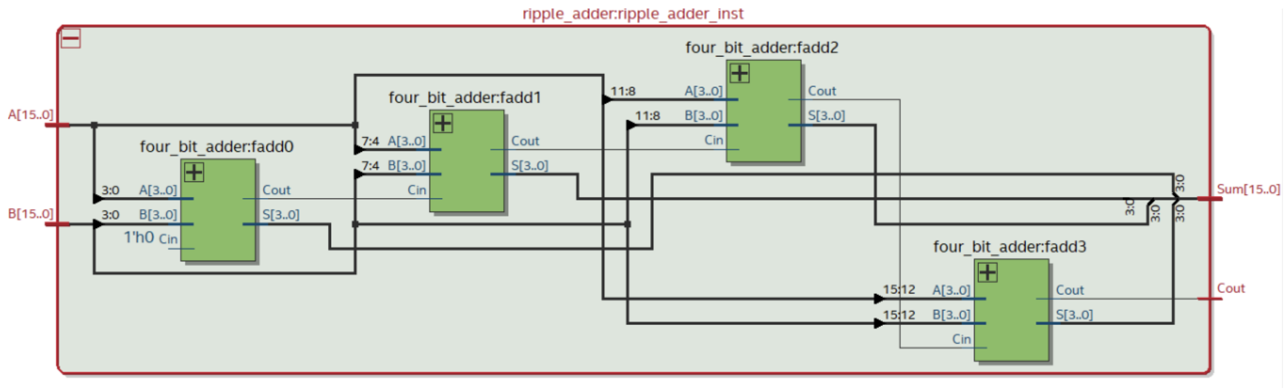


Full-Adder Block Diagram

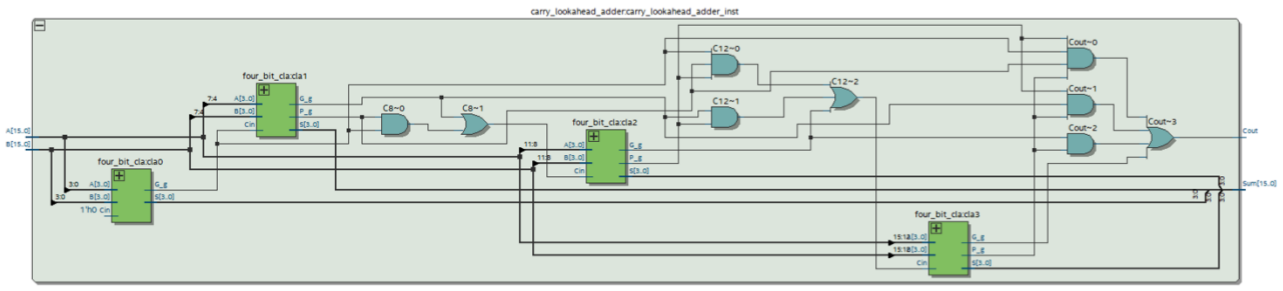


N-bit Carry-Ripple Adder Block Diagram

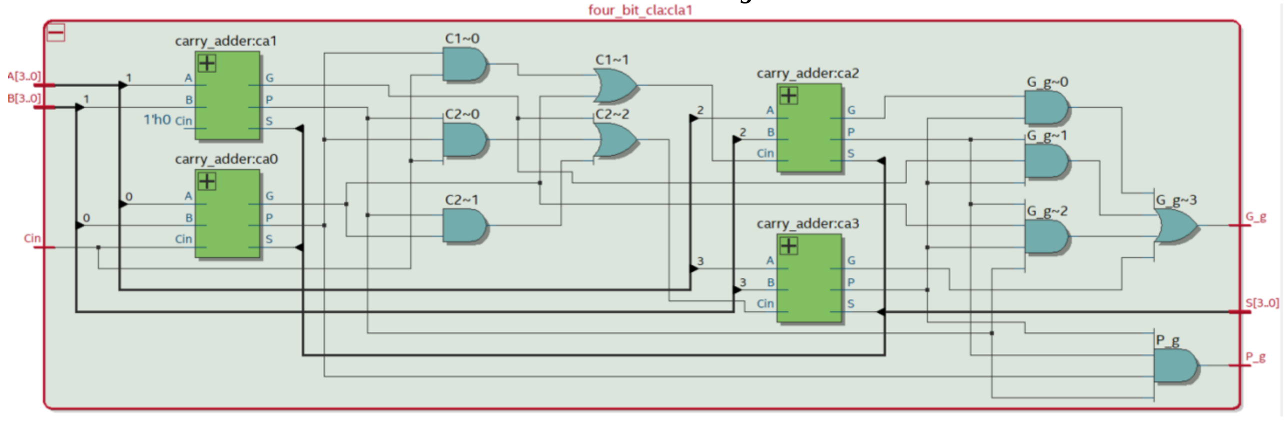




16-bit CRA Circuit Diagram



16-bit CLA Circuit Diagram



4-bit CLA Circuit Diagram

b. Carry Lookahead Adder

i. Written description of the architecture of the adder

1. Describe how the P and G logic are used

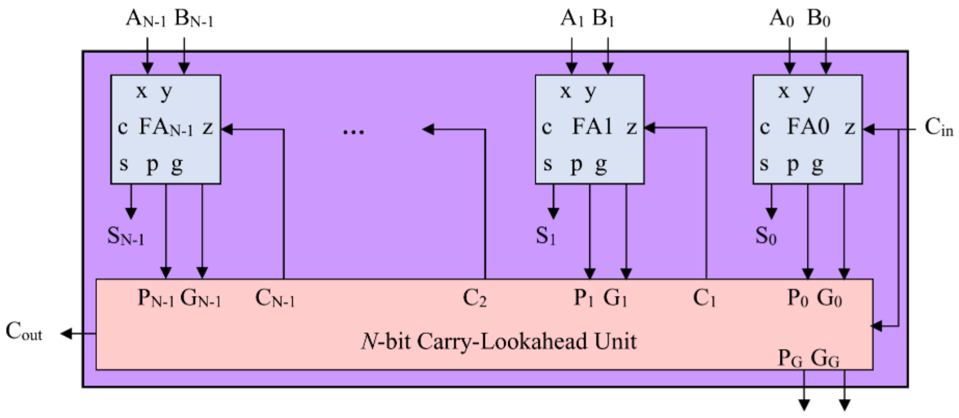
Carry-Lookahead Adder (CLA) uses the concept of generating (G) and propagating (P) logic. Every bit of the CLA makes predictions using its immediate available inputs (A and B), and predicts what its carry-out would be for any value of its carry-in. A carry-out is generated (G) if and only if both available inputs (A and B) are 1, regardless of the carry-in. The equation is $G\,(A, B) = A \cdot B$. On the other hand, a carry-out has the possibility of being propagated (P) if either A or B is 1, which is written as $P\,(A, B) = A \oplus B$. With P and G defined, the Boolean expression for the carry-out C_{i+1} giving a potential C_i is then $C_{i+1} = G_i + (P_i \cdot C_i)$. C_{i+1} can be expressed in terms of C_i which in turn can be expressed in terms of C_{i-1}. However, if C_{i+1} still depends on C_i, it will behave like a ripple adder without giving any gain in speed. Therefore, to avoid the slow rippling of the carry bits, the expression of C_{i+1} should be expanded and computed directly from P_{iS}, G_{iS}.

2. *Describe how you partitioned the adder (Did you chain together 4 4-bit CLA blocks to make a 16-bit CLA block?)*

   We use four full adders to construct a 4-bit CLA. Through CLU, we chain together four 4-bit CLA blocks to make a 16-bit CLA block. Each 4-bit CLA generates a group propagate PG and a group generate GG which are used to compute values of Cin for all CLA units. And Cin would be sent back to the CLU.

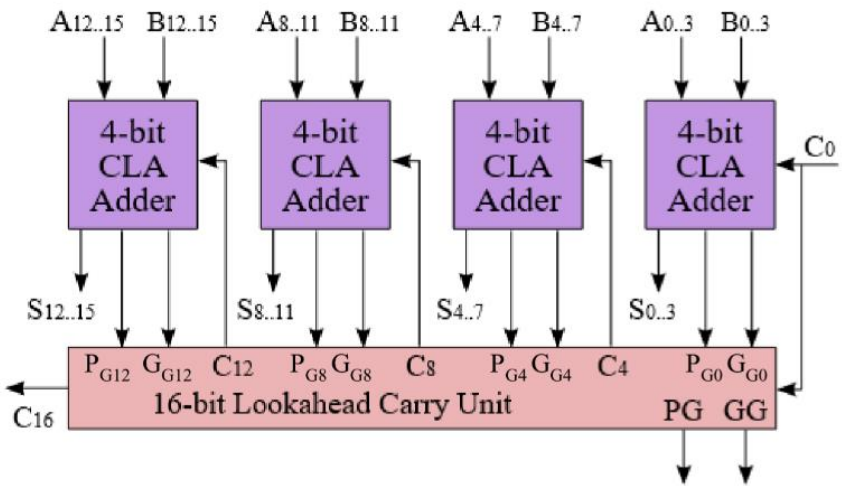
ii. Block diagram

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



Single Carry-Lookahead Adder Block Diagram

2. Block diagram of how each CLA was chained together



4x4-bit Hierarchical Carry-Lookahead Adder Block Diagram

c. Carry Select Adder

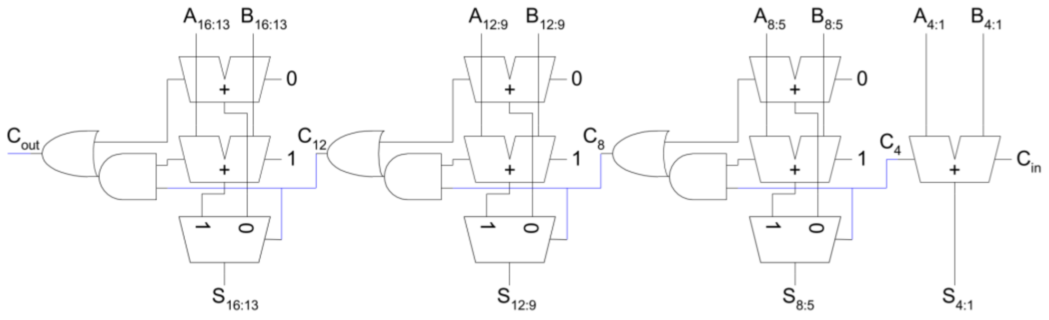
i. Written description of the architecture of the adder

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

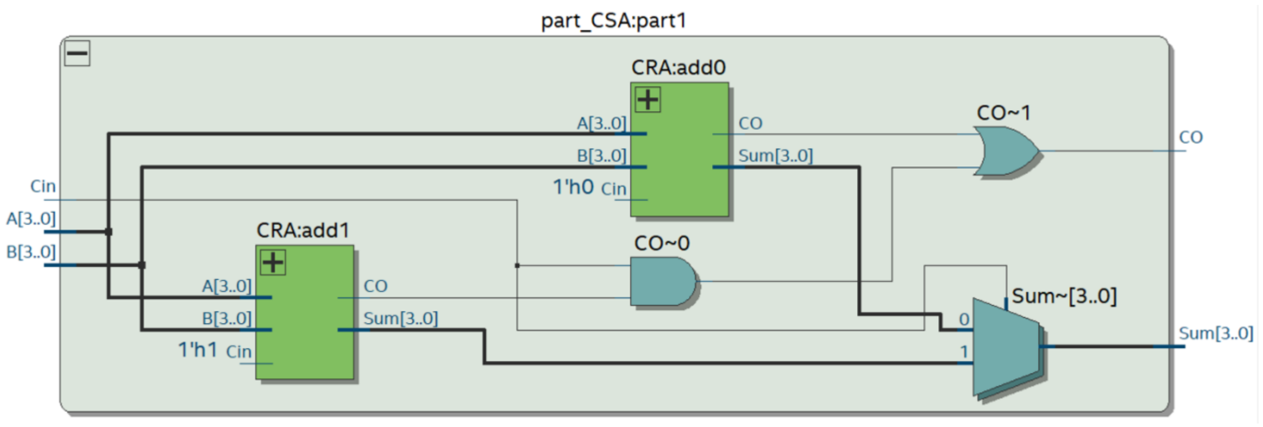
We design a 16-bit CSA with 4x4-bit hierarchical structure, which consists of 28 full adders which is divided into 4 groups (the first unit has available Cin, which only require one Full Adder). For each group of 4-bit inputs, there are two CRAs and one MUX.

To compute the multiple sums, we use two CRAs to calculate two versions of the results in parallel, one with carry-in bit assumed to be 0 and the other to be 1. Then both possible outcomes are pre-computed. Once the carry-in bit calculated by previous logic arrives, the corresponding sum and carry-out is selected by MUX and to be delivered to the next stage.

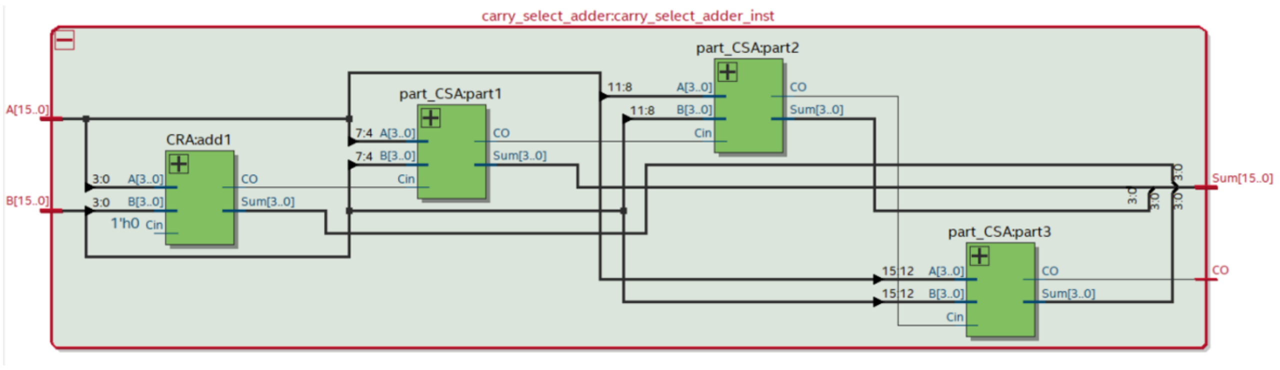
ii. Block Diagram of the whole CSA circuit containing adders, multiplexers, and glue logic.



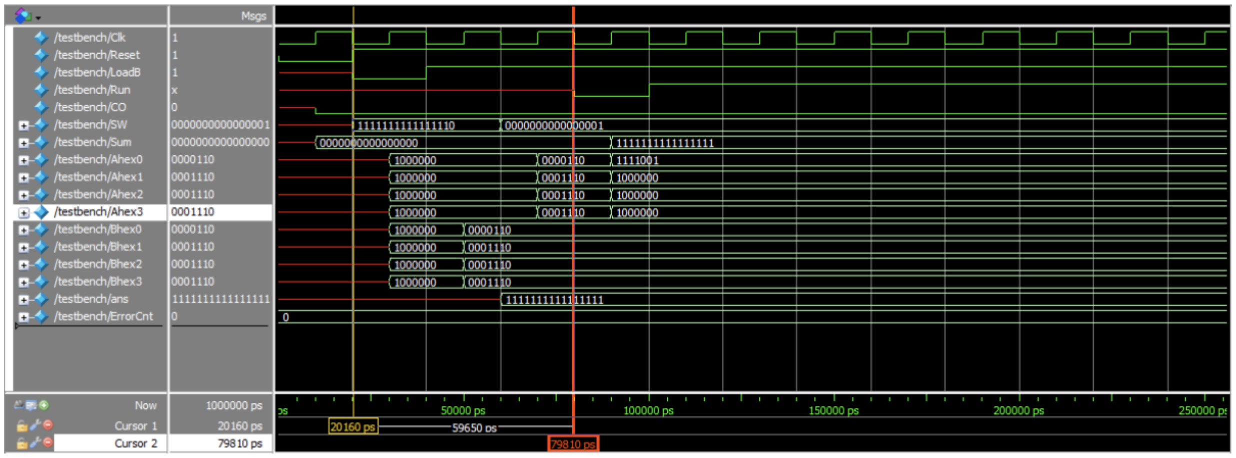
16-bit Carry-Select Adder Block Diagram



4-bit Carry-Select Adder Circuit



16-bit CLA composed of 3 4-bit CLA and 1 CRA



Simulation Waveform of Adders

d. Written Description of all .sv Modules

i. Ripple Carry Adder:

Module: ripple_adder.sv
Inputs: logic [15:0] A, logic[15:0] B
Outputs: logic [15:0] Sum, logic Cout
Description: This is a 4x4 bits adder constructed with 4 four_bits_adders.
Purpose: Used to calculate A+B, two 16 bits numbers.

Module: four_bit_adder.sv
Inputs: logic [3:0] A, logic [3:0] B, Cin
Outputs: logic [3:0] S, logic Cout
Description: A 4bits adder constructed with four full_adders.
Purpose: Submodule of ripple_adder and CLA.

Module: full_adder.sv
Inputs: logic A, logic B, logic Cin,
Outputs: logic S, logic Cout
Description: A one bit adder constructed with some basic logic.
Purpose: Submodule of four_bit_adder.

ii. Carry Lookahead Adder:

Module: carry_lookahead_adder.sv
Inputs: logic [15:0] A, logic[15:0] B
Outputs: logic [15:0] Sum, logic Cout
Description: A 4*4 bits adder which constructed with four four bit cla and a Carry lookhead unit.
Purpose: Calculate A+B, two 16bits numbers, in a more efficient way.

Module: four_bit_cla.sv
Inputs: logic Cin, logic [3:0] A, logic [3:0] B
Outputs: logic [3:0] S, logic P_g, logic G_g
Description: A 4bits carry adder which construct with four carry adders and cla unit.
Purpose: Submodule of carry_lookahead_adder.sv Calculate A+B, two 4 bits numbers, in a more efficient way with P,G.

Module: carry adder.sv
Inputs: logic Cin, logic A, logic B
Outputs: logic S, logic P, logic G
Description: A carry adder which constructed with & ^logic.
Purpose: Submodule of four_bit_cla, calculate A+B and P, G.

iii. Carry Select Adder:

Module: carry_select_adder.sv
Inputs: logic [15:0] A, logic[15:0] B
Outputs: logic [15:0] Sum, logic Cout
Description: A 4*4 bits adder which constructed with some multiplexers, CRAs and some logic gates.
Purpose: Calculate A+B, two 16bits numbers, in a more efficient way.

Module:part_CSA.sv
Inputs: logic [3:0] A, logic[3:0] B, logic Cin
Outputs: logic [3:0] Sum, logic Co
Description: A 4 bit adder which constructed with two 4 bit CRA to pre-calculate values and 1 multiplexer to choose the value that match the Cin.
Purpose: Submodule of carry_select_adder.sv, calculate A + B and the carry out.

iv. Others

Module: testbench_adder.sv
Inputs:
Outputs:
Description:
Purpose: To test your adder one in a time.

Module: lab4_adders_toplevel.sv
Inputs: logic Clk, logic Reset, logic LoadB, logic Run, logic [15:0] SW
Outputs: logic CO, logic[15:0] Sum, logic[6:0] Ahex0, logic[6:0] Ahex1, logic[6:0] Ahex2, logic[6:0] Ahex3, logic[6:0] Bhex0, logic[6:0] Bhex1, logic[6:0] Bhex2, logic[6:0] Bhex3
Description: This is the top-level entity.
Purpose: It connects an adder circuit to LEDs and buttons on the device. It also declares some registers on the inputs and outputs of the adder to help generate timing information (fmax) and multiplex the DE115's 16 switches onto the adder's 32 inputs.

Module: HexDriver.sv
Inputs: logic[3:0] In0
Outputs: logic [6:0] Out0
Description:
Purpose:

e. **Describe at a high level the area, complexity, and performance tradeoffs between the adders.**

For Area:

CRA has fewest gates which result in less area. CLA require more area than CRA because there are extra gates logic to compute P, G and C. CSA require almost twice of the area of CRA because it needs to calculate both Cin = 0 and 1 and then select from them.

For Complexity:

The logic of CRA is easy with fewer gates and complexity. CLA needs to compute P, G and C and thus has high complexity. CSA is the most complex because for each block it needs two CRA to compute two results and a MUX to select from them.

For Performance:

In CRA, every full-adder must wait for their lower-bit neighbor to produce a carry-out before it can correctly compute its sum and carry-out. This means that the propagation delay of the CRA increases with N. CLA could compute all the Cin with initial Cin and input value, in this way, the computation time of the CLA is much faster than that of the CRA, resulting in a higher operating frequency. CSA computes two sets of results ahead so the Cin can directly select value and has better performance than CRA.

f. **Document the performance of each adder by creating a graph as specified in Prelab part C (page 4.6 in the manual).**

The performance of each adder has been shown in Post Lab section below.

4. Answers to the Post lab Questions

a. *Compare the usage of LUT, Memory, and Flip-Flop of your bit-serial logic processor exercise in the IQT with your TTL design in Lab 3. Make an educated guess of the usage of these resources for TTL assuming the processor is extended to 8-bit. Which design is better, and why?*

	IQT	TTL
LUT	72	24
Memory (BRAM)	0	0
Flip-Flop	43	12

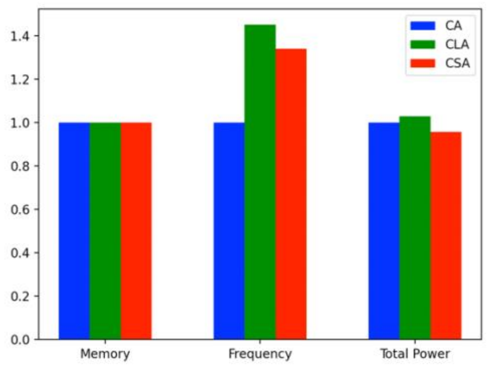
The TTL design is better than IQT for this experiment as it needs less LUTs and flip-flops. Because the TTL design is more straightforward and takes components to implement the circuit. For IQT, we utilize the System Verilog language to construct circuit and verify its function, which could cost more compared to direct circuit design as it use look-up table to construct function and requires much more flip-flops.

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 System Verilog circuit.*

	CRA	CLA	CSA
LUT	114	121	123
DSP	0	0	0
Memory (BRAM)	0	0	0
Flip-Flop	105	121	105
Frequency	62.78MHz	84.21MHz	91.08MHz
Static Power	98.50mW	98.48mW	98.51mW
Dynamic Power	0.00mW	0.00mW	0.00mW
Total Power	140.23mW	134.14mW	144.28mW

c. *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?*

The maximum operating frequency of the carry-lookahead adder is higher than the carry-ripple adder. The carry-select adder consumes more power than others because it does more calculations and consume more resources compared to other two adders. With more resource and power consumed, it could have better, and faster performance compared to CRA.



5. Conclusion

a. *Describe any bugs and countermeasures taken during this lab*

Bug1: The first bug occurs due to the '*sync*' used in our experiment which is needed to set the input signals such as *Reset*, *ClearA_LoadB*, *Clk*. As we did not correctly set the *'sync'* module in our project file, we failed to get the simulation waveform.

Bug2: The second bug occurs in our state machine design and was observed when we test on FPGA board. When a series of multiplications is performed, the register A is automatically cleared and does not hold the value calculated for each multiplication. The reason is that our state machine automatically returns to *S* state in which we perform *ClearA*. To fix it, we add an extra state after *S* state.

b. *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 doesn't get changed.*

The lab manual is well-designed and contains all information we need to implement our experiment. Specifically, the detailed block diagram for each adder is helpful for us to understand how each adder is realized.

c. *Any additional summary you want to include*

Nope.