ECE 385

Spring 2024

Experiment 3

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

By Wendi Wang (wendiw2)

1. Introduction

The primary objective of this project is to design a 16-bit adder using three methods: Carry Ripple, Carry Lookahead, and Carry Select. Their performances and optimizations will be evaluated by the various adders' area, power, and maximum operating frequencies.

2. Adders

a. Ripple Carry Adder

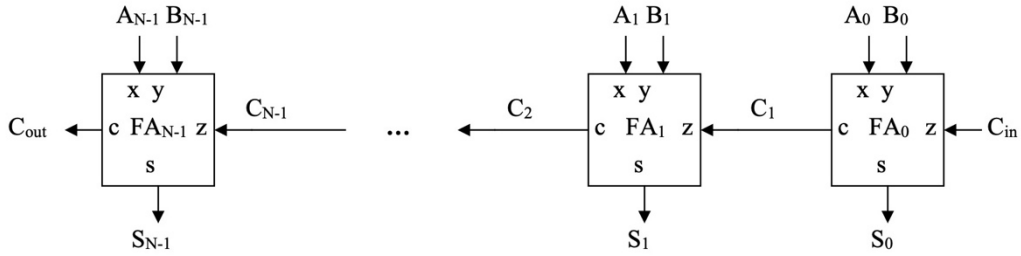
a) Written Description of the Architecture of the Adder

Carry-Ripple Adder 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 C_{in}) are inputted through a set of logic gates to produce a single-bit sum (S) and a single-bit carry-out C_{out}. In a Carry-Ripple Adder, 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. In each full-adder, output bits S and C_{out} follows that:

$S = A \oplus B \oplus C_{in}$

$C_{out} = A \cdot B + B \cdot C_{in} + C_{in} \cdot A$

b) Block Diagram (An N-bit Ripple-Carry Adder)



b. Carry Lookahead Adder

a) Written Description of the Architecture of the Adder

To reduce the computation time of Carry-Ripple Adder, it is apparent that the computation of the carry-out bits must be somehow parallelized. And this is precisely how a Carry Lookahead Adder operates. Carry Lookahead Adder is constructed using two new values (P and G) to calculate the carry-in bit instead of waiting the carry-out bit from the previous full-adder. An N-bit Carry Lookahead Adder (both hierarchical and flat version) still utilizes N full adders, but the propagation process and bits delivery situations are optimized to speed up, compared to Carry-Ripple Adders.

b) Description of P and G logic

P stands for *propagating* logic and G stands for *generating* logic. A carry-out is *generated* (G) if and only if both A and B are 1, regardless of the carry-in. A carry-out is *propagated* (P) if either A or B is 1, regardless of the carry-in.

With P and G defined as $P = A \oplus B$ and $G = A \cdot B$, the Boolean expression for the

carry-out C_{i+1} given a potential C_i is then $C_{i+1} = G_i + (P_i \cdot C_i)$. However, C_{i+1} still depends on C_i, causing the CLA like rippling again. To avoid the slow rippling of the carry bits, the expression of C_{i+1} should be expanded and computed directly from P_is and G_is. For example,

$C_0 = C_{in}$

$C_1 = C_{in} \cdot P_0 + G_0$

$C_2 = C_{in} \cdot P_0 \cdot P_1 + G_0 \cdot P_1 + G_1$

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

In this way, the computation time of the CLA is much faster than that of the CRA.

c) Description of Hierarchical 4x4 Adder

In the hierarchical CLA design, the 16-bit A and B are divided into 4 groups of 4 bits. First, each group of 4 bits go through a 4-bit CLA, which generates two additional output signals, the group propagate (P_G) and the group generate (G_G) with logics being:

$P_G = P_0 \cdot P_1 \cdot P_2 \cdot P_3$

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

Since there are four such 4-bit CLAs, each CLA will produce one bit of P_G and one bit of G_G. We orderly denote them as $P_{G0}, P_{G4}, P_{G8}, P_{G12}$ and $G_{G0}, G_{G4}, G_{G8}, G_{G12}$. Next, instead of cascading the four 4-bit CLAs by connecting the C_{out} from the previous 4-bit CLA to the C_{in} of the next 4-bit CLA, which brings us back to the slow rippling situation again, we should generate the C_{in}s of the 4-bit CLAs using $P_{G0}, P_{G4}, P_{G8}, P_{G12}$ and $G_{G0}, G_{G4}, G_{G8}, G_{G12}$ as follows:

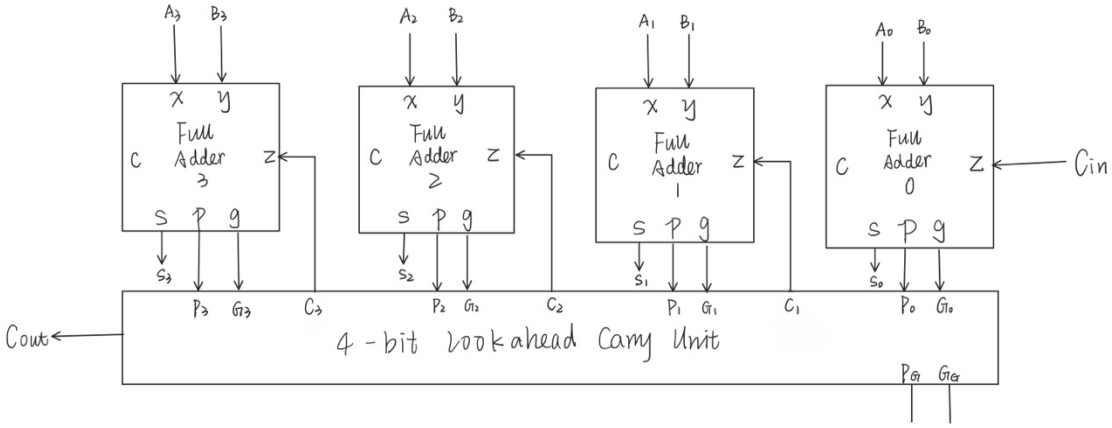
$C_4 = C_0 \cdot P_{G0} + G_{G0}$

$C_8 = C_0 \cdot P_{G0} \cdot P_{G4} + G_{G0} \cdot P_{G4} + G_{G4}$

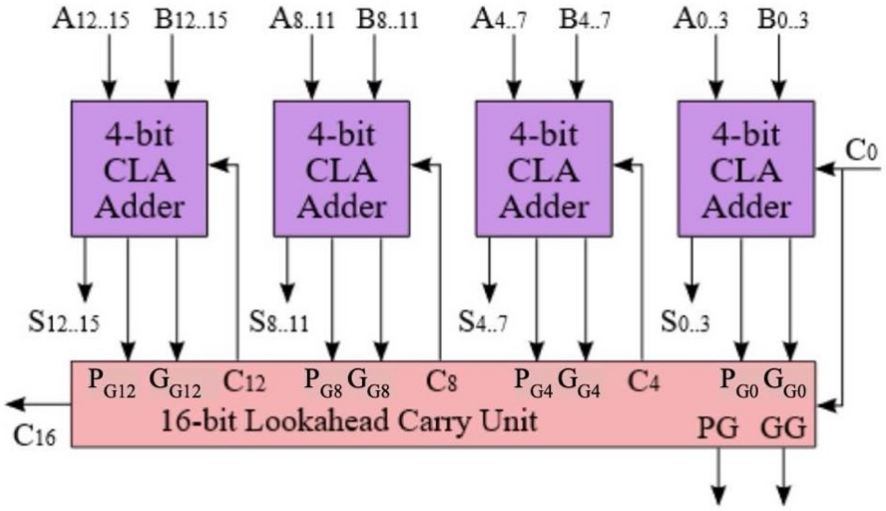
$C_{12} = C_0 \cdot P_{G0} \cdot P_{G4} \cdot P_{G8} + G_{G0} \cdot P_{G8} \cdot P_{G4} + G_{G4} \cdot P_{G8} + G_{G8}$

Therefore, this design becomes hierarchical. We can directly take a copy of the 4-bit Carry Lookahead Unit in the 4-bit CLA, but instead of the inputs coming from full adders, this time the inputs are $P_{G0}, P_{G4}, P_{G8}, P_{G12}$ and $G_{G0}, G_{G4}, G_{G8}, G_{G12}$ from the 4-bit CLAs at the upper level.

d) Block Diagram (Inside a Single CLA, 4-bit)



e) Block Diagram (Chaining CLAs)



c. Carry Select Adder

a) Written Description of the Architecture of the Adder

A Carry-select Adder consists of two full adders (or two Carry Ripple Adders if multiple bits are grouped) and one multiplexor. One adder computes the sum and the carry-out bit based on the assumption that the carry-in bit is 0, while the other adder computes the sum and the carry-out bit based on the assumption that the carry-in bit is 1. In this way, both possible outcomes are pre-computed. Once the real carry-in bits arrive, the corresponding sum and carry-out is selected to be delivered to the next stage. By paying the price of almost twice the numbers of adders, Carry-select Adders gain speedup compared to Carry Ripple Adders.

b) Description of Carry Select Adders at a high level

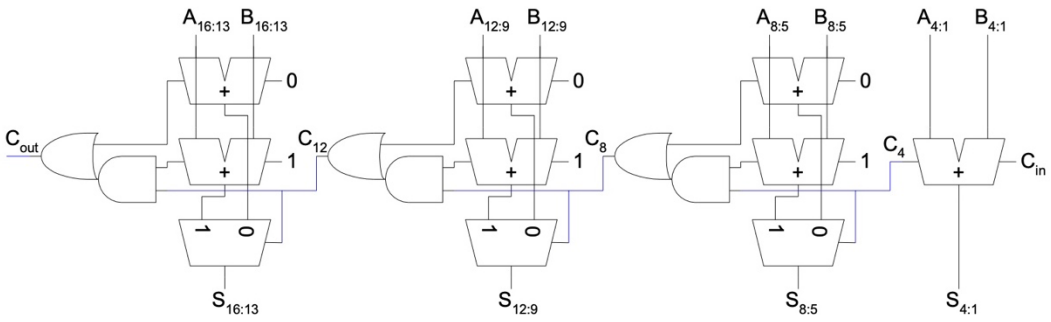
A Carry-select Adder consists of two full adders (or two Carry Ripple Adders if multiple bits are grouped) and one multiplexor. One adder computes the sum and the carry-out bit based on the assumption that the carry-in bit is 0, while the other adder

computes the sum and the carry-out bit based on the assumption that the carry-in bit is 1. Since these calculations are done without knowing the carry-in bits, they can be done in parallel.

Rapidly selecting correct sums can be done with some logic gates. There are two cases where we select the outcomes of assuming the carry-in bit is 1 between two pre-calculated values. The first one is that both A and B are 1, and the second one is either A or B is one, but the carry-out bit of the previous Carry-select Adder is also 1. The first situation can be obtained by OR gates when gluing Carry-select Adders. The second situation can be obtained by ANDing the carry-out bit of the previous Carry-select Adder and the carry-out bit of the adder whose carry-in bit assuming to be 1.

The carry-out bit of the previous Carry-select Adder is fed into the next Carry-select Adder's multiplexer, serving as the select bit of the 2-to-1 MUX. The final carry-out bit of the whole calculation is then determined by first ANDing the carry-out bit of the last Carry-select Adder and the carry-out bit of the adder whose carry-in bit assuming to be 1, finally ORing the output of the AND gate and the carry-out bit of the adder whose carry-in bit assuming to be 0.

c) Block Diagram of the Whole Carry Select Adder



d. Written Description of all .SV modules

a) adder_toplevel.sv

Module: adder_toplevel.sv

Inputs: clk, reset, run_i, [15:0] sw_i

Outputs: sign_led, [7:0] hex_seg_a, [3:0] hex_grid_a, [7:0] hex_seg_b, [3:0] hex_grid_b

Description: This file contains a high-level module of completing adding two 16-bit numbers A and B using three types of adders. It defines an adder (among three types of adders), a negative-edge detector, two register units, two hex drivers, and a synchronizing debounce.

Purpose: Provide a top-level running process of adding two 16-bit binary numbers and demonstrating the result on LEDs.

b) full_adder.sv

Module: full_adder.sv

Inputs: x, y, z

Outputs: s, c

Description: This file contains a module of adding two bits and outputs the sum as well as the carry-out bit. It takes a carry-in bit and two bits to be added as inputs (namely x, y, and z), and uses some logic expressions to represent the sum bit s and the carry-out bit c ($s = x \oplus y \oplus z, c = x \cdot y + y \cdot z + z \cdot x$).

Purpose: Perform whole process of adding two single bits and output the sum.

c) full_adder4.sv

Module: full_adder4.sv

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

Outputs: [3:0] sum, out

Description: This file contains a module of adding two 4-bit binary numbers. It takes two 4-bit binary numbers A and B and a carry-in bit as inputs, and it outputs the final sum as well as the carry-out bit. This module uses four full-adders rippled one by one to calculate the final sum of A and B.

Purpose: Perform whole process of adding two 4-bit binary numbers and output the sum.

d) hex_driver.sv

Module: hex_driver.sv

Inputs: clk, reset, [3:0] in[4]

Outputs: [7:0] hex_seg, [7:0] hex_grid

Description: This file contains a module of visualizing outputs. To be specific, it contains a module of demonstrating output values on LEDs using 7-segment method. An LED digit on the Urbana Board is divided into 7 segments, with different segment combinations we can express numbers from 0 to F in hexadecimal.

Purpose: To translate hex numbers (from 0 to F) into figures appearing on LEDs.

e) load_reg.sv

Module: load_reg.sv

Inputs: clk, reset, load, [DATA_WIDTH-1:0] data_i

Outputs: [DATA_WIDTH-1:0] data_q

Description: This file contains the module of loading registers of length DATA_WIDTH. With DATA_WIDTH specified when instantiating the register and the load signal held high, this module can be reset or loaded the value in a module instantiation according to the values of its inputs.

Purpose: Load the input value to a specified register when certain signal is held high, otherwise load zero to the specified register.

f) lookahead_adder.sv

Module: lookahead_adder.sv

Inputs: [15:0] a, [15:0] b, cin

Outputs: [15:0] s, cout

Description: This file contains a module of adding two 16-bit binary numbers using a 16-bit carry-lookahead adder (a hierarchical adder consisting of four 4-bit carry-lookahead adders). It takes two 16-bit binary numbers a, b and the carry-in bit cin as inputs, and outputs the sum s as well as the carry-out bit cout using the algorithm of carry-lookahead adders. Since it is a hierarchical adder, it also uses four 4-bit carry-lookahead adders and performs the "lookahead" process on four 4-bit carry-lookahead adders, using the group propagation bits and group generation bits.

Purpose: Perform whole process of adding two 16-bit binary numbers using a 16-bit carry-lookahead adder and output the sum as well as the carry-out/overflow bit.

g) lookahead_adder4.sv

Module: lookahead_adder4.sv

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

Outputs: [3:0] sum, pg, gg

Description: This file contains a module of adding two 4-bit binary numbers using a 4-bit carry-lookahead adder (flat version). It takes two 4-bit binary numbers A, B and the carry-in bit as inputs, and outputs the sum as well as the group propagation bit and group generation bit instead of the carry-out bit. The group propagation bit and group generation bit will be fed into the carry-lookahead unit for further bits calculation.

Purpose: Perform whole process of adding two 4-bit binary numbers using a 16-bit carry-lookahead adder and output the sum as well as the group propagation bit and group generation bit for further usage.

h) negedge_detector.sv

Module: hex_driver.sv

Inputs: clk, in

Outputs: out

Description: This file contains a module of detecting the negative edge of the input signal, to guarantee a certain process only run once by raising the output signal for a single clock cycle.

Purpose: To make sure a certain process only run once.

i) ripple_adder.sv

Module: ripple_adder.sv

Inputs: [15:0] a, [15:0] b, cin

Outputs: [15:0] s, cout

Description: This file contains a module of adding two 16-bit binary numbers using a 16-bit carry-ripple adder (flat version). It takes two 16-bit binary numbers a, b and the carry-in bit cin as inputs, and outputs the sum s as well as the carry-out bit cout using the algorithm of carry-ripple adders. Since it is a flat version, it just ripples 16 full adders to calculate the result.

Purpose: Perform whole process of adding two 16-bit binary numbers using a 16-bit carry-ripple adder and output the sum as well as the carry-out/overflow bit.

j) select_adder.sv

Module: select_adder.sv

Inputs: [15:0] a, [15:0] b, cin

Outputs: [15:0] s, cout

Description: This file contains a module of adding two 16-bit binary numbers using a 16-bit carry-select adder (a hierarchical adder consisting of seven 4-bit full adders). It takes two 16-bit binary numbers a, b and the carry-in bit cin as inputs, and outputs the sum s as well as the carry-out bit cout using the algorithm of carry-select adders. Since it is a hierarchical adder, it uses seven 4-bit full adders and performs the "select" process on four 4-bit full adders. After pre-calculating the results of all possible carry-in bit values, it uses several AND gates, OR gates, and four 2-to-1 MUX to "select" the sum and the carry-out bits when true carry-in bit arrives.

Purpose: Perform whole process of adding two 16-bit binary numbers using a 16-bit carry-select adder and output the sum as well as the carry-out/overflow bit.

k) sync_debounce.sv

Module: sync_debounce.sv

Inputs: clk, d

Outputs: q

Description: This file contains a module implementing a denouncer circuit and synchronizer. It serves as a synchronizer for pushbutton and switch inputs, addressing potential instability by detecting stable input cycles using a counter. The choice of counter width is conditional, using a larger width for synthesis to optimize performance.

Purpose: It addresses potential instability by detecting stable input cycles using a counter.

e. Description at a high level of the area, complexity, and performance tradeoffs among the adders

Carry Ripple Adder is the simplest (least complexed) design among the three types of adders, although it has the longest computation time (thus worst performance) among the three types of adders.

On one hand, every full adder must wait for the previous full adder's carry-out bit before it can perform its bits' calculation, which increases its propagation delay with the size of N (for two N-bit binary numbers adding).

On the other hand, since an N-bit Carry Ripple Adder only uses N full adders, it has no extra gates or multiplexers or other in the design, Carry Ripple Adder has the least area and power consumption compared to Carry-lookahead Adder and Carry-select Adder, both of which are more complexed than Carry Ripple Adder.

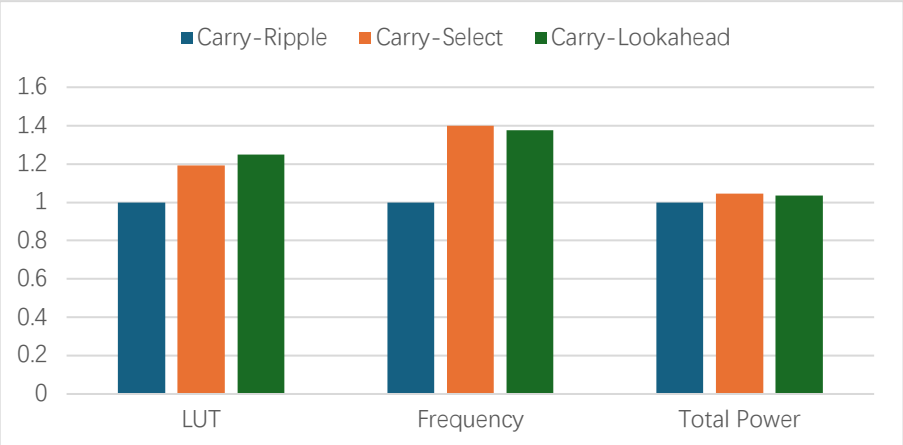
Carry-lookahead Adder (middle complexed) and Carry-select Adder (most complexed) have similar computation time, which is less than that of Carry Ripple adder, since in our design they are both 4x4 hierarchical 16-bit adders, where there is no need to wait for the previous full adder's carry-out bit in both adders' design. However, since there are extra gates or multiplexers to avoid waiting for the previous full adder's carry-out bit in CLA and CSA, CLA, and CLA occupy more area and consume more power than CRA and CLA since it has more logic gates. For CSA, where we use almost twice the number of full adders than CRA, the area taken by CSA is significantly more than CRA.

Combining the area, power, and timing together, CSA is assumed to have the best performance among the three types of adders because it has an optimized computation time as well as smaller area and power consumption compared to CLA.

f. (Post-lab Q1)

Frequency = 1 / (10 ns - WNS)

	Carry-Ripple	Carry-Select	Carry-Lookahead
LUT	88	105	110
Frequency	143.88 MHz	201.29 MHz	198.10 MHz
Total Power	0.088 W	0.092 W	0.091 W



LUT, frequency, and total power breakdown comparison from the plot all make sense. They comply with the theoretical design expectations such as the maximum operating frequency of the carry-lookahead adder is higher than the carry-ripple adder.

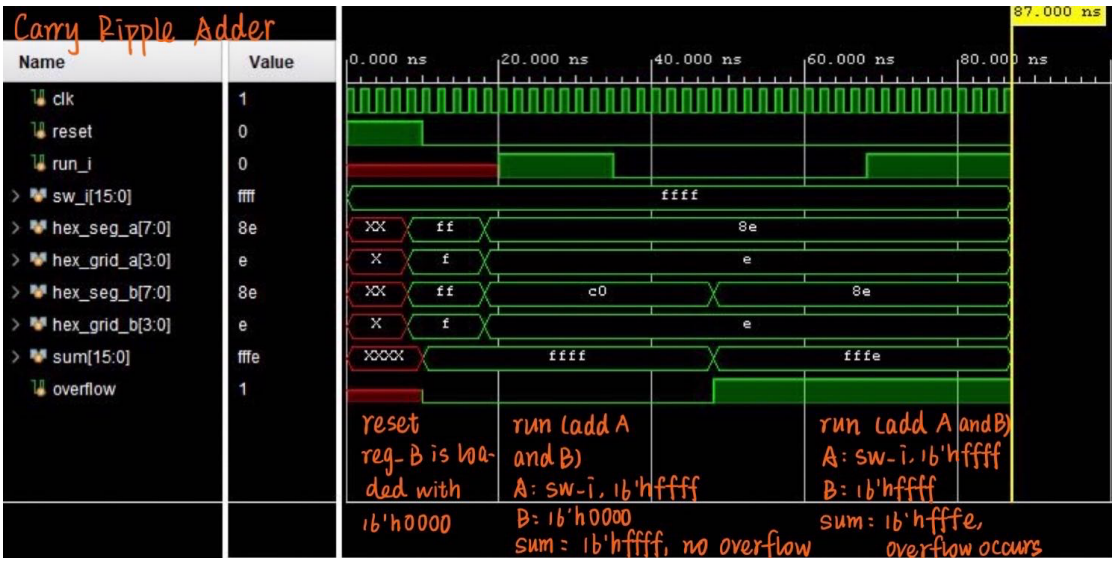
Among three types of adders, Carry-select Adder consumes more power as expected, since it uses almost twice more full adders, three extra multiplexers, and several AND, OR logic gates.

g. (Post-lab Q3)

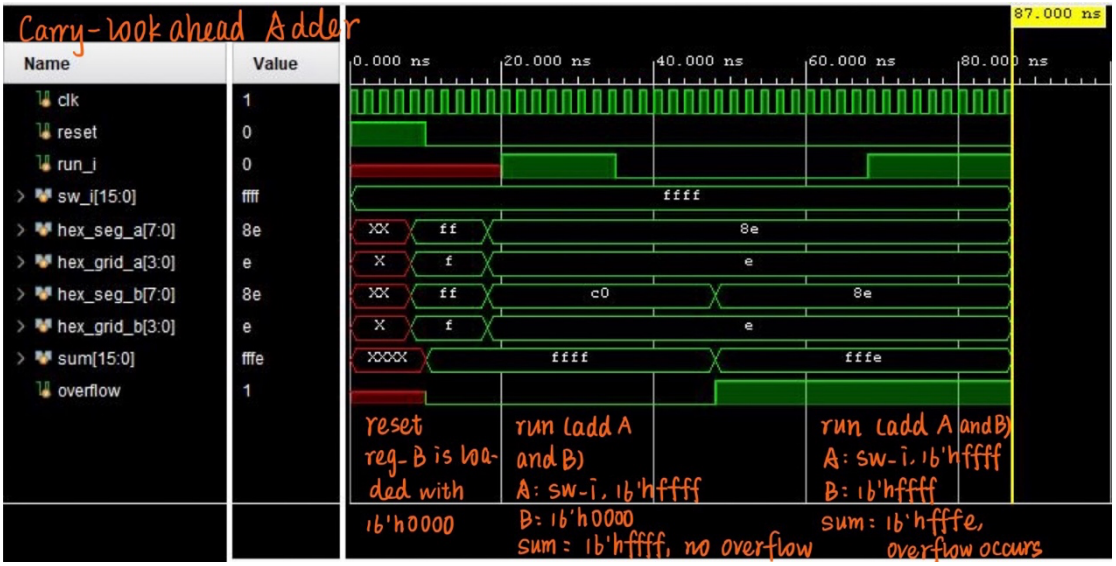
	Carry-Ripple	Carry-Select	Carry-Lookahead
LUT	88	105	110
DSP	0	0	0
Memory (BRAM)	0	0	0
Flip-Flop	90	90	90
Frequency	143.88 MHz	201.29 MHz	198.10 MHz
Static Power	0.076 W	0.076 W	0.076 W
Dynamic Power	0.012 W	0.016 W	0.015 W
Total Power	0.088 W	0.092 W	0.091 W

h. Annotated Simulation Trace

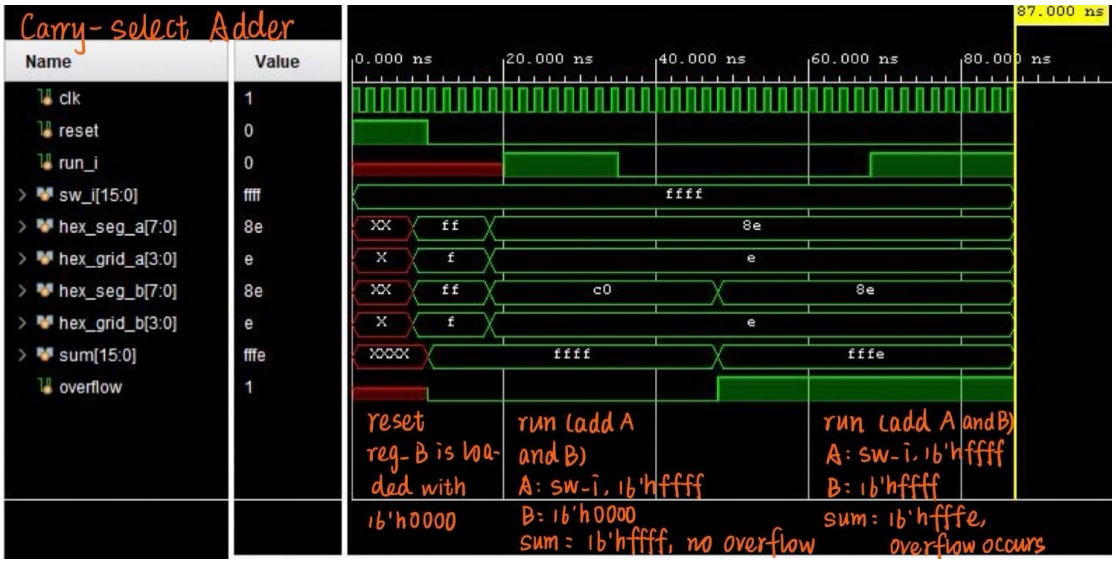
Carry Ripple Adder:



Carry-lookahead Adder:



Carry-select Adder:



Introduction to inputs and outputs used in simulation:

clk: clock signal of 100 MHz

reset: input signal to initialize registers

run_i: input signal to perform adding once a time

sw_i[15:0]: input switch values

hex_seg_a[7:0]: output hex driver's segment value for input switch values

hex_grid_a[3:0]: output hex driver's grid value for input switch values

hex_seg_b[7:0]: output hex driver's segment value for output sum values

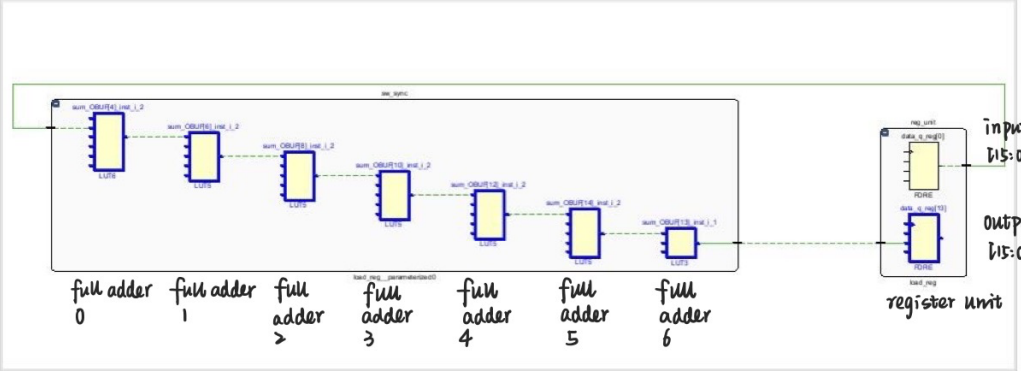
hex_grid_b[3:0]: output hex driver's grid value for output sum values

sum[15:0]: output sum after one adding

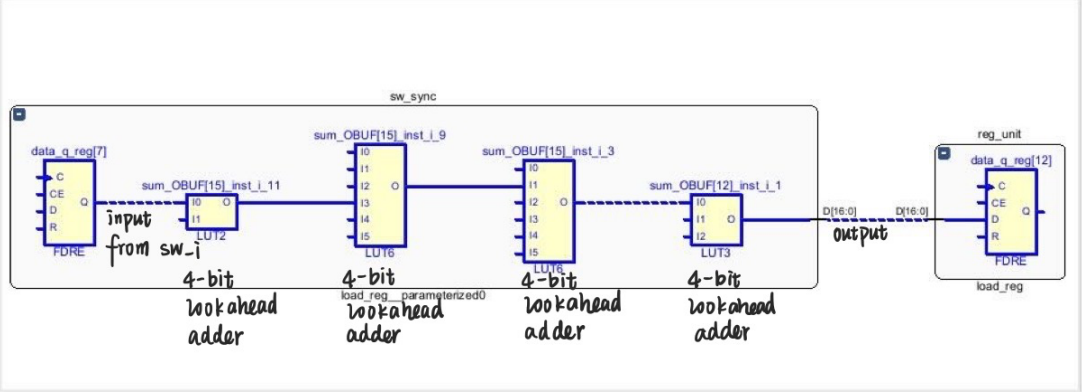
overflow: output sum's carry-out bit, indicating whether overflow occurs in one adding

i. Critical Path Analysis

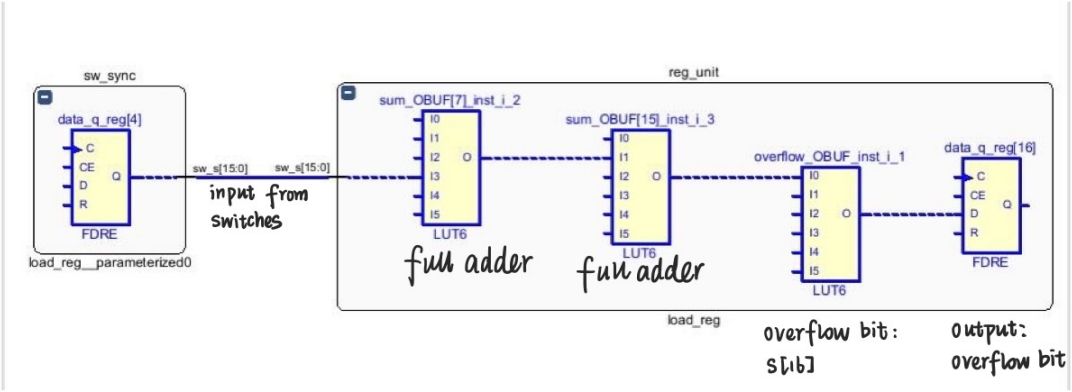
Critical path of Carry Ripple Adder:



Critical path of Carry-lookahead Adder:



Critical path of Carry-select Adder:



The critical paths given by Vivado are not quite the same as the theoretical critical paths. The theoretical critical paths contain all carry chains from switch inputs to one of the outputs. However, for example, the critical path for the Vivado carry-ripple adder did not consist of the carry chain through each full adder module. Maybe it is because FPGA does not use discrete gates (AND, OR etc.) to implement combinational logic, so the "chaining" rules of discrete gates might not be applicable to it. Therefore, the critical path

is not the one where we assume all full adders are chained together as shown in the theoretical block diagram.

As for CSA and CLA's critical paths, they both start from input switch values sw_i[15:0] and end at one of the outputs. These observations are consistent with our theoretical critical paths, though still not all carry chains appear on the critical paths.

3. Answers to the Post-lab Questions

(See the answers to Q1, Q3, and Q4 in Section 2.f, 2.g, and 2.i.)

Q2: 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?)

Answer:

The 4x4 hierarchy is not ideal for Carry-select Adder since the gate delays for this CSA design are the delay from four full adders and three 2-to-1 multiplexers. To find the ideal structure of a hierarchical CSA, we need to compare the time delay of a full adder and the time delay of a 2-to-1 multiplexer, and then we can figure out how many full adders have the same amount of time delays of how many 2-to-1 multiplexers, in which case the time delays of whole CSA design is minimized.

To test this idea, we should conduct the experiment as follows: We should first list all possibilities of a hierarchical 16-bit Carry-select Adder. Next, we need to measure the time delays of each possibility to find out which one has the minimized amount of time delays. In other words, it is just a process of increasing the number of full adders and in the meantime decreasing the number of multiplexers.

Continuing the experiment, we can finally figure out which division of the 16-bit adder is ideal. Some additional information we need is how to calculate the delay from the total number of bits you are adding, what the delays of an extended number of full adders are, what the delay of a single full adder or a single full adder of a certain size (4-bit, 8-bit, …) is, and what the delay of a single 2-to-1 multiplexer is.

4. Conclusion

a. Main Bug

When implementing the Carry Lookahead adder, I didn't use the 16-bit lookahead unit to pass in and pass out the group propagation bit as well as the group generation bit, causing four 4-bit lookahead adders "rippling" together.

b. Summary

I think all graphs given in the manual help me a lot in understanding the logic and writing codes. Accompanied with word description, this lab is straightforward and easy to understand.