ECE 385

Spring 2024

Lab 3

Ziheng Li (zihengl5)

Xin Yang (xiny9)

Section AL1

Prof. Zuofu Chen

Introduction

In this lab, we designed a binary adder that takes in two 16-bit values and outputs one 16-bit number, also shown in hex on the LED. We analyzed and observed the performance of implementing three types of adders: carry-ripple adder, carry lookahead adder, carry-select adder.

CRA functions by propagating the carry bits sequentially through each full adder. CLA uses the concept of generating and propagating to choose the carry bits in advance that avoids slow rippling. CSA reduced operating time by doing the computation in parallel and choosing the result of each part dynamically based on the actual carry-in bit.

Adders

CRA:

The Carry-Ripple Adder (CRA) consists of 16 full-adders. For each full-adder, it takes one bit of A/B, and *Cin* as input. While its output is *Cout*, and *S*. From the least significant bit to the most significant bit, each full-adder will compute the result of the addition based on A/B and Cin (previous Cout). Then output one bit Cout to the next full-adder. This pattern repeats for the next states of CRA.

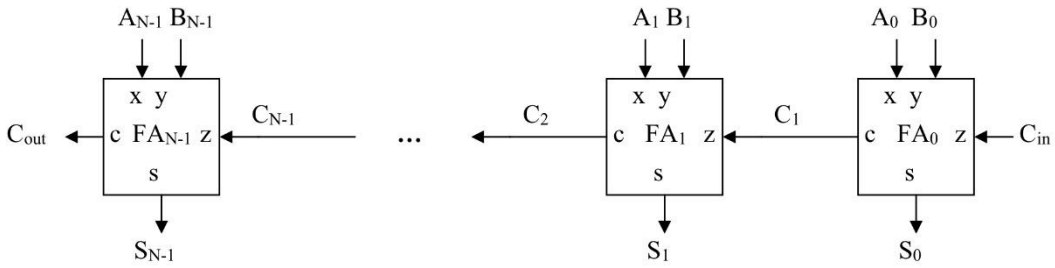

Fig1. Block diagram for CRA

CLA:

The Carry Lookahead Adder (CLA) used a 4*4-bit hierarchical implementation instead of 16-bit. It introduces the concept of generating and propagating. Within each 4-bit block, these two signals are computed based on the input bits. The generating (G) is computed as A AND B. While the propagating (P) is computed as A XOR B. In this way, the carries for each stage can be determined independently of the previous carry in. The design of CLA reduces much propagation delay compared to CRA.

For a single 4-bit carry lookahead adder, the adder use P & G signal to avoid propagation delay as discussed above. In addition, a graph for individual CRA adder is shown below.

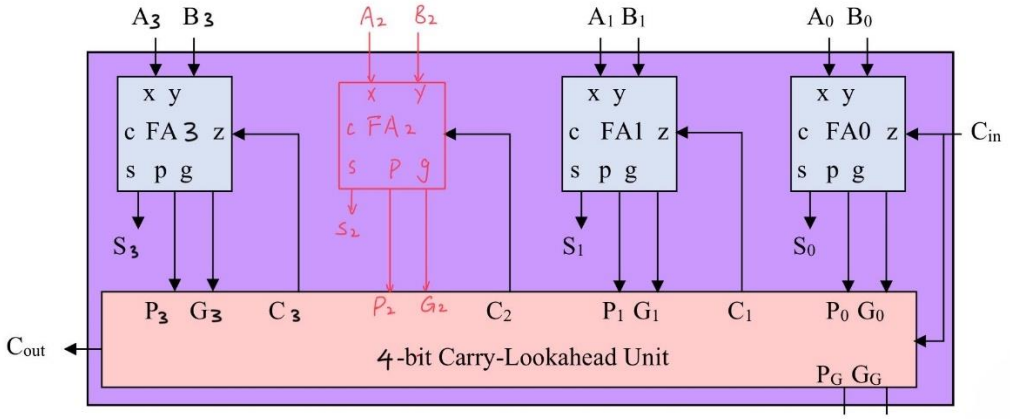

Fig2. Block diagram for 4-bits CLA

As for the 16-bits carry lookahead adder, cannot implement it using the same P & G logic in 4-bits CLA. Even though calculation the carry out requires only two layers of gates (one AND, one OR), as the number of bits increased, the gates are getting wider and wider, exceeding LUT width inside Urbana Board. Hence, we use 4*4 hierarchy and utilize PG & GG instead of P & G. The logic for the final carry out is exactly the same, except

$$PG = P0 \cdot P1 \cdot P2 \cdot P3$$

$$GG = G3 + G2 \cdot P3 + G1 \cdot P3 \cdot P2 + G0 \cdot P3 \cdot P2 \cdot P1$$

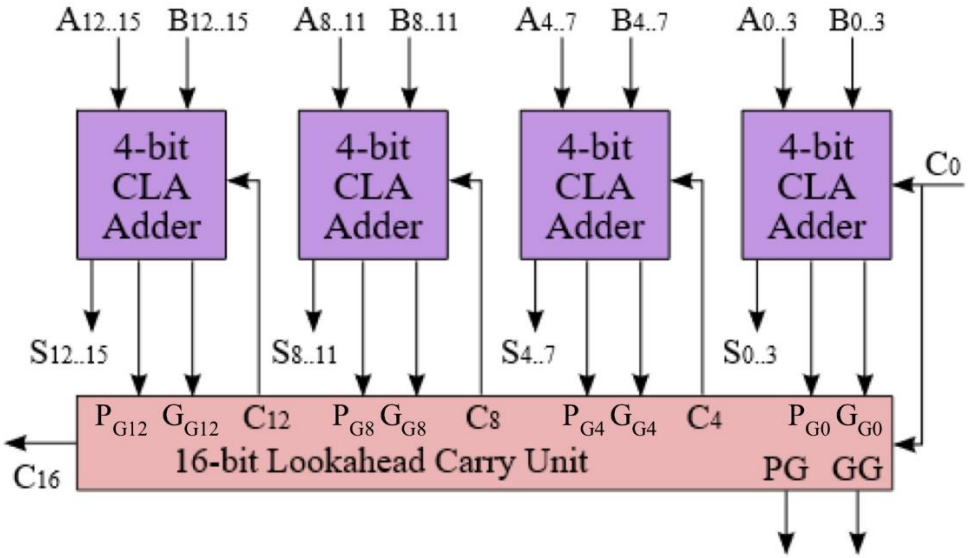

Fig3. Block diagram for 16-bits CLA implemented by 4*4 hierarchy

CSA:

The Carry Select Adder (CSA) is the most efficient circuit to implement adding operation among the three adders. For single 4-bits CSA, it consists of two 4-bits CRA and 5 Muxes. The key feature of the CSA is its ability to compute the sum and carry outputs for both possible carry input values (0 or 1) in parallel, even before the actual carry input is determined. This feature successfully reduces the time delay associated with the propagation of the carry bit through adders when there is more bits. Let us take 4-bits CSA as an example, there are two 4-bits CRA in the design, one CRA calculates the output and carry out assuming the carry in as 0, while another CRA assumes the carry in as 1. Once the actual carry in C has been input from the previous block, the correct sum and carry outputs for each block are selected by C using multiplexers.

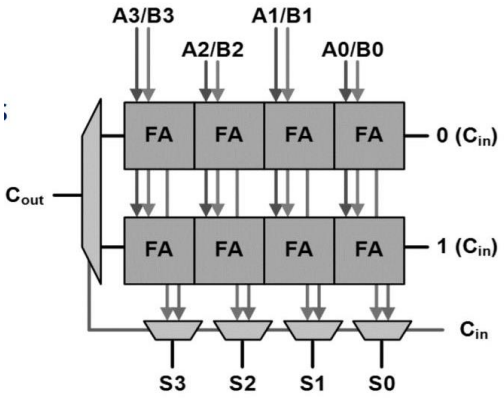

Fig4. Block diagram for 4 bits CSA

As for the 16 bits CSA, we just chain up four 4-bits CSA, feed the Cout of the previous adder as the Cin of the current one, as shown below.

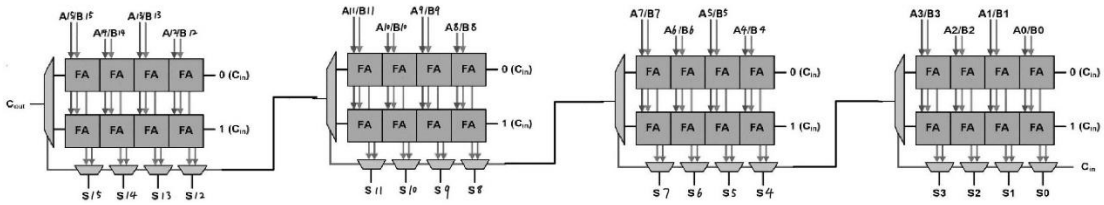

Fig5. Block diagram for 16 bits CSA

Module descriptions

Module: ripple_adder.sv
Inputs: [15:0] a, [15:0] b, cin
Outputs: [15:0] s, cout
Description: This module implements a ripple adder. It consists of two sub-modules: full_adder and adder4.
Purpose: Produce binary summation of A and B utilizes CRA.

Module: lookahead_adder.sv
Inputs: [15:0] a, [15:0] b, cin
Outputs: [15:0] s, cout
Description: This module implements a ripple adder. It consists of one sub-module: lookahead4.
Purpose: Produce binary summation of A and B utilizes CLA.

Module: select_adder.sv
Inputs: [15:0] a, [15:0] b, cin
Outputs: [15:0] s, cout
Description: This module implements a ripple adder. It consists of one sub-module: select_adder_fourbits.
Purpose: Produce binary summation of A and B utilizes CSA.

Module: hex_driver.sv
Inputs: clk, reset, in[4]
Outputs: [7:0] hex_seg, [3:0] hex_grid
Description: The module uses a loop to instantiate four nibble_to_hex converters, mapping each of the 4-bit inputs to their corresponding 7-segment display patterns. Then choose the hex_grid to determine which of the four LEDs is active.
Purpose: Convert binary into hexadecimal and then display them on the LEDs.

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 is the top-level module for various adders. First it loads the data from sw_i into a register. Second, it sums the output of the register with another data input, and then pass final result into a display module. It also has some basic loading and debouncers modules used for registers and switches.
Purpose: This module brings everything together, fetch in the two inputs and show the output through LED.

Module: load_reg.sv
Inputs: clk, reset, load, [DATA_WIDTH-1:0] data_i
Outputs: [DATA_WIDTH-1:0] data_q
Description: When the Load signal is high, the register will store the data, and when reset is 1, the data stored will be clear.
Purpose: Storing data of given width.

Module: sync_debounce.sv
Inputs: clk, d
Outputs: q
Description: This module takes the signal d from the fpga button and switch. And outputs q for other modules to use. Firstly, it flops the input d twice. And then change the button when 2^COUNTER_WIDTH stable input cycles are recorded.
Purpose: It is used as a debouncer and a synchronizer for push buttons and switches.

Module: negedge_detector.sv
Inputs: clk, in
Outputs: out
Description: This module is a negative edge detector which will set out as 1 for 1 clock cycle if it detects the in drops from 1 to 0.
Purpose: Allows the register to load once, and not during the full duration of button press.

Area, complexity, and performance

CRA

Area: CRA generally requires less area compared to CLA and CSA because the design is simpler, n-bits CRA only requires n full adders.

Complexity: The design and implementation complexity is the lowest among the three adders, consisting only of a series of full adders chained together.

Performance: The main drawback of CRA is their performance, particularly in terms of speed. The carry signal will propagate through all full adders from the LSB to MSB. Without getting the previous carry out, the next stage full adders all stops to work, leading to significant delays when getting more bits.

CLA

Area: CLA require more area than CRA and CSA due to the additional logic (G & P) needed to compute carry bits in advance. This complexity increases the number of gates and hence the overall area.

Complexity: The design of CLA is more complex than that of CRA and CSA. We are required to calculate the carry bits using generation and propagation delays in advance. And as the number of bits increased, we will have wider and wider gates, hence increasing the design complexity.

Performance: CLA has significantly better performance than CRA by quickly determining carry bits without waiting for the propagation. However, it’s slightly slower than CSA.

CSA

Area: CSA requires much more area than CRAs and slightly less area compared to CLAs. This is because CSA utilizes two sets of full adders and computes all possible inputs in parallel, leading to an increased number of gates and muxes.

Complexity: The design complexity of CSAs is moderate. Even though they are more complex than CRA due to the parallel computation and selection logic, but personally speaking, they are simpler than CLA because the carry computation is more straightforward.

Performance: CSA has the best performance among the three adders. By separately computing sums and carry outs and then selecting the correct one with latter carry in, adders in CSA can pre-calculate the result without waiting propagation delay. In my point of view, CSA is the best adder in terms of design complexity and performance.

Performance (See Post Lab for details)

Simulation Trace

Fig6. Simulation trace

We load A & B with three different values at different times, and the results are all correct as shown in the figure.

Critical Path Analysis

CRA:

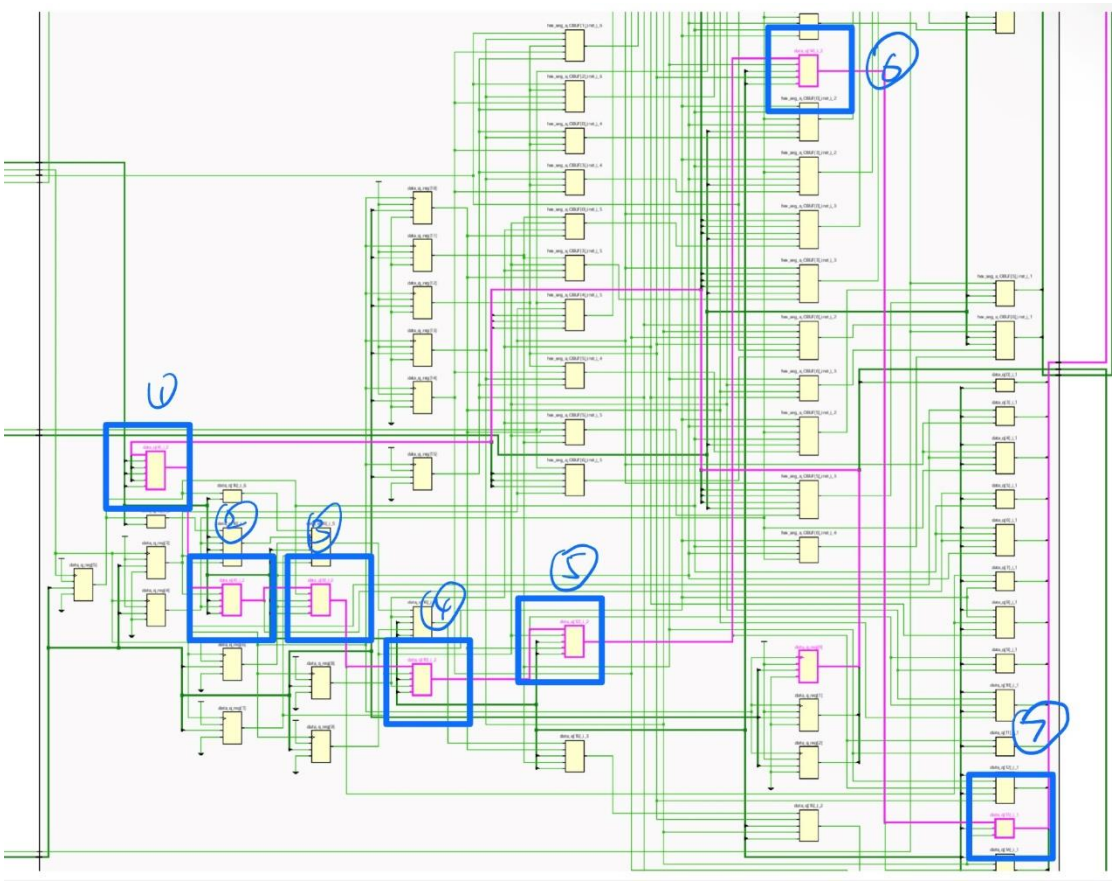

Fig7. Critical Path (CRA)

The critical path for the carry ripple adder pass through 7 lookup table, it is the most among the three adders as expected.

CLA:

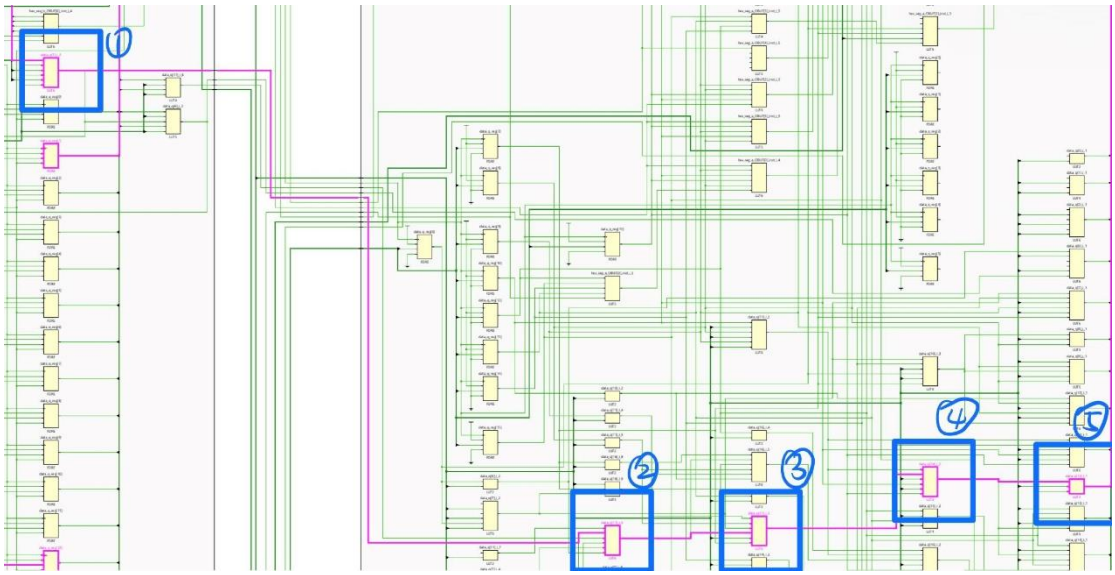

Fig8. Critical Path (CLA)

The critical path for the carry lookahead adder pass through 5 lookup table, less than CRA but more than CSA.

CSA:

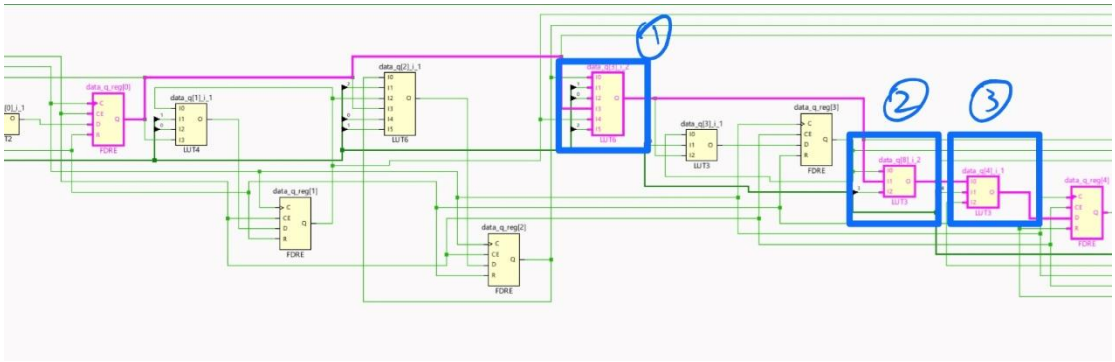

Fig9. Critical Path (CSA)

The critical path for the carry select adder pass through only 3 lookup table, it is the least among the three adders as expected.

Post Lab Question

1. Use the WNS value you got from the timing report (refer to IVT) to compute the maximum frequency. Explain how you computed the frequency from the WNS value in your report.

The formula we used to calculate the frequency based on WNS is 1 / (10ns-WNS). Since WNS is the time left in the design before the propagation delay of the critical path.

	Carry-Ripple	Carry-Select	Carry-Lookahead
LUT	88	96	102
Frequency (Gigahertz)	0.14	0.182	0.227
Total Power (W)	0.086	0.087	0.085

Fig10. Design statistics

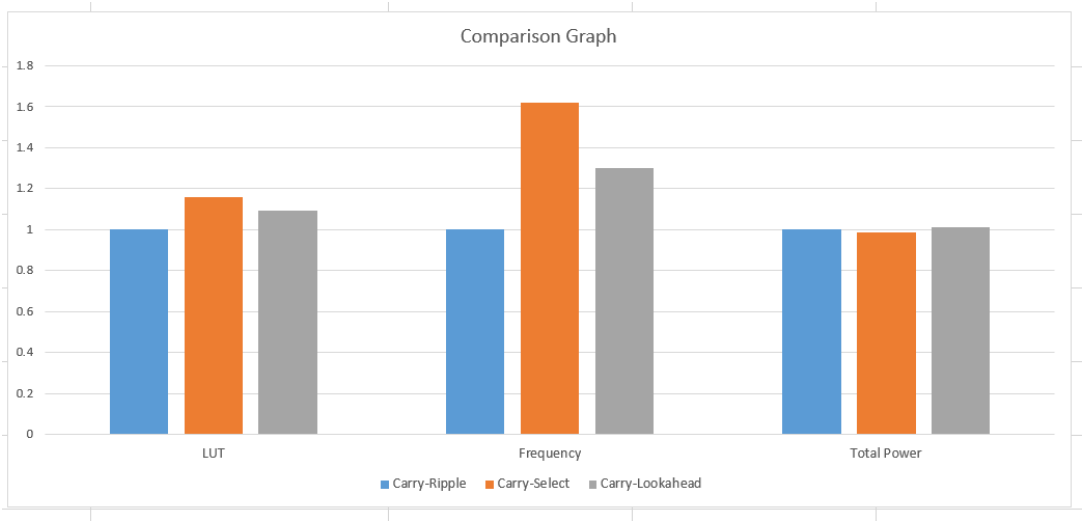

Fig11. Comparison graph

For frequency part, our comparison graph make sense, the result is the same as we expected where CSA faster than CLA faster than CRA. In addition, the number of LUT also make sense, as CRA is the most complicated design requiring most LUT, and CRA requires less since it the simplest. However, the power consumption for the three adders are almost the same, and CSA even consume less power than the other two adders. We thought that number of LUTs used in a digital circuit is related to the power consumption of the device. After doing some research, we found possible explanation

for this phenomenon. First, CSA is being optimized somehow, it use additional LUTs to reduce the depth of logic, leading to lower signal propagation delays and less frequent switching, and causing it to consume less power. Second, a design with more LUTs but with a lower overall activity factor (less frequent logic state changes) could also consume less power than a design with fewer LUTs but higher activity. Finally, there might be some slight errors in Vivado.

2. 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?)

Although 4x4 is clear in format, it is not an ideal design using CSA. We can improve its efficiency by moving some computation logic from the previous part to the later part. In this case, a developer has to do several experiments to figure out. We have to test the gate delay of each full adder and the time consumed for 2-to-1 mux. Then we can redesign the circuit based on the data to see if we can optimize the running time.

3. In addition to the design statistics shown in graph form above, you will need to fill in the remaining columns for each adder.

	Carry-Ripple	Carry-Lookahead	Carry-Select
LUT	88	96	102
DSP	0	0	0
Memory (BRAM)	0	0	0
Flip-Flop	90	90	90
Frequency (Gigahertz)	0.14	0.182	0.227
Static Power (W)	0.072	0.072	0.072
Dynamic Power (W)	0.015	0.015	0.014
Total Power (W)	0.086	0.087	0.085

Fig12. Design Statistics

4. Compare the critical path as given by Vivado to the theoretical critical path, which we discussed in class. For example, did the critical path for the Vivado carry-ripple adder consist of the carry chain through each full adder module? Why or why not? Include the screenshots of the critical path for each adder alongside your theoretical analysis and annotations.

Theoretically, we assume the CRA will consume more LUTs and Registers than the other two. But in the actual critical path given by Vivado, it does not contain the carry chain for through each full adder module. We thought this is caused by the automatic optimize tool in synthesis of Vivado which varied the implementation from our design. However, we can see the difference of LUT used in different algorithms: 7 LUT for CRA, 5 LUT for CLA, and 3 LUT for CSA. More LUT will cause longer propagation delay, and hence the result match with our theoretical analysis.

Conclusion & Bugs

Overall speaking, we did not meet much difficulty when writing the code; however, during the design of the Carry Select Adder, we planned to use just one mux for the output bit of the 4 adders since they share a common select bit which is c_in. However, we failed to find a proper expression in Vivado. The countermeasure is to separate them into 4 independent alwasy_comb modules.

The instruction of the debug core is not that clear, at least for me. I suggest that a detailed file can be provided for using it.

In this lab, we learned the tradeoff between performance and complexity based on 3 different designs of adders. It is a valuable lesson since we can achieve the same goal by various methods while we can determine which one to use based on the specific demand in each situation.