# ECE 385

Fall 2023

Experiment #

# Lab3

Ziyi Lin
Zhuoyang Lai
TA: Jerry Wang

1. Introduction

(i) Ripple adder:

Ripple adders are sequence of full adders, each full adder ahead takes carryout value of the last full adder, it will wait for n times for n-bit full adders.

（ii) Carry Lookahead adder：

CLA use logic expression to compute carryout value. The input involves C0（the input carry） and Pi and Gi the Generation and Propagation ofeach adder, this allows the parallel computation of every cout and the final sum as well. But the longer the adder are, the logicexpreesion will also become more complex which will require more gate.

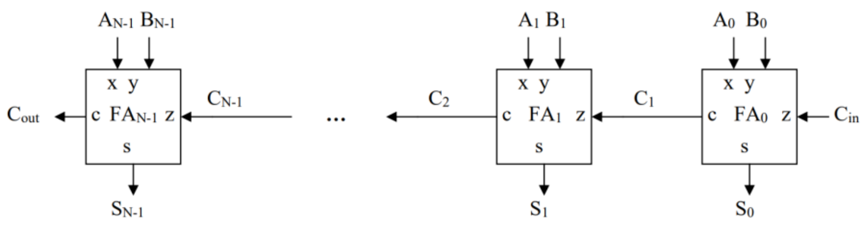
（iii） Carry Select adder:

CSA computes use two sets of adders to compute the two occasions of carryout of the last adder are 0 or 1. This allows parralell computation of each unit, and once the carryout of the last unit arrives, it can use a 2-1 MUX to decide which result to use. This design saves time cost but takes two times of gate consumption.

2.Adders

a. (i) It is a sequence of full adders, it takes input A and B and cin, and output cout = （A&B）|（A&C）|（B&C）and sum = A xor B xor C

(ii)



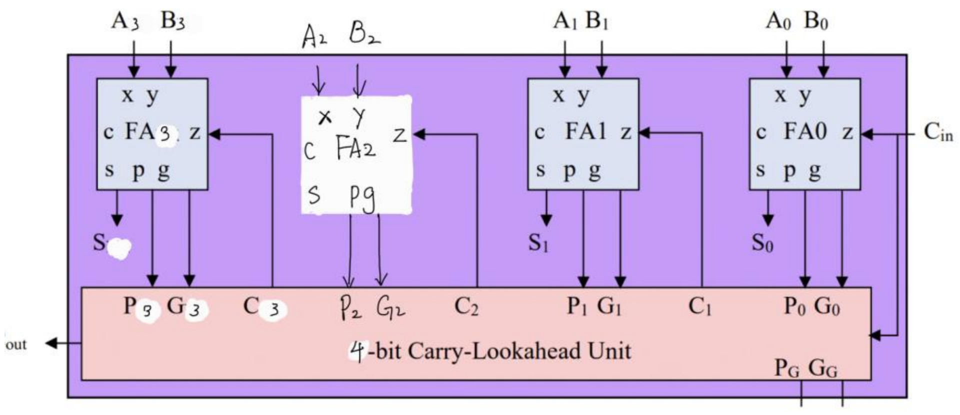
b.（i） The Carry Lookahead Adder （CLA） computes carryout directly using input values, without waiting for carry values from previous stages. It does this by using the concepts of Generate （G） and Propagate （P）. By pre-computing potential carries, the CLA drastically reduces the time to calculate the final sum compared to an RCA.

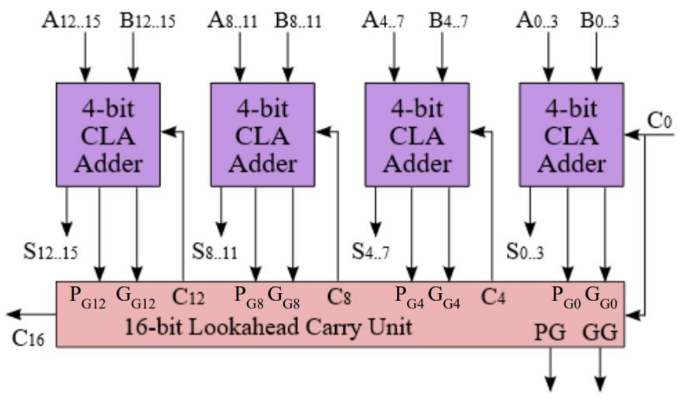
(ii) Propagate (P) is used to determine if a carry from a previous stage will propagate through the current stage. Pi = Ai xor Bi

Generate （G） is used to determine if the current stage will generate a carry, regardless of whether a carry is received from a previous stage. Gi = Ai & Bi

(iii)4*4 hierarchical adder is made of a sequence of 4 4-bit CLA, each 4-bit unit wait for the previous unit carryout to arrive and then start computation

(iv)





c. Carry select adder:

(I) written descripsion:

The carry-select adder generally consists of ripple-carry adders and a multiplexer. The carry-select adder typically employs a combination of ripple-carry adders and a multiplexer. To add two n-bit numbers using a carry-select adder, the process involves utilizing two adders, specifically two ripple-carry adders. These adders perform the calculation twice: once assuming the carry-in is zero and another assuming it's one. Subsequently, after obtaining both sets of results, the multiplexer comes into play. It selects the accurate sum and carry-out based on the known carry-in value.

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

Both cin (0/1) cases are calculated at the same time, which is in parallel. All the computation value is pre-calculated, though there is some delay to get the real cin, but as soon as actual cin is valid, the result will be selected quickly.

(iii)

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

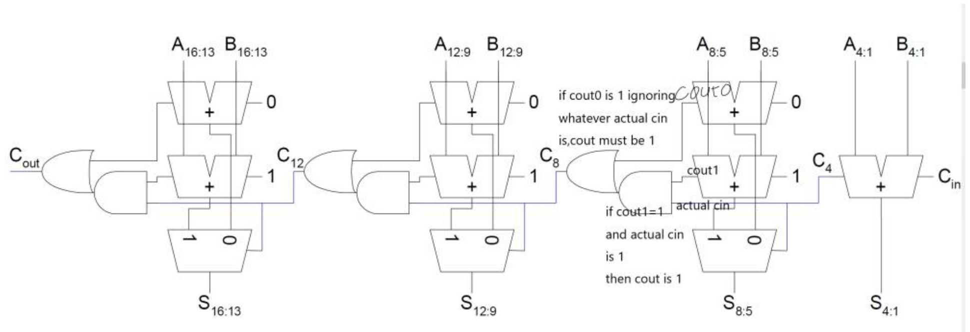


**Figure 5: 16-bit Carry-Select Adder Block Diagram**

d. Written description of all .SV modules (including the provided modules).

Module: **control.sv**
Inputs: Clk, Reset, Run
Outputs: Run_O
Description: This is a simple state machine (state A B C), also convert a switch input to a one clock cycle long event.
Purpose: control the behavior of a digital system based on the input signals. It serves as a control unit that generates an output signal, Run_O, based on the current state and input conditions.

Module: **mux2_1_17.sv**
Inputs: S, [15:0] A_In, [16:0] B_In
Outputs: [16:0] Q_Out
Description: This is a 17-bit 2:1 multiplexer (MUX) module that selects one of two data inputs, A_In or B_In, based on the value of the select signal S. When S is low (0), the output Q_Out is driven by the 16-bit input A_In with a leading zero. When S is high (1), the output Q_Out is driven by the 17-bit input B_In.
Purpose: This module serves as a versatile data selection unit , allowing for the routing of data from two sources to a single output based on a control signal (S).

Module: **reg_17.sv**
Inputs: Clk, Reset, Load, [16:0] D
Outputs: [16:0] Data_Out
Description: This is a 17-bit register module with synchronous reset and synchronous load functionality. It is designed to store a 17-bit data input, D, and can be controlled by clock signals (Clk), a synchronous reset signal (Reset), and a synchronous load signal (Load). When the reset signal

is
active (Reset asserted), the register clears its contents to all zeros. When the load signal is asserted (Load active), the contents of the register are set to the value of the 17-bit input D on the positive edge of the clock signal Clk.
Purpose: This module serves as a 17-bit register, providing the capability to store data temporarily and control its state through clocked operations such as resetting and loading.

Module: **lookahead_adder**
Inputs: A (16-bit), B (16-bit), cin (carry-in)
Outputs: S (16-bit), cout (carry-out)
Description: This module performs addition of two 16-bit numbers, A and B, using a Carry Lookahead Adder (CLA). It calculates the sum (S) and carry-out (cout) for the addition operation. The CLA design allows for fast and efficient addition without the need for sequential logic.
Purpose: The module is used for fast arithmetic operations in digital systems

Module: **lookahead_adder_small.sv**
Inputs: A (4-bit), B (4-bit), cin (carry-in)
Outputs: P_G (Generate Propagation), G_G (Generate Generation), sum (4-bit)
Description: This module is a building block for a Carry Lookahead Adder (CLA). It calculates the generate (G_G) and propagate (P_G) signals for each bit, as well as the sum of two 4-bit numbers (A and B), considering a carry-in (cin).
Purpose: The module is used to efficiently compute the generate and propagate signals for a // CLA-based adder, enabling faster arithmetic operations in digital systems.

Module: **adder_toplevel.sv**
Inputs: Clk , Reset_Clear , Run_Accumulate,SW
Outputs: sign_LED, hex_segA , hex_gridA , hex_segB , hex_gridB
Description: This is the top-leveladders lab. It handles the addition and accumulation of 16-bit numbers using an accumulator-based approach. By comment or uncomment the sentence we can pick whihc methos we could use for adding.
Purpose: This module serves as the main controller for the adder lab.

Module: **HexDriver.sv**
Inputs: clk,reset,in[4] [3:0] (4x 4-bit)
Outputs:hex_seg [7:0] (8-bit): Output for 7-segment hex displays (segments).
hex_grid [3:0] (4-bit): Output for 7-segment hex displays (grid control).
Description: The HexDriver module is responsible for converting four 4-bit nibbles into 7-segment hex display outputs. It generates the necessary control signals for each display segment and grid control, allowing the display of hexadecimal characters (0-9 and A-F).
Purpose: This module is used to show hexadecimal values based on the input nibbles.

Module: **select_adder.sv**

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

Outputs: cout, [15:0] S

Description: This module is a 16-bit adder that selects between two different inputs based on the value of the carry-in (cin) signal. It consists of three 4-bit ripple-carry adders and multiplexers for selecting the appropriate results. The carry-in signal determines whether the addition is performed with a carry-in of 0 or 1 for each 4-bit section of the inputs A and B. The final result S is a 16-bit sum, and the cout output represents the carry-out from the most significant bit of the addition.

Purpose: This module is used add two 16-bit operands, A and B, with the option to include a carry-in (cin) from a previous stage. The module performs the addition in a segmented way.

Module: **ripple_adder.sv**

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

Outputs: [15:0] S logic cout

Description: This module adds two 16-digit numbers together, just like you would add them manually on paper. It works by dividing the addition into four 4-bit segments and using the ripple_adder_small module for each segment. The carry-out from one segment is propagated to the carry-in of the next segment. The result is the sum S, and the carry-out (cout) tells you if there was a carry beyond the leftmost digit.

Purpose: add two 16-digit binary numbers together. not the fastest method.

Module: **ripple_adder_1bit.sv**

Inputs: logic a, b, cin

Outputs:logic sum, cout

Description: This module represents a 1-bit ripple-carry adder, which is the basic building block for larger adders. It takes two input bits,'a' and 'b,' and a carry-in signal, 'cin.' It computes the sum bit, 'sum,' which is the result of adding 'a,' 'b,' and the carry-in, and it also calculates the carry-out bit, 'cout,' which represents any carry generated during the addition.

Purpose: This module is used to add two 1-bit binary numbers together.

Module: **ripple_adder_small**

Inputs: [3:0] a, [3:0] b , cin (carry-in)

Outputs: [3:0] sum (4-bit sum), cout (carry-out)

Description: This module performs 4-bit addition using a ripple-carry adder. It consists of four 1-bit adders connected in series. Each 1-bit adder takes two bits from the input numbers (a and b), a carry-in signal (cin), and produces a sum bit (sum) and a carry-out bit (cout). The carry-out bit from one 1-bit adder becomes the carry-in for the next adder in the sequence. The final 4-bit sum (sum) is obtained by combining the individual sum bits, and the carry-out from the last 1-bit adder is the overall carry-out (cout) from the 4-bit addition.

Purpose: This module is used to add two 4-bit binary numbers together.

Module: **mux2to1_3**

Inputs: logic S , A_In [3:0] B_In [3:0]

Outputs: Q_Out [3:0]

Description: This module is a 4-to-1 multiplexer If S is 0, it passes the data from A_In to Q_Out; if S is 1, it passes the data from B_In to Q_Out.

Purpose: This module is used when you need to choose between two sets of 4-bit data based on a control signal (S).

e.

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

For area: both CSA and CLA will occupy larger area because for CLA, it has more logic gate to pre-calculate the cout and for CSA, each segment will take double area for cin=0 and cin=1.

For complexity, CRA is easiest and CLA will be more complexilicated if bits goes up

For performance: CRA has the worst performance because waiting time for cin propergating through each adder. For CSA it pre_calculate all the possilble result, as long as cin is valid, by selecting the result we can finish the computation. For CLA, it pre-compute the value of Cout merely using P and G and does not depend on the previous segement.

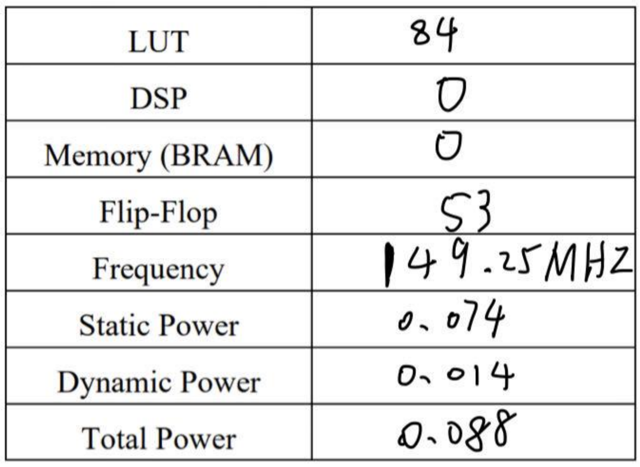
f.

| | Carry-Ripple | Carry-Select | Carry-Lookahead |
|---|---|---|---|
| LUT | 1 | 1.071 | 1.083 |
| Frequency | 1 | 1.306 | 1.426 |
| Total Power | 1 | 1.011 | 0.989 |

Frequency is compute by the forumla: f_max = (（time_actual）/（time_needed）)*f_origin

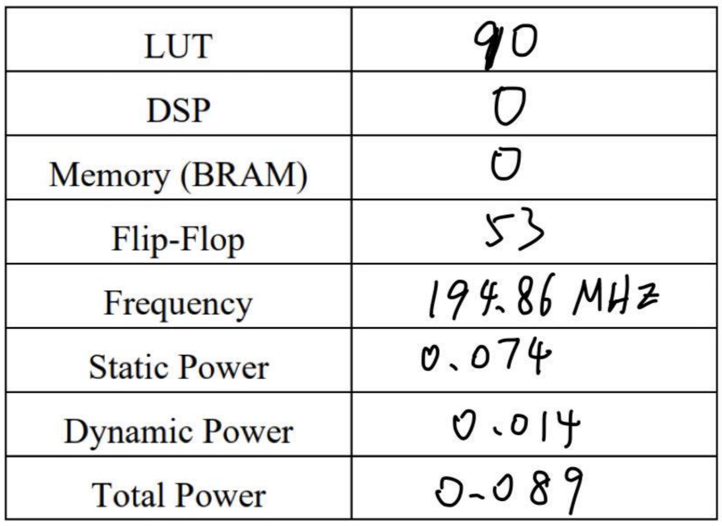
In this case f_max = (（10ns）/(10ns-WNS))*100MHZ

g.



| | |
|---|---|
| LUT | 84 |
| DSP | 0 |
| Memory (BRAM) | 0 |
| Flip-Flop | 53 |
| Frequency | 149.25MHZ |
| Static Power | 0.074 |
| Dynamic Power | 0.014 |
| Total Power | 0.088 |

(ripple adder)



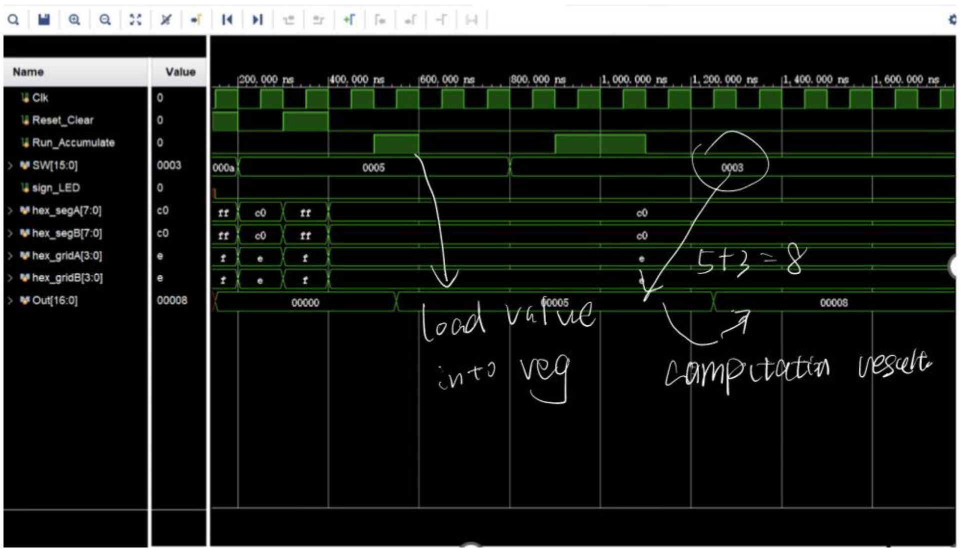
| | |
|---|---|
| LUT | 90 |
| DSP | 0 |
| Memory (BRAM) | 0 |
| Flip-Flop | 53 |
| Frequency | 194.86 MHZ |
| Static Power | 0.074 |
| Dynamic Power | 0.014 |
| Total Power | 0.089 |

(CLA)

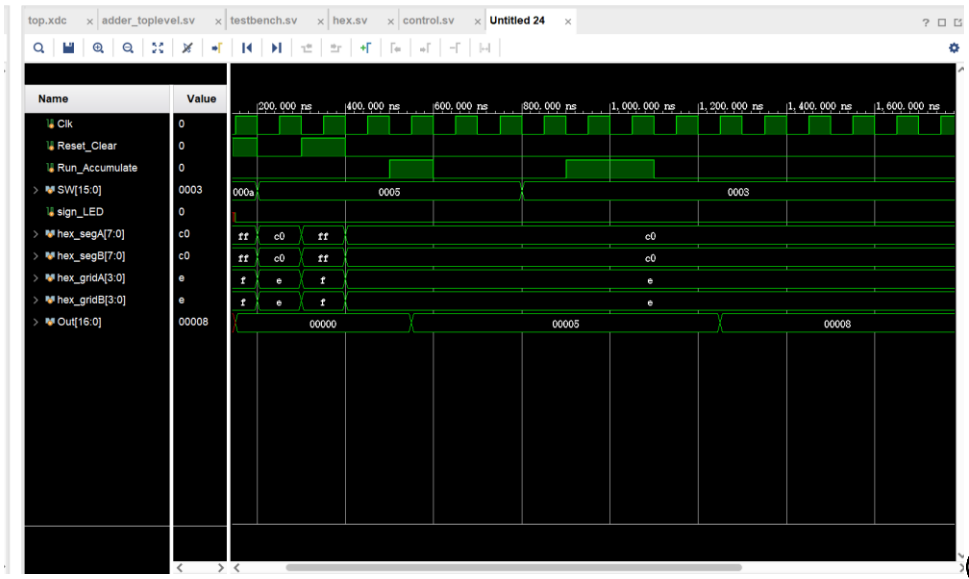
| | |
|---|---|
| LUT | 91 |
| DSP | 0 |
| Memory (BRAM) | 0 |
| Flip-Flop | 53 |
| Frequency | 212.81 MHz |
| Static Power | 0.074 |
| Dynamic Power | 0.013 |
| Total Power | 0.087 |

(CSA)

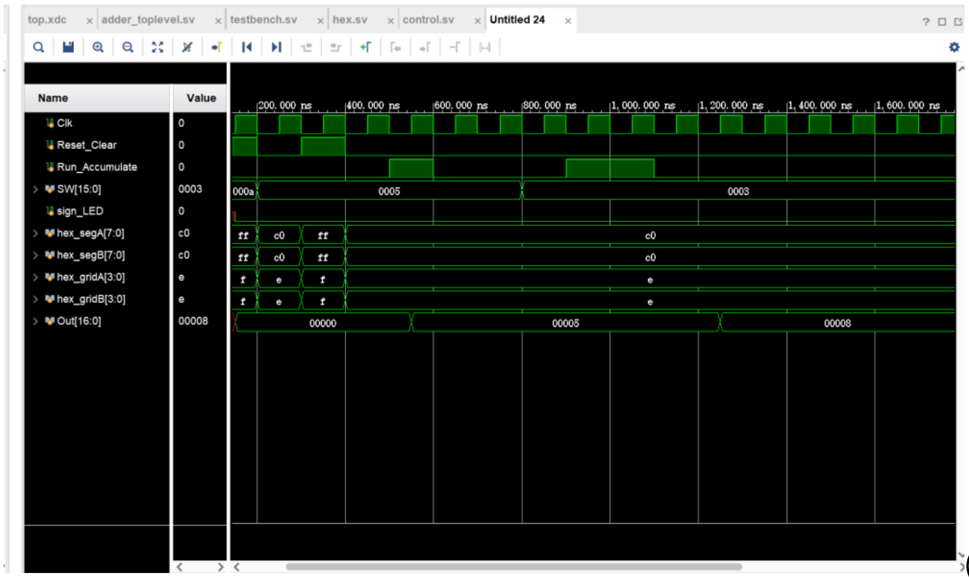
h.



(RCA)



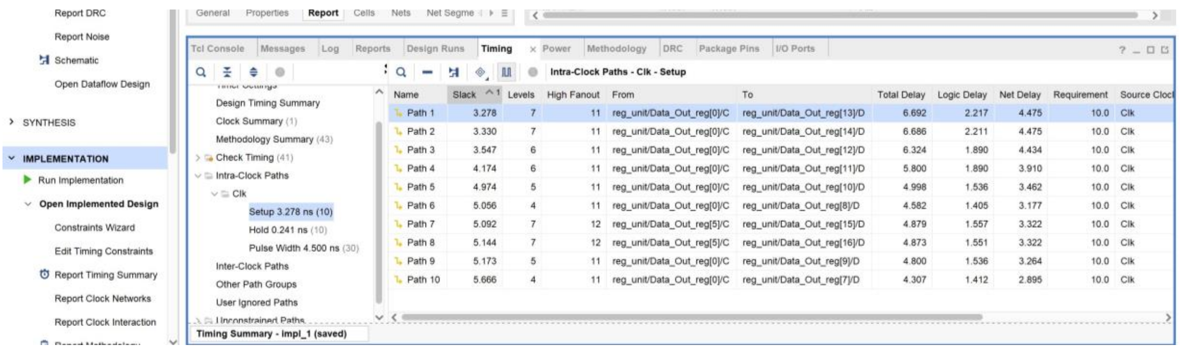
(CLA)



(CSA)

i.

Ripple adder'

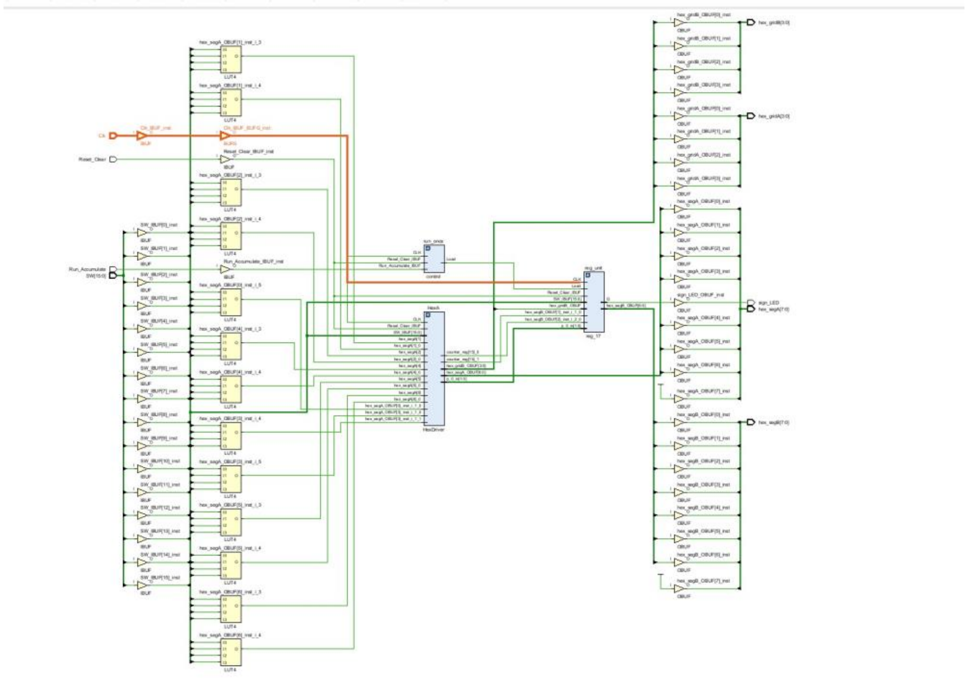


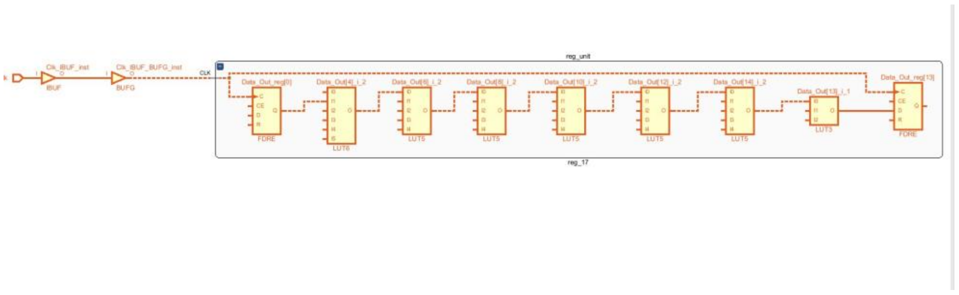
Intra-Clock Paths - Clk - Setup

| Name | Slack | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | 3.278 | 7 | 11 | reg_unit/Data_Out_reg[0]/C | reg_unit/Data_Out_reg[13]/D | 6.692 | 2.217 | 4.475 | 10.0 | Clk |
| Path 2 | 3.330 | 7 | 11 | reg_unit/Data_Out_reg[0]/C | reg_unit/Data_Out_reg[14]/D | 6.686 | 2.211 | 4.475 | 10.0 | Clk |
| Path 3 | 3.547 | 6 | 11 | reg_unit/Data_Out_reg[0]/C | reg_unit/Data_Out_reg[12]/D | 6.324 | 1.890 | 4.434 | 10.0 | Clk |
| Path 4 | 4.174 | 6 | 11 | reg_unit/Data_Out_reg[0]/C | reg_unit/Data_Out_reg[11]/D | 5.800 | 1.890 | 3.910 | 10.0 | Clk |
| Path 5 | 4.974 | 5 | 11 | reg_unit/Data_Out_reg[0]/C | reg_unit/Data_Out_reg[10]/D | 4.998 | 1.536 | 3.462 | 10.0 | Clk |
| Path 6 | 5.056 | 4 | 11 | reg_unit/Data_Out_reg[0]/C | reg_unit/Data_Out_reg[8]/D | 4.582 | 1.405 | 3.177 | 10.0 | Clk |
| Path 7 | 5.092 | 7 | 12 | reg_unit/Data_Out_reg[5]/C | reg_unit/Data_Out_reg[15]/D | 4.879 | 1.557 | 3.322 | 10.0 | Clk |
| Path 8 | 5.144 | 7 | 12 | reg_unit/Data_Out_reg[5]/C | reg_unit/Data_Out_reg[16]/D | 4.873 | 1.551 | 3.322 | 10.0 | Clk |
| Path 9 | 5.173 | 5 | 11 | reg_unit/Data_Out_reg[0]/C | reg_unit/Data_Out_reg[9]/D | 4.800 | 1.536 | 3.264 | 10.0 | Clk |
| Path 10 | 5.666 | 4 | 11 | reg_unit/Data_Out_reg[0]/C | reg_unit/Data_Out_reg[7]/D | 4.307 | 1.412 | 2.895 | 10.0 | Clk |

Design Timing Summary
Clock Summary (1)
Methodology Summary (43)
Check Timing (41)
Intra-Clock Paths
Clk
Setup 3.278 ns (10)
Hold 0.241 ns (10)
Pulse Width 4.500 ns (30)
Inter-Clock Paths
Other Path Groups
User Ignored Paths
Unconstrained Paths

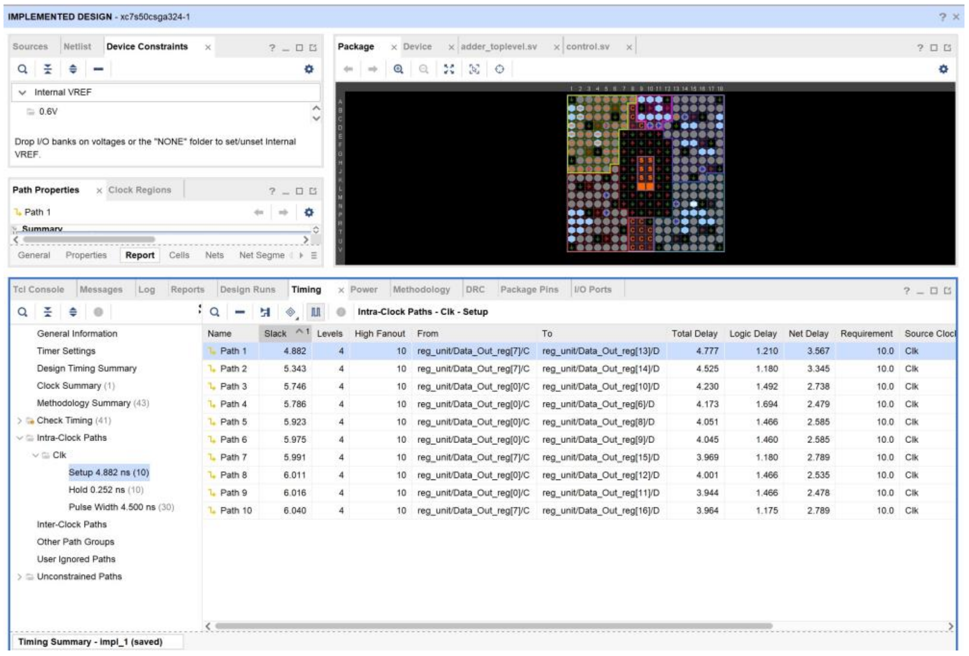
Timing Summary - impl_1 (saved)





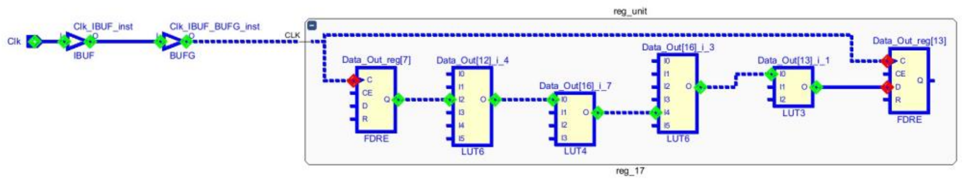
Analysis of CRA's critical path: the design of CRA hold more LUT than CSA and CLA so that might be the reason why slack is the lowest. The start point is data_regsiter_out[0] which make sense because that is exactly where the computation started. After 8 process of the look-up table(possibly 2 bit each), then we got the result.

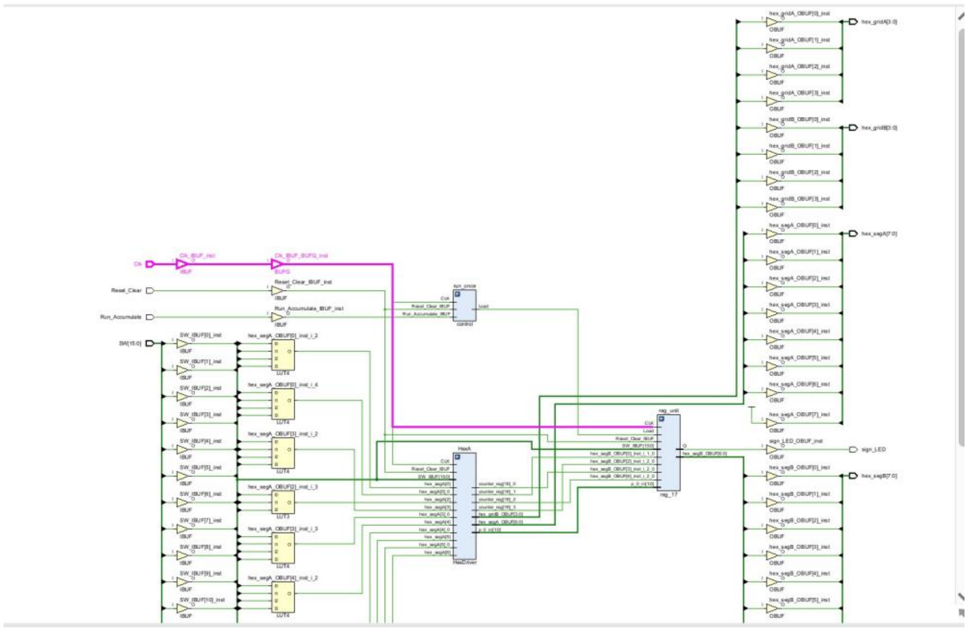
Look ahead adder:



Intra-Clock Paths - Clk - Setup

| Name | Slack | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | 4.882 | 4 | 10 | reg_unit/Data_Out_reg[7]/C | reg_unit/Data_Out_reg[13]/D | 4.777 | 1.210 | 3.567 | 10.0 | Clk |
| Path 2 | 5.343 | 4 | 10 | reg_unit/Data_Out_reg[7]/C | reg_unit/Data_Out_reg[14]/D | 4.525 | 1.180 | 3.345 | 10.0 | Clk |
| Path 3 | 5.746 | 4 | 10 | reg_unit/Data_Out_reg[0]/C | reg_unit/Data_Out_reg[10]/D | 4.230 | 1.492 | 2.738 | 10.0 | Clk |
| Path 4 | 5.786 | 4 | 10 | reg_unit/Data_Out_reg[0]/C | reg_unit/Data_Out_reg[6]/D | 4.173 | 1.694 | 2.479 | 10.0 | Clk |
| Path 5 | 5.923 | 4 | 10 | reg_unit/Data_Out_reg[0]/C | reg_unit/Data_Out_reg[8]/D | 4.051 | 1.466 | 2.585 | 10.0 | Clk |
| Path 6 | 5.975 | 4 | 10 | reg_unit/Data_Out_reg[0]/C | reg_unit/Data_Out_reg[9]/D | 4.045 | 1.460 | 2.585 | 10.0 | Clk |
| Path 7 | 5.991 | 4 | 10 | reg_unit/Data_Out_reg[7]/C | reg_unit/Data_Out_reg[15]/D | 3.969 | 1.180 | 2.789 | 10.0 | Clk |
| Path 8 | 6.011 | 4 | 10 | reg_unit/Data_Out_reg[0]/C | reg_unit/Data_Out_reg[12]/D | 4.001 | 1.466 | 2.535 | 10.0 | Clk |
| Path 9 | 6.016 | 4 | 10 | reg_unit/Data_Out_reg[0]/C | reg_unit/Data_Out_reg[11]/D | 3.944 | 1.466 | 2.478 | 10.0 | Clk |
| Path 10 | 6.040 | 4 | 10 | reg_unit/Data_Out_reg[7]/C | reg_unit/Data_Out_reg[16]/D | 3.964 | 1.175 | 2.789 | 10.0 | Clk |



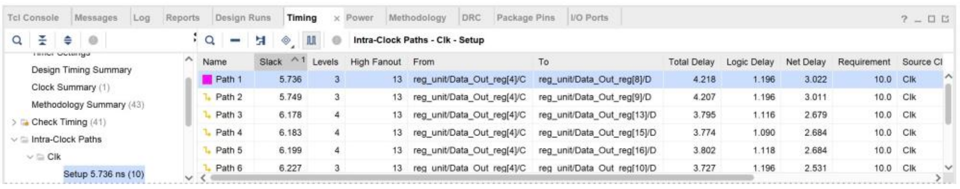


Analysis of CLA’s critical path:

The path of CLA share different path with ripple adder, they with LUT6*2， LUT4, LUT3. Since the number of the LUT is less, the slack is higher with better performance.

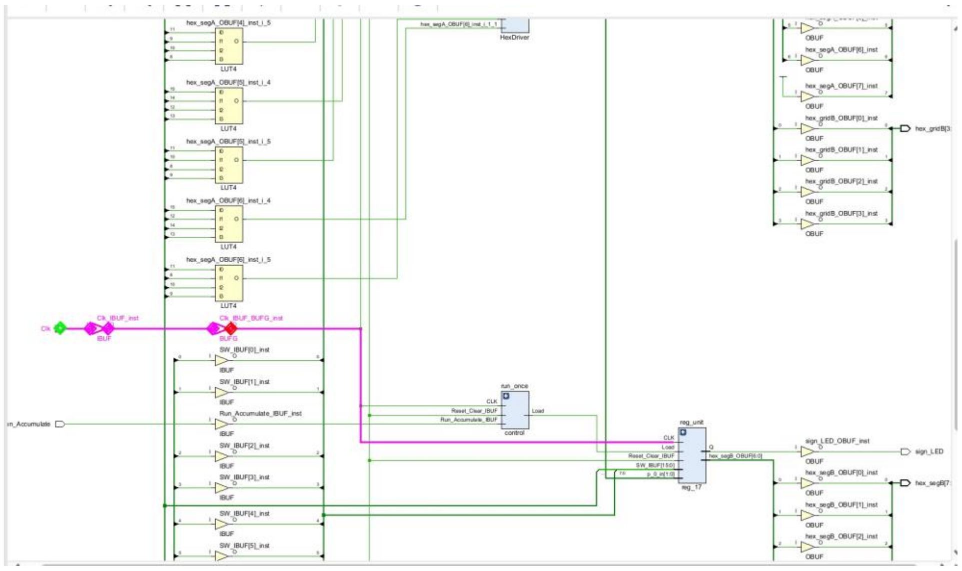
For CLA, the start point is reg[7] end with reg[13], the adder is divided into four segement, which make sense the start point and end point have 4 bits difference, second, since all the segement to the computation simultaneously, the longest path should start from bit[n] to bit[n+4]. Also, LUT6 might be used to find out the G and P which will be applied for furture computation.

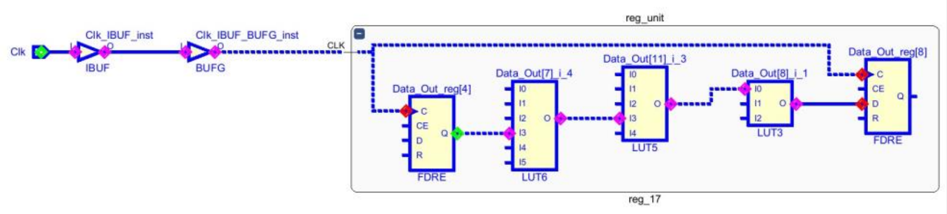
Select ripple adder



Intra-Clock Paths - Clk - Setup

| Name | Slack | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Cl |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | 5.736 | 3 | 13 | reg_unit/Data_Out_reg[4]/C | reg_unit/Data_Out_reg[8]/D | 4.218 | 1.196 | 3.022 | 10.0 | Clk |
| Path 2 | 5.749 | 3 | 13 | reg_unit/Data_Out_reg[4]/C | reg_unit/Data_Out_reg[9]/D | 4.207 | 1.196 | 3.011 | 10.0 | Clk |
| Path 3 | 6.178 | 4 | 13 | reg_unit/Data_Out_reg[4]/C | reg_unit/Data_Out_reg[13]/D | 3.795 | 1.116 | 2.679 | 10.0 | Clk |
| Path 4 | 6.183 | 4 | 13 | reg_unit/Data_Out_reg[4]/C | reg_unit/Data_Out_reg[15]/D | 3.774 | 1.090 | 2.684 | 10.0 | Clk |
| Path 5 | 6.199 | 4 | 13 | reg_unit/Data_Out_reg[4]/C | reg_unit/Data_Out_reg[16]/D | 3.802 | 1.118 | 2.684 | 10.0 | Clk |
| Path 6 | 6.227 | 3 | 13 | reg_unit/Data_Out_reg[4]/C | reg_unit/Data_Out_reg[10]/D | 3.727 | 1.196 | 2.531 | 10.0 | Clk |





Analysis of CSA's critical path: the slack of CSA is highest which means the gate delay is the smallest.
This is because we pre-calculate the possible value and merely select it. According to the path it has only three LUT so the processing speed is higher than the other two design. For CSA, since it is divided into 4 segments(4 bits each), reg[8] and reg[4] has difference of 4 bits which makes sense

3.Post-lab Question

(ii)（other answers are in section 2）

No, it is not ideal, the key thing is find a exact balance point of how many mux and ripple adder we should use. unreasonable number of AND and OR gate between the adders as well as the MUX propagation time will chew up the advantage parallel design brings. To design an ideal version ,we need to measure the CSA adder unit of different size (e.g. 2-bit 4-bit 8-bit) and the time delay of  different combination of MUX and AND and OR gate to identify the best length of a CSA unit to minimize the computation time.

## 4.Conculsion

(1) bug:

We encounter a bug that when we push the button, the add procedure will execute several times. It turned out to be that the button is not debounced.

(2) summary:

This is a basically system Verilog introduction lab, we learned to use assign key word to write simple system verilog program.