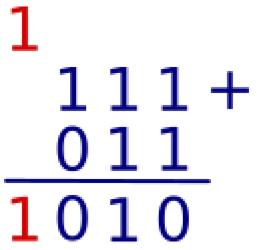
# 32-bit Adder

An adder is a digital circuit that performs the arithmetic operation of addition.

It takes three inputs A, B and carries from before and outputs sum and carry. Adders are the essential component of arithmetic operation in digital systems and come in various types, each with specific characteristics and uses cases.



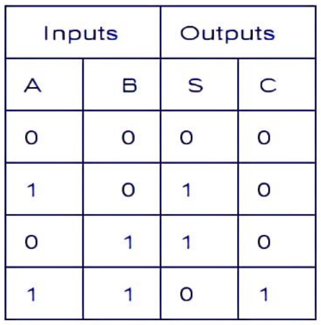
The above example shows a simple addition of 3 bit binary number. There are different adders and they are different because of their area usage, speed in there calculation, complexity of their circuits, etc.

## Types of adder’s:

i) Half adder
ii) Full adder
iii) Ripple carry adder.
iv) Carry look-Ahead adder.

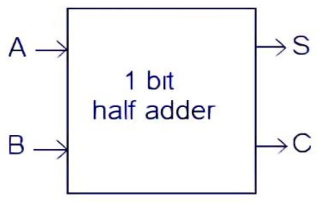
### i) Half adder:

A half adder takes two single-bit inputs, A and B. It computes two outputs: the sum (S) and the carry-out (Cout). The sum is the XOR of A and B, while the carry-out is the AND of A and B. However, it cannot account for a carry-in from a previous stage.

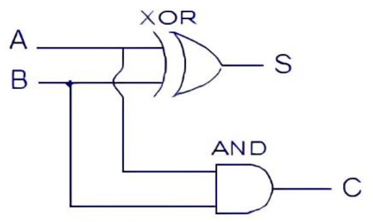


| Inputs | | Outputs | |
|---|---|---|---|
| A | B | S | C |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 |

Truth table



Schematic



Realization

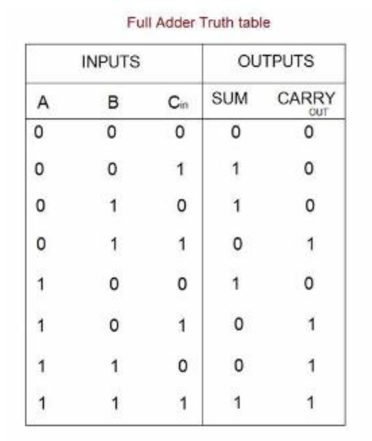
This type of adders is the simplest and they use less area and their circuits are less complex. A simple example of half adder is

**1**
**+1**
**=10**

where ‘0’ is the sum of two numbers and ‘1’ is the carry.

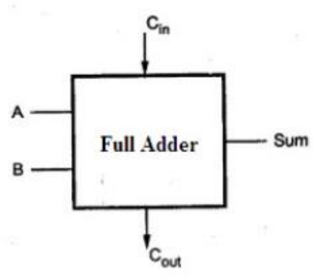
**ii) Full adder:**

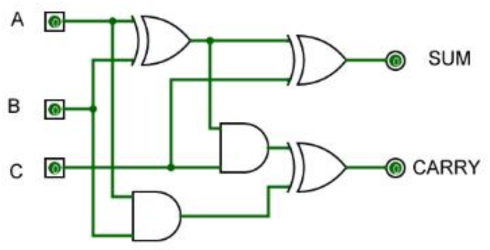
A full adder extends the functionality of a half adder by including a carry-in (Cin) input. It computes two outputs: the sum (S), the carry-out (Cout), and this can be used as a carry-in for the next stage. The sum is calculated similarly to a half adder, but it also considers the carry-in. The carry-out is determined by (A AND B) OR ((A XOR B) &Cin).



Full Adder Truth table

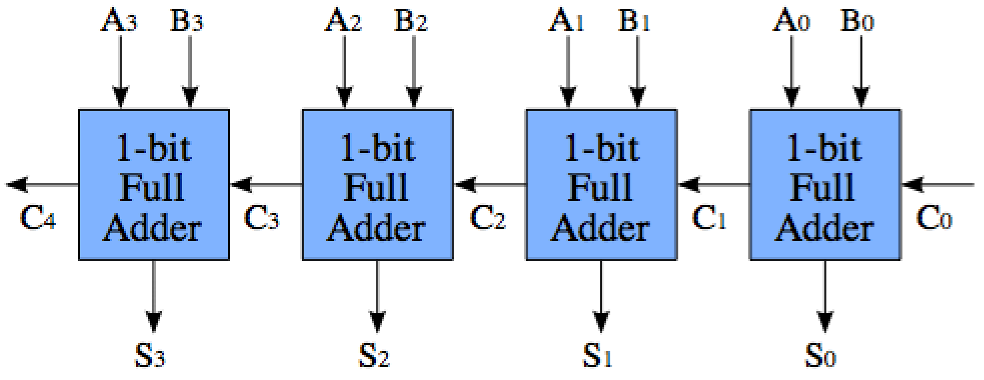
| INPUTS | | | OUTPUTS | |
|---|---|---|---|---|
| A | B | $C_{in}$ | SUM | CARRY OUT |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |





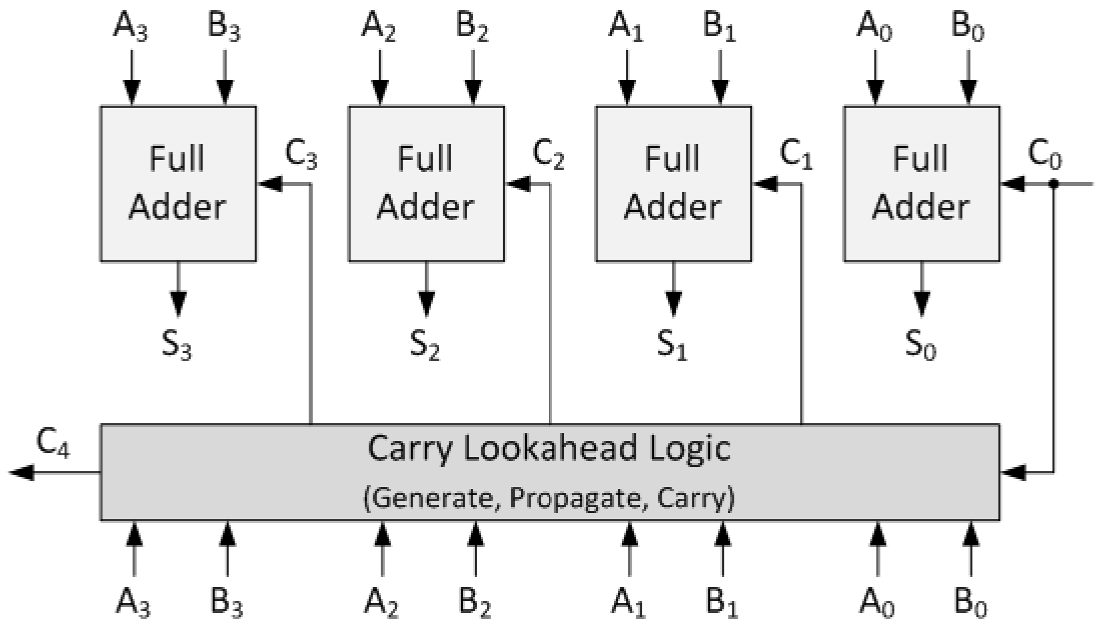
**iii) Ripple carry adder:**

A ripple carry adder (RCA) is built by cascading full adders in series. It's simple and easy to implement but can have long propagation delays because each stage depends on the previous one. The worst-case delay is linear with the number of bits.



**iv) Carry look-Ahead adder:**

A carry look-ahead adder (CLA) is designed to reduce the propagation delay by computing carry signals for each bit position independently and it uses additional circuit for this purpose and by this it eliminates the need for carry propagation between stages, significantly improving speed. The carry-in (Cin) signal is distributed directly to each full adder stage.

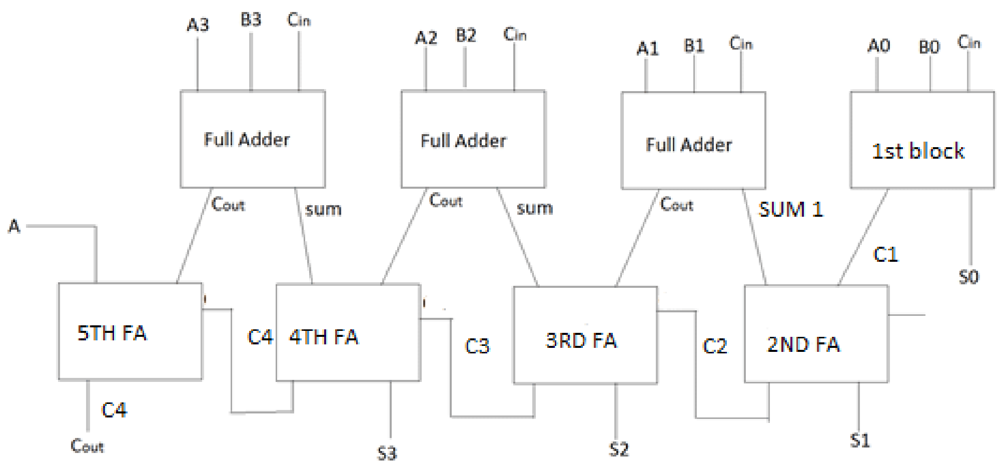


$A_3$ $B_3$ $A_2$ $B_2$ $A_1$ $B_1$ $A_0$ $B_0$
Full Adder
Full Adder
Full Adder
Full Adder
$C_3$
$C_2$
$C_1$
$C_0$
$S_3$
$S_2$
$S_1$
$S_0$
$C_4$
Carry Lookahead Logic
(Generate, Propagate, Carry)
$A_3$ $B_3$ $A_2$ $B_2$ $A_1$ $B_1$ $A_0$ $B_0$

This type of adders uses more area and they have more complex circuit than any other adder.

**v) Carry save adder:**

A Carry-Save Adder is a two-stage adder that separates the addition process into a carry-save stage and a carry-propagate stage. It is designed for high-speed addition operations and is particularly well-suited for applications where computational speed is critical.



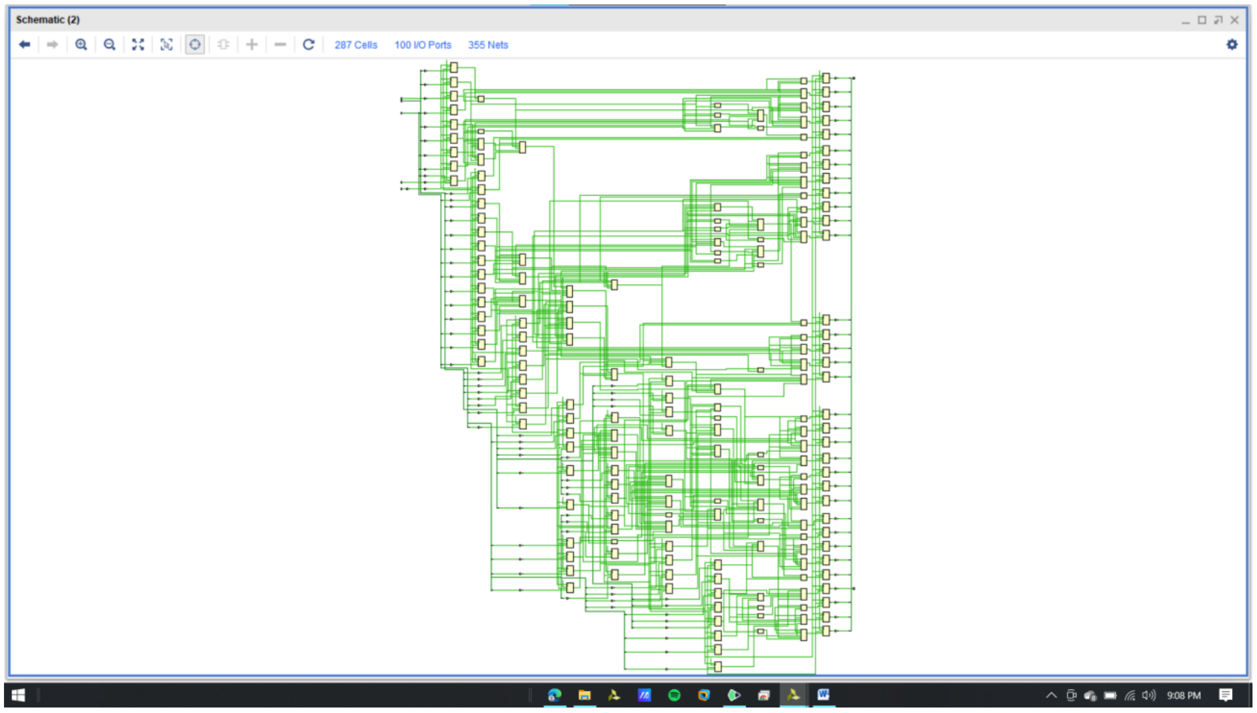
# Implementation of 32-bit Carry Look-Ahead Adder

- In the implementation of 32-bit adder, I have created a 4-bit CLA in which it generates all the carries independently using generate carry and propagation carries.
- Where generation carry is given by performing and operation of all the bits of the inputs **(A AND B).**
- The propagation carry is given by performing xor operation of all the bits of the inputs **(A XOR B).**
- The generation and propagation carries can be used to generate carry independently. There is a formula for this and the formula is

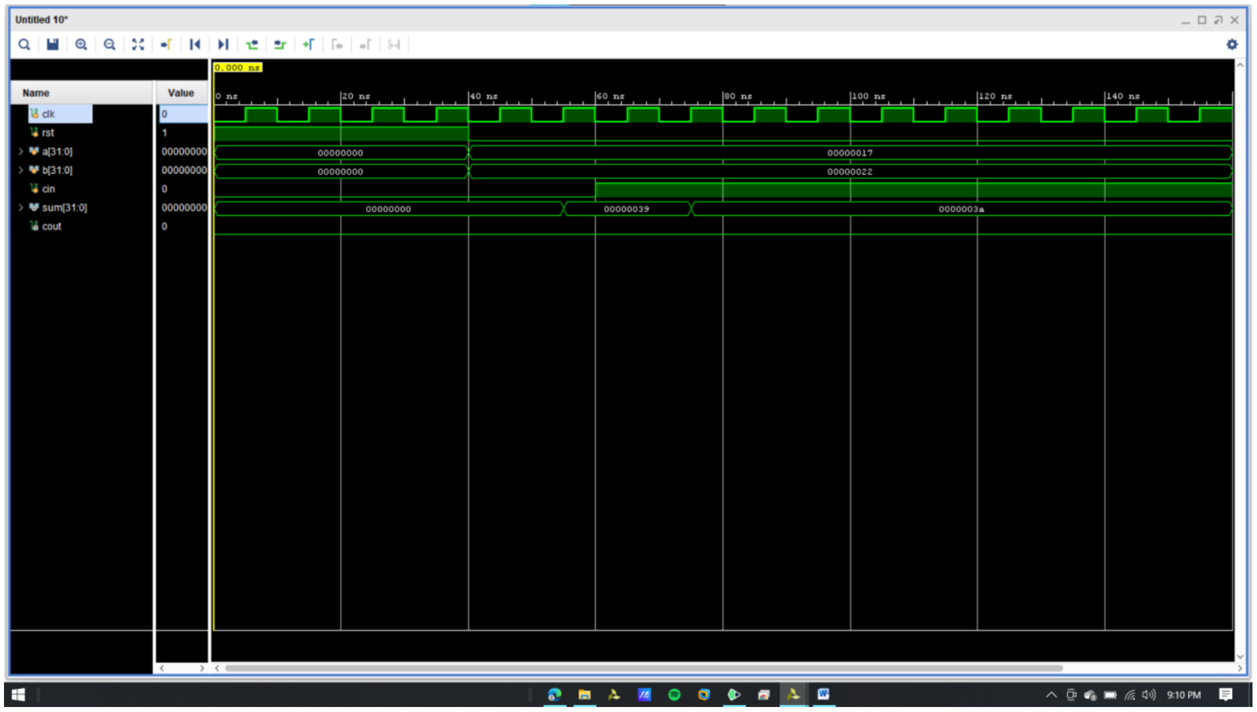
**c[i] = g[i] | (p[i] &Cin)**

- By this all the carries are generated and by performing **A xor B xor c** we will get the sum of the inputted numbers. To get the final carry out we should perform c[4] and the result is assigned to the carry output (cout).
- By instantiating this 4-bit CLA for 4 times I have created a 16- bit CLA.
- Using 16-bit CLA for another two times I have implemented 32-bit carry look-Ahead adder.

**1) Schematic:**



## 2) Waveform:

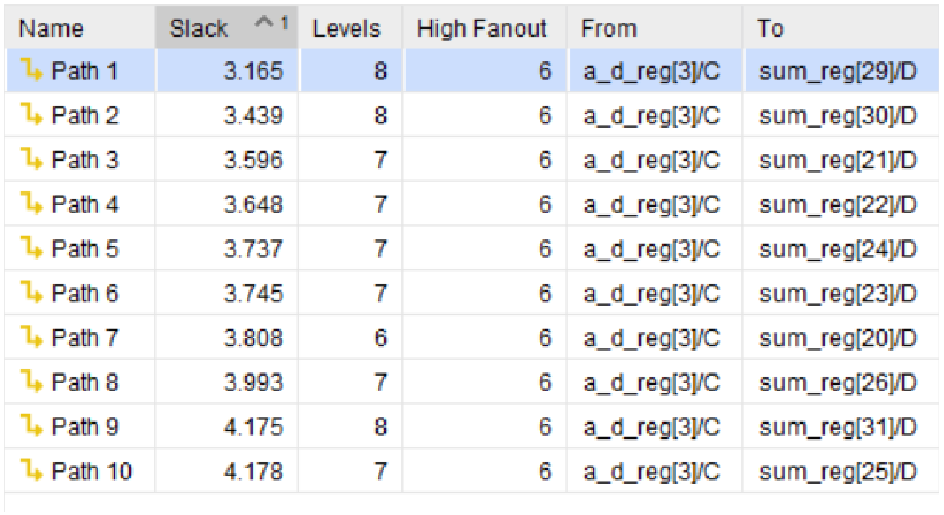


## 3) Timing report:

The clock used in this design has a time period of 10ns.

The total number of timing paths in this design is 20, in which 10 paths are setup timing paths and another 10 paths are hold timing paths.

The below table 10 setup timing paths, it contain source flip flop, destination flip flop and slack value of that respective path.



| Name | Slack ^1 | Levels | High Fanout | From | To |
|---|---|---|---|---|---|
| Path 1 | 3.165 | 8 | 6 | a_d_reg[3]/C | sum_reg[29]/D |
| Path 2 | 3.439 | 8 | 6 | a_d_reg[3]/C | sum_reg[30]/D |
| Path 3 | 3.596 | 7 | 6 | a_d_reg[3]/C | sum_reg[21]/D |
| Path 4 | 3.648 | 7 | 6 | a_d_reg[3]/C | sum_reg[22]/D |
| Path 5 | 3.737 | 7 | 6 | a_d_reg[3]/C | sum_reg[24]/D |
| Path 6 | 3.745 | 7 | 6 | a_d_reg[3]/C | sum_reg[23]/D |
| Path 7 | 3.808 | 6 | 6 | a_d_reg[3]/C | sum_reg[20]/D |
| Path 8 | 3.993 | 7 | 6 | a_d_reg[3]/C | sum_reg[26]/D |
| Path 9 | 4.175 | 8 | 6 | a_d_reg[3]/C | sum_reg[31]/D |
| Path 10 | 4.178 | 7 | 6 | a_d_reg[3]/C | sum_reg[25]/D |

As mentioned above there are different values of slacks for different paths. The path which is having the lowest value of slack will be the critical path, because the value of slack is the time that is left out with us and if this time is less it means that the time taken by this path is more which means it the critical path.

## Critical Path:

Source(Launch FF) : a_r_reg[3]/C

Destination (Capture FF) : sum_reg[29]/D

Source clock path: clock clk rise edge + net(0.0ns) + buffer(0.938ns) + net(1.967ns) + BUFG(0.096ns) + net(1.626ns) **= 4.626ns**

Data path: FDCE (0.456ns) + net(0.150ns) + LUT2(0,15ns) + LUT6(0.326ns) + LUT6(0.124ns) + LUT6(0.124ns) + LUT6(0.124ns) + LUT6(0.124ns) + LUT5(0.124ns) + LUT3(0.15ns)

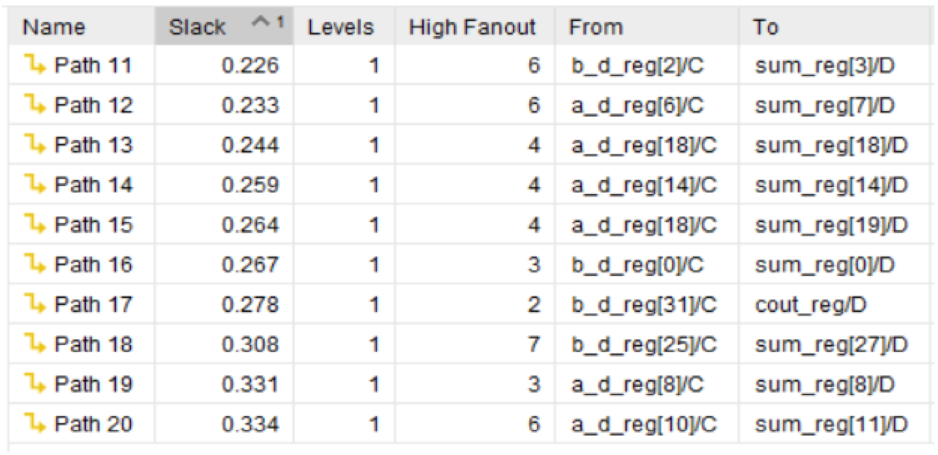
**= 6.554ns**

Destination clock path: clock rise edge (10ns) + net(0ns) + IBUF(0.804ns) + net(1.862ns) + BUFG(0.091ns) + net(1.520ns) + clock pessimism(0.335ns) + clock uncertainty(-0.035ns) + FDCE(-0.232ns) **= 14.345ns**

**Arrival Time = source clock path + Data path**
**= 4.626 + 6.554**
**= 11.180ns.**

**Required Time = Destination clock path**
**= 14.345ns.**

**Slack = Required time – Arrival time**
**= 14.345 – 11.180**
**= 3.165ns.**



| Name | Slack ^1 | Levels | High Fanout | From | To |
|---|---|---|---|---|---|
| Path 11 | 0.226 | 1 | 6 | b_d_reg[2]/C | sum_reg[3]/D |
| Path 12 | 0.233 | 1 | 6 | a_d_reg[6]/C | sum_reg[7]/D |
| Path 13 | 0.244 | 1 | 4 | a_d_reg[18]/C | sum_reg[18]/D |
| Path 14 | 0.259 | 1 | 4 | a_d_reg[14]/C | sum_reg[14]/D |
| Path 15 | 0.264 | 1 | 4 | a_d_reg[18]/C | sum_reg[19]/D |
| Path 16 | 0.267 | 1 | 3 | b_d_reg[0]/C | sum_reg[0]/D |
| Path 17 | 0.278 | 1 | 2 | b_d_reg[31]/C | cout_reg/D |
| Path 18 | 0.308 | 1 | 7 | b_d_reg[25]/C | sum_reg[27]/D |
| Path 19 | 0.331 | 1 | 3 | a_d_reg[8]/C | sum_reg[8]/D |
| Path 20 | 0.334 | 1 | 6 | a_d_reg[10]/C | sum_reg[11]/D |

## Improvements in speed:

- **<u>Pipelined</u>**: Speed can be improved by implementing pipeline stages in the CLA adder to break down the addition into multiple stages. This reduces the critical path length and allows for higher clock frequencies. Their frequencies should be in the ration

$$\frac{\textbf{frequenc of normal}}{\textbf{frequenc of pipelined}} = \frac{\textbf{no of clocks for result of normal}}{\textbf{no of clocks for result for pipelined}}$$

- **<u>Hierarchical Design:</u>** Use a hierarchical CLA structure, where smaller CLA blocks are combined to form a larger adder. This reduces the complexity of each CLA block and can improve performance.

## Conclusion:

As mentioned above I have designed 4-bit Carry Look-Ahead adder and cascaded them four times to form a 16-bit adder and cascaded this 16-bit adder twice to form 32-bit adder. From the overall point of view it looks as a 32-bit ripple adder with 8 times instantiation of 4bit carry look-ahead adder. Propagation delay of this adder is far better than propagation delay of 32-bit ripple carry adder.