

Design and implement 16-bit carry look ahead adder using 4-bit carry look ahead adder.

A Major Project Report

Submitted by:

SL.NO	NAME	ROLLNO
1.	Sappidi Nischinth Reddy	CH.EN.U4AIE20057
2.	Bhumaraju Mani teja	CH.EN.U4AIE20007
3.	Bhagavatula Yogiraj	CH.EN.U4AIE20005
4.	Pendem Mukhtesh Venkata Sri Sai	CH.EN.U4AIE20047
5.	Rohith N D	CH.EN.U4AIE20054

**In partial fulfilment for the award of the degree
of
Bachelor of Technology
In
Elements of computing systems(19AIE101)**



Amrita School of Engineering, Chennai
Amrita Vishwa Vidyapeetham
Chennai – 601 103,
TamilNadu, India.

Amrita School of Engineering
Amrita Vishwa Vidyapeetham, Chennai – 601 103,



BONAFIDE CERTIFICATE

This is to certify that the major project report entitled “**Design and implement 16-bit carry look ahead adder using 4-bit carry look ahead adder**”.

submitted by:

- 1) Bhumaraju Maniteja
- 2) Bhagavatula Yogiraj
- 3) Sappidi Nischinth Reddy
- 4) Rohith N D
- 5) Pendem Mukhtesh Venkata Sri Sai

In partial fulfilment of the requirements for the award of the **Degree of Bachelor of Technology in Elements of computing systems** is a Bonafide record of the work carried out under my guidance and supervision at Amrita School of Engineering, Chennai.

Signature

Dr.Prasanna Kumar

Asst.Professor (Sr.Gr),ECS Dept

Dr.....

Chairperson, Dept. of ECS

This project report was evaluated by us on

INTERNAL EXAMINER

Signature

.....

Asst.Professor, ECS Dept

EXTERNAL EXAMINER

INDEX

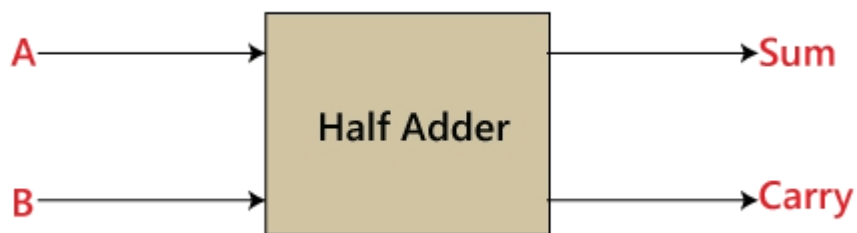
S. No	TOPIC	Page No.
1	Introduction	4
2	Half Adder	5
3	2-input Exclusive-OR Gate	6
4	2-input AND gate	7
5	Full Adder	8
6	4-Bit Carry Look-ahead Adder	11
7	Carry look ahead logic	12
8	16-Bit Carry Look-ahead Adder	13
9	HDL scripts	15
10	output	17
11	Output(no ripple carry propagation)	20

INTRODUCTION

Fast addition is an essential arithmetic function for most advanced digital systems. It heavily impacts the overall Performance of digital systems. Various adder structures can be used to execute addition, such as serial and parallel structures. Most research works of adders are focused on the design of high-speed, low-area, or low-power Adders. This project is aimed to study and implement 16-bit carry look ahead adder based on HDL. The purpose of using HDL is that it offers many features for Nand2tetris hard ware simulator. When the actual addition is performed, there is no delay from waiting by using carry look-ahead adder. For any circuit larger than 4-bit, the carry look-ahead adder, the circuit becomes very complicated. So 16-bit carry look-ahead adder will be constructed by combining 4-bit carry look-ahead adders.

Half Adder

The Half-Adder is a basic building block of adding two numbers as two inputs and produce out two outputs. The adder is used to perform OR operation of two single bit binary numbers. The A and B are two input states and '**carry**' and '**sum**' are two output states of the 0half adder.



Truth Table:

Inputs		Outputs	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

In the above table,

1. 'A' and 'B' are the input states, and 'sum' and 'carry' are the output states.
2. The carry output is 0 in case where both the inputs are not 1.

The SOP form of the sum and carry are as follows:

$$\text{Sum} = x'y + xy'$$

$$\text{Carry} = xy$$

Construction of Half Adder Circuit:

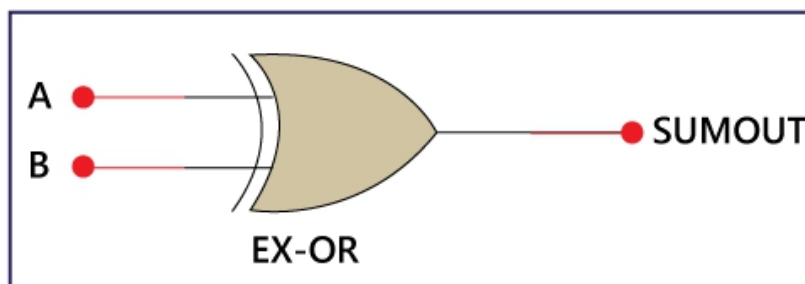
The half adder is designed with the help of the following two logic gates:

1.2-input AND Gate.

2.2-input Exclusive-OR Gate or Ex-OR Gate

2-input Exclusive-OR Gate or Ex-OR Gate:

The **Sum** bit is generated with the help of the **Exclusive-OR** or **Ex-OR** Gate.



The above is the symbol of the **EX-OR** gate. In the above diagram, 'A' and 'B' are the inputs, and the 'SUMOUT' is the final outcome after performing the XOR operation of both numbers.

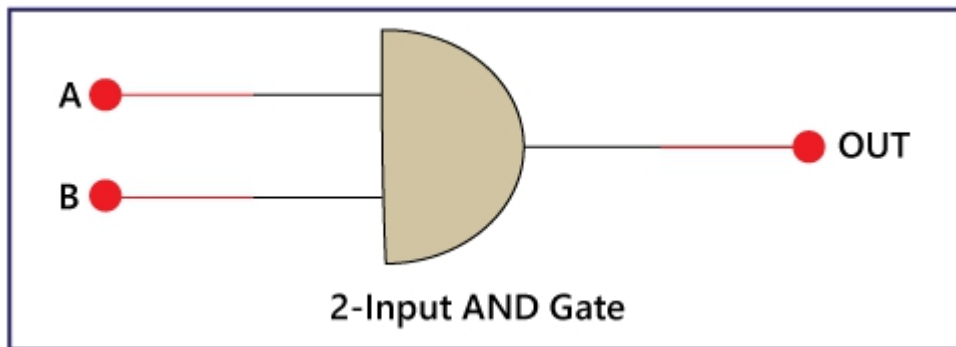
The truth table of the EX-OR gate is as follows:

Input		Output
A	B	SUMOUT
0	0	0
0	1	1
1	0	1
1	1	0

From the above table, it is clear that the **XOR gate** gives the result 1 when both of the inputs are different. When both of the inputs are the same, the XOR gives the result 0.

2-input AND Gate:

The XOR gate is unable to generate the carry bit. For this purpose, we use another gate called [AND Gate](#). The AND gate gives the correct result of the carry.



The above is the symbol of the **AND** gate. In the above diagram, 'A' and 'B' are the inputs, and 'OUT' is the final outcome after performing AND operation of both numbers.

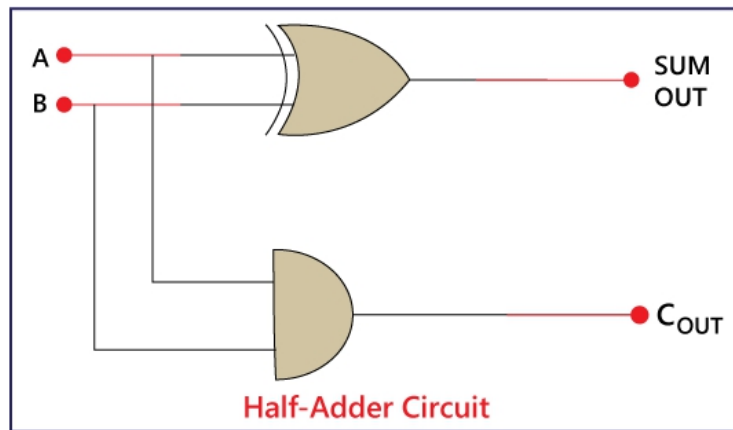
There is the following truth table of AND Gate:

Input		Output
A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1

From the above table, it is clear that the AND gate gives the result 1 when both of the inputs are 1. When both of the inputs are different and 0, the AND gates gives the result 0.

Half-Adder logical circuit:

So, the Half Adder is designed by combining the 'XOR' and 'AND' gates and provide the sum and carry.



There is the following **Boolean expression** of **Half Adder circuit**:

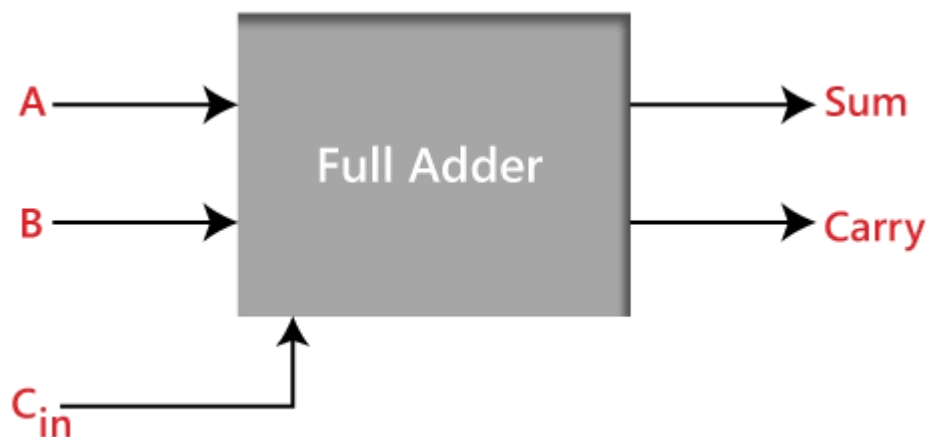
$$\text{Sum} = A \text{ XOR } B \text{ (A+B)}$$

$$\text{Carry} = A \text{ AND } B \text{ (A.B)}$$

Full Adder:

The half adder is used to add only two numbers. To overcome this problem, the full adder was developed. The full adder is used to add three 1-bit binary numbers A, B, and carry C. The full adder has three input states and two output states i.e., sum and carry.

Block diagram:



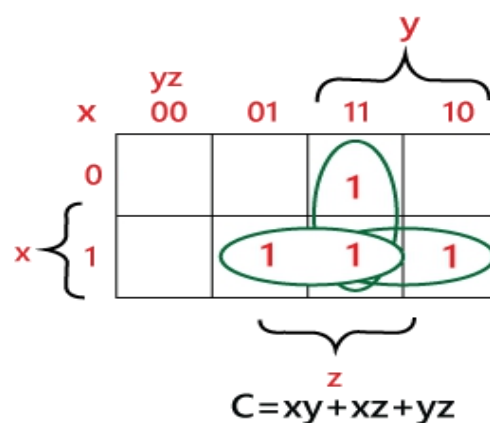
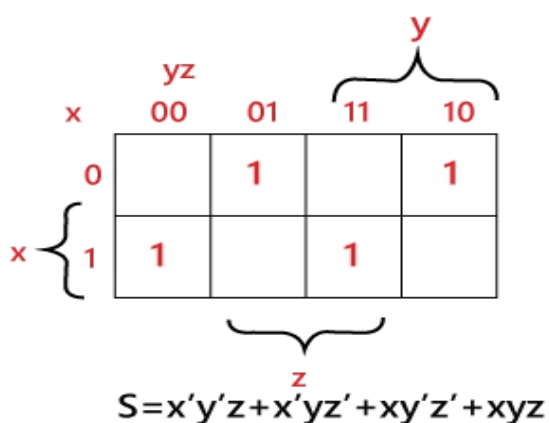
Truth Table

Inputs			Outputs	
A	B	C _{in}	Sum	Carry
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

In the above table,

1. 'A' and 'B' are the input variables. These variables represent the two significant bits which are going to be added
2. 'C_{in}' is the third input which represents the carry. From the previous lower significant position, the carry bit is fetched.
3. The 'Sum' and 'Carry' are the output variables that define the output values.
4. The eight rows under the input variable designate all possible combinations of 0 and 1 that can occur in these variables.

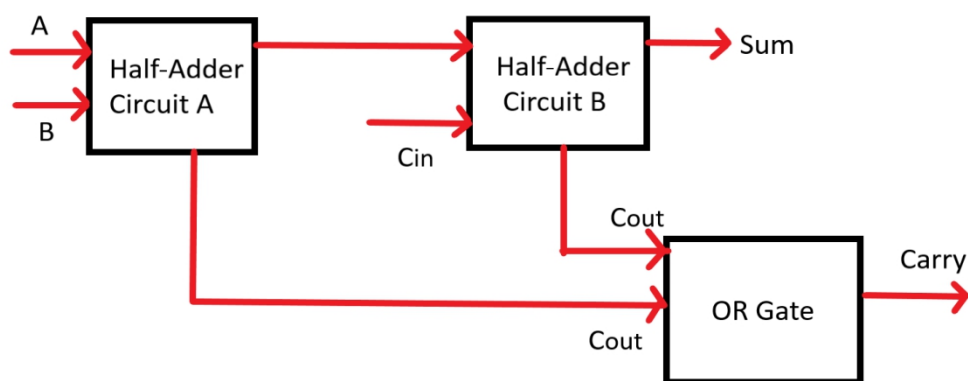
The SOP form can be obtained with the help of K-map as:



$$\text{Sum} = x' y' z + x' y z + x y' z' + x y z$$

$$\text{Carry} = xy + xz + yz$$

Construction of Full Adder Circuit:



The above block diagram describes the construction of the Full adder circuit. In the above circuit, there are two half adder circuits that are combined using the OR gate. The first half adder has two single-bit binary inputs A and B. As we know that, the half adder produces two outputs, i.e., Sum and Carry. The 'Sum' output of the first adder will be the first input of the second half adder, and the 'Carry' output of the first adder will be the second input of the second half adder. The second half adder will again provide 'Sum' and 'Carry'. The final outcome of the Full adder circuit is the 'Sum' bit. In order to find the final output of the 'Carry', we provide the 'Carry' output of the first and the second adder into the OR gate. The outcome of the OR gate will be the final carry out of the full adder circuit.

4-Bit Carry Look-ahead Adder

In this adder, the carry input at any stage of the adder is independent of the carry bits generated at the independent stages. Here the output of any stage is dependent only on the bits which are added in the previous stages and the carry input provided at the beginning

stage. Hence, the circuit at any stage does not have to wait for the generation of carry-bit from the previous stage and carry bit can be evaluated at any instant of time.

Truth Table of Carry Look-ahead Adder:

For deriving the truth table of this adder, two new terms are introduced – Carry generate and carry propagate. Carry generate $G_i = 1$ whenever there is a carry C_{i+1} generated. It depends on A_i and B_i inputs. G_i is 1 when both A_i and B_i are 1. Hence, G_i is calculated as $G_i = A_i \cdot B_i$.

Carry propagated P_i is associated with the propagation of carry from C_i to C_{i+1} . It is calculated as $P_i = A_i \oplus B_i$. The truth table of this adder can be derived from modifying the truth table of a full adder.

Using the G_i and P_i terms the Sum S_i and Carry C_{i+1} are given as below –

- $S_i = P_i \oplus G_i$.
- $C_{i+1} = C_i \cdot P_i + G_i$.

Therefore, the carry bits C_1 , C_2 , C_3 , and C_4 can be calculated as

- $C_1 = C_0 \cdot P_0 + G_0$.
- $C_2 = C_1 \cdot P_1 + G_1 = (C_0 \cdot P_0 + G_0) \cdot P_1 + G_1$.
- $C_3 = C_2 \cdot P_2 + G_2 = (C_1 \cdot P_1 + G_1) \cdot P_2 + G_2$.
- $C_4 = C_3 \cdot P_3 + G_3 = C_0 \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot G_1 + G_2 \cdot P_3 + G_3$.

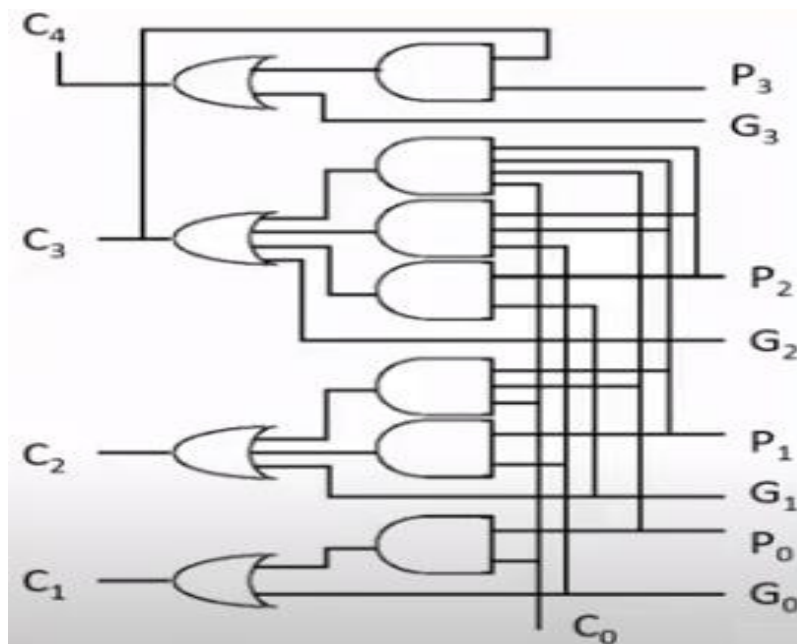
It can be observed from the equations that carry C_{i+1} only depends on the carry C_0 , not on the intermediate carry bits.

A	B	C_i	C_{i+1}	Condition
0	0	0	0	No carry generate
0	0	1	0	
0	1	0	0	
0	1	1	1	No carry propagate
1	0	0	0	
1	0	1	1	
1	1	0	1	Carry generate
1	1	1	1	

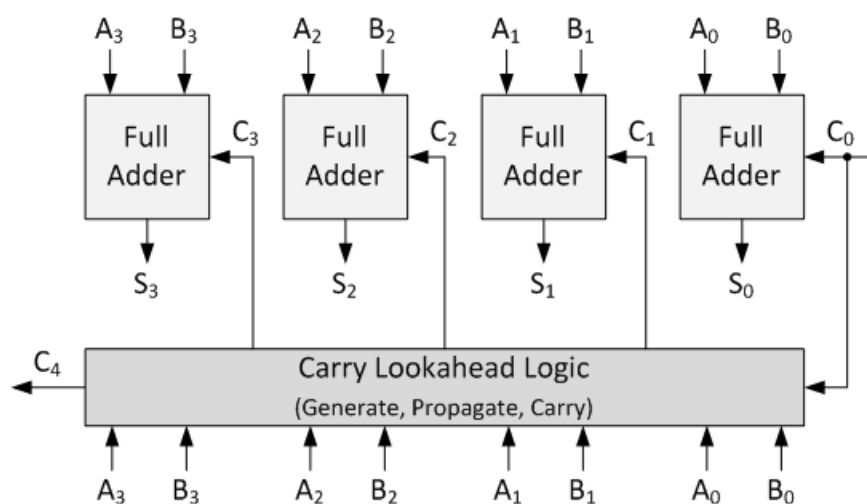
Circuit Diagram

The above equations are implemented using two-level combinational circuits along with AND, OR gates, where gates are assumed to have multiple inputs.

Carry Look Ahead Logic



The Carry Look-ahead Adder circuit from 4-bit is given below.



16-bit Carry Look-ahead Adder circuits can be designed by cascading the 4-bit adder circuit with carry logic.

16 Bit Carry Look Ahead Adder:

Carry bits are generated by a look ahead carry unit as follows:

$$c_0 = \text{input carry}$$

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 c_1 = g_1 + p_1(g_0 + p_0 c_0) = g_1 + p_1 g_0 + p_1 p_0 c_0$$

$$c_3 = g_2 + p_2 c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$

$$c_4 = g_3 + p_3 c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$$

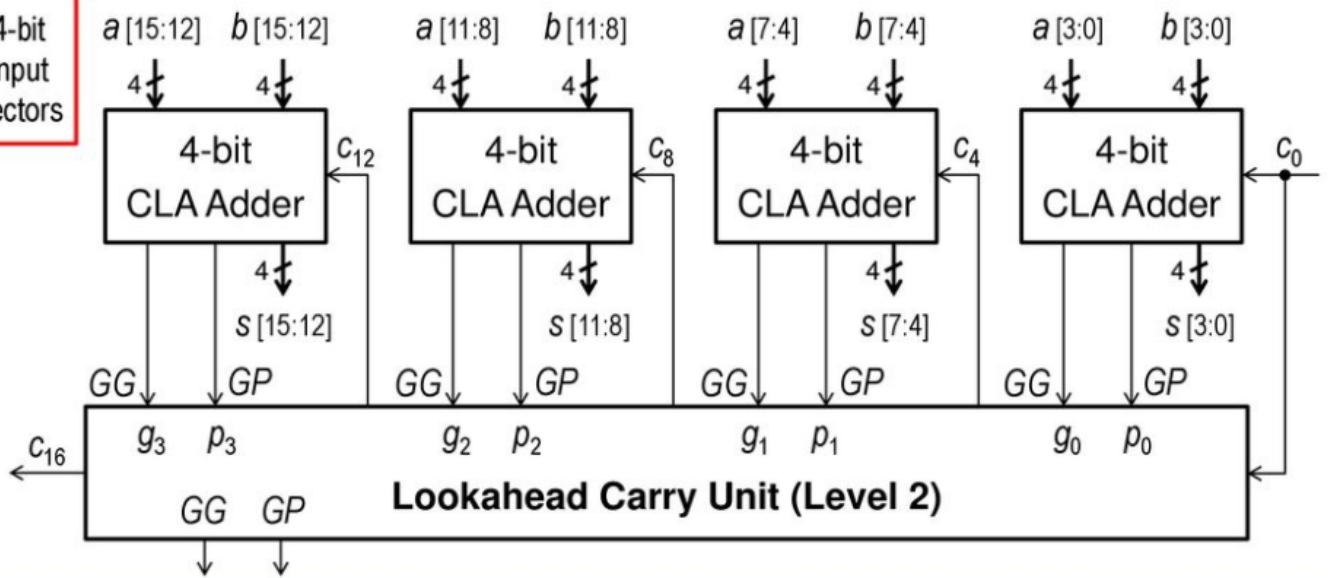
$$\text{Group generate: } GG = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$$

$$\text{Group propagate: } GP = p_3 p_2 p_1 p_0$$

$$c_4 = GG + GP c_0$$

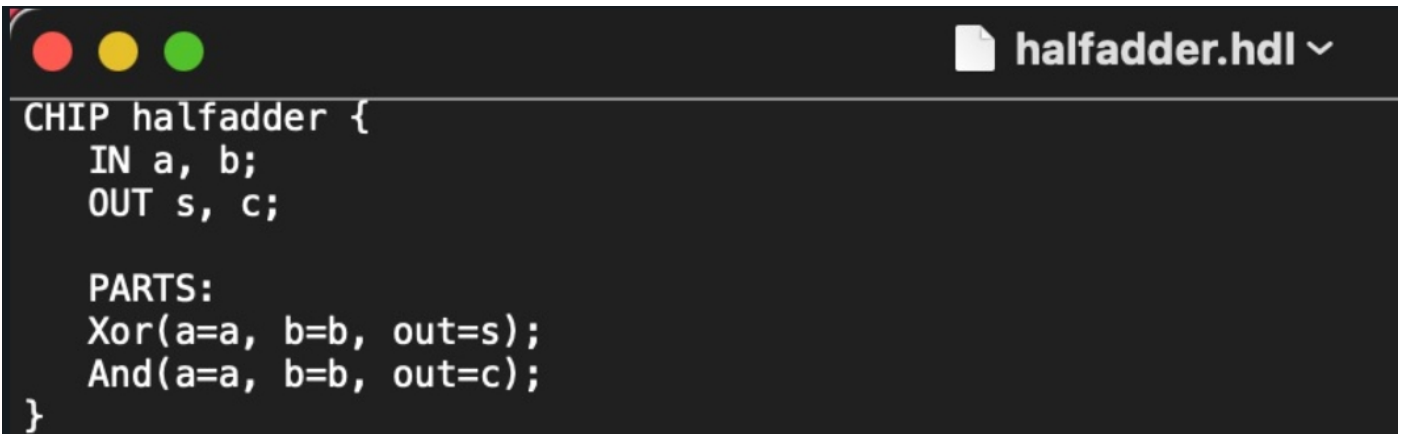
- **Carry does not ripple anymore**
- g_i is called **carry generate** : $g_i = a_i \cdot b_i$ [generates c_{i+1} regardless of c_i]
- In addition, define $p_i = (a_i \oplus b_i)$ [a_i or b_i is 1, not both]
- p_i is called **carry propagate**: propagates values of c_i to c_{i+1}
- Equation of output carry becomes: $c_{i+1} = g_i + p_i c_i$
- Observation: $c_{i+1} = a_i b_i + (a_i \oplus b_i) c_i$

4-bit
input
vectors



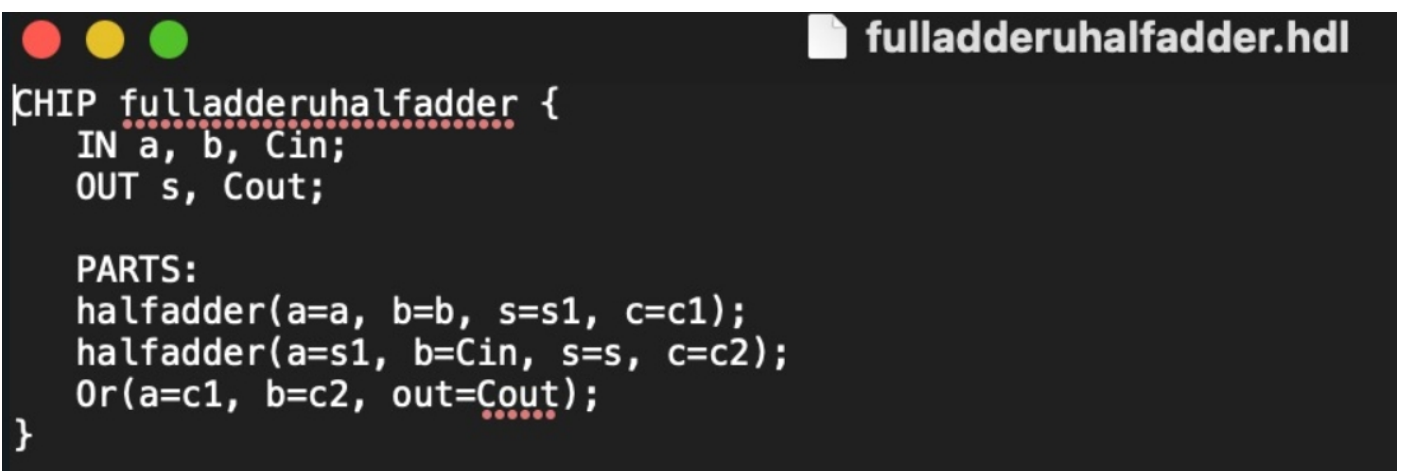
HDL SCRIPTS

HALF ADDER:




```
CHIP halfadder {  
    IN a, b;  
    OUT s, c;  
  
    PARTS:  
    Xor(a=a, b=b, out=s);  
    And(a=a, b=b, out=c);  
}
```

FULL ADDER USING HALF ADDER:



```
CHIP fulladderhalfadder {  
    IN a, b, Cin;  
    OUT s, Cout;  
  
    PARTS:  
    halfadder(a=a, b=b, s=s1, c=c1);  
    halfadder(a=s1, b=Cin, s=s, c=c2);  
    Or(a=c1, b=c2, out=Cout);  
}
```


4-BIT CARRY LOOK AHEAD ADDER:



carrylookadder.hdl

```
CHIP carrylookadder {  
    IN a[4], b[4], c0;  
    OUT s[4], c5;  
  
    PARTS:  
        fulladderhalfadder(a=a[0], b=b[0], Cin=c0, s=s[0], Cout=c1);  
  
        Xor(a=a[0], b=b[0], out=x1);  
        And(a=x1, b=c1, out=y1);  
        And(a=a[0], b=b[0], out=z1);  
        Or(a=y1, b=z1, out=c2);  
  
        fulladderhalfadder(a=a[1], b=b[1], Cin=c2, s=s[1]);  
  
        Xor(a=a[1], b=b[1], out=x2);  
        And(a=x2, b=c2, out=y2);  
        And(a=a[1], b=b[1], out=z2);  
        Or(a=y2, b=z2, out=c3);  
  
        fulladderhalfadder(a=a[2], b=b[2], Cin=c3, s=s[2]);  
  
        Xor(a=a[2], b=b[2], out=x3);  
        And(a=x3, b=c3, out=y3);  
        And(a=a[2], b=b[2], out=z3);  
        Or(a=y3, b=z3, out=c4);  
  
        fulladderhalfadder(a=a[3], b=b[3], Cin=c4, s=s[3], Cout=c5);  
}
```

16-BIT CARRY LOOK AHEAD ADDER USING 4-BIT CARRY LOOK AHEAD ADDER:



cla16ucla4.hdl

```
CHIP cla16ucla4 {  
    IN a[16], b[16], c0;  
    OUT s[16], c1;  
  
    PARTS:  
        carrylookadder(a=a[0..3], b=b[0..3], c0=false, s=s[0..3], c5=x1);  
        carrylookadder(a=a[4..7], b=b[4..7], c0=x1, s=s[4..7], c5=x2);  
        carrylookadder(a=a[8..11], b=b[8..11], c0=x2, s=s[8..11], c5=x3);  
        carrylookadder(a=a[12..15], b=b[12..15], c0=x3, s=s[12..15], c5=c1);  
}
```


IMPLEMENTATION IN HARDWARE SIMULATOR:

Chip Nam... cla16ucla4		Time : 0	
Input pins		Output pins	
Name	Value	Name	Value
a[16]	5243	s[16]	6488
b[16]	1245	c1	0
c0	0		
HDL		Internal pins	
<pre>CHIP cla16ucla4 { IN a[16], b[16], c0; OUT s[16], c1; PARTS: carrylookadder(a=a[0..3], b=b[0..3], c=c0, s=s[0..3], c1=c1); carrylookadder(a=a[4..7], b=b[4..7], c=c1, s=s[4..7], c1=c1); carrylookadder(a=a[8..11], b=b[8..11], c=c1, s=s[8..11], c1=c1); carrylookadder(a=a[12..15], b=b[12..15], c=c1, s=s[12..15], c1=c1); }</pre>		Name	Value
		x1	1
		x2	1
		x3	0

NO RIPPLE CARRY PROPAGATION IN 16-BIT CARRY LOOK AHEAD ADDER:

```
CHIP cla4Bit10 {
    IN a[4], b[4], c0;
    OUT gg, gp, sum[4];

    PARTS:
    Xor(a=a[0], b=b[0], out=p0);
    And(a=a[0], b=b[0], out=g0);

    Xor(a=a[1], b=b[1], out=p1);
    And(a=a[1], b=b[1], out=g1);

    Xor(a=a[2], b=b[2], out=p2);
    And(a=a[2], b=b[2], out=g2);

    Xor(a=a[3], b=b[3], out=p3);
    And(a=a[3], b=b[3], out=g3);

    And(a=p0, b=p1, out=x1);
    And(a=p2, b=p3, out=x2);
    And(a=x1, b=x2, out=gp); // For Group Propagation, we can also use And4Input

    And(a=g0, b=p1, out=x3);
    And(a=p2, b=p3, out=x4);
    And(a=x3, b=x4, out=y1);

    And(a=g1, b=p2, out=x5);
    And(a=x5, b=p3, out=y2);

    And(a=g2, b=p3, out=y3);

    Or(a=g3, b=y3, out=z1);
    Or(a=y2, b=y1, out=z2);
    Or(a=z1, b=z2, out=gg); // For Group Generation

    // For finding out the sum of 2 4-bit numbers
    fulladderuhalfadder(a=a[0], b=b[0], Cin=c0, s=sum[0], Cout=c1);

    Xor(a=a[0], b=b[0], out=m1);
    And(a=m1, b=c1, out=n1);
    And(a=a[0], b=b[0], out=e1);
    Or(a=n1, b=e1, out=c2);

    fulladderuhalfadder(a=a[1], b=b[1], Cin=c2, s=sum[1]);

    Xor(a=a[1], b=b[1], out=m2);
    And(a=m2, b=c2, out=n2);
    And(a=a[1], b=b[1], out=e2);
    Or(a=n2, b=e2, out=c3);

    fulladderuhalfadder(a=a[2], b=b[2], Cin=c3, s=sum[2]);

    Xor(a=a[2], b=b[2], out=m3);
    And(a=m3, b=c3, out=n3);
    And(a=a[2], b=b[2], out=e3);
    Or(a=n3, b=e3, out=c4);

    fulladderuhalfadder(a=a[3], b=b[3], Cin=c4, s=sum[3], Cout=c5);
}
```



```
CHIP cla16bit11 {
  IN a[16], b[16], c0;
  OUT sum[16], Cout;

  PARTS:
  cla4Bit10(a=a[0..3], b=b[0..3], c0=c0, gg=g0, gp=p0, sum=sum[0..3]);
  And(a=c0, b=p0, out=q1);
  Or(a=q1, b=g0, out=c4);

  cla4Bit10(a=a[4..7], b=b[4..7], c0=c4, gg=g1, gp=p1, sum=sum[4..7]);
  And(a=c4, b=p1, out=q2);
  Or(a=q2, b=g1, out=c8);

  cla4Bit10(a=a[8..11], b=b[8..11], c0=c8, gg=g2, gp=p2, sum=sum[8..11]);
  And(a=c8, b=p2, out=q3);
  Or(a=q3, b=g2, out=c12);

  cla4Bit10(a=a[12..15], b=b[12..15], c0=c12, gg=g3, gp=p3, sum=sum[12..15]);
  And(a=c12, b=p3, out=q4);
  Or(a=q4, b=g3, out=Cout);
}
```

TST FILE:



```
[load cla16bit11.hdl,
output-file cla16bit11.out,
compare-to cla16bit11.cmp,
output-list a%B1.16.1 b%B1.16.1 sum%B1.16.1;

set a %B0000000000000000,
set b %B0000000000000000,
eval,
output;

set a %B0000000000000000,
set b %B1111111111111111,
eval,
output;

set a %B1111111111111111,
set b %B1111111111111111,
eval,
output;

set a %B1010101010101010,
set b %B0101010101010101,
eval,
output;

set a %B00111110011000011,
set b %B0000111111111000,
eval,
output;
```

IMPLEMENTATION IN HARDWARE SIMULATOR:

Chip Nam... cla16bit11		Time : 0	
Input pins		Output pins	
Name	Value	Name	Value
a[16]	4352	sum[16]	5475
b[16]	1123	Cout	0
c0	0		