

# **21CSS201T COMPUTER ORGANIZATION AND ARCHITECTURE**

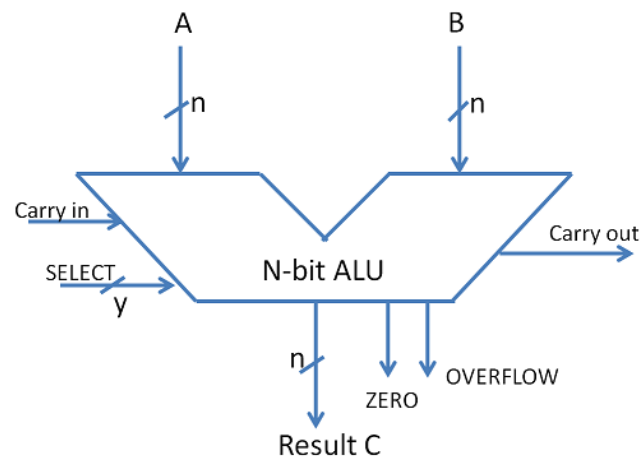
## **UNIT-3 Design of ALU and De Morgan's Law**

# Arithmetic Logic Unit

- ALU is the heart of any Central Processing Unit.
- A simple ALU is constructed with Combinational circuits.
- It is a digital circuit to do arithmetic operations like addition, subtraction, multiplication and division.
- Logical operations such as OR, AND, NOR etc.
- Data movement operations such as LOAD and STORE.
- Complex ALUs are designed for executing Floating point, decimal operations and other complex numerical operations. These are called as Co-processor and work in tandem with the main processor.
- The design specifications of ALU are derived from the Instruction set Architecture.
- The ALU must have the capabilities to execute the instructions of ISA.
- Modern CPUs have multiple ALU to improve the efficiency.

- ALU includes the following Configurations :
  - Instruction Set Architecture
  - Accumulator
  - Stack
  - Register – Register architecture
  - Register – Stack architecture
  - Register – memory architecture
- The size of input quantities of ALU is referred as word length of a computer

# ALU Symbol



- The basic building blocks of an ALU in digital computers are Adders.
- Types of Basic Adders are
  - Half Adder
  - Full Adder
- Parallel adders are nothing but cascade of full adders. The number of full adders used will depend on the number of bits in the binary digit which require to be added.
- Ripple carry adders are used when the input sequence is large. It is used to add two n-bit binary numbers.
- Carry look ahead adder is an improved version of Ripple carry adder.  
It generates the carry-in of each full adder simultaneously without causing any delay.

# Half Adder

- Half adder is a combinational logic circuit with two input and two output. The half adder circuit is designed to add two single bit binary number A and B. It is the basic building block for addition of two single bit numbers. This circuit has two outputs, carry and sum.



# Truth Table and Circuit Diagram

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

Truth table

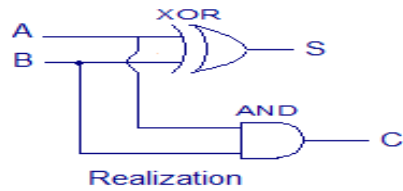
$$\text{SUM } S = A.\bar{B} + \bar{A}.B$$

$$\text{CARRY } C = A.B$$

Boolean Expression



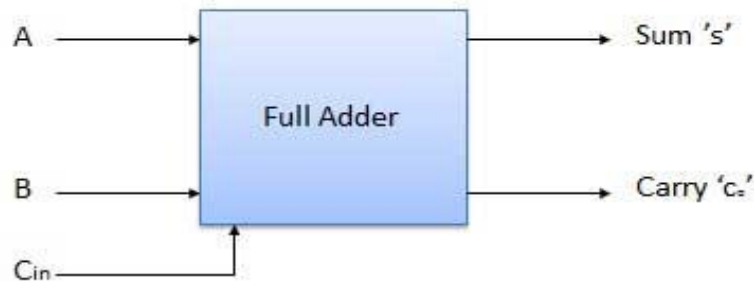
Schematic



Realization

# Full Adder

- Full adder is developed to overcome the drawback of Half Adder circuit. It can add two one-bit numbers A and B, and carry c. The full adder is a three input and two output combinational circuit.



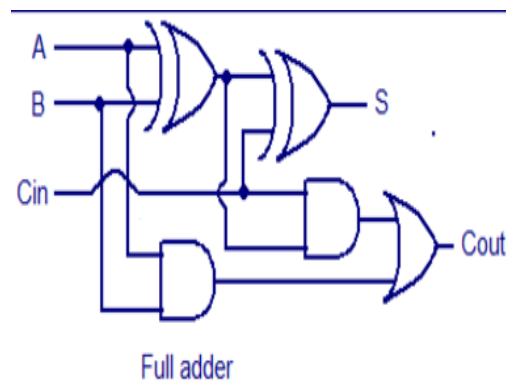


# Truth Table and Circuit Diagram

$$\begin{aligned}S &= \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + \overline{A}BC \\&= C(\overline{A}B + \overline{A}\overline{B}) + \overline{C}(\overline{A}B + A\overline{B}) \\&= C(\overline{A}(\overline{B} + B)) + \overline{C}(\overline{A}B + A\overline{B}) \\&= C(\overline{A}) + \overline{C}(A \oplus B) = A \oplus B \oplus C\end{aligned}$$

$$\begin{aligned}C_{out} &= \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC \\&= (\overline{A}B + AB)\overline{C} + AB(C + C) \\&= (A \oplus B) \cdot \overline{C} + AB\end{aligned}$$

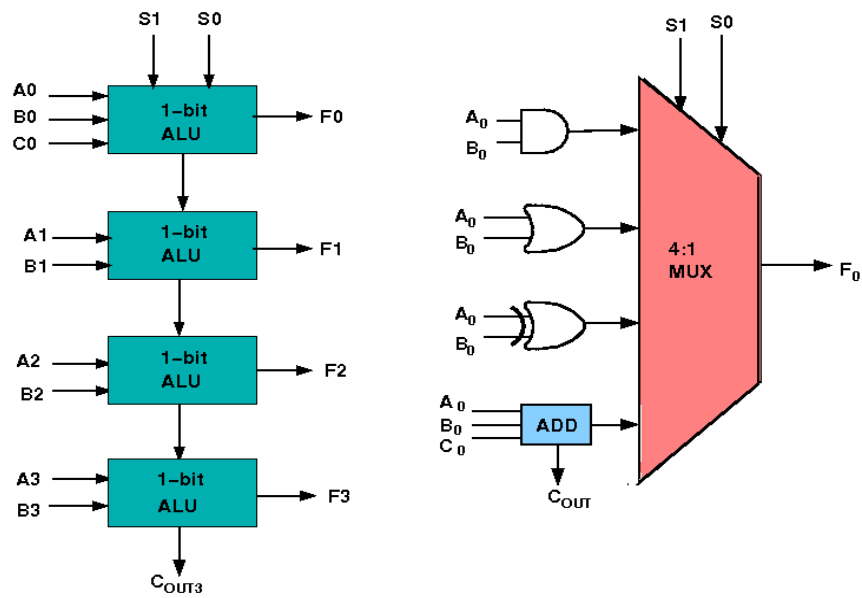
Inputs			Output	
A	B	C <sub>in</sub>	S	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



# Design of ALU

- ALU or Arithmetic Logical Unit is a digital circuit to do arithmetic operations like addition, subtraction, division, multiplication and logical operations like AND, OR, XOR, NAND, NOR etc.
- A simple block diagram of a 4 bit ALU for operations AND, OR, XOR and ADD is shown here :

The 4-bit ALU block is combined using 4 1-bit ALU block



# Design Issues

The circuit functionality of a 1 bit ALU is shown here, depending upon the control signal  $S_1$  and  $S_0$  the circuit operates as follows:

for Control signal  $S_1 = 0$  ,  $S_0 = 0$ , the output is **A And B**,

for Control signal  $S_1 = 0$  ,  $S_0 = 1$ , the output is **A Or B**,

for Control signal  $S_1 = 1$  ,  $S_0 = 0$ , the output is **A Xor B**,

for Control signal  $S_1 = 1$  ,  $S_0 = 1$ , the output is **A Add B**.

# 16 bit ALU Design

Design the 16-bit Arithmetic Logic Unit (ALU) shown in Figure 1. The function table of the ALU is shown in Table 1.

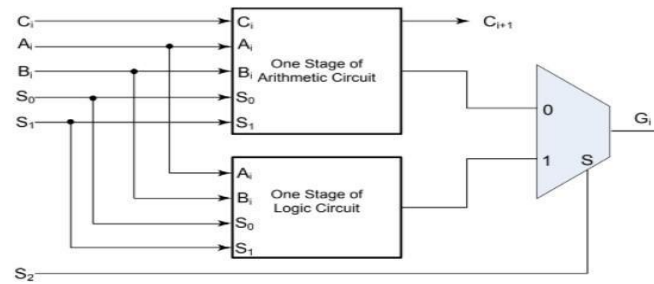


Figure 1: 16-bit ALU

Table 1: Function table of the ALU

Operation Select				Operation	Function
S <sub>2</sub>	S <sub>1</sub>	S <sub>0</sub>	C <sub>in</sub>		
0	0	0	0	$G = A$	Transfer $A$
0	0	0	1	$G = A + 1$	Increment $A$
0	0	1	0	$G = A + B$	Addition
0	0	1	1	$G = A + B + 1$	Add with carry input of 1
0	1	0	0	$G = A + \overline{B}$	$A$ plus 1's complement of $B$
0	1	0	1	$G = A + \overline{B} + 1$	Subtraction
0	1	1	0	$G = A - 1$	Decrement $A$
0	1	1	1	$G = A$	Transfer $A$
1	0	0	X	$G = A \wedge B$	AND
1	0	1	X	$G = A \vee B$	OR
1	1	0	X	$G = A \oplus B$	XOR
1	1	1	X	$G = \overline{A}$	NOT (1's complement)

# De Morgan's Theorem

- It is a very powerful tool used in digital design
- De Morgan's theorem are used to solve the expressions of Boolean Algebra.
- The theorem explains that the complement of the product of all the terms is equal to the sum of the complement of each term. Likewise, the complement of the sum of all the terms is equal to the product of the complement of each term.

# Two Theorems

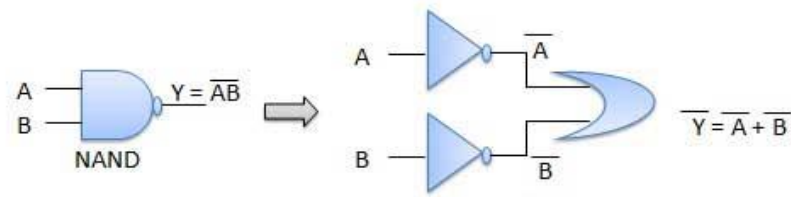
- Theorem1

$$\overline{A \cdot B} = \overline{A} + \overline{B}$$

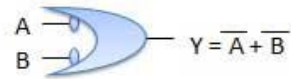
NAND = Bubbled OR

- The left hand side (LHS) of this theorem represents a NAND gate with inputs A and B, whereas the right hand side (RHS) of the theorem represents an OR gate with inverted inputs.
- This OR gate is called as **Bubbled OR**.

# Theorem1 – Logic diagram



NAND  $\equiv$  Bubbled OR



Bubbled OR



Table showing verification of the De Morgan's first theorem

A	B	$\overline{AB}$	$\overline{A}$	$\overline{B}$	$\overline{A} + \overline{B}$
0	0	1	1	1	1
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	0	0	0

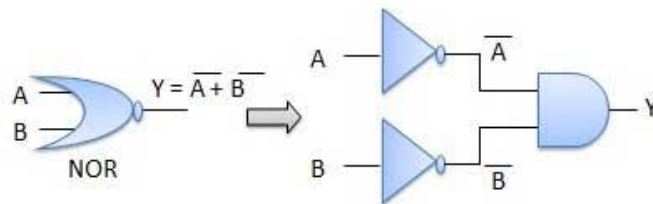
# Theorem 2

$$\overline{A + B} = \overline{A} \cdot \overline{B}$$

NOR = Bubbled AND

- The LHS of this theorem represents a NOR gate with inputs A and B, whereas the RHS represents an AND gate with inverted inputs.
- This AND gate is called as **Bubbled AND**.

# Theorem 2- Logic diagram



NOR  $\equiv$  Bubbled AND

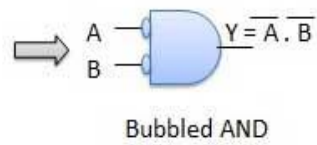


Table showing verification of the De Morgan's second theorem

A	B	$\overline{A+B}$	$\overline{A}$	$\overline{B}$	$\overline{A} \cdot \overline{B}$
0	0	1	1	1	1
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	0

# Applications of De Morgan's Theorem

- In the domain of engineering, using De Morgan's laws, Boolean expressions can be built easily only through one gate which is usually NAND or NOR gates. This results in hardware design at a cheaper cost.
- Used in the verification of SAS code.
- Implemented in computer and electrical engineering domain.
- De Morgan's laws are also employed in Java programming.

# Ripple Carry Adder



Typical Ripple Carry Addition is a Serial Process:

- Addition starts by adding LSBs of the augend and addend.
- Then next position bits of augend and addend are added along with the carry (if any) from the preceding bit.
- This process is repeated until the addition of MSBs is completed.
- Speed of a ripple adder is limited due to carry propagation or carry ripple.
- Sum of MSB depends on the carry generated by LSB.



# Ripple Carry Adder

## Example: 4-bit Carry Ripple Adder

- Assume to add two operands A and B where

A = A<sub>3</sub> A<sub>2</sub> A<sub>1</sub> A<sub>0</sub>

B = B<sub>3</sub> B<sub>2</sub> B<sub>1</sub> B<sub>0</sub>

A = 1 0 1 1 +

B = 1 1 0 1

-----

A+B = 1 1 0 0 0

Cout S<sub>3</sub> S<sub>2</sub> S<sub>1</sub> S<sub>0</sub>

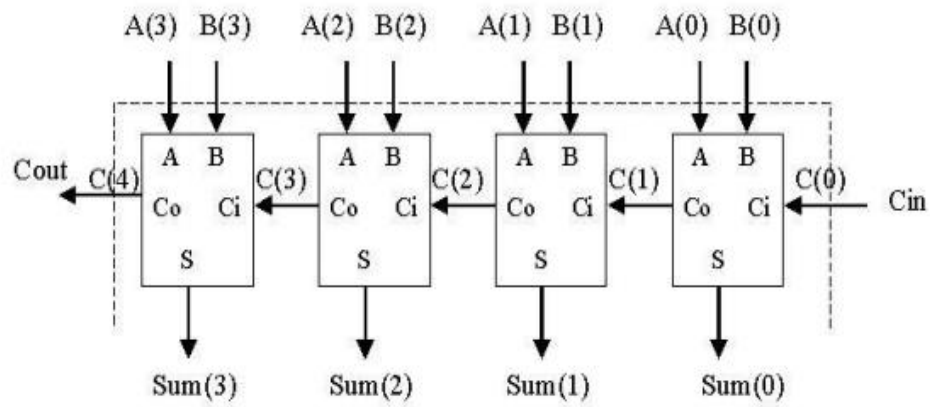
# Ripple Carry Adder



## Carry Propagation

- From the above example it can be seen that we are adding 3 bits at a time sequentially until all bits are added.
- A full adder is a combinational circuit that performs the arithmetic sum of three input bits: augends  $A_i$ , addend  $B_i$  and carry in  $C_{in}$  from the previous adder.
- Its result contain the sum  $S_i$  and the carry out,  $C_{out}$  to the next stage.





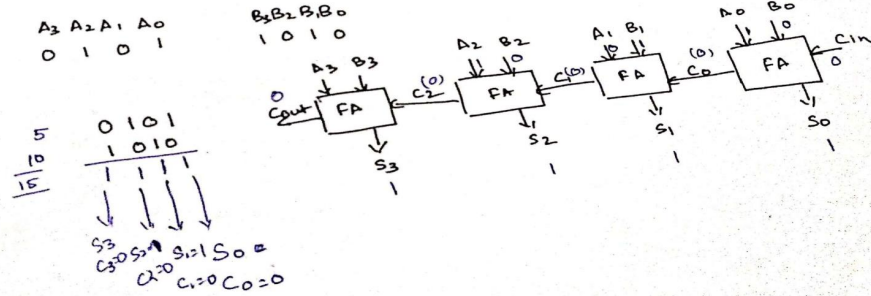
# example

## Ripple Carry Adder

It is useful for performing Multibit addition

E1

$A = 0101$ ,  $B = 1010$   $C_{in} = 0$  (Initial Condition)



# Ripple Carry Adder



## 4-bit Adder

- A 4-bit adder circuit can be designed by first designing the 1-bit full adder and then connecting the four 1-bit full adders to get the 4-bit adder as shown in the diagram above.
- For the 1-bit full adder, the design begins by drawing the Truth Table for the three input and the corresponding output SUM and CARRY.
- The Boolean Expression describing the binary adder circuit is then deduced.
- The binary full adder is a three input combinational circuit which satisfies the truth table given below.

# Ripple Carry Adder



## Full Adder

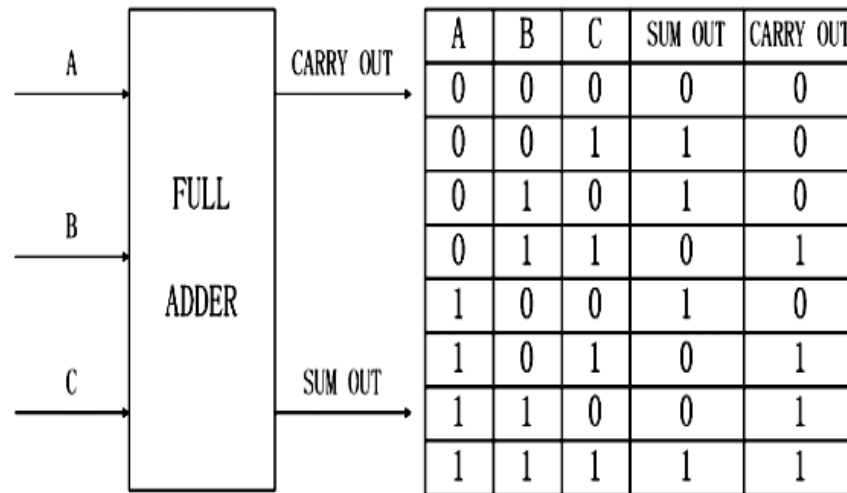
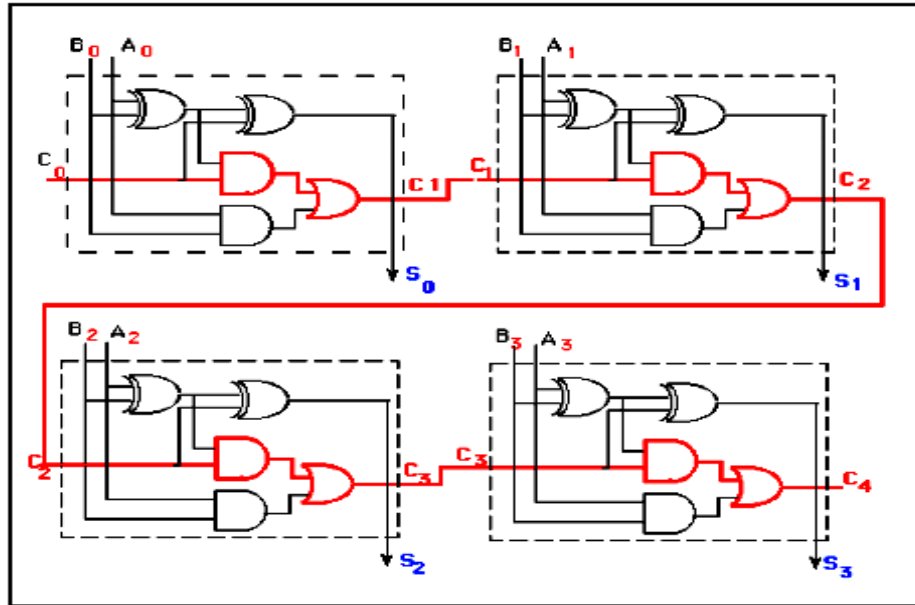


Fig.2. Diagram and Truth Table of Full Adder

# Ripple Carry Adder

## 4-bit Adder



# Design of Fast adder: Carry Look-ahead Adder



- A carry-look ahead adders (CLA) is a type of adder used in digital logic. A carry-look ahead adder improves speed by reducing the amount of time required to determine carry bits.
- It can be contrasted with the simpler, but usually slower, ripple carry adder for which the carry bit is calculated alongside the sum bit, and each bit must wait until the previous carry has been calculated to begin calculating its own result and carry bits (see adder for detail on ripple carry adders)

- The carry-look ahead adder calculates one or more carry bits before the sum, which reduces the wait time to calculate the result of the larger value bits.
- In a ripple adder the delay of each adder is 10 ns , then for 4 adders the delay will be 40 ns.
- To overcome this delay Carry Look-ahead Adder is used.

# Carry Look-ahead Adder

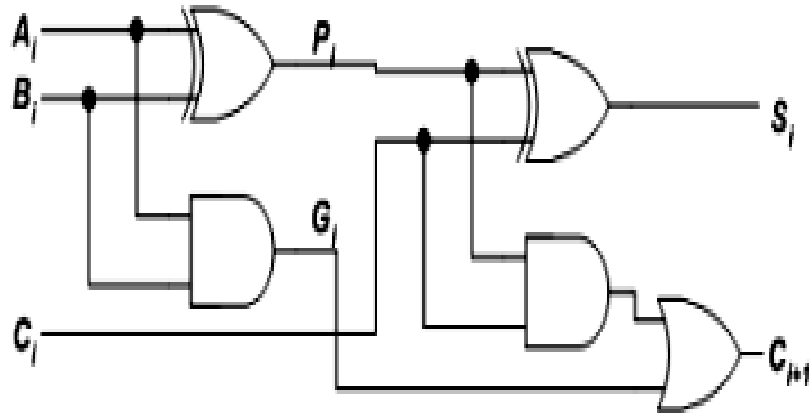


- Different logic design approaches have been employed to overcome the carry propagation delay problem of adders.
- One widely used approach employs the principle of carry look-ahead solves this problem by calculating the carry signals in advance, based on the input signals.
- This type of adder circuit is called as carry look-ahead adder (CLA adder). A carry signal will be generated in two cases:
  - (1) when both bits  $A_i$  and  $B_i$  are 1, or
  - (2) when one of the two bits is 1 and the carry-in (carry of the previous stage) is 1.



# Carry Look-ahead Adder

The Figure shows the full adder circuit used to add the operand bits in the  $I^{\text{th}}$  column; namely  $A_i$  &  $B_i$  and the carry bit coming from the previous column ( $C_i$ ).



# Carry Look-ahead Adder

In this circuit, the 2 internal signals  $P_i$  and  $G_i$  are given by:

$$P_i = A_i \oplus B_i \dots\dots\dots(1)$$

$$G_i = A_i B_i \dots\dots\dots(2)$$

The output sum and carry can be defined as

$$S_i = P_i \oplus C_i \dots\dots\dots(3)$$

$$C_{i+1} = G_i + P_i C_i \dots\dots\dots(4)$$

# Carry Look-ahead Adder



- $G_i$  is known as the carry Generate signal since a carry ( $C_{i+1}$ ) is generated whenever  $G_i = 1$ , regardless of the input carry ( $C_i$ ).
- $P_i$  is known as the carry propagate signal since whenever  $P_i = 1$ , the input carry is propagated to the output carry, i.e.,  $C_{i+1} = C_i$  (note that whenever  $P_i = 1$ ,  $G_i = 0$ ).
- Computing the values of  $P_i$  and  $G_i$  only depend on the input operand bits ( $A_i$  &  $B_i$ ) as clear from the Figure and equations.
- Thus, these signals settle to their steady-state value after the propagation through their respective gates.

# Carry Look-ahead Adder

- Computed values of all the  $P_i$ 's are valid one XOR-gate delay after the operands A and B are made valid.
- Computed values of all the  $G_i$ 's are valid one AND-gate delay after the operands A and B are made valid.
- The Boolean expression of the carry outputs of various stages can be written as follows:

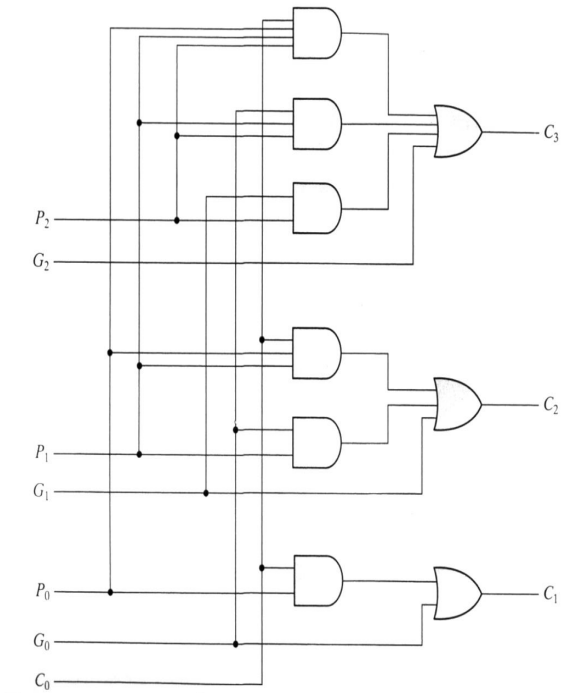
$$C_1 = G_0 + P_0 C_0$$

$$\begin{aligned} C_2 &= G_1 + P_1 C_1 = G_1 + P_1 (G_0 + P_0 C_0) \\ &= G_1 + P_1 G_0 + P_1 P_0 C_0 \end{aligned}$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

$$\begin{aligned} C_4 &= G_3 + P_3 C_3 \\ &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0 \end{aligned}$$

# Carry Look-ahead Adder



Implementing these expressions (for a 4-bit adder), results in the logic diagram.  
(IC- 74182)

$$c_1 = G_0 + P_0c_0$$

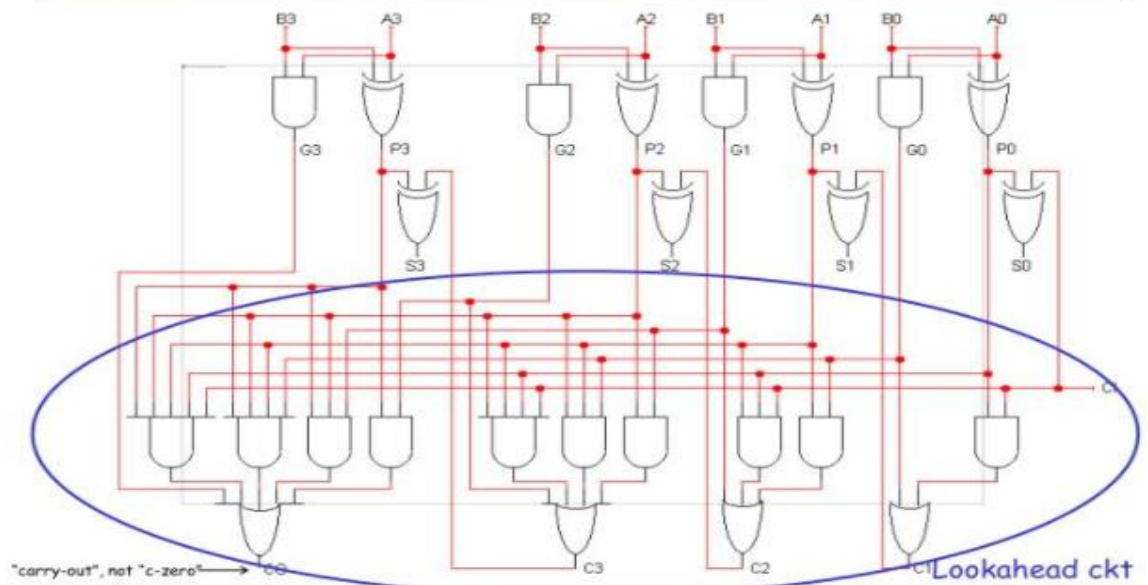
$$c_2 = G_1 + P_1G_0 + P_1P_0c_0$$

$$c_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0$$

$$c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$$

## A 4-bit carry lookahead adder circuit

Download

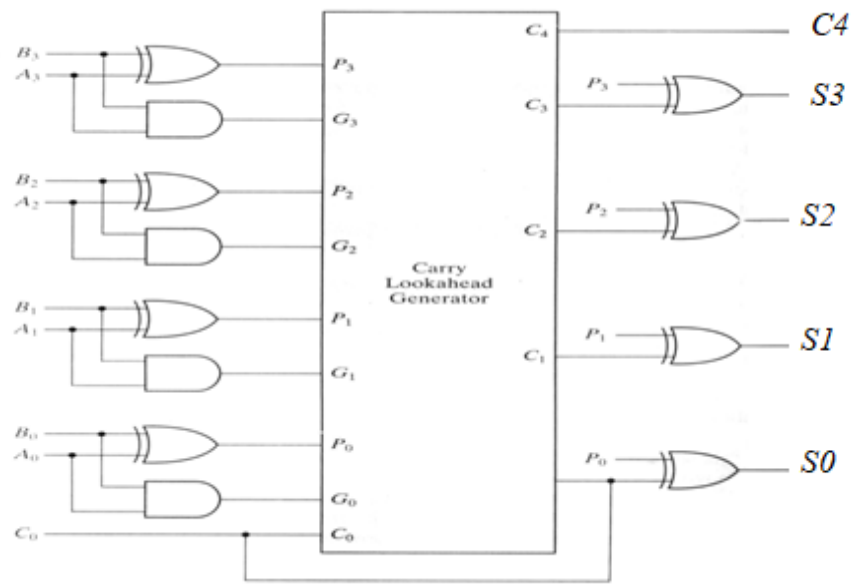


June 19, 2002

Addition and multiplication

3

# 4-bit Carry Look-ahead Adder



# 4-bit Carry Look-ahead Adder

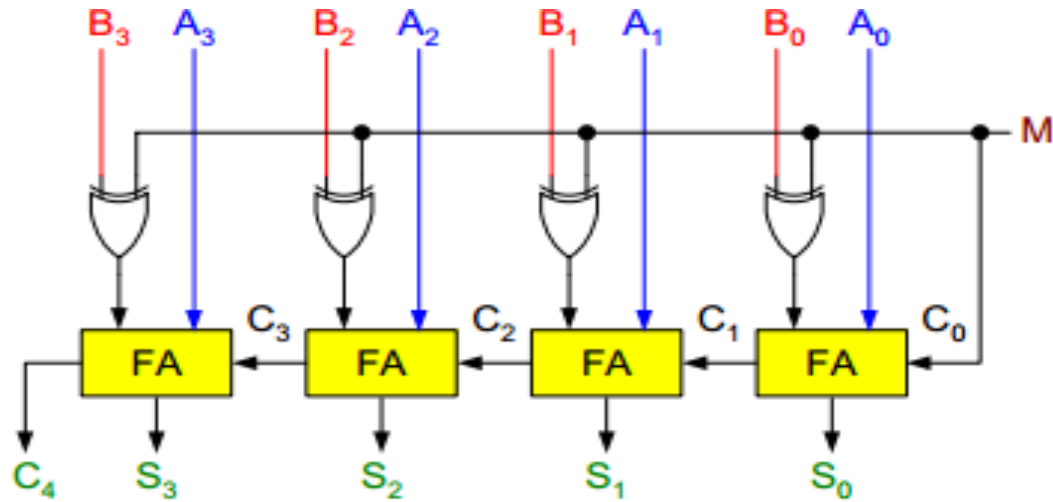


- independent of  $n$ ,
- the  $n$ -bit addition process requires only four gate delays (against  $2n$ )
- $C_{i+1}$  -- in 3 Gates delay;  $S_i$  -- in 4 Gates delay irrespective of  $n$ .  $C_1$  in 3 gates delay,  $C_2$  in 3 gates delay,  $C_3$  in 3 gates delay and so on.
- $S_0$  in 4 gates delay,  $S_1$  in 4 gates delay,  $S_2$  in 4 gates delay and so on.



# Binary Parallel Adder/Subtractor:

- The addition and subtraction operations can be done using an Adder-Subtractor circuit.
- The figure shows the logic diagram of a 4-bit Adder-Subtractor circuit.



## Binary Parallel Adder/Subtractor:

- The circuit has a mode control signal  $M$  which determines if the circuit is to operate as an adder or a subtractor.
- Each XOR gate receives input  $M$  and one of the inputs of  $B$ , i.e.,  $B_i$ . To understand the behavior of XOR gate consider its truth table given below.

- If one input of XOR gate is zero then the output of XOR will be same as the second input. While if one input of XOR gate is one then the output of XOR will be complement of the second input.

# Binary Parallel Adder/Subtractor:

- So when  $M = 0$ , the output of XOR gate will be  $B_i \oplus 0 = B_i$ . If the full adders receive the value of B, and the input carry  $C_0$  is 0, the circuit performs A plus B.
- When  $M = 1$ , the output of XOR gate will be  $B_i \oplus 1 = B_i'$ . If the full adders receive the value of  $B'$ , and the input carry  $C_0$  is 1, the circuit performs A plus 1's complement of B plus 1, which is equal to A minus B.

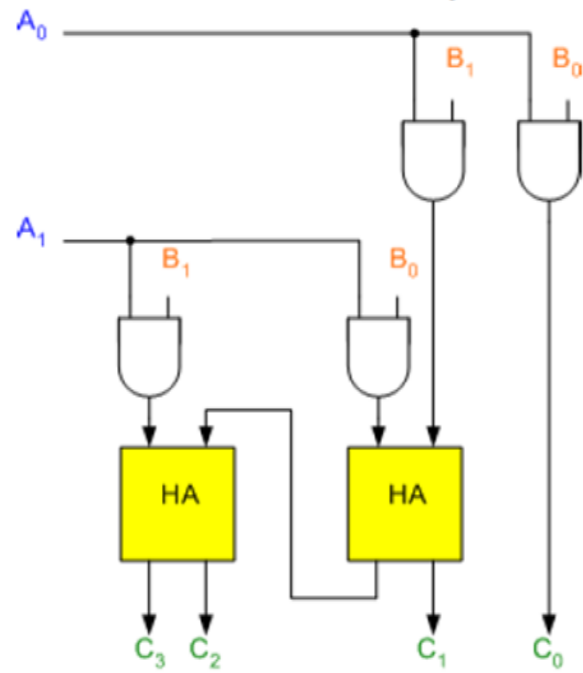
Multiplier –  
Unsigned, Signed  
Fast, Carry Save  
Addition of summands

# Binary Multiplier

- The usual method of long multiplication for decimal numbers applies also for binary numbers.
- Unsigned number multiplication : Two  $n$ -bit numbers;  
 $2n$ -bit result

# Binary Multiplier

$$\begin{array}{r}
 \begin{array}{c} B_1 \quad B_0 \\ \times A_1 \quad A_0 \\ \hline A_0 B_1 \quad A_0 B_0 \\ A_1 B_1 \quad A_1 B_0 \\ \hline \end{array} \\
 \begin{array}{cccc} C_3 & C_2 & C_1 & C_0 \end{array}
 \end{array}$$



# Binary Multiplier

- Unsigned number multiplication
- Two n-bit numbers; 2n-bit result

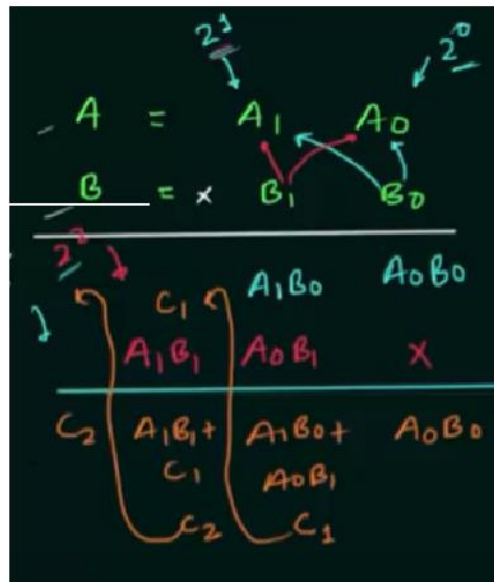
$$\begin{array}{r}
 \textcolor{blue}{(13)} \textcolor{red}{1101} \times \textcolor{red}{1011} \textcolor{blue}{(11)} \\
 \hline
 \textcolor{red}{1101} \\
 \textcolor{red}{1101} \\
 \textcolor{red}{0000} \\
 \textcolor{red}{1101} \\
 \hline
 \textcolor{red}{10001111} \textcolor{blue}{(143)} \\
 \hline
 \end{array}$$



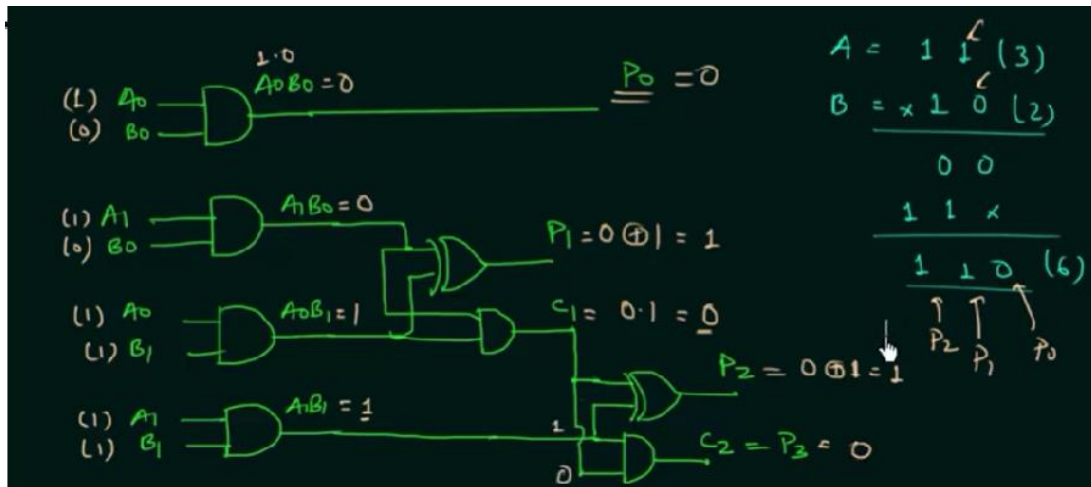
# Binary Multiplier

- The first partial product is formed by multiplying the B1B0 by A0. The multiplication of two bits such as A0 and B0 produces a 1 if both bits are 1; otherwise it produces a 0 like an AND operation. So the partial products can be implemented with AND gates.
- The second partial product is formed by multiplying the B1B0 by A1 and is shifted one position to the left.
- The two partial products are added with two half adders (HA). Usually there are more bits in the partial products, and then it will be necessary to use FAs.

# Binary Multiplier



# Binary Multiplier



# Binary Multiplier

- The least significant bit of the product does not have to go through an adder, since it is formed by the output of the first AND gate as shown in the Figure.

# Example

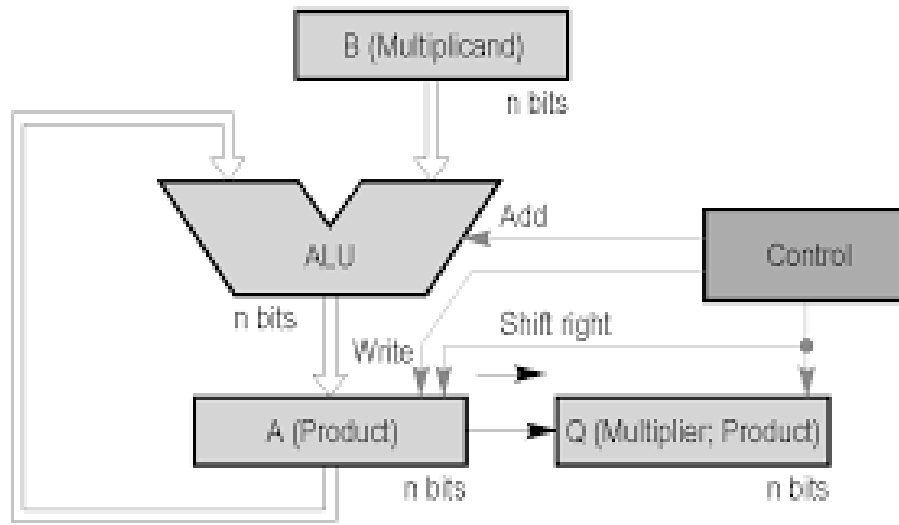
	$B_3$	$B_2$	$B_1$	$B_0$	
	1	1	0	1	
*			1	1	0
			<hr/>		
			0	0	0 0 $A_0$
+		1	1	0	1 $A_1$
		1	1	0	1 0
		<hr/>			
+	1	1	0	1	$A_2$
1	0	0	1	1	1 0
		<hr/>			

	1	3	
*		6	
	1	8	<hr/>
	6		
	7	8	<hr/>

# Multiplication of Positive Numbers: Shift-and-Add Multiplier

- Shift-and-add multiplication is similar to the multiplication performed by paper and pencil.
- This method adds the multiplicand  $X$  to itself  $Y$  times, where  $Y$  denotes the multiplier.
- To multiply two numbers by paper and pencil, the algorithm is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by a single digit of the multiplier and placing the intermediate product in the appropriate positions to the left of the earlier results.

# Shift-and-Add Multiplication



# Shift-and-Add Multiplier

- As an example, consider the multiplication of two unsigned 4-bit numbers,
- 8 (1000) and 9 (1001).

Multiplicand	1000 ×
Multiplier	<u>1001</u>
	1000
	0000
	0000
	<u>1000</u>
Product	1001000



# Shift-and-Add Multiplier

- In the case of binary multiplication, since the digits are 0 and 1, each step of the multiplication is simple.
- If the multiplier digit is 1, a copy of the multiplicand ( $1 \times \text{multiplicand}$ ) is placed in the proper positions;
- If the multiplier digit is 0, a number of 0 digits ( $0 \times \text{multiplicand}$ ) are placed in the proper positions.
- Consider the multiplication of positive numbers. The first version of the multiplier circuit, which implements the shift-and-add multiplication method for two n-bit numbers, is shown in Figure.

## Shift - and - Add multiplier

# Shift-and-Add Multiplier

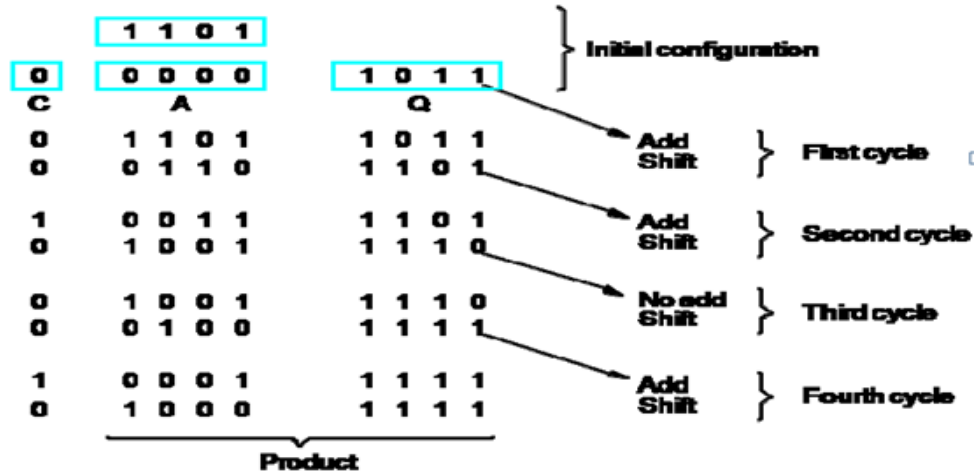
- For Example, Perform the multiplication  $9 \times 12$  ( $1001 \times 1100$ ). Finally, both A and Q contains the result of product.

Step	A	Q	B	Operation
0	0000	110 <u>0</u>	1001	Initialization
1	0000	011 <u>0</u>	1001	Shift right A_Q
2	0000	001 <u>1</u>	1001	Shift right A_Q
3	1001	001 <u>1</u>	1001	Add B to A
	0100	100 <u>1</u>	1001	Shift right A_Q
4	1101	100 <u>1</u>	1001	Add B to A
	0110	110 <u>0</u>	1001	Shift right A_Q

# Shift-and-Add Multiplier

## Example 2:

- A = 0000 M ( Multiplicand)  $\rightarrow$  13 ( 1 1 0 1)  
 Q(Multiplier)  $\rightarrow$  11 (1 0 1 1).



# Example to multiply 11 and 13

$11 \times 13$      $11 = 1011$  (M)     $N = 4$   
 $13 = 1101$  (Q)

N	C	A	Q	M	Operation
4	0	0000	1101	1011	$q_0 = 1 \rightarrow A = A + M, RS, N-1$ $0000$ $1011$ $1011$ $4-1=3$
3	0	0101	1101	1011	$q_0 = 0, RS, N-1$ $0010$ $1111$ $1011$ $3-1=2$
2	0	0010	1101	1011	$q_0 = 1, A = A + M, RS, N-1$ $0010$ $1011$ $1101$ $2-1=1$
1	0	0110	1101	1011	$q_0 = 1, A = A + M, RS, N-1$ $0110$ $1011$ $1001$ $N-1$ $1-1=0$

$N=0$  Stop the process  
 Product = A Q  
 $1000 1111$   
 $= 143$

Try for  $12 \times 10$

# Example to multiply 11 and 13

Example: Multiply 11 (Multiplicand) and 13 (Multiplier) using add-shift method.

M	C	A	Q <sub>n</sub>	Operation
1011	0	0000	1100	Initialization
	0	1011	1101	First cycle: ADD M with A $A = A + M$
	0	0101	1110	Shift-Right CAQ
	0	0010	1111	Second cycle: Shift Right CAQ
	0	1101	1111	Third cycle: $A = A + M$
	0	0110	1110	Shift-Right CAQ
	1	0001	1111	Fourth cycle: $A = A + M$
		1000	1111	Shift-Right CAQ

Handwritten notes on the right side of the table:

- 0010
- 1011
- 1101

## Signed Multiplication - Booth Algorithm

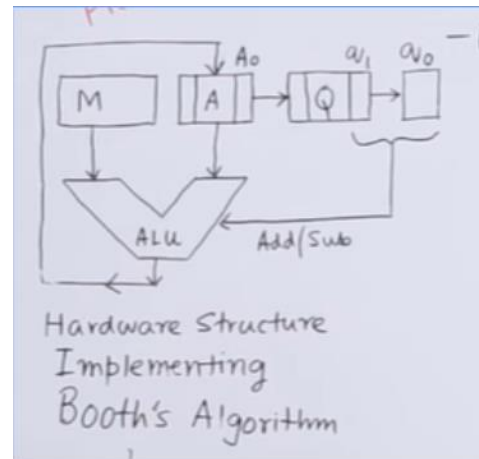
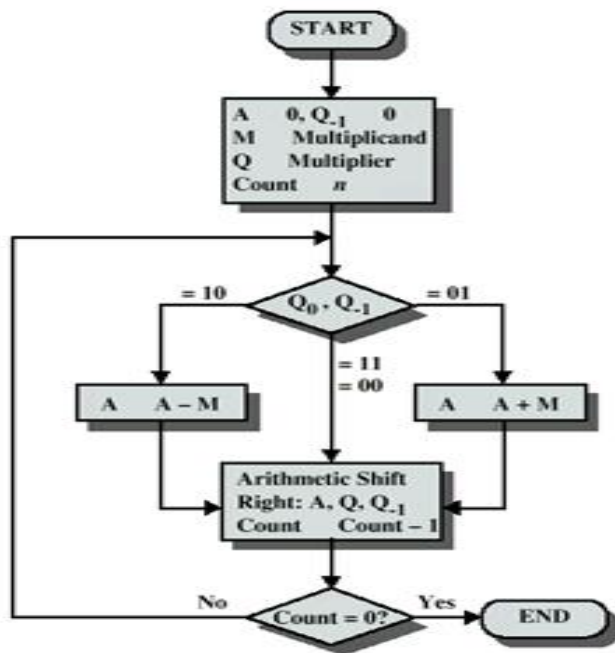
- A powerful algorithm for signed number multiplication is Booth's algorithm which generates a  $2n$  bit product and treats both positive and negative numbers uniformly.
- This algorithm suggests that we can reduce the number of operations required for multiplication by representing multiplier as a difference between two numbers.

# Signed Multiplication - Booth Algorithm

- A powerful algorithm for signed number multiplication is Booth's algorithm which generates a  $2n$  bit product and treats both positive and negative numbers uniformly.
- This algorithm suggest that we can reduce the number of operations required for multiplication by representing multiplier as a difference between two numbers.



## Signed Multiplication - Booth Algorithm



## Signed Multiplication - Booth Algorithm

Signed Multiplication: Booth's Algorithm

Example:  $(-7) \times (+3) = (-21)$  ✓

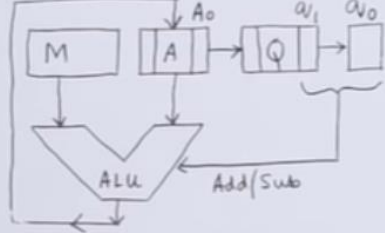
$M = -7_{10} = 2's \text{ Comp} = 1001_2$   
 $-M = 0110_2$

Tracing Table

n	A	$a_n Q$	$a_{n+1} q_0$	Action/Comment
4	0000	0011	0	Initialization
	0111	0011	0	$A = A - M$
3	0011	1001	1	ASR $AQq_0$
2	0001	1100	1	ASR $AQq_0$
	1010	1100	1	$A = A + M$
1	1101	0110	0	ASR $AQq_0$
0	1110	1011	0	ASR $AQq_0$

↓ 1's Compl

Hardware Structure Implementing Booth's Algorithm



## Signed Multiplication - Booth Algorithm

A	Q	Q <sub>-1</sub>	M	Initial Values	
0000	0011	0	0111		
1001	0011	0	0111	A    A - M	} First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	} Second Cycle
0101	0100	1	0111	A    A + M	
0010	1010	0	0111	Shift	} Third Cycle
0001	0101	0	0111	Shift	
					} Fourth Cycle

**Example for Signed Number Multiplication**

①  $-7 \times 3$   $q = 3 = 0011$   $q_0 = 0$   
 $M = -7$   $N = 4$   
 $A = 0111 \xrightarrow{1's} 1000 \xrightarrow{2's} 1001$   $-M = 0111$

N	A	q	q <sub>0</sub>	M	Steps
4	0000	0011	0	1001	$q_0 = 0$ $q_1 = 1$ $A = A - M$ $= A + (-M)$ $= 0000$ $0111$ $0111$
3	$0111$ $\swarrow$ $0011$	$0011$ $\swarrow$ $1001$	0	1001	$q_0 = 0$ $q_1 = 1$ $A = A - M$ $= 0000$ $0111$ $0111$
2	$0011$ $\swarrow$ $0001$	$1100$ $\swarrow$ $1100$	1	1001	$q_0 = 1$ $q_1 = 1$ $A = A + M$ $0001$ $1001$ $1010$
1	$1010$ $\swarrow$ $1101$	$1100$ $\swarrow$ $0110$	1	1001	$q_0 = 1$ $q_1 = 0$ $A = A + M$ $0001$ $1001$ $1010$
0	$1101$ $\swarrow$ $1110$	$0110$ $\swarrow$ $1011$	0	1001	$q_0 = 0$ $q_1 = 0$ $A = A$ $1110$

**Product = AB**

$= 1110 1011$   
 $= - (1110 1011)$   
 $\downarrow$  1's  
 $0001 0100$   
 $\downarrow$  2's  
 $000 10101$   
 $\downarrow$  2's  
 $21$

$\therefore -21$

### Example 2

Perform Multiplication for  $(-15) \times (-13)$

$M = -15$

$B = -13$

$15 = 8 \ 4 \ 2 \ 1$

$13 = 8 \ 4 \ 2 \ 1$

$0 \ 1 \ 1 \ 1$

$0 \ 1 \ 1 \ 0 \ 1$

$1's = 1 \ 0 \ 0 \ 0 \ 0$

$1's \rightarrow 1 \ 0 \ 0 \ 1 \ 0$

$2's = 1 \ 0 \ 0 \ 0 \ 1$

$2's \rightarrow 1 \ 0 \ 0 \ 1 \ 1$

$M = 10001$

$B = 10011, B_0 = 0$

$-M = 01111$

$N = 5$

N	A	q	q <sub>0</sub>	M	q <sub>0</sub> =0 A=A-M q <sub>1</sub> =1 A=A+(-M) q <sub>0</sub> =1, q <sub>1</sub> =1 $\Rightarrow$ ARS $\rightarrow$ N-1
5	00000 01111 00111	10011 10011 11001	0 0 1	10001 10001 10001	00000 01111 01111
4	00111 00011	11001 11100	1 1	10001 10001	
3	00011 10100 11010	11100 11100 01110	1 1 0	10001 10001 10001	00011 10001 10100
2					

N	A	q	q <sub>0</sub>	M	q <sub>0</sub> =0 q <sub>1</sub> =0 ARS $\rightarrow$ N-1
2	11010 11101	01110 00111	0 0	10001 10001	
1	11101 01100 00110	00111 00111 00011	0 0 1	10001 10001 10001	q <sub>0</sub> =0 A=A-M q <sub>1</sub> =1 A=A+(-M) 11101 01111 10110
0					ARS $\downarrow$ N-1

Product = AB

00110 00011 = 195

$-15 \times -13 = 195$

Try for  $115 \times -13$   
a)  $-6 \times 7$

## Alternate Method : The BOOTH Algorithm

- Booth multiplication reduces the number of additions for intermediate results, but can sometimes make it worse as we will see.
- [Booth multiplier recoding table](#)

Multiplier		Version of multiplicand selected by bit
Bit $i$	Bit $i-1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

## The BOOTH RECODED MULTIPLIER

- Booth Recoding:

(i)  $30_{10}$  : 0 1 1 1 1 0 0

- +1 0 0 0 -1 0

(ii)  $100_{10}$ : 0 1 1 0 0 1 0 0 0

- 

- +1 0 -1 0 +1 -1 0 0

(iii)  $985_{10}$ : 0 0 1 1 1 1 0 1 1 0 0 1 0

- 0 +1 0 0 0 -1 +1 0 -1 0 +1 -1

## BOOTH Algorithm

Booth algorithm treats both +ve &  
 -ve operands equally.  
 (a) + Md X + Mr

**Ex:**    0 1 1 0 1 (+13) X 0 1 0 1 1 (+11)

+1 -1 +1 0 -1

---

1 1 1 1 1 1 0 0 1 1  
 0 0 0 0 0 0 0 0 0  
 0 0 0 0 1 1 0 1  
 1 1 1 0 0 1 1  
 0 0 1 1 0 1

---

0 0 1 0 0 0 1 1 1 1 (+143)

---



## BOOTH Algorithm

Booth algorithm treats both +ve &  
 -ve operands equally.  
 (b) - Md X + Mr

**Ex:**    1 0 0 1 1 (-13) × 0 1 0 1 1 (+11)

          +1 -1 +1 0 -1


---

0	0	0	0	0	0	1	1	0	1
0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	1	1		
0	0	0	1	1	0	1			
1	1	0	0	1	1				

---

1 0 1 0 0 0 0 0 0 1 (-143)

---





## BOOTH Algorithm

Booth algorithm treats both +ve &  
 -ve operands equally.  
 (d) - Md X - Mr

**Ex:**    1 0 0 1 1 (-13) X 0 0 1 0 1 (-11)

-1+1-1 +1 -1

---

0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 1 1 0 1  
 1 1 1 1 1 0 0 1 1  
 0 0 0 0 1 1 0 1  
 1 1 1 0 0 1 1  
 0 0 1 1 0 1  
 0 0 1 0 0 0 1 1 1 1 (+143)

---

## BOOTH Algorithm

Good multiplier

0	0	0	0	1	1	1	1	1	0	0	0	0	1	1	1
↓															
0	0	0	+1	0	0	0	0	-1	0	0	0	+1	0	0	-1

Count = 4 Vs 8 speed improvement

Ordinary multiplier

1	1	0	0	0	1	0	1	1	0	1	1	1	1	0	0
↓															
0	-1	0	0	+1	-1	+1	0	-1	+1	0	0	0	-1	0	0

Count = 7 Vs 9 no speed improvement

Worst-case multiplier

0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
↓															
+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1	+1	-1

Count = 16 Vs 8 speed worsened.

On an average no improvement in speed



# FAST MULTIPLICATION

1. BIT PAIR RECODING OF MULTIPLIERS
2. CARRY SAVE ADDITION OF SUMMANDS

## Fast Multiplication

- There are two techniques for speeding up the multiplication operation.
- The first technique guarantees that the maximum number of summands (versions of the multiplicand) that must be added is  $n/2$  for  $n$ -bit operands.
- The second technique reduces the time needed to add the summands (carry-save addition of summands method).

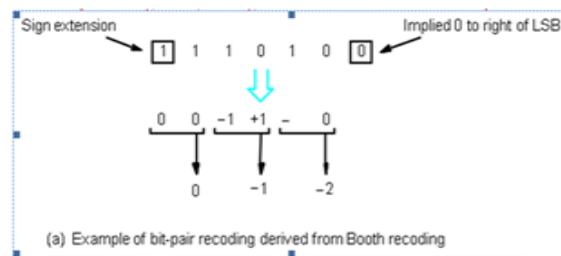
## Bit-Pair Recoding of Multipliers

- This bit-pair recoding technique halves the maximum number of summands. It is derived from the Booth algorithm.
- Group the Booth-recoded multiplier bits in pairs, and observe the following: The pair  $(+1 -1)$  is equivalent to the pair  $(0 +1)$ .
- That is, instead of adding  $-1$  times the multiplicand  $M$  at shift position  $i$  to  $+1 \times M$  at position  $i + 1$ , the same result is obtained by adding  $+1 \times M$  at position  $i$ . Other examples are:  $(+1 0)$  is equivalent to  $(0 +2)$ ,  $(-1 +1)$  is equivalent to  $(0 -1)$ . and so on





## Bit-Pair Recoding of Multipliers



original bit pair		Bit to right	Bit pair Recoded		Multiplier value
$i+1$	$i$	$i-1$	$Y_{i+1}$	$y_i$	
0	0	0	0	0	0
0	0	1	0	1	+1
0	1	0	1	-1	+1
0	1	1	1	0	+2
1	0	0	-1	0	-2
1	0	1	-1	1	-1
1	1	0	0	-1	-1
1	1	1	0	0	0

## BIT PAIR RECODING OF MULTIPLIER

$i+1$	$i$	$i-1$	Multiplicand Selected position $i$
0	0	0	$+0 \times M$
0	0	1	$+1 \times M$
0	1	0	$+1 \times M$
0	1	1	$+2 \times M$
1	0	0	$-2 \times M$
1	0	1	$-1 \times M$
1	1	0	$-1 \times M$
1	1	1	$0 \times M$



# Carry-Save Addition of Summands



- A **carry-save adder** is a type of digital adder, used to efficiently compute the sum of three or more binary numbers.
- A carry-save adder (CSA), or 3-2 adder, is a very fast and cheap adder that does not propagate carry bits.
- A Carry Save Adder is generally used in binary multiplier, since a binary multiplier involves addition of more than two binary numbers after multiplication.
- It can be used to speed up addition of the several summands required in multiplication
- It differs from other digital adders in that it outputs two (or more) numbers, and the answer of the original summation can be achieved by adding these outputs together.
- A big adder implemented using this technique will usually be much faster than conventional addition of those numbers.

## Fast Multiplication

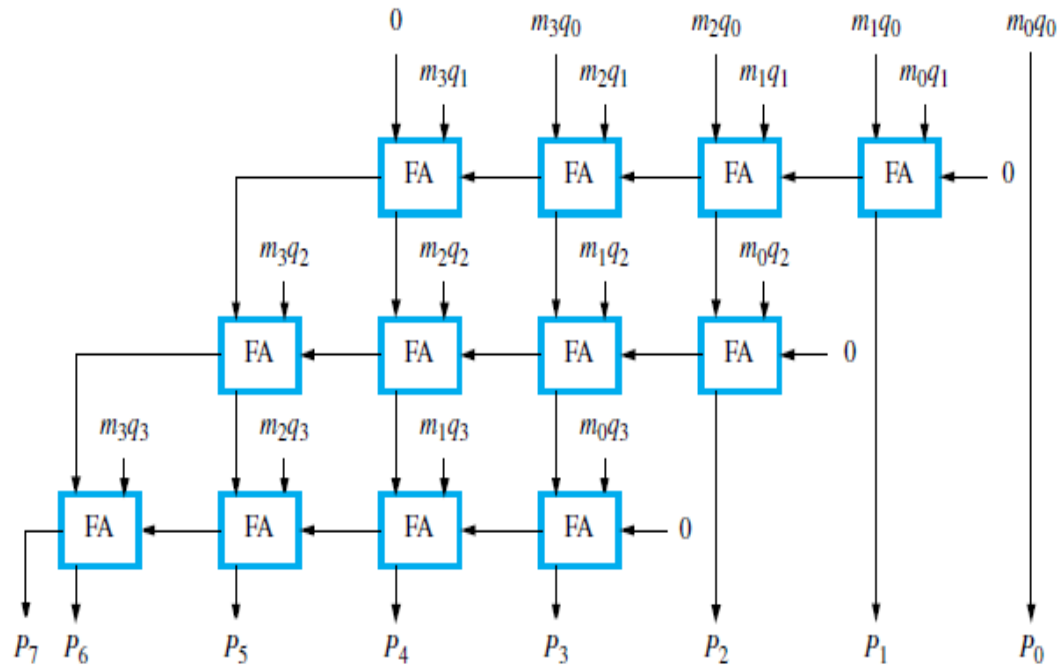
- Bit pair recoding reduces summands by a factor of 2
- Summands are reduced by carry save addition
- Final product can be generated by using carry look ahead adder

# Carry-Save Addition of Summands

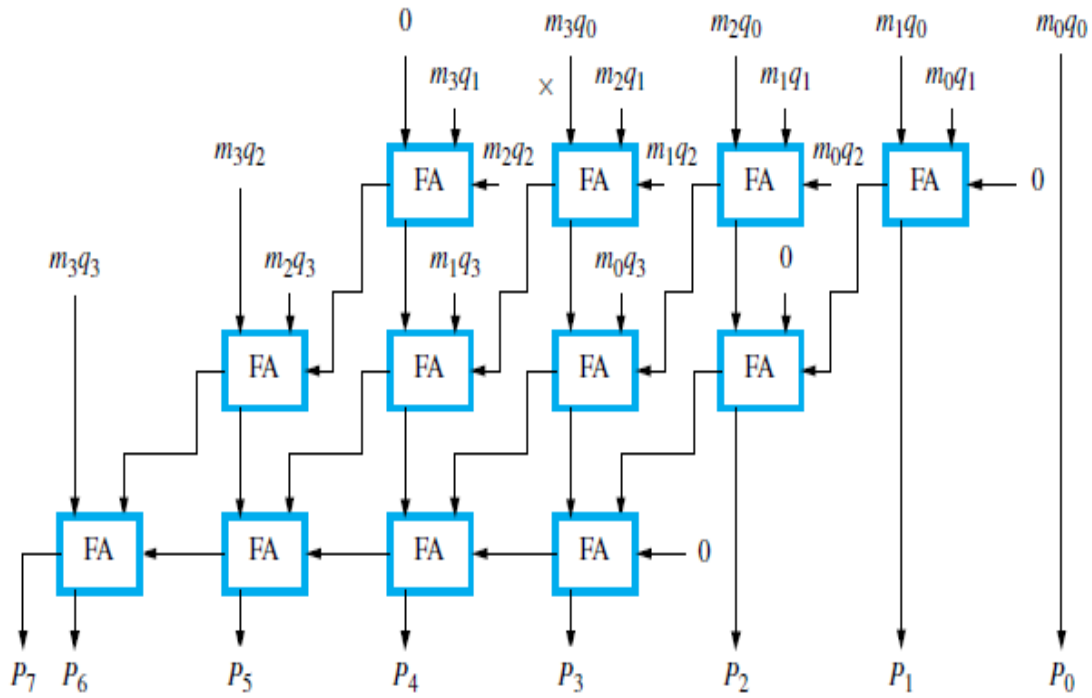


- **Disadvantage of the Ripple Carry Adder** - Each full adder has to wait for its carry-in from its previous stage full adder. This increase propagation time. This causes a delay and makes ripple carry adder extremely slow. RCAR is very slow when adding many bits.
- **Advantage of the Carry Look ahead Adder** - This is an improved version of the Ripple Carry Adder. Fast parallel adder. It generates the carry-in of each full adder simultaneously without causing any delay. So, CLAr is faster (because of reduced propagation delay) than RCAR.
- **Disadvantage the Carry Look-ahead Adder** - It is costlier as it reduces the propagation delay by more complex hardware. It gets more complicated as the number of bits increases.

# Ripple Carry Array



# Carry Save Array





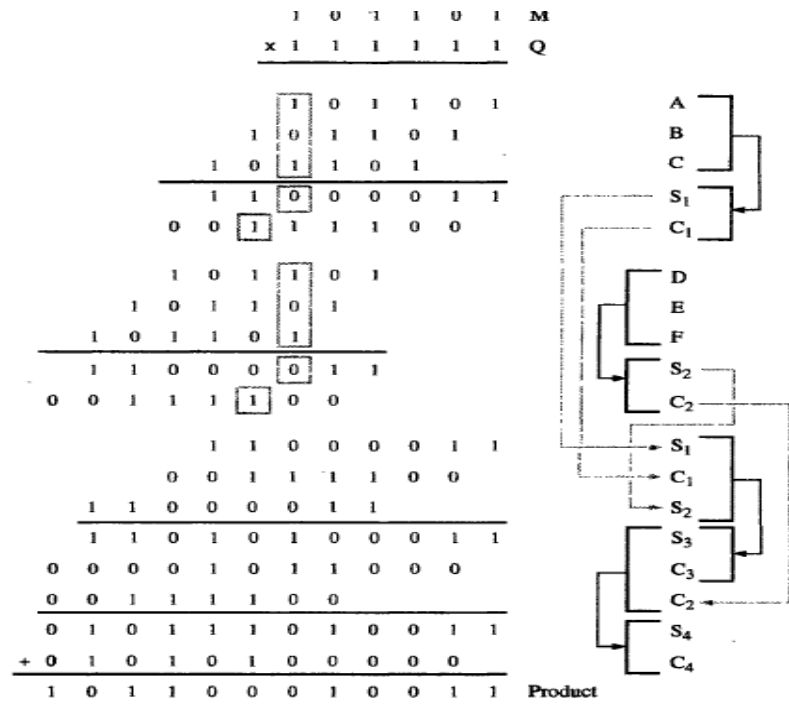
# Carry-Save Addition of Summands

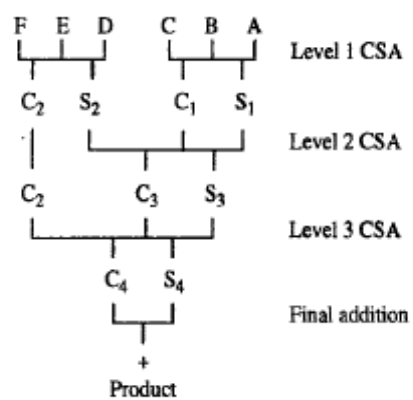


- Consider the addition of many summands, We can:
  - Group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay
  - Group all of the S and C vectors into threes, and perform carry-save addition of them, generating a further set of S and C vectors in one more full-adder delay
  - Continue with this process until there are only two vectors remaining
  - They can be added in a Ripple Carry Adder (RPA) or Carry Look-ahead Adder (CLA) to produce the desired product

$$\begin{array}{r}
 1\ 0\ 1\ 1\ 0\ 1 \\
 \times 1\ 1\ 1\ 1\ 1\ 1 \\
 \hline
 1\ 0\ 1\ 1\ 0\ 1 \\
 1\ 0\ 1\ 1\ 0\ 1 \\
 1\ 0\ 1\ 1\ 0\ 1 \\
 1\ 0\ 1\ 1\ 0\ 1 \\
 1\ 0\ 1\ 1\ 0\ 1 \\
 1\ 0\ 1\ 1\ 0\ 1 \\
 \hline
 1\ 0\ 1\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 1
 \end{array}$$

(45)      M  
 (63)      Q  
 A  
 B  
 C  
 D  
 E  
 F  
 (2,835)      Product

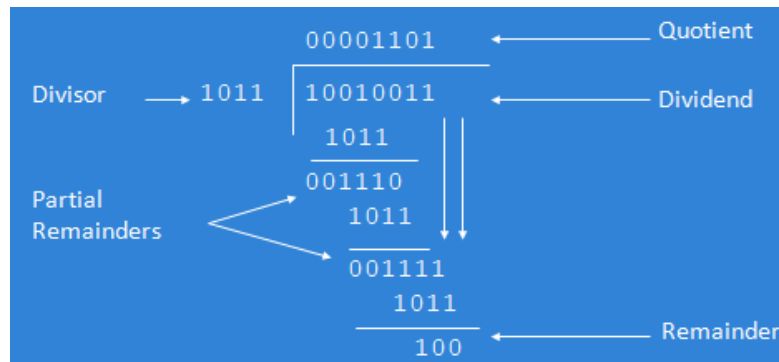




# **Division – Restoring and Non – Restoring**

# Integer Division

- More complex than multiplication
- Negative numbers are really bad!
- Based on long division



# Integer Division

- Decimal Division
- Binary Division

$\begin{array}{r} 21 \\ 13 \overline{) 274} \\ \underline{26} \phantom{0} \\ 14 \phantom{0} \\ \underline{13} \phantom{0} \\ 1 \phantom{0} \end{array}$	$\begin{array}{r} 10101 \\ 1101 \overline{) 100010010} \\ \underline{1101} \phantom{000} \\ 10000 \phantom{0} \\ \underline{1101} \phantom{000} \\ 1110 \phantom{00} \\ \underline{1101} \phantom{00} \\ 1 \phantom{00} \end{array}$
---	--

Figure: Longhand division examples

# Long Hand Division

Longhand Division operates as follows:

- Position the divisor appropriately with respect to the dividend and performs a subtraction.
- If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed.
- If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction



# Restoring Division

- Similar to multiplication circuit
- An **n-bit positive divisor** is loaded into register M and an **n-bit positive dividend** is loaded into register Q at the start of the operation.
- **Register A is set to 0**
- After the division operation is complete, the **n-bit quotient is in register Q and the remainder is in register A.**
- The required subtractions are facilitated by using 2's complement arithmetic.
- The extra bit position at the left end of both A and M accommodates the sign bit during subtractions.

# Restoring Division

## Strategy for unsigned division:

Shift the dividend one bit at a time starting from MSB into a register.

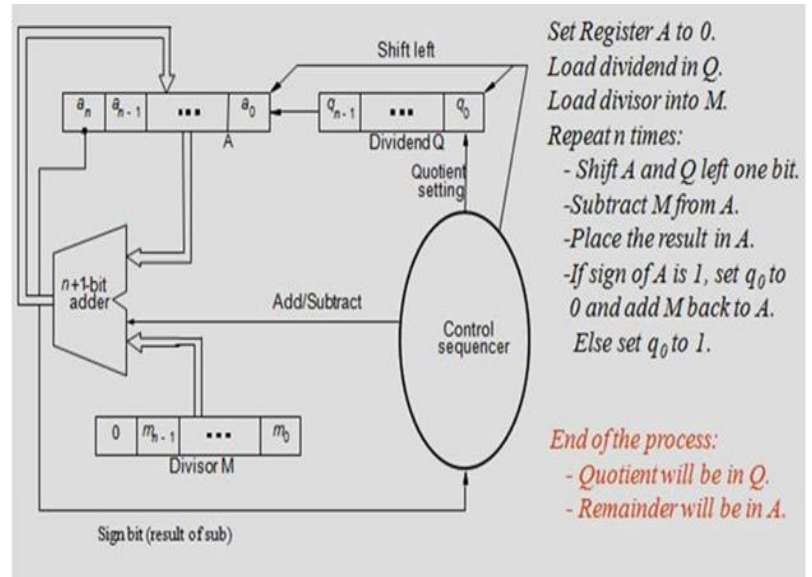
Subtract the divisor from this register.

If the result is negative ("didn't go"):

- Add the divisor back into the register.
- Record 0 into the result register.

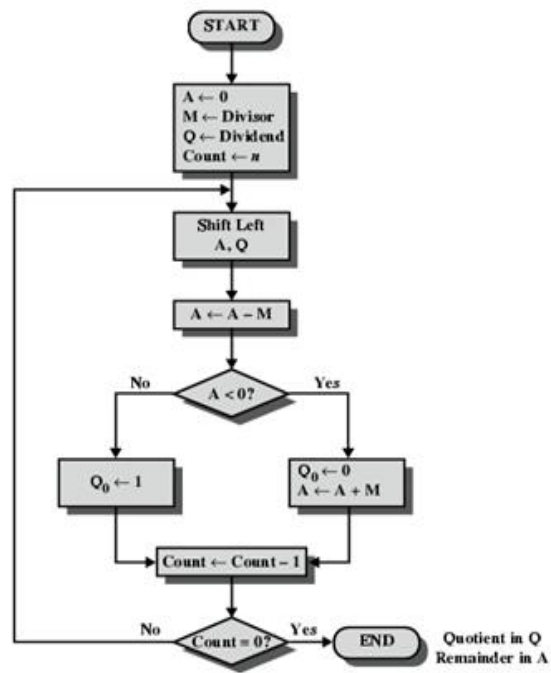
If the result is positive:

- Do not restore the intermediate result.
- Set a 1 into the result register.

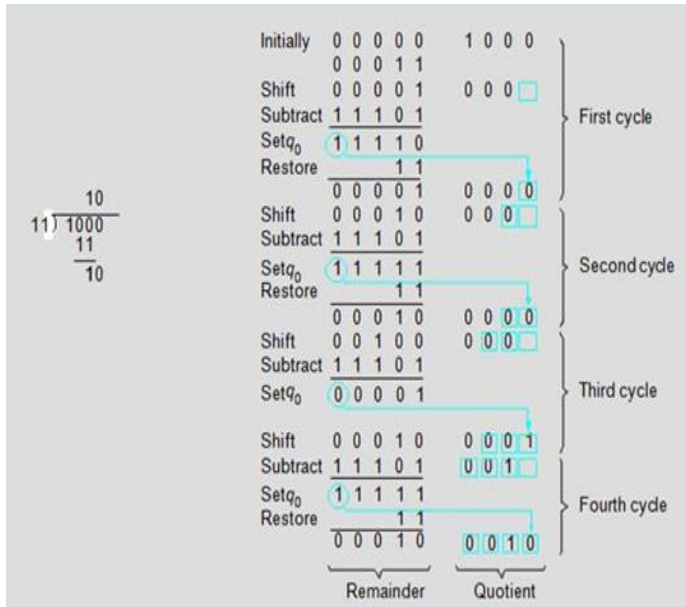


**Figure: Logic Circuit arrangement for binary Division (Restoring)**

# Flowchart for Restoring Division



# Restoring Division – Example

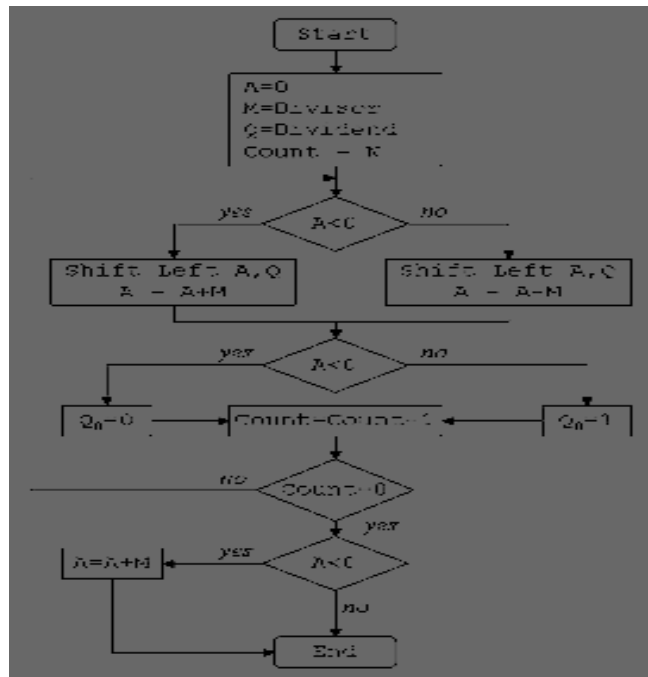


Restoring Division				
Count	A	Divisor	Dividend	Quotient
4	Initial	0 0 0 0	1 0 0 0	11/1000
	Shift left AQ	0 0 0 1	0 0 0 0	11
	Subtract (A ← A - D)	1 1 1 0	0 0 0 0	11
	A < 0; Q <sub>0</sub> = 0	0 0 0 1	0 0 0 0	11
	Add (A ← A + D) Restore	0 0 0 1	0 0 0 0	11
3	Shift left AQ	0 0 0 1	0 0 0 0	11
	Subtract (A ← A - D)	1 1 1 0	0 0 0 0	11
	A < 0; Q <sub>0</sub> = 0	0 0 0 1	0 0 0 0	11
	Add (A ← A + D) Restore	0 0 0 1	0 0 0 0	11
2	Shift left AQ	0 0 1 0	0 0 0 0	11
	Subtract (A ← A - D)	1 1 1 0	0 0 0 0	11
	A > 0; Q <sub>0</sub> = 1	0 0 0 1	0 0 0 0	11
1	Shift left AQ	0 0 1 0	0 0 0 0	11
	Subtract (A ← A - D)	1 1 1 0	0 0 0 0	11
	A < 0; Q <sub>0</sub> = 0	0 0 1 0	0 0 0 0	11
	Add (A ← A + D) Restore	0 0 1 0	0 0 0 0	11
0	Remainder	0 0 1 0	Quotient	0 0 1 0

# Non-Restoring Division

- Initially Dividend is loaded into register Q,  
and n-bit Divisor is loaded into register M
- Let  $M'$  is 2's complement of M
- Set Register A to 0
- Set count to n
- SHL AQ denotes shift left AQ by one position leaving  $Q_0$  blank.
- Similarly, a square symbol in  $Q_0$  position denote, it is to be calculated later

# Flowchart for Non-Restoring Division



# Non-Restoring Division

Restoring division can be improved using non-restoring algorithm

The effect of restoring algorithm actually is:

If  $A$  is positive, we shift it left and subtract  $M$ , that is compute  $2A - M$

If  $A$  is negative, we restore it ( $A + M$ ), shift it left, and subtract  $M$ , that is,  $2(A + M) - M = 2A + M$ .

Set  $q_0$  to 1 or 0 appropriately.

*Non-restoring algorithm is:*

*Set  $A$  to 0.*

*Repeat  $n$  times:*

*If the sign of  $A$  is positive:*

*Shift  $A$  and  $Q$  left and subtract  $M$ . Set  $q_0$  to 1.*

*Else if the sign of  $A$  is negative:*

*Shift  $A$  and  $Q$  left and add  $M$ . Set  $q_0$  to 0.*

*If the sign of  $A$  is 1, add  $A$  to  $M$ .*

# Non-Restoring Division - Example

Initially	0 0 0 0 0	1 0 0 0																					
Shift	0 0 0 1 1	0 0 0																					
Subtract	1 1 1 0 1		First cycle																				
Set $q_0$	1 1 1 1 0	0 0 0 0																					
Shift	1 1 1 0 0	0 0 0																					
Add	0 0 0 1 1		Second cycle																				
Set $q_0$	1 1 1 1 1	0 0 0 0																					
Shift	1 1 1 1 0	0 0 0																					
Add	0 0 0 1 1		Third cycle																				
Set $q_0$	0 0 0 0 1	0 0 0 1																					
Shift	0 0 0 1 0	0 0 1																					
Subtract	1 1 1 0 1		Fourth cycle																				
Set $q_0$	1 1 1 1 1	0 0 1 0																					
<table> <tr> <td colspan="2"></td><td>Quotient</td><td></td></tr> <tr> <td>Add</td><td>1 1 1 1 1</td><td></td><td></td></tr> <tr> <td></td><td>0 0 0 1 1</td><td></td><td>Restore remainder</td></tr> <tr> <td></td><td>0 0 0 1 0</td><td></td><td></td></tr> <tr> <td colspan="2"></td><td>Remainder</td><td></td></tr> </table>						Quotient		Add	1 1 1 1 1				0 0 0 1 1		Restore remainder		0 0 0 1 0					Remainder	
		Quotient																					
Add	1 1 1 1 1																						
	0 0 0 1 1		Restore remainder																				
	0 0 0 1 0																						
		Remainder																					

Count	Non-Restoring Division			
		A	Q (Dividend)	2 3 3
4	Initial	0 0 0 0 0	1 0 0 0	10
	$A \rightarrow +ve$ ; Shift left A	0 0 0 1 1		10
	$A \leftarrow A - M$ ; Subtract	1 1 1 0 1		10
3	$A < 0$ ; $Q_0 \leftarrow 0$	1 1 1 1 0	0 0 0 0	10
	$A \rightarrow -ve$ ; Shift left A	1 1 1 0 0	0 0 0 0	10
	$A \leftarrow A + M$ ; Add	0 0 0 1 1		10
2	$A < 0$ ; $Q_0 \leftarrow 0$	1 1 1 1 1	0 0 0 0	10
	$A \rightarrow -ve$ ; Shift left A	1 1 1 1 0	0 0 0 0	10
	$A \leftarrow A + M$ ; Add	0 0 0 1 1		10
	$A < 0$ ; $Q_0 \leftarrow 1$	0 0 0 0 1	0 0 0 1	10
1	$A \rightarrow +ve$ ; Shift left A	0 0 0 1 0	0 0 0 1	10
	$A \leftarrow A - M$ ; Subtract	1 1 1 0 1		10
0	$A < 0$ ; $Q_0 \leftarrow 0$	1 1 1 1 1	0 0 0 1	10
	$A \rightarrow -ve$ ; Add	0 0 0 1 1		10
	$A \leftarrow A + M$	0 0 0 1 0	0 0 0 1	10
		Remainder	Quotient	
		0 0 1 0	0 0 1 0	



# IEEE 754

## Floating point numbers and operations.

# Floating-Point Arithmetic (IEEE 754)

- ✓ The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation which was established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE).
- ✓ The standard addressed many problems found in the diverse floating point implementations that made them difficult to use reliably and reduced their portability.
- ✓ IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macs, and most Unix platforms.

IEEE 754 has 3 basic components:

## 1. The Sign of Mantissa

- ✓ 0 represents a positive number while
- ✓ 1 represents a negative number.

## 2. The Biased exponent

- ✓ The exponent field needs to represent both positive and negative exponents.
- ✓ A bias is added to the actual exponent in order to get the stored exponent.

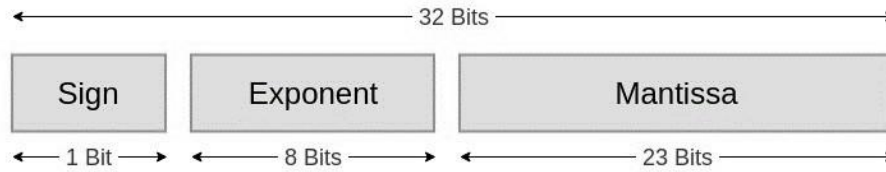
## 3. The Normalised Mantissa

- ✓ The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits.
- ✓ Here we have only 2 digits, i.e. 0 and 1. So a normalised mantissa is one with only one 1 to the left of the decimal.

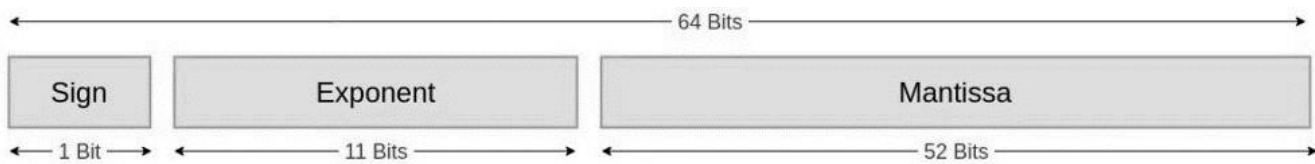
IEEE 754 numbers are divided into two based on the above three components:

- ✓ Single precision and
- ✓ Double precision.

## 1. Single precision



## 2. Double precision



# Example 1: Single precision

Biased exponent  $127+6=133$

$133 = 10000101$

Normalised mantisa =  $010101001$

we will add 0's to complete the 23 bits

**The IEEE 754 Single precision is:**

0 10000101 01010100100000000000000

This can be written in hexadecimal form **42AA4000**

## Example 2: Double precision

Biased exponent  $1023+6=1029$

$$1029 = 10000000101$$

Normalised mantisa = 010101001

we will add 0's to complete the 52 bits

## The IEEE 754 Double precision is:

0 10000000101 010101001000000000000000000000000000000000000000

This can be written in hexadecimal form **4055480000000000**

IEEE has reserved some values that can be ambiguous.

## 1. Zero

- ✓ Zero is a special value denoted with an exponent and mantissa of 0.
- ✓  $-0$  and  $+0$  are distinct values, though they both are equal.

## 2. Denormalised

- ✓ If the exponent is all zeros, but the mantissa is not then the value is a denormalized number.
- ✓ This means this number does not have an assumed leading one before the binary point.

## 3. Infinity

- ✓ The values  $+\infty$  and  $-\infty$  are denoted with an exponent of all ones and a mantissa of all zeros.
- ✓ The sign bit distinguishes between negative infinity and positive infinity.

# IEEE 754 Special Values

- ✓ The value NAN is used to represent a value that is an error.
- ✓ This is represented when exponent field is all ones with a zero sign bit or a mantissa that is not 1 followed by zeros.
- ✓ This is a special value that might be used to denote a variable that doesn't yet hold a value.

Single Precision		
Exponent	Mantisa	Value
0	0	exact 0
255	0	Infinity
0	not 0	denormalised
255	not 0	Not a number (NAN)

Double Precision		
Exponent	Mantisa	Value
0	0	exact 0
2049	0	Infinity
0	not 0	denormalised
2049	not 0	Not a number (NAN)



# Ranges of Floating point numbers

- ✓ The range of positive floating point numbers can be split into normalized numbers, and denormalized numbers which use only a portion of the fractions's precision.
- ✓ Since every floating-point number has a corresponding, negated value, the ranges above are symmetric around zero.
- ✓ There are five distinct numerical ranges that single-precision floating-point numbers are not able to represent with the scheme presented so far:
  1. Negative numbers less than  $-(2 - 2^{-23}) \times 2^{127}$  (negative overflow)
  2. Negative numbers greater than  $-2^{-149}$  (negative underflow)
  3. Zero
  4. Positive numbers less than  $2^{-149}$  (positive underflow)
  5. Positive numbers greater than  $(2 - 2^{-23}) \times 2^{127}$  (positive overflow)

# Ranges of Floating point numbers

- ✓ Overflow generally means that values have grown too large to be represented.  
Underflow is a less serious problem because it just denotes a loss of precision, which is guaranteed to be closely approximated by zero.

The total effective range of finite IEEE floating-point numbers is

	<b>Binary</b>	<b>Decimal</b>
<b>Single Precision</b>	$\pm (2 - 2^{-23}) \times 2^{127}$	approximately $\pm 10^{38.53}$
<b>Double Precision</b>	$\pm (2 - 2^{-52}) \times 2^{1023}$	approximately $\pm 10^{308.25}$

# IEEE 754 - Special Operations

Operation	Result
$n \div \pm\text{Infinity}$	0
$\pm\text{Infinity} \times \pm\text{Infinity}$	$\pm\text{Infinity}$
$\pm\text{nonZero} \div \pm 0$	$\pm\text{Infinity}$
$\pm\text{finite} \times \pm\text{Infinity}$	$\pm\text{Infinity}$
$\text{Infinity} + \text{Infinity}$ $\text{Infinity} - -\text{Infinity}$	$+\text{Infinity}$

Operation	Result
$-\text{Infinity} - \text{Infinity}$ $-\text{Infinity} + -\text{Infinity}$	$-\text{Infinity}$
$\pm 0 \div \pm 0$	NaN
$\pm\text{Infinity} \div \pm\text{Infinity}$	NaN
$\pm\text{Infinity} \times 0$	NaN
$\text{NaN} == \text{NaN}$	False