COMPUTER ORGANIZATION AND ARCHITECTURE

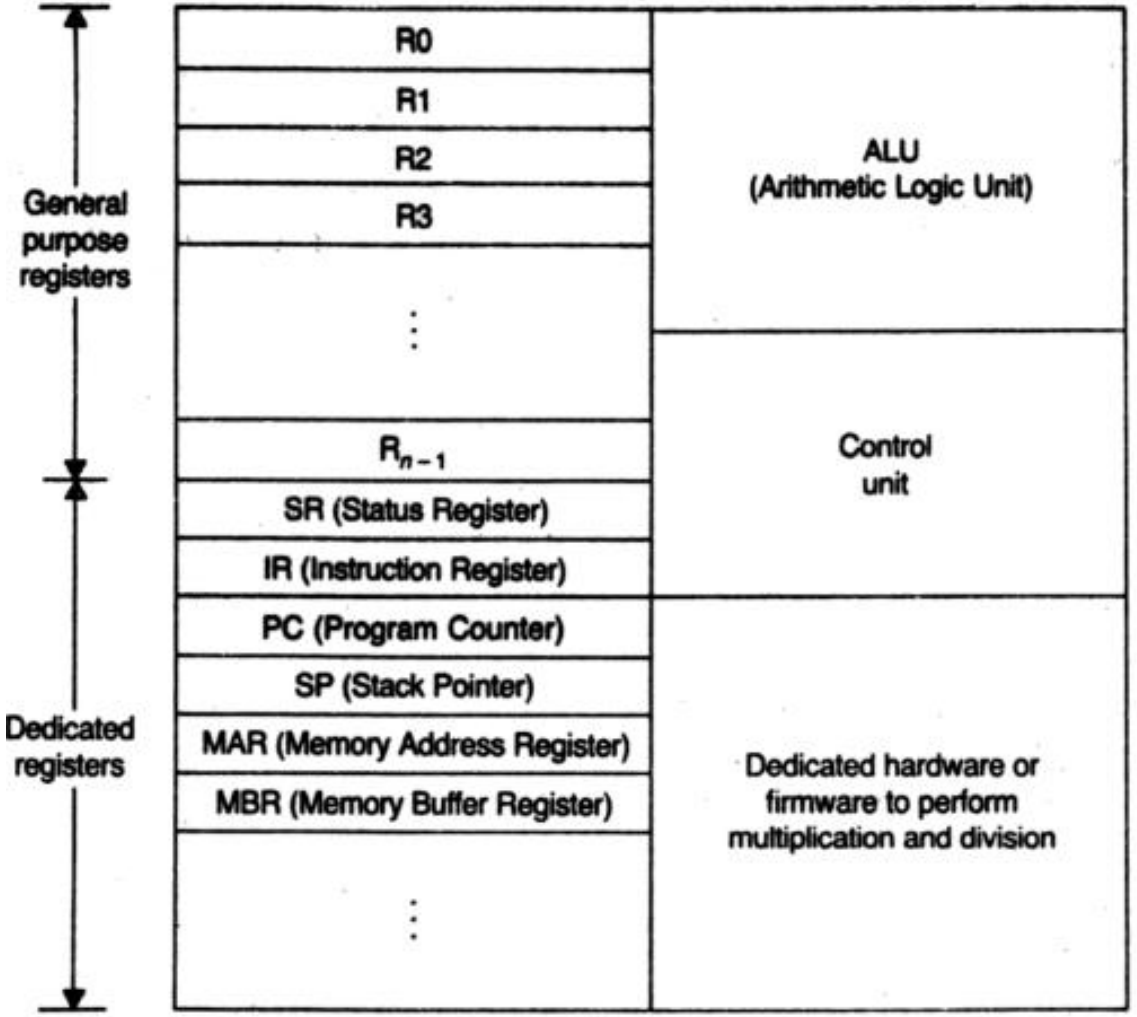
Execution Unit

Dr. Bore Gowda S B
Additional Professor
Dept. of ECE
MIT, Manipal

Introduction

A typical CPU model

- A typical CPU model is shown in figure
- A conventional CPU consists of the following:
  - ✓ General purpose registers
  - ✓ Dedicated registers
  - ✓ An ALU
  - ✓ Dedicated hardware or firmware elements that perform a special operations such as multiplication or division
  - ✓ A control unit



Register Section

- CPU with many register reduces the number of memory access
- ***General purpose register:*** may be configured as an accumulator, address pointer or a data pointer
- ***Dedicated register:*** *used for some specific tasks*

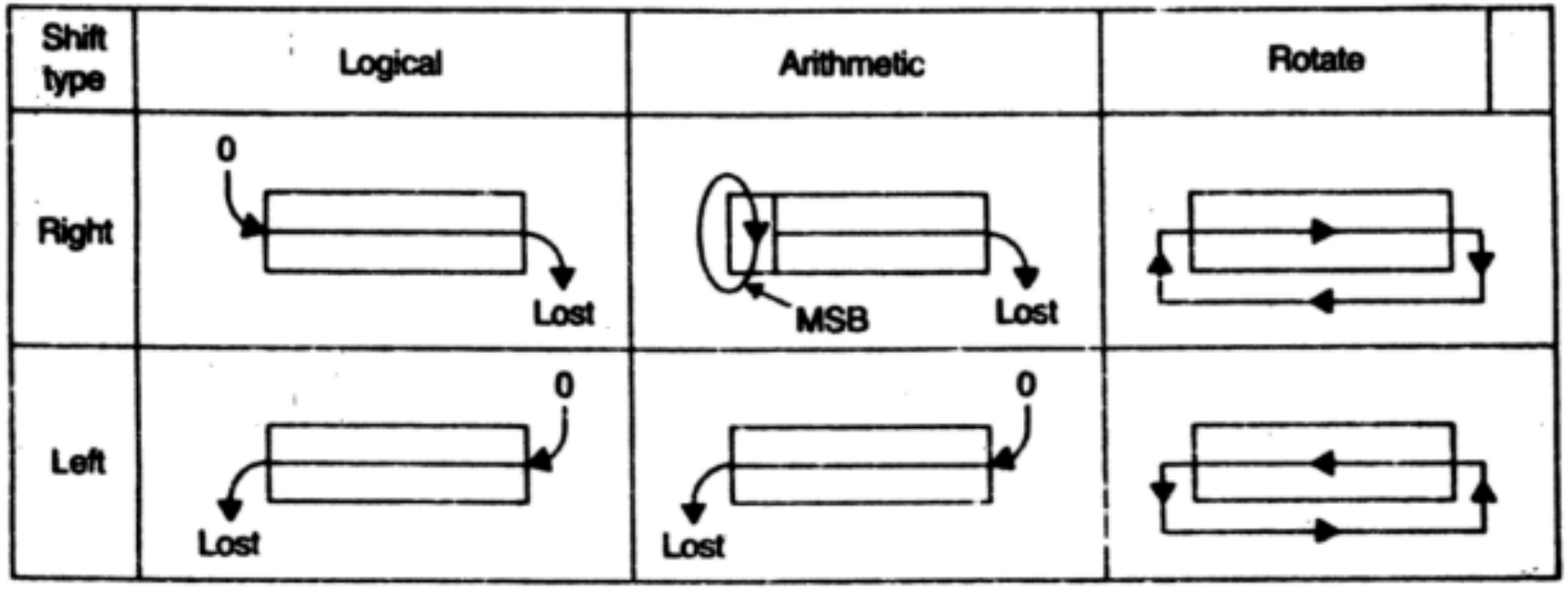
- Commonly known special purpose registers and their tasks rre:

REGISTER	TASK
PC	Usually holds the address of the next instruction to be executed.
SP	Usually holds the address of the top element of the stack.
IR	Holds the instruction code currently being executed.
MAR	Holds the address of the data item to be retrieved from the main memory.
MBR	Holds the data item retrieved from the main memory.
SR or PSW	Holds the condition code flags and other information that describe the status of the running program.

General Register Design

Typical shift operations:

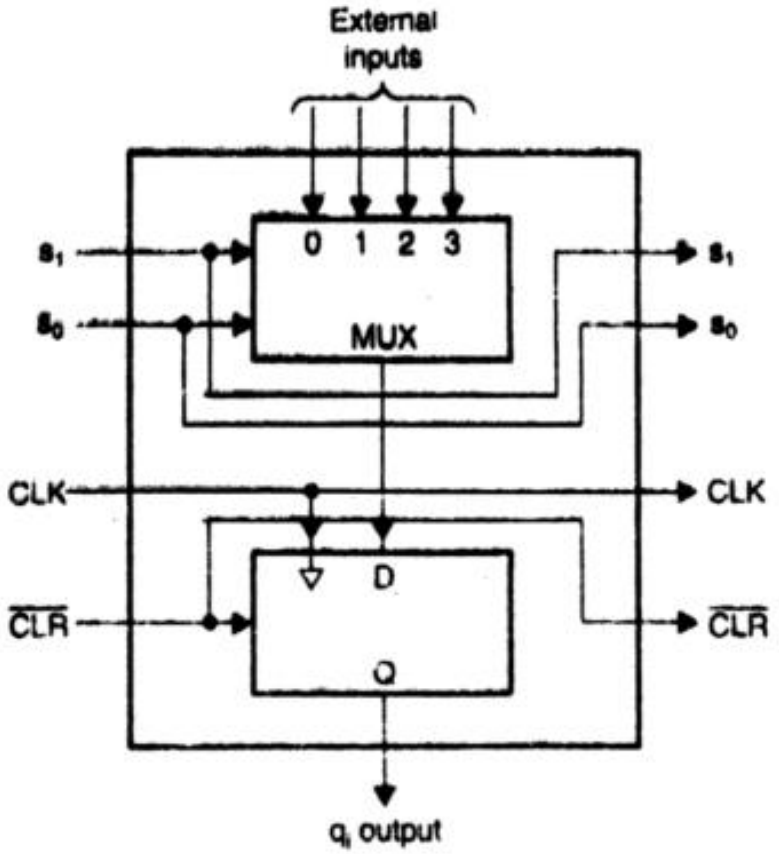
- GPR is to store address or data, then to be able to retrieve the data when needed
- GPR is also capable of manipulating the stored data by shift left or shift right operation
- Logical shift operations
- Arithmetic shift operations



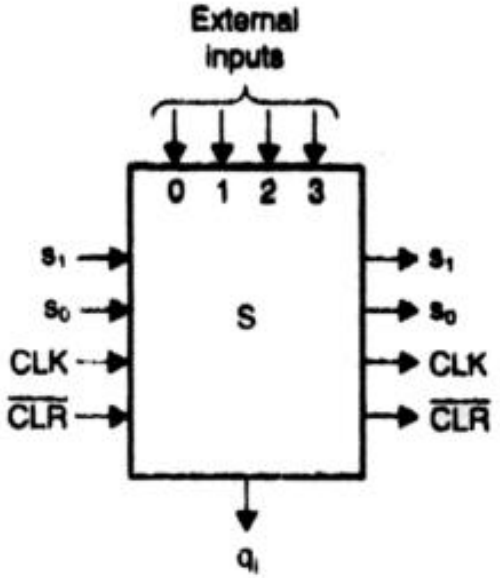
General Register Design

Basic cell for designing GPR

- The hypothetical instruction set of a computer consists of eight instructions, I_0, I_1, I_2,,,,,,I_7. The relative frequency of these instructions are as follows:



a. Internal Organization of the Basic Cell S

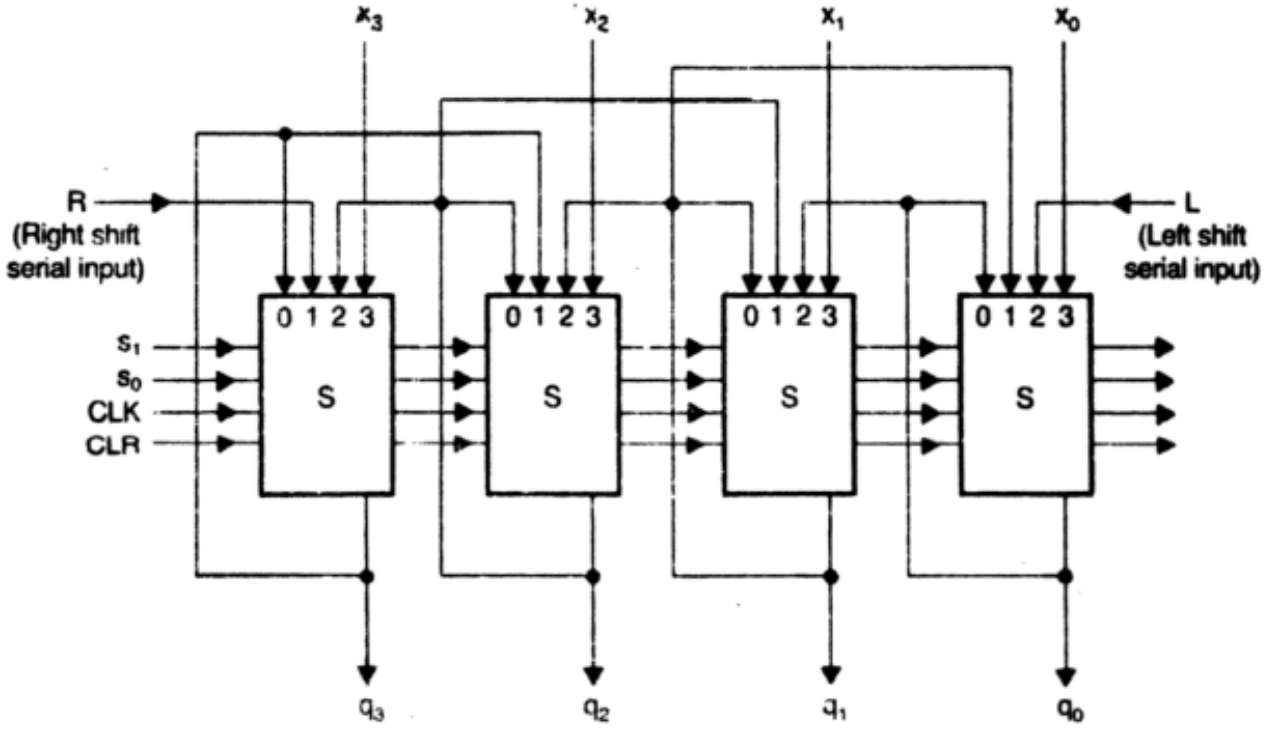


b. Block Diagram of the Basic Cell S

General Register Design

4-bit GPR:

- Parallel load, shift left, shift right, serial loading of data



Selection Inputs		Clock Input	Clear Input	Operation
s_1	s_0	CLK	$\overline{CLR}$	
X	X	X	0	Clear
0	0	↑	1	No operation
0	1	↑	1	Shift right
1	0	↑	1	Shift left
1	1	↑	1	Parallel load

Note: X – don't care; ↑ : rising edge of the clock

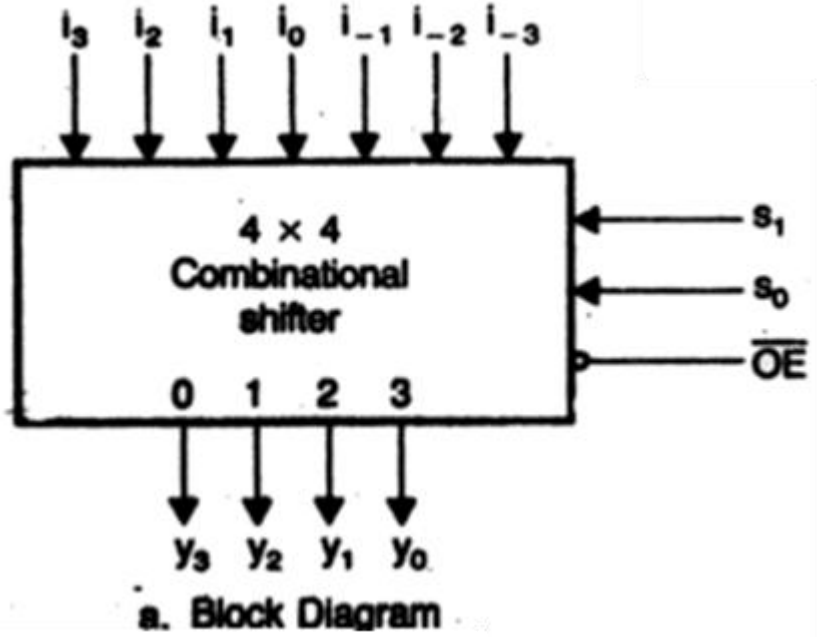
Three Variations of the Right Shift Operations

s_1s_0 = 01	
Value of R	Shift realized
0	Logical right shift
q_3	Arithmetic right shift
q_0	Rotate right

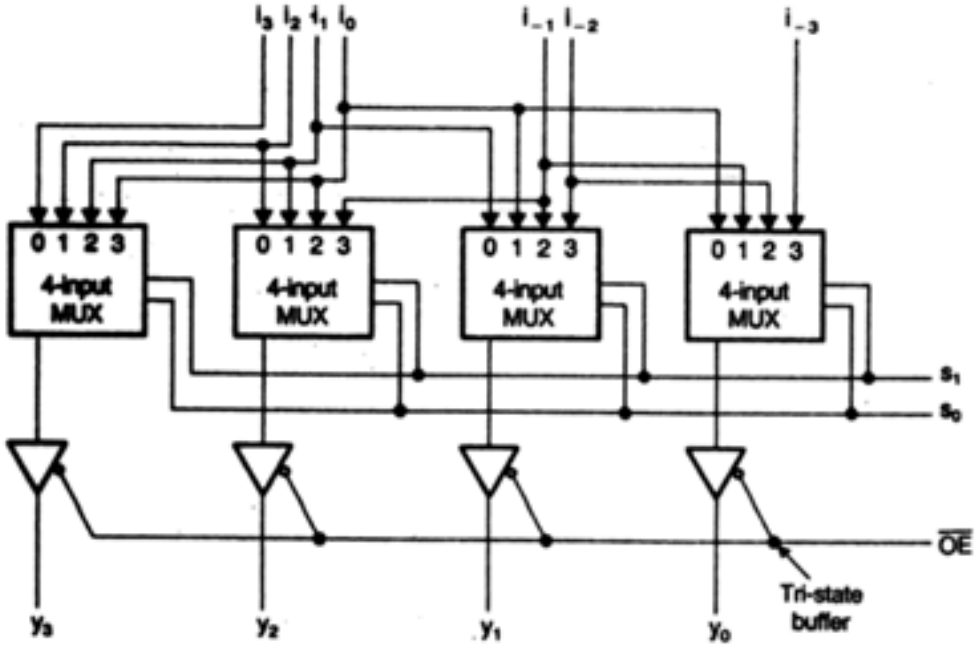
General Register Design

4x4 Combinational shifter design:

- The speed of a flip flop-based shifter is a function of the clock frequency
- A high-speed shifter can be designed using combinational circuit components such as a multiplexer



a. Block Diagram



b. Internal Schematic

General Register Design

4x4 Combinational shifter design:

	Shift Count		Output				
$\overline{OE}$	s_1	s_0	y_3	y_2	y_1	y_0	Comment
1	X	X	Z	Z	Z	Z	Output lines float
0	0	0	i_3	i_2	i_1	i_0	Pass (no shift)
0	0	1	i_2	i_1	i_0	i_{-1}	Left shift once
0	1	0	i_1	i_0	i_{-1}	i_{-2}	Left shift twice
0	1	1	i_0	i_{-1}	i_{-2}	i_{-3}	Left shift three times

Note: Z—High-impedance state.
X—Don't care.

c. Truth Table

$$Y_3 = s_1's_0'i_3 + s_1's_0i_2 + s_1s_0'i_1 + s_1s_0i_0$$

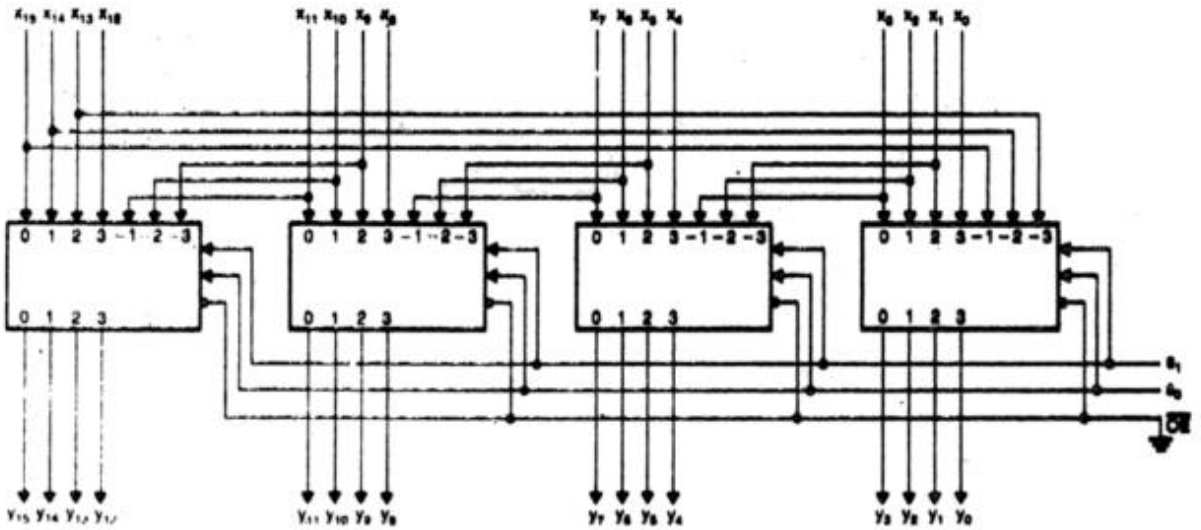
$$Y_2 = s_1's_0'i_2 + s_1's_0i_1 + s_1s_0'i_0 + s_1s_0i_{-1}$$

$$Y_1 = s_1's_0'i_1 + s_1's_0i_0 + s_1s_0'i_{-1} + s_1s_0i_{-2}$$

$$Y_0 = s_1's_0'i_0 + s_1's_0i_{-1} + s_1s_0'i_{-2} + s_1s_0i_{-3}$$

General Register Design

- **Combinational shifter capable of rotating 16-bit data to the left by 0, 1, 2 and 3 positions:**



a. Logic Diagram

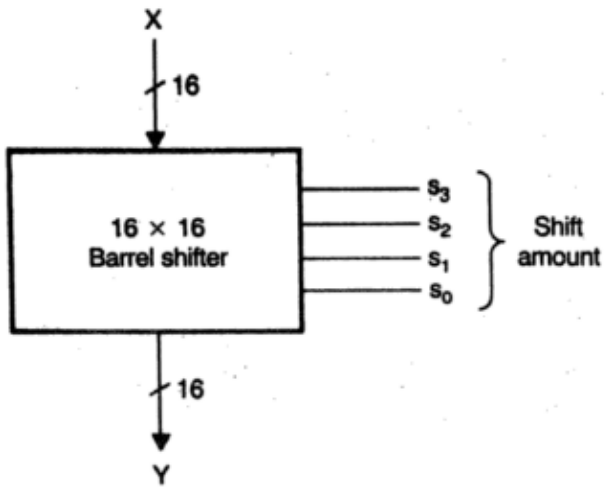
Shift Count	Output															
s_1 s_0	y_{15}	y_{14}	y_{13}	y_{12}	y_{11}	y_{10}	y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0 0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
0 1	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}
1 0	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}
1 1	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}

b. Truth Table

General Register Design

16x16 Barrel shifter:

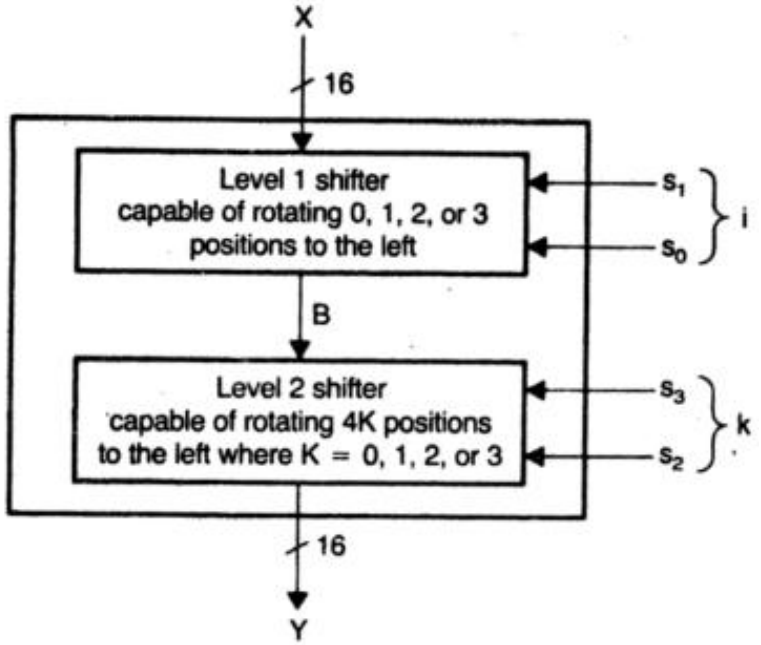
- Capable of rotating the given 16-bit data to the left n position where $0 \leq n \leq 15$



a. Block Diagram of a 16 × 16 Block Shifter

Shift Amount				Output													
s_3	s_2	s_1	s_0	y_{15}	y_{14}	y_{13}	y_{12}	y_{11}	y_{10}	y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	0	0	0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
0	0	0	1	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}
0	0	1	0	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}
0	0	1	1	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}
0	1	0	0	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}
0	1	0	1	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}
0	1	1	0	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}
0	1	1	1	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9
1	0	0	0	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8
1	0	0	1	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7
1	0	1	0	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6
1	0	1	1	x_4	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5
1	1	0	0	x_3	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4
1	1	0	1	x_2	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3
1	1	1	0	x_1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2
1	1	1	1	x_0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1

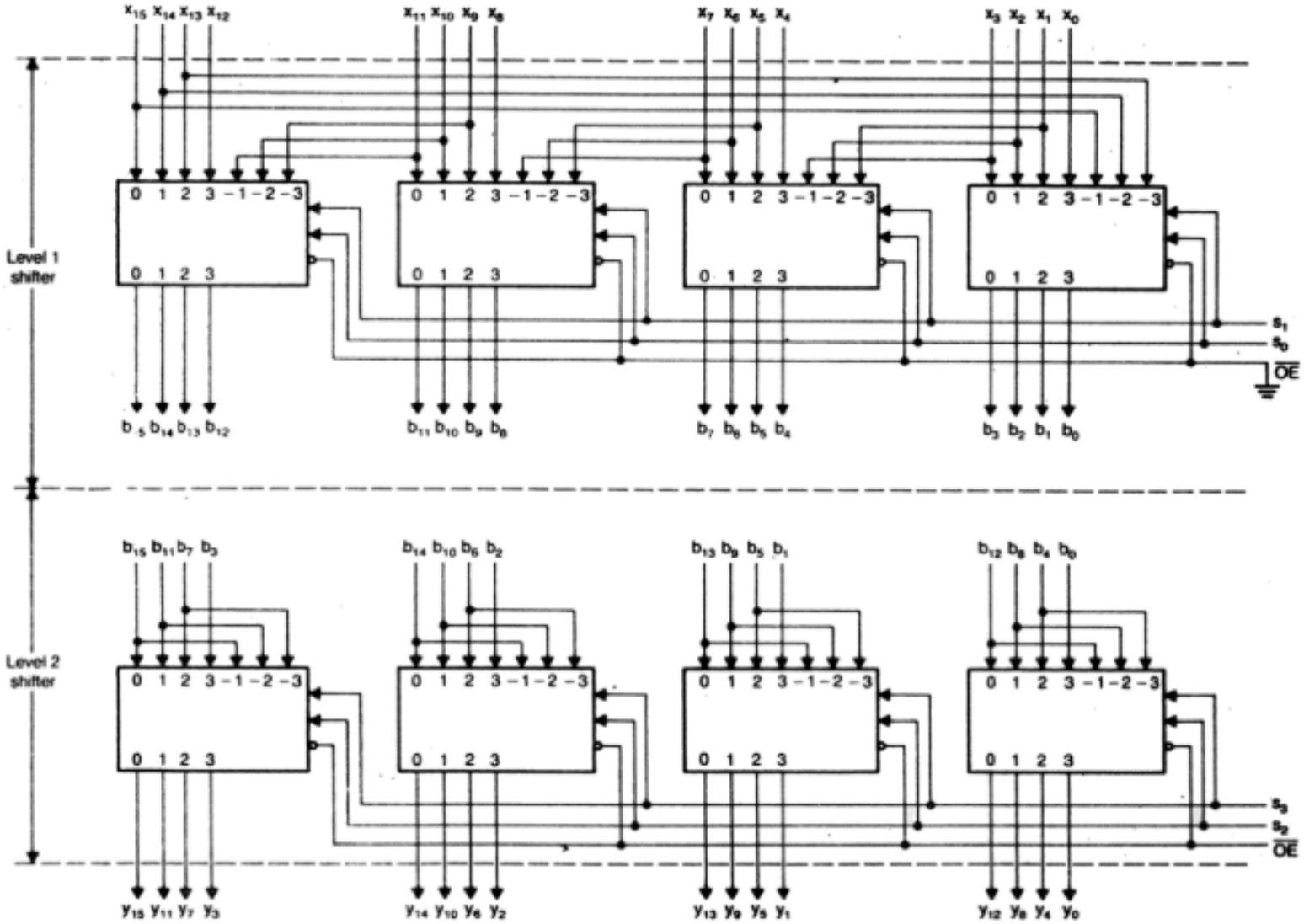
b. Truth Table of the 16 × 16 Barrel Shifter



c. Functional Block Diagram of a 16 × 16 Barrel Shifter

General Register Design

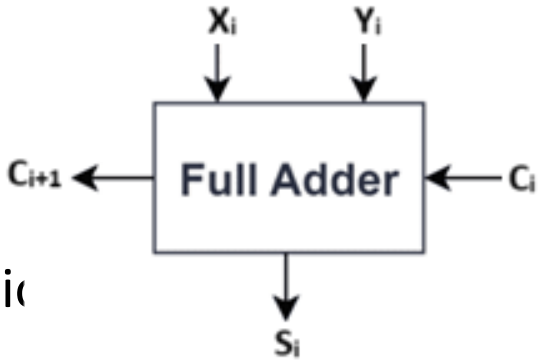
- **Complete Logic diagram of full 16-bit Barrel shifter:**



Adder Design

Full adder:

- Addition is the basic operation performed by an ALU
- Operation is versatile
- A- B = A + 2's complement of B
- B*C may be obtained by adding A to itself for C - 1 times
- The speed of the hardware unit is essential to the efficient operati of execution unit



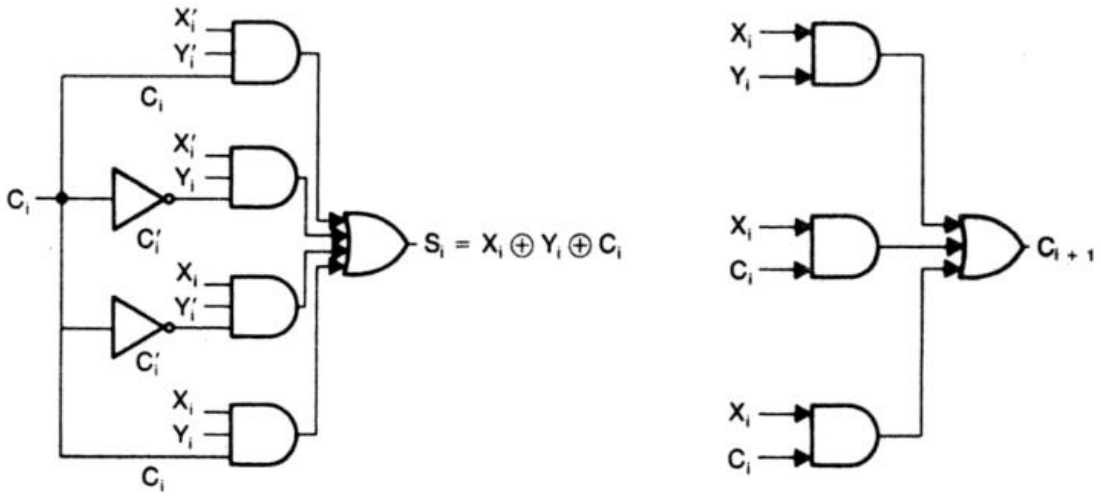
X_i	Y_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \overline{x_i}.\overline{y_i}.c_i + \overline{x_i}.y_i.\overline{c_i} + x_i.\overline{y_i}.\overline{c_i} + x_i.y_i.c_i$$

$$c_{i+1} = x_i.y_i + x_i.c_i + y_i.c_i$$

Adder Design

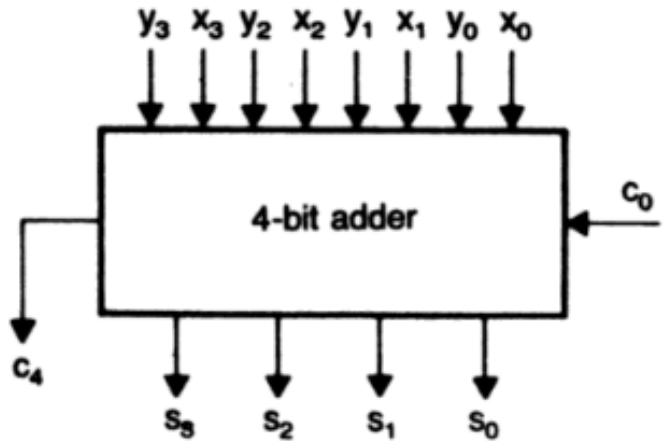
Full adder:



- To generate ***C_{i+1} from C_i, 2 gate delays*** are required
- To generate ***sum S_i from C_i, 3 gate delays*** are required
- Assume that the gate delay is Δ time units and the actual value of it is decided by the technology used
- For example, TTL logic circuits have a Δ will be 10 ns

Adder Design

4-bit ripple carry adder/Carry Propagate Adder(CPA)

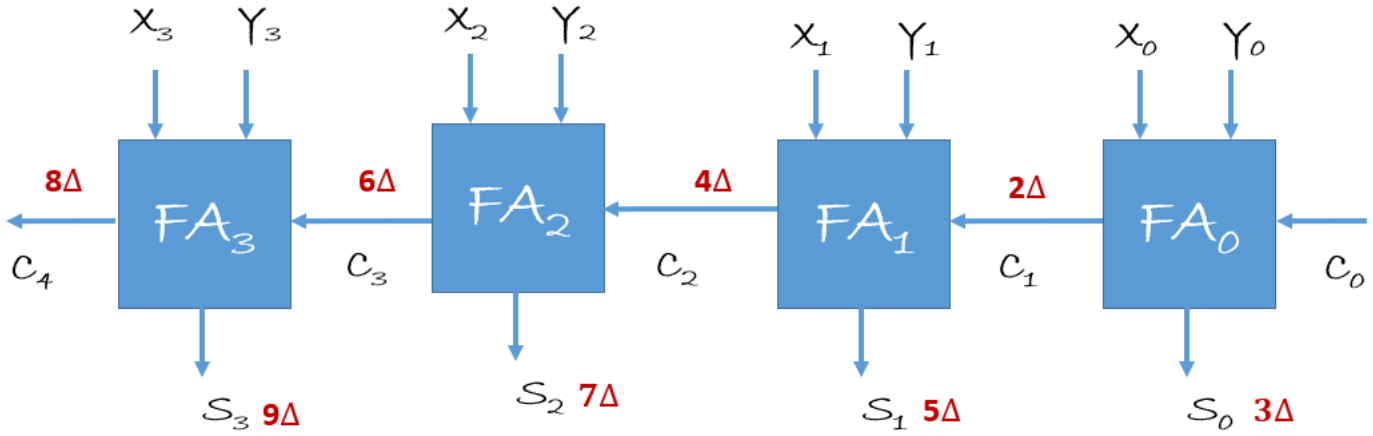


Total delay = (n-1)* delay between each block + last block delay

Wnere n is number of Full Adders

For n = 4 and if gate delay is 0.5ns, we get

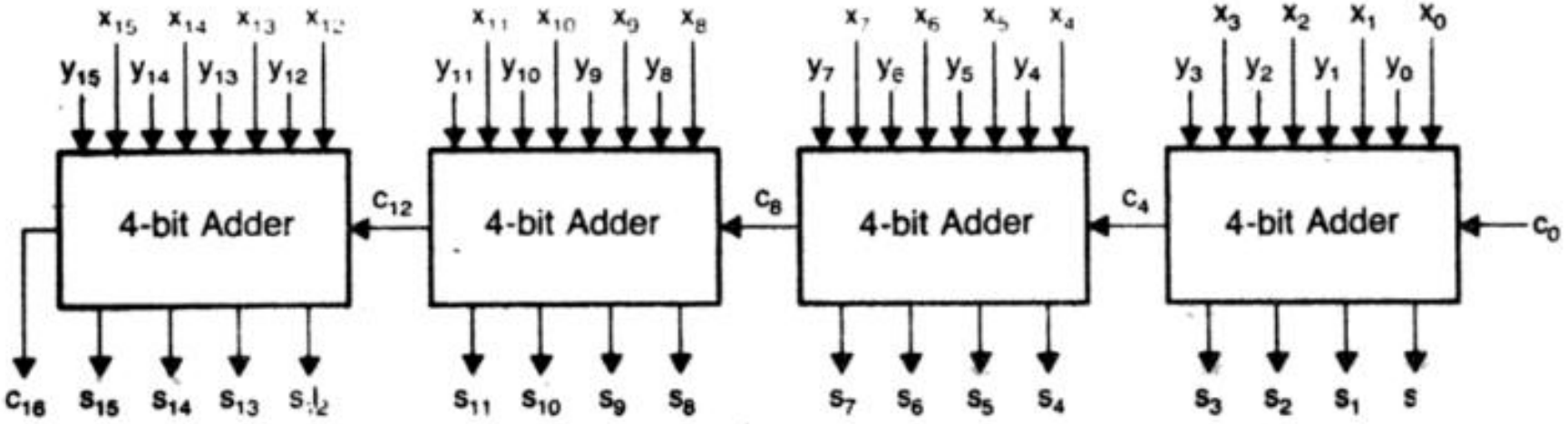
Delay = (4 – 1) * 1 + 1.5 = 4.5 ns



Adder Design

16-bit adder using 4-bit ripple carry adder as basic block

- Design of an n-bit CPA is straightforward the carry propagation time limits the speed of operation
- For example: 16-bit CPA when the addition operation is completed only when the sum bits S_0 through are S_{15} available
- In the worst case carry propagate through 15 full adders
- **Total Time delay**= Time taken for carry to propagate through 15 full adder + Time taken to generate S_{15} from C_{14} = $15 * 2\Delta + 3\Delta = 33\Delta$



- *Circuit is simple and easy to build*
- *Ripple of carry is causing delay and it increases as **n** increases.*

Fast Adder Design

Carry Look-Ahead Adder (CLA)

- It is a type of adder used in digital logic that improves the speed of arithmetic operations by reducing the time it takes to calculate carry bits
- Key Concepts:
  1. **Carry Propagation:** In a traditional ripple carry adder, each bit must wait for the carry bit from the previous bit to be calculated, which can be slow.
  2. **Carry Look-Ahead**: The CLA adder speeds up this process by calculating the carry bits in advance, using the concepts of generate and propagate.

- How It Works:
  1. **Carry Generate (G):** A bit generates a carry if both of its input bits are 1.
  2. **Carry Propagate (P):** A bit propagates a carry if at least one of its input bits is 1.

Adder Design

Carry Look-Ahead Adder (CLA)

- For Full adder, we know that:

$$c_{i+1} = x_i.y_i + x_i.c_i + y_i.c_i$$

- **Carry generate function**

$$G_i = x_i.y_i$$

- **Carry propagate function**

$$P_i = x_i + y_i$$

- **Carry function**

$$c_{i+1} = G_i + P_i.C_i$$

- These equations show that a carry signal will be generated in two cases:

1. if both bits x_i and y_i are 1
2. if either x_i or y_i is 1 and the carry-in C_i is 1.

Adder Design

4-bit Carry Look-Ahead Adder (CLA)

- Apply these equations for a 4-bit adder:

$$C_1 = G_0 + P_0C_0$$
$$C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$$
$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$
$$C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

- These expressions show that C_2, C_3 and C_4 do not depend on its previous carry-in.
- Therefore C_4 does not need to wait for C_3 to propagate.
- As soon as C_0 is computed, C_4 can reach steady state.
- The same is also true for C_2 and C_3

Adder Design

4-bit Carry Look-Ahead Adder (CLA)

- Apply these equations for a 4-bit adder:

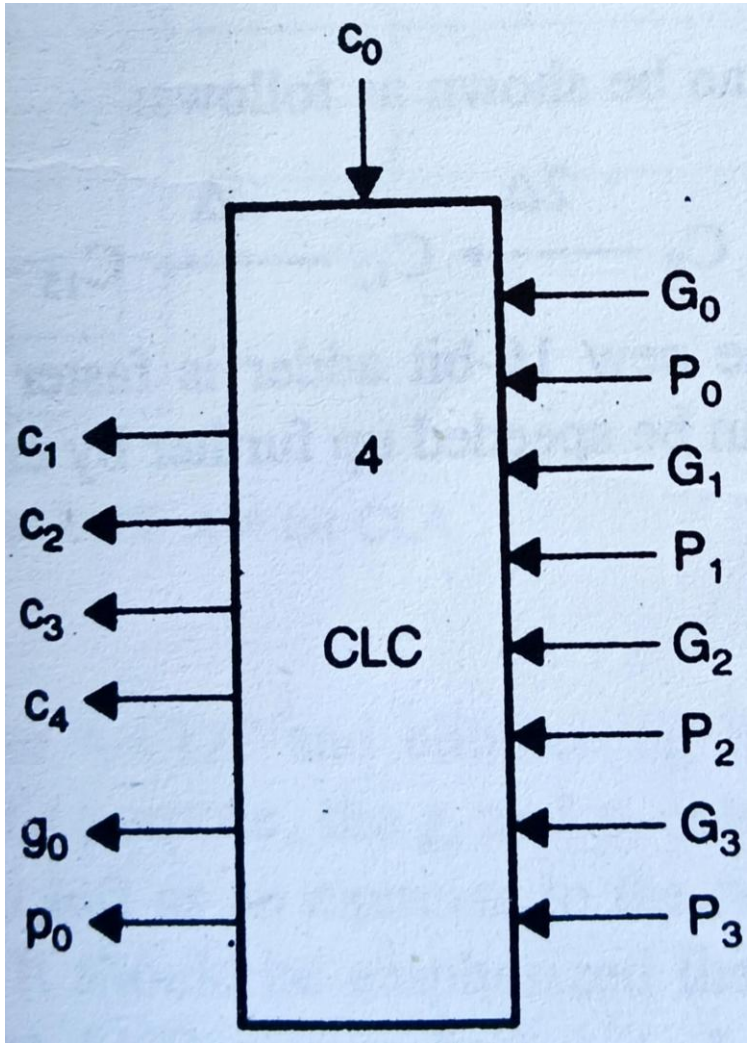
$$C_1 = G_0 + P_0C_0$$
$$C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$$
$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$
$$C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

- These expressions show that C_2, C_3 and C_4 do not depend on its previous carry-in.
- Therefore C_4 does not need to wait for C_3 to propagate.
- As soon as C_0 is computed, C_4 can reach steady state.
- The same is also true for C_2 and C_3
- For these reason, the equations are called *carry look ahead equations*
- The hardware that implements these equations is called *4-stage carry-look ahead circuit (4 CLC)*

General Register Design

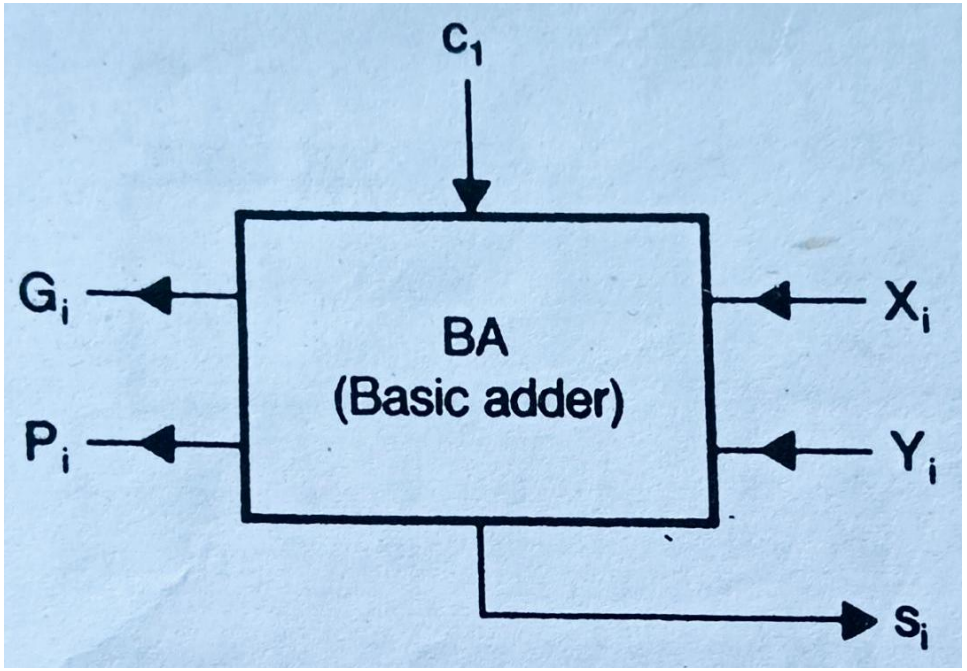
4-stage Carry Look-Ahead circuit(4-CLC)

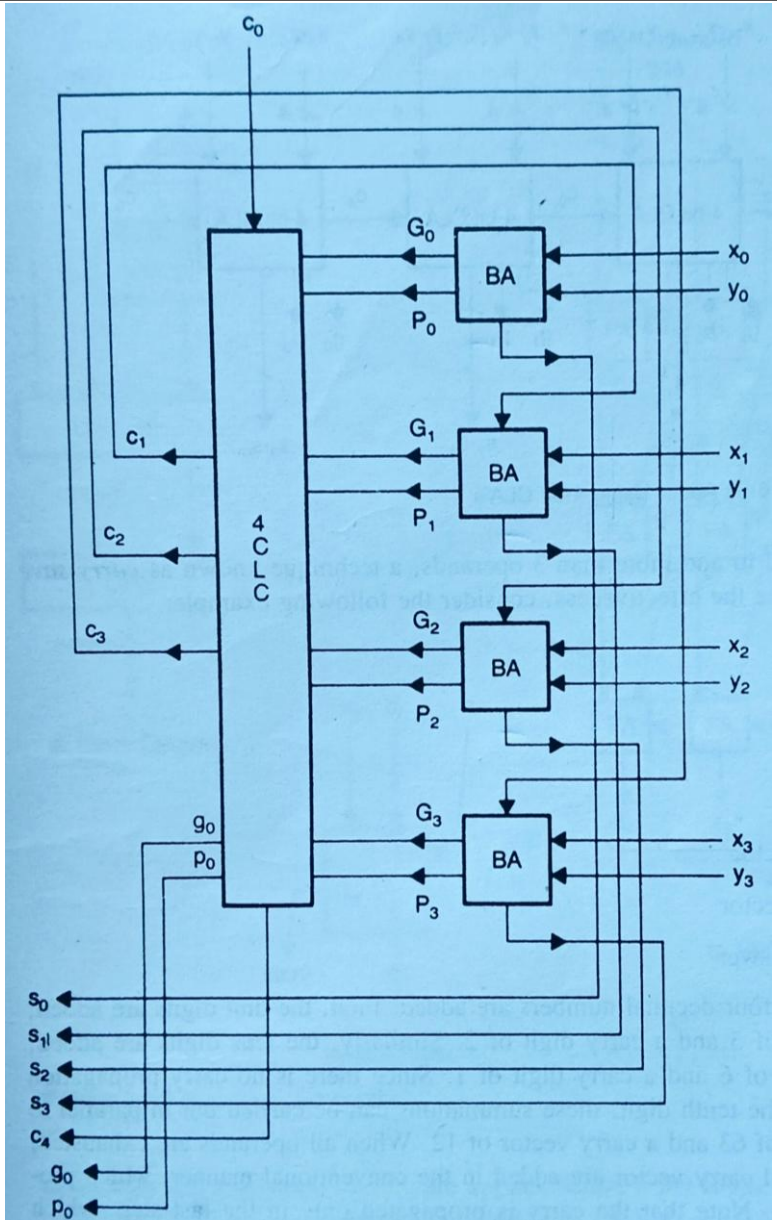
- A 4-CLC can be implemented as a teo-level AND/OR logic circuit
- To generate C4, 5-input **AND** gate and 4-input **OR** gate required
- If $G_i \leq (0 \leq i \leq 3)$ is available, then all C_i's with $(1 \leq i \leq 4)$ can be generated in 2 gate delays
- The output G_0 and P_0 are useful to obtain a higher order look ahead system



General Register Design

- 4-bit CLA using basic adder

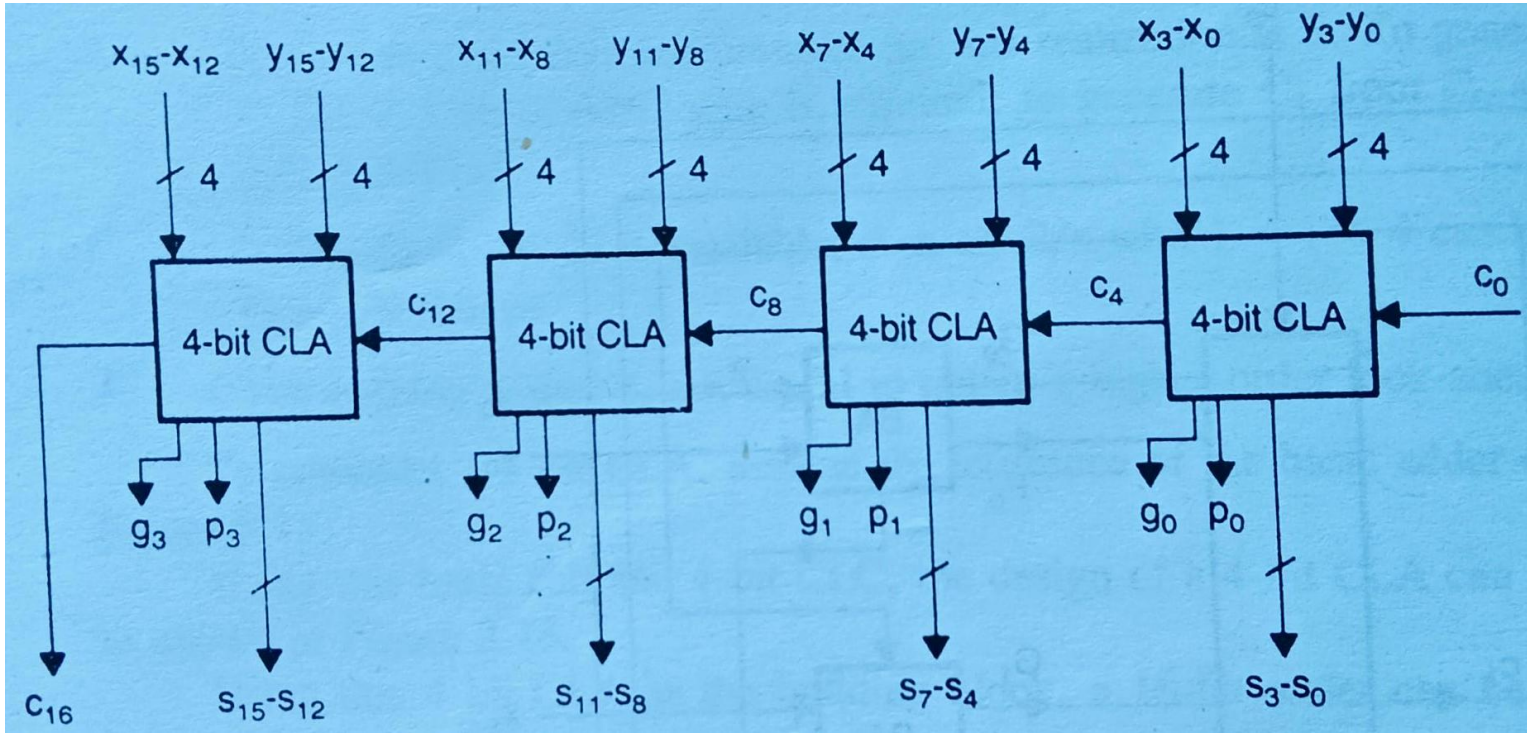




General Register Design

16-bit CLA using 4-bit CLA

- A 16-bit CLA is typically divided into four 4-bit CLAs.
- Each 4-bit CLA generates its own carry-out signals, which are then used to compute the carry-out for the next higher level.



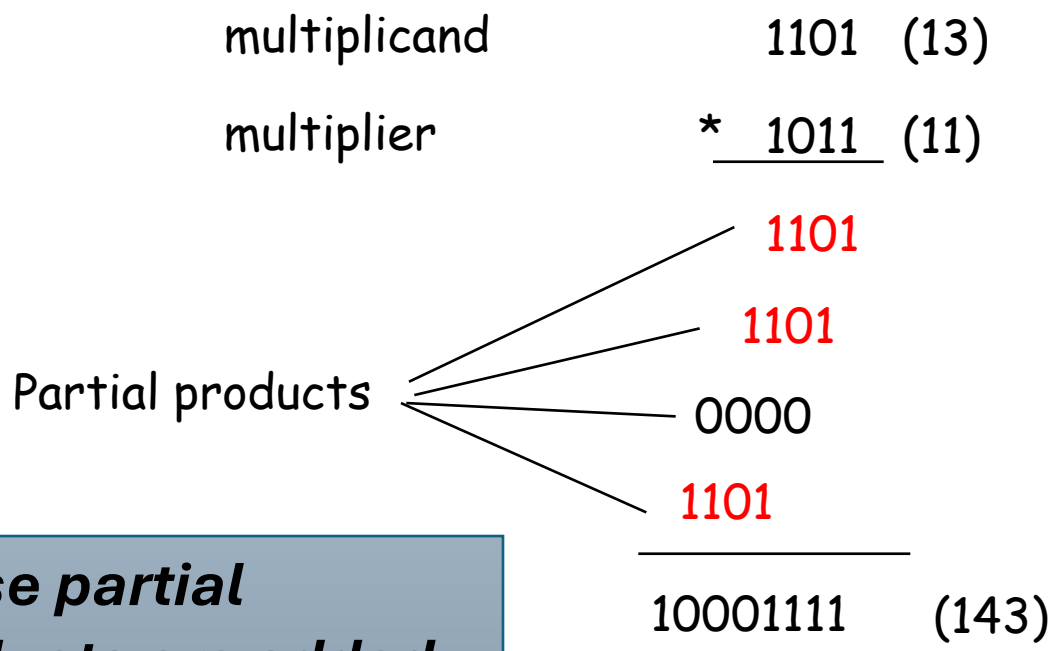
	Delay
For G_i P_i generation from X_iY_i $(0 \leq i \leq 15)$	Δ
To generate C_4 from C_0	2Δ
To generate C_8 from C_4	2Δ
To generate C_{12} from C_8	2Δ
To generate C_{15} from C_{12}	2Δ
To generate S_{15} from C_{15}	3Δ
Total delay	12Δ

A graphical illustration of this calculation can be shown as follows:

Data available $\xrightarrow{\Delta}$ G_iP_i $\xrightarrow{2\Delta}$ C_4 $\xrightarrow{2\Delta}$ C_8 $\xrightarrow{2\Delta}$ C_{12} $\xrightarrow{2\Delta}$ C_{15} $\xrightarrow{3\Delta}$ S_{15}

Multiplication of Binary numbers

Basic Concept



These partial products are added to get the final product.

product of two 4-bit numbers is an 8-bit number

Binary multiplication is easy

0 × multiplicand = 0

1 × multiplicand = multiplicand

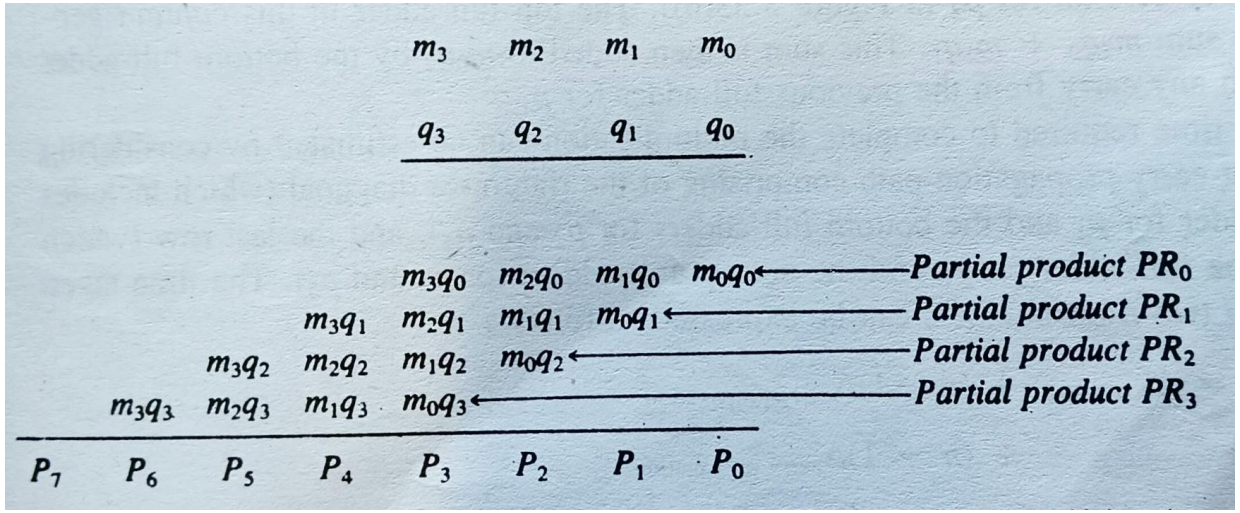
It's interesting to note that binary multiplication is a sequence of shifts and adds of one number (depending on the bits in the second number).

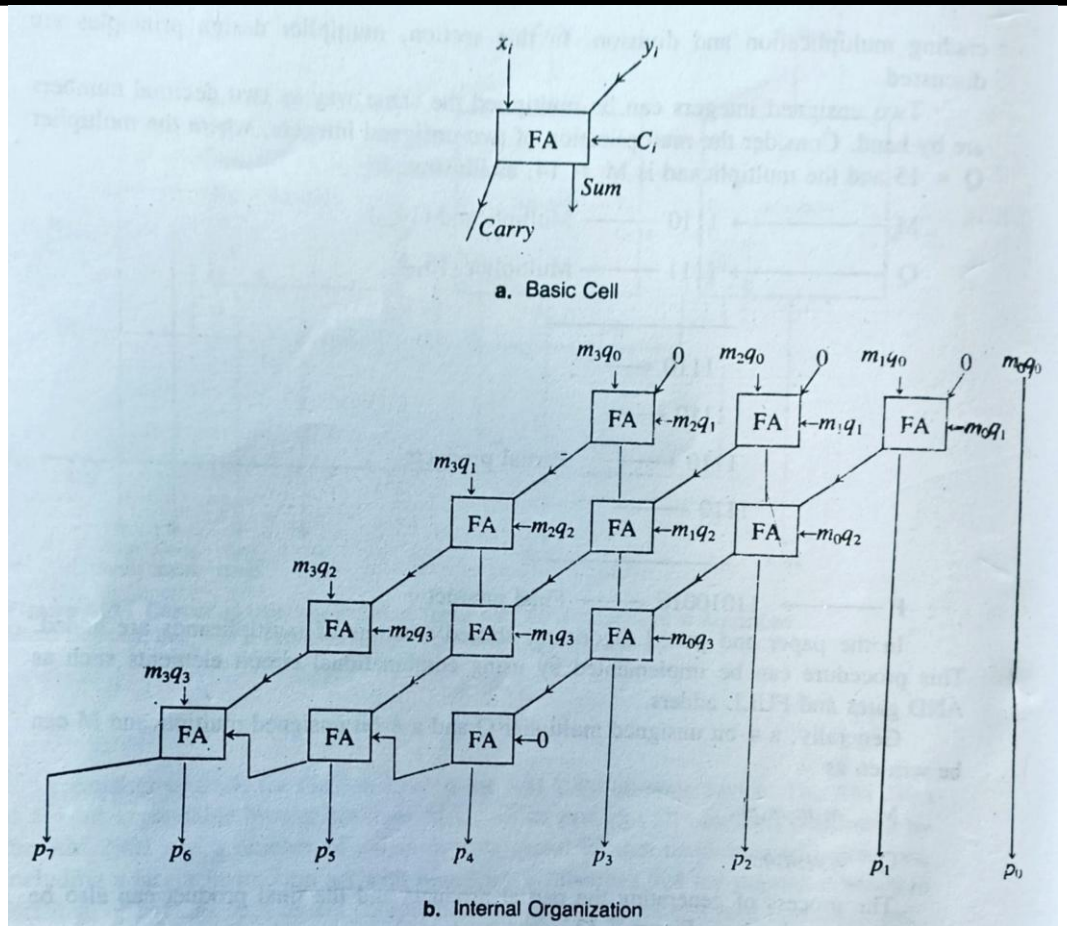
Multiplication of Binary numbers

- **4x4 array multiplication**

M: $m_3 m_2 m_1 m_0$

Q: $q_3 q_2 q_1 q_0$





- Each cross-product term in this can be generated using an AND gate
- This requires 16 AND gate to generate all cross-product terms that are summed by full adder arrays
- It is nonadditive multiplier(NM), since it does not include any additive inputs

$$T(n) = \Delta_{\text{AND gate}} + (n-1)\Delta_{\text{carry propagation}} + (n-1)\Delta_{\text{carry propagation}}$$

Multiplication of Binary numbers

4x4 array multiplication

- An additive multiplier(AM) includes an extra input, it computes products of the form

$$P = M*Q + R$$

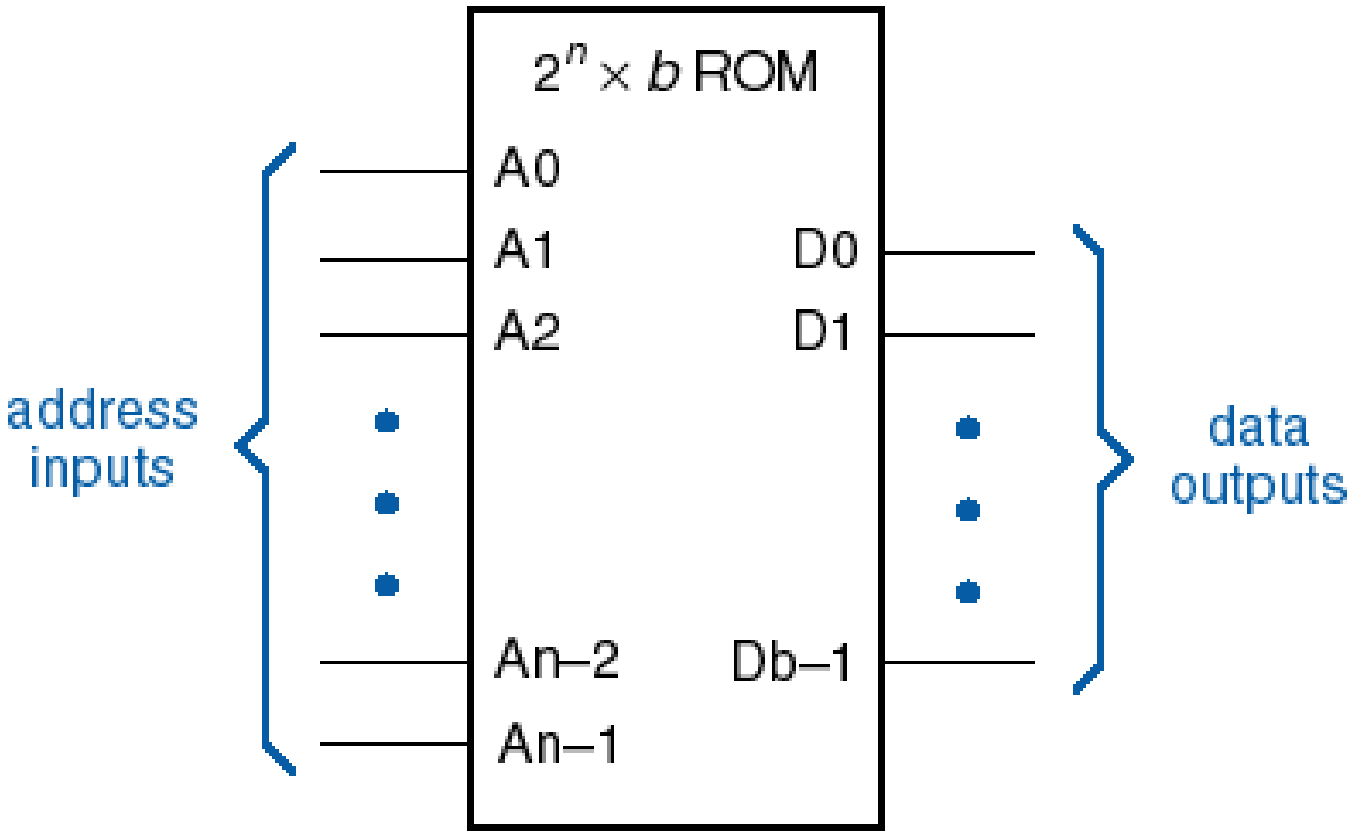
- Both NM and an AM are available as standard IC blocks

- Simple and repetitive structure
- Can multiply only unsigned numbers
- Need additional logic to multiply signed numbers
- Too big circuit and utilization factor is too less
- Delay grows as the number of bits to be multiplied increases

Multiplication of Binary numbers

- **Read Only Memory**

ROM is a combinational circuit; it's a truth-table

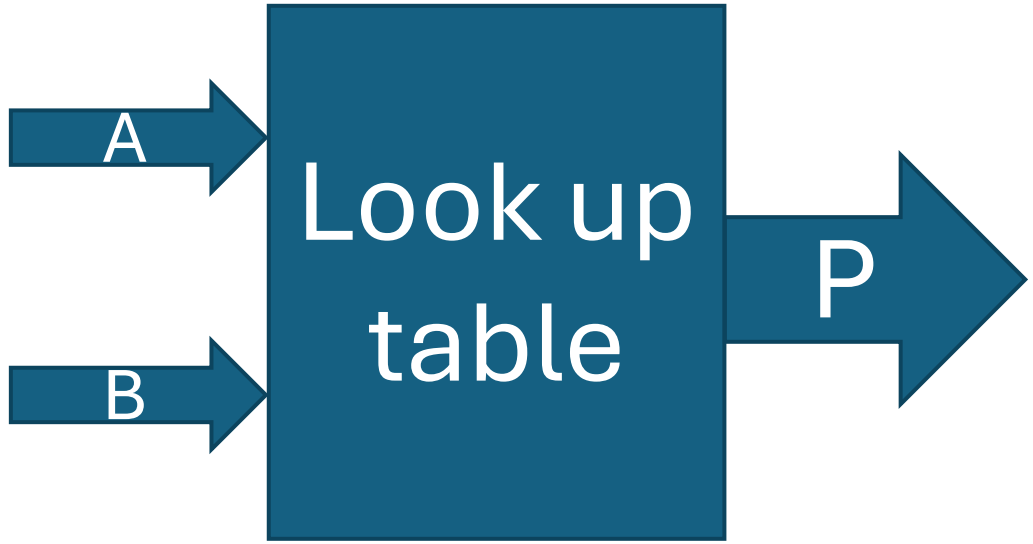


- Can perform any combinational logic function
  - Address inputs = function inputs
  - Data outputs = function outputs

you give input numbers to be multiplied and it will give you the product!!

Multiplication of Binary numbers

- **Read Only Memory**



Address input

Data output

A	B	Product
00	00	0000
00	01	0000
00	10	0000
00	11	0000
01	00	0000
01	01	0001
01	10	0010
01	11	0011
11	00	0000
11	01	0011
11	10	0110
11	11	1001

Consider A and B to be 2-bit numbers, How many bits will be in product, P?
What is the memory size of this ROM?

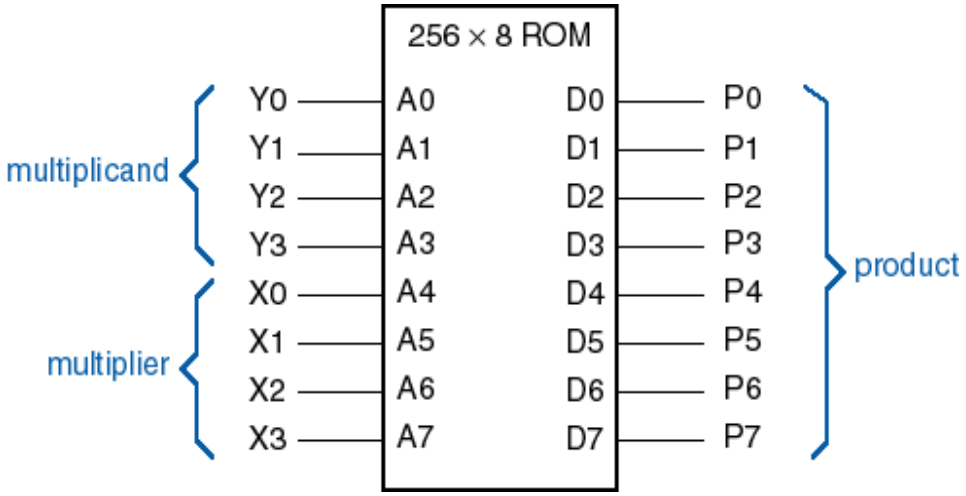
8 bytes!!

16 locations

Multiplication of Binary numbers

4x4 ROM based multiplier example

How many address and data lines?



	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10:	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
20:	00	02	04	06	08	0A	0C	0E	10	12	14	16	18	1A	1C	1E
30:	00	03	06	09	0C	0F	12	15	18	1B	1E	21	24	27	2A	2D
40:	00	04	08	0C	10	14	18	1C	20	24	28	2C	30	34	38	3C
50:	00	05	0A	0F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
60:	00	06	0C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
70:	00	07	0E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
80:	00	08	10	18	20	28	30	38	40	48	50	58	60	68	70	78
90:	00	09	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A0:	00	0A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B0:	00	0B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C0:	00	0C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D0:	00	0D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E0:	00	0E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F0:	00	0F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

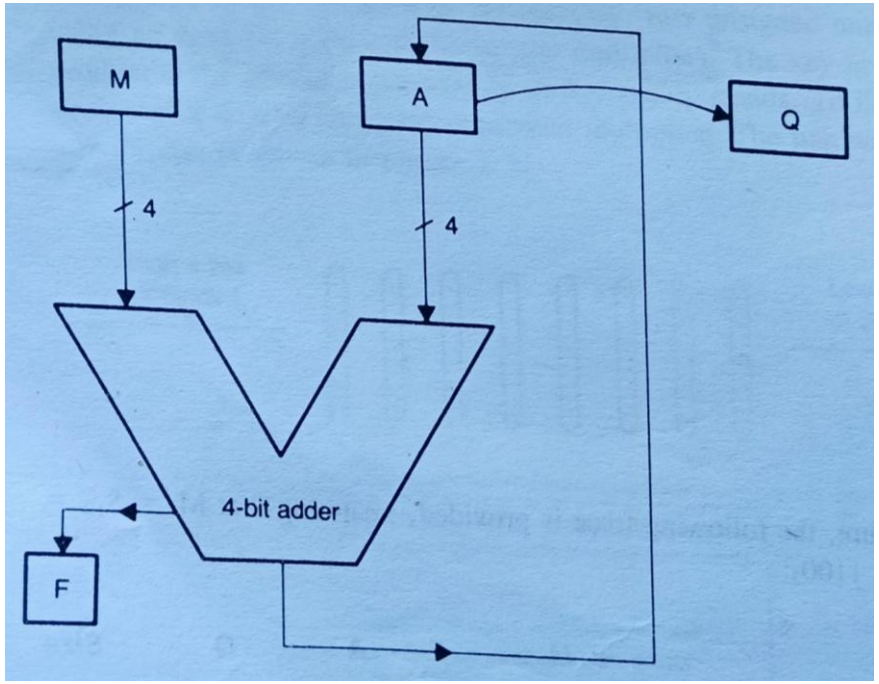
What is the memory size of this ROM?

265 bytes

Multiplication of Binary numbers

Sequential multiplier

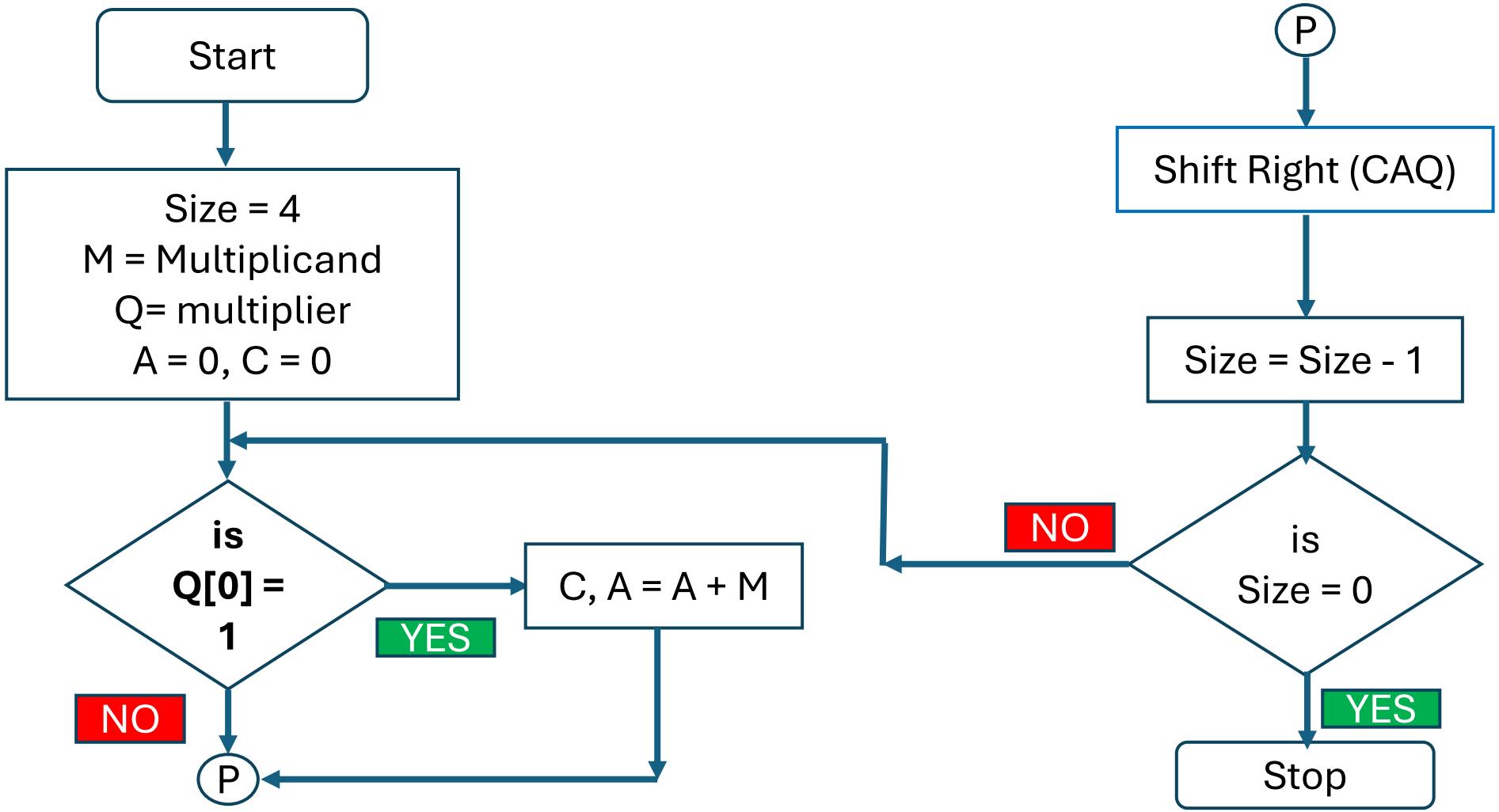
- Sequential multiplier use the fact that multiplication can be achieved using repeated addition
- It can multiply unsigned integers
- Next slides has a flowchart that can multiply two 4-bit unsigned numbers and a block diagram for implementing it
- **4x4 unsigned sequential multiplier**



Multiplication of Binary numbers

Sequential multiplier

Algorithm to multiply two unsigned 4-bit numbers



Multiplication of Binary numbers

Sequential multiplier

- **Example:** M = 1100, Q = 1001

Description	C	A	Q	Q[0]
Initialization	0	0000	1001	1
Ite-1, A = A+M	0	1100	1001	1
SHR CAQ	0	0110	0100	0
Ite-2, No add	0	0110	0100	0
SHR CAQ	0	0011	0010	0
Ite-3, No add	0	0011	0010	0
SHR CAQ	0	0001	1001	1
Ite-4, A=A+M	0	1101	1001	1
SHR CAQ	0	0110	1100	0

0xC*0x9 = 0x6C

Multiplication of Binary numbers

- **Sequential multiplier**

M = 01110, Q = 10100

Description	C	A	Q	Q[0]
Initialization	0	00000	10100	0
Ite-1, SHR CAQ	0	00000	01010	0
Ite-2, SHR CAQ	0	00000	00101	1
Ite-3 A=A+M	0	01110	00101	1
SHR CAQ	0	00111	00010	0
Ite-4, SHR CAQ	0	00011	10000	1
Ite- 5 A=A+M	0	10001	10000	1
SHR CAQ	0	01000	11000	0

Worst case: n additions and n right shift operations are needed
Speed increased

0x0E*0x14 = 0x118

Multiplication of Binary numbers

Signed multiplication

- There are three possible cases:
  1. M&Q are in signed magnitude form
  2. M and Q are in one's complement form
  3. M and Q are in Two's complement form
- **Case 1:**
  - Multiply the magnitudes either combinationally or sequentially
  - Sign of the product = sign of Multiplier XOR sig of Multiplicand
- **Case 2:**
  - Step 1: if multiplicand is negative, then compute the 1's complement of M
  - Step 2: if multiplier is negative, then compute the 1's complement of Q
  - Step 3: multiply the (n – 1) bits of the multiplier and the multiplicand either combinnationally or sequentially
  - Step 4: sign of the product = sign of multiplier XOR sign of multiplicand
  - Step 5: if sign of the product is negative, then compute the 1's of the result obtained in Step 3

Multiplication of Binary numbers

Signed multiplication

- **Case 3:**
  - When M & Q are in 2's complement form, the same procedure as in **Case 2** is repeated with the following exception:
  - Product must be 2's complemented when
    - both multiplicand and multiplier are negative or
    - sign of Multiplicand XOR Sign of Multiplier = 1
    - ***Example: M=1100 and Q=0111, take 2'c complement of M, then multiply with Q. since sign of product is –ve, take 2's complement of product.***
    - ***Result: 11100100***
  - **Demerits:**
  - extra processing when they are represented in 1's or 2's complement form
  - Overhead is maximum when multiplier and multiplicand are in 2's complement form, since incrementation is required besides the complementation.
  - Overhead can be eliminated by ***recoded multiplication approach***

Multiplication of Binary numbers

Booth's Recoded multiplication approach

- **String property:**
- *In a binary sequence a block of consecutive K ones may be replaced with a block of k - 1 consecutive 0's surrounded by the digits **1** and $\overline{1}$*

5 ones

0 0 $\overbrace{1\,1\,1\,1\,1}$ 0

By the string property it may be rewritten as follows;

0 0 11111 0

$\Downarrow$

0 1 0000$\overline{1}$ 0

- String property increases the density of zeros in a given multiplier
- number of additions operations to be performed is reduced
- allows for quick multiplication

- ***If a weight of -1 is assigned to the digit $\overline{1}$(over bar), assuming that the original sequence is in 2's complement form, both represent same number***

$$-2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$$

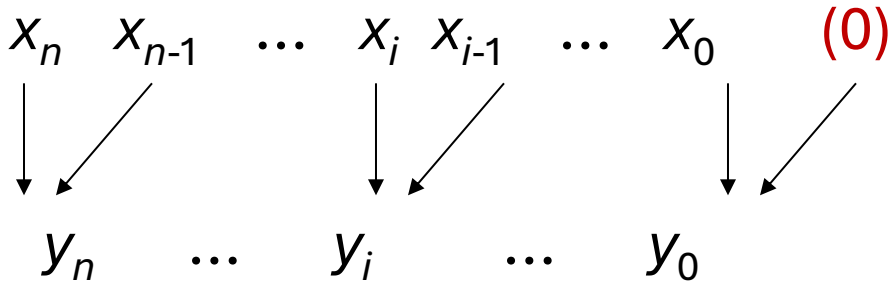
Original pattern: $0\,0\,1\,1\,1\,1\,1\,0 = 2^5 + 2^4 + 2^3 + 2^2 + 2^1 = 62_{10}$

Recoded pattern: $0\,1\,0\,0\,0\,0\,\overline{1}\,0 = 1.2^6 + (-1)2^1 = 62_{10}$

Multiplication of Binary numbers

Booth's Recoded multiplication approach

- Booth Observed that a String of 1's May be Replaced as: $2^j + 2^{j-1} + \cdots + 2^{i+1} + 2^i = 2^{j+1} - 2^i$



$y_i = x_{i-1} - x_i$

x_i	x_{i-1}	Comments	y_i
0	0	string of zeros	0
1	1	string of ones	0
1	0	beg. string of ones	-1
0	1	end string of ones	1

EXAMPLE: 0 0 1 1 1 1 0 0 1 1 (0)

0 1 0 0 0 $\bar{1}$ 0 1 0 $\bar{1}$

Booth's Recoding Drawbacks

- case can come up where number of 1s originally will be less than the recoded 1's

001010101 (0)

01$\bar{1}$1$\bar{1}$1$\bar{1}$1$\bar{1}$

Recode 15

01111 10001

Multiplication of Binary numbers

Booth's Algorithm

- Booth's Algorithm is a technique used in binary multiplication to optimize the process, especially for signed numbers.
- It reduces the number of partial products, which can simplify and speed up the multiplication.

- ***Key Concepts***
- **Recoding:** Booth's Algorithm recodes the multiplier to reduce the number of partial products. This is done by examining pairs of bits in the multiplier.
- **Handling Signed Numbers:** It efficiently manages both positive and negative multipliers by using 2's complement representation.
- **Efficiency with Consecutive Ones:** The algorithm is particularly effective when the multiplier has consecutive ones, converting them into fewer operations.

Multiplication of Binary numbers

Booth's Algorithm

1. Initialization:

- Set up the multiplicand (A) and the multiplier (B).
- Compute the negative of the multiplicand (-A) using 2's complement.

2. Recoding:

- Examine pairs of bits in the multiplier (B) along with an extra bit (initially 0).
- Depending on the pair of bits, decide whether to add, subtract, or do nothing with the multiplicand.

3. Partial Product Generation:

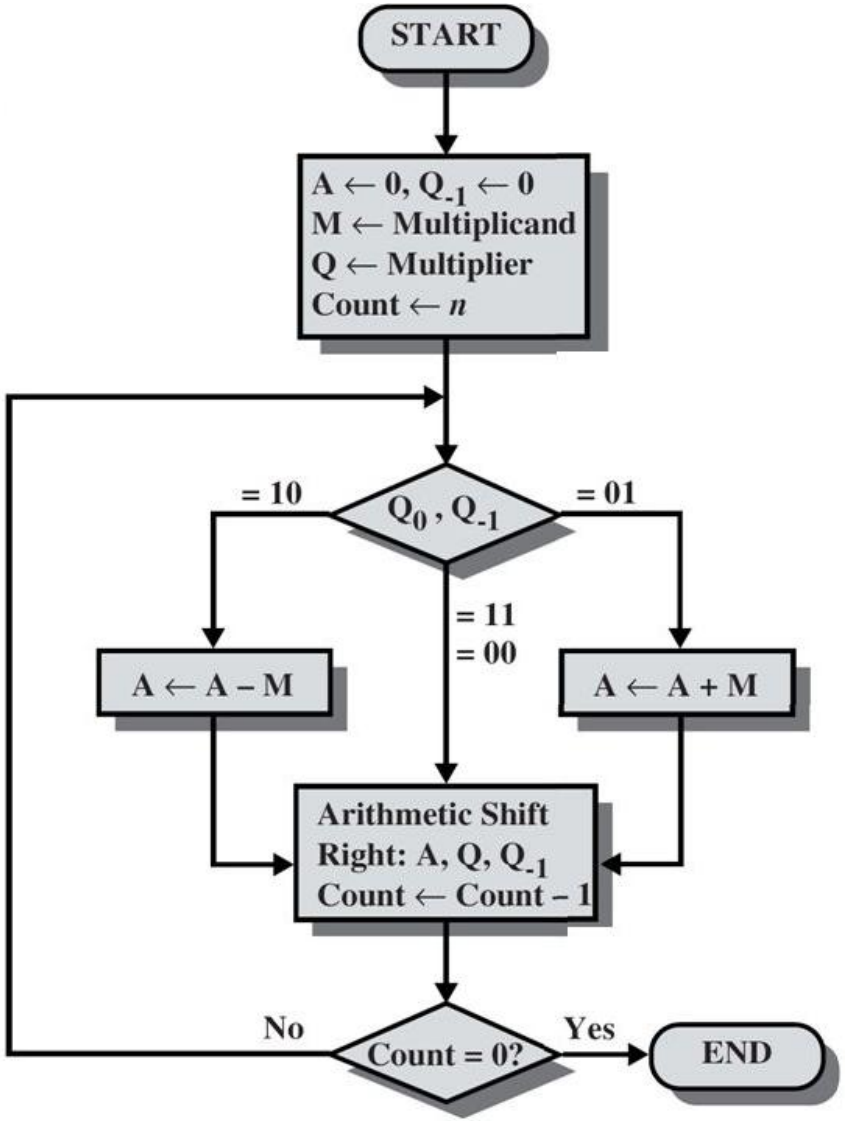
- Generate partial products based on the recoding step.
- Shift the partial products appropriately.

4. Summation:

- Sum the partial products to get the final result.

Multiplication of Binary numbers

Booth's Algorithm



- Starting from right to left, look at *two adjacent bits* of the multiplier
  - Before starting, place a zero at the right of the LSB
- If bits = 00, **do nothing**
- If bits = 10 [read from LSB side], **subtract** the multiplicand from the product
  - Beginning of a string of 1's
- If bits = 11, **do nothing**
  - Middle of a string of 1's
- If bits = 01 [read from LSB side], **add** the multiplicand to the product
  - End of a string of 1's
- apply **arithmetic right shift** by one bit on product register

Multiplication of Binary numbers

Booth's Algorithm

- Example: Multiply the numbers -- M = 0101, Q = 0111

Description	A	Q	q_{-1}
Initialization	0000	0111	0
1: A = A - M (subtract)	1011	0111	0
ASR AQ	1101	1011	1
2: ASR AQ	1110	1101	1
3: ASR AQ	1111	0110	1
4: A = A + M (Addition)	0100	0110	1
ASR AQ	0010	0011	0

5*7 = 23H

Multiplication of Binary numbers

Booth's Algorithm

- **Example:** *Multiply M=-9 and Q=-13*
- **2's complement of -13 = 10011 and 2's complement of -9 = 10111**

A	Q	Q_{-1}	M	Operation
00000	10011	0	10111	initial stage
01001	10011	0	10111	A<-A-M
00100	11001	1	10111	ASR (1st iteration)
00010	01100	1	10111	ASR (2nd iteration)
11001	01100	1	10111	A<-A+M
11100	10110	0	10111	ASR (3rd iteration)
11110	01011	0	10111	ASR (4th iteration)
00111	01011	0	10111	A<-A-M
00011	**10101**	1	10111	ASR (5th iteration)

- **Product : 0001110101 = $(117)_{10}$**

Multiplication of Binary numbers

Booth's Algorithm

- **Example: Multiply -8 and 12**
- **2's complement of -8 = 11000 and 12 = 01100**

A	Q	Q_{-1}	M	operation
00000	01100	0	11000	initial stage
00000	00110	0	11000	ASR (1^{st} iteration)
00000	00011	0	11000	ASR (2^{nd} iteration)
01000	00011	0	11000	A<-A-M
00100	00001	1	11000	ASR (3^{rd} iteration)
00010	00000	1	11000	ASR (4^{th} iteration)
11010	00000	1	11000	A<-A+M
11101	**00000**	0	11000	ASR (5^{th} iteration)

- **Product : 2's complement of $(1110100000)_2 = (-96)_{10}$**

Multiplication of Binary numbers

Drawbacks to Booth's Algorithm

- Variable number of add/subtract operations and of shift operations between two consecutive add/subtract operations
  - Inconvenient when designing a synchronous multiplier
- Algorithm inefficient with isolated 1's
- **Example:**
- $00101010\,1(0)$ recoded as $01\bar{1}1\bar{1}1\bar{1}1\bar{1}$, requiring 8 instead of 4 operations
- Situation can be improved by examining 3 bits at a time rather than 2

Adder Design

- **Carry Look-Ahead Adder (CLA)**