COMPUTER ORGANIZATION AND ARCHITECTURE

Execution Unit

Dr. Bore Gowda S B

Additional Professor

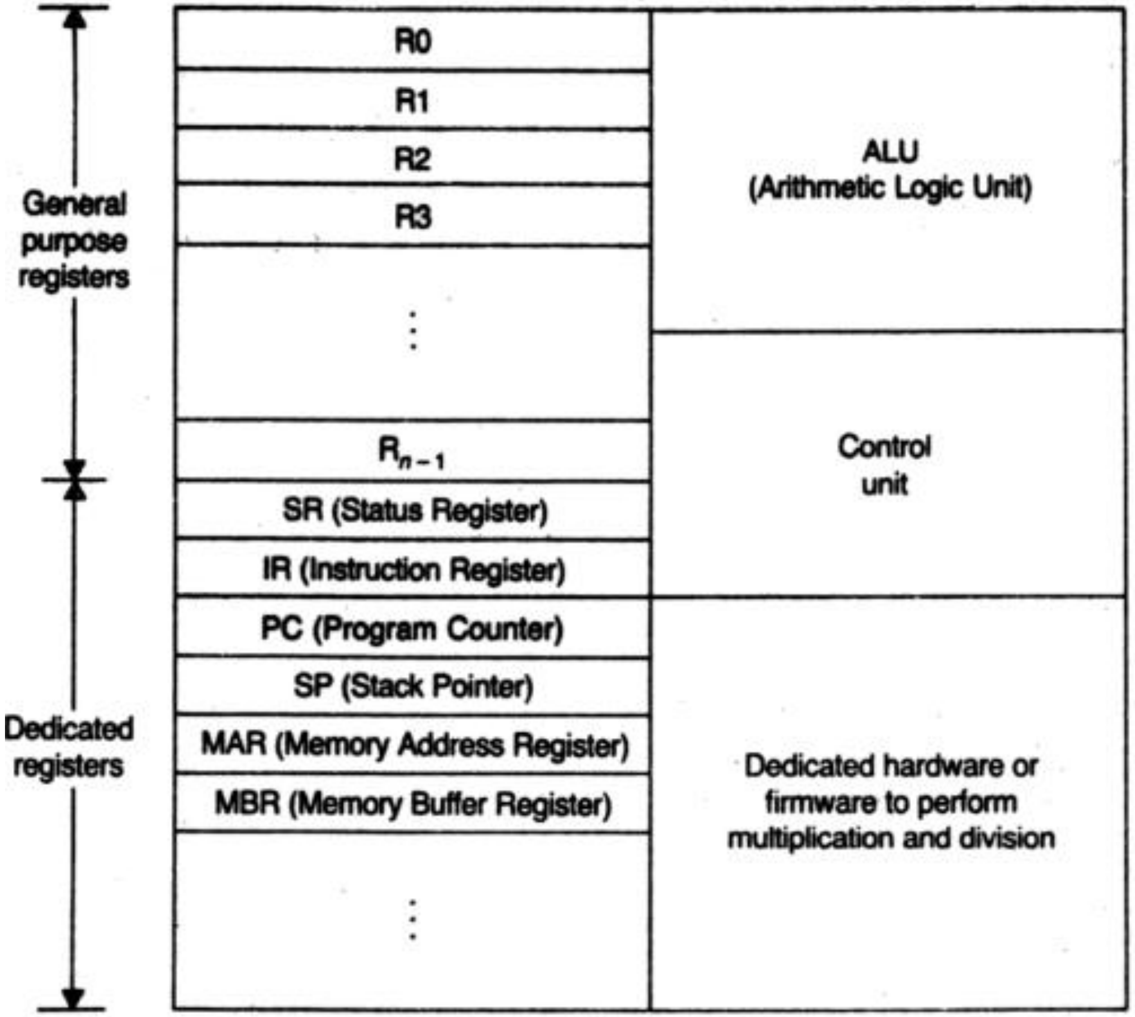
Dept. of ECE

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

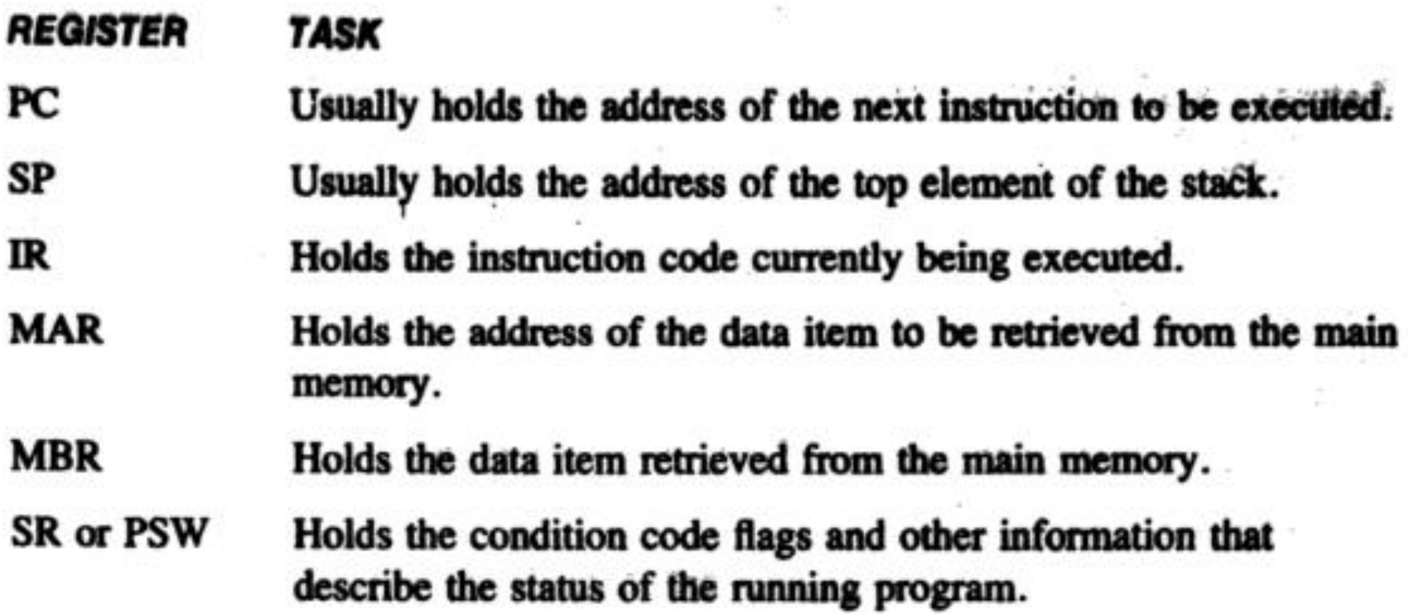


General purpose registers
Dedicated registers
R0
R1
R2
R3
R_{n-1}
SR (Status Register)
IR (Instruction Register)
PC (Program Counter)
SP (Stack Pointer)
MAR (Memory Address Register)
MBR (Memory Buffer Register)
ALU (Arithmetic Logic Unit)
Control unit
Dedicated hardware or firmware to perform multiplication and division

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:

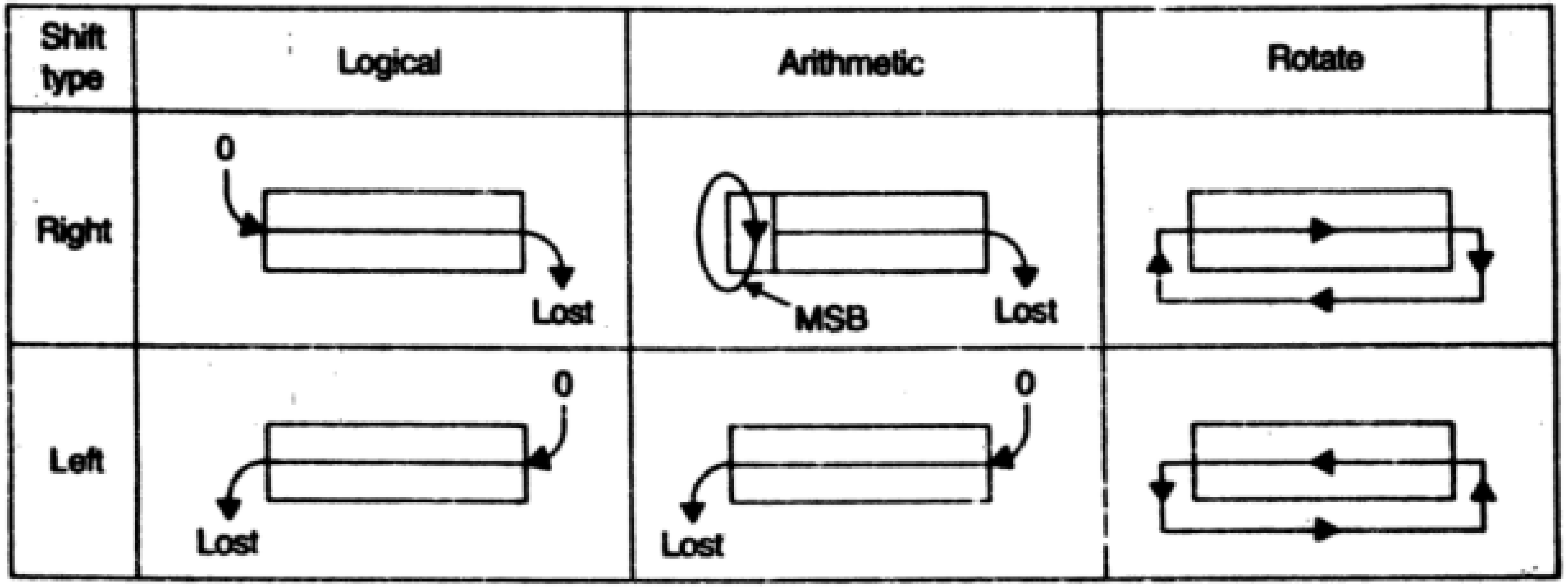


REGISTER	*TASK*
PC	Usually holds the address of the next instruction to be executed.
SP	Usually holds the address of the top element of the stack.
IR	Holds the instruction code currently being executed.
MAR	Holds the address of the data item to be retrieved from the main memory.
MBR	Holds the data item 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:

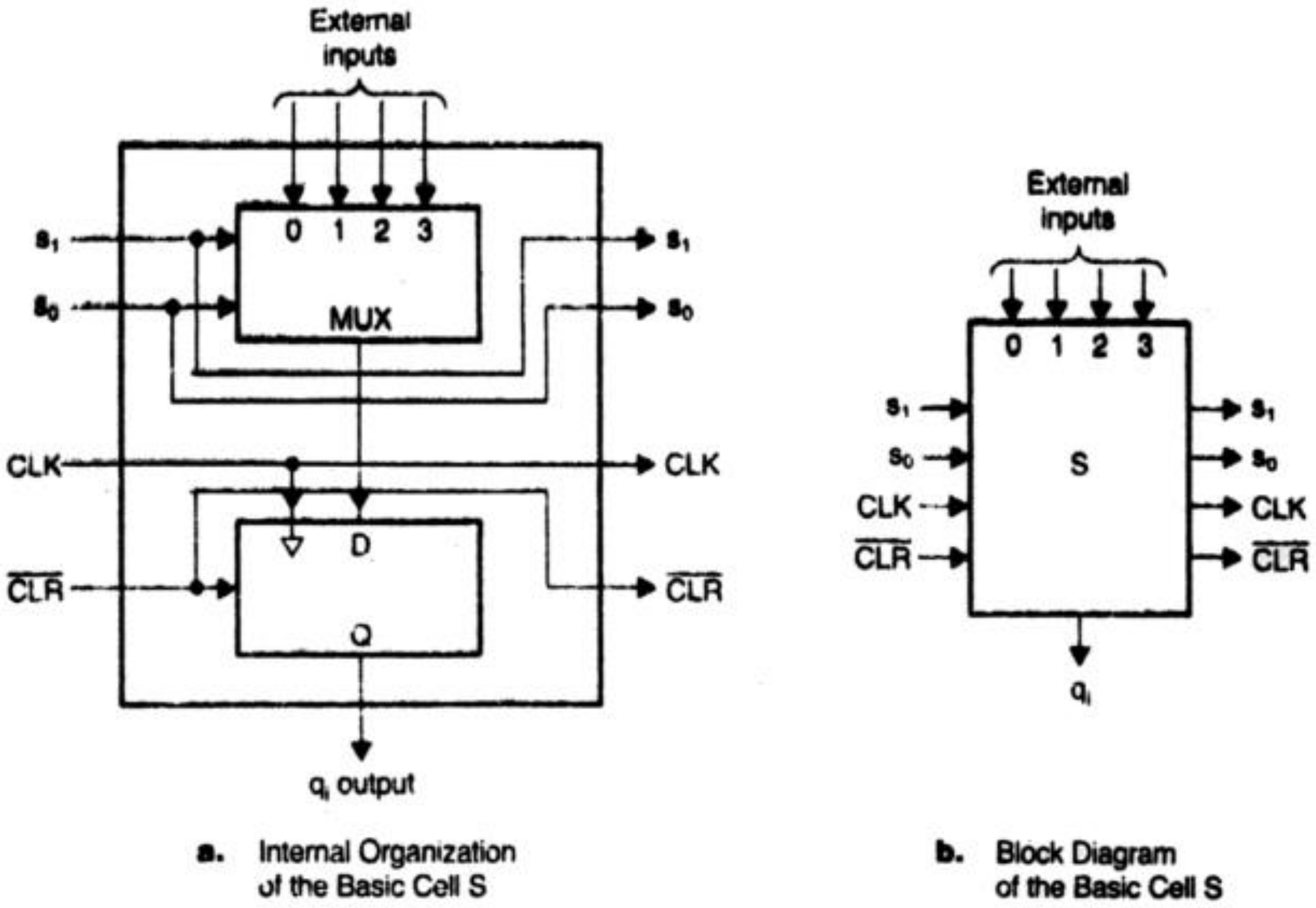
- GPR is to store address or data, then to be able to retrieve the data when needed
- GPR is also capable of manipulating the stored data by shift left or shift right operation
- Logical shift operations
- Arithmetic shift operations



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:



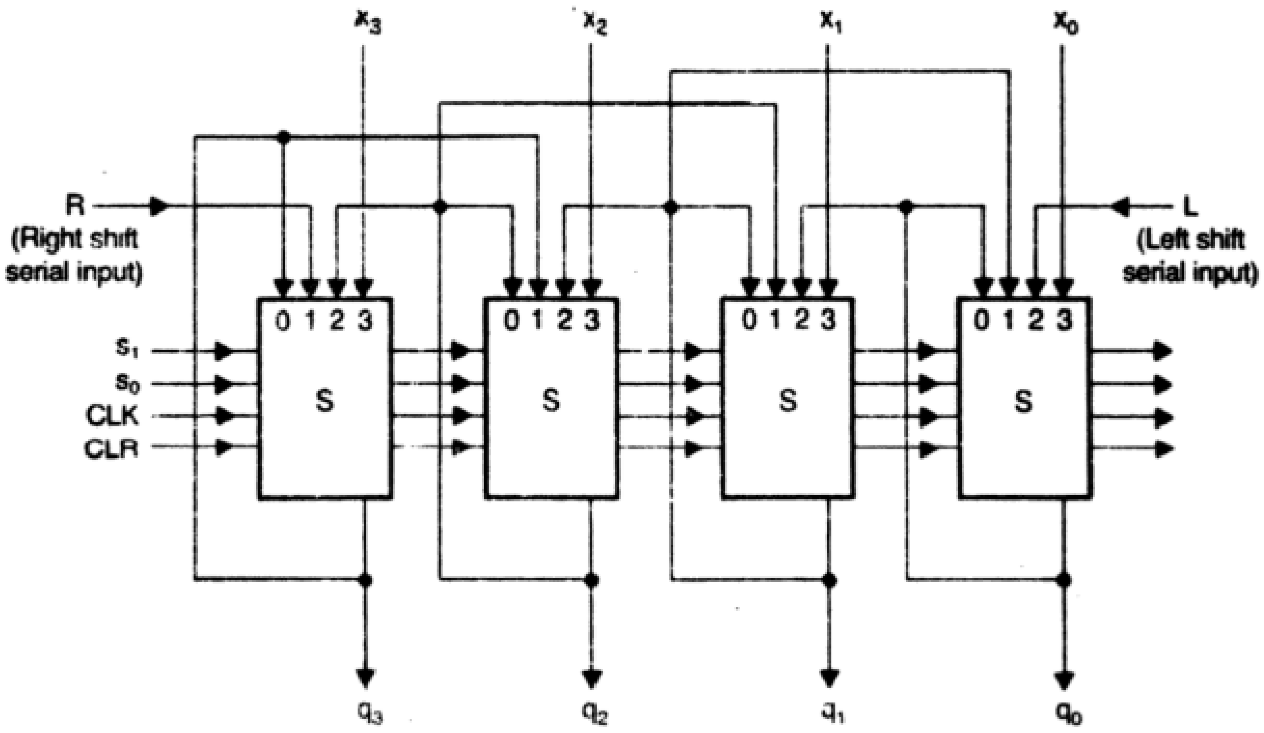
a. Internal Organization of the Basic Cell S

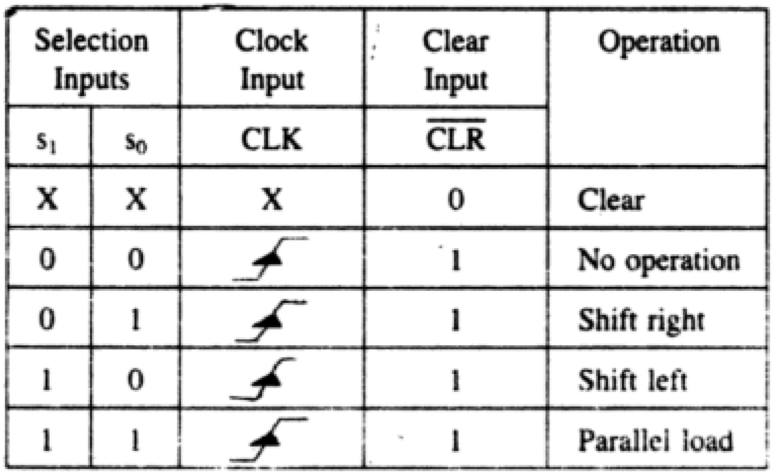
b. Block Diagram of the Basic Cell S

General Register Design

❑ 4-bit GPR:

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

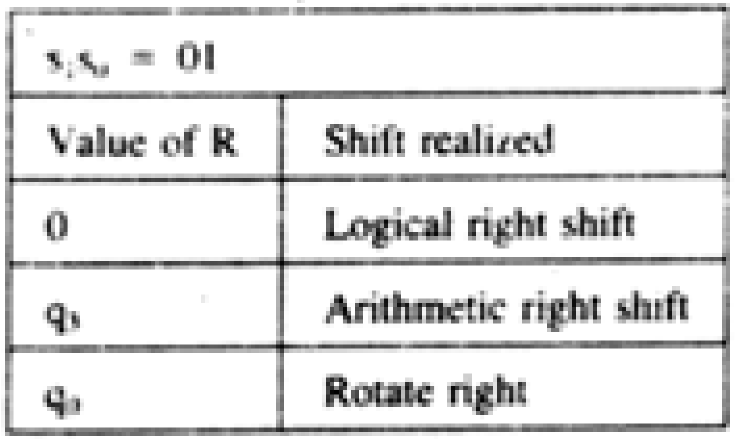




Selection Inputs		Clock Input	Clear Input	Operation
s_1	s_0	CLK	$\overline{\text{CLR}}$	
X	X	X	0	Clear
0	0	↑	1	No operation
0	1	↑	1	Shift right
1	0	↑	1	Shift left
1	1	↑	1	Parallel load

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

Three Variations of the Right Shift Operations

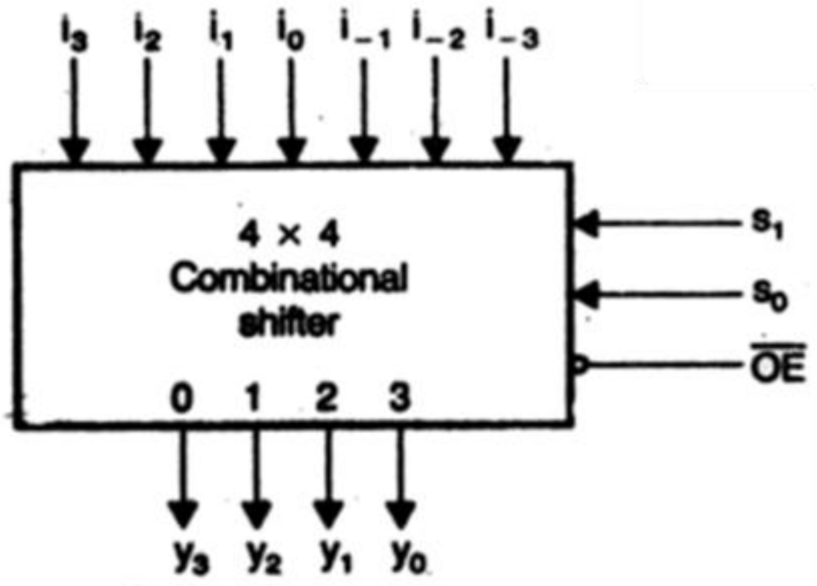


s_1s_0 = 01	
Value of R	Shift realized
0	Logical right shift
q_3	Arithmetic right shift
q_0	Rotate right

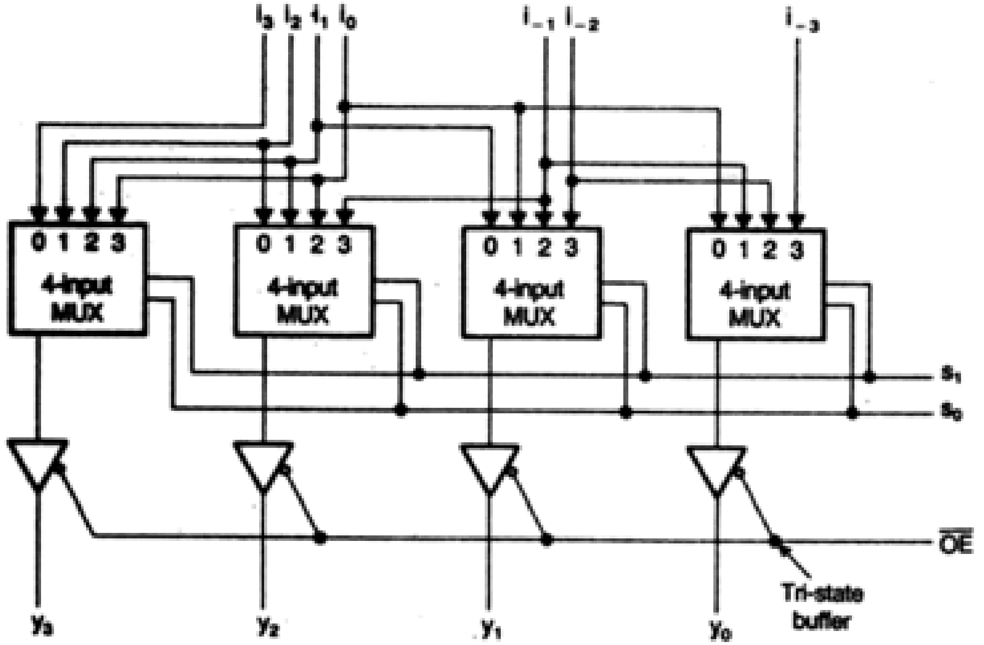
General Register Design

❑ 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



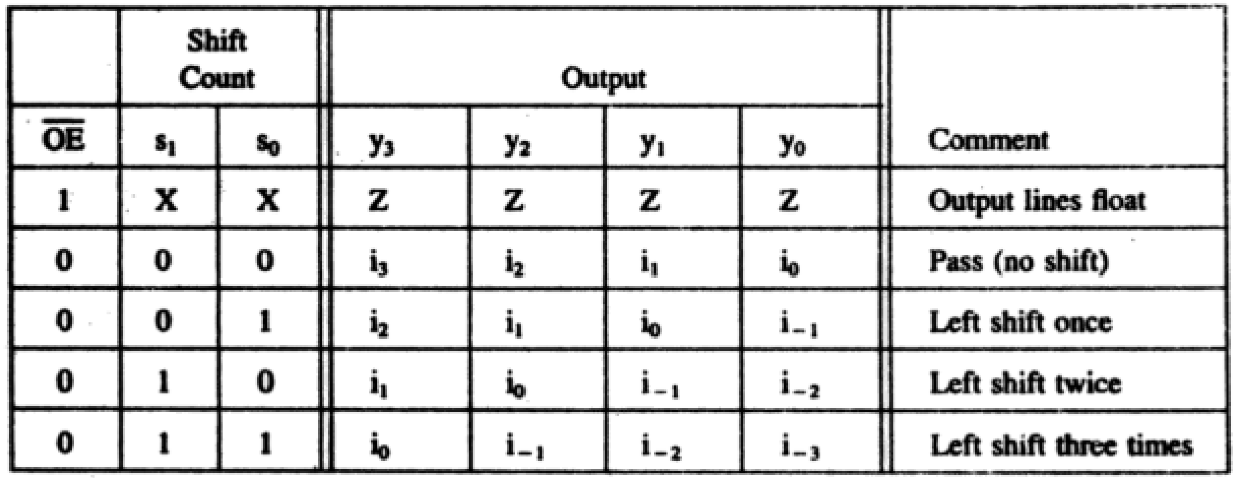
a. Block Diagram



b. Internal Schematic

General Register Design

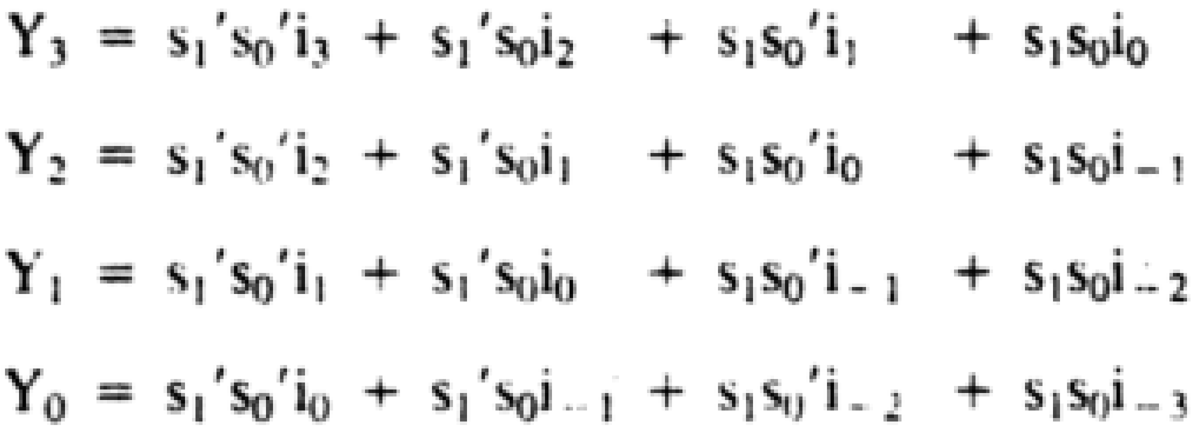
❑ 4x4 Combinational shifter design:



	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



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

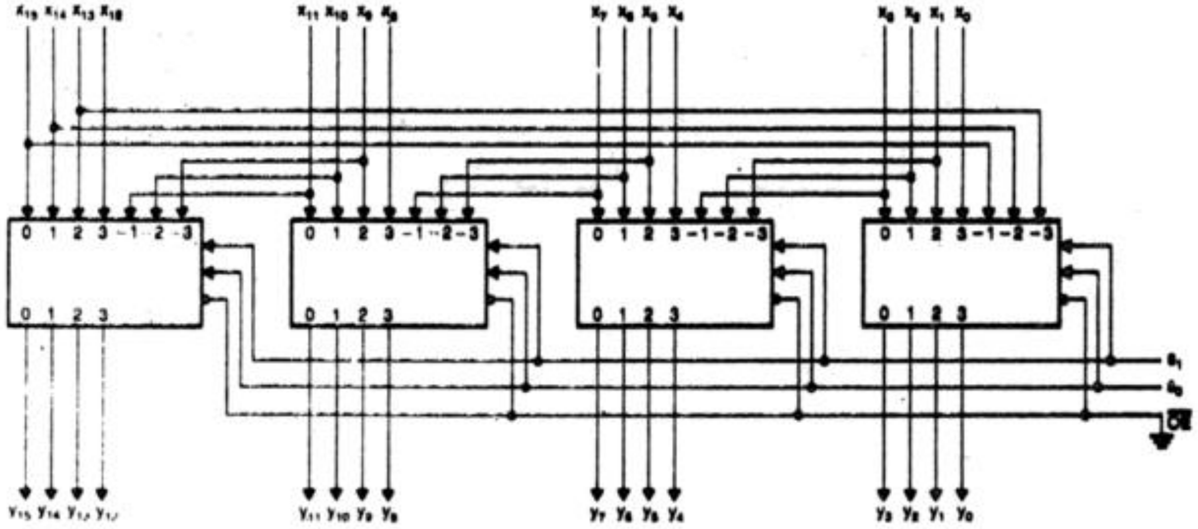
$$Y_2 = s_1's_0'i_2 + s_1's_0i_1 + s_1s_0'i_0 + s_1s_0i_{-1}$$

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

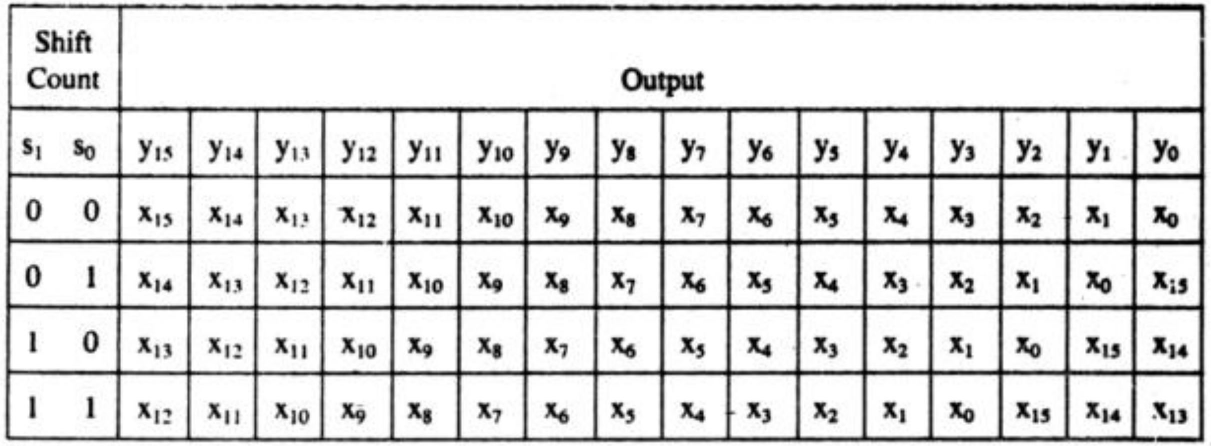
$$Y_0 = s_1's_0'i_0 + s_1's_0i_{-1} + s_1s_0'i_{-2} + s_1s_0i_{-3}$$

General Register Design

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



a. Logic Diagram



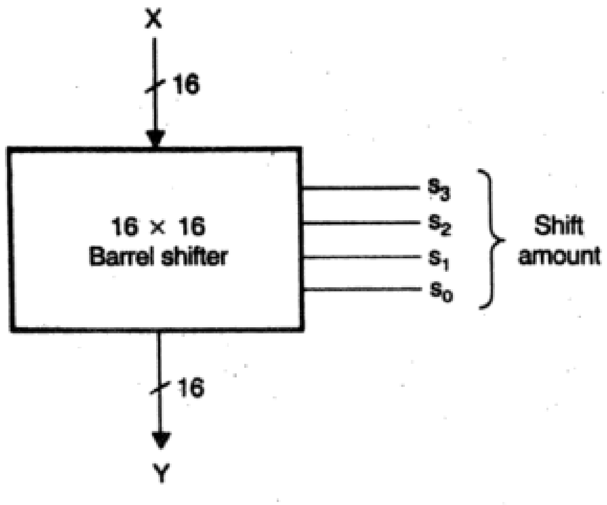
Shift Count		Output															
s_1	s_0	y_{15}	y_{14}	y_{13}	y_{12}	y_{11}	y_{10}	y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	0	x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
0	1	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{15}
1	0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

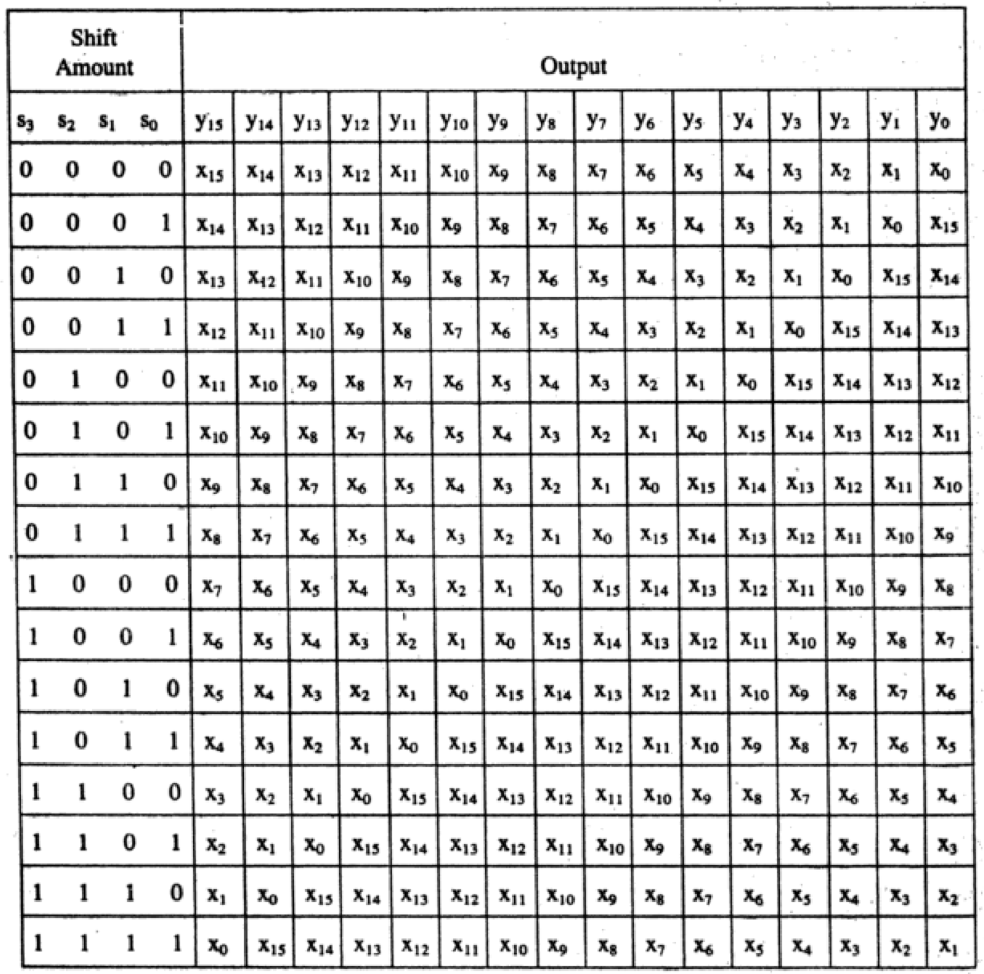
General Register Design

❑ 16x16 Barrel shifter:

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

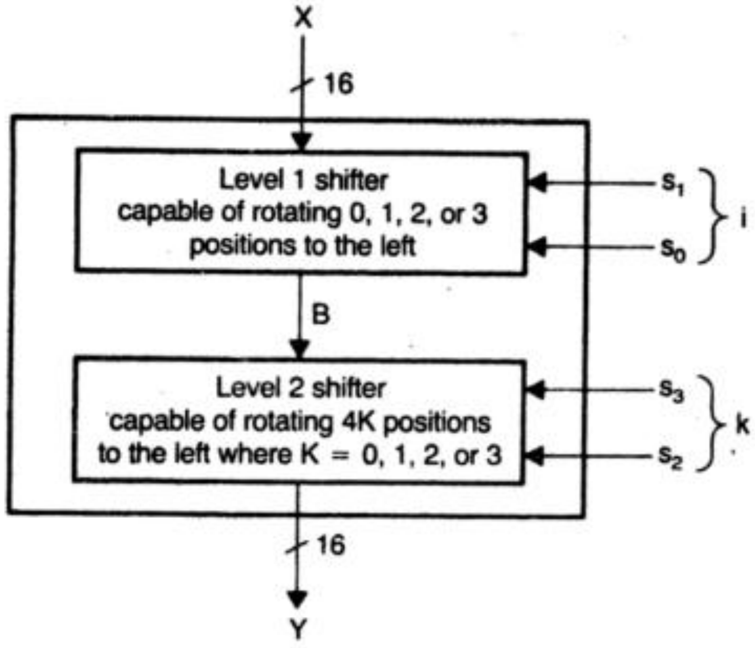


a. Block Diagram of a 16 × 16 Block Shifter



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

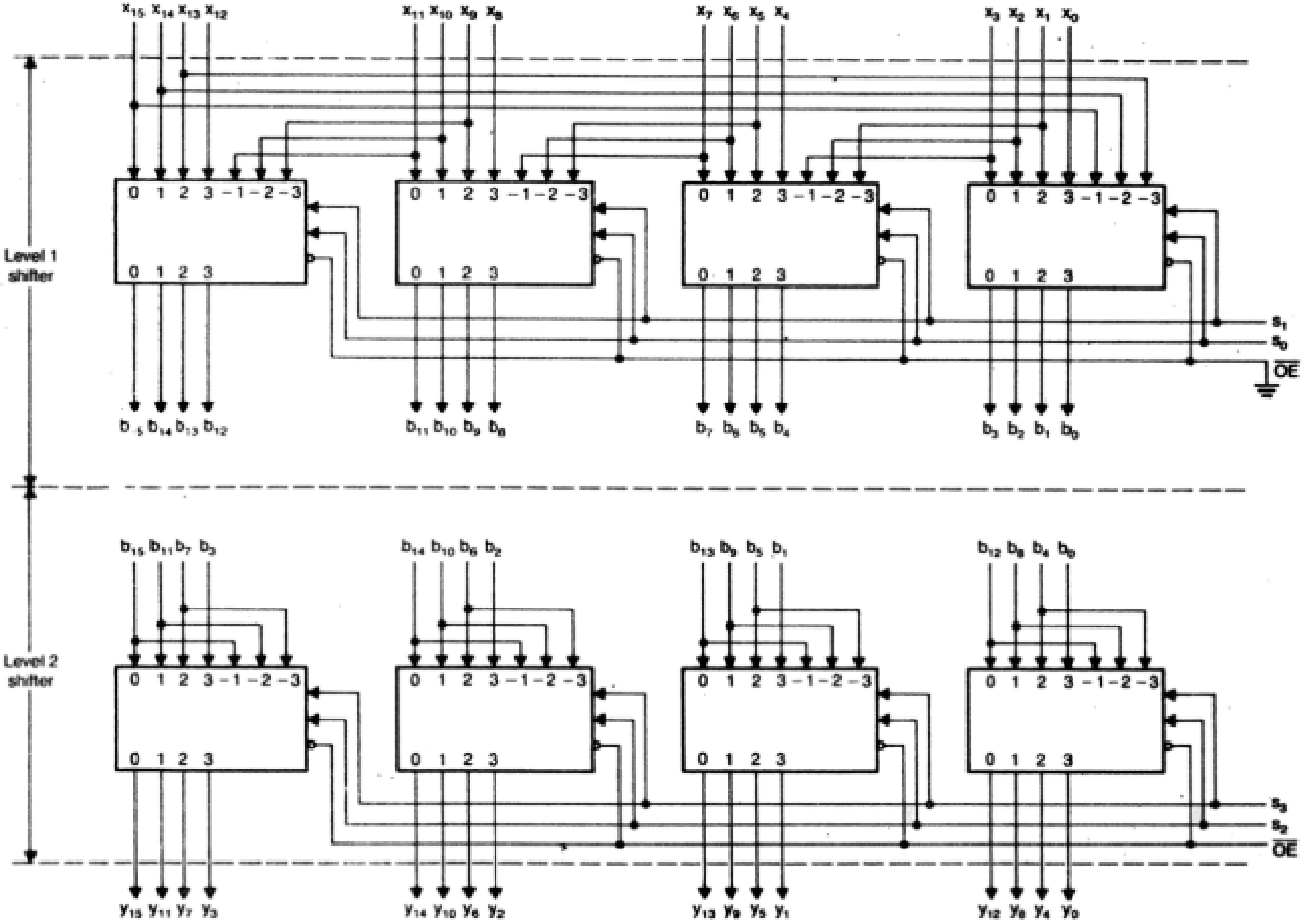
b. Truth Table of the 16 × 16 Barrel Shifter


X
16
Level 1 shifter
capable of rotating 0, 1, 2, or 3
positions to the left
s_1
s_0
i
B
Level 2 shifter
capable of rotating 4K positions
to the left where K = 0, 1, 2, or 3
s_3
s_2
k
16
Y

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

General Register Design

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

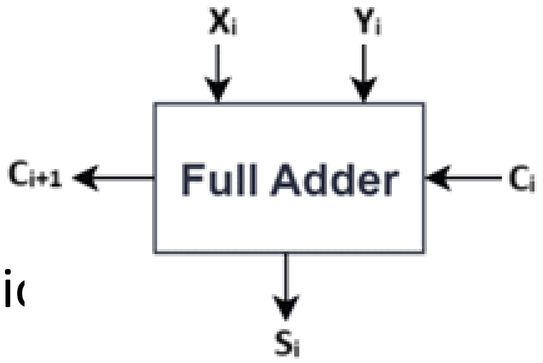


x_{15} x_{14} x_{13} x_{12}
x_{11} x_{10} x_9 x_8
x_7 x_6 x_5 x_4
x_3 x_2 x_1 x_0
0 1 2 3 −1 −2 −3
0 1 2 3
Level 1 shifter
s_1
s_0
$\overline{OE}$
b_{15} b_{14} b_{13} b_{12}
b_{11} b_{10} b_9 b_8
b_7 b_6 b_5 b_4
b_3 b_2 b_1 b_0
b_{15} b_{11} b_7 b_3
b_{14} b_{10} b_6 b_2
b_{13} b_9 b_5 b_1
b_{12} b_8 b_4 b_0
Level 2 shifter
s_3
s_2
$\overline{OE}$
y_{15} y_{11} y_7 y_3
y_{14} y_{10} y_6 y_2
y_{13} y_9 y_5 y_1
y_{12} y_8 y_4 y_0

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



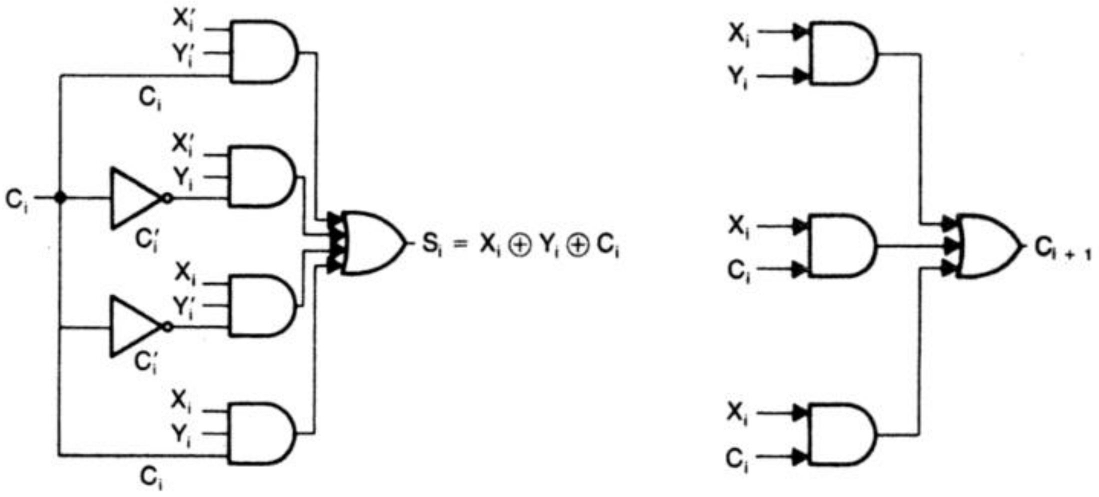
X_i	Y_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

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

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

Adder Design

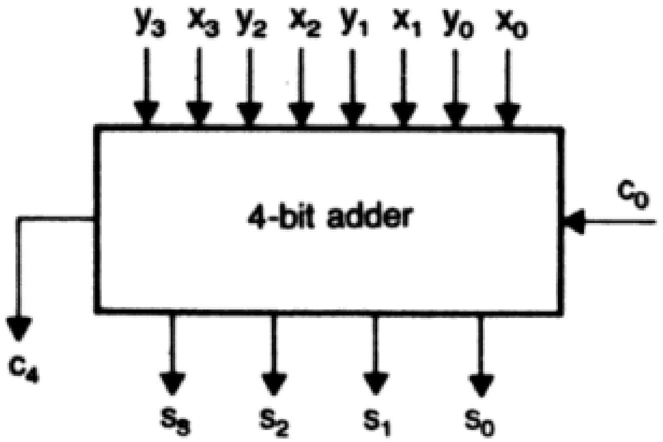
❑ Full adder:



- 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)

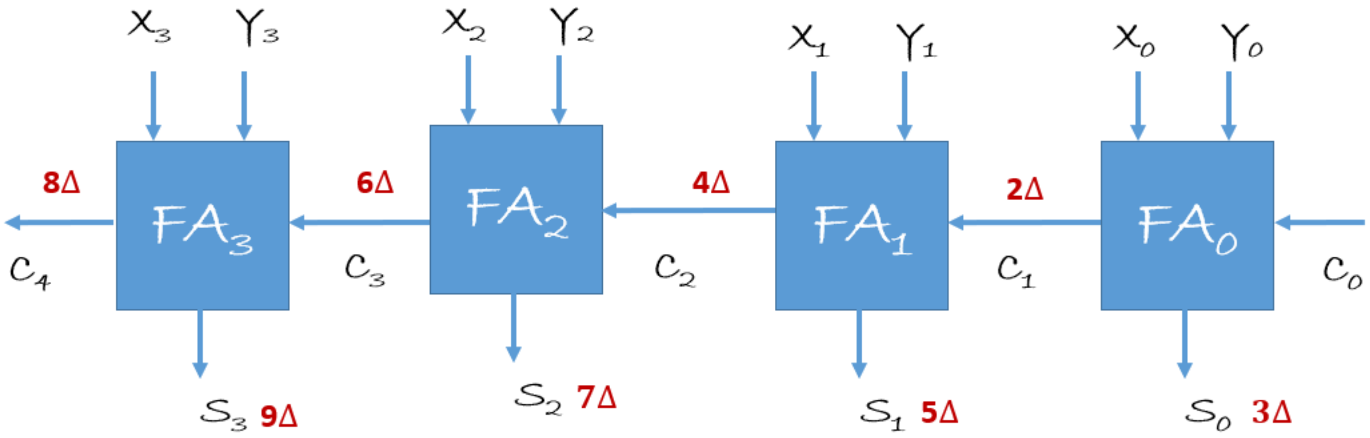


Total delay = (n-1)* delay between each block + last block delay

Wnere n is number of Full Adders

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

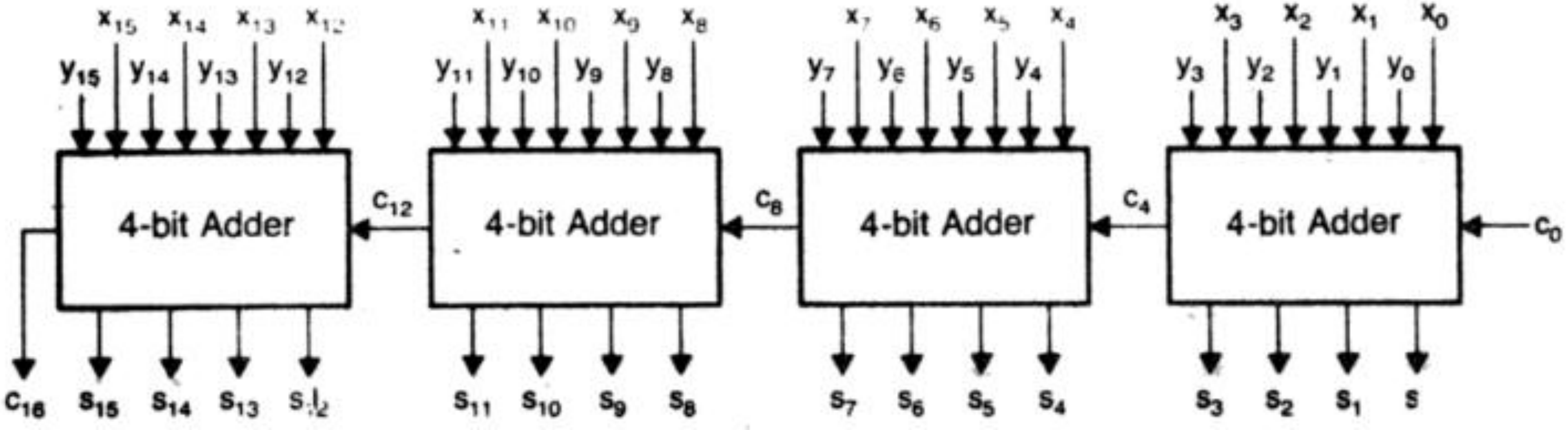
Delay = (4 – 1) * 1 + 1.5 = 4.5 ns



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:

$$
\begin{aligned}
C_1 &= G_0 + P_0C_0\\
C_2 &= G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0\\
C_3 &= G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0\\
C_4 &= G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0
\end{aligned}
$$

- 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:

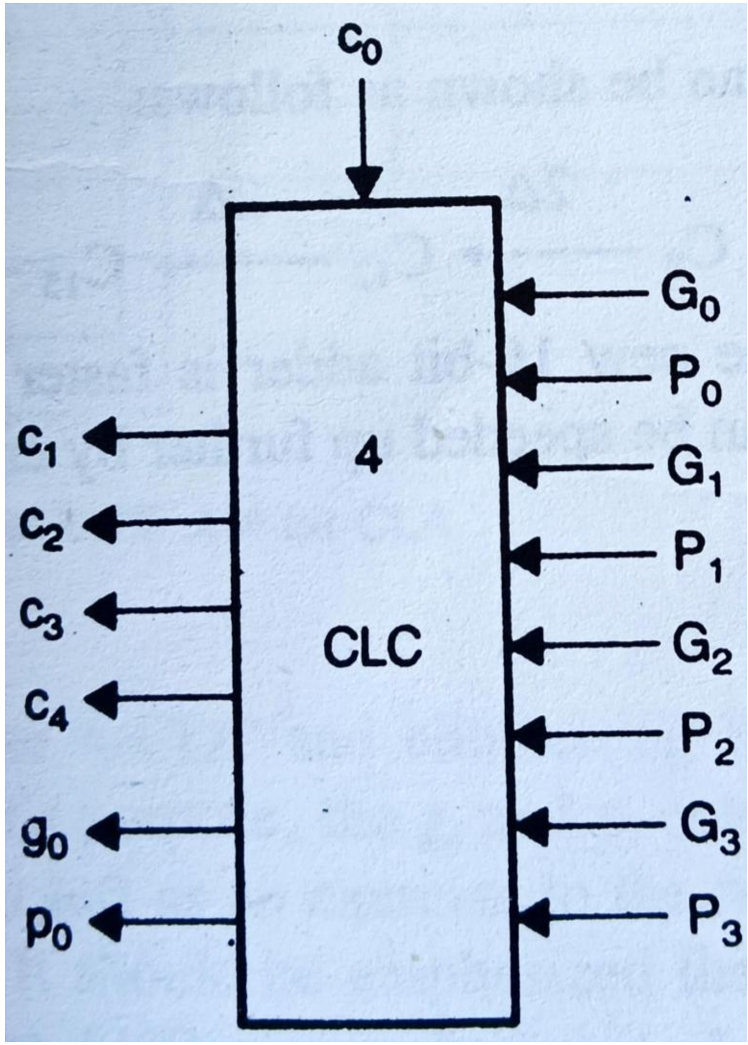
$$
\begin{aligned}
C_1 &= G_0 + P_0C_0 \\
C_2 &= G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0 \\
C_3 &= G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \\
C_4 &= G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0
\end{aligned}
$$

- 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

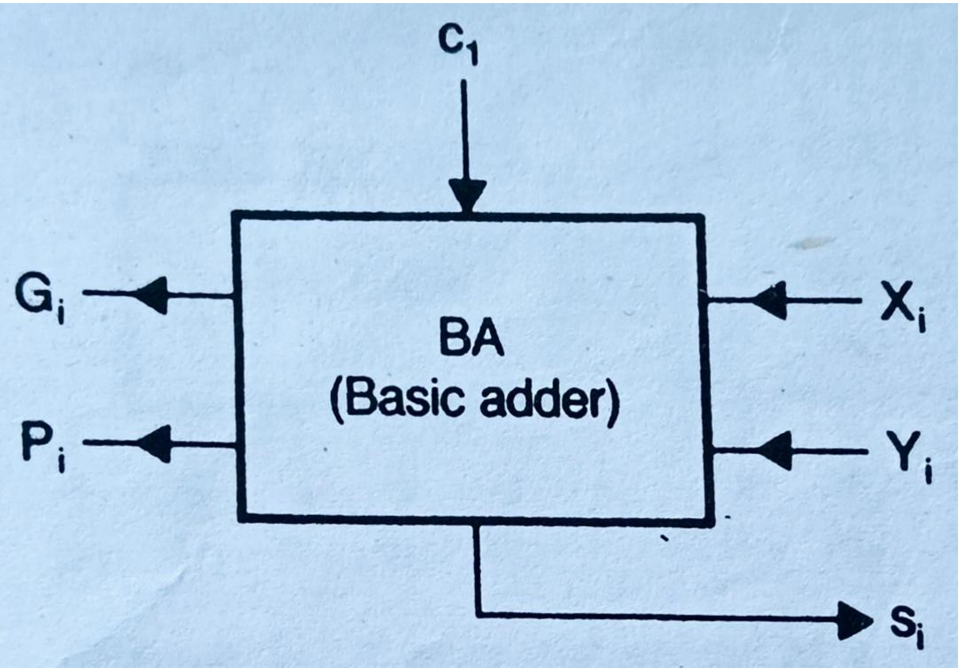
❑ 4-stage Carry Look-Ahead circuit(4-CLC)

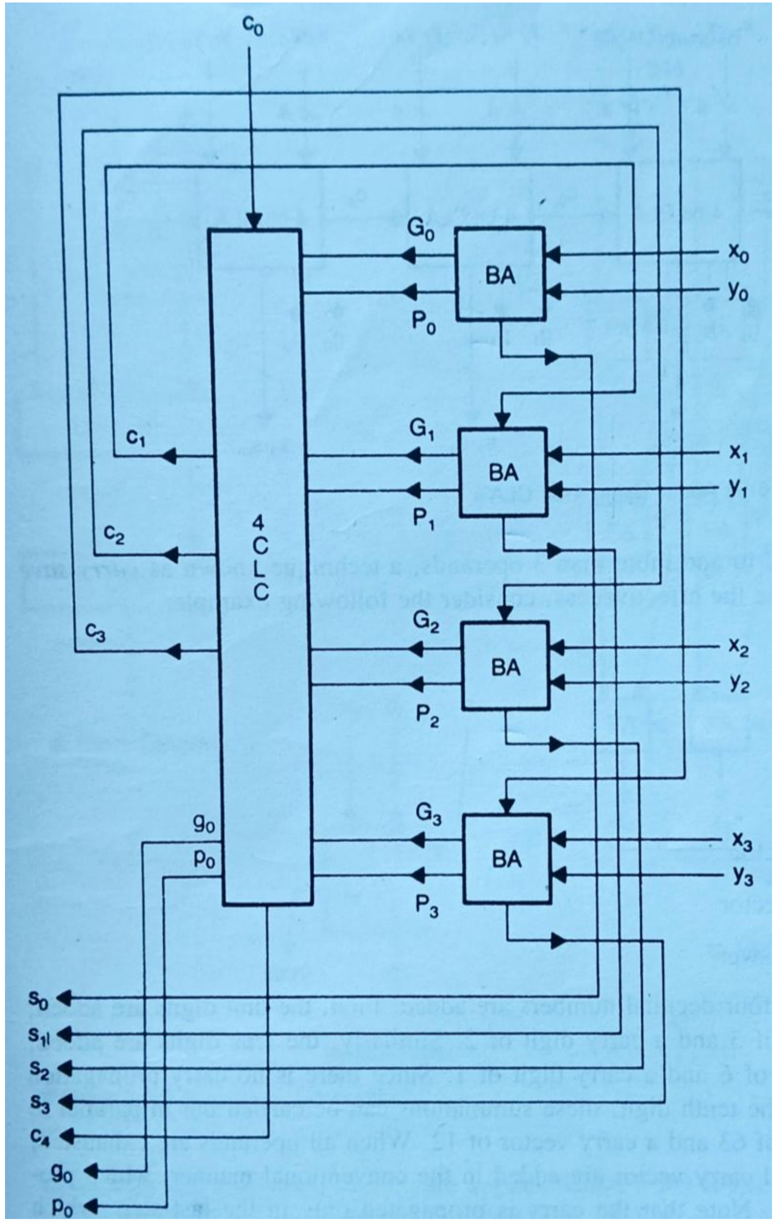
- A 4-CLC can be implemented as a teo-level AND/OR logic circuit
- To generate C4, 5-input **AND** gate and 4-input **OR** gate required
- If $G_i \leq (0 \leq i \leq 3)$ is available, then all C_i's with $(1 \leq i \leq 4)$ can be generated in 2 gate delays
- The output $\boldsymbol{G_0}$ and $\boldsymbol{P_0}$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



General Register Design

- 4-bit CLA using basic adder

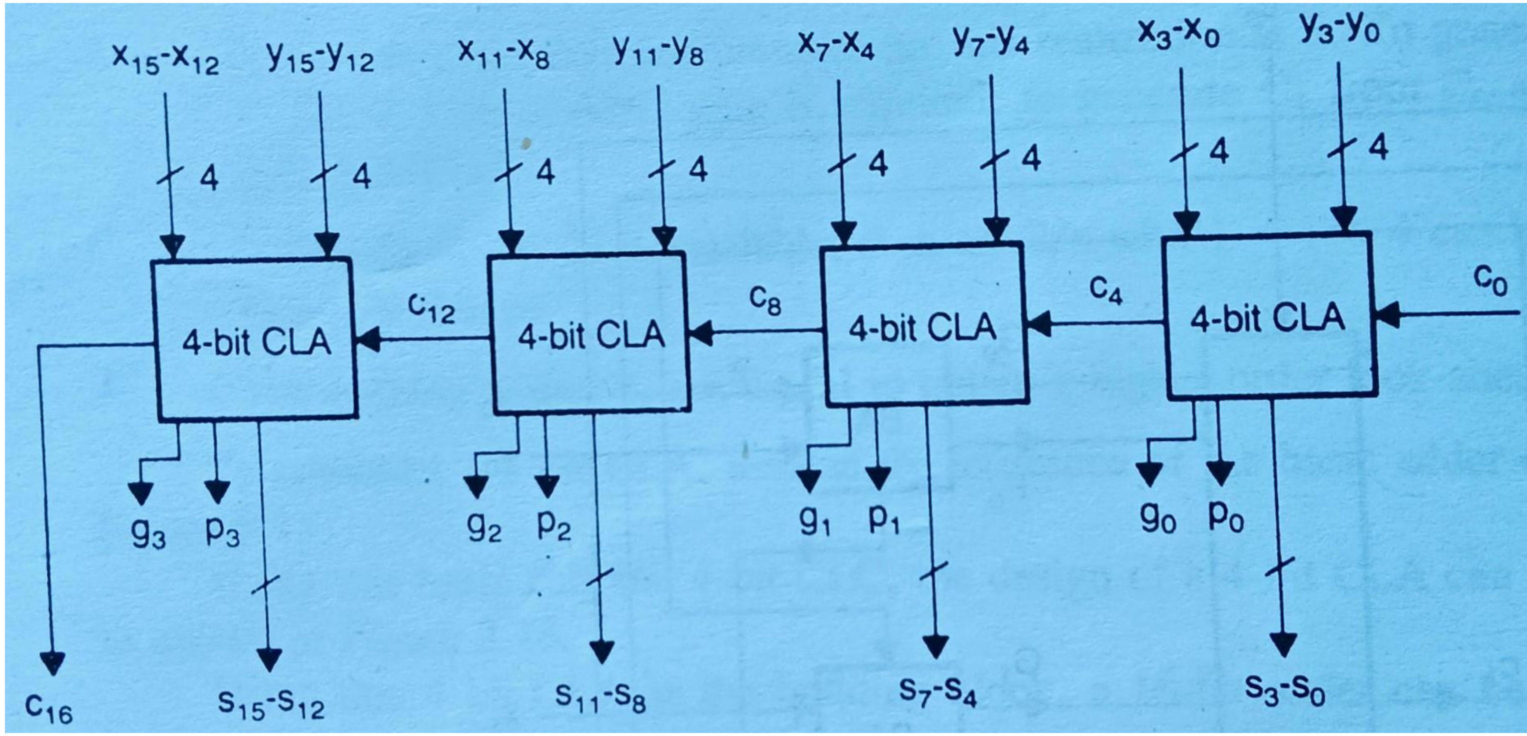


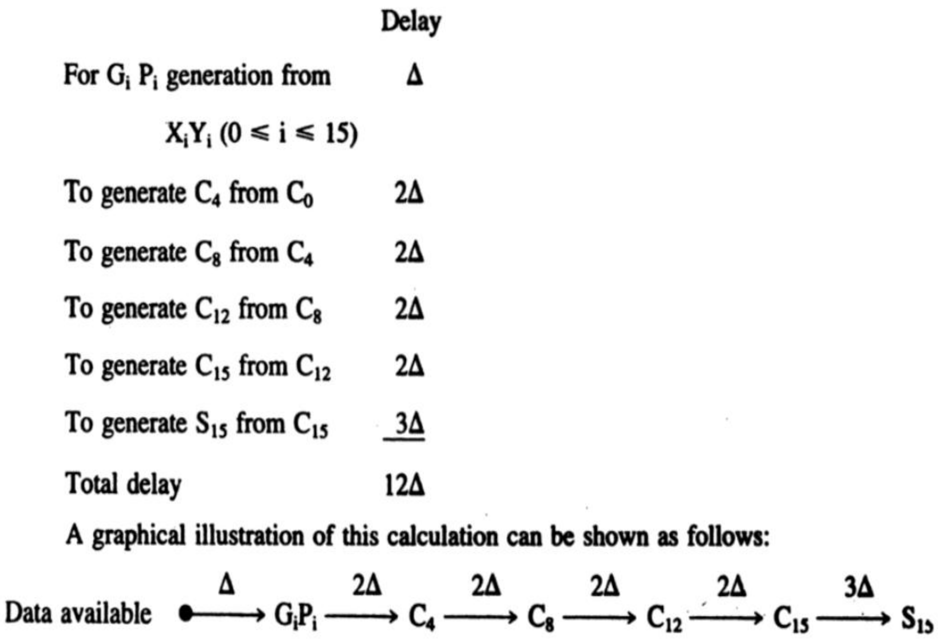


General Register Design

❑ 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.





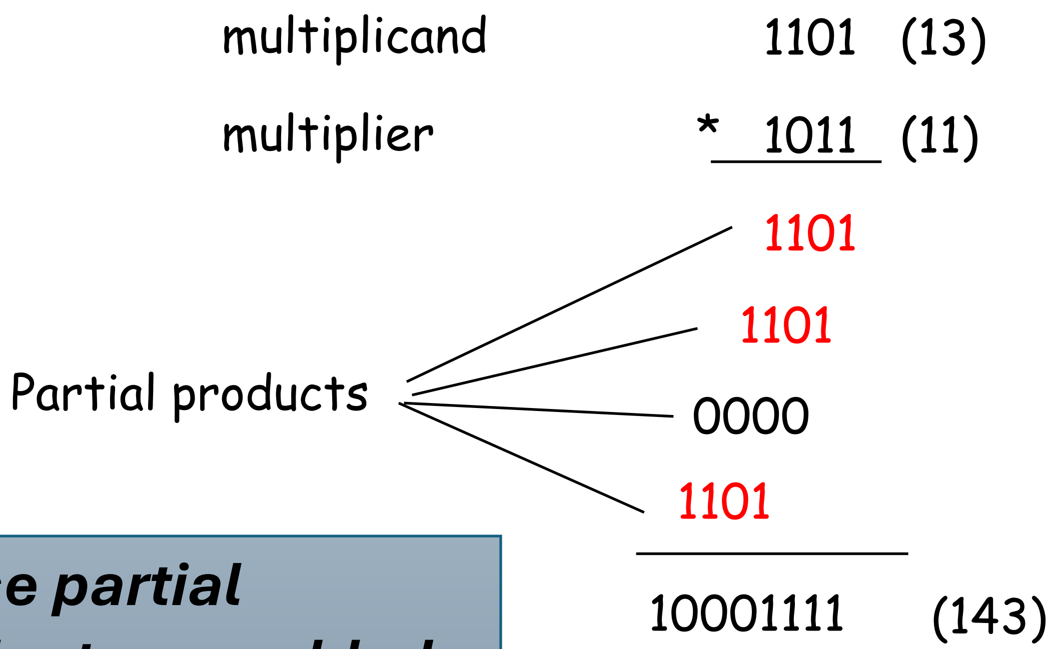
	Delay
For G_i P_i generation from X_iY_i $(0 \leq i \leq 15)$	Δ
To generate C_4 from C_0	2Δ
To generate C_8 from C_4	2Δ
To generate C_{12} from C_8	2Δ
To generate C_{15} from C_{12}	2Δ
To generate S_{15} from C_{15}	3Δ
Total delay	12Δ

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

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

Multiplication of Binary numbers

Basic Concept



multiplicand 1101 (13)

multiplier * 1011 (11)

Partial products

1101

1101

0000

1101

10001111 (143)

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

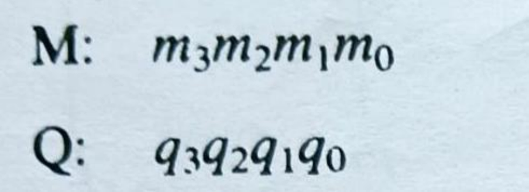
0 × multiplicand = 0

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).

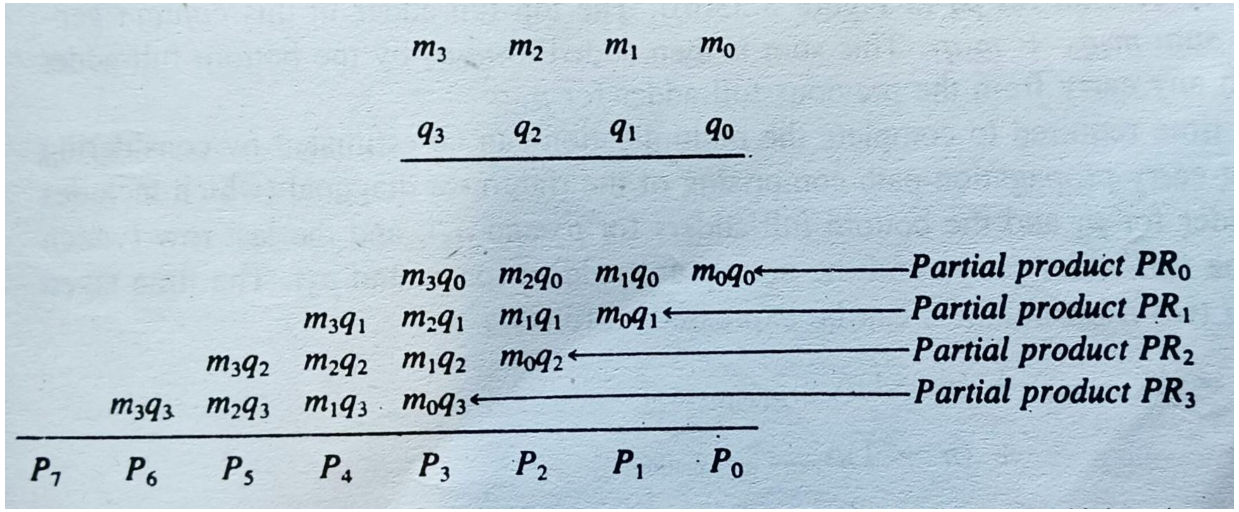
Multiplication of Binary numbers

❑ 4x4 array multiplication

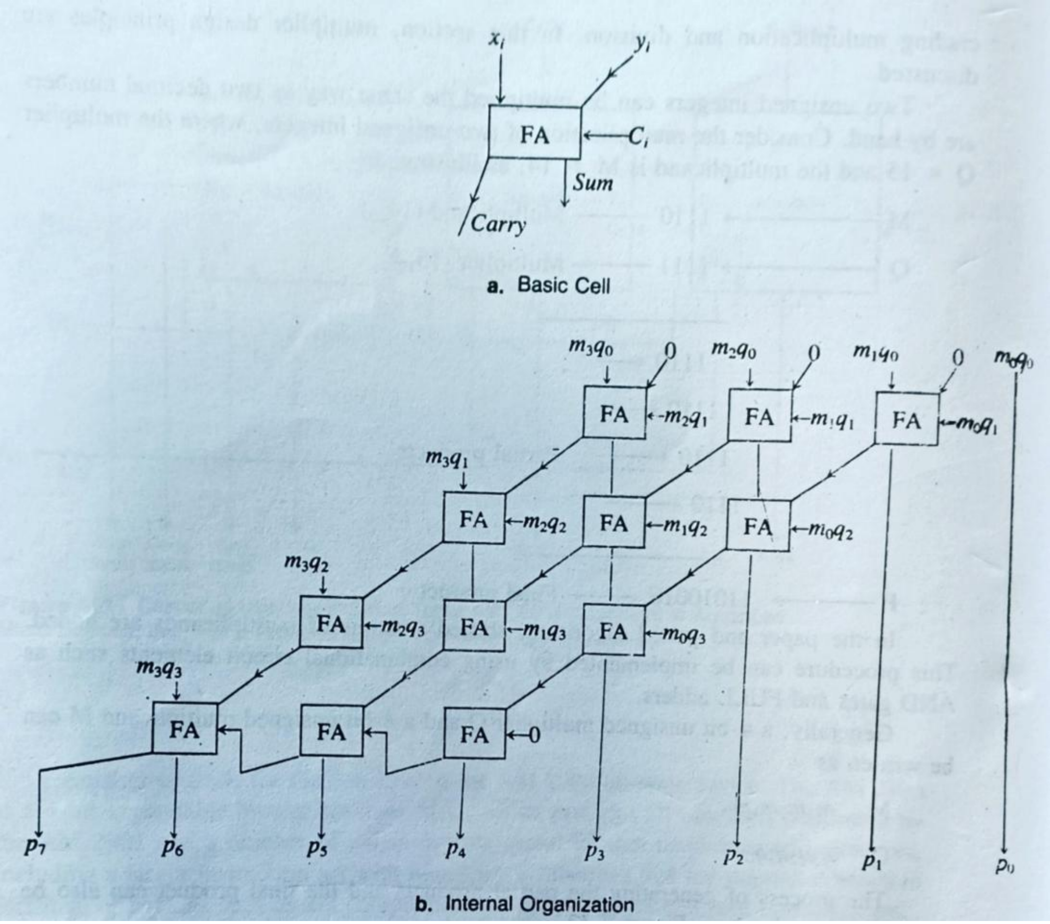


M: $m_3m_2m_1m_0$

Q: $q_3q_2q_1q_0$



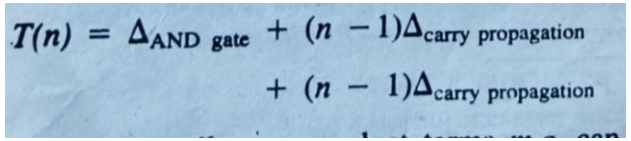
$$\begin{array}{cccccccc}
 & & & & m_3 & m_2 & m_1 & m_0 \\
 & & & & q_3 & q_2 & q_1 & q_0 \\
\hline
 & & & & m_3q_0 & m_2q_0 & m_1q_0 & m_0q_0 \leftarrow \text{Partial product } PR_0 \\
 & & & m_3q_1 & m_2q_1 & m_1q_1 & m_0q_1 & \leftarrow \text{Partial product } PR_1 \\
 & & m_3q_2 & m_2q_2 & m_1q_2 & m_0q_2 & & \leftarrow \text{Partial product } PR_2 \\
 & m_3q_3 & m_2q_3 & m_1q_3 & m_0q_3 & & & \leftarrow \text{Partial product } PR_3 \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}$$



a. Basic Cell

b. Internal Organization

- 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

$$P = M*Q + R$$

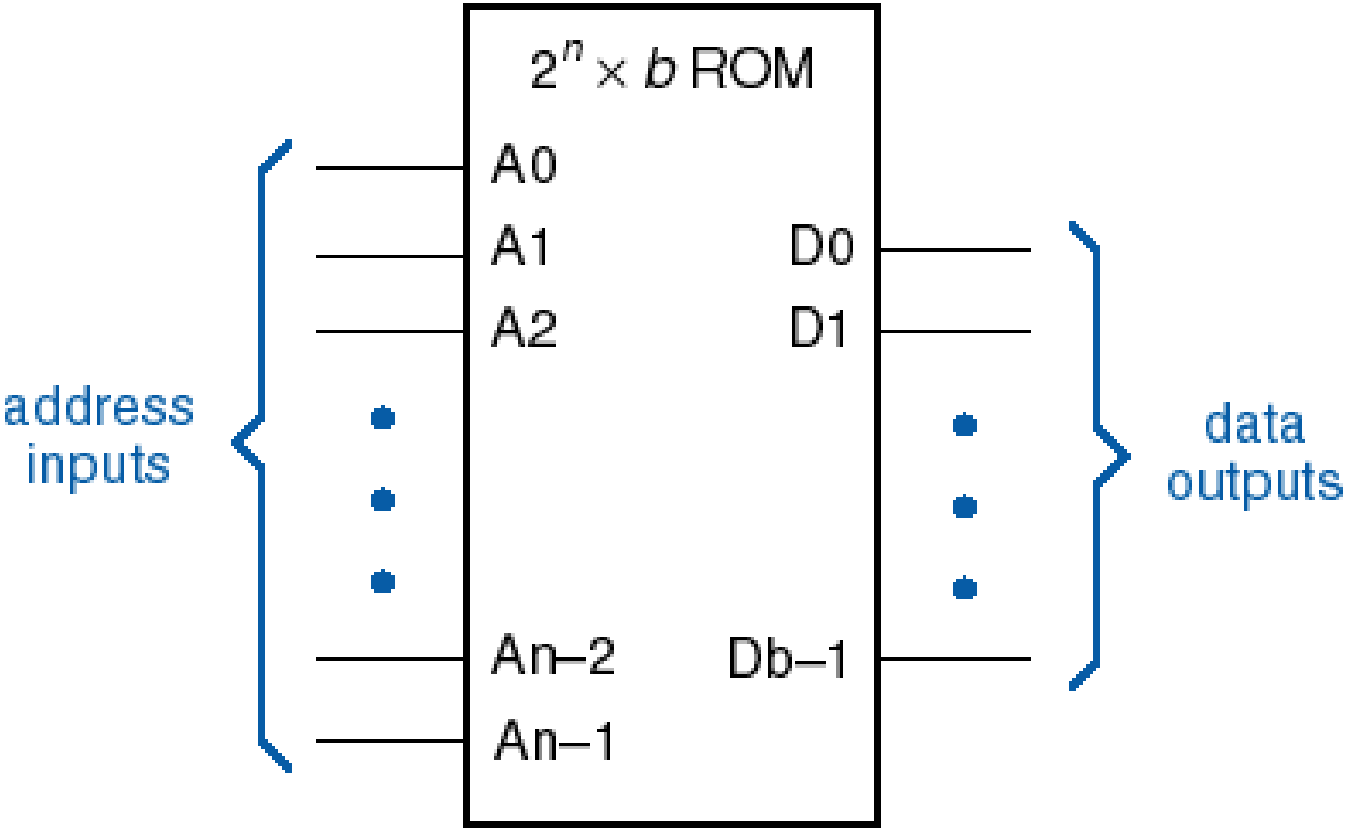
- Both NM and an AM are available as standard IC blocks

- 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

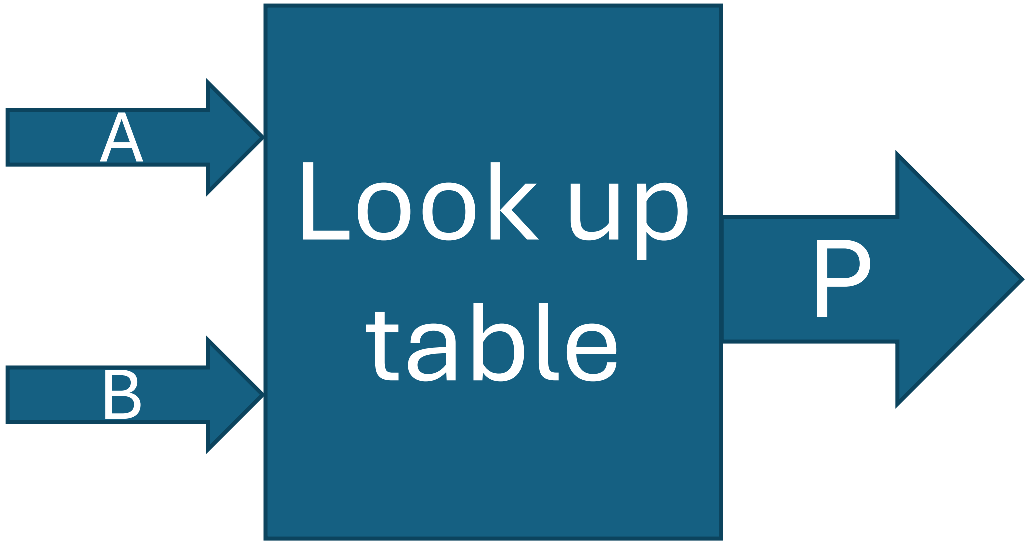


- Can perform any combinational logic function
  - Address inputs = function inputs
  - Data outputs = function outputs

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?

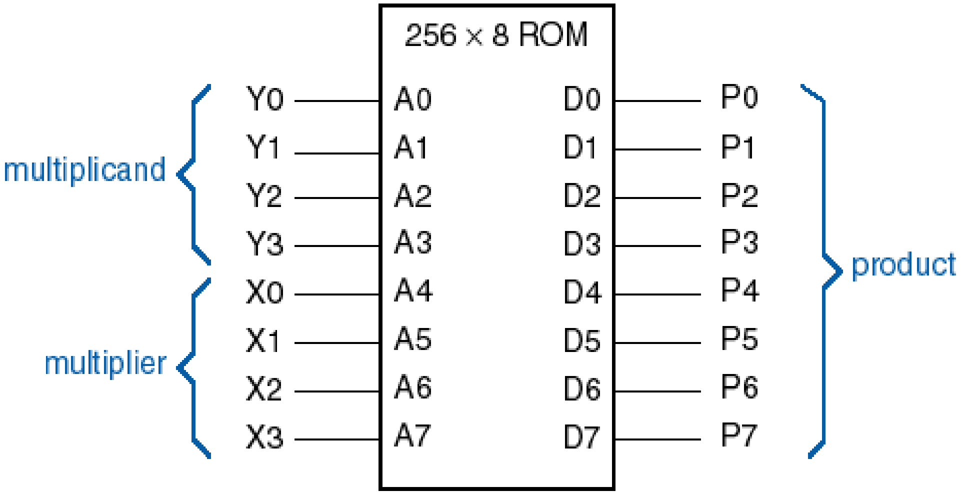
8 bytes!!

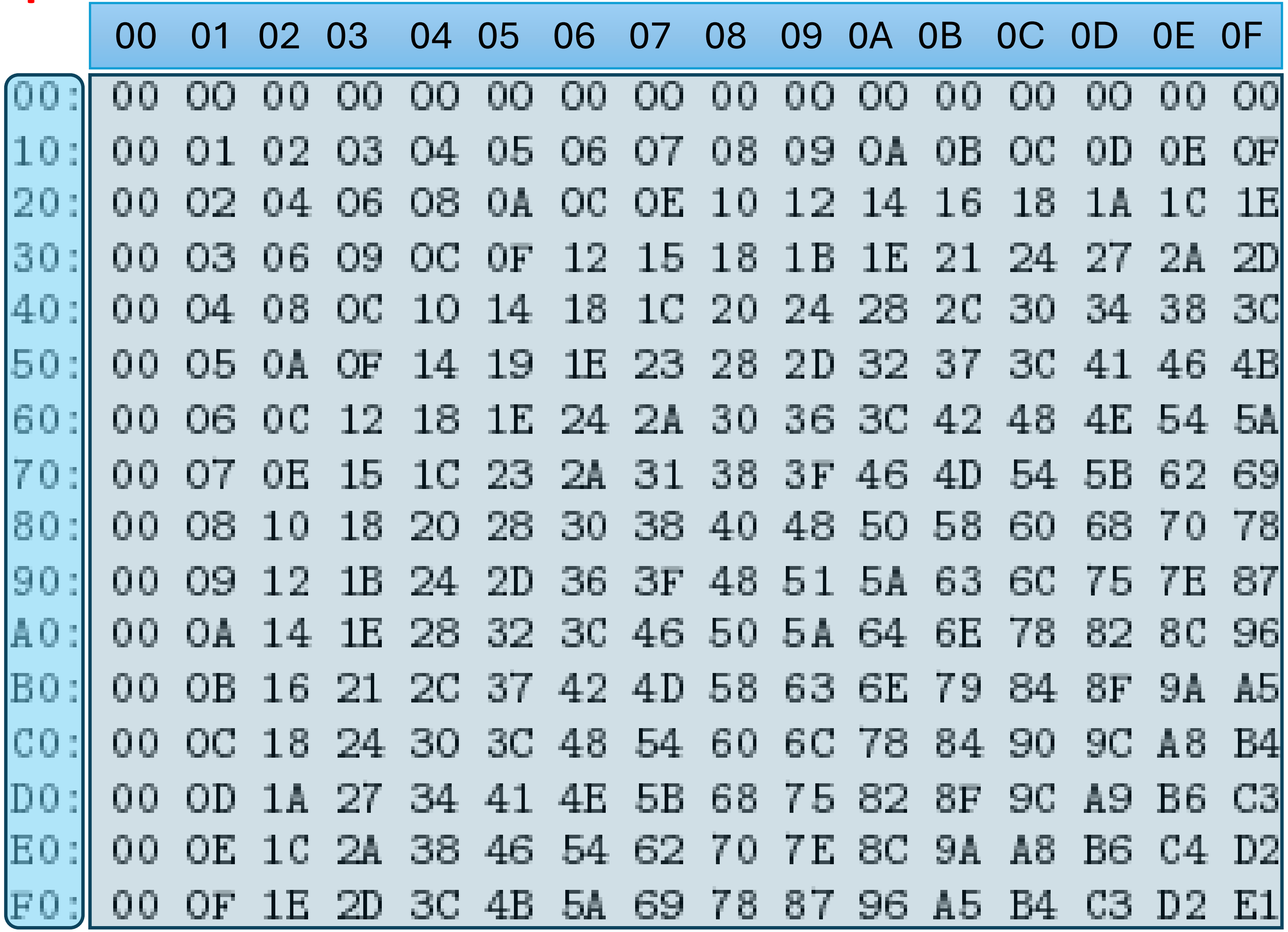
16 locations

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

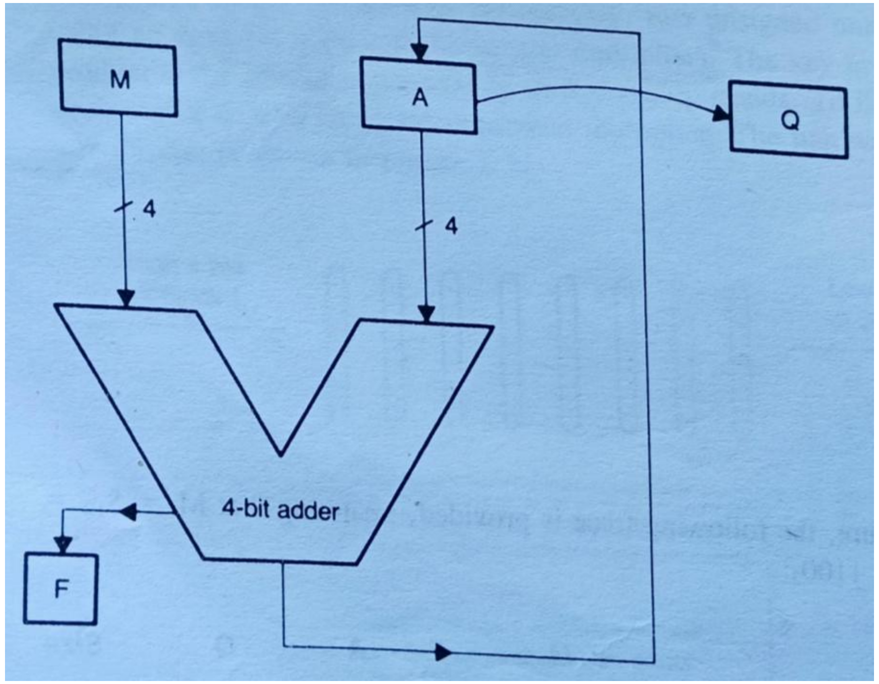
What is the memory size of this ROM?

265 bytes

Multiplication of Binary numbers

Sequential multiplier

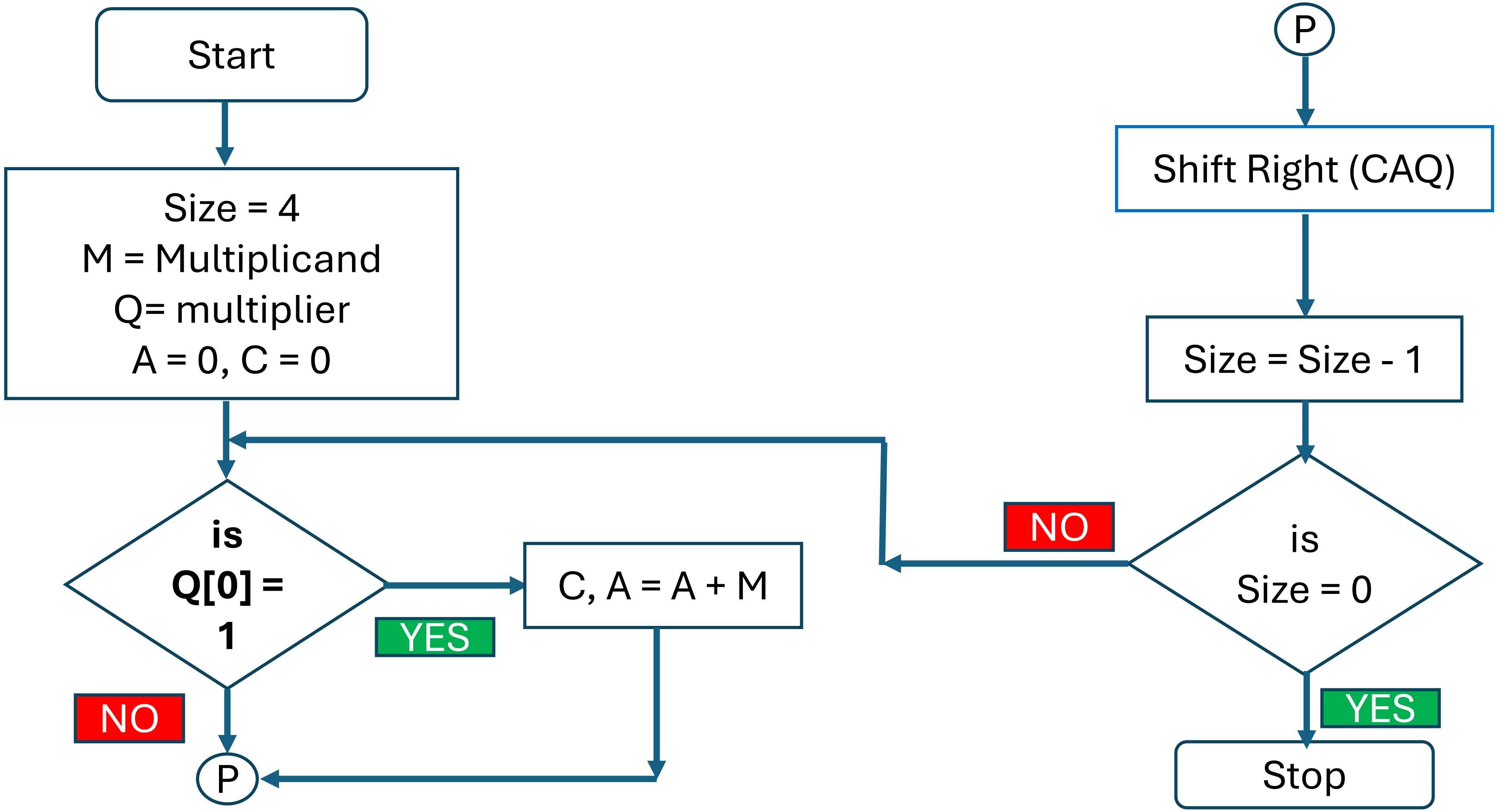
- Sequential multiplier use the fact that multiplication can be achieved using repeated addition
- It can multiply unsigned integers
- Next slides has a flowchart that can multiply two 4-bit unsigned numbers and a block diagram for implementing it
- **4x4 unsigned sequential multiplier**



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

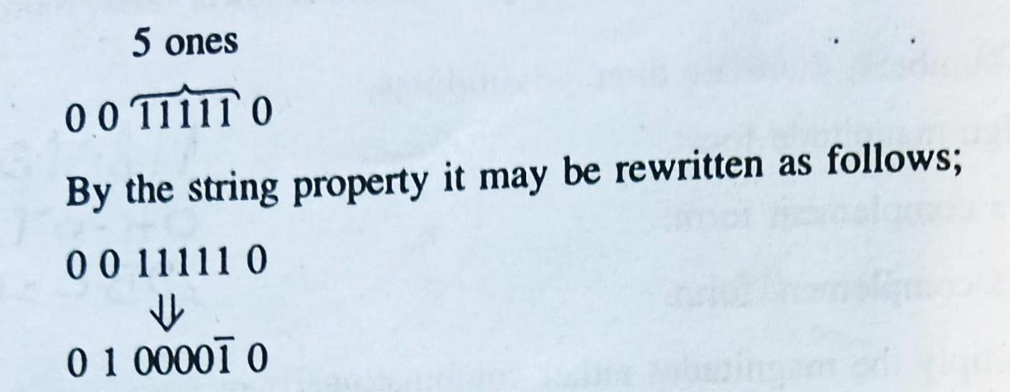
Signed multiplication

- **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* $\bar{1}$



5 ones

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

By the string property it may be rewritten as follows;

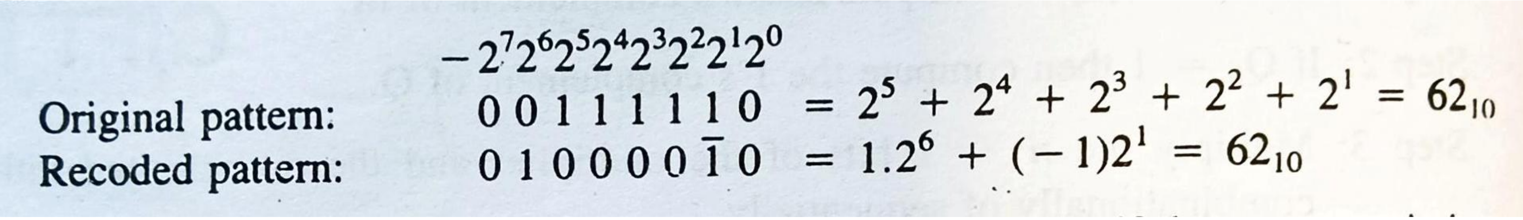
0 0 11111 0

$\Downarrow$

$0\;1\;0000\bar{1}\;0$

- 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*** $\bar{1}$***(over bar), assuming that the original sequence is in 2’s complement form, both represent same number***



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

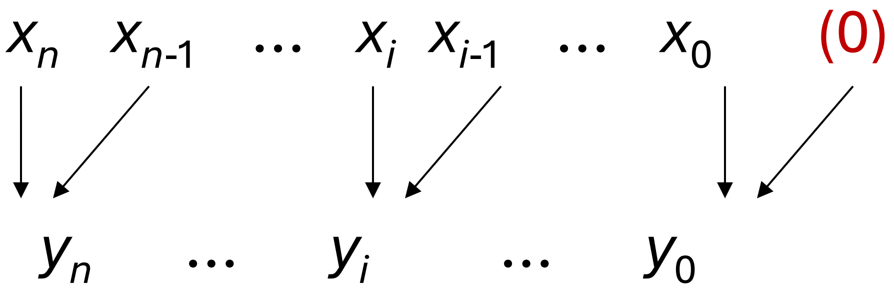
Original pattern: $0\;0\;1\;1\;1\;1\;1\;0 \;=\; 2^5 + 2^4 + 2^3 + 2^2 + 2^1 = 62_{10}$

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

Multiplication of Binary numbers

Booth's Recoded multiplication approach

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



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

x_i	$x_{i\text{-}1}$	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

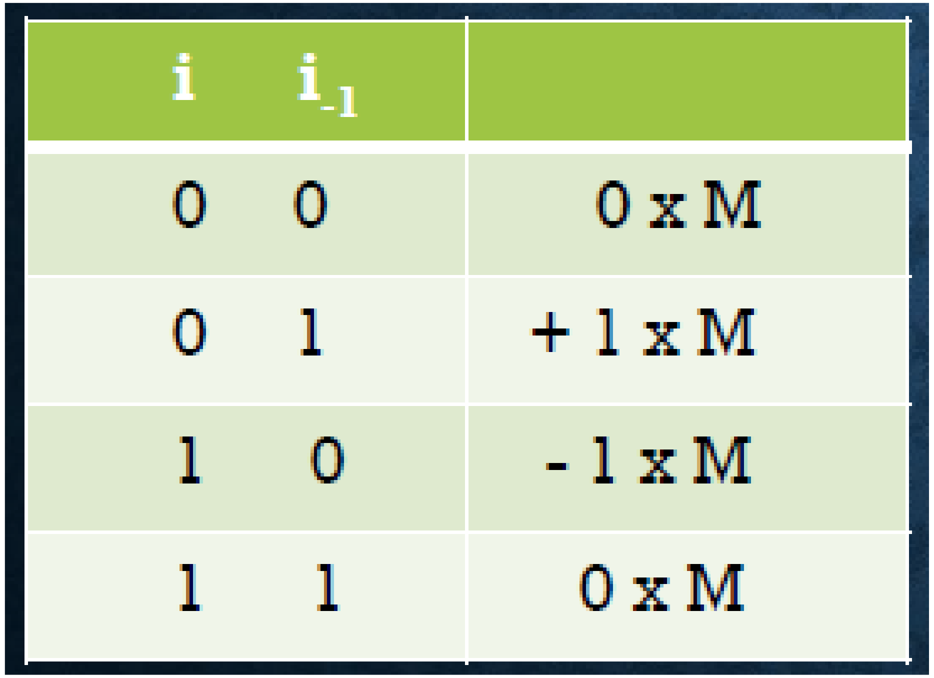
01111

$1000\bar{1}$

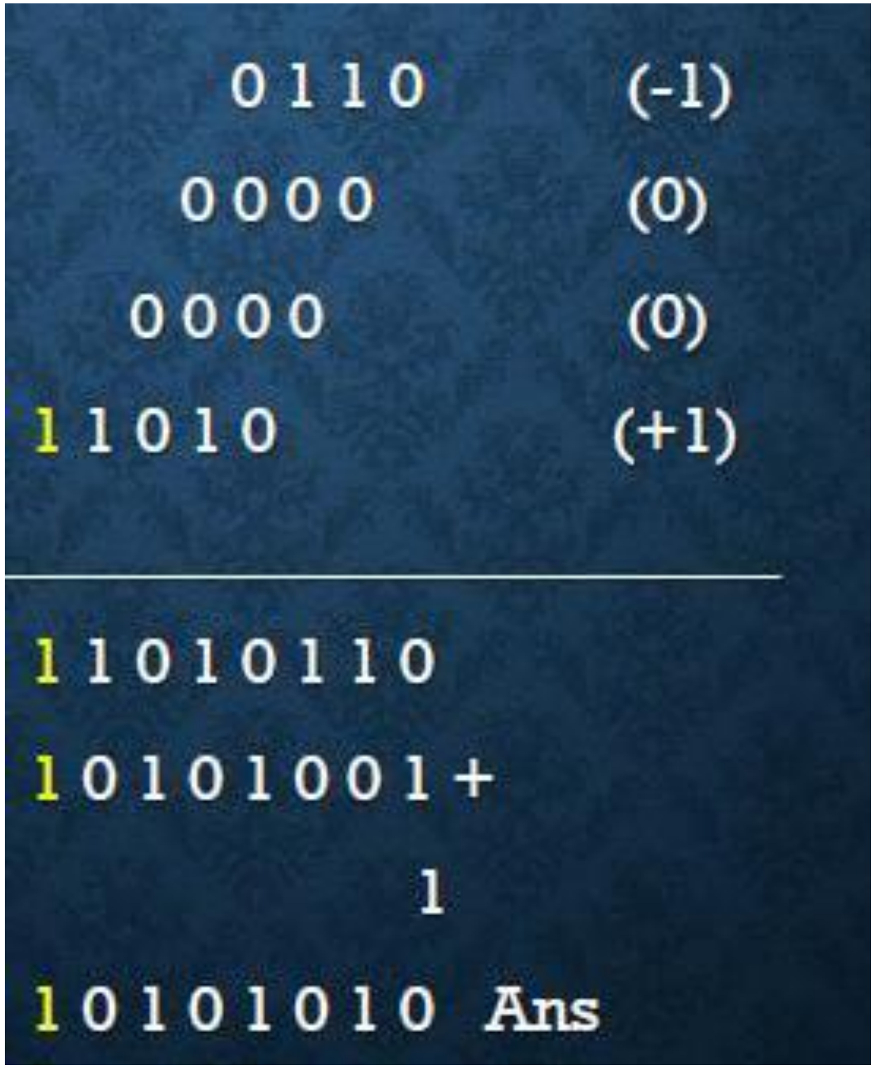
Multiplication of Binary numbers

❑ Booth's Recoded multiplier

- Perform -6x7 using Booths multiplication
- Multiplicand M=-6 = 1010 (2's complement of 0110)
- Multiplier Q=7=0111
- Recoded multiplier = +1, 0, 0, -1



i	i_{-1}	
0	0	0 x M
0	1	+ 1 x M
1	0	- 1 x M
1	1	0 x M



```
        0 1 1 0        (-1)
      0 0 0 0          (0)
    0 0 0 0            (0)
1 1 0 1 0              (+1)
--------------------------
1 1 0 1 0 1 1 0
1 0 1 0 1 0 0 1 +
              1
1 0 1 0 1 0 1 0  Ans
```

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.

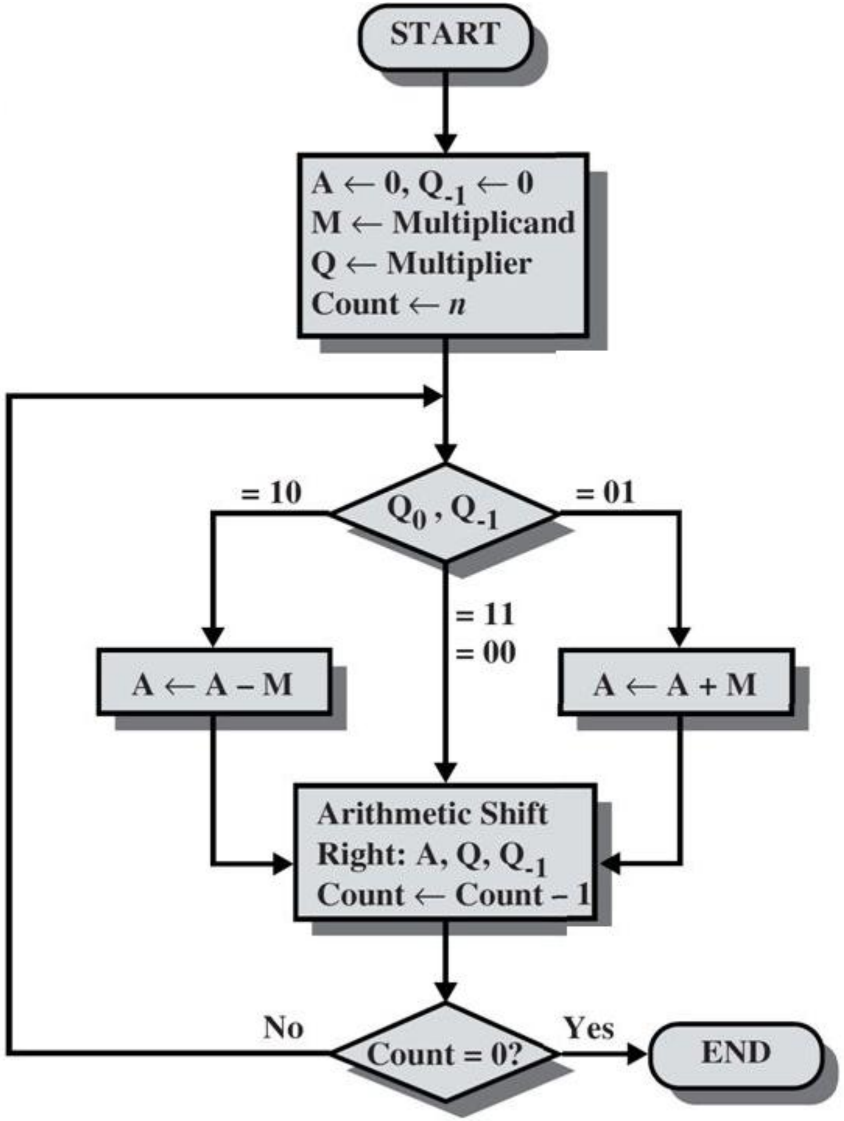
2. **Recoding:**
   - Examine pairs of bits in the multiplier (B) along with an extra bit (initially 0).
   - Depending on the pair of bits, decide whether to add, subtract, or do nothing with the multiplicand.

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 (1st iteration)
00000	00011	0	11000	ASR (2nd iteration)
01000	00011	0	11000	A<-A-M
00100	00001	1	11000	ASR (3rd iteration)
00010	00000	1	11000	ASR (4th iteration)
11010	00000	1	11000	A<-A+M
11101	**00000**	0	11000	ASR (5th 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:**
- 001010101(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

Multiplication of Binary numbers

❑ Modified Booth's Algorithm

- Instead of considering 2-bits of the multiplier at a time, 3-bits are considered, hence the number of computations are reduced
- Booth algorithm is modified to produce at most (n/2)+1 partial products, n is the number of bits
- It is also called as Radix-4/Bit pair Re-coding algorithm
- **Algorithm: (for unsigned numbers)**
  1. Pad the LSB with one zero.
  2. Pad the MSB with **2 zeros if n is even** and **1 zero if n is odd**.
  3. Divide the multiplier into overlapping groups of 3-bits.
  4. Determine partial product scale factor from **modified booth encoding table**.
  5. Compute the Multiplicand Multiples and get partial products
  6. Sum partial products to get final product

Multiplication of Binary numbers

❑ Modified Booth's Algorithm

Modified Booths Encoding Table			
Multiplier			Multiplicand selected
i+1	i	i-1	
0	0	0	0 x Multiplicand
0	0	1	+1 x Multiplicand
0	1	0	+1 x Multiplicand
0	1	1	+2 x Multiplicand
1	0	0	-2 x Multiplicand
1	0	1	-1 x Multiplicand
1	1	0	-1 x Multiplicand
1	1	1	0 x Multiplicand

- Can encode the digits by looking at three bits at a time
- Hardware must be able to add multiplicand times –2, -1, 0, 1 and 2
- Since it is having only doubling operation, generating partial products is not that hard (shifting and negating)

Multiplication of Binary numbers

❑ Modified Booth's Algorithm

- **Multiply 20 and 8 assuming n = 8-bits and unsigned**

1. **Pad LSB with 1 zero**

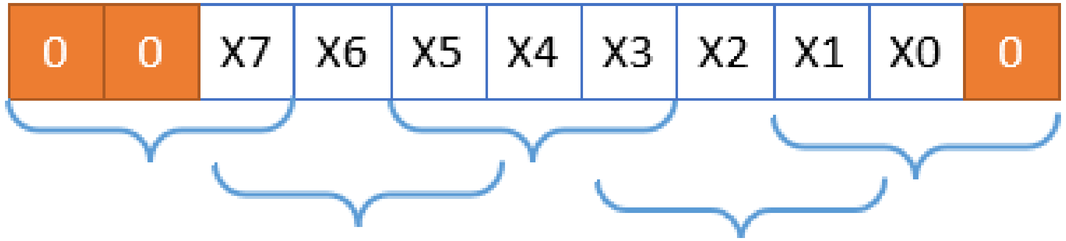
X7	X6	X5	X4	X3	X2	X1	X0	0

2. **n is even then pad the MSB with two zeros**

0	0	X7	X6	X5	X4	X3	X2	X1	X0	0

3. **Form 3-bit overlapping groups for n=8 we have 5 groups**

0	0	0	0	0	1	0	1	0	0	0



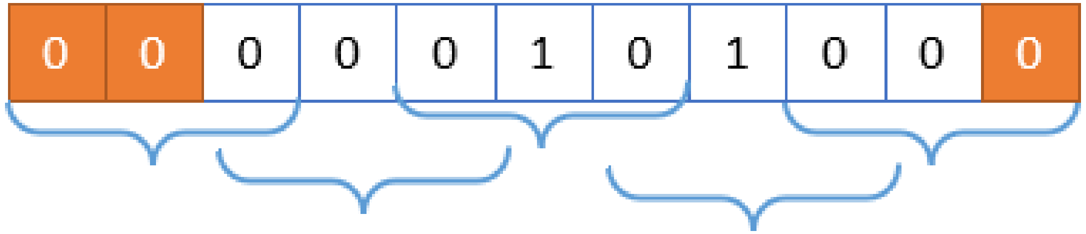
0	0	X7	X6	X5	X4	X3	X2	X1	X0	0

Multiplication of Binary numbers

❑ Modified Booth's Algorithm

- **Multiply 20 and 8 assuming n = 8-bits and unsigned**

4. Determine partial product scale factor from modified booth encoding table



Groups			Coding
0	0	0	0 × Y
0	1	0	1 × Y
0	1	0	1 × Y
0	0	0	0 × Y
0	0	0	0 × Y

X_{i+1}	X_i	X_{i-1}	Action
0	0	0	0 × Y
0	0	1	1 × Y
0	1	0	1 × Y
0	1	1	2 × Y
1	0	0	-2 × Y
1	0	1	-1 × Y
1	1	0	-1 × Y
1	1	1	0 × Y

Multiplication of Binary numbers

❑ Modified Booth's Algorithm

- **Multiply 20 and 8 assuming n = 8-bits and unsigned**

5. Compute the Multiplicand Multiples

Groups			Coding
0	0	0	0 × Y
0	1	0	1 × Y
0	1	0	1 × Y
0	0	0	0 × Y
0	0	0	0 × Y

```
                        0 0 0 0 1 0 0 0     8    = Y
                  ×     0 0 0 1 0 1 0 0    20    = X
--------------------------------------------
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   0 × Y
        0 0 0 0 0 0 0 0 0 0 1 0 0 0       1 × Y
        0 0 0 0 0 0 0 0 1 0 0 0           1 × Y
        0 0 0 0 0 0 0 0 0 0               0 × Y
        0 0 0 0 0 0 0 0                   0 × Y
--------------------------------------------
```

Multiplication of Binary numbers

❑ Modified Booth's Algorithm

- **Multiply 20 and 8 assuming n = 8-bits and unsigned**

5. **Sum Partial Products**

$$
\begin{array}{cccccccccccccccccc}
 & & & & & & & & & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 8 \\
 & & & & & & & \times & & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 20 \\
\hline
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \times Y \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & & & 1 \times Y \\
+ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & & & & & 1 \times Y \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & 0 \times Y \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & & & 0 \times Y \\
\hline
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 160
\end{array}
$$

Multiplication of Binary numbers

❑ Modified Booth's Algorithm

- **Algorithm: (for signed numbers)**
  1. Pad the LSB with one zero.
  2. If n is even don’t pad the MSB (n/2 PP’s) and if n is odd sign extend the MSB by 1 bit ((n+1)/2 PP’s).
  3. Divide the multiplier into overlapping groups of 3-bits.
  4. Determine partial product scale factor from modified booth encoding table.
  5. Compute the Multiplicand Multiples
  6. Sum Partial Products

Multiplication of Binary numbers

❑ Modified Booth's Algorithm

- Multiply -107 with 105

1. Pad LSB with 1 zero

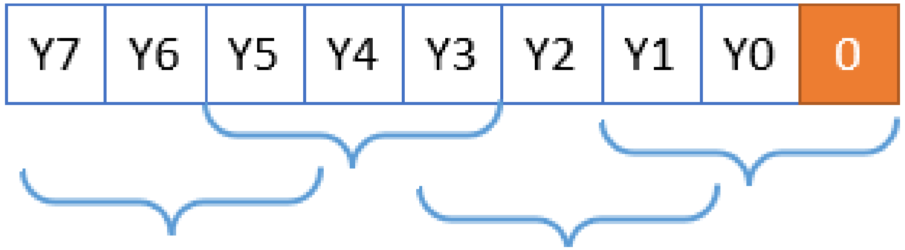
X7	X6	X5	X4	X3	X2	X1	X0	0

2. n is even then do not pad the MSB

X7	X6	X5	X4	X3	X2	X1	X0	0

3. Form 3-bit overlapping groups for n=8 we have 4 groups

0	1	1	0	1	0	0	1	0

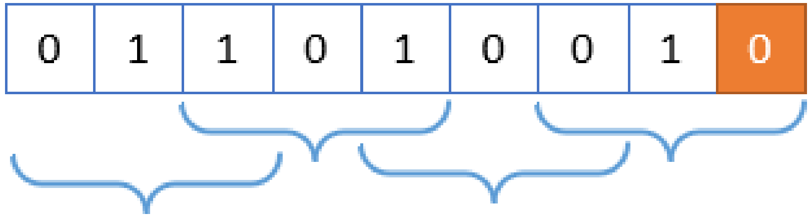


Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	0

Multiplication of Binary numbers

❑ Modified Booth's Algorithm

4. **Determine partial product scale factor from modified booth encoding table**



0	1	1	0	1	0	0	1	0

Groups			Coding
0	1	0	1 × Y
1	0	0	-2 × Y
1	0	1	-1 × Y
0	1	1	2 × Y

X_{i+1}	X_i	X_{i-1}	Action
0	0	0	0 × Y
0	0	1	1 × Y
0	1	0	1 × Y
0	1	1	2 × Y
1	0	0	-2 × Y
1	0	1	-1 × Y
1	1	0	-1 × Y
1	1	1	0 × Y

Multiplication of Binary numbers

❑ Modified Booth's Algorithm

5. Compute the Multiplicand Multiples

Note:

- Multiplication with 0 –> 0
- Multiplication with +1 –> Multiplicand
- Multiplication with -1 – >2’s complement of Multiplicand
- Multiplication with +2 –> shift Multiplicand left by 1-bit
- Multiplication with -2 –> 2’s complement of shifted Multiplicand

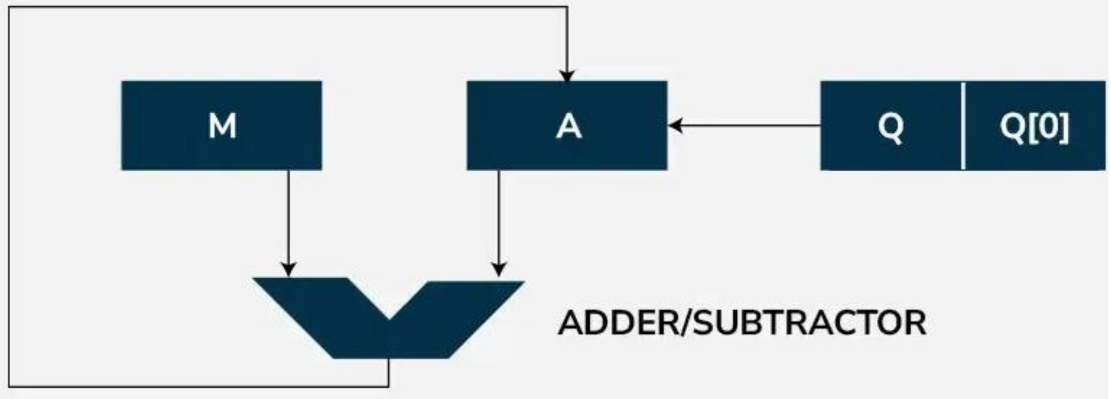
Groups			Coding
0	1	0	1 × Y
1	0	0	-2 × Y
1	0	1	-1 × Y
0	1	1	2 × Y

```
                1 0 0 1 0 1 0 1   -107 =Y
            ×   0 1 1 0 1 0 0 1   105 = X
-------------------------------
1 1 1 1 1 1 1 1 1 0 0 1 0 1 0 1   1 × Y
0 0 0 0 0 0 1 1 0 1 0 1 1 0       -2 × Y
0 0 0 0 0 1 1 0 1 0 1 1           -1 × Y
1 1 0 0 1 0 1 0 1 0               2 × Y
-------------------------------
1 1 0 1 0 1 0 0 0 0 0 1 1 1 0 1   -11235
```

Binary Division

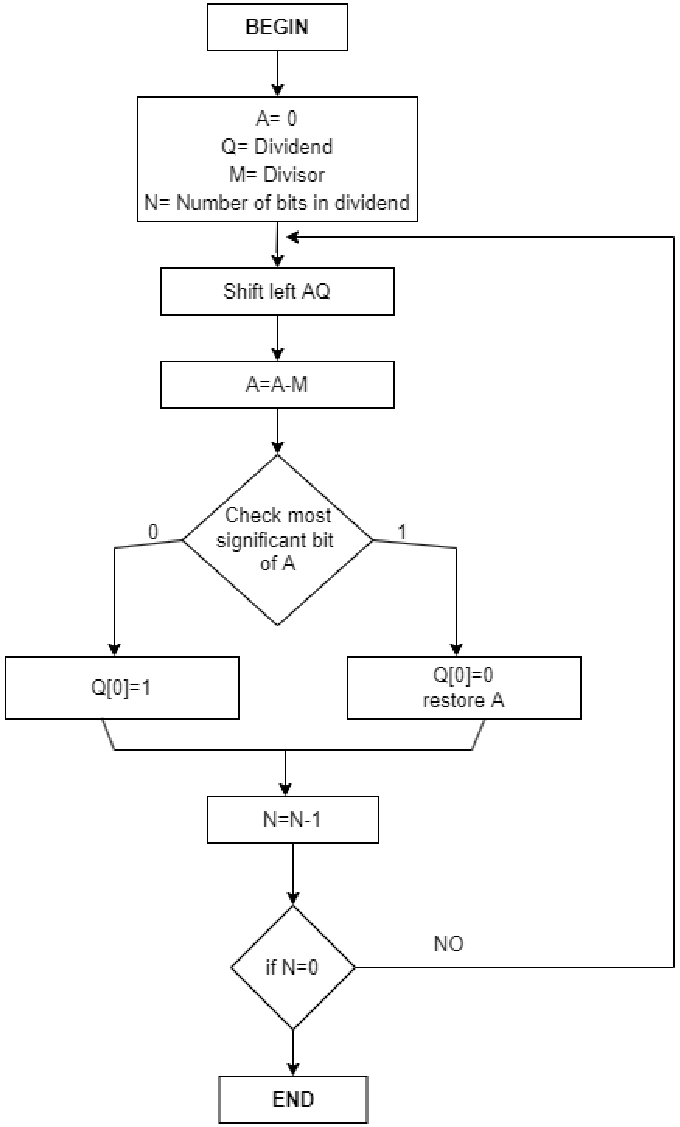
Restoring Division Algorithm for unsigned number

- The restoring division algorithm is a ***slow division algorithm*** that calculates the quotient digit by digit.
- Algorithm will generate a quotient and a remainder after the division
- Division algorithm uses registers for storing the numbers and calculations.
- ***Assumption:*** *dividend is greater than the divisor*.
- This division algorithm uses three registers.
  - *Register A* is initialized to 0,
  - *Register Q* stores the dividend, and
  - *Register M* stores the divisor.
- N is used as a counter and stores the number of bits present in the dividend.



Binary Division

❑ Restoring Division Algorithm for unsigned number



Step Involved:

- **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)
- **Step-2:** The content of register A and Q is shifted left as a single unit
- **Step-3:** The content of register M is subtracted from A and result is stored in A
- **Step-4:** The MSB of the A is checked,
    - if it is 0, the LSB of Q is set to 1, otherwise, the LSB of Q is set to 0 and value of register A is restored i.e the value of A before the subtraction with M
- **Step-5:** The value of counter n is decremented
- **Step-6:** If the value of n becomes ‘0’, we get out of the loop, otherwise we repeat from step 2
- **Step-7:** Finally, the register Q contain the quotient and A contain remainder

Binary Division

❑ Restoring Division Algorithm for unsigned number

- **Example: Divide 7/3   Dividend=0111(7) and Divisor=0011(3)**

N	M	A	Q	Operation
4	0011	0000	0111	Initialization
	0011	0000	111_	SHL AQ
	0011	1101	111_	A= A-M
	0011	0000	1110	Q[0]=0 and restore A
3	0011	0001	110_	SHL AQ
	0011	1110	110_	A=A-M
	0011	0001	1100	Q[0]=0 and restore A
2	0011	0011	100_	SHL AQ
	0011	0000	100_	A=A-M
	0011	0000	1001	Q[0]=1
1	0011	0001	001_	SHL AQ
	0011	1110	001_	A=A-M
	0011	0001	0010	Q[0]= 0 and restore A

Result:
quotient = Q = 0010 (2) and
remainder = A = 0001 (1)

Binary Division

Non-Restoring Division Algorithm for unsigned number

- It is a division technique for unsigned binary values that simplifies the procedure by eliminating the restoring phase.
- The non-restoring division is simpler and more effective than restoring division
- It just employs addition and subtraction operations instead of restoring division
- Restoring requires extra steps to restore the original result after a failed subtraction.
- The non-restoring division algorithm is more complex as compared to the restoring division algorithm
- It has an advantage that it contains only one decision and addition/subtraction per quotient bit.
- After performing the subtraction operation, there will not be any restoring steps.
- Due to this, the numbers of operations basically cut down up to half.
- Due to reduced number of operation, the execution of this algorithm will be fast.

Binary Division

Non-Restoring Division Algorithm for unsigned number

- **Step-1:** Initialize registers

  Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend
- **Step-2:** If sign bit of register A is 1 then shift left content of AQ and A=A+M,

  otherwise shift left content of AQ and A=A-M (means add 2's complement of M to A and store it to A)
- **Step-3:** If sign bit of register A is 1

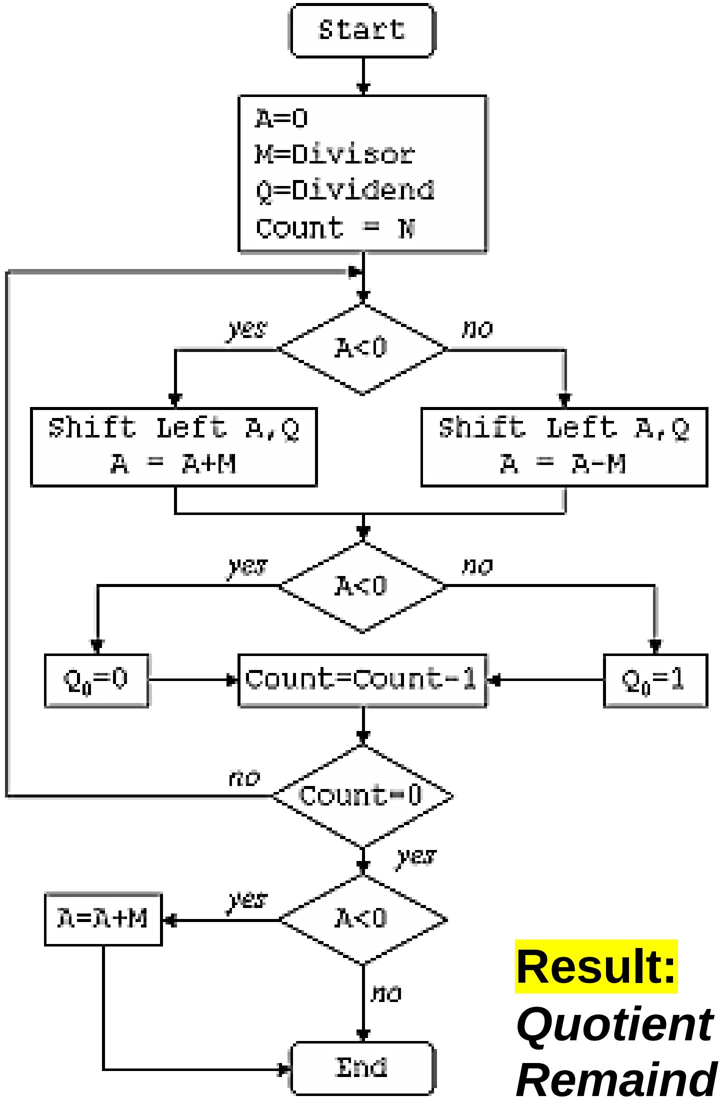
  then least significant bit of Q, i.e. Q[0]=0

  otherwise Q[0]=1
- **Step-4:** The value of counter n is decremented by 1
- **Step-5:** If value of n becomes 0 then goto next step otherwise repeat from step-2
- **Step-6:** If sign bit of register A is 1

  then A=A+M
- **Step-7:** Now register Q contains Quotient value and A contains Remainder value

Binary Division

❑ Non-Restoring Division Algorithm for unsigned number: Divide 11 by 3



n	M	A	Q	Operation
4	00011	00000	1011	initialize
4	00011	00001	011_	shift left AQ
		11110	011_	A=A-M
		11110	0110	Q[0]=0
3	00011	11100	110_	shift left AQ
		11111	110_	A=A+M
		11111	1100	Q[0]=0
2	00011	11111	100_	shift left AQ
		00010	100_	A=A+M
		00010	1001	Q[0]=1
1	00011	00101	001_	shift left AQ
		00010	001_	A=A-M
		00010	0011	Q[0]=1

Result:

Quotient in register Q =3

Remainder in register A=2

Binary Division

❑ Non-Restoring Division Algorithm for unsigned number

- **Example: divide 13 by 4**

n	M	A	Q	Operation
4	00100	00000	1101	initialize
4	00100	00001	101_	shift left AQ
		11101	101_	A=A-M
		11101	1010	Q[0]=0
3	00100	11011	010_	shift left AQ
		11111	010_	A=A+M
		11111	0100	Q[0]=0
2	00100	11110	100_	shift left AQ
		00010	100_	A=A+M
		00010	1001	Q[0]=1
1	00100	00101	001_	shift left AQ
		00001	001_	A=A-M
		00001	0011	Q[0]=1

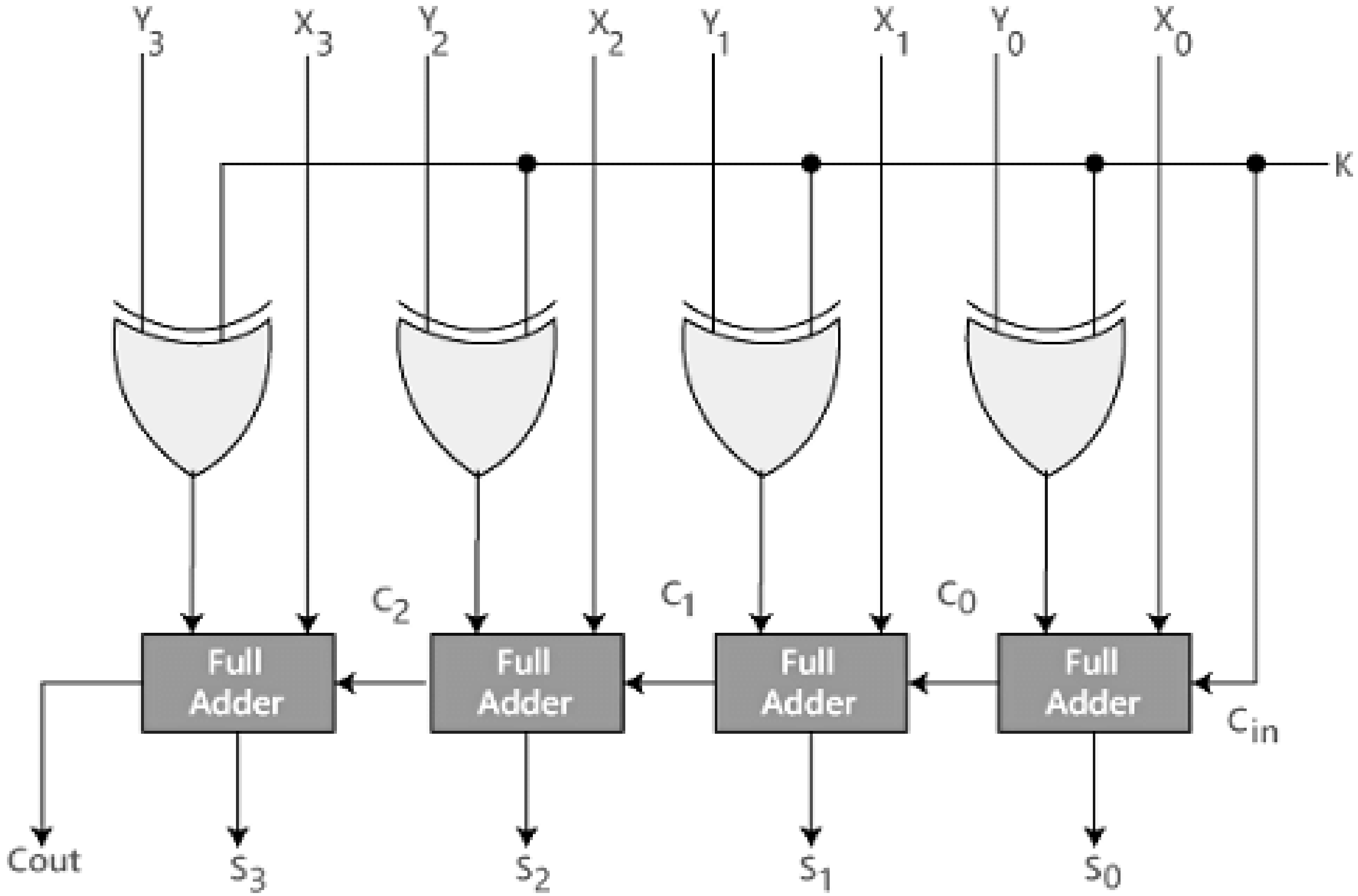
Result:

Register Q = 3 (quotient)

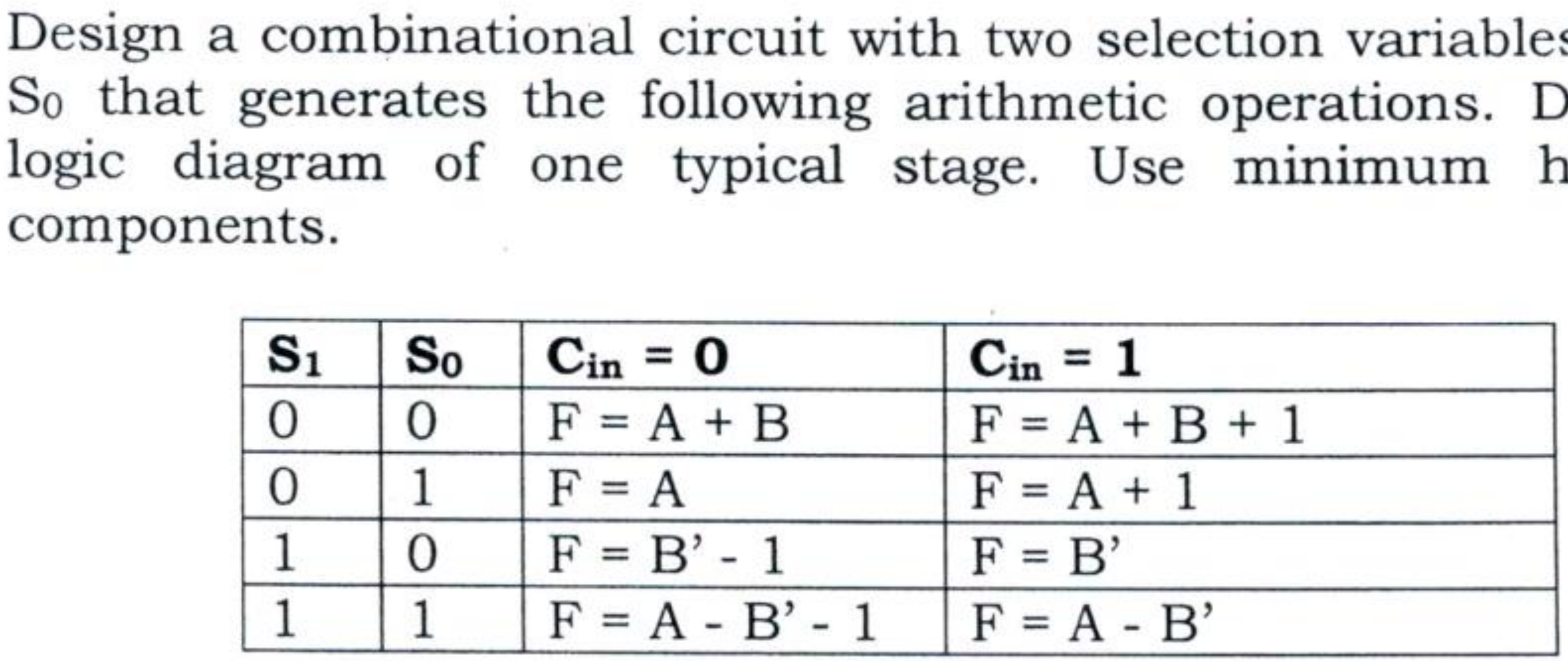
Register A = 1(remainder)

Design Examples

- **Design an adder or subtractor circuit with one selection variable ‘K’ and two inputs X and Y. When K=0 the circuit performs X+Y. When K=1, the circuit performs X-Y by taking 2’s complement of Y?**



Design Examples



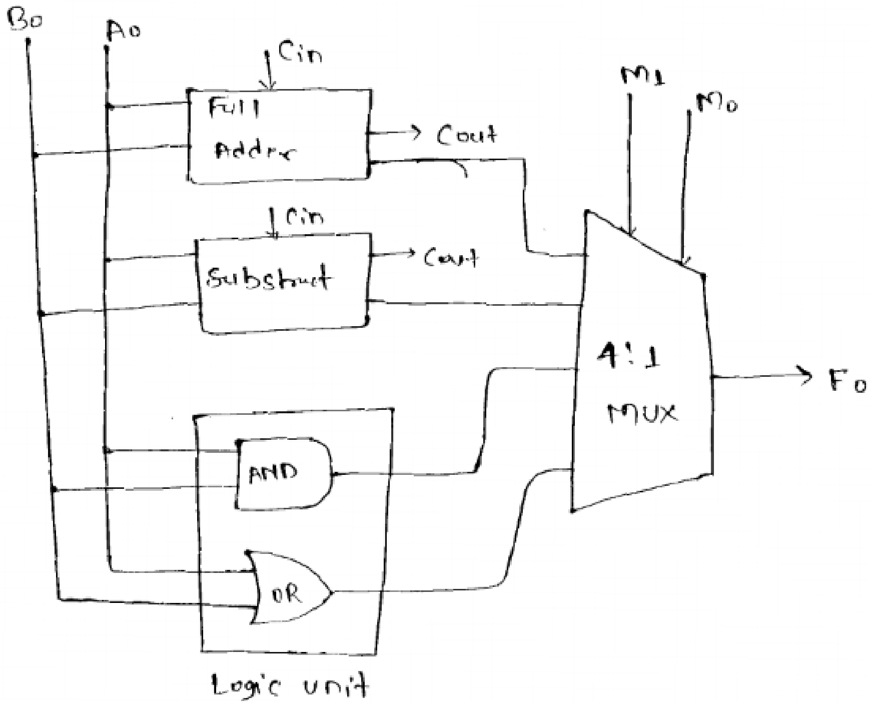
Design a combinational circuit with two selection variables S_1 and S_0 that generates the following arithmetic operations. Draw the logic diagram of one typical stage. Use minimum hardware components.

S_1	S_0	$C_{in} = 0$	$C_{in} = 1$
0	0	$F = A + B$	$F = A + B + 1$
0	1	$F = A$	$F = A + 1$
1	0	$F = B' - 1$	$F = B'$
1	1	$F = A - B' - 1$	$F = A - B'$

Binary Division

- **Design an 1-bit ALU that performs AND, OR logic operations and ADD, SUB arithmetic operations**

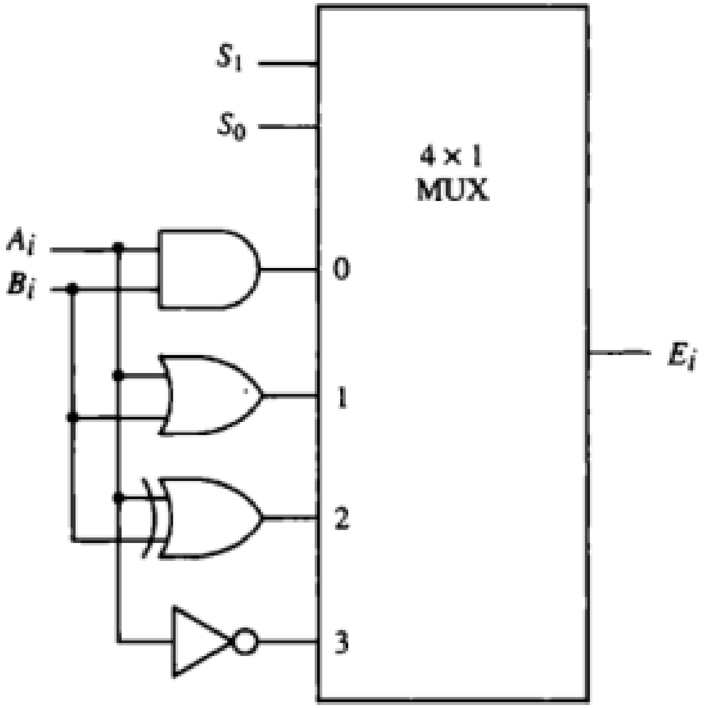
M1	M0	Operation
0	0	ADD
0	1	SUB
1	0	AND
1	1	OR



Binary Division

- Design a combinational circuit which perform operations given in the table:

S1	S0	Output	Operation
0	0	F=A AND B	Logical AND
0	1	F=A OR B	Logical OR
1	0	F= A XOR B	Logical EX-OR
1	1	F= complement A	Inverting a bit



Control Unit

- **Reference:**
- **Chapter 15: Control Unit Operation**

William Stallings, “COMPUTER ORGANIZATION AND ARCHITECTURE *DESIGNING FOR PERFORMANCE”,* EIGHTH EDITION

Control Unit

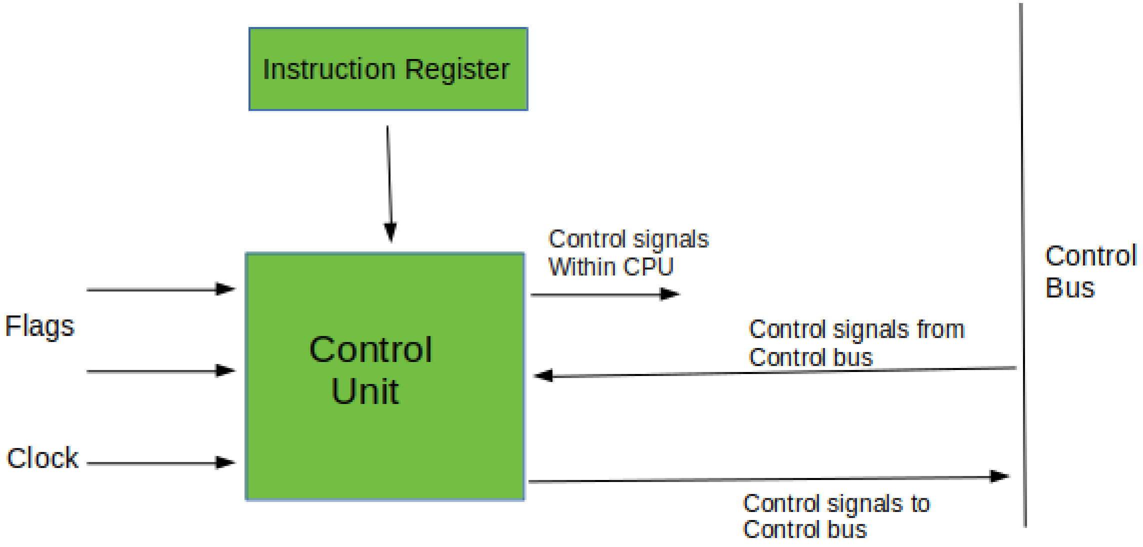
❑ Introduction

- A CPU is the most important component of a computer system.
- A control unit is a part of the CPU.
- A control unit controls the operations of all parts of the computer, but it does not carry out any data processing operations.
- The execution of an instruction involves the execution of a sequence of substeps, generally called cycles.
- ***Example:*** an execution may consist of fetch, indirect, execute, and interrupt cycles.
- Each cycle is in turn made up of a sequence of more fundamental operations, called ***micro-operations.***
- A single micro-operation generally involves:
  - a transfer between registers
  - a transfer between a register and an external bus, or
  - a simple ALU operation.

Control Unit

❑ Introduction

- The control unit of a processor performs two tasks:
  1. It causes the processor to step through a series of micro-operations in the proper sequence, based on the program being executed, and
  2. it generates the control signals that cause each micro-operation to be executed..
- The control signals generated by the control unit cause the opening and closing of logic gates, resulting in the transfer of data to and from registers and the operation of the ALU.



Instruction Register
Control Unit
Flags
Clock
Control signals Within CPU
Control signals from Control bus
Control signals to Control bus
Control Bus

Block Diagram of the Control Unit

Control Unit

❑ Functions of the Control Unit

- It coordinates the sequence of data movements into, out of, and between a processor’s many sub-units.
- It interprets instructions.
- It controls data flow inside the processor.
- It receives external instructions or commands to which it converts to sequence of control signals.
- It controls many execution units(i.e.ALU , data buffers and registers) contained within a CPU.
- It also handles multiple tasks, such as fetching, decoding, execution handling and storing results.

The control unit of a CPU fetches and executes instructions, playing a critical role in system performance. Its design ensures smooth operation of various components.

Control Unit

❑ Micro-operations

- The operation of a computer, in executing a program, consists of a sequence of instruction cycles, with one machine instruction per cycle
- Each instruction cycle is made up of a number of smaller units.
- One subdivision is *fetch*, *indirect*, *execute*, *and interrupt*, with only fetch and execute cycles always occurring.
- Each of the smaller cycles involves a series of steps known as ***Micro-operations***
- Micro-operations are the functional, or atomic, operations of a processor

Control Unit

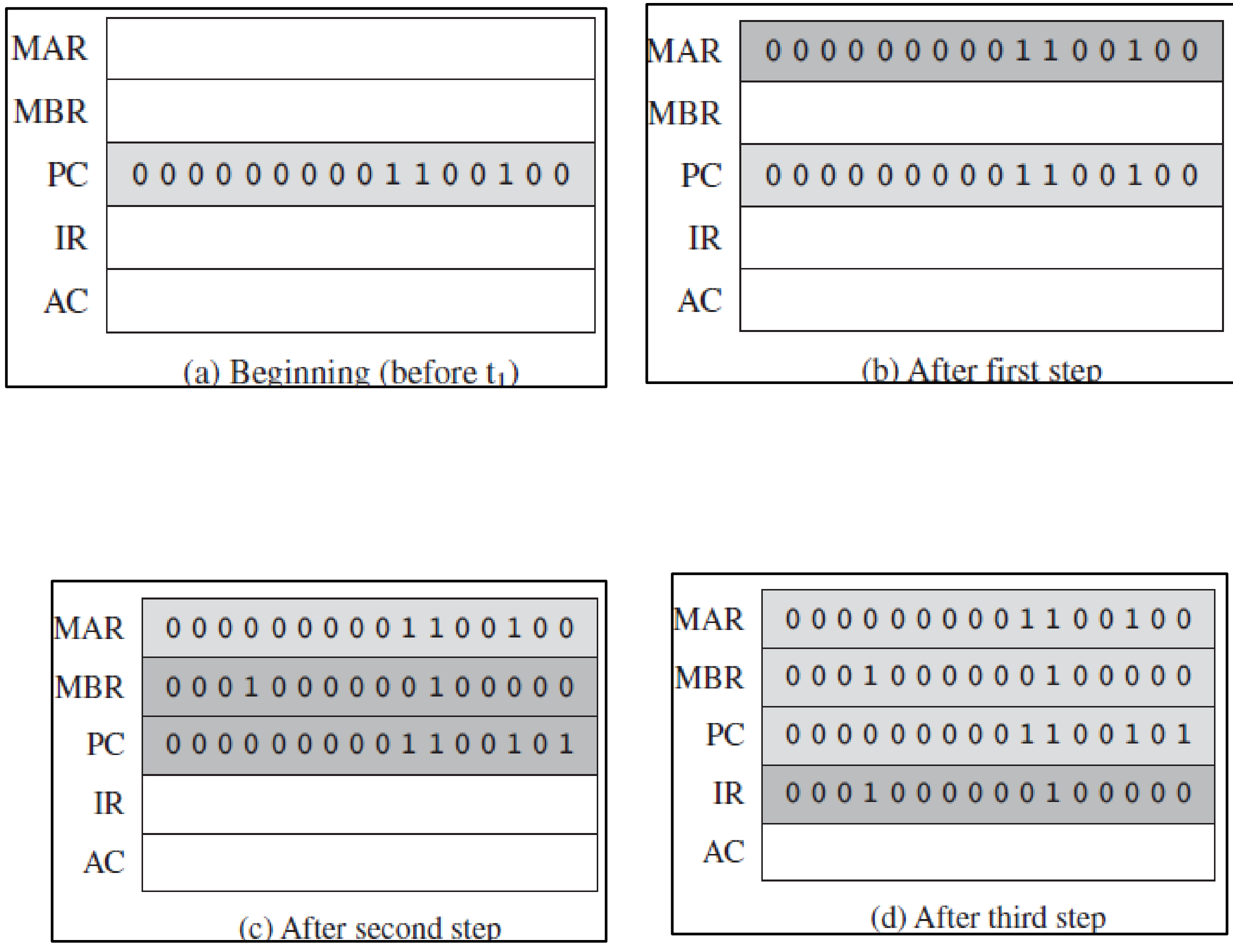
Micro-operation – Example -The Fetch Cycle

- Occurs at the beginning of each instruction cycle and causes an instruction to be fetched from memory
- Four registers are involved:
  - ***Memory address register (MAR):*** connected to the address lines of the system bus. It specifies the address in memory for a read or write operation.
  - ***Memory buffer register (MBR):*** connected to the data lines of the system bus. It contains the value to be stored in memory or the last value read from memory.
  - ***Program counter (PC):*** Holds the address of the next instruction to be fetched.
  - ***Instruction register (IR):*** Holds the last instruction fetched.

Control Unit

❑ Micro-operation – Example -The Fetch Cycle

- **At beginning:** the address of the next instruction to be executed is in the program counter (PC)
- **First step:** move that address to the memory address register (MAR) because this is the only register connected to the address lines of the system bus.
- ***Second step:*** bring in the instruction
- The desired address (in the MAR) is placed on the address bus, the control unit issues a READ command on the control bus, and the result appears on the data bus and is copied into the memory buffer register (MBR).
- ***Third step:*** move the contents of the MBR to the instruction register (IR).This frees up the MBR for use during a possible indirect cycle.



(a) Beginning (before t_1)

(b) After first step

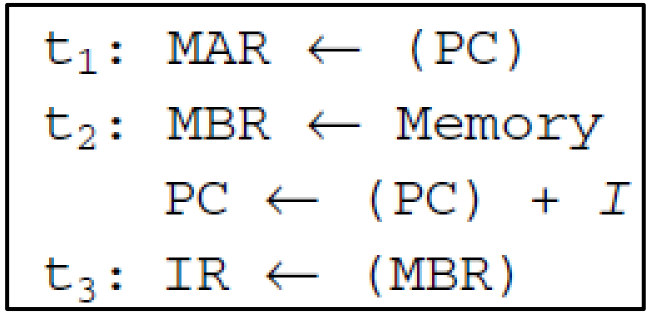
(c) After second step

(d) After third step

Control Unit

Micro-operation – Example -The Fetch Cycle

- Each clock pulse defines a time unit.
- All time units are of equal duration.
- Each micro-operation can be performed within the time of a single time unit.
- The notation(t_1, t_2, t_3) represents successive time units.
- **Symbolical representation of sequence of events as follows:**



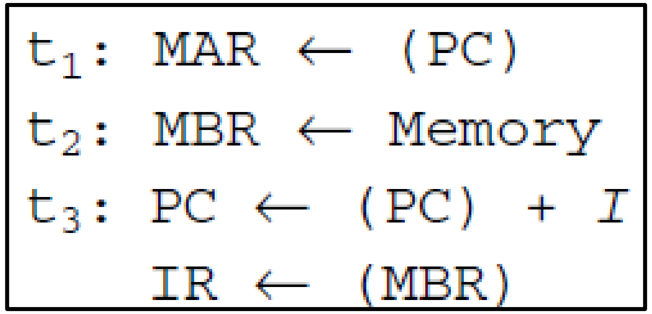
t_1: MAR ← (PC)

t_2: MBR ← Memory

PC ← (PC) + I

t_3: IR ← (MBR)

- **micro-operation could have been grouped without affecting the fetch operation:**



t_1: MAR ← (PC)

t_2: MBR ← Memory

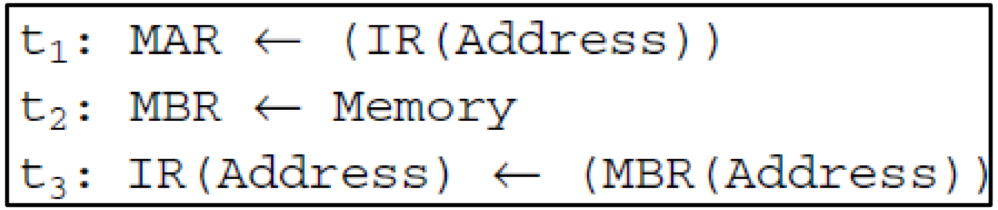
t_3: PC ← (PC) + I

IR ← (MBR)

Control Unit

Micro-operation –

- ***The Indirect Cycle***
- The address field of the instruction is transferred to the MAR. This is then used to fetch the address of the operand. Finally, the address field of the IR is updated from the MBR

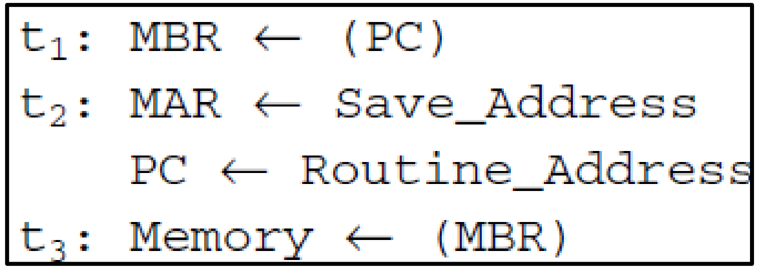


t_1: MAR ← (IR(Address))

t_2: MBR ← Memory

t_3: IR(Address) ← (MBR(Address))

- ***The Interrupt Cycle***
- In the first step, the contents of the PC are transferred to the MBR, so that they can be saved for return from the interrupt. Then the MAR is loaded with the address at which the contents of the PC are to be saved, and the PC is loaded with the address of the start of the interrupt-processing routine.



t_1: MBR ← (PC)

t_2: MAR ← Save_Address

PC ← Routine_Address

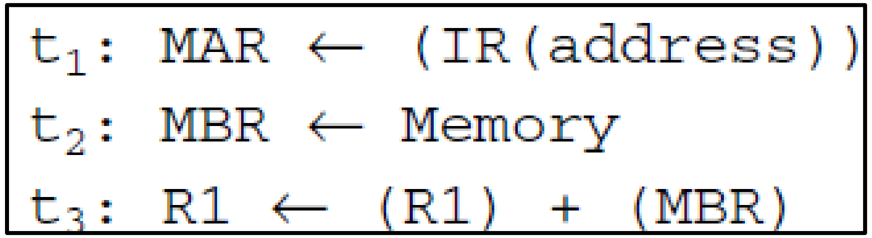
t_3: Memory ← (MBR)

Control Unit

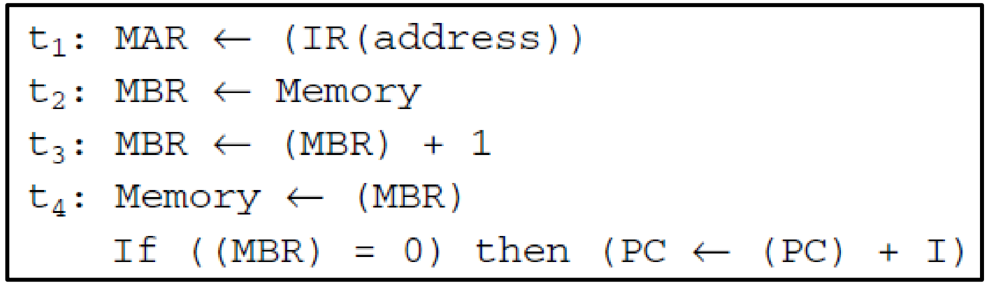
❑ Micro-operation

- ***The Execute Cycle***
- The fetch, indirect, and interrupt cycles are simple and predictable.
- Each involves a small, fixed sequence of micro-operations and, in each case, the same micro-operations are repeated each time around.
- This is not true of the execute cycle.
- Because of the variety opcodes, there are a number of different sequences of micro-operations that can occur.
- **Examples:** **ADD R1, X** **ISZ X**

(add memory content X to R1) (increment and skip if zero)



t_1: MAR ← (IR(address))
t_2: MBR ← Memory
t_3: R1 ← (R1) + (MBR)



t_1: MAR ← (IR(address))
t_2: MBR ← Memory
t_3: MBR ← (MBR) + 1
t_4: Memory ← (MBR)
If ((MBR) = 0) then (PC ← (PC) + I)

Control Unit

Micro-operation

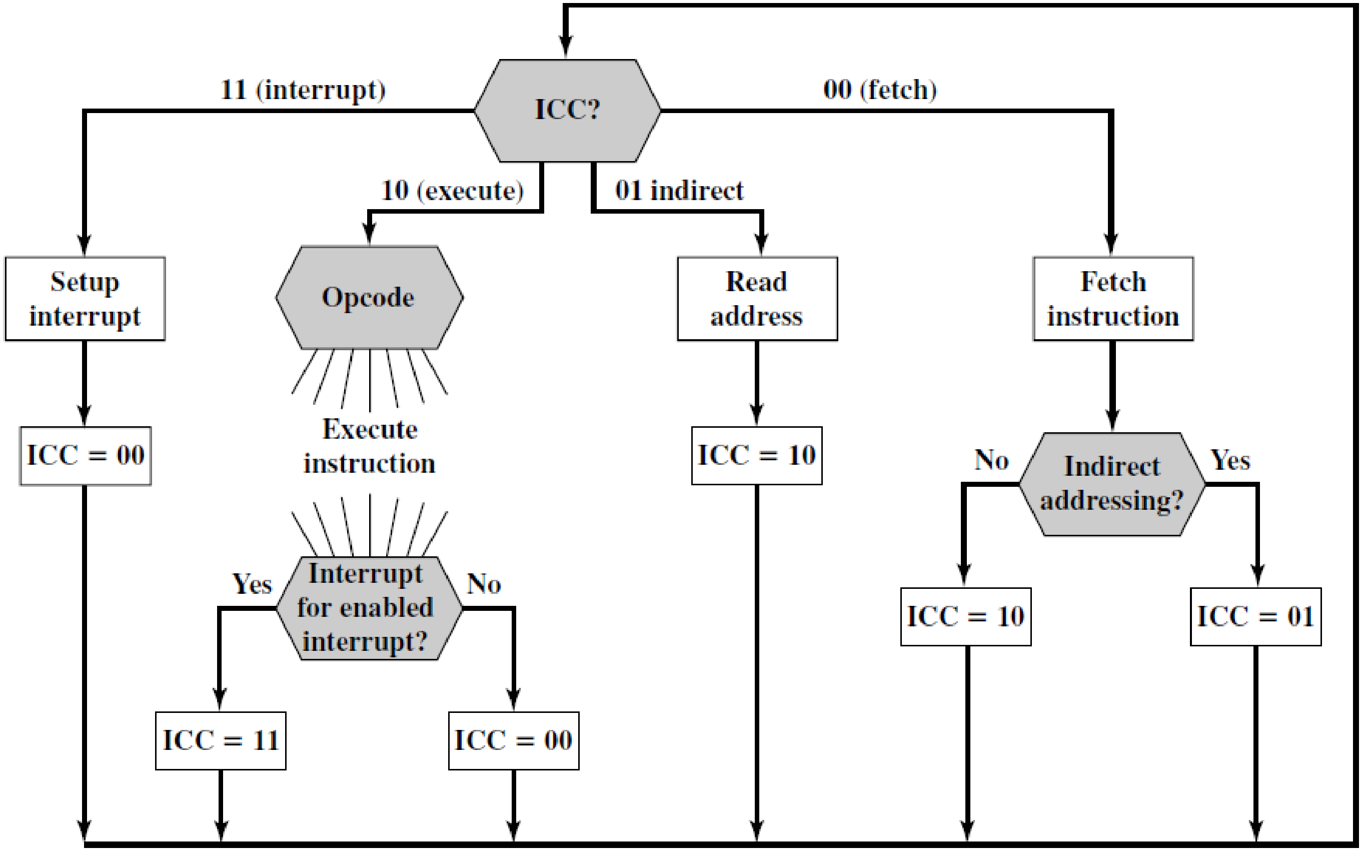
- ***The Instruction Cycle***
- 2-bit register called the instruction cycle code (ICC) designates the state of the processor in terms of which portion of the cycle it is in:

00: Fetch

01: Indirect

10: Execute

11: Interrupt



Control of the processor

❑ Functional Requirements

- The decomposing of behavior or functioning of the processor into elementary operations, called micro-operations.
- By reducing the operation of the processor to its most fundamental level, we are able to define exactly what it is **that the control unit must cause to happen**.
- Thus, we can define the functional requirements for the control unit: those functions that the control unit must perform.
- A definition of these functional requirements is the basis for the design and implementation of the control unit.
- The following three-step process leads to a characterization of the control unit:
  1. Define the basic elements of the processor.
  2. Describe the micro-operations that the processor performs.
  3. Determine the functions that the control unit must perform to cause the micro-operations to be performed.

Control of the processor

❑ Basic functional elements of the processor

- **ALU:** the functional essence of the computer
- **Registers:**
  - ○ used to store data internal to the processor.
  - ○ Some registers contain status information needed to manage instruction sequencing (e.g., a program status word).
  - ○ Others contain data that go to or come from the ALU, memory, and I/O modules
- **Internal data paths:** used to move data between registers, between register and ALU
- **External data paths:** links registers to memory and I/O modules, often by means of a system bus.
- **Control unit:** causes operations to happen within the processor.
- **The execution of a program consists of operations involving these processor elements.**

Control of the processor

❑ micro-operations that the processor performs

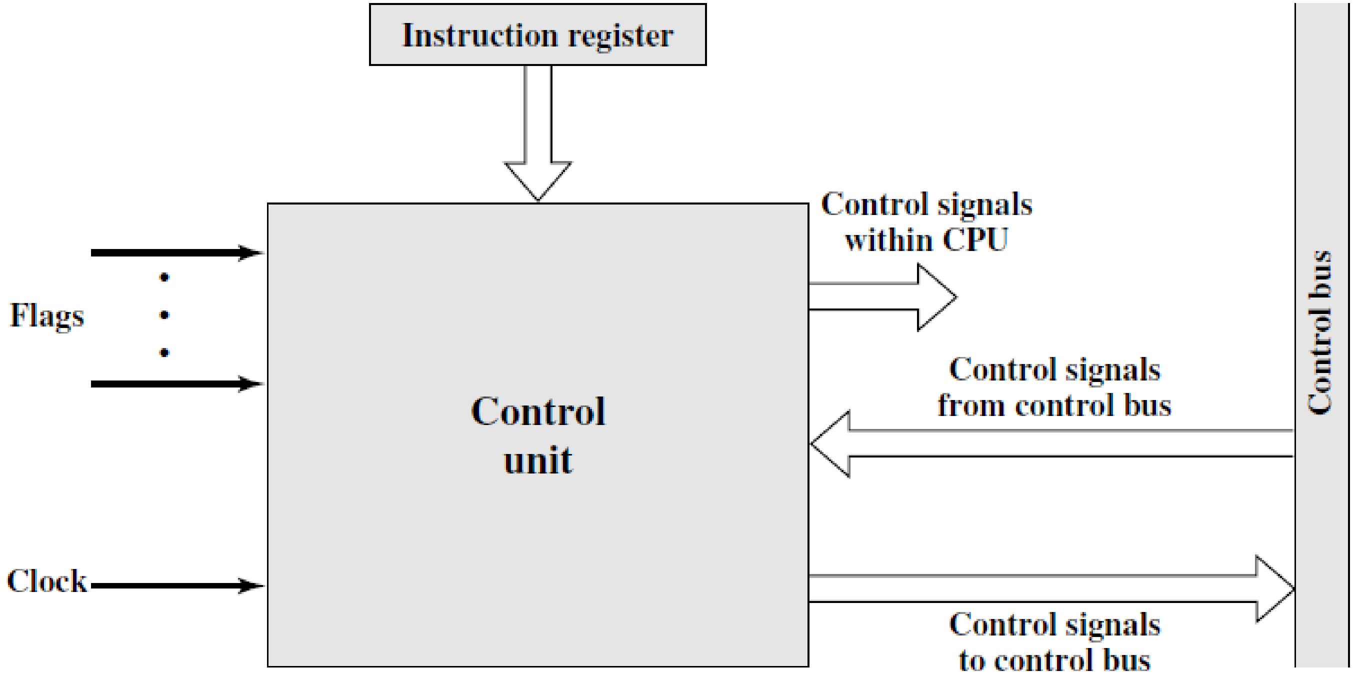
- All micro-operations fall into one of the following categories:
  - Transfer data from one register to another.
  - Transfer data from a register to an external interface (e.g., system bus).
  - Transfer data from an external interface to a register.
  - Perform an arithmetic or logic operation, using registers for input and output
- All of the micro-operations to execute every instruction in the instruction set, fall into one of these categories

- The control unit performs two basic tasks:
  - ***Sequencing:*** The control unit causes the processor to step through a series of micro-operations in the proper sequence, based on the program being executed.
  - ***Execution:*** The control unit causes each micro-operation to be performed.

Control Unit

❑ Control signals

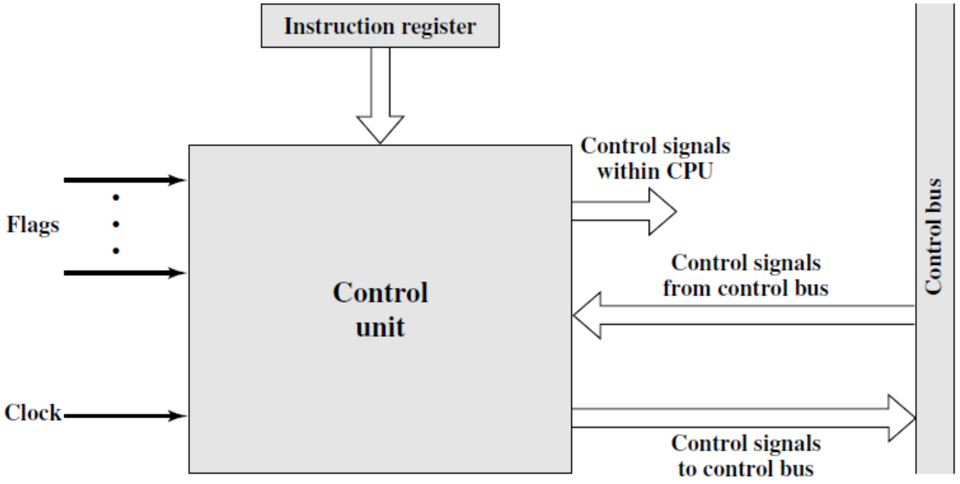
- **External specifications of the control unit:** Control unit to perform its function, it must have inputs that allow it to determine the state of the system and outputs that allow it to control the behavior of the system.
- **Internal specifications of the control unit:** the control unit must have the logic required to perform its sequencing and execution functions.


Instruction register
Flags
Clock
Control unit
Control signals within CPU
Control signals from control bus
Control signals to control bus
Control bus

Control Unit

❑ Control signals

- **The inputs are**
  - ○ **Clock:** This is how the control unit "keeps time." The control unit causes one micro-operation to be performed for each clock pulse.
  - ○ **Instruction register:** The opcode and addressing mode of the current instruction are used to determine which micro-operations to perform during the execute cycle.
  - ○ **Flags:** These are needed to determine the status of the processor and the outcome of previous ALU operations.
  - ○ **Control signals from control bus:** The control bus portion of the system bus provides signals to the control unit.
- **The outputs are as follows:**
  - ○ **Control signals within the processor:** These are two types:
    - ✓ those that cause data to be moved from one register to another, and
    - ✓ those that activate specific ALU functions.
  - ○ **Control signals to control bus:** These are also of two types:
    - ✓ control signals to memory, and
    - ✓ control signals to the I/O modules.


Instruction register
Control signals within CPU
Flags
Control unit
Control signals from control bus
Control bus
Clock
Control signals to control bus

Control Unit

❑ Types of control signals

- Three types of control signals are used:
  1. Signals that activate an ALU function,
  2. Signal that activate a data path, and
  3. Signals that are on the external system bus or other external interface.
- All of these signals are ultimately applied directly as binary inputs to individual logic gates.

Control Unit

❑ **Example: Fetch Cycle:**

- **First step:** To transfer the contents of the PC to the MAR. The control unit does this by activating the control signal that opens the gates between the bits of the PC and the bits of the MAR.
- **Second step:** To read a word from memory into the MBR and increment the PC. The control unit does this by sending the following control signals simultaneously:
  - ✓ A control signal that opens gates, allowing the contents of the MAR onto the address bus
  - ✓ A memory read control signal on the control bus
  - ✓ A control signal that opens the gates, allowing the contents of the data bus to be stored in the MBR
  - ✓ Control signals to logic that add 1 to the contents of the PC and store the result back to the PC
- Following this, the control unit sends a control signal that opens gates between the MBR and the IR.

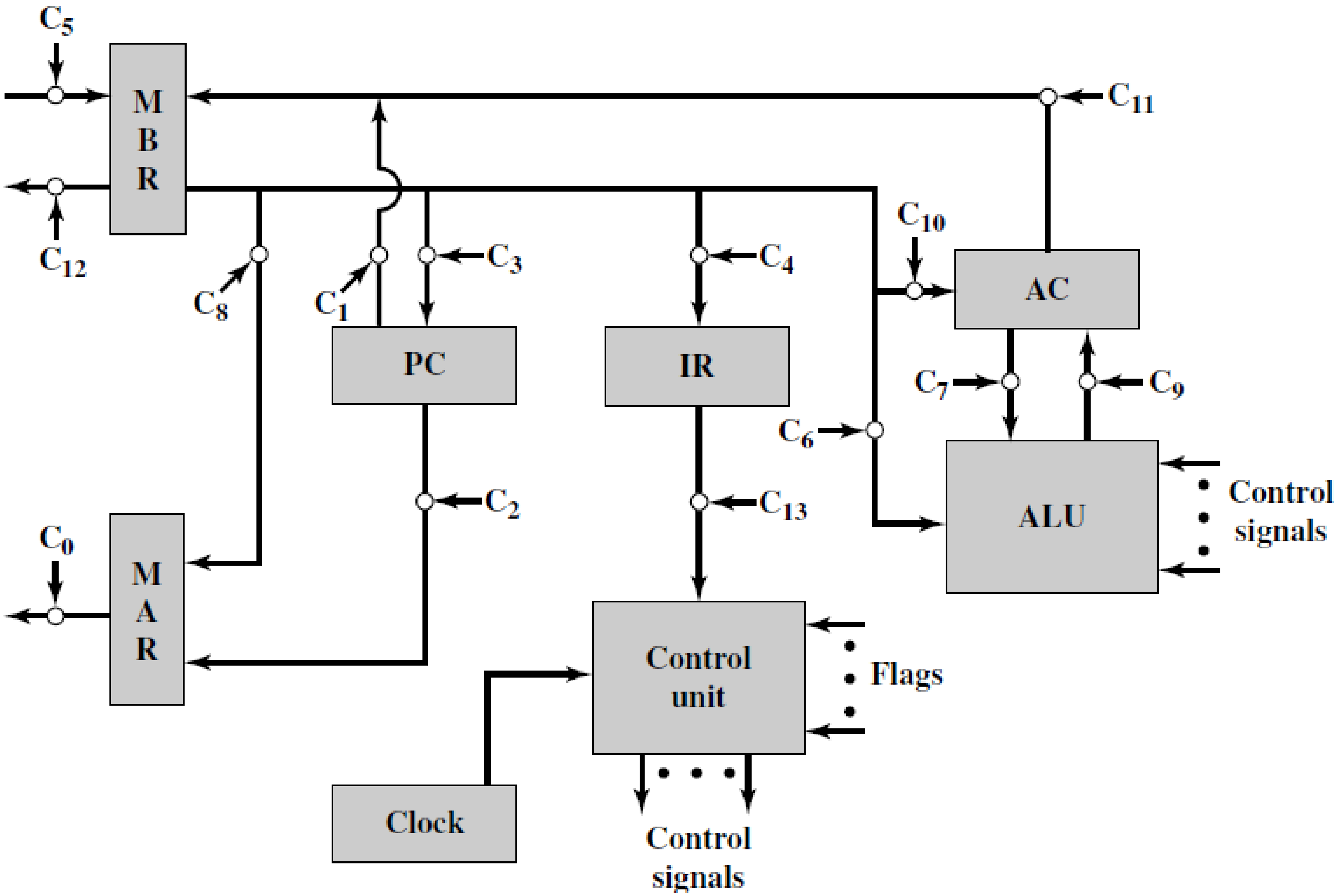
Control Unit

❑ Data Paths and Control Signals

- **Example:** simple processor with a single accumulator
- The control unit receives inputs from the clock, the instruction register, and flags.
- With each clock cycle, the control unit reads all of its inputs and emits a set of control signals.
- Control signals go to three separate destinations:
  - **Data paths:** The control unit controls the internal flow of data.
  - **ALU:** The control unit controls the operation of the ALU by a set of control signals.
  - **System bus:** The control unit sends control signals out onto the control lines of the system bus (e.g., memory READ).
- The control unit must maintain knowledge of where it is in the instruction cycle.
- Using this knowledge, and by reading all of its inputs, the control unit emits a sequence of control signals that causes micro-operations to occur.
- It uses the clock pulses to time the sequence of events, allowing time between events for signal levels to stabilize.

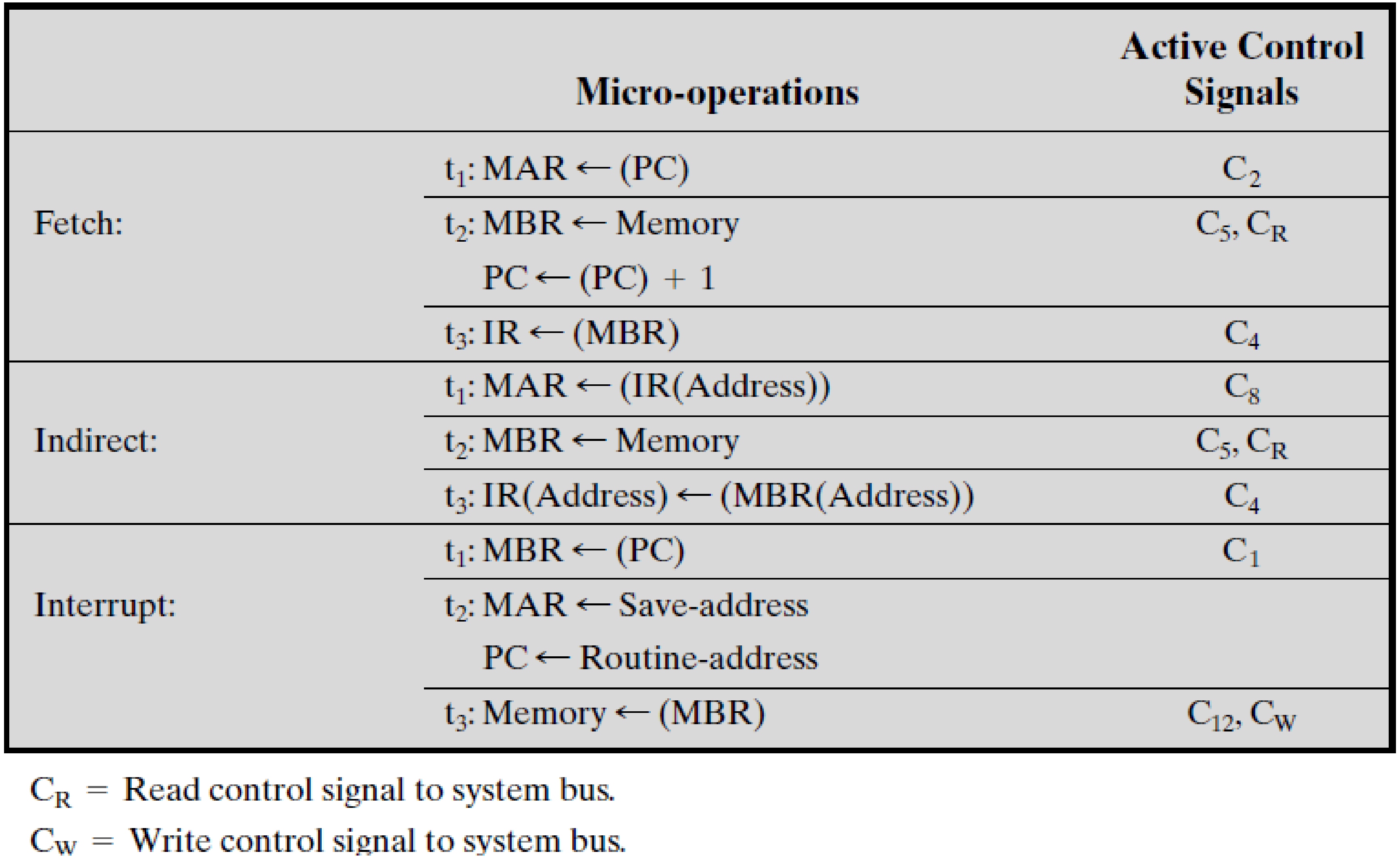
Control Unit

Data Paths and Control Signals



Control Unit

❑ Micro-operations and Control Signals



	Micro-operations	Active Control Signals
Fetch:	t_1: MAR ← (PC)	C_2
	t_2: MBR ← Memory PC ← (PC) + 1	C_5, C_R
	t_3: IR ← (MBR)	C_4
Indirect:	t_1: MAR ← (IR(Address))	C_8
	t_2: MBR ← Memory	C_5, C_R
	t_3: IR(Address) ← (MBR(Address))	C_4
Interrupt:	t_1: MBR ← (PC)	C_1
	t_2: MAR ← Save-address PC ← Routine-address	
	t_3: Memory ← (MBR)	C_{12}, C_W

C_R = Read control signal to system bus.

C_W = Write control signal to system bus.