

# UNIT-2 REGISTER TRANSFER AND MICROOPERATIONS

(REF. CH-5 MORRIS MANO EDITED 3<sup>RD</sup>  
ED.)

Gaurang Patel

# OUTLINE

Digital Systems and Microoperations

Register Transfer Language

Register Transfer

Bus and Memory Transfers

Arithmetic Microoperations

Logic Microoperations

Shift Microoperations

Arithmetic Logic Shift Unit

# DIGITAL SYSTEM

Combinational and sequential circuits can be used to create simple digital systems.

**Digital System:** An interconnection of hardware modules that do a certain task on the information.

Modules are interconnected with common data and control paths to form a digital computer system

# DIGITAL SYSTEM

Simple digital systems are frequently characterized in terms of

- the registers they contain
- the operations that they perform
- the control that initiates the sequence of microoperations

Digital Module = Registers + Microoperations Hardware + Control Functions

# MICROOPERATIONS

The operations executed on the data in registers are called microoperations

Examples of microoperations:

- Shift
- Load
- Clear
- Increment
- Add
- Subtract
- Complement
- .....

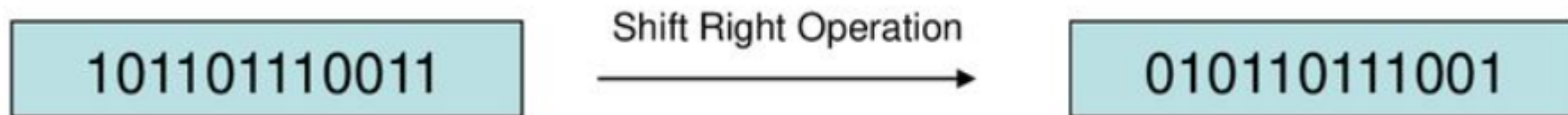
# MICROOPERATIONS

For any function of the computer, a sequence of microoperations is used to describe it

The result of the operation may be:

- replace the previous binary information of a register or transferred to another register

Shift Right Operation



# REGISTER TRANSFER LANGUAGE

Register Transfer Language (RTL) : a symbolic notation to describe the microoperation transfers among registers

**Register transfer**  $\Rightarrow$  the availability of hardware logic circuits that can perform a stated microoperation and transfer the result to the same or another register

**Language**  $\Rightarrow$  programming language

A convenient tool for describing the internal organization of digital computers

Can also be used to facilitate the design process of digital systems.

# REGISTER TRANSFER (OUR FIRST MICROOPERATION)

Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register

R1: processor register

MAR: Memory Address Register (holds an address for a memory unit)

PC: Program Counter

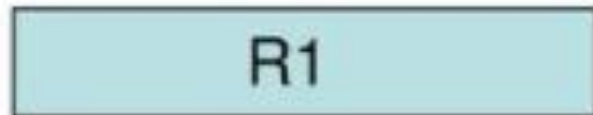
IR: Instruction Register

SR: Status Register

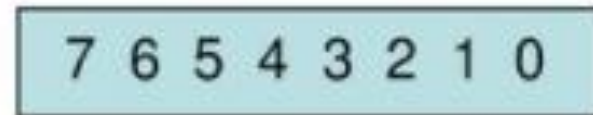


# REGISTER TRANSFER

The individual flip-flops in an n-bit register are numbered in sequence from 0 to n-1 (from the right position toward the left position)



Register R1

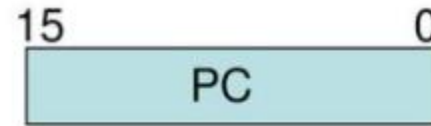


Showing individual bits

A block diagram of a register

# REGISTER TRANSFER

Other ways of drawing the block diagram of a register:



Numbering of bits



Partitioned into two parts

# REGISTER TRANSFER

Information transfer from one register to another is described by a replacement operator:  **$R2 \leftarrow R1$**

This statement denotes a transfer of the content of register R1 into register R2

The transfer happens in one clock cycle

The content of the R1 (source) does not change

The content of the R2 (destination) will be lost and replaced by the new data transferred from R1

We are assuming that the circuits are available from the outputs of the source register to the inputs of the destination register, and that the destination register has a parallel load capability

# REGISTER TRANSFER

Conditional transfer occurs only under a control condition

Representation of a (conditional) transfer

- $P: R2 \leftarrow R1$

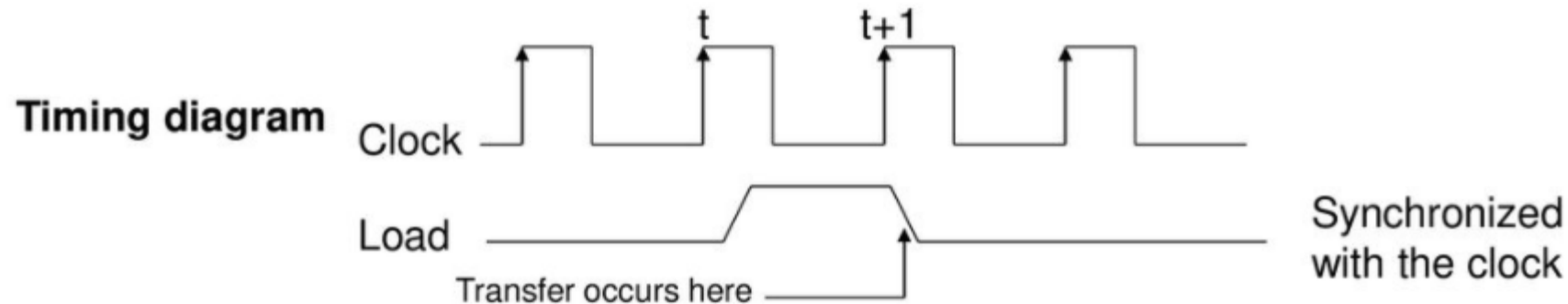
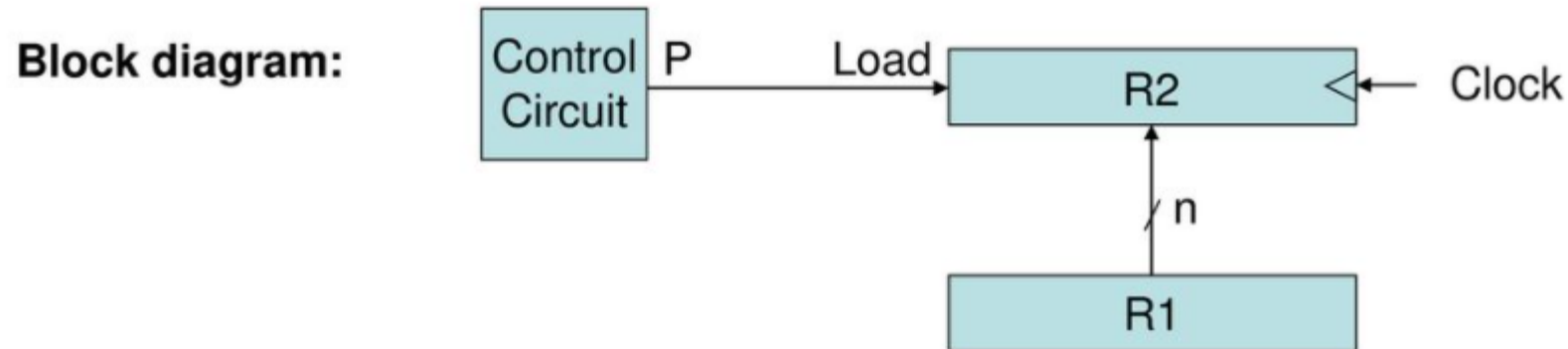
A binary condition ( $P$  equals to 0 or 1) determines when the transfer occurs

The content of  $R1$  is transferred into  $R2$  only if  $P$  is 1

**Note:**  $P$  is called control function (Boolean variable that is equal to 0 or 1)

# REGISTER TRANSFER

Hardware implementation of a controlled transfer: **P: R2 ← R1**



# REGISTER TRANSFER

Basic Symbols for Register Transfers		
Symbol	Description	Examples
Letters & numerals	Denotes a register	MAR, R2
Parenthesis ( )	Denotes a part of a register	R2(0-7), R2(L)
Arrow $\leftarrow$	Denotes transfer of information	R2 $\leftarrow$ R1
Comma ,	Separates two microoperations	R2 $\leftarrow$ R1, R1 $\leftarrow$ R2

# BUS AND MEMORY TRANSFERS

Paths must be provided to transfer information from one register to another

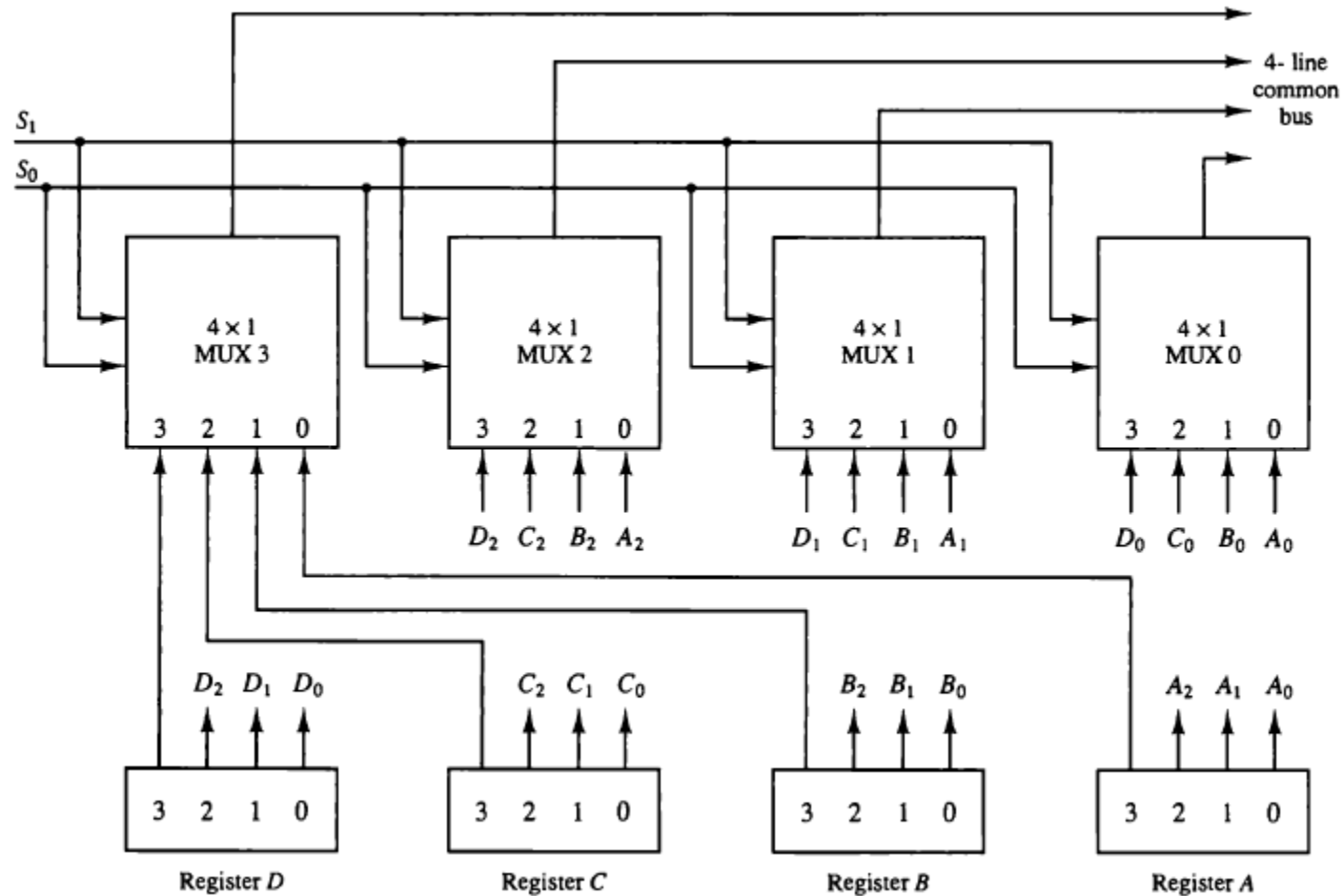
The number of wires will be excessive if separate lines are used between each register and all other registers in the system

A **Common Bus System** is a scheme for transferring information between registers in a multiple-register configuration

**A bus:** set of common lines, one for each bit of a register, through which binary information is transferred one at a time

Control signals determine which register is selected by the bus during each particular register transfer

# BUS AND MEMORY TRANSFERS (BUS SYSTEM FOR 4 REGISTERS)





# BUS AND MEMORY TRANSFERS

The transfer of information from a bus into one of many destination registers is done:

- By connecting the bus lines to the inputs of all destination registers and then:
- activating the load control of the particular destination register selected

We write:  $R2 \leftarrow C$  to symbolize that the content of register  $C$  is loaded into the register  $R2$  using the common system bus

It is equivalent to:  $BUS \leftarrow C, (\text{select } C) R2 \leftarrow BUS (\text{Load } R2)$

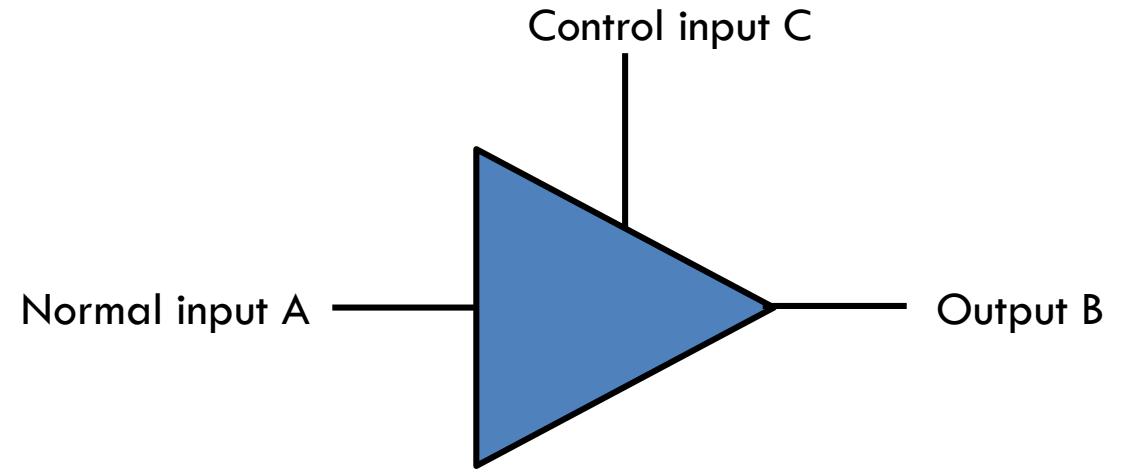
# BUS AND MEMORY TRANSFERS: THREE-STATE BUS BUFFERS

A bus system can be constructed with three-state buffer gates instead of multiplexers

A three-state buffer is a digital circuit that exhibits three states: logic-0, logic-1, and high-impedance (Hi-Z)

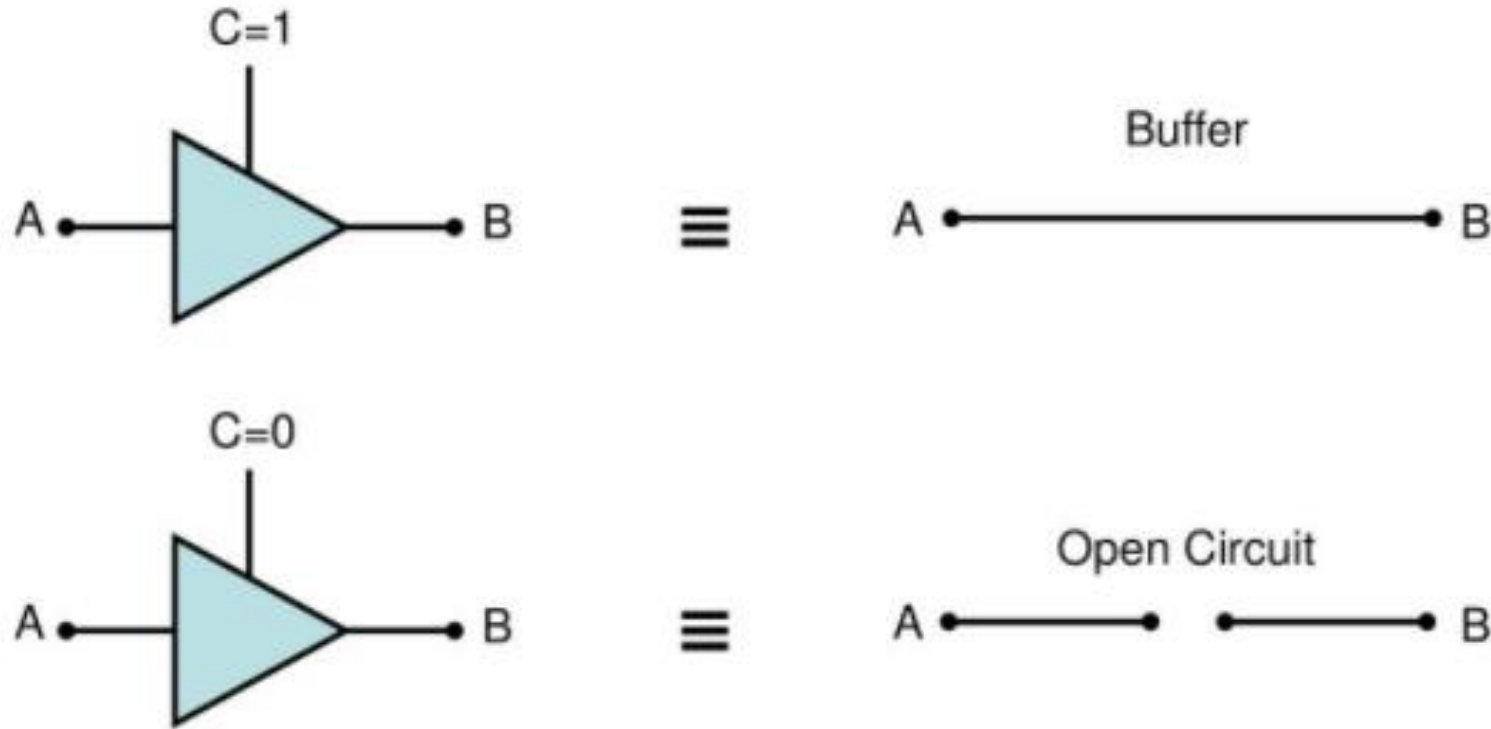
The Hi-Z state behaves like an open circuit

The output is disconnected and does not have a logic significance

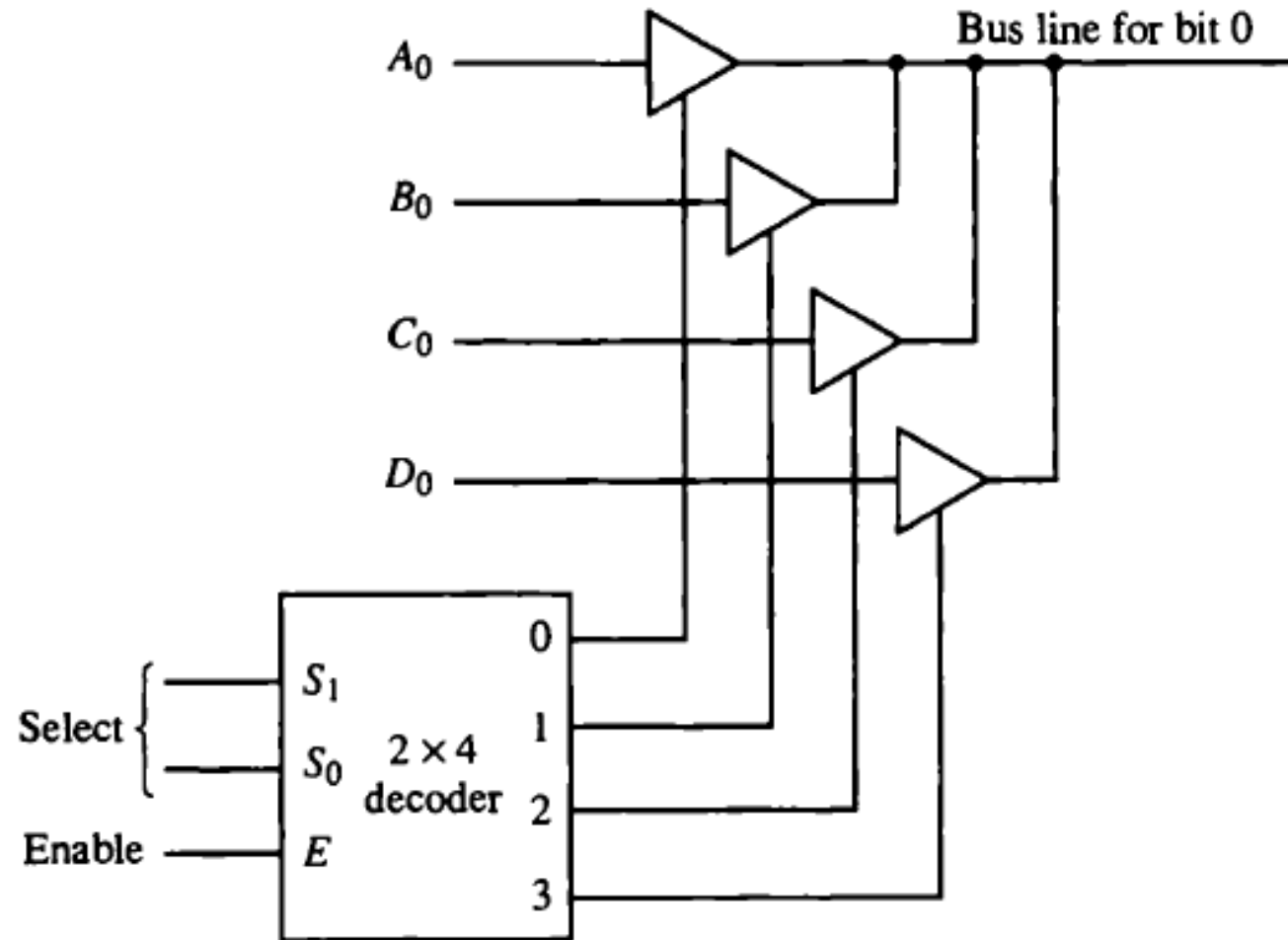


**Three-State Buffer**

# BUS AND MEMORY TRANSFERS: THREE-STATE BUS BUFFERS



# BUS AND MEMORY TRANSFERS: THREE-STATE BUS BUFFERS



# BUS AND MEMORY TRANSFERS: MEMORY TRANSFER

Memory read : Transfer from memory

Memory write : Transfer to memory

Data being read or wrote is called a memory word (called  $M$ ) It is necessary to specify the address of  $M$  when writing /reading memory

This is done by enclosing the address in square brackets following the letter  $M$

Example:  $M[0016]$  : the memory contents at address  $0x0016$

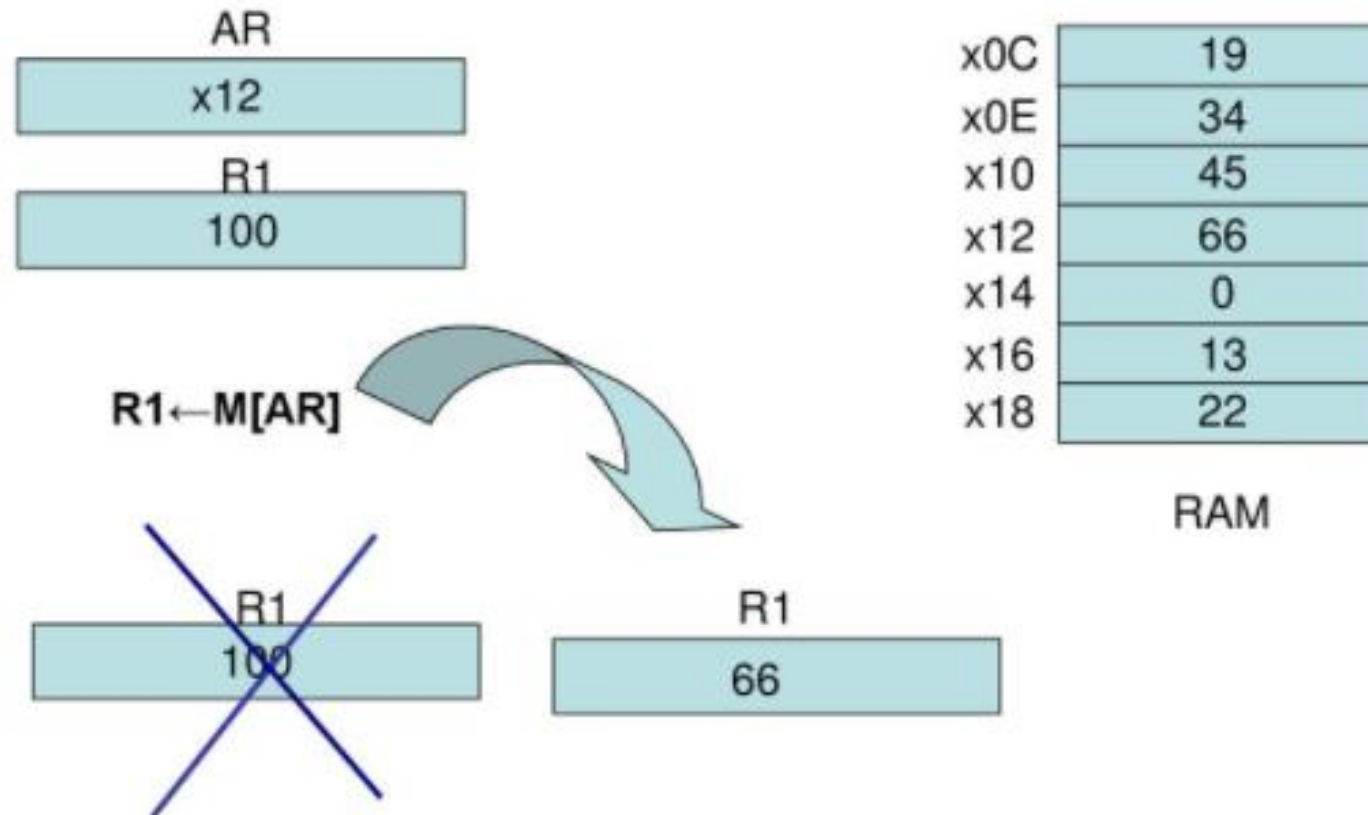
# BUS AND MEMORY TRANSFERS: MEMORY TRANSFER

Assume that the address of a memory unit is stored in a register called the Address Register AR

Lets represent a Data Register with DR, then:

- Read:  $DR \leftarrow M[AR]$
- Write:  $M[AR] \leftarrow DR$

# BUS AND MEMORY TRANSFERS: MEMORY TRANSFER



# ARITHMETIC MICROOPERATIONS

The microoperations most often encountered in digital computers are classified into four categories:

- Register transfer microoperations
- Arithmetic microoperations (on numeric data stored in the registers)
- Logic microoperations (bit manipulations on non-numeric data)
- Shift microoperations



# ARITHMETIC MICROOPERATIONS

The basic arithmetic microoperations are: addition, subtraction, increment and decrement

Addition Microoperation:

$$\mathbf{R3 \leftarrow R1 + R2}$$

Subtraction Microoperation:

$$\mathbf{R3 \leftarrow R1 - R2 \text{ or :}}$$

$$\mathbf{R3 \leftarrow R1 + R2' + 1 \text{ (with 2's complement)}}$$

# ARITHMETIC MICROOPERATIONS

One's Complement Microoperation:

$$\mathbf{R2 \leftarrow R2'}$$

Two's Complement Microoperation:

$$\mathbf{R2 \leftarrow R2' + 1}$$

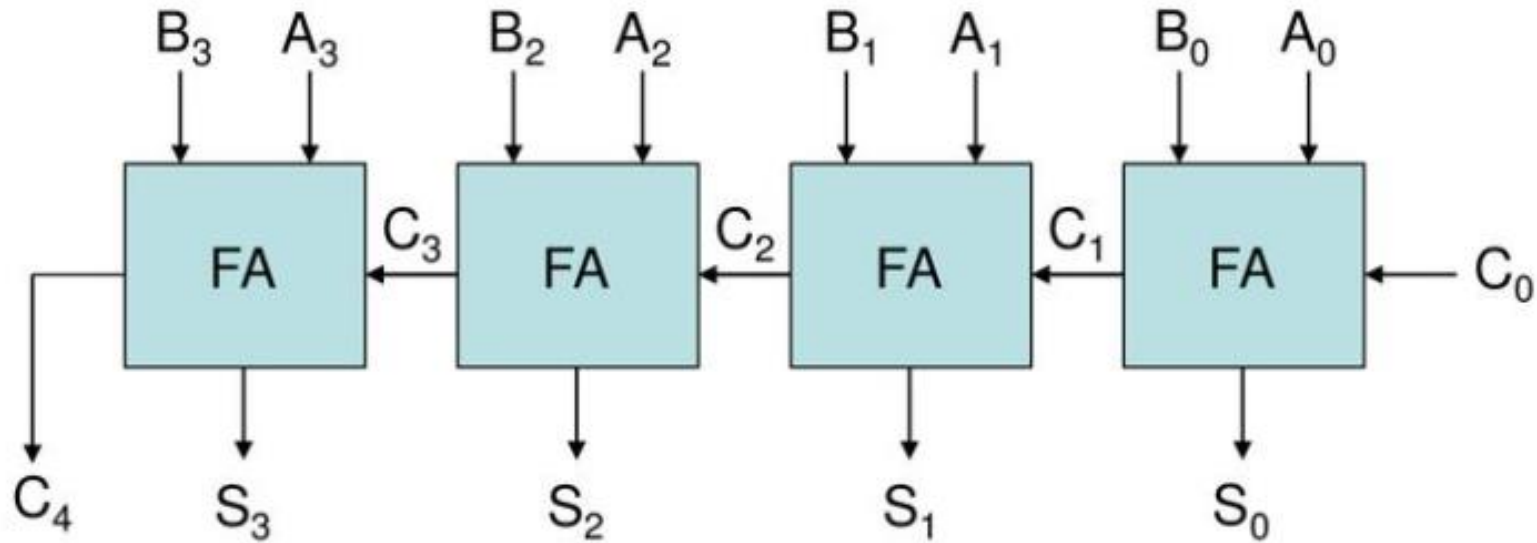
Increment Microoperation:

$$\mathbf{R2 \leftarrow R2 + 1}$$

Decrement Microoperation:

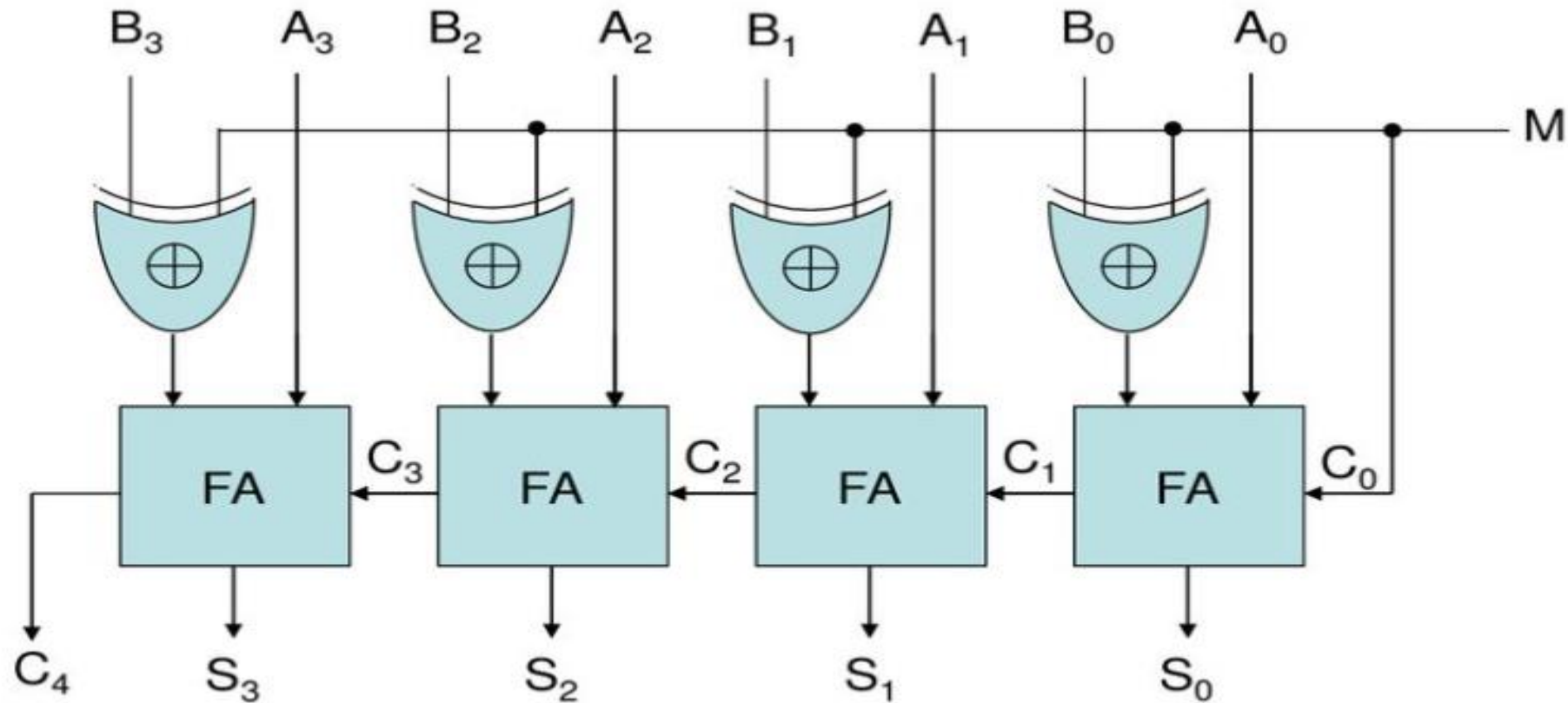
$$\mathbf{R2 \leftarrow R2 - 1}$$

# BUS AND MEMORY TRANSFERS: BINARY ADDER



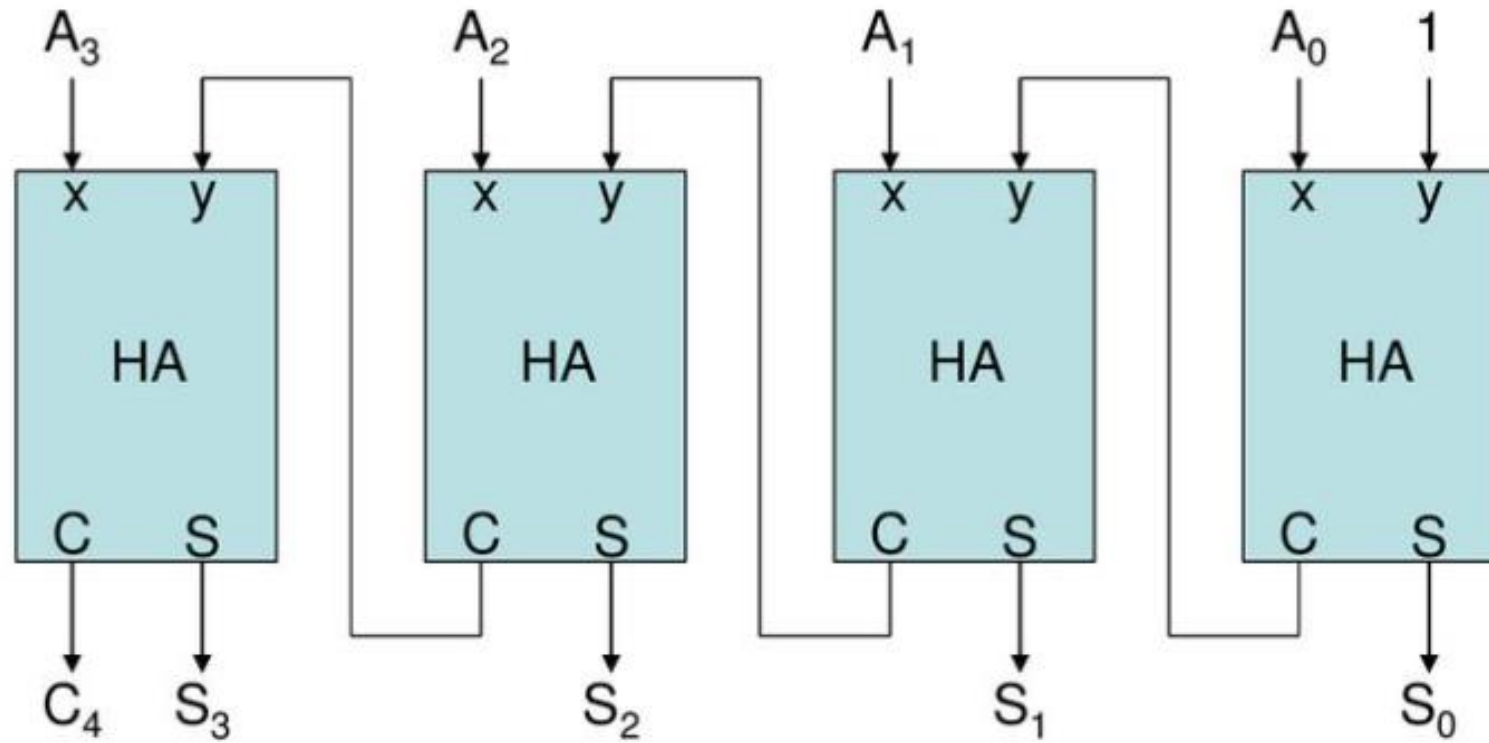
**4-bit binary adder  
(connection of FAs)**

# BUS AND MEMORY TRANSFERS: BINARY ADDER-SUBTRACTOR



**4-bit adder-subtractor**

# BUS AND MEMORY TRANSFERS: BINARY INCREMENTER



**4-bit Binary Incrementer**

# BUS AND MEMORY TRANSFERS: BINARY INCREMENTER

Binary Incrementer can also be implemented using a counter

A binary decrementer can be implemented by adding 1111 to the desired register each time!

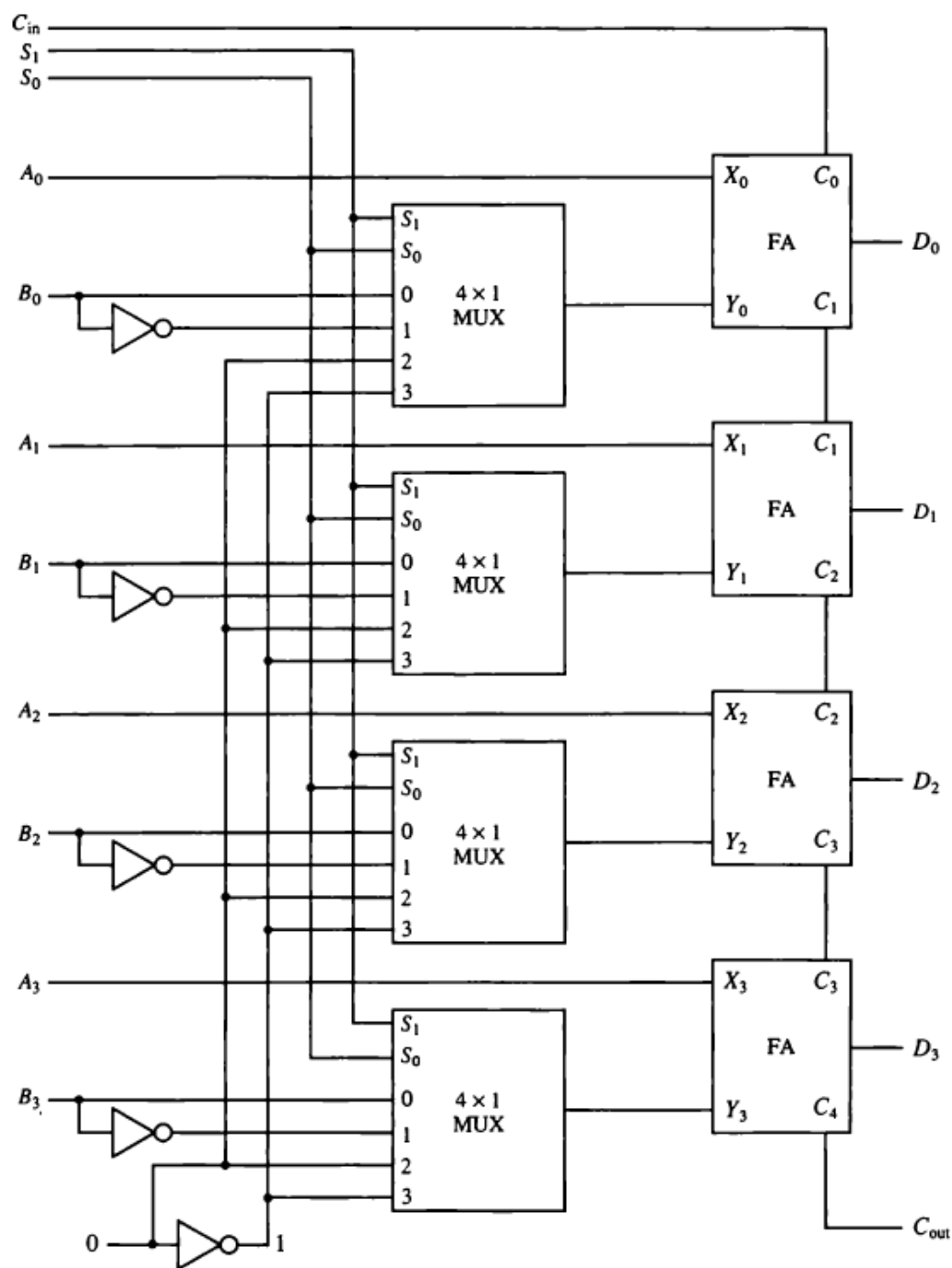


Figure 4-9 4-bit arithmetic circuit.

TABLE 4-4 Arithmetic Circuit Function Table

Select			Input $Y$	Output $D = A + Y + C_{in}$	Microoperation
$S_1$	$S_0$	$C_{in}$			
0	0	0	$B$	$D = A + B$	Add
0	0	1	$B$	$D = A + B + 1$	Add with carry
0	1	0	$\overline{B}$	$D = A + \overline{B}$	Subtract with borrow
0	1	1	$\overline{B}$	$D = A + \overline{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer $A$
1	0	1	0	$D = A + 1$	Increment $A$
1	1	0	1	$D = A - 1$	Decrement $A$
1	1	1	1	$D = A$	Transfer $A$

# LOGIC MICROOPERATIONS: THE FOUR BASIC MICROOPERATIONS

## OR Microoperation

- Symbol:  $\vee$ , +

- Gate: 

- Example:  $100110_2 \vee 1010110_2 = 1110110_2$

P+Q:  $R1 \leftarrow R2 + R3, R4 \leftarrow R5 \vee R6$


Diagram illustrating the microoperation P+Q:  $R1 \leftarrow R2 + R3, R4 \leftarrow R5 \vee R6$ . The expression is annotated with callouts: "OR" above the  $\vee$  operator, "ADD" below the  $+$  operator, and another "OR" above the  $\vee$  operator.



# LOGIC MICROOPERATIONS: THE FOUR BASIC MICROOPERATIONS

## AND Microoperation

- Symbol:  $\wedge$

- Gate: 

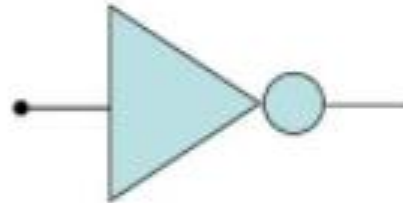
- Example:  $100110_2 \wedge 1010110_2 = 0000110_2$

# LOGIC MICROOPERATIONS: THE FOUR BASIC MICROOPERATIONS

## Complement (NOT) Microoperation

- Symbol:  $\overline{\phantom{x}}$

- Gate:



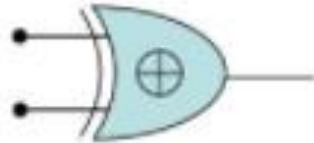
- Example:  $\overline{1010110_2} = 0101001_2$

# LOGIC MICROOPERATIONS: THE FOUR BASIC MICROOPERATIONS

## XOR (Exclusive-OR) Microoperation

- Symbol:  $\oplus$

- Gate:

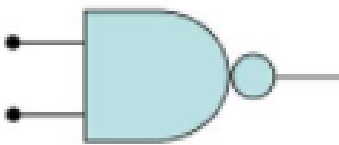


- Example:  $100110_2 \oplus 1010110_2 = 1110000_2$

# LOGIC MICROOPERATIONS: OTHER LOGIC MICROOPERATIONS

## NAND Microoperation

- Symbols:  $\wedge$  and  $\overline{\phantom{x}}$

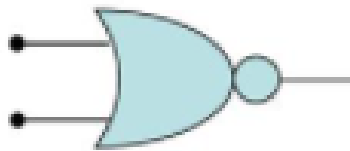
- Gate: 

- Example:  $100110_2 \wedge 1010110_2 = 1111001_2$

# LOGIC MICROOPERATIONS: OTHER LOGIC MICROOPERATIONS

## NOR Microoperation

- Symbols:  $\vee$  and  $\bar{\phantom{x}}$

- Gate: 

- Example:  $\overline{100110_2 \vee 1010110_2} = 0001001_2$

# LOGIC MICROOPERATIONS: OTHER LOGIC MICROOPERATIONS

## **Set (Preset) Microoperation**

Force all bits into 1's by ORing them with a value in which all its bits are being assigned to logic-1

Example:  $100110_2 \vee 111111_2 = 111111_2$

## **Clear (Reset) Microoperation**

Force all bits into 0's by ANDing them with a value in which all its bits are being assigned to logic-0

Example:  $100110_2 \wedge 000000_2 = 000000_2$

**TABLE 4-6** Sixteen Logic Microoperations

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer $A$
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer $B$
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement $B$
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement $A$
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow \text{all 1's}$	Set to all 1's

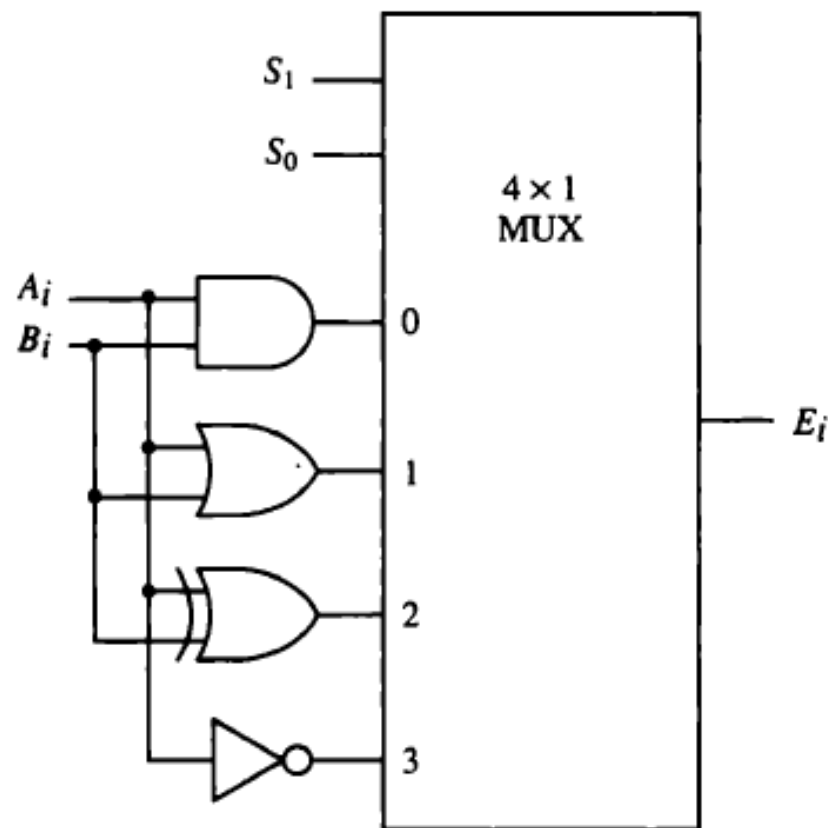
# LOGIC MICROOPERATIONS: HARDWARE IMPLEMENTATION

The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function

Most computers use only four (AND, OR, XOR, and NOT) from which all others can be derived.



Figure 4-10 One stage of logic circuit.



(a) Logic diagram

$S_1$	$S_0$	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \bar{A}$	Complement

(b) Function table

# SHIFT MICROOPERATIONS

Used for serial transfer of data

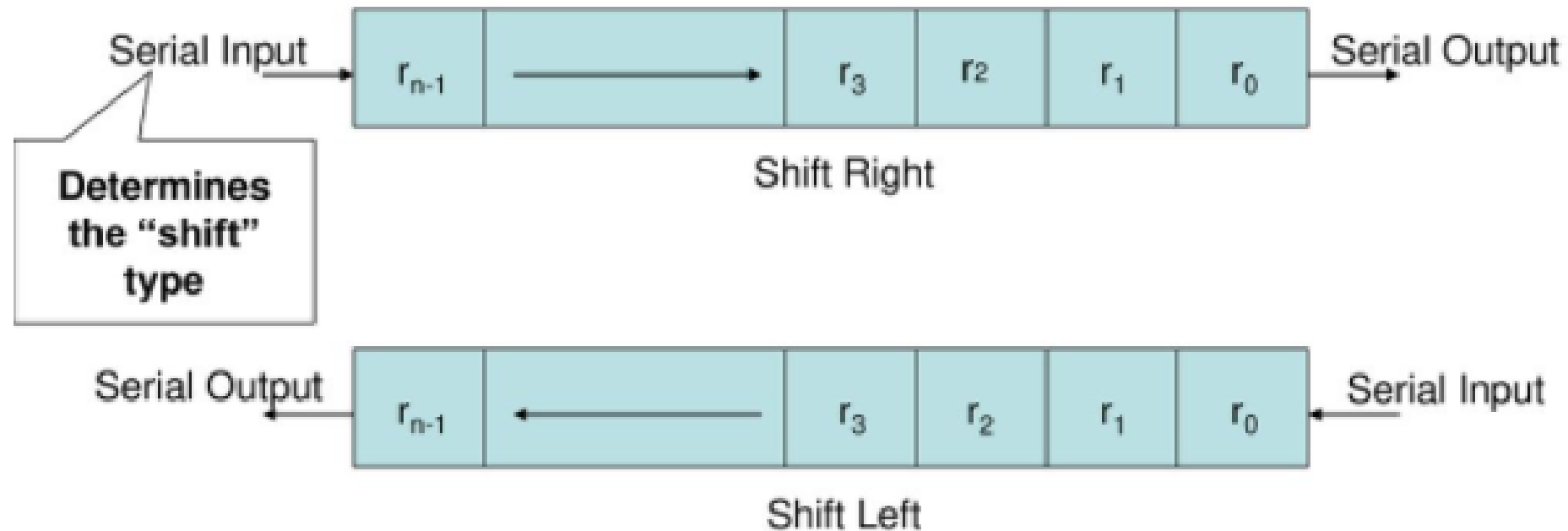
Also used in conjunction with arithmetic, logic, and other data-processing operations

The contents of the register can be shifted to the left or to the right

As being shifted, the first flip-flop receives its binary information from the serial input

Three types of shift: Logical, Circular, and Arithmetic

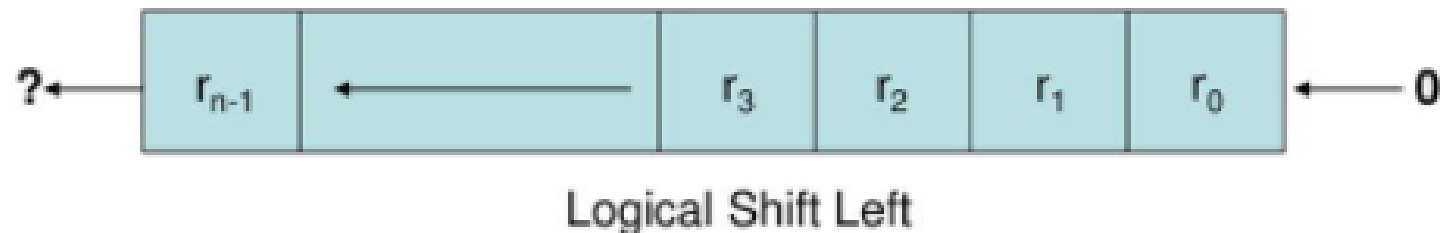
# SHIFT MICROOPERATIONS



\*\*Note that the bit  $r_i$  is the bit at position (i) of the register

# SHIFT MICROOPERATIONS: LOGICAL SHIFTS

- Transfers 0 through the serial input
- Logical Shift Right:  $R1 \leftarrow \text{shr } R1$   
The same
- Logical Shift Left:  $R2 \leftarrow \text{shl } R2$   
The same



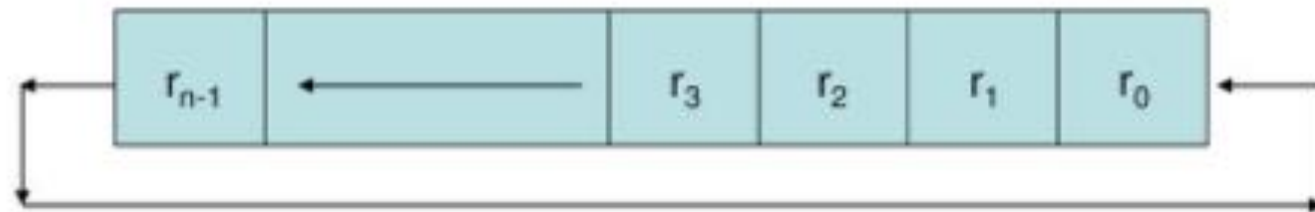
# SHIFT MICROOPERATIONS: CIRCULAR SHIFTS (ROTATE OPERATION)

- Circulates the bits of the register around the two ends without loss of information
- Circular Shift Right:  $R1 \leftarrow \text{cir } R1$

The same

- Circular Shift Left:  $R2 \leftarrow \text{cil } R2$

The same



Circular Shift Left

# SHIFT MICROOPERATIONS: ARITHMETIC SHIFTS

Shifts a signed binary number to the left or right

An arithmetic shift-left multiplies a signed binary number by 2:

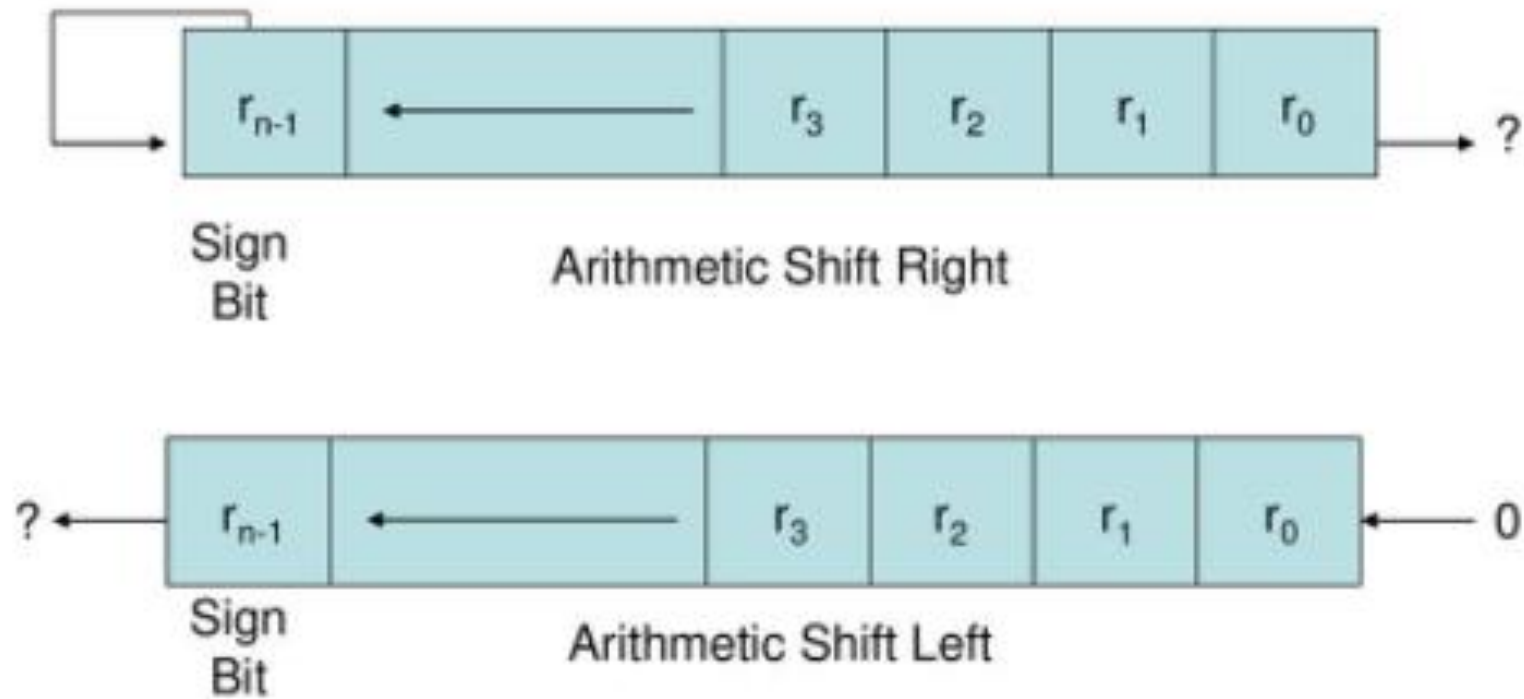
- `ashl (00100):`

An arithmetic shift-right divides the number by 2:

- `ashr (00100) : 00010`

An overflow may occur in arithmetic shift-left, and occurs when the sign bit is changed (sign reversal)

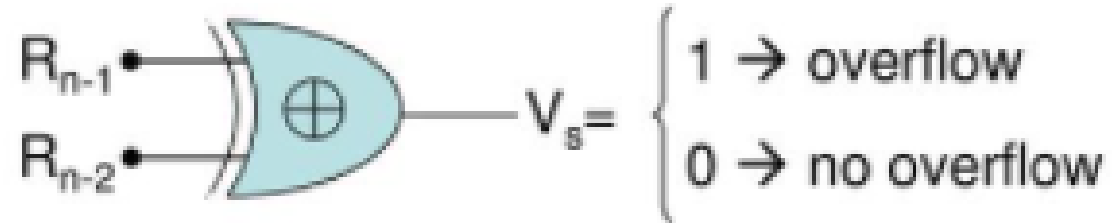
# SHIFT MICROOPERATIONS: ARITHMETIC SHIFTS



# SHIFT MICROOPERATIONS: ARITHMETIC SHIFTS

An overflow flip-flop  $V_s$  can be used to detect an arithmetic shift-left overflow

$$V_s = R_{n-1} \oplus R_{n-2}$$





# SHIFT MICROOPERATIONS

Example: Assume  $R1 = 11001110$ , then:

- Arithmetic shift right once :  $R1 = 11100111$
- Arithmetic shift right twice :  $R1 = 11110011$
- Arithmetic shift left once :  $R1 = 10011100$
- Arithmetic shift left twice :  $R1 = 00111000$
- Logical shift right once :  $R1 = 01100111$
- Logical shift left once :  $R1 = 10011100$
- Circular shift right once :  $R1 = 01100111$
- Circular shift left once :  $R1 = 10011101$

# SHIFT MICROOPERATIONS: HARDWARE IMPLEMENTATION

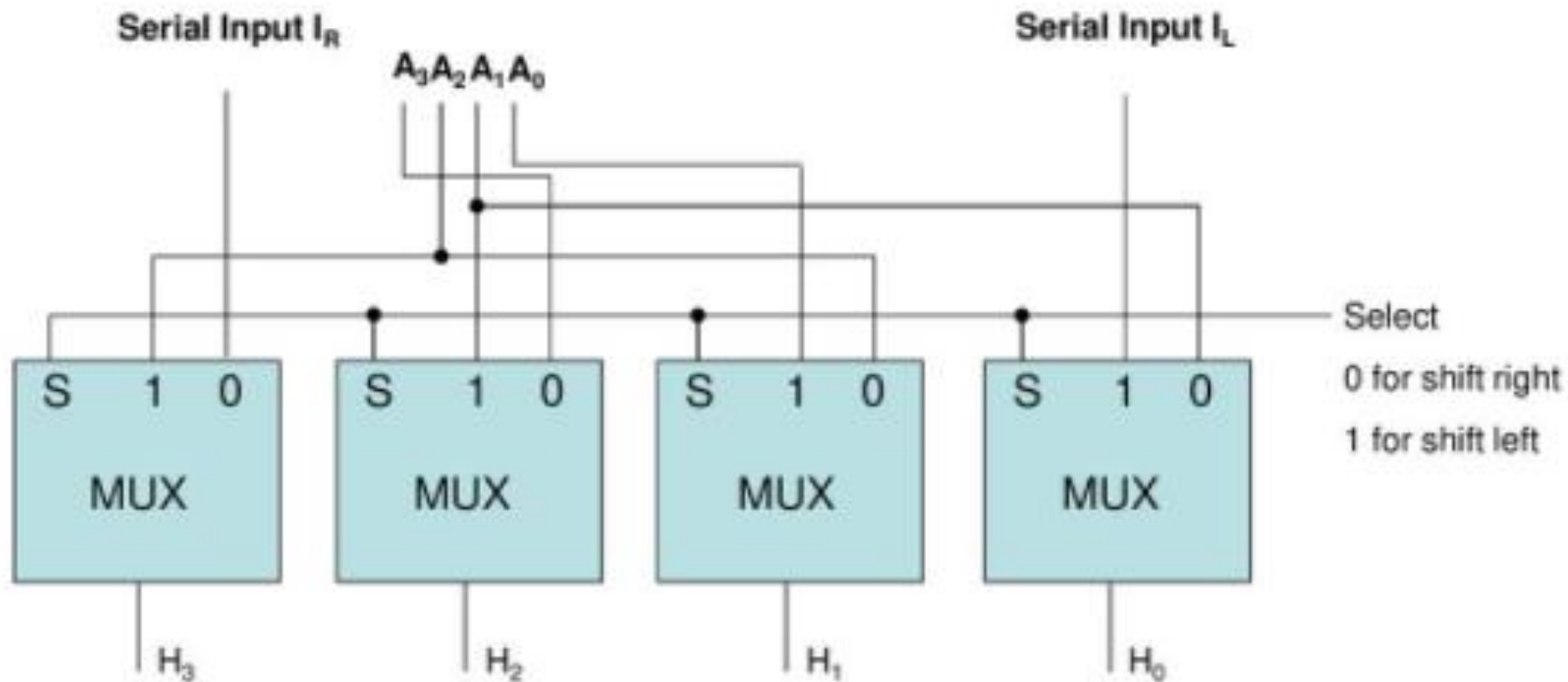
A possible choice for a shift unit would be a bidirectional shift register with parallel load (refer to Fig 2-9). Has drawbacks:

Needs two pulses (the clock and the shift signal pulse)

Not efficient in a processor unit where multiple number of registers share a common bus

It is more efficient to implement the shift operation with a combinational circuit

# SHIFT MICROOPERATIONS: HARDWARE IMPLEMENTATION



4-bit Combinational Circuit Shifter

Function table

Select	Output			
	$H_0$	$H_1$	$H_2$	$H_3$
0	$I_R$	$A_0$	$A_1$	$A_2$
1	$A_1$	$A_2$	$A_3$	$I_L$

# ARITHMETIC LOGIC SHIFT UNIT

Instead of having individual registers performing the microoperations directly, computer systems employ a number of storage registers connected to a common operational unit called an Arithmetic Logic Unit (ALU)

# ARITHMETIC LOGIC SHIFT UNIT

Fig.A-> slide no. 31

Fig.B-> slide no. 41

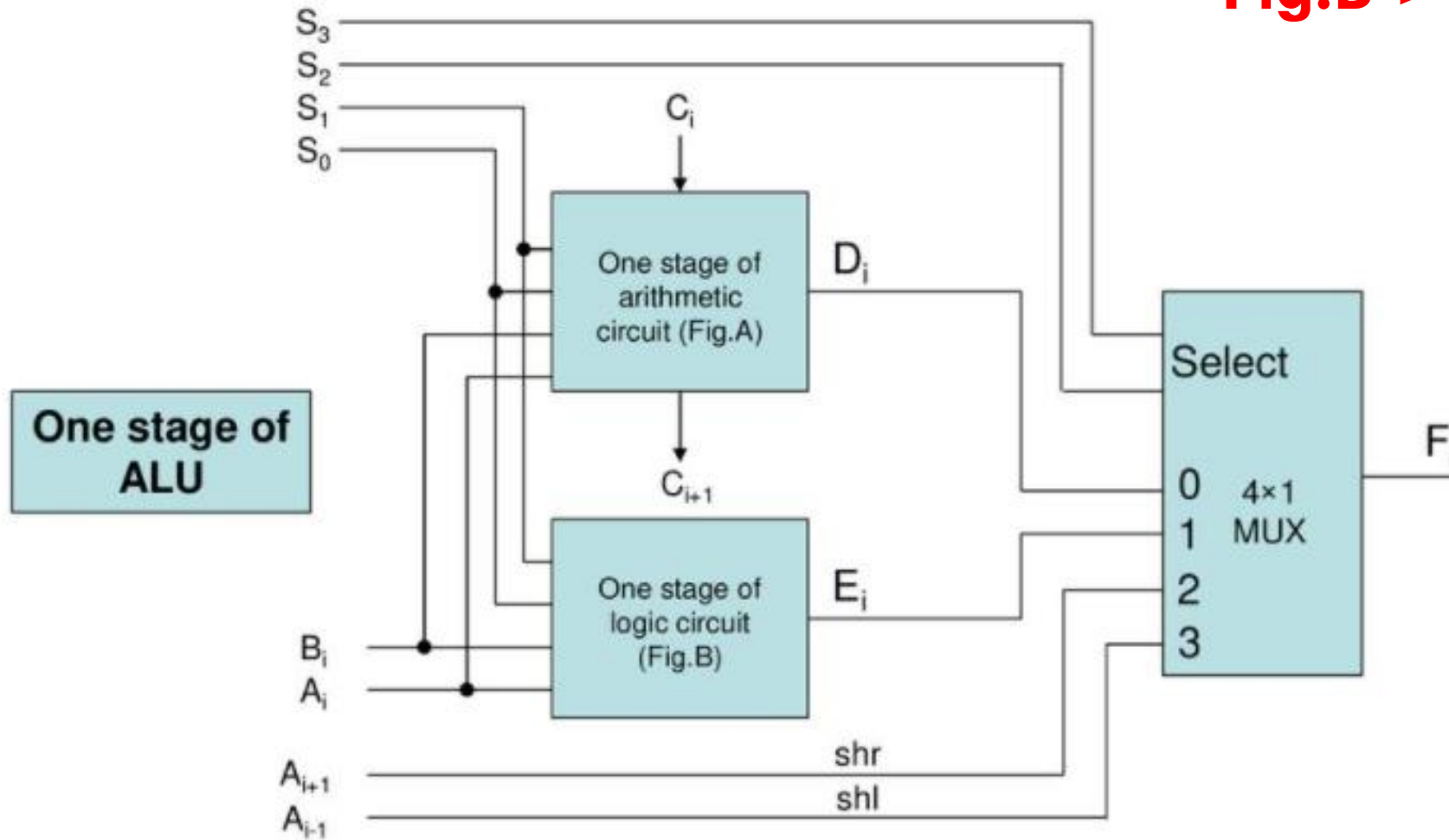


TABLE 4-8 Function Table for Arithmetic Logic Shift Unit

Operation select					Operation	Function
$S_3$	$S_2$	$S_1$	$S_0$	$C_{in}$		
0	0	0	0	0	$F = A$	Transfer $A$
0	0	0	0	1	$F = A + 1$	Increment $A$
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \bar{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \bar{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement $A$
0	0	1	1	1	$F = A$	Transfer $A$
0	1	0	0	$\times$	$F = A \wedge B$	AND
0	1	0	1	$\times$	$F = A \vee B$	OR
0	1	1	0	$\times$	$F = A \oplus B$	XOR
0	1	1	1	$\times$	$F = \bar{A}$	Complement $A$
1	0	$\times$	$\times$	$\times$	$F = \text{shr } A$	Shift right $A$ into $F$
1	1	$\times$	$\times$	$\times$	$F = \text{shl } A$	Shift left $A$ into $F$

**Thank You**