UNIT - 2 : REGISTER TRANSFER AND MICRO-OPERATIONS

Basic Definitions:

¬ A digital system is an interconnection of digital hardware modules.

¬ The modules are registers, decoders, arithmetic elements, and control logic.

¬ The various modules are interconnected with common data and control paths to form a digital computer system.

¬ Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them.

¬ The operations executed on data stored in registers are called **Microoperations.**

¬ A microoperation is an elementary operation performed on the information stored in one or more registers.

¬ The result of the operation may replace the previous binary information of a register or may be transferred to another register.

¬ Examples of microoperations are shift, count, clear, and load.

¬ The internal hardware organization of a digital computer is best defined by specifying:

1. The set of registers it contains and their function.
2. The sequence of microoperations performed on the binary information stored in the registers.
3. The control that initiates the sequence of microoperations.

Registers in Computer Architecture:

Register is a very fast computer memory, used to store data/instruction in-execution.

A **Register** is a group of flip-flops with each flip-flop capable of storing **one bit** of information. An *n-bit* register has a group of *n flip-flops* and is capable of storing binary information of *n-bits*.

A register consists of a group of flip-flops and gates. The flip-flops hold the binary information and gates control when and how new information is transferred into a register. Various types of registers are available commercially. The simplest register is one that consists of only flip-flops with no external gates.

These days registers are also implemented as a register file.

Loading the Registers:

The transfer of new information into a register is referred to as loading the register. If all the bits of register are loaded simultaneously with a common clock pulse than the loading is said to be done in parallel.

Register Transfer Language (RTL):

The symbolic notation used to describe the micro-operation transfers amongst registers is called **Register transfer language**.

The term **register transfer** means the availability of **hardware logic circuits** that can perform a stated micro-operation and transfer the result of the operation to the same or another register.

The word **language** is borrowed from programmers who apply this term to programming languages. This programming language is a procedure for writing symbols to specify a given computational process.

Following are some commonly used registers:

1. **Accumulator**: This is the most common register, used to store data taken out from the memory.
2. **General Purpose Registers**: This is used to store data intermediate results during program execution. It can be accessed via assembly programming.
3. **Special Purpose Registers**: Users do not access these registers. These registers are for Computer system,
    - **MAR:** Memory Address Register are those registers that holds the address for memory unit.
    - **MBR:** Memory Buffer Register stores instruction and data received from the memory and sent from the memory.
    - **PC:** Program Counter points to the next instruction to be executed.
    - **IR:** Instruction Register holds the instruction to be executed.

Register Transfer:

Information transferred from one register to another is designated in symbolic form by means of replacement operator. ← acts as a **replacement operator**.

R2 ← R1

It denotes the transfer of the data from register R1 into R2.

Normally we want the transfer to occur only in predetermined control condition. This can be shown by following **if-then** statement: if (P=1) then (R2 ← R1)

Here P is a control signal generated in the control section.

Control Function:

A control function is a Boolean variable that is equal to 1 or 0. The control function is shown as:

P: R2 ← R1

The control condition is terminated with a colon. It shows that transfer operation can be executed only if P=1.

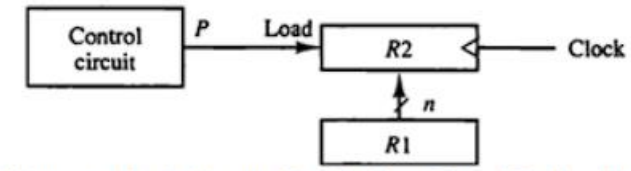


Figure (a) Block diagram Transfer from R1 to R2 when P = 1.

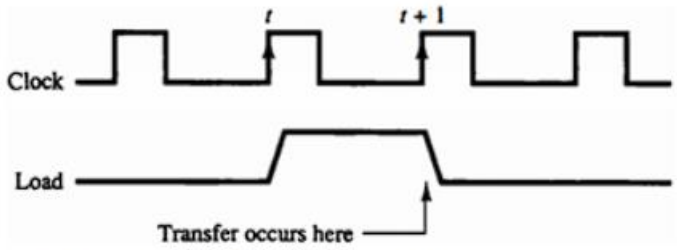
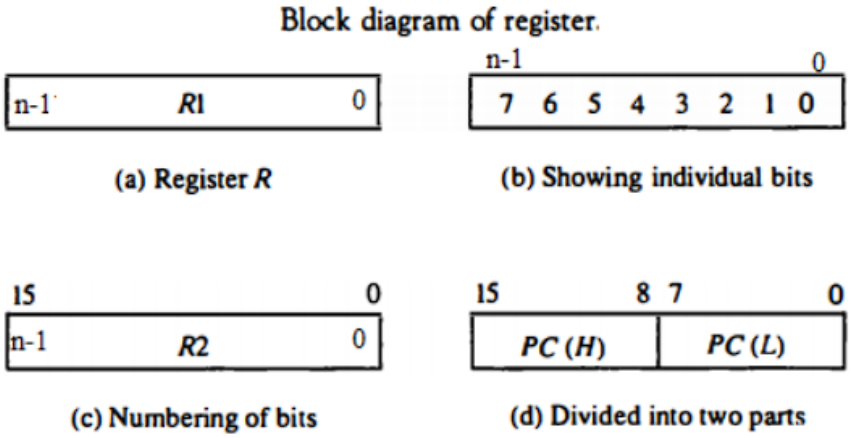


Figure b. Clock and Control Signal

Ways of representing a Register –

1. General way of representing a register is by the name of the register enclosed in a rectangular box as shown in (a).
2. Register is numbered in a sequence of 0 to (n-1) as shown in (b).
3. The numbering of bits in a register can be marked on the top of the box as shown in (c).
4. A 16-bit register PC is divided into 2 parts- Bits (0 to 7) are assigned with lower byte of 16-bit address and bits (8 to 15) are assigned with higher bytes of 16-bit address as shown in (d).



Basic symbols of RTL:

Symbol	Description	Example
Letters and Numbers	Denotes a Register	MAR, R1, R2
( )	Denotes a part of register	R1(8-bit) R1(0-7)
←	Denotes a transfer of information	R2 ← R1
,	Specify two micro-operations of Register Transfer	R1 ← R2 R2 ← R1
:	Denotes conditional operations	P : R2 ← R1 if P=1
Naming Operator (:=)	Denotes another name for an already existing register/alias	Ra := R1
→	If-then	A ∨ B → R1 ← R2
;	Sequential separator	
∧ ∨ ' XOR	Logical operators	

Register Transfer Operations:

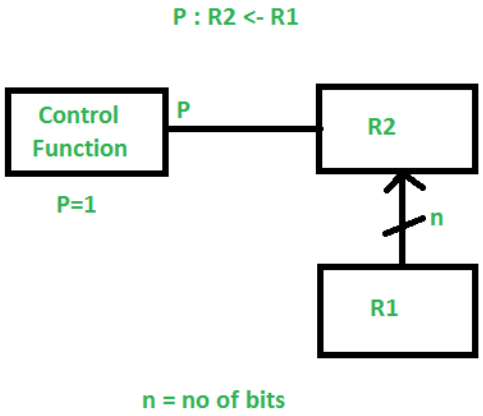
The operation performed on the data stored in the registers are referred to as register transfer operations.

There are different types of register transfer operations:

1. Simple Transfer – R2 ← R1

The content of R1 are copied into R2 without affecting the content of R1. It is an unconditional type of transfer operation.

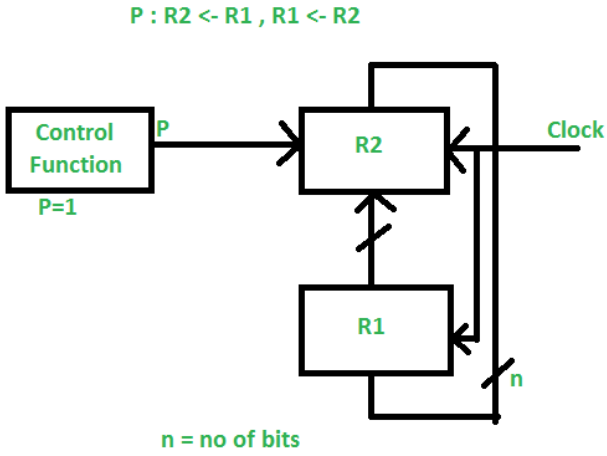
2. Conditional Transfer –



It indicates that if P=1, then the content of R1 is transferred to R2. It is a unidirectional operation.

3. Simultaneous Operations –

If 2 or more operations are to occur simultaneously then they are separated with comma **(,)**.



If the control function P=1, then load the content of R1 into R2 and at the same clock load the content of R2 into R1.

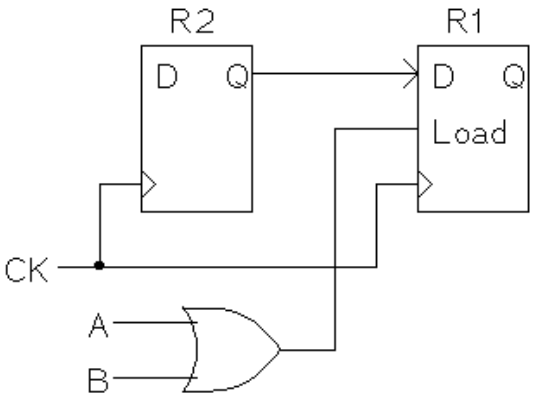
RTL Examples:

A typical RTL statement will look like the following:

$$A \vee B \rightarrow R1 \leftarrow R2;$$

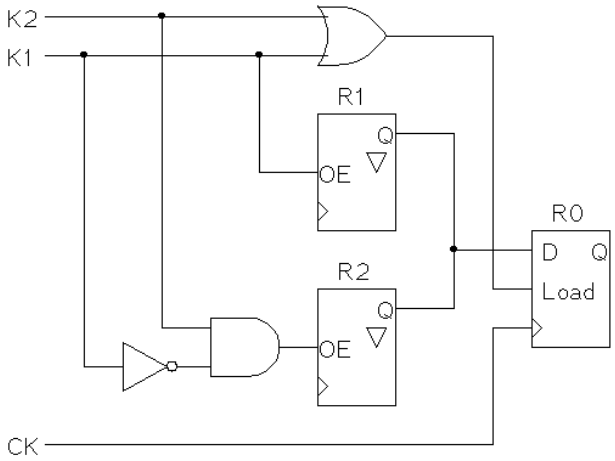
This is read as "***if signal A or signal B is true then register R2 is transferred to register R1***". The first part, A ∨ B, is a logical expression that must be true for the transfer to take place. The → symbol separates the logical expression from the microinstruction. It is the if-then part of the statement. If there isn't a logical expression, → isn't in the statement and the microinstruction will always take place. To the right of → is the microinstruction. It describes a transfer of data and operations on the data from register to register.

The above RTL statement is equivalent to the following schematic:



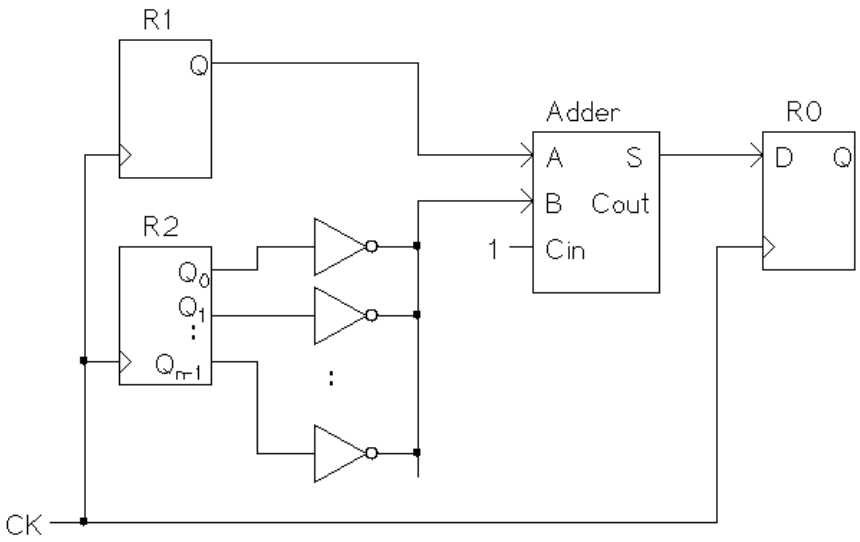
Example 1:

K1 → R0 ← R1 : K1' ∧ K2 → R0 ← R2 ;



Example 2:

R0 ← R1 + R2' + 1



The above circuit is equivalent to **R0 ← R1 - R2**

Bus and Memory Transfers:

Bus Transfer:

- In a digital system of registers, a path must be provided to move information.
- Suppose separate lines are used between each register and all other registers in the system. In that case, the number of wires connecting all of the registers will be excessive because we are connecting each register with another register.
- A bus structure will not require an excessive connection. Thus it is very useful in transferring information.
- A bus is made up of a collection of common lines, one for each bit of a register, that are used to transfer binary data one by one.

There are two methods in bus transfer:

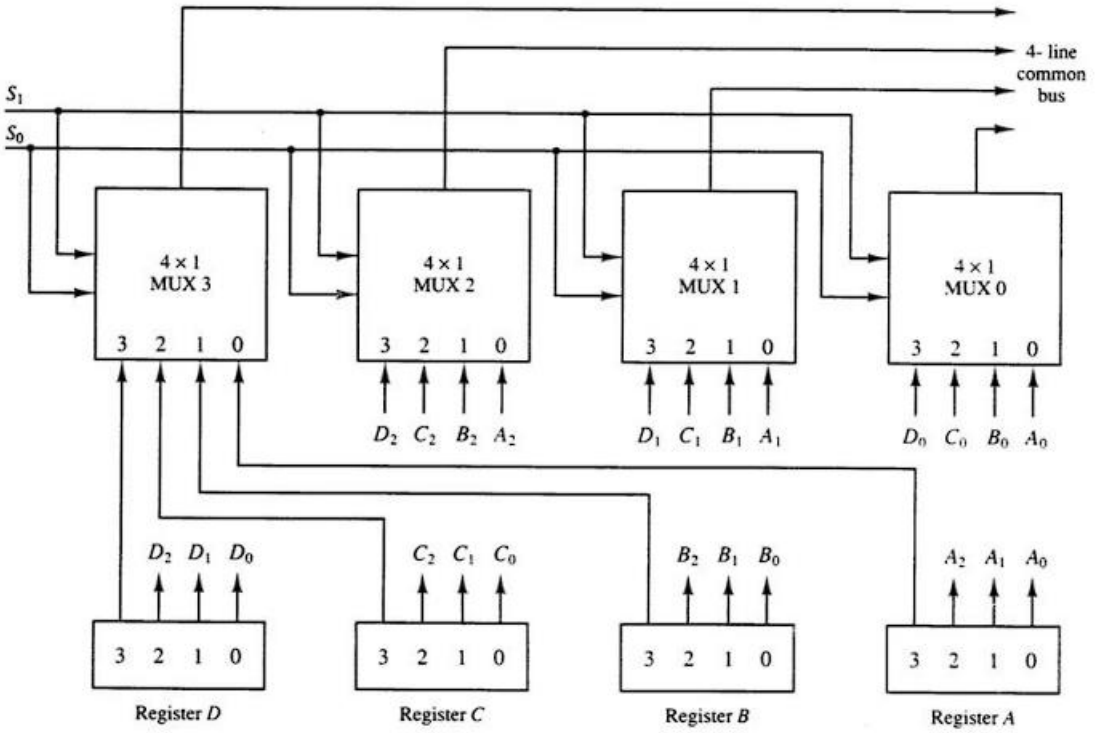
- Bus transfer using Multiplexer
- Bus transfer using Three states bus buffer

Bus Transfer Using Multiplexer:

The following block diagram shows a Bus system for four registers. It is constructed with the help of four 4 * 1 Multiplexers each having four data inputs (0 through 3) and two selection inputs (S0 and S1).

We have used labels to make it more convenient for you to understand the input-output configuration of a Bus system for four registers. For instance, output 1 of register A is connected to input 0 of MUX1.

Figure 4-3 Bus system for four registers.



The two selection lines S1 and S0 are connected to the selection inputs of all four multiplexers. The selection lines choose the four bits of one register and transfer them into the four-line common bus.

When both of the select lines are at low logic, i.e. S1S0 = 00, the 0 data inputs of all four multiplexers are selected and applied to the outputs that forms the bus. This, in turn, causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0 data inputs of the multiplexers.

Similarly, when S1S0 = 01, register B is selected, and the bus lines will receive the content provided by register B.

The following function table shows the register that is selected by the bus for each of the four possible binary values of the Selection lines.

S1	S0	Register Selected
0	0	A
0	1	B
1	0	C
1	1	D

Note: The number of multiplexers needed to construct the bus is equal to the number of bits in each register. The size of each multiplexer must be 'k * 1' since it multiplexes 'k' data lines. For instance, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus. Each multiplexer must have eight data input lines and three selection lines to multiplex one significant bit in the eight registers.

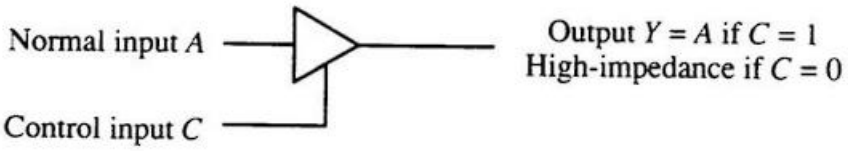
Bus Transfer Using Three States Bus Buffer:

A bus system can also be constructed using **three-state gates** instead of multiplexers.

The **three state gates** can be considered as a digital circuit that has three gates, two of which are signals equivalent to logic 1 and 0 as in a conventional gate. However, the third gate exhibits a high-impedance state.

The most commonly used three state gates in case of the bus system is a **buffer gate**.

The graphical symbol of a three-state buffer gate can be represented as:



The following diagram demonstrates the construction of a bus system with three-state buffers.

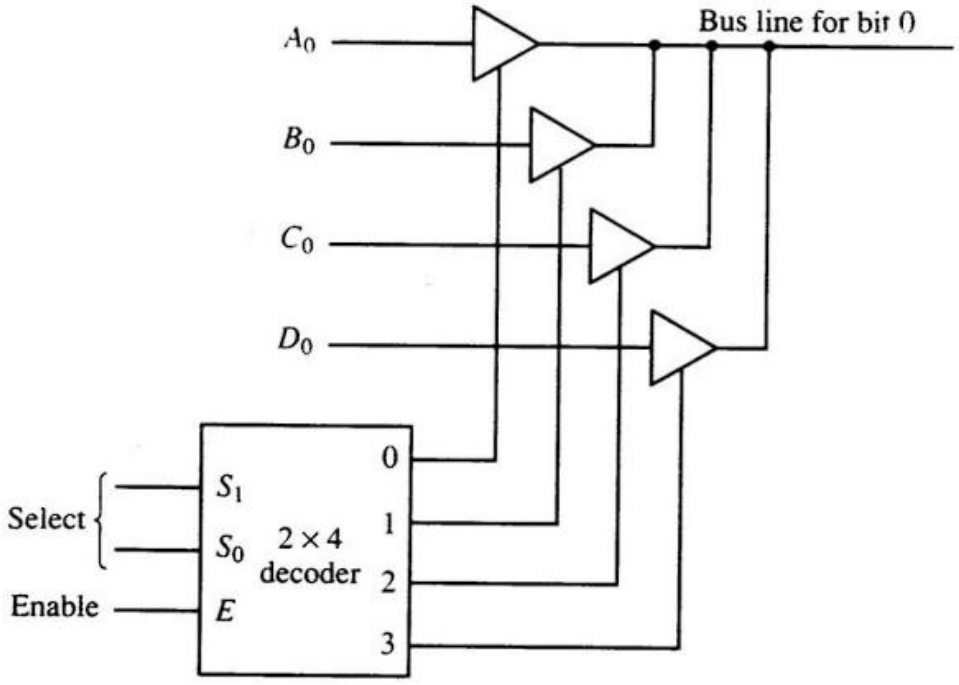


Figure 4-5 Bus line with three state-buffers.

- The outputs generated by the four buffers are connected to form a single bus line.
- Only one buffer can be in active state at a given point of time.
- The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line.
- A 2 * 4 decoder ensures that no more than one control input is active at any given point of time.

Memory Transfer:

Most of the standard notations used for specifying operations on memory transfer are stated below.

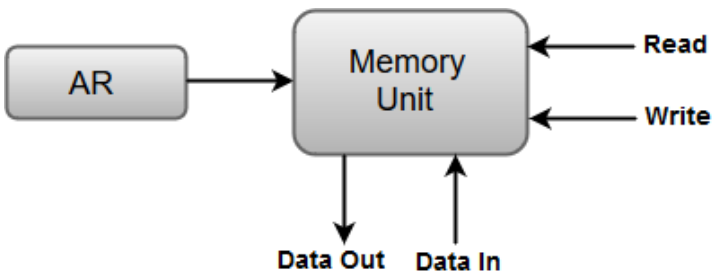
- The transfer of information from a memory unit to the user end is called a **Read** operation.
- The transfer of new information to be stored in the memory is called a **Write** operation.
- A memory word is designated by the letter **M**.
- We must specify the address of memory word while writing the memory transfer operations.
- The **address register** is designated by **AR** and the **data register** by **DR**.
- Thus, a read operation can be stated as:

```
Read:  DR ← M [AR]
```

- The **Read** statement causes a transfer of information into the data register (DR) from the memory word (M) selected by the address register (AR).
- And the corresponding write operation can be stated as:

```
Write: M [AR] ← R1
```

- The Write statement causes a transfer of information from register R1 into the memory word (M) selected by address register (AR).



Let us have a look at some of the registers and abbreviations used for memory:

- **MBR:** MBR stands for memory buffer register. It is also known as (Memory Data Register) MDR. It stores the data being transferred to and from memory.
- **MAR:** The Memory Address Register (MAR) is the CPU register used to store the memory's address at which the data is stored and fetched. It is often represented as AR (Address Register). M represents the memory word.

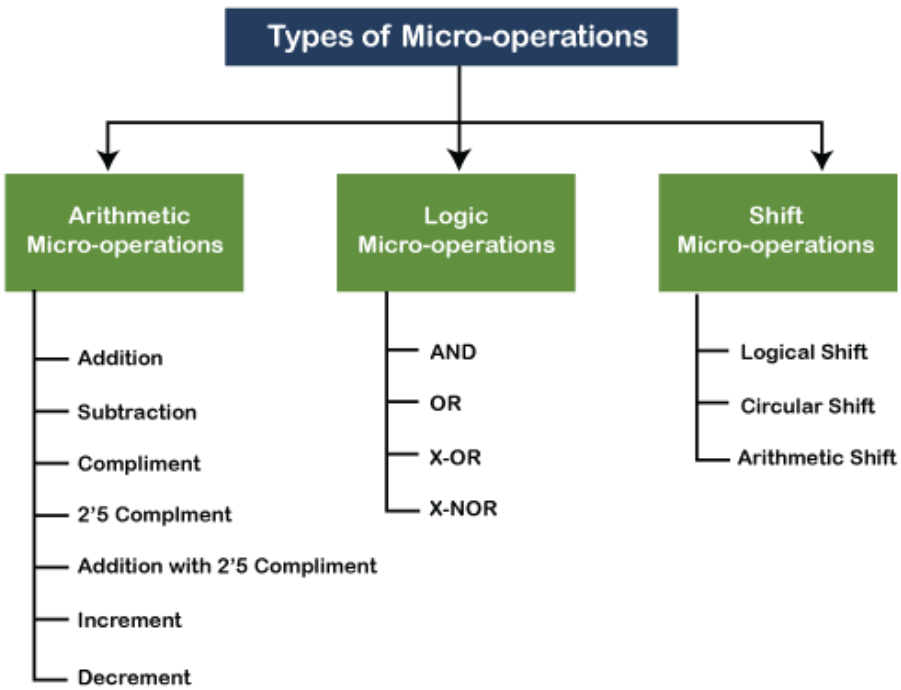
Micro-Operations:

As you know, a computer works on instructions. Our systems consist of machine instructions to perform some operations like add, subtract, multiply, divide etc. To achieve these operations, the system stores the data in registers. Now, to operate this data, the CPU has micro-operations.

A **micro-operation** is a simple operation performed on the data stored in one or more registers. They transfer the data between registers.

There are four types of micro-operations: -

1. Register micro-operations
2. Arithmetic micro-operations
3. Logic micro-operations
4. Shift micro-operations



Arithmetic Micro-Operations:

Arithmetic micro-operations perform operations on the numeric data stored in the registers.

The basic arithmetic micro-operations are-

1. Addition:

The Add arithmetic micro-operation adds the values of the two registers and stores the output in the desired register.

The symbolic notation for the Add arithmetic micro-operation is-

R3 ← R1 + R2

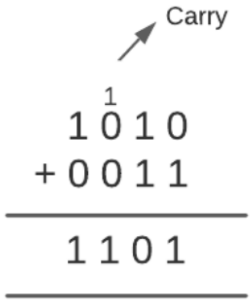
Here, R1 and R2 are the registers whose contents we want to add and, R3 is the desired register for storing the output.

Note: We can either store the output in another register or the same register, i.e., R1 or R2.

For example, consider the value of register R1 as 1010 and the value of register R2 as 0011. For performing the add arithmetic micro-operation remember the following rules:

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 10 (here, 0 is placed in the result and 1 is transferred as carry to the next column)

If we add R1 and R2, the output will be-



2. Subtraction:

The Subtract arithmetic micro-operation subtracts the values of the two registers and stores the output in the desired register.

The symbolic notation for the Add arithmetic micro-operation is-

R3 ← R1 - R2

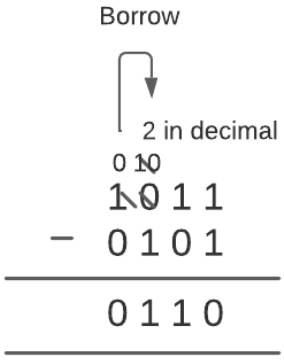
Here, R1 and R2 are the registers whose contents we want to subtract and, R3 is the desired register for storing the output.

Note: We can either store the output in another register or the same register, i.e. R1 or R2.

For example, consider the value of register R1 as 1011 and register R2 as 0101. For performing the subtract arithmetic micro-operation remember the following rules:

- 0 - 0 = 0
- 0 - 1 = 1 (because 10 is borrowed from next high order digit which is equal to 2 in decimal so 2 - 1 = 1)
- 1 - 0 = 1
- 1 - 1 = 0

If we subtract R2 from R1, the output will be-



Besides the above way, there is also an alternate way of doing the arithmetic subtraction. This way includes the use of the 2's complement.

To subtract the values of two registers, we need to add the first register, the complemented value of the second register and one.

The symbolic notation is-

R3 ← R1 + R2' + 1

Here, R1 and R2 are the registers whose contents we want to subtract and,

R3 is the desired register for storing the output.

Using this method, we get the same output as R1 - R2.

Note: We can either store the output in another register or the same register, i.e. R1 or R2.

For example, consider the value of register R1 as 1011 and register R2 as 0101 (same as we take firstly). Now, we will perform subtraction using the alternate method.

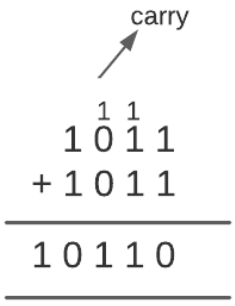
First, we will complement the value of the register R2. 0 will be converted to 1 and 1 to 0.

Therefore, the content of R2 will become 1010.

Second, we will add R2 and 1.

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

Finally, we will add R1 and R2.



We will ignore the overflow bit (1 in this case). So, our output will be 0110.

3. Increment:

The Increment arithmetic micro-operation increments the value of a register by 1. This means this operation adds 1 to the value of the given register and stores the output in the desired register.

The symbolic notation for the Increment arithmetic micro-operation is-

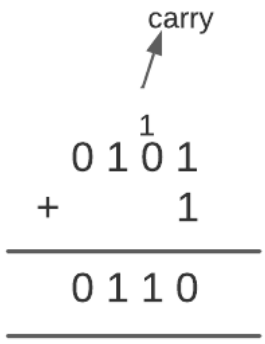
R1 ← R1 + 1

Here, R1 is the register whose value we want to increment and,

R1 is also the desired register for storing the output.

Note: We can store the output in another register or the same register.

For example, consider the value of register R1 as 0101. For performing the increment arithmetic micro-operation, we will add 1 to R1.



The Increment arithmetic micro-operations is carried out with the help of a combinational circuit or a binary up-down counter.

4. Decrement:

The Decrement arithmetic micro-operation decreases the value of a register by 1. This means this operation subtracts one from the value of the given register and stores the output in the desired register.

The symbolic notation for the Increment arithmetic micro-operation is-

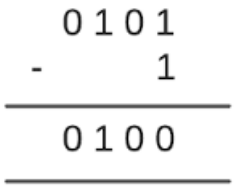
R1 ← R1 - 1

Here, R1 is the register whose value we want to decrement and,

R1 is also the desired register for storing the output.

Note: We can store the output in another register or the same register.

For example, consider the value of register R1 as 0101. For performing the decrement arithmetic micro-operation, we will subtract one from R1.



The Decrement arithmetic micro-operation is carried out with the help of a combinational circuit or a binary up-down counter.

5. 1's Complement:

The 1's complement arithmetic micro-operation complements the contents of a register. In this micro-operation, 0 is converted to 1 and 1 is converted to 0.

The symbolic notation for the 1's complement arithmetic micro-operation is-

R1 ← R1'

Here, R1 is the register whose value we want to complement and,

R1 is also the desired register for storing the output.

Note: We can store the output in another register or the same register.

For example, consider the value of register R1 as 0101. For performing the 1's complement arithmetic micro-operation, we will just convert 0 to 1 and 1 to 0.

Therefore, 1's complement of R1 will be 1010.

6. 2's Complement:

The 2's complement arithmetic micro-operation first complements the contents of the given register and then adds 1 to it. This micro-operation is also known as **Negation**.

The symbolic notation for the 2's complement arithmetic micro-operation is-

R2 ← R2' + 1

Here, R2 is the register on whose value we want to perform 2's complement and,

R2 is also the desired register for storing the output.

Note: We can store the output in another register or the same register.

For example, consider the value of register R2 as 0101. For performing the 2's complement arithmetic micro-operation first, we will find the 1's complement of R2.

The 1's complement of R2 will be 1010. Then we will add 1 to it.

$$\begin{array}{r} 1010 \\ +\quad\ 1 \\ \hline 1011 \\ \hline \end{array}$$

The signals that implement these operations propagate through gates in this case, and the result of the process can be transferred into a destination register via a clock pulse immediately after the output signal propagates through the combinational circuit.

Besides the above-described arithmetic micro-operations, there are two more arithmetic micro-operations- **multiply** and **divide**. These two operations are valid arithmetic operations, but they are not part of the required set of micro-operations. A series of add and shift micro-operations are used to perform the multiply micro-operation.

A series of subtracting and shifting micro-operations are used to complete the divide micro-operation.

The following table shows the symbolic representation of various Arithmetic Micro-operations.

Symbolic Representation	Description
R3 ← R1 + R2	The contents of R1 plus R2 are transferred to R3.
R3 ← R1 - R2	The contents of R1 minus R2 are transferred to R3.
R2 ← R2'	Complement the contents of R2(1's complement).
R2 ← R2' + 1	2's complement the contents of R2(negate).
R3 ← R1 + R2' + 1	R1 plus the 2's complement of R2(subtraction).
R1 ← R1 + 1	Increment the contents of R1 by one.
R1 ← R1 - 1	Decrement the contents of R1 by one.

Logic Micro-Operations:

Logic micro-operations are used on the bits of data stored in registers. These micro-operations treat each bit independently and create binary variables from them.

There are a total of 16 micro-operations available. These are-

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
$F_{15} = 1$	$F \leftarrow$ all 1's	Set to all 1's

Before discussing these logic micro-operations, let's discuss their truth tables.

The below diagram shows the truth table for all the 16 logic micro-operations mentioned above. Here, x and y are the variables or registers in which the data is stored and F0, F1, ….., F15 are the outputs that occur after performing these logic micro-operations.

TABLE 4-5 Truth Tables for 16 Functions of Two Variables

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Now, we will discuss these logic micro-operations one by one.

1. Clear:

The Clear logic micro-operation is used to clear the register or set the bits of the register to 0. To use this micro-operation, we need to feed 0 to the register. In the above truth table, F0 represents the truth table of Clear logic micro-operation.

For example, **F ← 0** means the value of the register F is set to 0 or is cleared. The previous value of register F will be removed.

Boolean expression-

The boolean expression for the Clear logic micro-operation is **F0 = 0**

2. AND:

The AND logic micro-operation performs the logical AND between the bits of the data stored in the two registers. The symbol to represent the logical AND is ∧.

Case 1: Both x and y values are true.

In the first case, if the values of both two registers are true then the result of AND operation is 1; else, it is 0. F1 represents the truth table of AND logic micro-operation in the above truth table.

For example, **F ← A ∧ B** means the registers A and B value will undergo AND micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for the AND logic micro-operation will be **F1 = x.y**

Case 2: x is true, and y is false.

The logical AND operation we discussed above gives output 1 when both x and y are true. There is also another AND operation which includes x but not y. Also known as **inhibition**, here for performing the AND operation, the first value is taken from the x variable or register. The second value is taken as the **complement** of the y variable or register. If the value of the x register is true and of the y register is false, then the result of AND operation is 1; else, it is 0.

F2 represents the truth table of inhibition AND logic micro-operation in the above truth table.

For example, **F <- A ∧ B'** means the value of the registers A and complement B will undergo AND micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for the AND logic micro-operation will be **F2 = x.y'**

Case 3: x is false, and y is true.

The third case of logical AND operation includes y but not x. Also known as **inhibition**, here for performing the AND operation, the first value is taken as the **complement** of the x variable or register, and the second value is taken from the y variable or register. If the value of the x register is false and of the y register is true, then the result of AND operation is 1; else, it is 0.

F4 represents the truth table of inhibition AND logic micro-operation in the above truth table.

For example, **F <- A' ∧ B** means the value of the complement register A and as it is B will undergo AND micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for the AND logic micro-operation will be **F4 = x'.y**

3. Transfer A:

The Transfer A logic micro-operation transfers the contents of register A (first register) to the output register.

F3 represents the truth table of Transfer A logic micro-operation in the above truth table. Since there is a transfer of data from the first register to the output register in this micro-operation, its truth table is the same as the taken values of the x variable (0, 0, 1, 1).

For example, **F ← A** means the value of register A is moved to register F. The previous value of register F will be removed.

Boolean expression-

The boolean expression for the Transfer A logic micro-operation is **F3 = x**

4. Transfer B:

The Transfer B logic micro-operation transfers the contents of register B (second register) to the output register.

F5 represents the truth table of Transfer B logic micro-operation in the above truth table. Since there is a transfer of data from the second register to the output register in this micro-operation, its truth table is the same as the taken values of the y variable (0, 1, 0, 1).

For example, **F ← B** means the value of register B is moved to register F. The previous value of register F will be removed.

Boolean expression-

The boolean expression for the Transfer B logic micro-operation is **F5 = y**

5. Exclusive OR:

Also known as XOR, this logic micro-operation performs the logical XOR between the data bits stored in the two registers. The logical XOR means either x should be true or y but not both. The symbol to represent the Exclusive OR is ⊕.

F6 represents the truth table of Exclusive OR logic micro-operation in the above truth table. The output will be 1 when either x =1 and y = 0 or x = 0 and y = 1.

For example, **F ← A ⊕ B** means the registers A and B value will undergo XOR micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for the Exclusive OR logic micro-operation will be **F6 = x.y' + x'.y**

6. OR:

The OR logic micro-operation performs the logical OR between the data bits stored in the two registers. The symbol to represent the logical OR is ∨.

Case 1: Either x or y or both x and y values are true.

In the first case, if either the value of x register is true and y register is false, or the value of x register is false, and y register is true, or both the values of x and y registers are true, then the result of OR operation is 1 else it is 0. F7 represents the truth table of OR logic micro-operation in the above truth table.

For example, **F ← A ∨ B** means the registers A and B value will undergo OR micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for the OR logic micro-operation will be **F7 = x + y**

Case 2: If y, then x else not.

In the second case, the output for 1 follows the condition that

- If the value of the y register is true, then the value of the x register must be true. If this condition is satisfied, then the output is 1.
- If the value of the y register is false, then we don't need to look for the value of the x register, and the output is 1.
- Else the output is 0.

To perform this logic micro-operation, we need to perform the logical OR of the values of the x register and the complement value of the y register.

In the above truth table, F11 represents the truth table of this logic micro-operation.

For example, **F ← A ∨ B'** means the value of the registers A and complement B will undergo OR micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for this OR logic micro-operation will be **F11 = x + y'**

Case 3: If x, then y else not.

In the second case, the output for 1 follows the condition that

- If the value of the x register is true, then the y register's value must be true. If this condition is satisfied, then the output is 1.
- If the value of the x register is 0, then we don't need to look for the value of the y register, and the output is 1.
- Else the output is 0.

To perform this logic micro-operation, we need to perform the logical OR of the complemented value of the x register and the value of the y register.

In the above truth table, F13 represents the truth table of this logic micro-operation.

For example, **F ← A' ∨ B** means the complemented register A and B value will undergo OR micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for this OR logic micro-operation will be **F13 = x' + y**

7. NOR:

The NOR logic micro-operation is simply the opposite of OR logic micro-operation. As the name suggests, it is Not OR. The output of OR micro-operation is 1 when the value of either x register or y register or both x and y registers are true. In contrast, in NOR, the output is 0 when the value of either x register or y register or both x and y registers are true, and it is 1 when both x and y registers are false. In the above truth table, F8 represents the truth table of NOR logic micro-operation.

For example, **F ← (A ∨ B)'** means the registers A and B value will undergo NOR micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for the Transfer A logic micro-operation is **F8 = (x + y)'**

8. Exclusive NOR:

If we perform the Exclusive NOR micro-operation, the output will be 1 when the values of both the x and y registers will be the same. They can be true or false, but they have to be the same.

F9 represents the truth table of Exclusive NOR logic micro-operation in the above truth table. The output will be 1 when either x = 0 and y = 0 or x = 1 and y = 1.

For example, **F ← (A ⊕ B)'** means the registers A and B value will undergo Exclusive NOR micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for the Exclusive NOR logic micro-operation will be **F9 = x.y + x'.y'**

9. Complement B:

The Complement B logic micro-operation transfers the complemented contents of register B (second register) to the output register. First, the content of the register is complemented and then moved to the desired register.

In the above truth table, F10 represents the truth table of Complement B logic micro-operation. Since there is a transfer of complemented data from the second register to the output register in this micro-operation, its truth table is just the opposite of the taken values of the y variable (1, 0, 1, 0).

For example, **F ← B'** means the complemented value of register B is moved to register F. The previous value of register F will be removed.

Boolean expression -

The boolean expression for the Complement B logic micro-operation is **F10 = y'**

10. Complement A:

The Complement A logic micro-operation transfers the complemented contents of register A (first register) to the output register. First, the content of the register is complemented and then moved to the desired register.

F12 represents the truth table of Complement A logic micro-operation in the above truth table. Since there is a transfer of complemented data from the first register to the output register in this micro-operation, its truth table is just the opposite of the taken values of the y variable (1, 1, 0, 0).

For example, **F ← A'** means the complemented value of register A is moved to register F. The previous value of register F will be removed.

Boolean expression-

The boolean expression for the Complement A logic micro-operation is **F12 = x'**

11. NAND:

The NAND logic micro-operation is simply the opposite of AND logic micro-operation. As the name suggests, it is Not AND. The output of AND micro-operation is 1 when the value of both the x register and y register is true. In contrast, in NAND, the output is 0 when the value of both x register and y register is true, and it is 1 when either x is false, or y is false, or both are false.

In the above truth table, F14 represents the truth table of NAND logic micro-operation.

For example, **F ← (A ∧ B)'** means the registers A and B value will undergo NAND micro-operation, and the output will be stored in register F.

Boolean expression-

The boolean expression for the NAND logic micro-operation is **F14 = (x.y)'**

12. Set to all 1's:

The set to all 1's logic micro-operations is used to set all the register bits to 1. To use this micro-operation, we just need to feed 1 to the register. In the above truth table, F15 represents the truth table of Set to all 1's logic micro-operation.

For example, **F ← 1** means the value of the register F is set to 1. The previous value of register F will be removed.

Boolean expression-

The boolean expression for the Clear logic micro-operation is **F15 = 1**

Examples of Logic Micro Operations in Real-World Computing:

Logical microoperations are performed on binary data at a hardware level in the processor. Let's discuss some examples of logical micro-operations in real-world computation.

- Logical microoperations are used in **MUX** (multiplexers) and **DEMUX** (Demultiplexers). MUX is used in address decoding, data routing, etc. While DEMUX is used in signal demodulation and functions as a serial-to-parallel converter.
- Logical microoperations are used in **cryptography, image processing**, etc., where we need accurate control over the individual bits.
- Logical microoperations, such as shift microoperations, are used in data storage, compression algorithms, and in other areas to use memory effectively.
- They are used in **machine learning algorithms** for manipulating the data and for making decisions according to the patterns in the dataset. The algorithms are further used in **image recognition, voice recognition, natural language processing,** etc.
- Logical microoperations are used for **masking** purposes. The AND microoperation is used to mask bits of a register.
- Logical microoperations are used to build **arithmetic circuits** such as adders and subtracters. They are formed by the combination of XOR, AND, and OR for performing subtraction and addition operations.

Advantages of Logic Micro Operations in Processor Design:

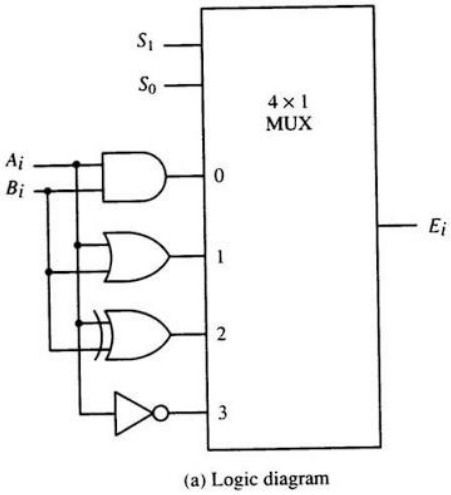
Below are some of the advantages of logic micro-operations.

- Logical microoperations **consume less power**. Therefore, they help in building low-power consumption systems.
- They are **easy to scale** and can hold larger amounts of datasets.
- Logical microoperations can **reduce the complexity** of computations by breaking them into manageable components so that they can be optimized effectively.
- Logic micro-operations are **very fast** and allow efficient processing of huge amounts of data.
- Logical micro-operations are very **flexible** and can be combined in multiple ways to perform a wide number of computations.

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.
- Although there are 16 logic microoperations, most computers use only four--AND, OR, XOR (exclusive-OR), and complement from which all others can be derived.
- Figure 4-10 shows one stage of a circuit that generates the four basic logic microoperations.
- It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic.
- The outputs of the gates are applied to the data inputs of the multiplexer. The two selection inputs S1 and S0 choose one of the data inputs of the multiplexer and direct its value to the output.

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	$E = \overline{A}$	Complement

(b) Function table

Shift Micro-Operations:

Shift micro-operations are used when the data is stored in registers. These micro-operations are used for the serial transmission of data. Here, the data bits are shifted from left to right. These micro-operations are also combined with arithmetic and logic micro-operations and data-processing operations.

There are three types of shift micro-operations-

1. Logical Shift
2. Arithmetic Shift
3. Circular Shift

Let's start with logical shift micro-operation.

1. Logical Shift:

The logical shift micro-operation moves the 0 through the serial input. There are two ways to implement the logical shift.

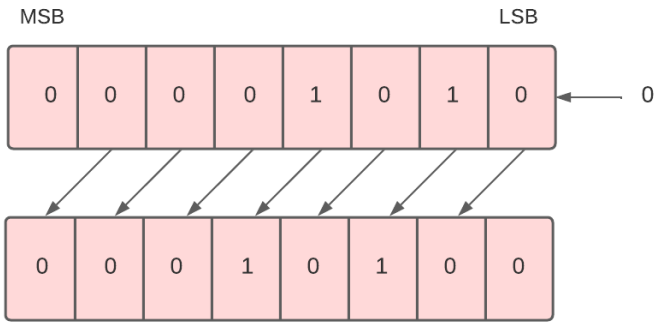
1. Logical Shift Left
2. Logical Shift Right

Let's discuss both of them one by one.

Logical Shift Left:

Each bit in the register is shifted to the left one by one in this shift micro-operation. The most significant bit (MSB) is moved outside the register, and the place of the least significant bit (LSB) is filled with 0.

For example, in the below data, there are 8 bits 00001010. When we perform a logical shift left on these bits, all these bits will be shifted towards the left. The MSB or the leftmost bit i.e. 0 will be moved outside, and at the rightmost place or LSB, 0 will be inserted as shown below.



To implement the logical shift left micro-operation, we use the **shl** symbol.

For example, R1 -> shl R1.

This command means the 8 bits present in the R1 register will be logically shifted left, and the result will be stored in register R1.

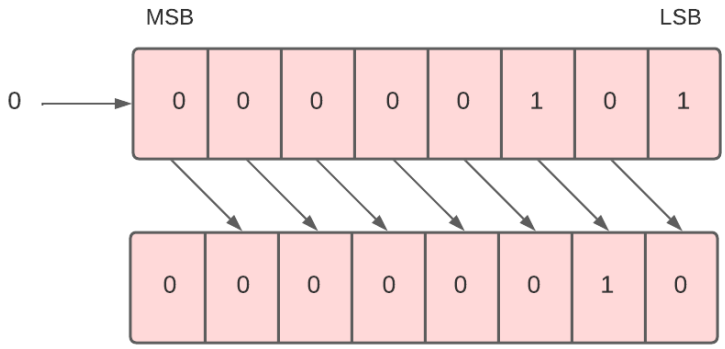
Moreover, the logical shift left microoperation denotes the multiplication of 2. The example we've taken above when converted into decimal forms the number 10. And the result after the logical shift operation when converted to decimal forms the number 20.

Next, we will see the logical shift right micro-operation.

Logical Shift Right:

Each bit in the register is shifted to the right one by one in this shift micro-operation. The least significant bit (LSB) is moved outside the register, and the place of the most significant bit (MSB) is filled with 0.

For example, in the below data, there are 8 bits 00000101. When we perform a logical shift right on these bits, all these bits will be shifted towards the right. The LSB or the rightmost bit i.e. 1 will be moved outside, and at the leftmost place or MSB, 0 will be inserted as shown below.



To implement the logical shift right micro-operation, we use the **shr** symbol.

For example, R1 -> shr R1.

This command means the 8 bits present in the R1 register will be logically shifted right, and the result will be stored in register R1.

Logical right shift micro-operation generally denotes division by 2. The inputted bits when converted into decimal form the number 5. And the outcome when converted into decimal forms the number 2.

Next, we will study arithmetic shift micro-operation.

2. Arithmetic Shift:

The arithmetic shift micro-operation moves the signed binary number either to the left or to the right position. There are two ways to implement the arithmetic shift.

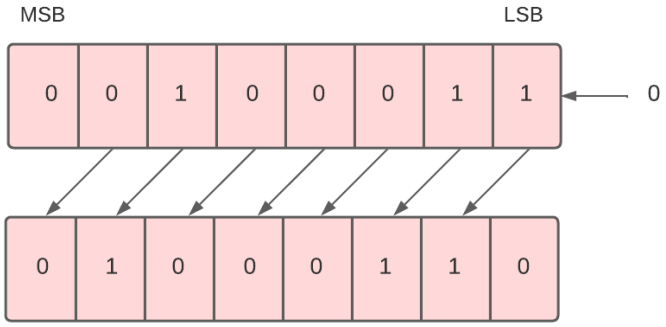
1. Arithmetic Shift Left
2. Arithmetic Shift Right

Let's discuss both of them one by one.

Arithmetic Shift Left:

The arithmetic shift left micro-operation is the same as the logical shift left micro-operation. Each bit in the register is shifted to the left one by one in this shift micro-operation. The most significant bit (MSB) is moved outside the register, and the place of the least significant bit (LSB) is filled with 0.

For example, in the below data, there are 8 bits 00100011. When we perform the arithmetic shift left on these bits, all these bits will be shifted towards the left. The MSB or the leftmost bit i.e. 0 will be moved outside, and at the rightmost place or LSB, 0 will be inserted as shown below.



The given binary number (00100011) represents 35 in the decimal system. And the binary number after logical shift left (01000110) represents 70 in a decimal system. Since 35 * 2 = 70. Therefore, we can say that the arithmetic shift left multiplies the number by 2.

To implement the arithmetic shift left micro-operation, we use the **ashl** symbol.

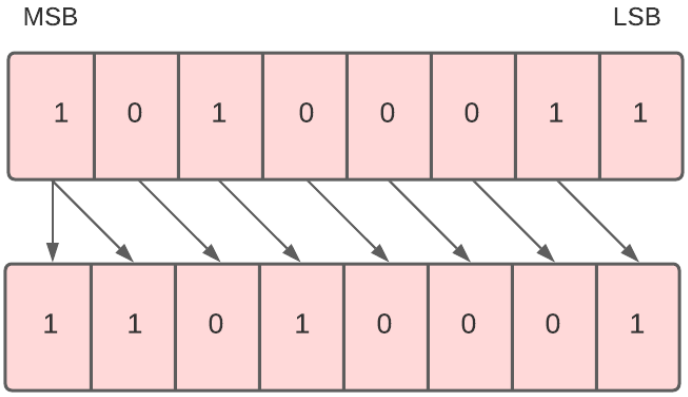
For example, R1 -> ashl R1.

This command means the 8 bits present in the R1 register will be arithmetic shifted left, and the result will be stored in register R1.

Arithmetic Shift Right:

Each bit in the register is shifted to the right one by one in this shift micro-operation. The least significant bit (LSB) is moved outside the register, and the place of the most significant bit (MSB) is filled with the previous value of MSB.

For example, in the below data, there are 8 bits 10100011. When we perform an arithmetic shift right on these bits, all these bits will be shifted towards the right. The LSB or the rightmost bit i.e. 1 will be moved outside, and at the leftmost place or MSB, the previous MSB value, i.e. 1, will be inserted as shown below.



Arithmetic right shift divides the number by 2.

To implement the arithmetic shift right micro-operation, we use the **ashr** symbol.

For example, R1 -> ashr R1.

This command means the 8 bits present in the R1 register will be arithmetic shifted right, and the result will be stored in register R1.

Next, we will study circular shift micro-operation.

3. Circular Shift:

The circular shift, also known as the rotate shift, moves the bits in the register's sequence around both ends, thus ensuring no loss of information. There are two ways to implement the circular shift.

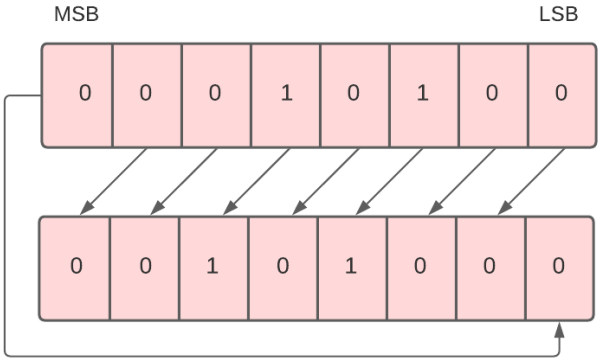
1. Circular Shift Left
2. Circular Shift Right

Let’s discuss both of them one by one.

Circular Shift Left:

Each bit in the register is shifted to the left one by one in this shift micro-operation. After shifting, the least significant bit (LSB) place becomes empty, so it is filled with the value at the most significant bit (MSB).

For example, in the below data, there are 8 bits 00010100. When we perform a circular shift left on these bits, all these bits will be shifted towards the left. The MSB or the leftmost bit i.e. 0 will be placed at the rightmost place or LSB as shown below.



To implement the circular shift left micro-operation, we use the **cil** symbol.

For example, R1 -> cil R1.

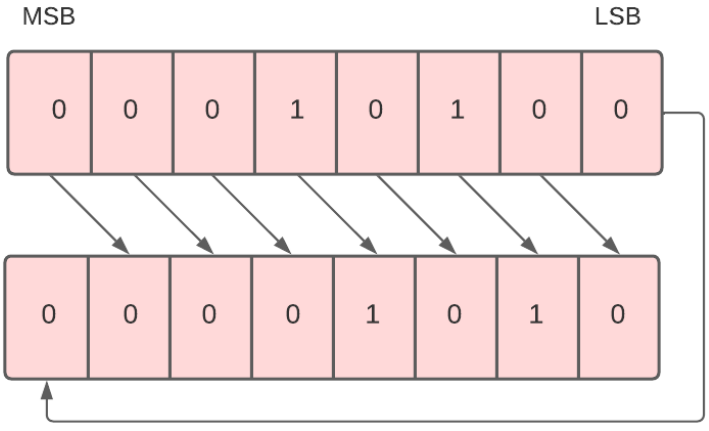
This command means the 8 bits present in the R1 register will be circular shifted left, and the result will be stored in register R1.

Next, we will discuss circular shift right micro-operation.

Circular Shift Right:

Each bit in the register is shifted to the right one by one in this shift micro-operation. After shifting, the most significant bit (MSB) place becomes empty, so it is filled with the value at the least significant bit (LSB).

For example, in the below data, there are 8 bits 00010100. When we perform a circular shift right on these bits, all these bits will be shifted towards the right. The LSB or the leftmost bit, i.e. 0, will be placed at the rightmost place or MSB.



To implement the circular shift right micro-operation, we use the **cir** symbol.

For example, R1 -> cir R1.

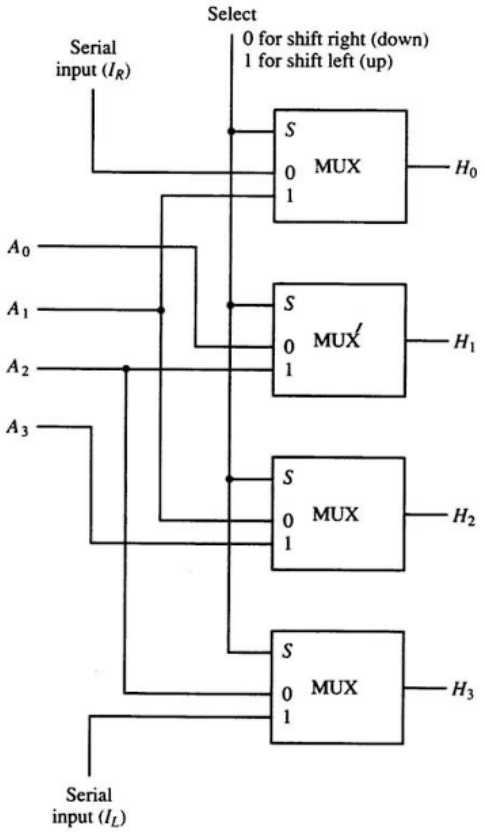
This command means the 8 bits present in the R1 register will be circular shifted right, and the result will be stored in register R1.

TABLE 4-7 Shift Microoperations

Symbolic designation	Description
$R \leftarrow \text{shl } R$	Shift-left register R
$R \leftarrow \text{shr } R$	Shift-right register R
$R \leftarrow \text{cil } R$	Circular shift-left register R
$R \leftarrow \text{cir } R$	Circular shift-right register R
$R \leftarrow \text{ashl } R$	Arithmetic shift-left R
$R \leftarrow \text{ashr } R$	Arithmetic shift-right R

Hardware Implementation:

- A combinational circuit shifter can be constructed with multiplexers as shown in Fig. 4-12.
- The 4-bit shifter has four data inputs, A0 through A3, and four data outputs, H0 through H3.
- There are two serial inputs, one for shift left (IL) and the other for shift right (IR).
- When the selection input S=0 the input data are shifted right (down in the diagram).
- When S = 1, the input data are shifted left (up in the diagram).
- The function table in Fig. 4-12 shows which input goes to each output after the shift.
- A shifter with n data inputs and outputs requires n multiplexers.
- The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.



Function table

Select	Output			
S	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

Figure 4-12 4-bit combinational circuit shifter.

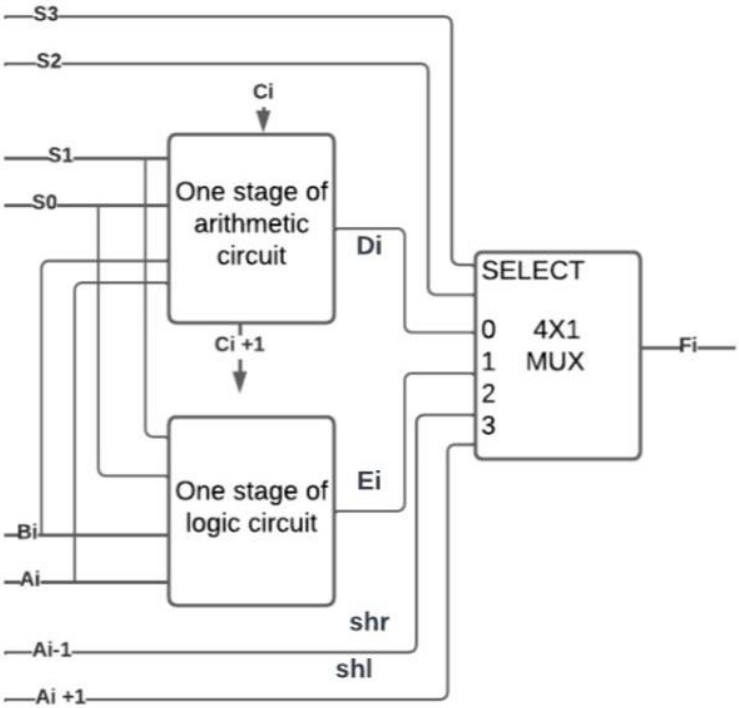
Arithmetic Logic Shift Unit (ALU):

The Arithmetic Logic Shift Unit (ALSU) is part of a computer system's Arithmetic Logic Unit (ALU). It can be defined as a digital circuit that performs arithmetic, logical, and shift operations. Instead of having individual registers performing microoperations directly, computer systems employ many storage registers connected to a common operational unit ALU.

The major characteristic of ALSU is to perform all the logical, arithmetic, and shift operations. Operations like addition, subtraction, multiplication, and division are called arithmetic operations. Logical operation refers to operation on numbers and special character operations, and it connects two or more phrases of information(expressions). Shift micro-operations are operations for serial transfer of information.

One Stage of Arithmetic Logic Shift Unit:

The arithmetic, logic, and shift circuits are combined to one ALU with common selection variables. The diagram below shows one stage of an arithmetic logic shift unit. The subscript i designates a typical stage.



With inputs S1 and S0, a specific microoperation is chosen. At the output, a 4 x 1 multiplexer selects between an arithmetic output and a logic output. Inputs S3 and S2 select the data in the multiplexer. The multiplexer's other two data inputs receive inputs Ai — 1 for the shift-right operation(**shr**) and Ai + 1 for the shift-left operation (**shl**).

It should be noted that the circuit diagram only depicts one typical stage. For an n-bit ALU, the circuit shown above must be repeated n times. The output that carries C_{i+1} of a given arithmetic stage must be connected to the input carry C_i of the next stage in the sequence. The input carried to the first stage is the input carry Cin, which provides a selection variable for the arithmetic operations.

The circuit whose one stage is specified in the diagram given above provides eight arithmetic, four logical, and two-shift operations. Each operation is selected with five variables S3, S2, S1, S0, and C_{in}. Here C_{in} is used for selecting an arithmetic operation only. The table given below is the function table of the Arithmetic Logic Shift Unit.

TABLE 4-8 Function Table for Arithmetic Logic Shift Unit

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

The above table shows 14 operations of **ALU** (Arithmetic Logical Unit).

1. The first eight are arithmetic operations (where S3S2=0).
2. The next four operations are logical and are selected with S3S2=01.
3. The final two operations are shift operations, selected with S3S2=10 and 11.

THE END 😴