Introduction

Instruction formats in computer architecture refer to the way instructions are encoded and represented in Machine Language. Types of instruction formats in computer architecture are: zero, one, two, and three-address instructions.

A computer program is a set of instructions that suggests the computer to perform a specific task. A computer program can contain multiple statements. The instruction length is generally multiples of the character length (8 bits).

First, let us recall and discuss some important concepts related to Instruction Format in Computer Architecture:

1. Instruction includes a set of operational codes, operands, opcode, and addressing mode.
2. Instruction length is the most fundamental issue of the format design. The longer the instruction, the longer will be the time taken to fetch the instruction.
3. The number of bits is directly proportional to the memory range. i.e., the larger the range requirement, the more number bits will be required.
4. If a system supports the virtual memory, then the memory range that needs to be addressed by the instruction will be larger than the physical memory.
5. Instruction length should be equal to or the multiple of data bus length.

Types of Instructions:

There are certain basic operations included in every computer's instructions set. The computer instructions are classified into three categories:

Data Transfer Instructions:

These instructions involve the movement of data between different storage locations within the computer system. This can include transferring data between registers, between memory and registers, or between the CPU and external devices.

Examples:

LOAD: Transfers data from memory to a register.
STORE: Moves data from a register to memory.
MOVE: Copies data from one location to another.

Data Manipulation Instructions:

Instructions in this category perform various operations on data. This can involve arithmetic operations for numerical calculations, logical operations for decision-making, and bitwise operations for manipulating individual bits.

Examples:

ADD: Adds two values together.
SUBTRACT: Subtracts one value from another.
AND, OR, XOR: Perform logical operations on binary data.
SHIFT: Move bits left or right in a binary number.

Program Control Instructions:

These instructions manage the flow of the program. They control which instruction is executed next based on conditions or explicitly defined jumps.

Examples:

JUMP: Unconditionally transfers control to another part of the program.

BRANCH: Transfers control based on a condition (conditional jump).

CALL: Jumps to a subroutine or function and stores the return address.

RETURN: Resumes execution at the return address from a subroutine call.

These three categories cover the fundamental operations necessary for a computer to perform any task. In addition to these primary categories, there may be other specialized instructions that support specific operations or architectures. For example, vector instructions for parallel processing, floating-point instructions for handling decimal numbers, and privileged instructions for interacting with the operating system.

The combination and arrangement of these basic instructions in a program form the basis of machine code or assembly language programs, which are then translated into executable code by an assembler or compiler for the target architecture.

What is Instruction Format in Computer Architecture?

The instruction formats are a sequence of bits (0 and 1). These bits, when grouped, are known as fields. Each field of the machine provides specific information to the CPU related to the operation and location of the data.

The instruction format also defines the layout of the bits for an instruction. It can be of variable lengths with multiple numbers of addresses. These address fields in the instruction format vary as per the organization of the registers in the CPU. The formats supported by the CPU depend upon the Instructions Set Architecture implemented by the processor.

Depending on the multiple address fields, the instruction is categorized as follows:

1. Three address instruction
2. Two address instruction
3. One address instruction
4. Zero address instruction

The operations specified by a computer instruction are executed on data stored in memory or processor registers. The operands residing in processor registers are specified with an address. The registered address is a binary number of k bits that defines one of the 2^k registers in the CPU. Thus, a CPU with 16 processors registers R0 through R15 and will have a four-bit register address field.

Example: The binary number 0011 will designate register R3.

A computer can have instructions of different lengths containing varying numbers of addresses. The number of address fields of a computer depends on the internal design of its registers. Most of the computers fall into one of three types of CPU organizations:

1. Single accumulator organization.
2. General register organization.
3. Stack organization.

1. Single Accumulator Organization

All the operations on a system are performed with an implied accumulator register. The instruction format in this type of computer uses one address field.

For example, the instruction for arithmetic addition is defined by an assembly language instruction 'ADD.'

Where X is the operand's address, the ADD instruction results in the operation.

$AC \leftarrow AC + M[X]$.

AC is the accumulator register, M[X] symbolizes the memory word located at address X.

2. General Register Organization

The general register type computers employ two or three address fields in their instruction format. Each address field specifies a processor register or a memory. An instruction symbolized by ADD R1, X specifies the operation $R1 \leftarrow R + M[X]$.

This instruction has two address fields: register R1 and memory address X.

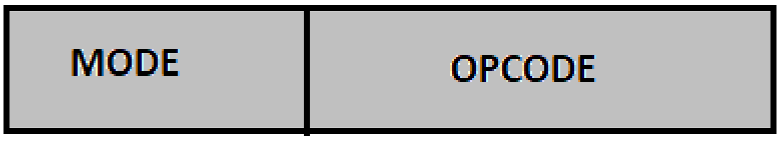
3. Stack Organization

A computer with a stack organization has PUSH and POP instructions that require an address field. Hence, the instruction PUSH X pushes the word at address X to the top of the stack. The stack pointer updates automatically. In stack-organized computers, the operation type instructions don't require an address field as the operation is performed on the two items on the top of the stack.

Types of Instruction Formats

1. Zero Address Instruction

This instruction does not have an operand field, and the location of operands is implicitly represented. The stack-organized computer system supports these instructions. To evaluate the arithmetic expression, it is required to convert it into reverse polish notation.



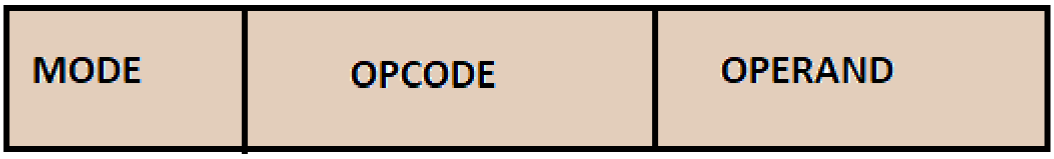
Example: Consider the below operations, which shows how X = (A + B) ∗ (C + D) expression will be written for a stack-organized computer.

TOS: Top of the Stack

PUSH	A	TOS ← A
PUSH	B	TOS ← B
ADD		TOS ← (A + B)
PUSH	C	TOS ← C
PUSH	D	TOS ← D
ADD		TOS ← (C + D)
MUL		TOS ← (C + D) ∗ (A + B)
POP	X	M [X] ← TOS

2. One Address Instruction

This instruction uses an implied accumulator for data manipulation operations. An accumulator is a register used by the CPU to perform logical operations. In one address instruction, the accumulator is implied, and hence, it does not require an explicit reference. For multiplication and division, there is a need for a second register. However, here we will neglect the second register and assume that the accumulator contains the result of all the operations.



Example: The program to evaluate X = (A + B) ∗ (C + D) is as follows:

LOAD	A	AC ← M [A]
ADD	B	AC ← A [C] + M [B]
STORE	T	M [T] ← AC
LOAD	C	AC ← M [C]
ADD	D	AC ← AC + M [D]
MUL	T	AC ← AC ∗ M [T]
STORE	X	M [X] ← AC

All operations are done between the accumulator (AC) register and a memory operand.

M [ ] is any memory location.

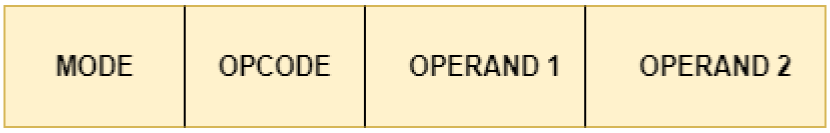
M[T] addresses a temporary memory location for storing the intermediate result.

This instruction format has only one operand field. This address field uses two special instructions to perform data transfer, namely:

- LOAD: This is used to transfer the data to the accumulator.
- STORE: This is used to move the data from the accumulator to the memory.

3. Two Address Instructions

This instruction is most commonly used in commercial computers. This address instruction format has three operand fields. The two address fields can either be memory addresses or registers.

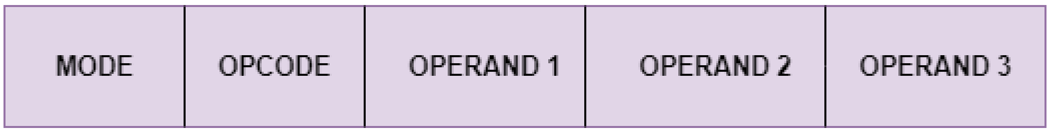


Example: The program to evaluate X = (A + B) ∗ (C + D) is as follows:

```
MOV        R1, A        R1 ← M [A]
ADD        R1, B        R1 ← R1 + M [B]
MOV        R2, C        R2 ← M [C]
ADD        R2, D        R2 ← R2 + M [D]
MUL        R1, R2       R1 ← R1∗R2
MOV        X, R1        M [X] ← R1
```
The MOV instruction transfers the operands to the memory from the processor registers. R1, R2 registers.

4. Three Address Instruction

The format of a three-address instruction requires three operand fields. These three fields can be either memory addresses or registers.



Example: The program in assembly language X = (A + B) ∗ (C + D) Consider the instructions given below that explain each instruction's register transfer operation.

```
ADD    R1, A, B       R1 ← M [A] + M [B]
ADD    R2, C, D       R2 ← M [C] + M [D]
MUL    X, R1, R2      M [X] ← R1 ∗ R2
```
Two processor registers, R1 and R2.

The symbol M [A] denotes the operand at memory address symbolized by A. The operand1 and operand2 contain the data or address that the CPU will operate. Operand 3 contains the result's address.

Advantages and Disadvantages

Here are some advantages and disadvantages of each instruction format:

1. **Single accumulator organization:**
   Advantages: Simple design, low memory requirements, efficient for certain types of computations.
   Disadvantages: Limited parallelism, limited functionality.

2. **General register organization:**
   Advantages: More versatile, supports parallel processing, more efficient for certain types of computations.
   Disadvantages: More complex design, higher memory requirements.

3. **Stack organization:**
   Advantages: Simple design, low memory requirements, supports recursive function calls.
   Disadvantages: Limited parallelism, slow access to non-top elements.

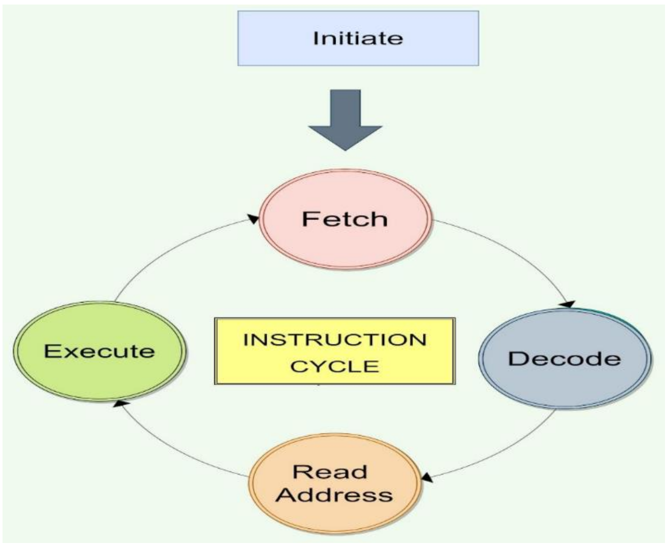
Instruction Cycle

The instruction cycle is defined as the basic cycle in which a computer system fetches an instruction from memory, decodes it, and then executes it. Fetch-Execute-Cycle is another name for it. All instructions in a computer system are executed in the RAM of the computer system. The CPU is in charge of carrying out the instruction.

Each instruction cycle in a basic computer includes the following procedures:

- It has the ability to retrieve instructions from memory.
- It's used to decode the command.
- If the instruction has an indirect address, it can read the effective address from memory.
- It is capable of carrying out the command.

The instruction cycle is divided into five stages, which are described below:



Initiating Cycle

During this phase, the computer system boots up and the Operating System loads into the central processing unit's main memory. It begins when the computer system starts.

Fetching of Instruction

The first phase is instruction retrieval. Each instruction executed in a central processing unit uses the fetch instruction. During this phase, the central processing unit sends the PC to MAR and then the READ instruction to a control bus. After sending a read instruction on the data bus, the memory returns the instruction that was stored at that exact address in the memory. The CPU then copies data from the data bus into MBR, which it then copies to registers. The pointer is incremented to the next memory location, allowing the next instruction to be fetched from memory.

Decoding of Instruction

The second phase is instruction decoding. During this step, the CPU determines which instruction should be fetched from the instruction and what action should be taken on the instruction. The instruction's opcode is also retrieved from memory, and it decodes the related operation that must be performed for the instruction.

Read of an Effective Address

The third phase is the reading of an effective address. The operation's decision is made during this phase. Any memory type operation or non-memory type operation can be used. Direct memory instruction and indirect memory instruction are the two types of memory instruction available.

Execution of Instruction

The last step is to carry out the instructions. The instruction is finally carried out at this stage. The instruction is carried out, and the result is saved in the register. The CPU gets prepared for the execution of the next instruction after the completion of each instruction. The execution time of each instruction is calculated, and this information is used to determine the processor's processing speed.

Different Instruction Cycles

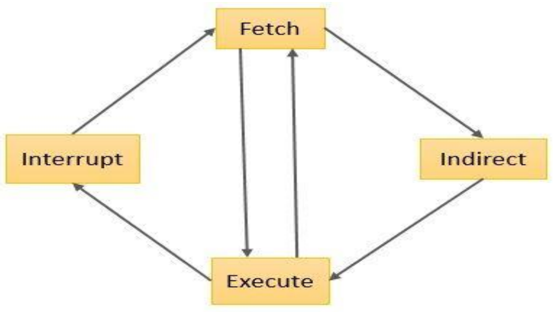
The concept of instruction cycles is integral to understanding how a computer's central processing unit (CPU) executes instructions. Here's an explanation of each of these cycles:

Fetch Cycle

- **Description:** The fetch cycle is the initial stage of the instruction cycle. It involves retrieving the next instruction from memory.
- **Operation:** The CPU uses the program counter (PC) to access the memory location where the next instruction is stored. The instruction is fetched and placed in the instruction register (IR).
- **Purpose:** This cycle ensures that the CPU has the next instruction ready for decoding and execution.

Indirect Cycle

- **Description:** The indirect cycle is sometimes required when instructions involve accessing memory locations that contain addresses or pointers to the actual data.
- **Operation:** During this cycle, the CPU may use an address obtained from the previous instruction to access another memory location, which holds the data or another address to be used in the next cycle.
- **Purpose:** The indirect cycle enables the CPU to follow memory references and retrieve the actual data required for execution.



Instruction Cycle with Indirect Cycle

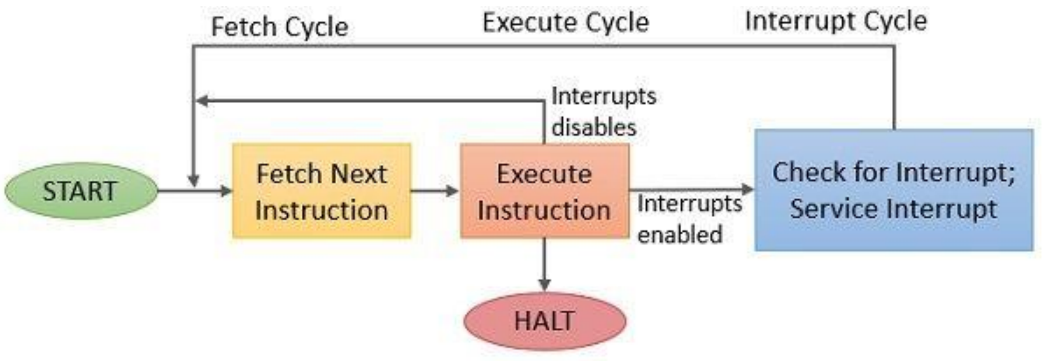
Execute Cycle

- **Description:** The execute cycle is where the central processing unit performs the operation specified by the decoded instruction.
- **Operation:** The CPU carries out arithmetic computations, logical operations, data transfers, or any other actions as dictated by the instruction. This may involve accessing data from registers or memory, performing calculations, and updating registers or memory locations.
- **Purpose:** The execution stage accomplishes the intended operation and is where the actual work of the instruction takes place.

Interrupt Cycle

- **Description:** The interrupt cycle comes into play when an external event or condition triggers an interrupt, causing the CPU to temporarily suspend its current execution to handle the interrupt request.

- **Operation:** The CPU saves its current state (program counter and other relevant information) before jumping to an interrupt service routine (ISR). After servicing the interrupt, the CPU may restore its state and continue execution.



Instruction Cycle with Interrupts

- **Purpose:** Interrupt cycles enable a CPU to respond to external events or asynchronous inputs promptly without losing important data or program context.

Uses of Different Instruction Cycles

The different instruction cycles (fetch, indirect, execute, and interrupt) in a computer's operation have different purposes and applications, ensuring efficient and responsive processing. Here are the uses of each instruction cycle:

Fetch Cycle

- **Use:** Retrieving the next instruction from memory.
- **Application:** Essential for the sequential execution of program instructions, ensuring the CPU has the next instruction ready for decoding and execution.
- **Example:** Fetching the opcode of the next instruction from memory to be decoded and executed.

Indirect Cycle

- **Use:** Handling instructions that involve accessing memory locations containing addresses or pointers.
- **Application:** Facilitates memory referencing, allowing the CPU to navigate through multiple levels of indirection to access the actual data or instructions.
- **Example:** Accessing data through a memory location that contains a pointer to the actual data's location.

Execute Cycle

- **Use:** Performing the operation specified by the decoded instruction.
- **Application:** Where the actual computation or data manipulation occurs, making it the heart of instruction execution.
- **Example:** Carrying out arithmetic calculations, logical operations, data transfers, or any actions dictated by the instruction.

Interrupt Cycle

- **Use:** Handling external events or requests for interrupting the CPU's current execution.
- **Application:** Ensures prompt response to hardware or software events such as hardware interrupts, system calls, or exceptions, allowing the CPU to temporarily switch tasks.

- **Example:** Responding to a keyboard input interrupt, saving the CPU's current state, and invoking an interrupt service routine (ISR).

Why do we Need an Instruction Cycle?

- The instruction cycle of a computer system is necessary for understanding the flow of instructions and the execution of an instruction in a computer processor.
- It is responsible for the complete flow of instructions from the start of the computer system through its shutdown. The instruction cycle helps to understand the internal flow of the central processing unit, allowing any faults to be immediately resolved.
- It deals with a computer processor's basic operations and demands a detailed understanding of the many steps involved.
- All instructions for the computer processor system follow the fetch-decode-execute cycle.

Importance of Instruction Cycle

- The instructions are the basic activities conducted in the main memory of the central processing unit. That is why they are so crucial to the processor system.
- It's a set of stages that helps us to understand how instruction flows. The instruction cycle allows the computer processor to see the sequence of instructions from start to finish.
- It is common for all instruction sets to require a thorough understanding to perform all operations efficiently.
- The processing time of a programme can be easily calculated using the instruction cycle, which aids in determining the processor's speed.
- The processor's speed determines how many instructions can be executed simultaneously in the central processing unit.

Advantages of Instruction Cycle

- **Efficiency:** The fetch-decode-execute cycle, consisting of instruction cycles, allows CPUs to execute instructions sequentially and efficiently, ensuring that each instruction is processed in a well-defined manner.
- **Flexibility:** CPUs can handle a wide range of instructions, from arithmetic operations to data transfers, by following the execution cycle for each instruction type.
- **Control Flow:** The instruction cycle controls the flow of program execution, advancing to the next instruction after each cycle, allowing for precise execution and program control.
- **Responsiveness:** CPUs can quickly respond to external events and handle interrupts or exceptions using the interrupt cycle, making them versatile and suitable for various tasks.

Disadvantages of Instruction Cycle

- **Clock Speed:** The speed of instruction execution is often constrained by the system's clock speed, limiting the number of instructions that can be executed in a given time period.
- **Pipeline Stalls:** In pipelined architectures, where multiple instructions are processed simultaneously, issues like pipeline stalls can lead to inefficiencies if instructions depend on one another.
- **Resource Limitations:** CPU execution is subject to resource limitations, such as the availability of registers, memory access times, and cache sizes, which can affect performance.
- **Instruction Set Limitations:** CPUs are limited by their instruction set architectures (ISAs), which may not include certain specialized instructions or features required for specific applications.

- **Complexity:** The fetch-decode-execute cycle is an intricate process, and the complexity of instruction execution can lead to design challenges and potential errors in the processor's microarchitecture.

Micro-Operations Introduction

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
2. Subtraction
3. Increment
4. Decrement
5. 1's complement
6. 2's complement

Let's discuss these arithmetic micro-operations one by one.

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,

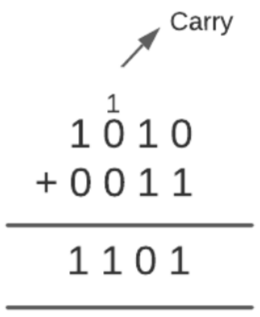
R3 is the desired register for storing the output.

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-



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,

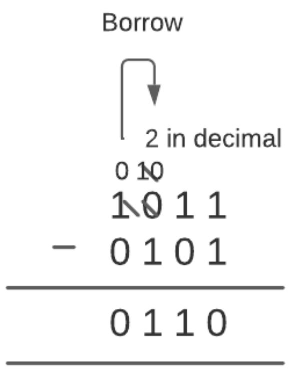
R3 is the desired register for storing the output.

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.

$$\begin{array}{r} 1\,0\,1\,0 \\ +\quad\quad 1 \\ \hline 1\,0\,1\,1 \\ \hline \end{array}$$

Finally, we will add R1 and R2.

$$\begin{array}{r} \nearrow\text{carry} \\ 1\,\overset{1}{0}\,\overset{1}{1}\,1 \\ +1\,0\,1\,1 \\ \hline 1\,0\,1\,1\,0 \\ \hline \end{array}$$

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

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.

$$\begin{array}{r} \nearrow\text{carry} \\ 0\,1\,\overset{1}{0}\,1 \\ +\quad\quad\quad 1 \\ \hline 0\,1\,1\,0 \\ \hline \end{array}$$

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

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.

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

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

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.

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.

What are the applications of arithmetic operations?

Microoperations are little operations performed on data kept in registers. An elementary operation carried out on data stored in one or more registers is known as a micro-operation. The operations carried out on the numerical data kept in the registers are covered by the arithmetic micro-operations. The content of information is altered via arithmetic microoperations.

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-

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = x.y$	$F \leftarrow A \wedge B$	AND
$F_2 = x.y'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	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
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (x.y)'$	$F \leftarrow \overline{A \wedge B}$	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.

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 V 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' V 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 registers 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 registers 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 V 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 registers 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** (De-multiplexers). 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.

Shift Micro-Operations in Computer Architecture

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.

Logical Shift

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

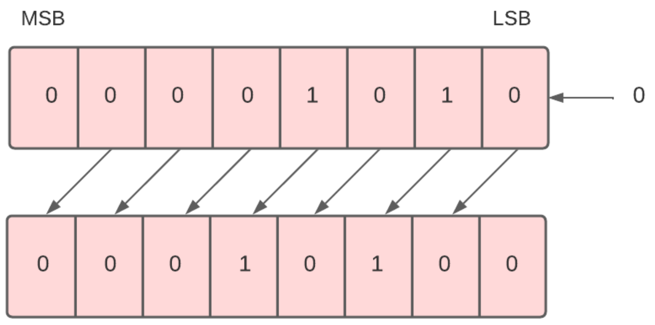
1. Logical Shift Left
2. Logical Shift Right

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.

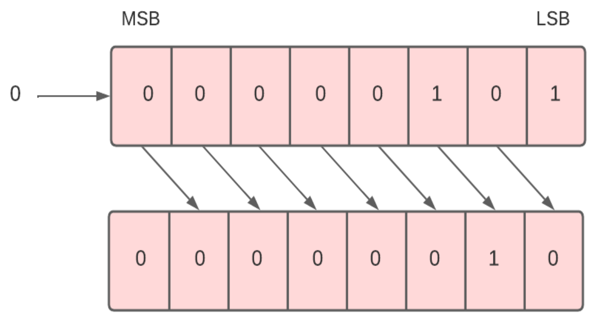
Moreover, the logical shift left microoperation denotes the multiplication of 2. The example we've taken above when converted into decimal forms the number 10. And the result after the logical shift operation when converted to decimal forms the number 20.

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.

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.

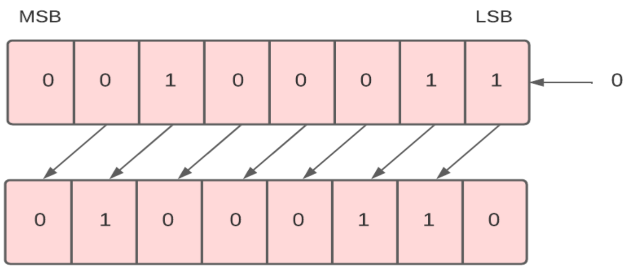
1. Arithmetic Shift Left
2. Arithmetic Shift Right

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.

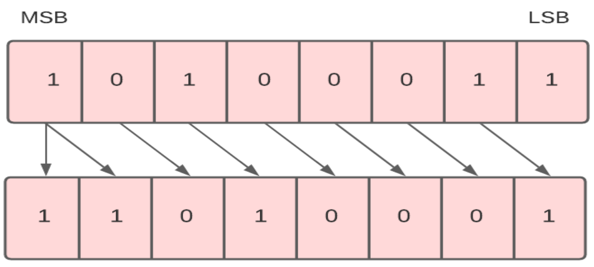
For example, R1 -> ashl R1.

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.

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.

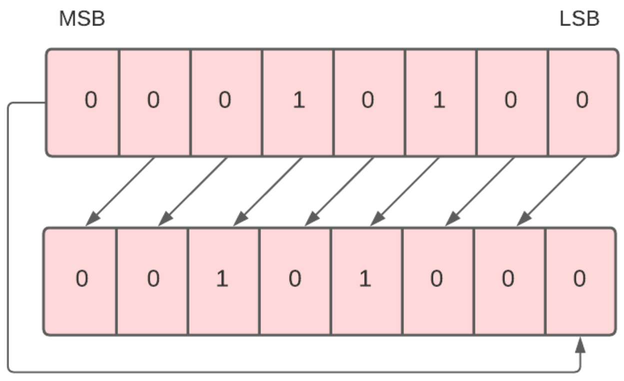
1. Circular Shift Left
2. Circular Shift Right

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.

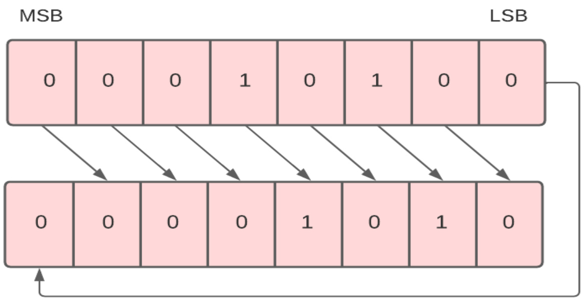
This command means the 8 bits present in the R1 register will be circular shifted left, and the result will be stored in register R1.

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.

The execution of a complete instruction

The execution of a complete instruction in a computer involves several stages, typically known as the instruction cycle or machine cycle. The instruction cycle consists of the following stages:

1. **Fetch (IF - Instruction Fetch):**
   - The instruction is fetched from memory. The program counter (PC) holds the address of the next instruction to be executed.
   - The contents of the memory location pointed to by the program counter are read, and the instruction is transferred to the instruction register (IR).
2. **Decode (ID - Instruction Decode):**
   - The instruction in the instruction register is decoded to determine what operation needs to be performed and what operands are involved.
   - The control unit generates the necessary control signals to coordinate the execution of the instruction.
3. **Execute (EX - Execution):**
   - The actual operation (arithmetic, logic, data transfer, etc.) specified by the instruction is performed.
   - If the instruction involves the manipulation of data, the data may be fetched from registers or memory, and the operation is carried out.
4. **Memory Access (MEM - Memory Access):**
   - If the instruction involves accessing memory (e.g., loading or storing data), the necessary memory operations are performed.
   - This stage is often skipped for instructions that don't involve memory access.
5. **Write Back (WB - Write Back):**
   - The results of the execution or the final data are written back to the registers or memory, depending on the nature of the instruction.
   - This stage ensures that the updated information is stored for future use.

These stages collectively form a complete cycle, and the process repeats for each instruction in a program. It's important to note that some instructions might take multiple cycles to complete, while others can be executed in a single cycle. The efficiency of instruction execution and the overall performance of a computer system are influenced by factors such as the design of the instruction set, the architecture of the processor, and the organization of the memory hierarchy.

Program Control (PC)

Program control in the context of digital systems and computer architecture refers to the management and coordination of the execution flow of instructions within a program. It is the instruction that alters the sequence of the program's execution, which means it changes the value of the program counter, due to which the execution of the program changes.

This involves the control unit's ability to fetch, decode, and execute instructions in a sequential and organized manner. The primary components related to program control are the program counter, instruction register, and the control signals generated by the control unit.

Features:

- These instructions cause a change in the sequence of the execution of the instruction.
- This change can be through a condition or sometimes unconditional.
- Flags represent the conditions.
- Flag-Control Instructions.
- Control Flow and the Jump Instructions include jumps, calls, returns, interrupts, and machine control instructions.
- Subroutine and Subroutine-Handling Instructions.
- Loop and Loop-Handling Instructions.

Here are key aspects related to program control:

1. **Program Counter (PC):**
    - The program counter is a special-purpose register that keeps track of the memory address of the next instruction to be fetched.
    - After each instruction is fetched, the program counter is incremented to point to the next sequential instruction.
2. **Instruction Register (IR):**
    - The instruction register holds the currently fetched instruction.
    - During the execution cycle, the control unit decodes the instruction stored in the IR to determine the operation to be performed.
3. **Control Signals:**
    - The control unit generates control signals based on the decoded instruction and the current state of the processor.
    - These control signals coordinate the activities of various components, such as the ALU, registers, and memory, to execute the instruction.
4. **Instruction Cycle:**
    - The process of fetching, decoding, and executing an instruction is collectively known as the instruction cycle.
    - The program counter is updated to point to the next instruction, and the cycle repeats until the program is completed.
5. **Branching and Jump Instructions:**
    - Branching instructions alter the normal sequential flow of instruction execution based on certain conditions (e.g., conditional branches).
    - Jump instructions transfer control to a different part of the program, often specified by an absolute address or an offset.
6. **Interrupts and Exceptions:**
    - Program control also involves handling interrupts and exceptions. These are events that temporarily suspend the normal execution flow to handle specific conditions or events.
7. **Pipelining and Superscalar Execution:**
    - Advanced processors use techniques like pipelining and superscalar execution to overlap the execution of multiple instructions, improving throughput and efficiency.

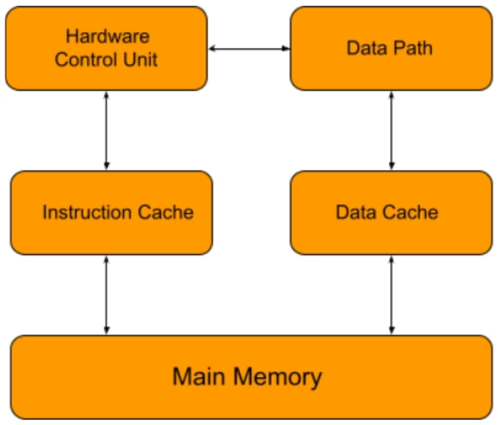
Program Control Instructions	Description
Branch (BR)	Branch which means it is an unconditional jump. It is unconditional branching wherever we specify the address we need to branch.

Skip (SKP)	Skip instructions is used to skip one(next) instruction. It can be conditional or unconditional. It does not need an address field. In the case of conditional skip instruction, the combination of conditional skip and an unconditional branch can be used as a replacement for the conditional branch.
Jump (JMP)	The jump instruction transfers the program sequence to the memory address given in the operand based on the specified flag.
Compare (CMP)	The Compare instruction performs a comparison via a subtraction, with difference not retained. CMP compares register sized values, with one exception.
CALL and RETURN	The CALL and RETURN instructions interrupt the flow of a program by passing control to an internal or external subroutine. An external subroutine is another program. The RETURN instruction returns control from a subroutine back to the calling program and optionally returns a value.
TEST	TEST instructions perform the AND of two operands without retaining the result, and so on.

In summary, program control is a crucial aspect of digital systems as it ensures that instructions are executed in the correct sequence and that the processor responds appropriately to various conditions and events during program execution. The coordination of the program counter, instruction register, and control signals is essential for the proper functioning of a computer system.

Reduced Instruction Set Computer (RISC)

A Reduced Instruction Set Computer (RISC) is a type of microprocessor architecture that uses a small, highly optimized set of instructions with a focus on providing high-performance computing by optimizing the instruction execution cycle. The key characteristics of RISC architecture include simplicity, a small and fixed instruction set, and a high clock speed. Here are some of the fundamental features of RISC architecture:



1. **Simplicity and Regularity:**
   - RISC architectures are designed to have a simple and regular instruction set.
   - Instructions are generally of uniform length, making it easier to decode and execute them.

2. **Small Instruction Set:**
    - RISC processors typically have a small number of instructions compared to Complex Instruction Set Computers (CISC).
    - The reduced instruction set allows for a more straightforward and faster instruction execution.
3. **Load-Store Architecture:**
    - RISC architectures often use a load-store architecture, where operations only involve data stored in registers. Memory operations are handled through load and store instructions.
    - Arithmetic and logic operations are performed between data in registers, promoting a more efficient use of the processor.
4. **Single-Cycle Execution:**
    - RISC processors aim for single-cycle execution for most instructions, leading to faster instruction throughput.
    - Instructions are generally executed in one clock cycle, simplifying the pipeline design and improving performance.
5. **Register Usage:**
    - RISC architectures heavily rely on a large number of general-purpose registers. Register-to-register operations are common.
    - The availability of more registers reduces the need for frequent data movement between registers and memory.
6. **Pipeline Design:**
    - RISC processors often employ pipeline architectures to overlap the execution of multiple instructions.
    - Instructions are divided into stages, allowing different stages of different instructions to be executed simultaneously.
7. **Efficient Compiler Support:**
    - RISC architectures are designed to work well with optimizing compilers.
    - The simplicity of the instruction set allows compilers to generate efficient machine code for the processor.
8. **High Clock Speed:**
    - RISC architectures are known for their ability to achieve high clock speeds, contributing to better overall performance.
9. **Reduced Complexity:**
    - By simplifying the instruction set and focusing on basic operations, RISC architectures reduce the complexity of the control unit and the overall processor design.

RISC instruction set

In RISC, the length of the instruction format is fixed. It took one-word memory. The fixed size of the instruction format benefits the program counter as it knows that the next instruction starts from where due to the fixed length of all instructions. In RISC, each instruction requires only one clock execution cycle. In addition, RISC architectures are designed to be highly scalable and accommodate a more significant number of instructions.

Advantages of RISC architecture

The advantages of RISC architecture are as follows:

- **Simplified instruction set:** RISC architecture uses a small instruction set that is highly optimized and simple; instruction executes quickly.
- **Format length is fixed:** In RISC architecture format length of instruction is fixed, which makes the execution or decoding of instructions faster.
- **Register-based architecture:** It stores data in the register within the processor, which is frequently used. This improves the performance because it reduces the number of memory access.
- **Fewer cycles:** In RISC architecture, the instruction requires a smaller number of cycles to execute because of the simple instruction set.
- **Load-Store architecture:** RISC architecture uses load-store architecture, which means memory access differs from logical and arithmetic operations. Therefore, the processor’s resources are used more efficiently, improving performance.

Disadvantages of RISC architecture

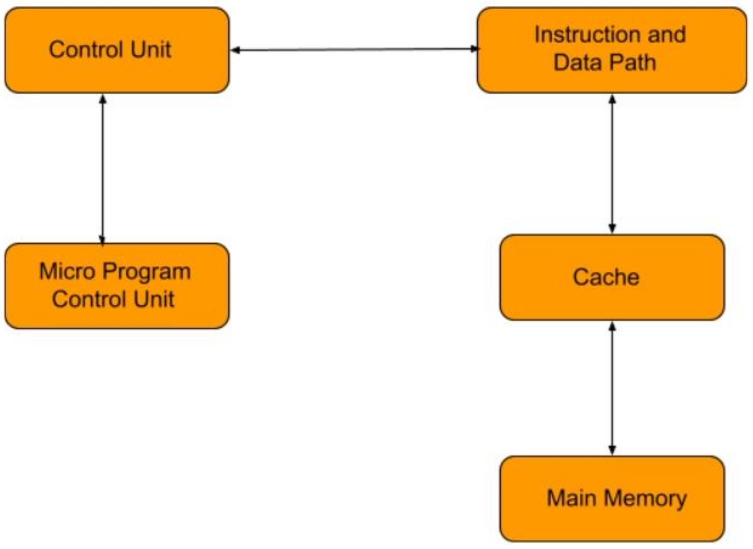
The disadvantages of RISC architecture are as follows:

- **Complex instructions and addressing modes:** It isn't easy to process complex instructions and complex addressing modes in the RISC architecture.
- **Direct memory-to-memory transfer:** It uses load-store architecture. Hence, it doesn't allow a direct memory-to-memory transfer.
- **Increase in the program length:** RISC architecture has a small and simple instruction set. However, it requires more instruction to operate CISC architecture, increasing the program's length.

Popular RISC architectures include ARM, MIPS, and PowerPC. RISC processors are commonly found in applications requiring high-performance computing, such as embedded systems, networking equipment, and mobile devices.

Complex Instruction Set Computer (CISC)

A Complex Instruction Set Computer (CISC) is a type of computer architecture where a single instruction can execute multiple low-level operations or perform a complex operation that might take several instructions in other architectures. The main idea behind CISC architectures is to provide a rich set of instructions that can perform diverse operations, aiming to reduce the number of instructions a programmer needs to write for a given task. Here are some key characteristics and features of CISC architectures:



1. **Rich Instruction Set:**
    - CISC architectures typically have a large and diverse set of instructions, ranging from simple arithmetic and logic operations to more complex operations.
    - Instructions can be quite powerful and perform operations that might require multiple instructions in a Reduced Instruction Set Computer (RISC) architecture.
2. **Variable-Length Instructions:**
    - Instructions in CISC architectures can have variable lengths. Some instructions may be one byte long, while others may be much longer.
    - Variable-length instructions allow the encoding of more complex operations but can make instruction fetching and decoding more complex.
3. **Memory Access Modes:**
    - CISC architectures often support multiple addressing modes for memory access, providing flexibility for programmers.
    - Addressing modes may include direct addressing, indirect addressing, indexed addressing, and others.
4. **Hardware-based Stack Manipulation:**
    - CISC architectures may include instructions specifically designed for stack manipulation, making it easier to implement high-level language constructs and function calls.
5. **Complex Instructions:**

    - CISC architectures often have complex instructions that can perform multi-step operations in a single instruction.
    - For example, a single CISC instruction might combine loading data from memory, performing arithmetic, and storing the result back in memory.
6. **Hardware Support for High-Level Language Constructs:**
    - CISC architectures often include instructions that directly support high-level language constructs, making it easier for compilers to generate efficient code.
7. **Microprogramming:**
    - Some CISC processors use microprogramming, where a complex instruction is implemented by a sequence of simpler microinstructions.
    - This allows for a more straightforward implementation of complex instructions at the hardware level.
8. **Examples of CISC Architectures:**
    - x86 architecture, including processors from Intel and AMD, is one of the most widely used examples of a CISC architecture.
    - Other historical examples include the VAX architecture and the Motorola 68k architecture.

CISC instruction set

CISC processors boast an extensive and varied instruction set tailored to handle diverse tasks. This encompasses arithmetic, logical, data movement, and control flow operations. The length of CISC instruction formats can vary due to the complexity of the operations. Each instruction comprises fields specifying the operation, operands, and addressing modes.

Advantages of CISC architecture

CISC architecture presents several advantages:

- **Versatility and rich instruction set:** A primary strength of CISC architecture lies in its comprehensive and diverse instruction set. This wealth of instructions streamlines coding for intricate operations, avoiding the need to deconstruct them into multiple simpler instructions.
- **Reduced code size:** Integrating multi-functional instructions can lead to shorter code sequences than RISC architectures. This can be advantageous in memory-constrained scenarios, as fewer instructions are necessary to accomplish a given task.
- **Programmer-friendly:** CISC architecture is often lauded for its capacity to execute complex operations using single instructions. This simplifies the programming process for developers engaged in intricate calculations or data manipulations.
- **Efficient memory use:** CISC processors facilitate memory-to-memory operations, allowing direct data manipulation without intermediate register transfers. This enhances memory space efficiency and potentially diminishes the demand for supplementary instructions.

Disadvantages of CISC Architecture

CISC architecture presents certain drawbacks:

- **Complex instruction decoding:** CISC architecture's extensive, variable-length instruction set necessitates intricate decoding logic. This complexity involves additional circuitry and time, potentially impeding overall processor performance.
- **Execution time variability:** Due to the varying lengths of CISC instructions, execution times for distinct instructions can differ significantly. This hinders accurate prediction of program execution time, affecting real-time applications requiring precise timing.
- **Pipelining challenges:** Effective pipelining, a technique that enhances processor performance by overlapping instruction execution, can be intricate in CISC architectures due to irregular instruction lengths and complex instructions.
- **Increased power consumption:** The intricacy of decoding and executing intricate instructions demands more power in CISC processors. This can result in higher energy consumption than simpler, more streamlined RISC architectures.

- **Limited compiler optimization:** The rich instruction set of CISC architectures can constrain compiler optimization efficacy. Compilers might encounter challenges in generating optimal code for such processors, potentially influencing overall performance.
- **Manufacturing complexity:** The intricate nature of CISC architecture, characterized by variable-length instructions and complex execution paths, can elevate the intricacy of processor design, manufacturing, and testing. This could lead to augmented costs and potential manufacturing hurdles.

While CISC architectures offer the advantage of reduced code size and more complex instructions, they can face challenges in terms of instruction decoding complexity, pipeline efficiency, and power consumption. In contrast, RISC architectures take a simpler approach with a smaller, fixed set of instructions designed for efficient execution, and they have gained popularity in many modern processors.

Pipelining

Pipelining is a technique used in computer architecture to enhance the throughput and efficiency of instruction execution in a processor. It involves breaking down the execution of instructions into several stages, with each stage performing a specific operation. In a pipelined architecture, different stages of multiple instructions can be overlapped, allowing the processor to work on several instructions simultaneously.
Here are the key concepts associated with pipelining:

1. **Stages of Instruction Execution:**
   - Instructions are divided into a sequence of stages, and each stage represents a specific operation in the instruction execution process.
   - Common pipeline stages include instruction fetch, instruction decode, execute, memory access, and write-back.
2. **Parallel Processing:**
   - Pipelining enables parallel processing of multiple instructions at different stages of the pipeline.
   - While one instruction is being executed, the next instruction can be fetched, and the one after that can undergo decoding, creating an overlapping or parallel execution.
3. **Pipeline Registers:**
   - Pipeline registers are used to store the intermediate results between stages.
   - These registers facilitate communication between the different stages of the pipeline.
4. **Increased Throughput:**
   - By allowing multiple instructions to be in different stages of execution simultaneously, pipelining increases the overall throughput of the processor.
   - The effective throughput is improved compared to non-pipelined architectures.
5. **Hazards in Pipelining:**
   - Data Hazard: Dependencies between instructions where the result of one instruction is needed by another.
   - Control Hazard: Situations where the control flow of the program affects the pipeline, such as branch instructions.
6. **Data Forwarding:**
   - To overcome data hazards, data forwarding (or data bypassing) is used to transfer data directly from the output of one stage to the input of another, bypassing intermediate pipeline registers.
7. **Branch Prediction:**
   - To mitigate the impact of control hazards, branch prediction techniques are employed to predict the outcome of branch instructions before they are actually resolved.
8. **Pipeline Flush:**
   - In case of mis predicted branches or other hazards, the pipeline may need to be flushed to eliminate incorrect instructions from the pipeline.
9. **Instruction-Level Parallelism (ILP):**
   - Pipelining is a form of ILP, where multiple instructions are processed simultaneously.
   - Super pipelining and superscalar architectures are extensions of pipelining that further exploit ILP.
10. **Examples of Pipelined Architectures:**

    - Many modern processors, including those in personal computers, servers, and embedded systems, use pipelining to improve performance.
    - Pipelining is also utilized in graphics processing units (GPUs) and digital signal processors (DSPs).

Pipelining is a fundamental concept in computer architecture that contributes to the overall speed and efficiency of modern processors by allowing them to execute instructions in parallel.

Hardwired control and microprogrammed control

Hardwired control and microprogrammed control are two contrasting approaches to designing the control unit of a computer system. The control unit is responsible for coordinating and directing the operations of other components in the CPU, ensuring that instructions are fetched, decoded, and executed in the correct sequence.

Hardwired Control:

1. Definition:

- **Hardwired control** is a control unit design where the control signals and logic are implemented using combinational logic circuits (e.g., AND gates, OR gates, flip-flops).

2. Characteristics:

- **Fixed Circuitry:** The control signals are generated by fixed hardware circuits.
- **Dedicated Logic:** Each control signal is directly generated by dedicated logic circuits, and there is a one-to-one correspondence between an instruction and the control signal.
- **Speed:** Generally faster than microprogrammed control because there is no need for the additional step of reading control memory.

3. Advantages:

- **Speed:** Hardwired control can be faster in terms of execution since control signals are directly generated by the hardware.
- **Simplicity:** It tends to be simpler in terms of design complexity.

4. Disadvantages:

- **Limited Flexibility:** Changes or modifications to the control unit require changes to the hardware, which can be impractical.
- **Complexity for Complex Instruction Sets:** For complex instruction set architectures (CISC), hardwired control may become complex due to the large number of instructions and addressing modes.

Microprogrammed Control:

1. Definition:

- **Microprogrammed control** is a control unit design where the control signals are stored in a control memory, and the control unit fetches these control signals as a sequence of microinstructions.

2. Characteristics:

- **Control Memory:** The control signals are stored in a separate control memory or control store.
- **Microinstructions:** Each instruction is represented as a sequence of microinstructions stored in control memory.
- **Flexibility:** It provides more flexibility as changes to the control unit can be made by modifying the microinstructions in control memory.

3. Advantages:

- **Flexibility:** Easier to modify or update since changes are made to the control memory, allowing for greater flexibility in accommodating new instructions or changes.
- **Complex Instruction Sets:** Well-suited for complex instruction set architectures (CISC) where a large number of instructions and addressing modes are present.

4. Disadvantages:

- **Speed:** Can be slower compared to hardwired control due to the additional step of fetching microinstructions from control memory.
- **Complexity:** The control memory and microinstruction sequencing logic can introduce additional complexity.

Hybrid Approaches:

In some cases, computer systems may use a combination of hardwired and microprogrammed control to leverage the benefits of both approaches, referred to as a hybrid approach. This allows for flexibility in handling a variety of instructions while maintaining the speed advantages of hardwired control for frequently executed instructions.

micro-program sequencing:

Microprogram sequencing refers to the process of determining the sequence of microinstructions to be executed to perform a machine-level instruction. In a computer system with microprogrammed control, the control unit fetches a series of microinstructions from the control memory to execute a given instruction. The microinstructions specify the control signals and operations needed to execute each stage of the instruction.

Here is an overview of micro-program sequencing:

1. **Control Memory:**
    - Microinstructions are stored in a control memory (or control store) within the control unit. This control memory contains the microprograms that control the various operations of the computer system.
2. **Microinstruction Format:**
    - Each microinstruction typically includes fields for control signals, next address (address of the next microinstruction), and other control information.
    - Control signals activate specific components of the computer system, such as the ALU, memory, or input/output.
3. **Microinstruction Execution:**
    - The control unit fetches the microinstructions from the control memory based on the current state of the machine and the specific instruction being executed.
    - The microinstructions are executed in sequence to perform the necessary operations for the given instruction.
4. **Microprogram Counter:**
    - A microprogram counter (μPC) or address register keeps track of the address of the next microinstruction to be fetched from the control memory.
    - The microprogram counter is updated during the execution of each microinstruction.
5. **Sequencing Logic:**
    - The sequencing logic in the control unit determines the next address for the microprogram counter.
    - The next address is based on the current microinstruction, and it may depend on conditions such as the outcome of arithmetic/logic operations, status flags, or control signals.
6. **Conditional Branching:**
    - Micro-program sequencing often involves conditional branching, where the next microinstruction address is determined based on certain conditions.
    - Conditions may include results of comparison operations, status flags, or external signals.
7. **Branching Control Signals:**
    - Control signals associated with branching determine whether to take a branch or follow the sequential order of microinstructions.
8. **Example:**
    - For example, during the execution of an instruction, the microprogram sequencing might involve fetching microinstructions to perform operations such as fetching operands, executing arithmetic operations, storing results, and updating the program counter.
9. **Microinstruction Execution Cycle:**
    - The execution of a single microinstruction is often referred to as a microinstruction execution cycle.
    - During each cycle, the control unit fetches a microinstruction, updates the microprogram counter, and executes the specified control signals.

Microprogram sequencing is a critical aspect of microprogrammed control, and the design of effective sequencing logic ensures that instructions are executed correctly and efficiently. It allows for the precise control of the various components within a computer system at the microinstruction level.

Horizontal and vertical microprogramming

Horizontal and vertical microprogramming refer to two different approaches in designing the control memory of a microprogrammed control unit. These approaches determine how microinstructions are organized and stored in the control memory.

1. Horizontal Microprogramming:

Definition:

- In horizontal microprogramming, each bit or field of a microinstruction corresponds to a control signal or a specific action.

Characteristics:

- **Parallelism:** Horizontal microinstructions are often wide, and multiple control signals are executed simultaneously.
- **Bit-Slice Microinstructions:** Each bit-slice represents a specific control signal, and several bits together form a microinstruction.
- **Multiple Control Lines:** Each bit in the microinstruction corresponds to a specific control line in the computer system.

Advantages:

- Efficient for machines with parallel architectures.
- Suitable for simple control units with a small number of control signals.

Disadvantages:

- Requires a large number of bits for each microinstruction.
- Can lead to inefficient use of control memory for machines with sequential architectures.

2. Vertical Microprogramming:

Definition:

- In vertical microprogramming, each microinstruction is viewed as a sequence of micro-operations, and each bit position represents a control signal or a specific action at a particular time step.

Characteristics:

- **Sequential Execution:** Microoperations are executed sequentially, and each bit in the microinstruction represents a micro-operation for a specific clock cycle.
- **Bit-Parallelism:** Although the micro-operations are sequential, several microinstructions can be fetched in parallel.

Advantages:

- More compact representation of microinstructions.
- Efficient for machines with sequential architectures.
- Allows for better utilization of control memory.

Disadvantages:

- May require multiple cycles to execute complex instructions.
- Suitable for machines with sequential architectures but may not fully exploit parallelism.

Comparison:

- **Storage Efficiency:**
    - Horizontal microprogramming tends to be less space-efficient as it requires a larger number of bits for each microinstruction.
    - Vertical microprogramming is more space-efficient as it represents a sequence of micro-operations with each bit in a microinstruction.
- **Execution Efficiency:**
    - Horizontal microprogramming is more efficient for machines with parallel architectures as it can execute multiple control signals simultaneously.
    - Vertical microprogramming is more suitable for machines with sequential architectures as it represents the micro-operations executed in each clock cycle.
- **Complexity:**
    - Horizontal microprogramming may result in more complex control units due to the parallel execution of control signals.
    - Vertical microprogramming may result in a simpler control unit structure, especially for machines with sequential architectures.

The choice between horizontal and vertical microprogramming depends on the architecture of the target machine, the desired level of parallelism, and the efficiency of the control memory utilization.

Differences between Horizontal and Vertical Micro-programmed CU

There are various differences between the vertical programmed CU and horizontal programmed CU, which are described as follows:

Horizontal Micro-programmed CU	Vertical Micro-programmed CU
This control unit is able to support **longer control words.**	This control unit is able to support **shorter control words.**
In this CU, we don't need any type of additional hardware.	In this CU, we need additional hardware to generate the control signals. These types of hardware must be in the form of decoders.
Compared to the vertical micro-programmed control unit, this control unit **is less flexible.**	Compared to the horizontal micro-programmed control unit, this control unit is **more flexible.**
Compared to the vertical micro-programmed control unit, this control unit is **faster.**	Compared to the horizontal micro-programmed control unit, this control unit is **slower.**
Compared to the vertical micro-programmed control unit, this control unit makes **less use** of **ROM encoding.**	Compared to the horizontal micro-programmed control unit, this control unit makes **more use** of **ROM encoding** so that it can reduce the control world's length.
A higher degree of parallelism is allowed by the horizontal micro-programmed CU. If there are is 'n' number of degrees, n control signals will be enabled at a time.	The low degree of parallelism is allowed by the vertical micro-programmed CU. That means there can either be 0 or 1 degree of parallelism.
The horizontal microinstruction is used by the horizontal micro-programmed CU. Here control line is attacked with every bit of the control field.	The vertical microinstruction is used by the vertical micro-programmed CU. Here each action will be performed with the help of a code, and this code will be translated into the individual control signals with the help of a decoder.