QUESTION BANK

21) What is meant by Memory-Reference instructions? Explain.

A computer instruction is a set of commands that tells the computer hardware to perform a specific operation. Computer instructions are able to perform a variety of operations like data transfer, arithmetic operations, logical operations, control flow, I/O operations, etc.

Based on the functionality, the basic computer instructions can be classified into the following three major types:

1) Memory Reference Instructions
2) Register Reference Instructions
3) Input/Output Instructions

The memory reference instructions are used for manipulation of data stored in the computer memory. These instructions enable the computer's CPU to access and process the data stored in a specific memory address. Therefore, memory reference instructions provide a mean for writing data to memory or reading data from memory. These instructions allow the processor to communicate with the computer's memory system.

The format of memory reference instruction consists of three parts namely, opcode (operation code), operands, and addressing mode. Where, opcode represents the operation to be performed, while the operand represents the memory address where the data is stored. A memory reference instruction typically uses 12 bits to specify a memory address, 3 bits to specify the opcode, and 1 bit to identify the addressing mode.

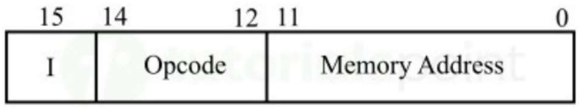


Figure - Memory Reference Instruction

The memory reference instructions are further classified into the following two types:

a) **Store Instruction** − The store instruction is used to write data into the memory.
b) **Load Instruction** − This instruction is used to retrieve data stored in the memory.

TABLE 5-4 Memory-Reference Instructions

Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], \quad E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

AND to AC

This is an instruction that performs the AND logic operation on pairs of bits in AC and the memory word specified by the effective address. The result of the operation is transferred to AC. The microoperations that execute this instruction are:

$$D_0T_4: \quad DR \leftarrow M[AR]$$
$$D_0T_5: \quad AC \leftarrow AC \wedge DR, \quad SC \leftarrow 0$$

D0T4: The control signal D0 indicates an AND operation. During timing signal T4, the memory word at the address specified in the Address Register (AR) is loaded into the Data Register (DR) (**DR ← M[AR]**).

D0T5: In the next step (T5), the AND operation is executed between the contents of the Accumulator (AC) and the Data Register (DR). The result is stored back in AC (**AC ← AC ^ DR**), and the Sequence Counter (SC) is reset to 0 to begin the next instruction cycle (**SC ← 0**).

<u>ADD to AC</u>

This instruction adds the content of the memory word specified by the effective address to the value of AC. The sum is transferred into AC and the output carry Cout is transferred to the E (extended accumulator) flip-flop. The microoperations needed to execute this instruction are:

$$D_1T_4: \quad DR \leftarrow M[AR]$$
$$D_1T_5: \quad AC \leftarrow AC + DR, \quad E \leftarrow C_{out}, \quad SC \leftarrow 0$$

D1T4: The control signal D1 indicates an ADD operation. During timing signal T4, the memory word at the address specified in the Address Register (AR) is loaded into the Data Register (DR) (**DR ← M[AR]**).

D1T5: In the next step (T5), the contents of the Accumulator (AC) and the Data Register (DR) are added. The sum is stored back in AC (**AC ← AC + DR**), the output carry is transferred to the E (extended accumulator) flip-flop (**E ← Cout**), and the Sequence Counter (SC) is reset to 0 to begin the next instruction cycle (**SC ← 0**).

<u>**LDA**</u>: Load to AC

This instruction transfers the memory word specified by the effective address to AC. The microoperations needed to execute this instruction are:

$$D_2T_4: \quad DR \leftarrow M[AR]$$
$$D_2T_5: \quad AC \leftarrow DR, \quad SC \leftarrow 0$$

D2T4: The control signal D2 indicates a LOAD operation. During timing signal T4, the memory word at the address specified in the Address Register (AR) is loaded into the Data Register (DR) (**DR ← M[AR]**).

D2T5: In the next step (T5), the contents of the Data Register (DR) are transferred to the Accumulator (AC) (**AC ← DR**), and the Sequence Counter (SC) is reset to 0 to begin the next instruction cycle (**SC ← 0**).

<u>**STA**</u>: Store AC

This instruction stores the content of AC into the memory word specified by the effective address. The microoperation needed to execute this instruction is:

$$D_3T_4: \quad M[AR] \leftarrow AC, \quad SC \leftarrow 0$$

D3T4: The control signal D3 indicates a STORE AC operation. During timing signal T4, the contents of the Accumulator (AC) are directly stored into the memory word at the address specified in the Address Register (AR) (**M[AR] ← AC**), and the Sequence Counter (SC) is reset to 0 to begin the next instruction cycle (**SC ← 0**).

<u>**BUN**</u>: Branch Unconditionally

This instruction is used to change the flow of the program to a new instruction address specified by the effective address. This allows the program to jump to a different part of the code, regardless of the current sequence. The microoperation needed to execute this instruction is:

$$D_4T_4: \quad PC \leftarrow AR, \quad SC \leftarrow 0$$

D4T4: The control signal D4 indicates a BUN operation. During timing signal T4, the effective address from the Address Register (AR) is transferred to the Program Counter (PC) (**PC ← AR**), and the Sequence Counter (SC) is reset to 0 to begin the next instruction cycle (**SC ← 0**). This causes the program to jump to the new instruction address specified in PC.

<u>**BSA**</u>: Branch and Save Return Address

This instruction is useful for branching to a portion of the program called a subroutine or procedure. When executed, the BSA instruction stores the address of the next instruction in sequence (which is available in the Program Counter (PC)) into a memory location specified by the effective address. The effective address plus one is then transferred to the PC to serve as the address of the first instruction in the subroutine. The microoperations needed to execute this instruction are:

$$\begin{aligned} &D_5T_4: && M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1 \\ &D_5T_5: && PC \leftarrow AR, \quad SC \leftarrow 0 \end{aligned}$$

D5T4: The control signal D5 indicates a BSA operation. During timing signal T4, the Program Counter (PC) is saved to the memory location specified by the Address Register (AR) (**M[AR] ← PC**), and the Address Register (AR) is incremented by 1 (**AR ← AR + 1**).

D5T5: During timing signal T5, the incremented Address Register (AR) value is transferred to the Program Counter (PC) (**PC ← AR**), and the Sequence Counter (SC) is reset to 0 to begin the next instruction cycle (**SC ← 0**).

<u>**ISZ**</u>: Increment and Skip if Zero

This instruction is used to increment a value in memory and possibly skip the next instruction based on the result. The value at the specified memory address is read into the Data Register (DR), incremented by 1, and written back to memory. If the incremented value becomes zero, the Program Counter (PC) is incremented by 1 to skip the next instruction in the program. This process allows the program to repeat an operation until a condition is met, typically starting from a negative value and incrementing until zero is reached. The microoperations needed to execute this instruction are:

$$\begin{aligned} &D_6T_4: && DR \leftarrow M[AR] \\ &D_6T_5: && DR \leftarrow DR + 1 \\ &D_6T_6: && M[AR] \leftarrow DR, \quad \textbf{if } (DR = 0) \textbf{ then } (PC \leftarrow PC + 1), \quad SC \leftarrow 0 \end{aligned}$$

D6T4: The control signal D6 indicates an ISZ operation. During timing signal T4, the memory word specified by the Address Register (AR) is read into the Data Register (DR) (**DR ← M[AR]**).

D6T5: During timing signal T5, the Data Register (DR) is incremented by 1 (**DR ← DR + 1**).

D6T6: During timing signal T6, the incremented value in DR is written back to the memory location specified by AR (**M[AR] ← DR**). If the value in DR is now zero, the Program Counter (PC) is incremented by 1 to skip the next instruction (**if DR = 0 then PC ← PC + 1**). The Sequence Counter (SC) is reset to 0 to begin the next instruction cycle (**SC ← 0**).

The execution process of memory reference instruction is as follows:

1) **Fetch the Instruction**: Firstly, the computer's CPU uses the program counter to fetch the instruction from the memory.
2) **Decode the Instruction**: This instruction is decoded to determine the opcode and operands.
3) **Fetch or Store Data**: If the opcode is for load instruction, the CPU accesses the memory location of the provided memory address to retrieve the stored data. If the opcode is for store instruction, the CPU writes the data into the provided memory address.
4) **Update the Program Counter**: Once the execution of the current memory reference instruction is completed, the CPU updates the program counter to execute the next memory reference instruction.

Example – IR register contains = 0001XXXXXXXXXXXX, i.e. ADD

Step 1: Fetch and Decode Instruction- The CPU retrieves the instruction from memory and determines it's a memory reference instruction for addition.

IR: 0001XXXXXXXXXXXX (Memory reference instruction for ADD)

Step 2: Fetch Data from Memory- The CPU fetches the data from the memory address specified by the Address Register (AR) and stores it in the Data Register (DR).

DR ← M[AR]

Step 3: Perform Addition- The CPU adds the contents of the Data Register (DR) to the Accumulator (AC).

AC ← AC + DR

Step 4: Update Status- The CPU sets the Status Code (SC) to 0, indicating successful completion of the operation.

SC ← 0

Hence, memory reference instructions are very important for many computing tasks like data processing, file handling, database operations, etc., as these computer instructions allow for accessing and manipulation of data stored in the computer memory.

22) Briefly explain the registers in basic computer with a neat diagram.

Flip-flops are fundamental building blocks in digital circuit design. They are capable of holding a single bit of information, which can be either 0 or 1. A register is a digital circuit component constructed by combining multiple flip-flops. Registers are much faster than accessing data from main memory. Storing frequently accessed data and intermediate results in registers reduces the need to access slower memory locations, thereby speeding up program execution. Registers are used for storing control information, such as instruction pointers, flags, and status bits, which are essential for managing the execution flow and state of the CPU during program execution.

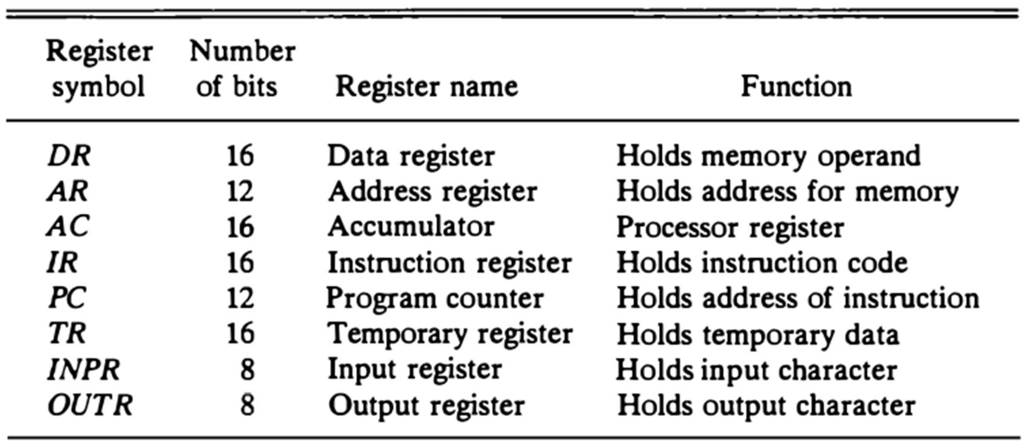


TABLE 5-1 List of Registers for the Basic Computer

Register symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

Data Register (DR):

The Data Register (DR) is a temporary storage unit within the CPU used to hold data being transferred in and out of the CPU. It holds data temporarily during processing. Data from memory or input/output devices is loaded into the DR for processing by the CPU. Results of arithmetic or logical operations are often stored back into the DR before being transferred elsewhere.

Address Register (AR):

The Address Register (AR) is a CPU register that holds the memory address of the data being referenced. It holds the address of the memory location currently being accessed or operated on. During memory read or write operations, the AR contains the address to be accessed. It facilitates the communication between the CPU and memory by specifying memory locations for data retrieval or storage.

Accumulator (AC):

The Accumulator (AC) is a special-purpose register within the CPU used for arithmetic and logic operations. It serves as the primary register for arithmetic and logic operations performed by the CPU. Results of arithmetic operations are often stored in the AC. It holds intermediate results during complex calculations.

Instruction Register (IR):

The Instruction Register (IR) is a CPU register that holds the current instruction being executed. It holds the opcode and operands of the instruction currently being executed. During the instruction fetch cycle, the IR stores the fetched instruction. It facilitates the decoding and execution of instructions by providing the necessary data and control signals to the CPU.

Program Counter (PC):

The Program Counter (PC) is a CPU register that stores the address of the next instruction to be fetched and executed. It keeps track of the memory address of the next instruction to be fetched from memory. During the instruction fetch cycle, the PC is incremented to point to the next instruction. It plays a crucial role in controlling the execution flow of the program by determining the sequence of instructions to be executed.

Temporary Register (TR):

The Temporary Register (TR) is a temporary storage unit used for holding intermediate data during CPU operations. It serves as a temporary storage location for data being transferred between different CPU components or during intermediate processing steps. It may hold data temporarily during arithmetic or logic operations before being stored back into memory or the accumulator. In some architectures, the TR may also be used for holding data during input/output operations.

Input Register (INPR):

The Input Register (INPR) is a register used for holding data received from input devices. It serves as a temporary buffer for holding data received from input devices such as keyboards, mice, or sensors. Input data is transferred from input devices to the INPR before being processed by the CPU. It facilitates the communication between input devices and the CPU by temporarily holding input data until it is processed.

Output Register (OUTR):

The Output Register (OUTR) is a register used for holding data to be sent to output devices. It serves as a temporary buffer for holding data to be sent to output devices such as displays, printers, or speakers. Output data processed by the CPU is transferred from memory or other registers to the OUTR before being sent to output devices. It facilitates the communication between the CPU and output devices by temporarily holding output data until it is transmitted.

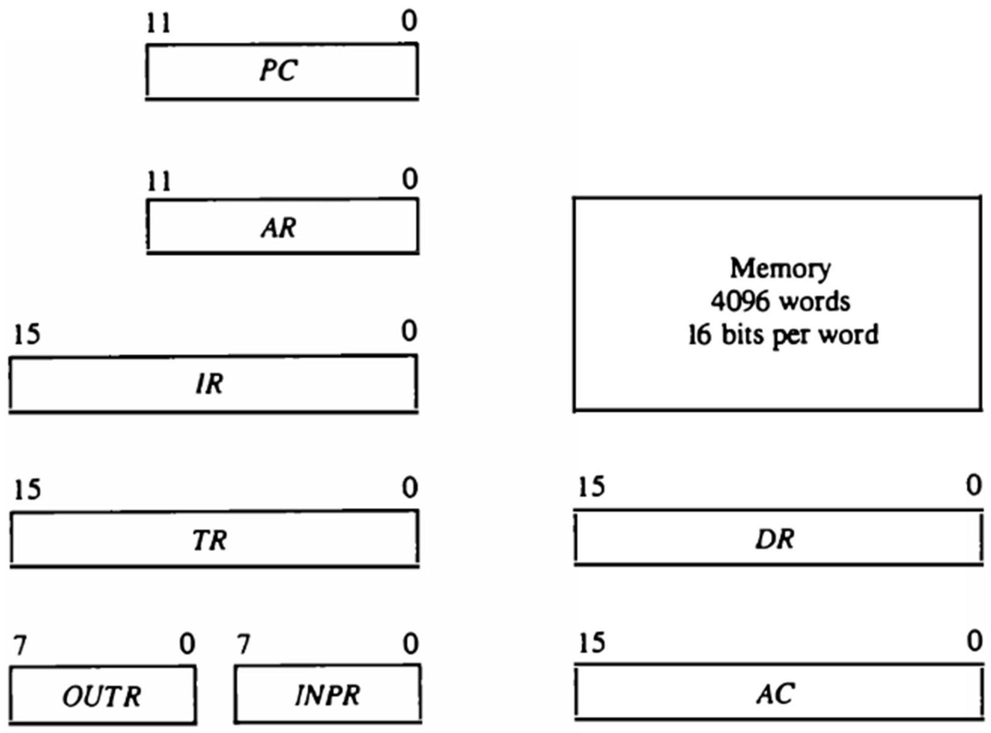
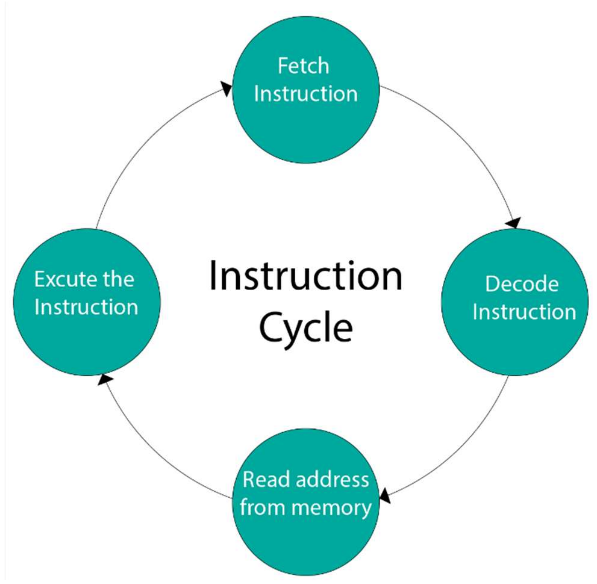


Figure 5-3 Basic computer registers and memory.

23) Describe Instruction cycle with a neat sketch.

The instruction cycle, also known as the fetch-decode-execute cycle, is a sequence of steps the CPU follows to execute instructions. The instruction cycle is the process by which a computer retrieves, decodes, and executes a program instruction. In the basic computer each instruction cycle consists of the following phases:

1) Fetch an instruction from memory.
2) Decode the instruction.
3) Read the effective address from memory if the instruction has an indirect address.
4) Execute the instruction.



1) **Fetch an instruction from memory**: The CPU fetches the next instruction from memory. The Program Counter (PC) holds the memory address of the next instruction to be executed. It points to the location in memory where the instruction is stored. The CPU accesses the memory location pointed to by the PC. The instruction stored at this memory address is then retrieved and loaded into the Instruction Register (IR) for further processing.

2) **Decode the instruction**: The CPU decodes the instruction stored in the IR. During the decode phase, the CPU identifies the opcode (operation code) of the instruction stored in the Instruction Register (IR). The opcode specifies the type of operation to be performed, such as arithmetic, logic, data movement, or control transfer. Once the opcode is identified, the CPU interprets the instruction based on the opcode and any associated operands. Depending on the instruction type, the CPU prepares to execute the appropriate operation specified by the opcode.

3) **Read the effective address from memory if the instruction has an indirect address**: This phase ensures that the CPU retrieves the correct memory address for the operand specified by the instruction. During the decode phase, the CPU determines if the instruction has an indirect addressing mode. In this mode, the memory address specified by the instruction does not directly contain the operand, but instead holds the address of the operand. If the instruction is determined to have an indirect address, the CPU performs an additional memory access to retrieve the actual operand address from the memory location specified by the indirect address. This operand address is then used in the subsequent execution phase to access the operand data from memory.

4) **Execute the instruction**: The CPU executes the instruction based on the opcode and any associated operands. During this phase, the CPU performs the operation specified by the opcode of the instruction. This may involve arithmetic calculations, logical operations, data movement, or control transfers, depending on the type of instruction. Once the operation is executed, the CPU handles the results appropriately. This may include storing the result in registers, updating memory locations, modifying flags or status registers, or preparing for the next instruction in the program sequence.

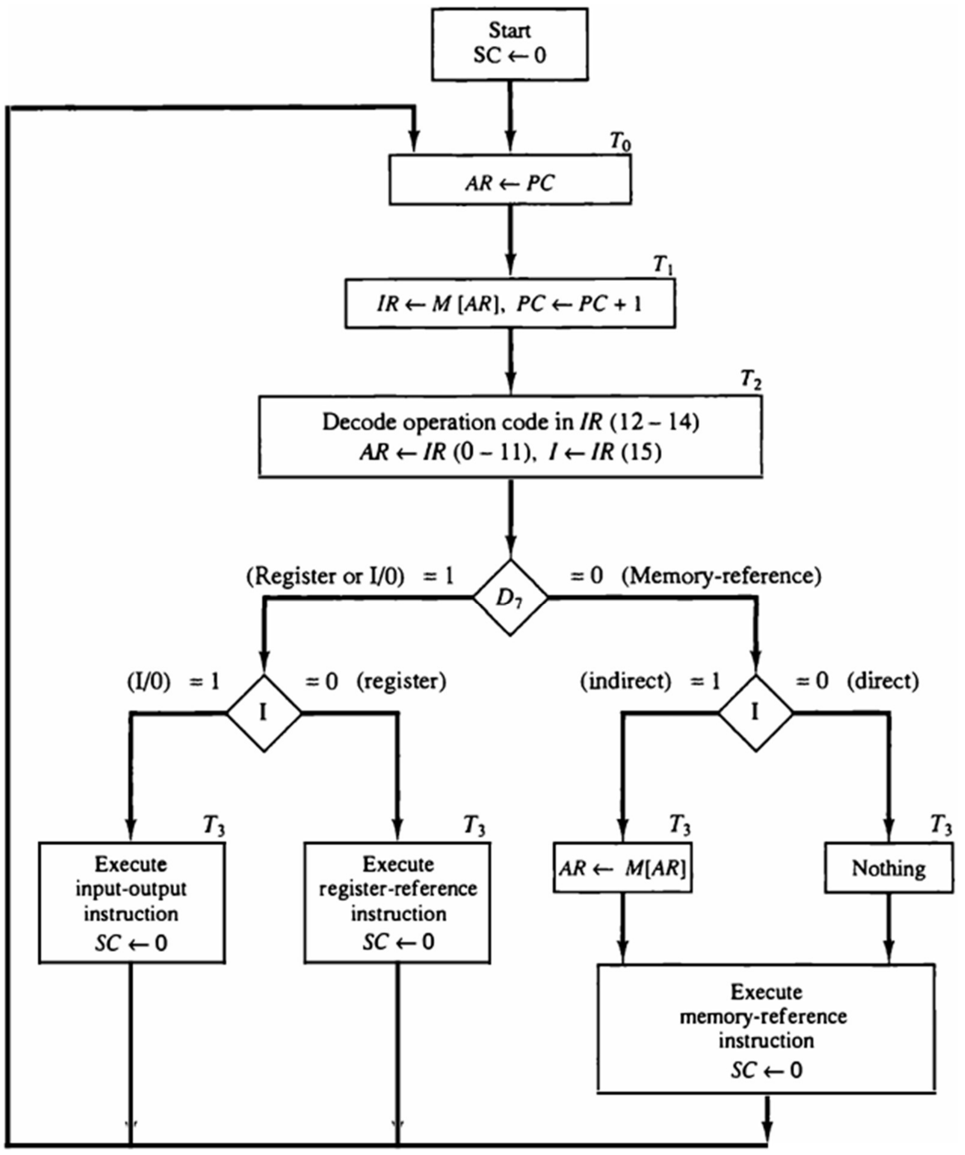

Start
SC ← 0
T0
AR ← PC
T1
IR ← M [AR], PC ← PC + 1
T2
Decode operation code in IR (12 – 14)
AR ← IR (0 – 11), I ← IR (15)
(Register or I/0) = 1
D7
= 0 (Memory-reference)
(I/0) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
T3
Execute input-output instruction SC ← 0
T3
Execute register-reference instruction SC ← 0
T3
AR ← M[AR]
T3
Nothing
Execute memory-reference instruction SC ← 0

Figure 5-9 Flowchart for instruction cycle (initial configuration).

Start: The sequence counter (SC) is reset to 0 to begin the instruction cycle.

SC 🡐 0

T0 (Fetch the Instruction Address): The address of the next instruction is moved from the Program Counter (PC) to the Address Register (AR).

T0: AR 🡐 PC

T1 (Fetch the Instruction): The instruction is fetched from memory at the address in AR and loaded into the Instruction Register (IR). The Program Counter (PC) is incremented to point to the next instruction in memory.

T1: IR 🡐 M[AR], PC 🡐 PC + 1

T2 (Decode the Instruction): The operation code (opcode) in IR (bits 12-14) is decoded to determine the type of instruction. The Address Register (AR) is loaded with the address part of the instruction (bits 0-11). The I (indirect) bit is extracted and stored.

T2: Decode operation code in IR (12-14)

AR ← IR (0-11), I ← IR (15)

Determine Instruction Type: Check if the instruction is a Register or Input/Output instruction or a Memory Reference instruction. `(Register or I/O) = 1` indicates it is either a Register or I/O instruction. `Memory-reference = 0` indicates it is a Memory Reference instruction.

T3 (Execute Based on Instruction Type):

- If **D7 = 1 (Register or I/O Instruction)**:
  - If **I/O Instruction (I = 1)**:

    T3: Execute input-output instruction, SC ← 0

  - If **Register Reference Instruction (I = 0)**:

    T3: Execute register-reference instruction, SC ← 0

- If **Memory Reference Instruction (D7 = 0)**:
  - If **Indirect Addressing (I = 1)**:

    Read the effective address from memory.

    T3: AR ← M[AR]

    Continue to execute memory reference instruction.

    Execute memory reference instruction, SC ← 0

  - If **Direct Addressing (I = 0)**:

    Directly execute the memory reference instruction without any further operations.

    T3: Nothing (no further micro-operations needed here)

    Execute memory reference instruction, SC ← 0

This flowchart shows how the CPU goes through the instruction cycle by fetching, decoding, and then determining the type of instruction to execute. Each step involves specific register transfers and updates to ensure that the correct instruction is executed, whether it is a memory reference, register reference, or input/output instruction. The sequence counter (SC) is used to control the timing of each step and is reset to 0 at the end of each instruction execution to prepare for the next instruction cycle.

24) Explain with diagram about Computer Description.

The complete computer description includes both, the instruction cycle and the interrupt cycle. The interrupt cycle is essential for handling external events that require immediate attention, such as I/O operations or other asynchronous events. When an interrupt occurs, the CPU suspends its current operation and enters the interrupt cycle to address the interrupt request. The instruction cycle is the fundamental process by which the CPU executes program instructions stored in memory. In the absence of interrupts, the CPU continuously executes instructions in a loop, known as the instruction cycle, until instructed otherwise. Each instruction is fetched from memory, decoded, and executed sequentially, one after the other. The CPU remains busy processing instructions during the instruction cycle, maximizing computational efficiency.

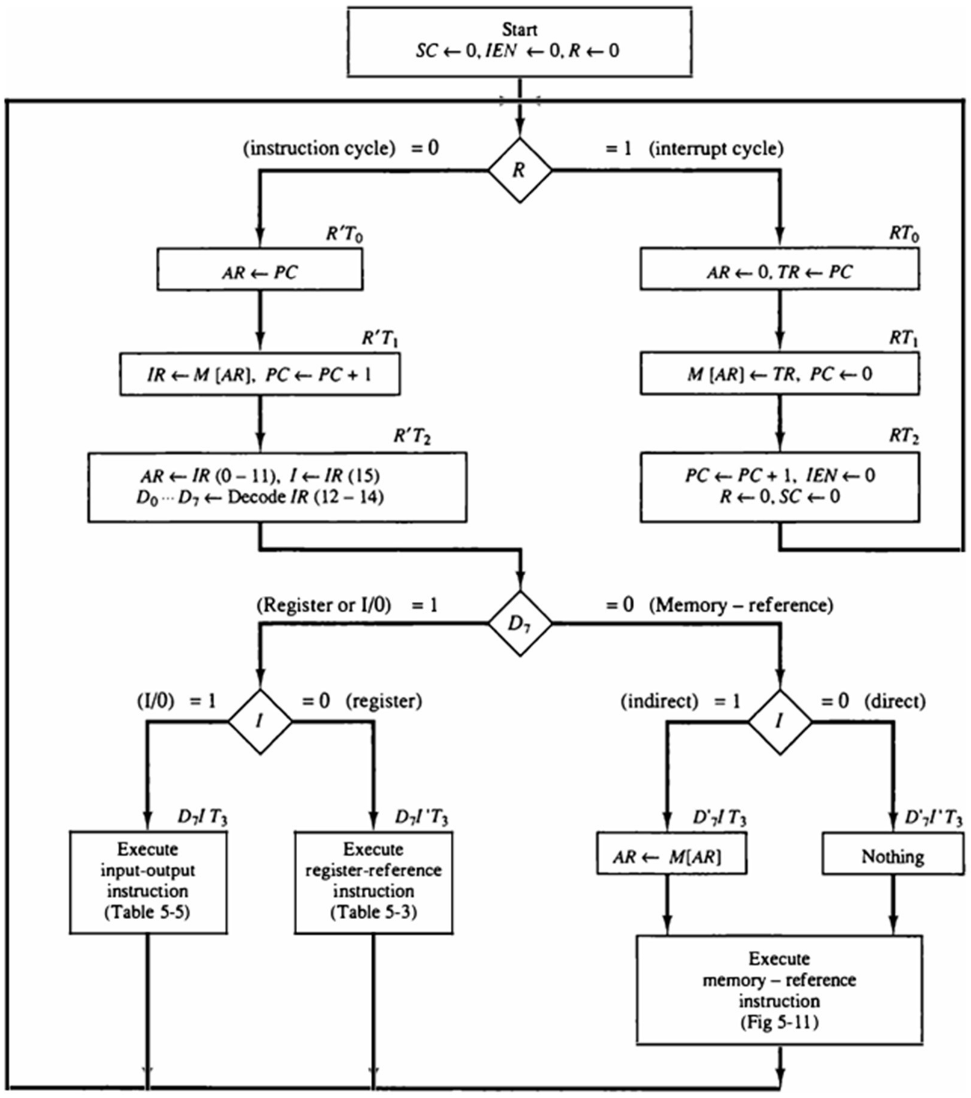

Start
$SC \leftarrow 0, IEN \leftarrow 0, R \leftarrow 0$
(instruction cycle) = 0
R
= 1 (interrupt cycle)
$R'T_0$
$AR \leftarrow PC$
$R'T_1$
$IR \leftarrow M[AR],\ PC \leftarrow PC + 1$
$R'T_2$
$AR \leftarrow IR(0-11),\ I \leftarrow IR(15)$
$D_0 \cdots D_7 \leftarrow$ Decode $IR(12-14)$
RT_0
$AR \leftarrow 0, TR \leftarrow PC$
RT_1
$M[AR] \leftarrow TR,\ PC \leftarrow 0$
RT_2
$PC \leftarrow PC + 1,\ IEN \leftarrow 0$
$R \leftarrow 0, SC \leftarrow 0$
(Register or I/0) = 1
D_7
= 0 (Memory – reference)
(I/0) = 1
I
= 0 (register)
(indirect) = 1
= 0 (direct)
D_7IT_3
$D_7I'T_3$
D'_7IT_3
$D'_7I'T_3$
Execute input-output instruction (Table 5-5)
Execute register-reference instruction (Table 5-3)
$AR \leftarrow M[AR]$
Nothing
Execute memory – reference instruction (Fig 5-11)

Figure 5-15 Flowchart for computer operation.

Initialization:

- **SC 🡐 0**: This sets the Sequence Counter to 0, indicating the beginning of the instruction cycle.
- **IEN 🡐 0**: The Interrupt Enable flip-flop is set to 0, meaning interrupts are currently disabled.
- **R 🡐 0**: The Reset flip-flop is set to 0, implying the system is not in a reset state.

Instruction Cycle vs. Interrupt Cycle:

- **Instruction Cycle (R Instruction Cycle = 0)**: This state indicates that the CPU is actively executing an instruction.
- **Interrupt Cycle (R Instruction Cycle = 1)**: This state signifies that the CPU is processing an interrupt request.

Instruction Fetch and Execution:

- **Instruction Cycle = 0**:

  R'T0: Transfer the content of Program Counter (PC) to Address Register (AR). This step involves copying the value stored in the Program Counter (PC), which holds the memory address of the next instruction to be executed, into the Address Register (AR). The AR will hold this memory address for fetching the instruction from memory.

  AR 🡐 PC

R'T1: Fetch the instruction from memory at the address stored in AR and store it in Instruction Register (IR). After transferring the memory address from PC to AR, the CPU fetches the instruction stored at that memory address. This instruction is retrieved from memory and loaded into the Instruction Register (IR), which temporarily holds the fetched instruction for decoding and execution.

IR ← M[AR]

Once the instruction is fetched, the Program Counter (PC) is incremented by 1. This prepares the PC to point to the memory address of the next instruction to be fetched in the subsequent instruction cycle.

PC ← PC + 1

R'T2: Extract the address part of the instruction stored in IR (bits 0-11) and load it into AR. In this step, the CPU extracts the address part of the instruction stored in the Instruction Register (IR). Typically, the address part occupies bits 0 to 11 in the instruction. This address is then loaded back into the Address Register (AR) to prepare for memory access in subsequent steps.

AR ← IR (0 – 11)

The remaining bits of the instruction, typically bits 12 to 14, contain information about the type of operation to be performed. These bits are decoded into microoperations, represented as D0 through D7, which dictate the specific actions the CPU needs to take to execute the instruction.

D0…D7 ← Decode IR (12 – 14)

The interrupt bit (I) is set according to the value of the 15th bit in the Instruction Register (IR), determining whether an interrupt is pending or not.

I ← IR (15)

Based on the decoded microoperations, the CPU determines the type of instruction being executed. This can include instructions for register operations, input/output operations, or memory-reference operations.

Execute the instruction:

- If **D7 = 1 (Register or I/O Instruction)**:
  - ➢ If the instruction is an I/O operation **(I/0) = 1**:

    D7IT3: Execute input-output instruction.

  - ➢ If the instruction is a register operation **(register = 0)**:

    D7I'T3: Execute register-reference instruction.

- If **D7 = 0 (Memory-reference instruction)**:
  - ➢ If indirect addressing is indicated **(indirect = 1)**:

    D'7IT3: Fetch data indirectly from memory and load it into AR.

    AR ← M[AR]

    Execute memory reference instruction
  - ➢ If direct addressing is indicated **(direct = 0)**:

    D7I'T3: Nothing (no further micro-operations needed here).

    Execute memory reference instruction

Interrupt Handling:

- **Interrupt Cycle = 1**:

RT0 (Initialization and Saving State): Clear the Address Register (AR). Transfer the content of the Program Counter (PC) to the Temporary Register (TR).

AR ← 0, TR ← PC

RT1 (Saving State and Preparing for Reset): Store the content of the Temporary Register (TR) into memory at the address specified by AR. Reset the Program Counter (PC) to 0.

M[AR] ←TR, PC ← 0

RT2 (Finalization and Reset): Increment the Program Counter (PC) to point to the next instruction. Disable interrupts. Reset the CPU. Reset the Sequence Counter.

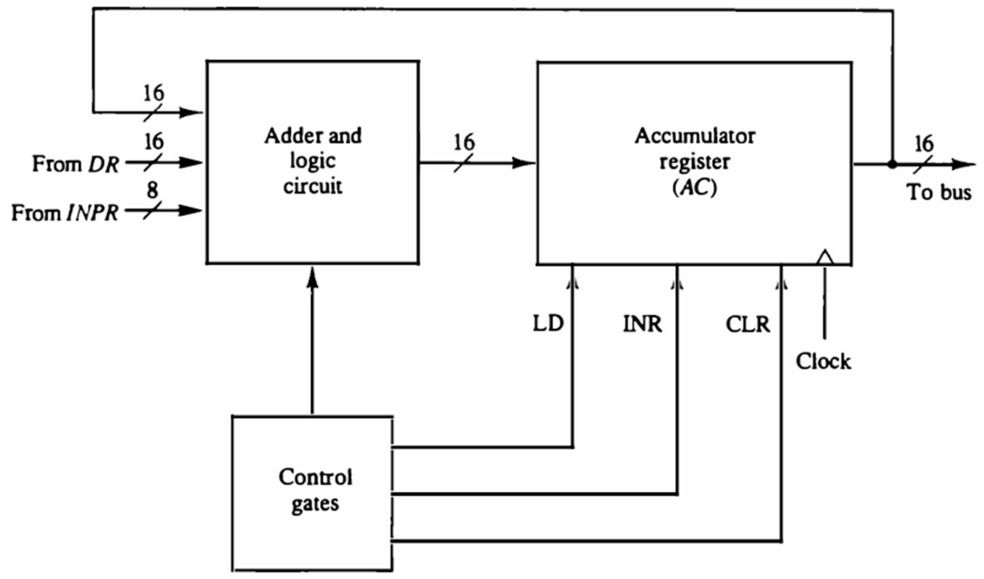
PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0

This flowchart shows the sequential steps involved in the operation of a CPU during the instruction cycle and interrupt handling. It serves as a visual representation of how the CPU fetches instructions, executes them, handles interrupts, and manages its internal state.

25) Describe the Design of Accumulator Logic.

Accumulator logic is a digital circuit design used for accumulating or summing binary numbers. It's commonly employed in arithmetic and control units within digital computers.

Figure 5-19 Circuits associated with AC.



The design considerations for the accumulator (AC) register within a digital circuit are as follows:

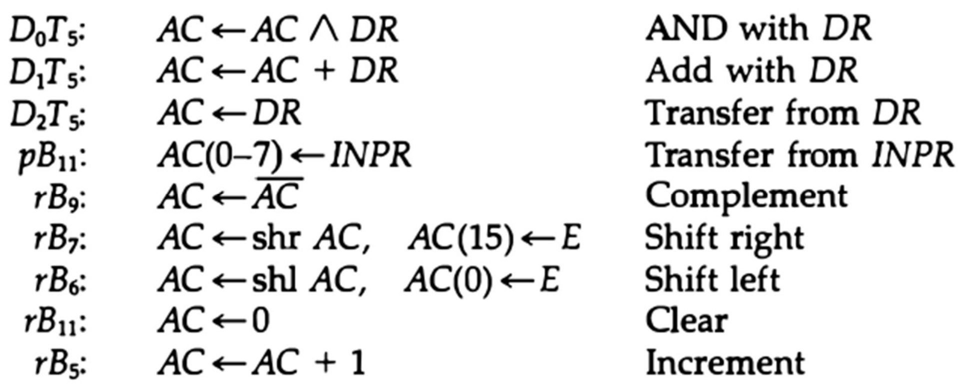
- **Inputs to the Adder and Logic Circuit**: The adder and logic circuit has three sets of inputs-
  - ➢ 16 inputs from the outputs of the AC register.
  - ➢ 16 inputs from the data register (DR).
  - ➢ 8 inputs from the input register (INPR).

- **Outputs of the Adder and Logic Circuit**: The outputs of this circuit provide the data inputs for the accumulator register.

- **Control Logic for AC Register**:
  - ➢ **LD**: Load control - Indicates when to load data into the accumulator.
  - ➢ **INR**: Increment control - Specifies when to increment the accumulator content.

- **CLR**: Clear control - Signals when to clear the accumulator content.

- **Register Transfer Statements**: The below table lists various register transfer statements that change the content of the accumulator (AC). These statements define the operations to be performed on the AC register.



D_0T_5:	$AC \leftarrow AC \wedge DR$	AND with *DR*
D_1T_5:	$AC \leftarrow AC + DR$	Add with *DR*
D_2T_5:	$AC \leftarrow DR$	Transfer from *DR*
pB_{11}:	$AC(0\text{–}7) \leftarrow INPR$	Transfer from *INPR*
rB_9:	$AC \leftarrow \overline{AC}$	Complement
rB_7:	$AC \leftarrow \text{shr } AC, \quad AC(15) \leftarrow E$	Shift right
rB_6:	$AC \leftarrow \text{shl } AC, \quad AC(0) \leftarrow E$	Shift left
rB_{11}:	$AC \leftarrow 0$	Clear
rB_5:	$AC \leftarrow AC + 1$	Increment

D0T5: This step involves performing a bitwise AND operation between the content of the accumulator (AC) and the content of the data register (DR). In a bitwise AND operation, each bit of the result is set to 1 only if both corresponding bits in AC and DR are 1; otherwise, the result bit is set to 0. This operation is useful for selectively preserving or filtering specific bits in AC based on the corresponding bits in DR.

AC ← AC ^ DR

D1T5: Here, we are adding the content of the data register (DR) to the accumulator (AC). Addition involves summing up the binary values stored in AC and DR. If the addition exceeds the capacity of the binary representation, overflow may occur.

AC ← AC + DR

D2T5: This step directly transfers the content of the data register (DR) into the accumulator (AC). Whatever value is stored in DR replaces the content of AC entirely. This operation is useful for copying data from one register to another.

AC ← DR

pB11: Transfer the content of bits 0-7 of the input register (INPR) to bits 0-7 of the accumulator (AC). This step selectively copies the lower byte of data from INPR into the corresponding positions in the accumulator (AC).

AC(0 – 7) ← INPR

rB9: Compute the one's complement of the content of the accumulator (AC). Each bit in AC is inverted, turning 0s into 1s and vice versa. This operation effectively computes the one's complement of AC.

AC ← ~AC

rB7: Right shift the content of the accumulator (AC) by one position. Each bit in AC is shifted one position to the right, effectively dividing its value by 2. The most significant bit (AC (15)) is replaced by the value stored in register E.

AC ← shr AC, AC (15) ← E

rB6: Left shift the content of the accumulator (AC) by one position. Each bit in AC is shifted one position to the left, effectively multiplying its value by 2. The least significant bit (AC(0)) is replaced by the value stored in register E.

AC ← shl AC, AC (0) ← E

rB11: Set the content of the accumulator (AC) to zero. All bits in AC are set to zero, effectively resetting or clearing the register.

AC ← 0

rB5: Increment the content of the accumulator (AC) by one. The value stored in AC is increased by one, effectively performing an increment operation.

AC ← AC + 1

26) Discuss about timing and control of basic computer.

In a basic computer system, the timing and control mechanisms ensure that all components operate in sync to perform the required tasks. The master clock generator produces clock pulses that are distributed to all the flip-flops and registers in the computer, including those in the control unit. These clock pulses are essential for synchronizing operations across the system. The clock pulses alone do not change the state of any register. A register will only change its state if it is enabled by a specific control signal generated by the control unit. This ensures that changes in the state of registers occur only when intended. The control unit generates control signals that perform several critical functions:

- Control inputs for multiplexers in the common bus
- Control inputs for processor registers
- Microoperations for the accumulator and other components

Types of Control Organization:

Hardwired Control

- Uses fixed wiring to generate control signals
- It's fast but hard to modify if changes are needed

Microprogrammed Control

- Uses a control memory that stores microinstructions
- Easier to modify since changes are done by updating the microinstructions

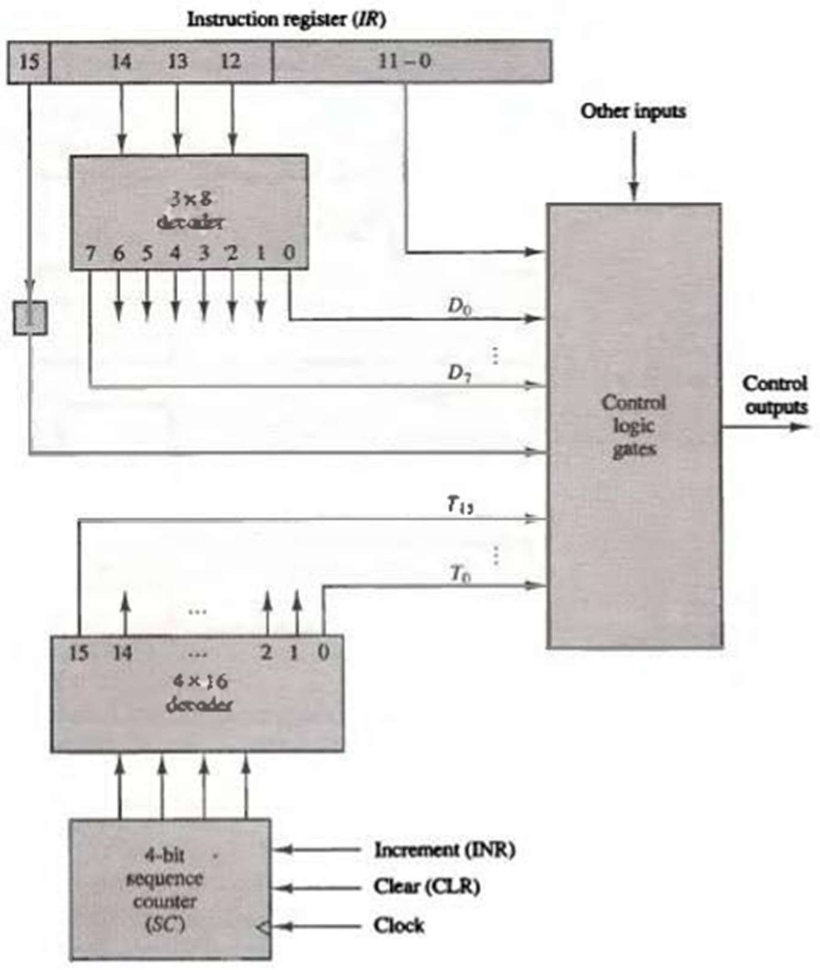


Figure 5-6 Control unit of basic computer.

This block diagram illustrates the internal workings of a basic computer's control unit, highlighting how instructions are decoded and how control signals are generated and synchronized using timing signals. The use of decoders and sequence counters ensures that each micro-operation occurs in the correct sequence, enabling the computer to execute instructions methodically and efficiently.

The Instruction Register (IR) is loaded with an instruction from memory. It holds the current instruction being executed. The bits 12-14 are used as inputs to a decoder. 3 x 8 decoder takes 3 bits from the IR (bits 12, 13 and 14) and decodes them into one of 8 output lines D0 to D7. This helps to identify which operation the instruction is. Control logic gates receives inputs from the 3x8 decoder and other inputs. It generates control signals for various parts of the CPU. 4-bit Sequence Counter (SC) keeps track of the steps in the execution cycle (like fetch, decode, execute). It can be incremented (INR), cleared (CLR), and is clock-driven. 4x16 Decoder takes the 4 bits from the sequence counter and decodes them into one of 16 output lines T0 to T15. This helps control the timing of the execution steps.

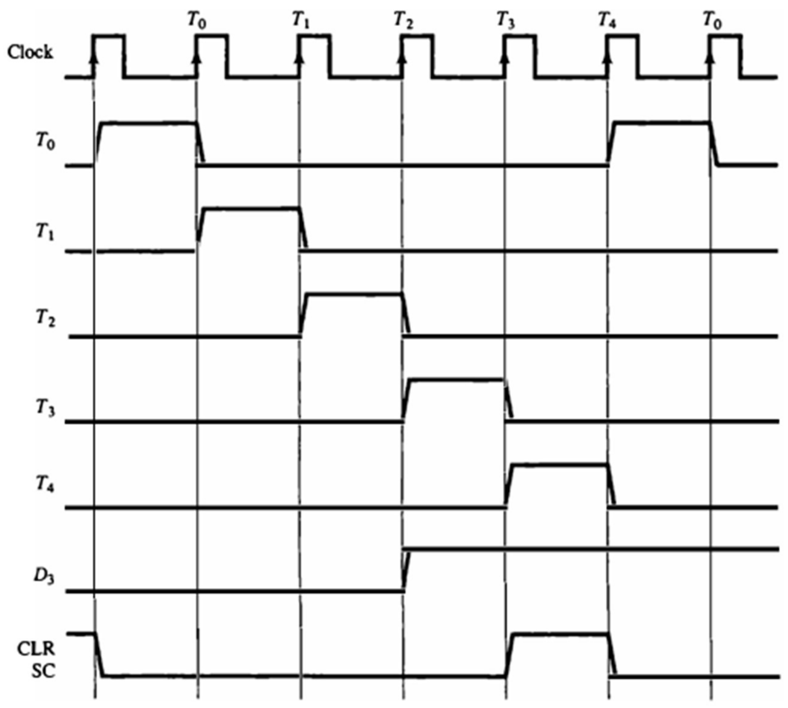


Figure 5-7 Example of control timing signals.

This diagram illustrates the sequence of control timing signals in a basic computer system. It shows how the control unit uses the clock signal and sequence counter to generate a series of timing signals (T0 to T4) that coordinate the steps of instruction execution. Each timing signal corresponds to a specific phase in the instruction cycle. The diagram also shows how certain conditions (e.g., D3 being active during T4) can trigger the sequence counter to reset, ensuring the control unit starts a new instruction cycle. The topmost waveform represents the clock signal, which consists of a series of regular pulses. The clock signal synchronizes the operation of all components in the computer system. Each positive edge of the clock pulse triggers the next step in the sequence. The waveforms labelled T0 through T4 represent specific timing signals generated by the 4x16 decoder based on the output of the sequence counter (SC). Each timing signal is active (high) for one clock cycle, and they are activated sequentially. These signals control the specific steps in the execution of an instruction.

Signal T0: Signal T0 actives during the first clock cycle. It triggers operations that need to occur in the first step of the instruction cycle, such as fetching the instruction from memory.

Signal T1: Signal T1 activates during the second clock cycle. It might control the loading of the fetched instruction into the instruction register (IR).

Signal T2: Signal T2 activates during the third clock cycle. It could be used for decoding the instruction or preparing operands.

Signal T3: Signal T3 activates during the fourth clock cycle. This signal might initiate the execution of the instruction, such as performing an ALU operation.

Signal T4: Signal T4 activates during the fifth clock cycle. It may complete the execution phase and store results or prepare for the next instruction cycle.

Opcode Signal (D3): The signal labelled D3 represents one of the decoded opcode signals from the 3x8 decoder. This signal is activated when the opcode in the instruction register corresponds to the specific instruction associated with D3. In the diagram, D3 is shown as active during timing signal T4.

Clear Signal (CLR): The signal labelled CLR represents the clear signal for the sequence counter (SC). When CLR is active, it resets the sequence counter to 0, starting the timing sequence again from T0. In the diagram, CLR is activated when D3 and T4 are both active, indicating the end of the current instruction execution and the start of the next instruction cycle.

Sequence Counter (SC): The waveform labelled SC shows the state of the 4-bit sequence counter. The sequence counter advances with each clock pulse, generating a new state that is decoded into the timing signals T0 to T15. When CLR is activated, the counter resets to 0.

Timing Relationships: At the beginning, the sequence counter SC is cleared (CLR is active), and T0 is the first timing signal to be active. The sequence counter increments with each clock pulse, causing the timing signals T0, T1, T2, T3, and T4 to become active in sequence. When D3 is active during T4, the CLR signal is activated, resetting the sequence counter to 0. This causes the timing sequence to restart from T0 on the next clock pulse.

27) Demonstrate input-output configuration.

Input-output (I/O) configuration is crucial for the interaction between the computer system and the external environment. I/O configuration refers to the hardware and software setup that allows the computer to communicate with peripheral devices like keyboards, mice, printers, storage devices, and network interfaces.

Components:

- **Terminal:** Sends and receives information, each unit being an 8-bit alphanumeric code (e.g., letter 'A' is an alphanumeric code).
- **Keyboard:** Input device that sends information to the computer.
- **Printer:** Output device that receives information from the computer and prints characters.
- **Input Register (INPR):** Stores 8-bit alphanumeric code received from the keyboard.
- **Output Register (OUTR):** Stores 8-bit alphanumeric code to be sent to the printer.
- **Communication Interface:** Handles serial communication with keyboard and printer.
- **Accumulator (AC):** General-purpose register within the CPU.
- **Input Flag (FGI):** 1-bit flag indicating new data in INPR (1) or data accepted by CPU (0).
- **Output Flag (FGO):** 1-bit flag indicating data ready in AC (1) or data sent to printer (0).

Data Flow:

1. **Keyboard Input:**
    - ➢ User presses a key.
    - ➢ 8-bit code representing the key is shifted serially into INPR.
    - ➢ Input Flag (FGI) is set to 1, indicating new data.
2. **CPU Checks Flag:**
    - ➢ CPU checks FGI.
    - ➢ If FGI is 1 (new data available), the information from INPR is transferred in parallel to the AC register.
    - ➢ FGI is cleared to 0, allowing new data entry.
3. **Output to Printer:**
    - ➢ Initially, Output Flag (FGO) is set to 1.
    - ➢ CPU checks FGO.
    - ➢ If FGO is 1 (ready to accept data), information from AC is transferred in parallel to OUTR.
    - ➢ FGO is cleared to 0, indicating data sent.
4. **Printer Prints:**
    - ➢ Printer receives the code from OUTR.
    - ➢ Printer interprets the code and prints the corresponding character.
    - ➢ Once printing is complete, the printer sets FGO back to 1, indicating readiness for new data.

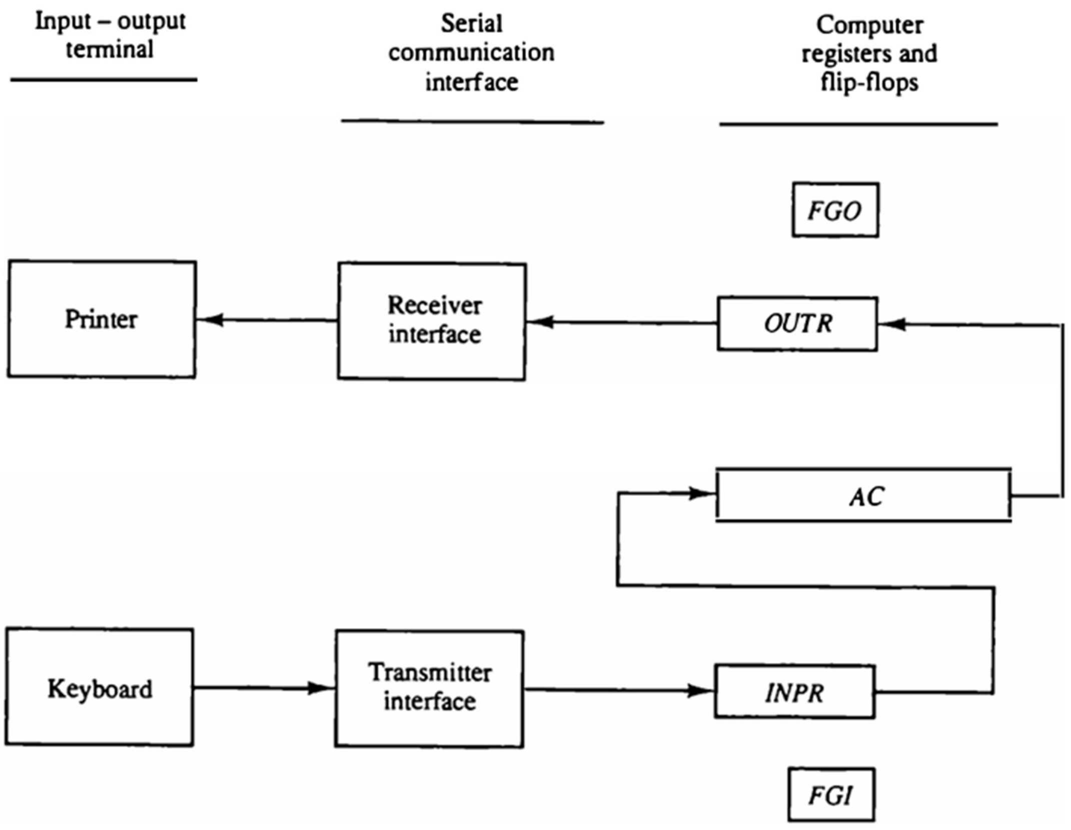
Synchronization:

- The flags (FGI and FGO) ensure synchronized data transfer between slow peripherals (keyboard, printer) and the faster CPU.
- FGI prevents overwriting data in INPR with a new key press before the CPU processes the current data.

- FGO prevents the CPU from sending new data to OUTR while the printer is still busy printing the previous character.

Overall, this configuration allows the computer to efficiently communicate with external devices like the keyboard and printer using serial communication and manages data flow with the help of flags for synchronization.

Figure 5-12 Input-output configuration.



The I/O terminal is a device that allows users to interact with the computer. The keyboard is an example of an I/O terminal. The receiver and transmitter interfaces are devices that allow the computer to communicate with external devices. For example, a receiver interface might be used to connect the computer to a printer, while a transmitter interface might be used to connect the computer to a monitor.

28) Briefly explain and design of a basic computer.

In the architecture of the basic computer, various hardware components play crucial roles in its operation. The memory unit serves as the storage space for data and instructions, offering a capacity of 4096 words, each consisting of 16 bits. These words hold the programs and data necessary for the computer's operation. Registers, including the Address Register (AR), Program Counter (PC), and Accumulator (AC), facilitate rapid data access and manipulation within the CPU. They store temporary data, addresses, and control information essential for executing instructions.

Flip-flops, such as the Instruction (I), Sign (S), and Reset (R) flip-flops, provide the fundamental state storage elements necessary for controlling various aspects of the computer's operation. They hold binary states representing specific conditions or signals, influencing the flow of operations. Additionally, flip-flops like Input Flag (FGI) and Output Flag (FGO) manage communication between the CPU and external devices, indicating the availability of input data and the readiness of the output device, respectively.

Decoders are critical in interpreting binary signals and translating them into specific outputs. The 3 x 8 operation decoder and the 4 x 16 timing decoder play roles in selecting operations and timing control within the system, respectively. They enable the CPU to execute instructions accurately and efficiently by decoding binary signals into meaningful actions.

The common bus acts as a communication pathway, allowing data and control signals to flow between different components of the computer system. With a width of 16 bits, it facilitates parallel transfer of data, enhancing the system's throughput and efficiency. Control logic gates form the backbone of the computer's control unit, generating control signals based on instruction decoding and system states. They coordinate the operation of various components, ensuring proper execution of instructions and smooth functioning of the system.

Finally, the adder and logic circuit connected to the Accumulator (AC) perform arithmetic and logical operations on data stored within the CPU. These operations include addition, subtraction, and logical operations like AND and OR. The results of these operations are then stored back into the Accumulator, ready for further processing or output. Together, these hardware components form the foundation of the basic computer architecture, enabling it to perform computation and process data effectively.

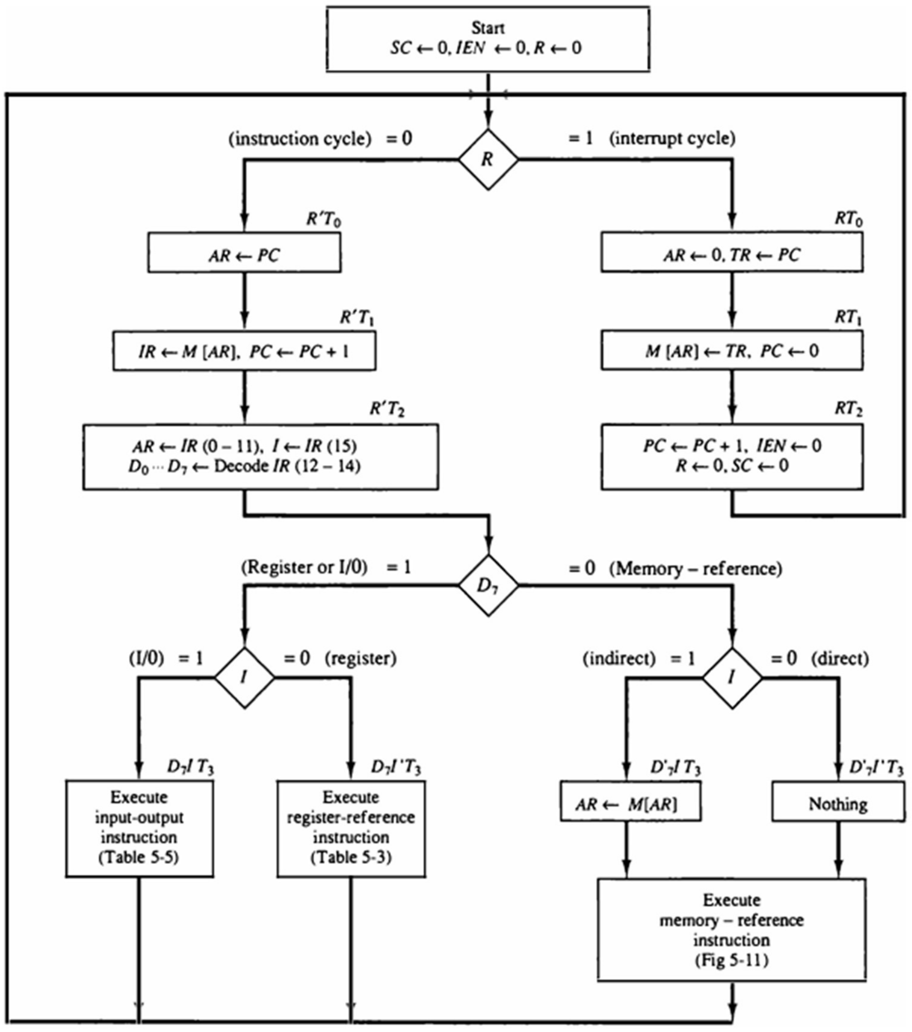


Figure 5-15 Flowchart for computer operation.

Initialization:

- **SC ← 0**: This sets the Sequence Counter to 0, indicating the beginning of the instruction cycle.
- **IEN ← 0**: The Interrupt Enable flip-flop is set to 0, meaning interrupts are currently disabled.
- **R ← 0**: The Reset flip-flop is set to 0, implying the system is not in a reset state.

Instruction Cycle vs. Interrupt Cycle:

- **Instruction Cycle (R Instruction Cycle = 0)**: This state indicates that the CPU is actively executing an instruction.
- **Interrupt Cycle (R Instruction Cycle = 1)**: This state signifies that the CPU is processing an interrupt request.

Instruction Fetch and Execution:

- **Instruction Cycle = 0**:

  R'T0: Transfer the content of Program Counter (PC) to Address Register (AR). This step involves copying the value stored in the Program Counter (PC), which holds the memory address of the next instruction to be executed, into the Address Register (AR). The AR will hold this memory address for fetching the instruction from memory.

  AR ← PC

  R'T1: Fetch the instruction from memory at the address stored in AR and store it in Instruction Register (IR). After transferring the memory address from PC to AR, the CPU fetches the instruction stored at that memory address. This instruction is retrieved from memory and loaded into the Instruction Register (IR), which temporarily holds the fetched instruction for decoding and execution.

  IR ← M[AR]

  Once the instruction is fetched, the Program Counter (PC) is incremented by 1. This prepares the PC to point to the memory address of the next instruction to be fetched in the subsequent instruction cycle.

  PC ← PC + 1

  R'T2: Extract the address part of the instruction stored in IR (bits 0-11) and load it into AR. In this step, the CPU extracts the address part of the instruction stored in the Instruction Register (IR). Typically, the address part occupies bits 0 to 11 in the instruction. This address is then loaded back into the Address Register (AR) to prepare for memory access in subsequent steps.

  AR ← IR (0 – 11)

  The remaining bits of the instruction, typically bits 12 to 14, contain information about the type of operation to be performed. These bits are decoded into microoperations, represented as D0 through D7, which dictate the specific actions the CPU needs to take to execute the instruction.

  D0…D7 ← Decode IR (12 – 14)

  The interrupt bit (I) is set according to the value of the 15th bit in the Instruction Register (IR), determining whether an interrupt is pending or not.

  I ← IR (15)

  Based on the decoded microoperations, the CPU determines the type of instruction being executed. This can include instructions for register operations, input/output operations, or memory-reference operations.

Execute the instruction:

- If **D7 = 1 (Register or I/O Instruction)**:
  - If the instruction is an I/O operation **(I/0) = 1**:

D7IT3: Execute input-output instruction.

- If the instruction is a register operation **(register = 0)**:

  D7I’T3: Execute register-reference instruction.

- If **D7 = 0 (Memory-reference instruction)**:
  - If indirect addressing is indicated **(indirect = 1)**:

    D'7IT3: Fetch data indirectly from memory and load it into AR.

    AR ← M[AR]

    Execute memory reference instruction
  - If direct addressing is indicated **(direct = 0)**:

    D7I'T3: Nothing (no further micro-operations needed here).

    Execute memory reference instruction

Interrupt Handling:

- **Interrupt Cycle = 1**:

  RT0 (Initialization and Saving State): Clear the Address Register (AR). Transfer the content of the Program Counter (PC) to the Temporary Register (TR).

  AR ← 0, TR ← PC

  RT1 (Saving State and Preparing for Reset): Store the content of the Temporary Register (TR) into memory at the address specified by AR. Reset the Program Counter (PC) to 0.

  M[AR] ←TR, PC ← 0

  RT2 (Finalization and Reset): Increment the Program Counter (PC) to point to the next instruction. Disable interrupts. Reset the CPU. Reset the Sequence Counter.

  PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0

This flowchart shows the sequential steps involved in the operation of a CPU during the instruction cycle and interrupt handling. It serves as a visual representation of how the CPU fetches instructions, executes them, handles interrupts, and manages its internal state.

29) Explain about program interrupt with an example.

A program interrupt is a mechanism that allows an external device (like a keyboard, printer, or network card) to temporarily halt the CPU's execution of the current program and get its attention. This is a much more efficient way for the CPU to handle slow peripherals compared to constantly checking their status.

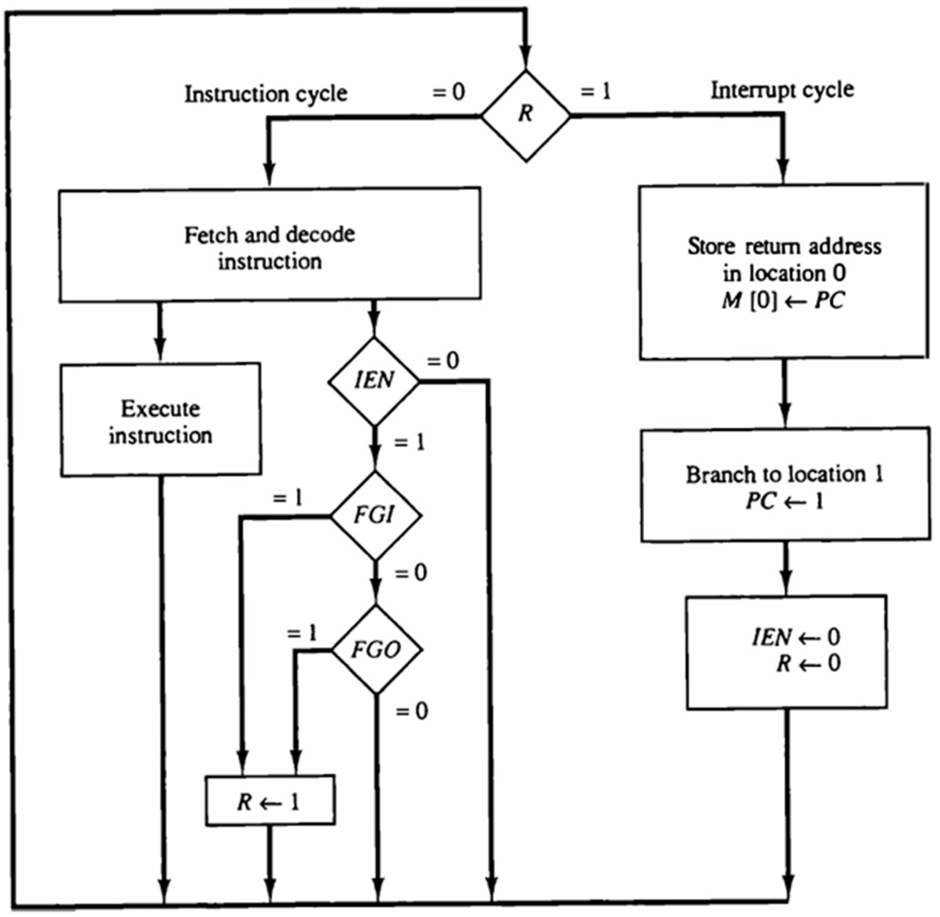
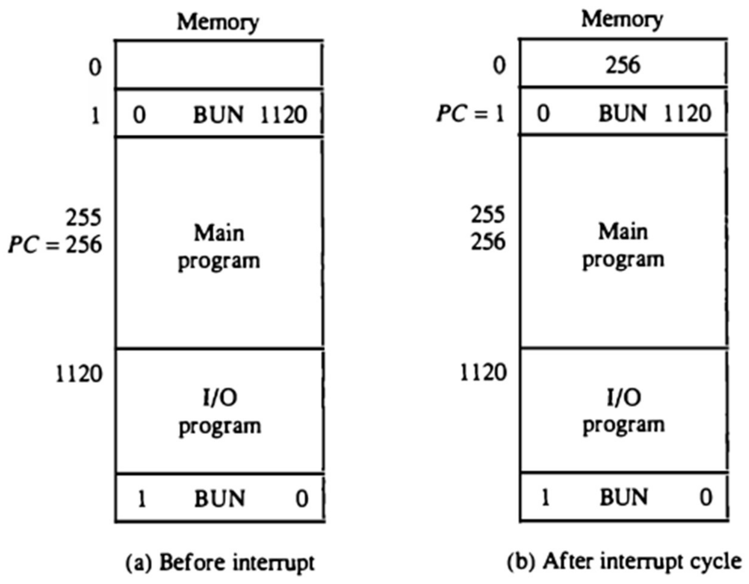


Figure 5-13 Flowchart for interrupt cycle.

This flowchart depicts the steps involved in handling a program interrupt.

1. **Instruction Cycle:** The CPU executes instructions in a program.
2. **Interrupt Enable Check (IEN):** During instruction execution, the control unit checks if IEN is enabled.
    - If IEN is 0 (disabled), interrupts are ignored, and the cycle continues.
    - If IEN is 1 (enabled), the control unit proceeds to check the flags.
3. **Flag Check:** The control unit checks both input and output flags.
    - If both flags are 0 (no data ready), the cycle continues with the next instruction.
    - If either flag is 1 (data ready), the Interrupt Flip-Flop (R) is set to 1.
4. **Interrupt Cycle Trigger:** At the end of the instruction execution phase, the control unit checks R.
    - If R is 0, the normal instruction cycle continues.
    - If R is 1 (interrupt occurred), the interrupt cycle starts.

Figure 5-14 Demonstration of the interrupt cycle.



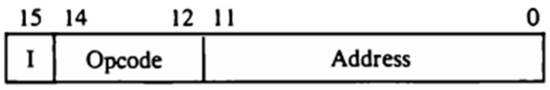
The above example depicts a situation where an interrupt occurs while the CPU is executing an instruction at address 255. The return address (256) is stored in memory location 0. The interrupt service routine (ISR) is located at address 1120.

- Interrupt occurs, setting the Interrupt Flip-Flop (R) to 1.
- The CPU stores the return address (256) in memory location 0.
- The PC is set to 1, and R is cleared to 0.
- The CPU fetches and executes the instruction at address 1, which branches to the ISR at address 1120.
- The ISR processes the interrupt (e.g., handles I/O data transfer).
- The ISR executes an instruction to enable interrupts (ION).
- The ISR executes an instruction to return from the interrupt, which involves fetching the return address (256) from memory location 0 and loading it back into the PC.
- The CPU resumes execution of the program at address 256 (the interrupted instruction).

Hence the interrupt cycle ensures efficient communication between the CPU and external devices. It allows the CPU to focus on its primary tasks while responding promptly to external requests when needed.

30) Demonstrate the instruction format.

The instruction format, also known as machine code format, refers to the way instructions are encoded and represented in a computer's memory. It defines how the CPU interprets and executes those instructions.



(a) Instruction format

Opcode (Operation Code): This field specifies the operation the CPU needs to perform. It's like a verb in a sentence, telling the CPU what to do (e.g., add, subtract, load data from memory). The opcode typically takes up a smaller portion of the instruction (e.g., 4 or 8 bits) compared to the operand fields.

Operands (Optional): These fields specify the values or memory locations that the operation acts on. They can be numbers, registers, or memory addresses. The number of operands and their size (in bits) depend on the specific instruction. An instruction might have zero operands (for operations that don't require any data), one operand (for operations on a single value), two operands (for operations like add or subtract), or even more in some cases.

Instruction Length:
The length of an instruction can vary depending on the computer architecture. Some architectures use a fixed-length instruction format (e.g., 32 bits), while others use variable-length instructions (e.g., 16 bits, 32 bits, or even longer). A fixed-length instruction format simplifies decoding for the CPU but might waste some space if some instructions don't need all the bits. Variable-length instructions can be more efficient in terms of space usage but might require additional processing for the CPU to determine the instruction length.

Addressing Modes:

The instruction format often includes information about how to access the operands. This is called the addressing mode.

1. **Direct Addressing:**

   In direct addressing, the operand's memory address is explicitly stored within the instruction itself. This simplifies the instruction execution process for the CPU as it directly knows where to find the data.

   Advantages:

   - **Fast:** The CPU can access the operand quickly because the address is readily available within the instruction.
   - **Simple:** The instruction format is straightforward, making it easier for the CPU to decode.

   Disadvantages:

   - **Limited Scope:** The instruction size is limited because it needs to accommodate the memory address. This can be inefficient for large memory spaces.
   - **Less Flexible:** Instructions become less flexible as they are tied to specific memory locations.

   Example:

   Consider an instruction to add two numbers stored in memory locations 100 and 101. The instruction format might be:

   Opcode | Direct Address 1 (100) | Direct Address 2 (101)

2. **Indirect Addressing:**

   In indirect addressing, the instruction doesn't contain the actual memory address of the operand. Instead, it specifies a memory location that holds the operand's address. This introduces an extra step for the CPU:

   1. Fetch the instruction.
   2. Extract the memory address stored in the instruction (indirect address).
   3. Fetch the operand's actual address from the memory location pointed to by the indirect address.
   4. Access the operand using the retrieved address.

   Advantages:

- **Flexibility:** Instructions become more flexible as they don't need to specify the exact memory location of the operand. This allows for dynamic memory allocation and data structures like linked lists.
- **Compact Instructions:** The instruction size can be smaller as it only needs to store the indirect address, not the full operand address.

Disadvantages:

- **Slower:** Accessing the operand takes more time due to the extra step of fetching the actual address.
- **More Complex:** The CPU needs to perform additional processing to locate the operand.

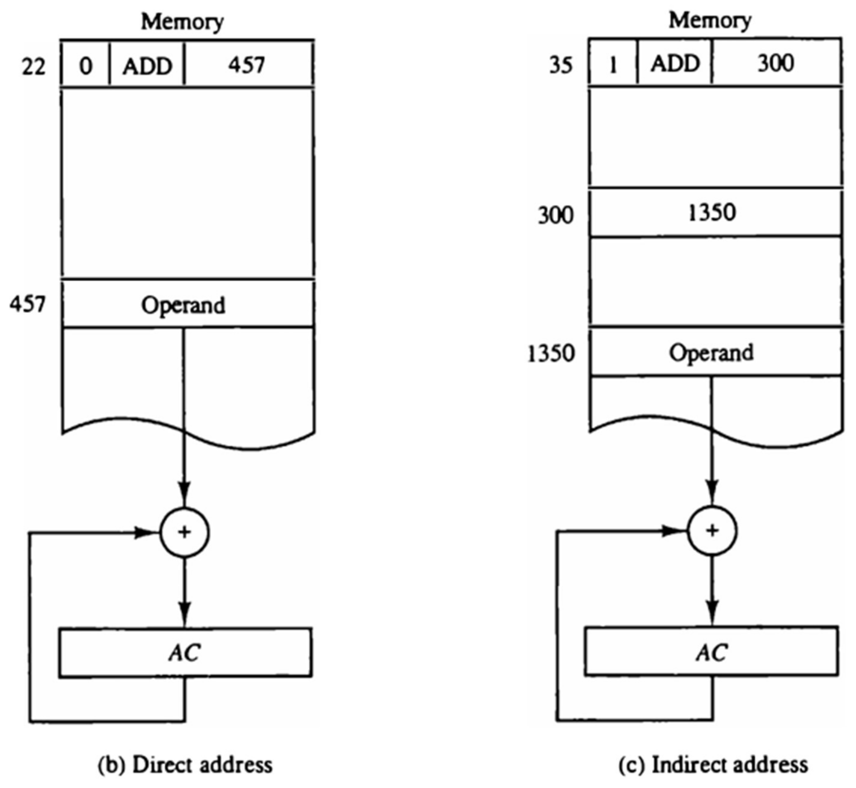


Figure 5-2 Demonstration of direct and indirect address.