# Basic Computer Organization

# In this lecture, we will study

i. Computer Instructions, Instruction Codes, and Instruction Cycles

ii. Timing and Control

iii. Memory-Reference Instructions

iv. Input/Output and Interrupts

v. Complete Computer Description and Design of a Basic Computer

**Chapter Exercises**

# Computer Instructions and Instruction Codes

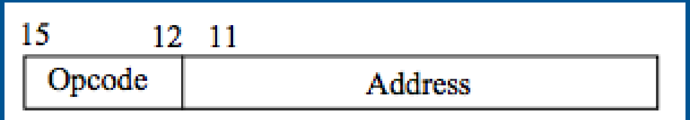
# Computer Instructions

A **computer instruction** is a binary code that specifies a *sequence of microoperations* for the computer.

# Instruction Codes

An **instruction code** is group of bits that instruct the computer to perform a *specific operation*.

An instruction code is usually divided into different parts, and each part has its own particular interpretation.




Instruction format

# Instruction Codes

The most basic part of an instruction code is its OPERATION CODE (or OPCODE); it is a group of bits that define operations such as add, subtract, multiply, shift, and complement.

For instance, the ADD operation can be assigned the opcode 110010. When the computer sees this opcode, it automatically knows to add the operands.

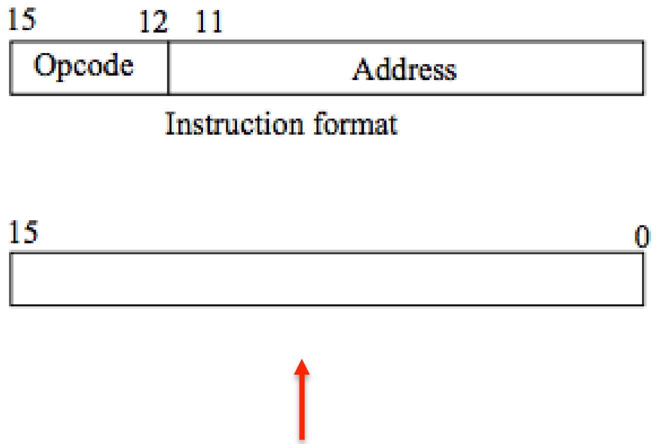
# Stored Program Organization

The simplest way to organize a computer is to have one **processsor register** & an **instruction code format with two parts.**

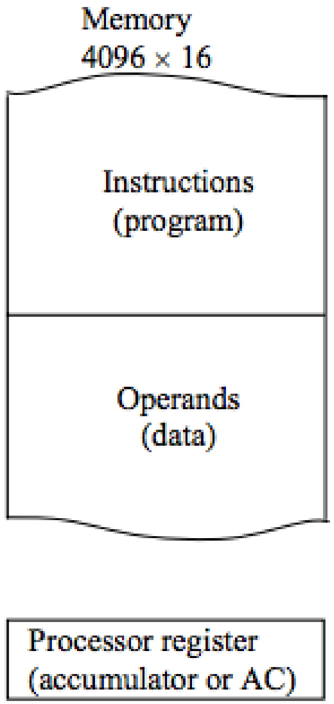
The first part specifies the operation to be performed and the second part specifies the address of the required operand.

The operand is read from the memory and is then operated on together with the data stored in the processor register.

# Stored Program Organization




For operations such as ‘clear AC’ or ‘complement AC’ (that do not require an operand from the memory, the instruction only has one part.




For a memory unit with 4096 words, we need 12 bits to specify an address (since $2^{12} = 4096$).

If we store each instruction in a 16-bit memory word, then we have 4 bits available for the operation code (opcode).With a 4-bit opcode, we will be able to specify a total of $2^4 = 16$ distinct operations.

# Addressing Modes

The first part of an instruction i.e., the opcode, specifies a particular operation to be performed. This operation has to be performed on some operand i.e., data, stored in a computer register or in the main memory.

The way any operand is selected during program execution is dependent on the **addressing mode** being used in the instruction. The three addressing modes we will discuss at this point are:

> Immediate

> Direct

> Indirect

# Addressing Modes

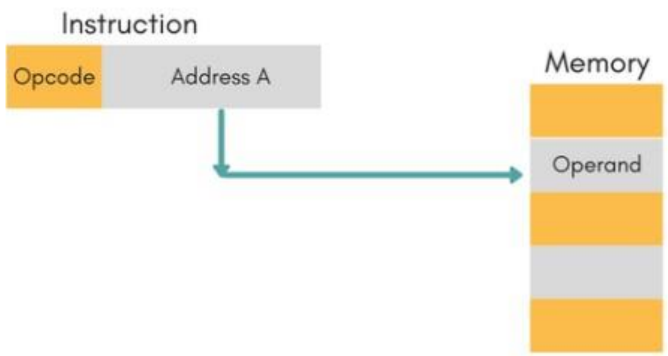
**Immediate Addressing:** When the second part of the instruction code specifies the required operand, the instruction is referred to as an “Immediate Instruction” (because it has an *immediate operand*).

**Direct Addresssing:** When the second part of the instruction code specifies the address of the required operand, the instruction is said to have a “Direct Address” (because the *address of the operand is directly present in the instruction itself*).

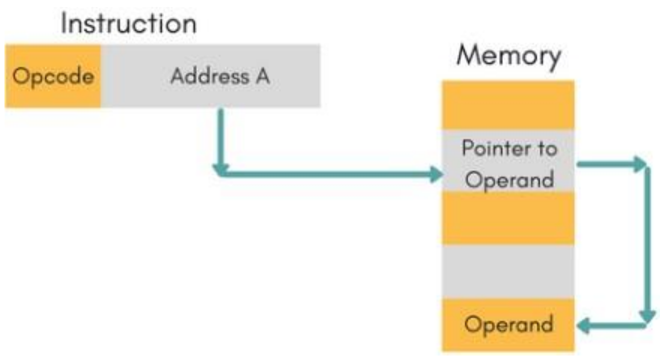
**Indirect Addressing:** When the second part of the instruction code contains not the address of the operand but rather the address of the memory word in which the address of the operand is present, the instruction is said to have an “Indirect Address”.

Immediate Addressing


Direct Addressing



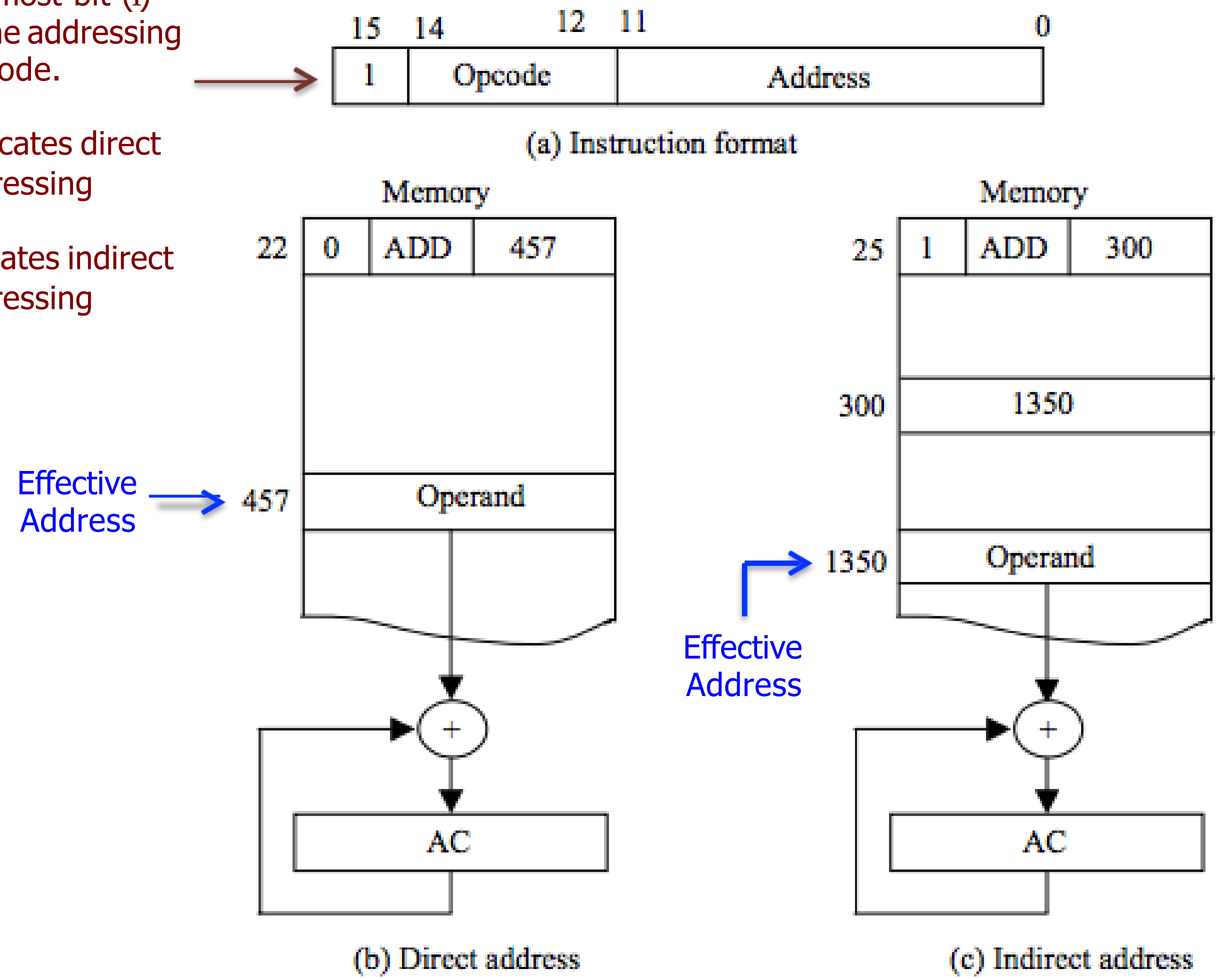

Indirect Addressing



The leftmost bit (I) signifies the addressing mode.

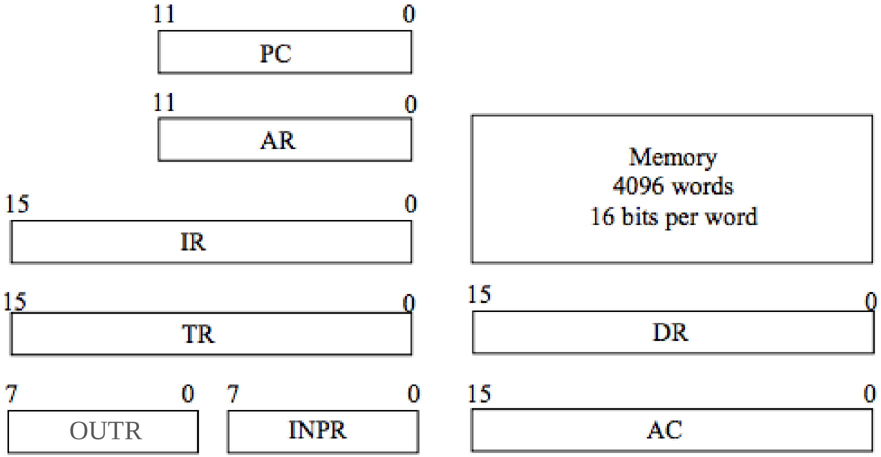
I = 0 indicates direct addressing

I = 1 indicates indirect addressing



# Computer Registers

# Computer Registers




| 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 |

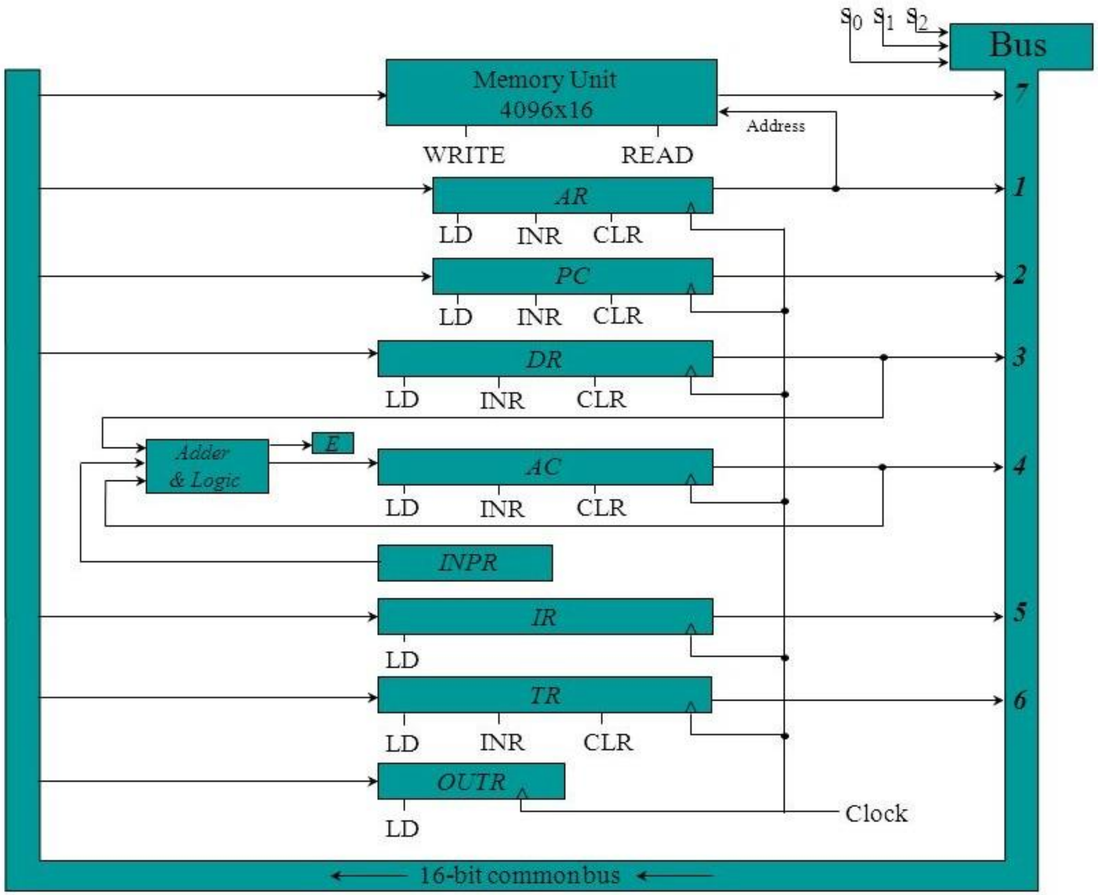
# Common Bus System

A basic computer has eight registers, a memory unit, and a control unit; paths must be provided to transfer information from one unit to another.

**If separate lines of communication are used between each unit, the number of wires will be excessive.**

**It is more efficient to use a COMMON BUS SYSTEM.**

# Common Bus System

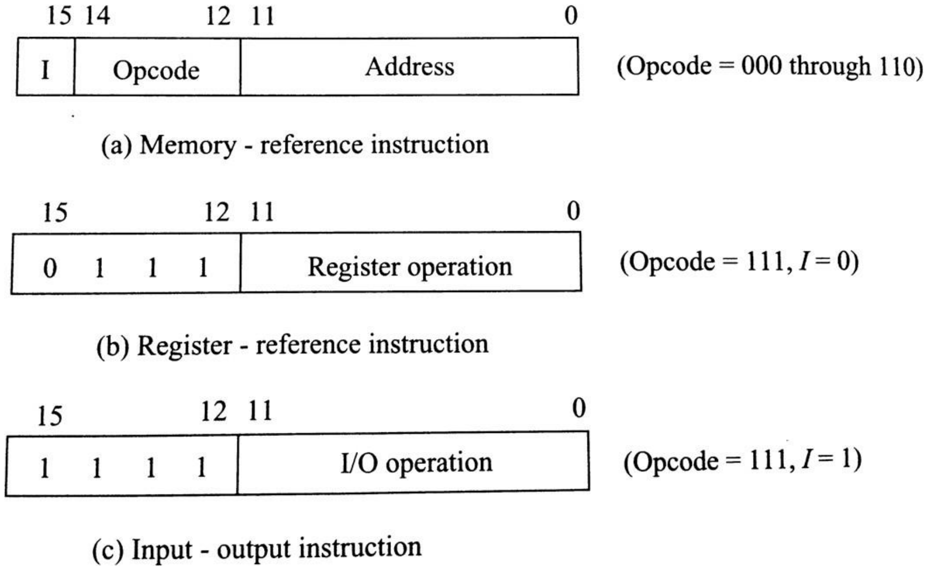


# Computer Instructions

# Computer Instructions

A basic computer has three instruction code formats:

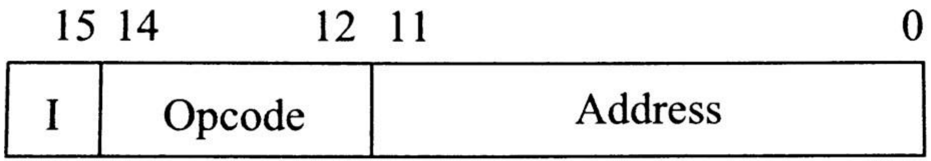
> Memory-Reference Instruction Format
> Register-Reference Instruction Format
> Input-Output Instruction Format



# Memory-Reference Instructions

A memory-reference instruction refers to a memory address as operands.

The other operand is always the accumulator.



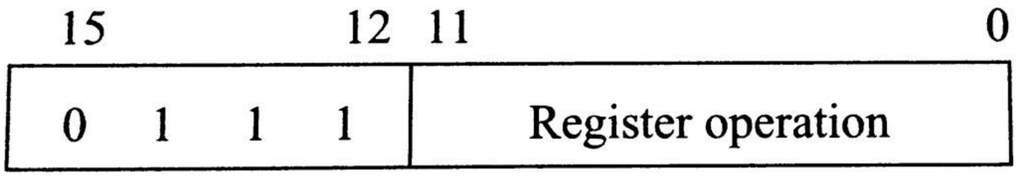

(Opcode = 000 through 110)

A memory-reference instruction uses 12 bits to specify the address of the operand and 1 bit to specify the addressing mode I; I=0 indicates direct addressing mode while I = 1 indicates indirect addressing mode.

The opcode in a memory-reference instruction can take any value from 000 to 110.

# Register-Reference Instructions

Register-reference instructions perform operations on registers instead of memory.



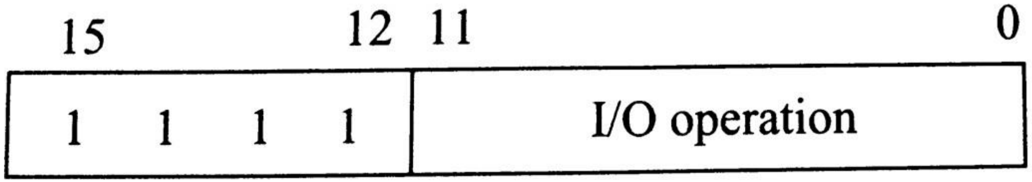

(Opcode = 111, $I = 0$)

A register-reference instruction is identified by the opcode 111, with a 0 in the leftmost bit (i.e., bit 15).

A register-reference instruction specifies an operation on or a test of the accumulator (AC). An operand from memory is not needed, and so the 12 bits (that are generally reserved for the address) are used to specify the register operation.

# Input-Output Instruction

These instruction enable communication between the computer and the outside environment.




(Opcode = 111, $I$ = 1)

An input-output instruction is identified by the opcode 111, with a 1 in the leftmost bit (i.e., bit 15).

A input-output instruction does not need an operand from the memory, and so the 12 bits (that are generally used to specify the address) are used to specify the input-output operation instead.

# Basic Computer Instructions

| Symbol | Hexadecimal code I = 0 | Hexadecimal code I = 1 | Description |
|---|---|---|---|
| AND | 0xxx | 8xxx | AND memory word to *AC* |
| ADD | 1xxx | 9xxx | Add memory word to *AC* |
| LDA | 2xxx | Axxx | Load memory word to *AC* |
| STA | 3xxx | Bxxx | Store content of *AC* in memory |
| BUN | 4xxx | Cxxx | Branch unconditionally |
| BSA | 5xxx | Dxxx | Branch and save return address |
| ISZ | 6xxx | Exxx | Increment and skip if zero |
| INP | F800 | | Input character to *AC* |
| OUT | F400 | | Output character from *AC* |
| SKI | F200 | | Skip on input flag |
| SKO | F100 | | Skip on output flag |
| ION | F080 | | Interrupt on |
| IOF | F040 | | Interrupt off |

| Symbol | Hexadecimal code | Description |
|---|---|---|
| CLA | 7800 | Clear *AC* |
| CLE | 7400 | Clear *E* |
| CMA | 7200 | Complement *AC* |
| CME | 7100 | Complement *E* |
| CIR | 7080 | Circulate right *AC* and *E* |
| CIL | 7040 | Circulate left *AC* and *E* |
| INC | 7020 | Increment *AC* |
| SPA | 7010 | Skip next instruction if *AC* positive |
| SNA | 7008 | Skip next instruction if *AC* negative |
| SZA | 7004 | Skip next instruction if *AC* zero |
| SZE | 7002 | Skip next instruction if *E* is 0 |
| HLT | 7001 | Halt computer |

# Instruction Set Completeness

**A set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:**

1. *Arithematic, logical, and shift instructions*

2. *Instructions for moving information to/from memory and processor registers*

3. *Program control and status checking instructions*

4. *Input and output instructions*

# Timing and Control

# Timing and Control

The timing for all registers in a basic computer is controlled by a **Master Clock Generator.**

The clock pulses are applied to all **flip-flops and registers in the system,** as well as, the **flip-flops and registers in the control unit.**

The clock pulses DO NOT change the state of a register **until the register is enabled by a control signal** generated in the control unit.

# Two Major Types of Control Organizations

## Hardwired Control

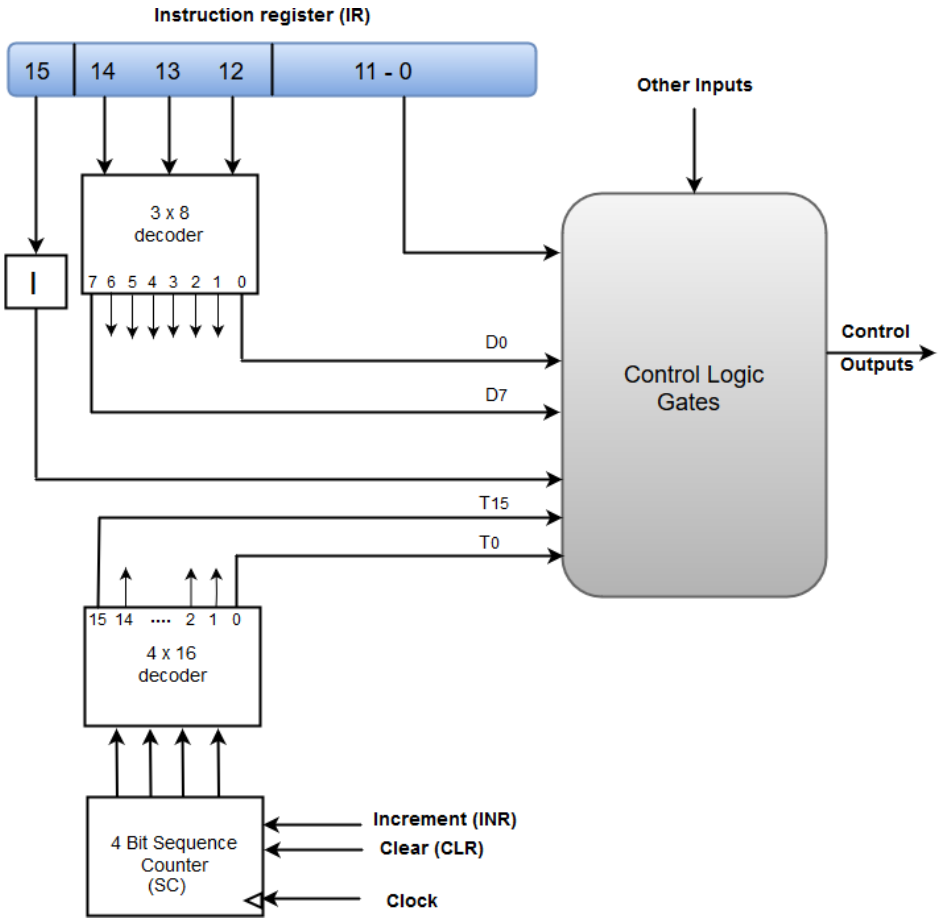
In this type of control organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits.

It can be optimized to produce a fast mode of operation.

## Microprogrammed Control

In this type of control organization, the control information is stored in a control memory. This memory is programmed to initiate the required sequence of microoperations.

# Block Diagram of a Typical Control Unit



# Block Diagram of a Typical Control Unit

An instruction read from the memory is placed in the IR

I signifies the addressing mode

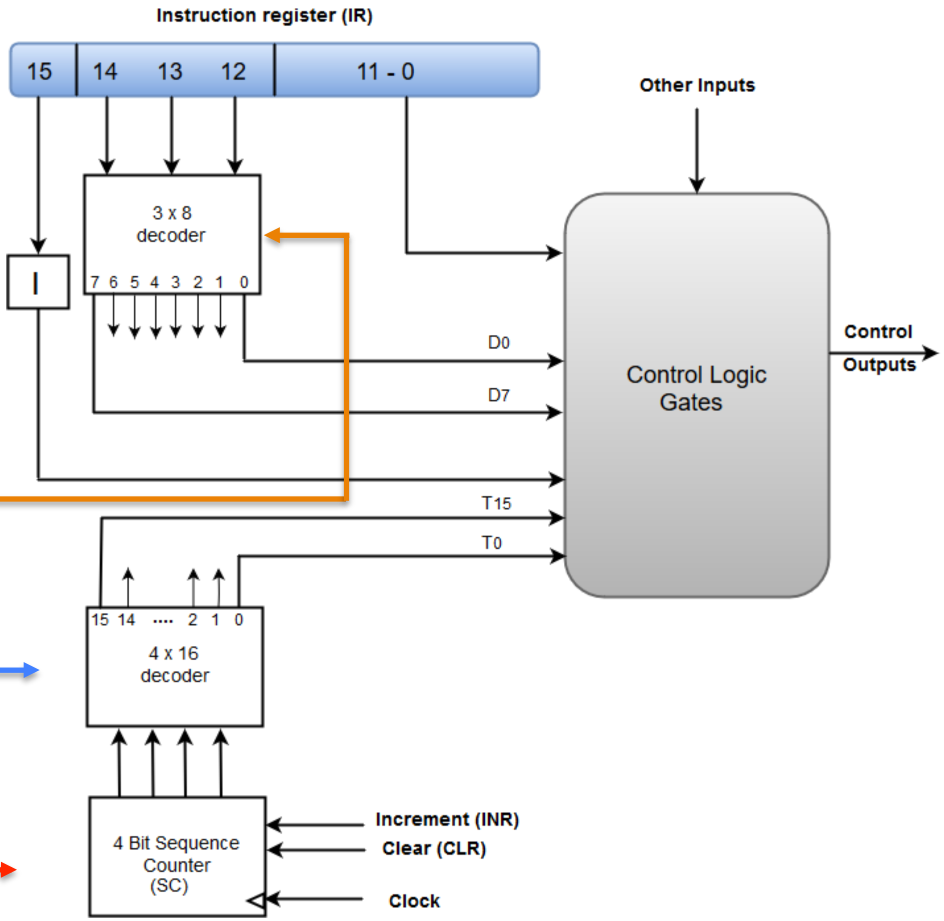
I = 0 :- direct addressing

I = 1 :- indirect addressing

This decoder decodes the opcode

This decoder decodes outputs of the SC into timing signals

The 4-bit SC can count from 0 to 15



# Instruction Cycle

# Instruction Cycle

A program residing in the memory is basically a sequence of instructions, and a specific cycle has to be followed for the execution of each instruction, as illustrated below:

A typical instruction cycle

- **Fetch** the instruction from memory
- **Decode** the instruction
- **Read the effective address** from memory (in case of indirect addressing)
- **Execute** the instruction

# Instruction Cycle: Fetch and Decode

Initially, the PC is loaded with the address of the first instruction in the program. The sequence counter (SC) is cleared to 0, providing a decoded timing signal $T_0$.
After each clock pulse, the SC is incremented by 1, so that the next timing signals $T_1$ , $T_2$, $T_3$ and so on can be generated.

The microoperations for the fetch and decode phases can be specified by the following register transfer statements:

**$T_0$ : AR ← PC**

**$T_1$ : IR ← M[AR], PC ← PC+1**

**$T_2$ : $D_0$, $D_1$,….., $D_7$ ← Decode IR (12-14), AR ← IR (0-11), I ← IR(15)**

# Fetch and Decode: Microoperation 1

$$T_0 : AR \leftarrow PC$$

The address of the instruction is transferred from PC to AR.

# Fetch and Decode: Microoperation 2

**$T_1$ : IR ← M[AR], PC ← PC+1**

The contents of the memory word M, whose address is present in AR, is moved to IR, and the PC is incremented by one (so that it now points to the next instruction to be fetched). This happens with the clock transition associated with the next timing signal, i.e., $T_1$.

# Fetch and Decode: Microoperation 3

**$T_2$ : $D_0$, $D_1$,….., $D_7$ ← Decode IR (12-14), AR ← IR (0-11), I ← IR(15)**

At time $T_2$ :

i) the opcode (which is present in bits 12, 13, and 14 of the IR) is decoded.

ii) the address of the operand (which is present in bits 0 – 11 of the IR) is transferred to AR.

iii) the single bit (bit 15 of the IR) which indicates the indirect addressing mode is transferred to flip-flop I.

# Memory-Reference Instructions

# Memory-Reference Instructions

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

**Execution of a memory-reference instruction starts with the timing signal $T_4$.**

# Memory-Reference Instructions

**1. AND to AC**

$$\mathbf{AC \leftarrow AC \wedge M[AR]}$$

This instruction performs the AND logic operation on pairs of bits in the AC (accumulator) and the memory word specified by the effective address present in AR. The result is then transferred to AC.

The microoperations that execute this instruction are:

$$\mathbf{D_0T_4: DR \leftarrow M[AR]}$$

$$\mathbf{D_0T_5: AC \leftarrow AC \wedge DR, SC \leftarrow 0}$$

# Memory-Reference Instructions

## 2. ADD to AC

$$AC \leftarrow AC + M[AR],\ E \leftarrow C_{out}$$

This instruction adds the contents of the memory word specified by the effective address to the value of AC. The sum is transferred to AC and the output carry is transferred to E (extended accumulator) flip-flop.

The microoperations needed to execute this instruction are:

$$D_1T_4\text{: } DR \leftarrow M[AR]$$

$$D_1T_5\text{: } AC \leftarrow AC + DR,\ E \leftarrow C_{out},\ SC \leftarrow 0$$

# Memory-Reference Instructions

**3. LDA: Load to AC**

$$AC \leftarrow M[AR]$$

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

$$D_2T_4\text{: } DR \leftarrow M[AR]$$

$$D_2T_5\text{: } AC \leftarrow DR,\ SC \leftarrow 0$$

# Memory-Reference Instructions

**4. STA: Store AC**

$$M[AR] \leftarrow AC$$

This instruction stores the contents of AC into the memory word specified by the effective address.

The microoperation needed to execute this instruction is:

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

# Memory-Reference Instructions

**5. BUN: Branch Unconditionally**

$$PC \leftarrow AR$$

This instruction transfers the control of the program to the instruction specified the effective address.

The microoperation needed to execute this instruction is:

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

# Memory-Reference Instructions

**6. BSA: Branch and Save Return Address**

$$M[AR] \leftarrow PC, \; PC \leftarrow AR+1$$

This instruction is useful for branching to a subroutine or procedure.

When executed, this intruction stores the address of the next instruction into a memory location specified by the effective address. The effective address plus one is then added to PC (which will serve as the first address in the subroutine).

# Memory-Reference Instructions

**6. BSA: Branch and Save Return Address**

$$M[AR] \leftarrow PC,\ PC \leftarrow AR+1$$

The microoperations needed to execute the BSA instruction are:

$$D_5T_4:\ M[AR] \leftarrow PC,\ AR \leftarrow AR + 1$$
$$D_5T_5:\ PC \leftarrow AR,\ SC \leftarrow 0$$

# Memory-Reference Instructions

**7. ISZ: Increment and Skip if Zero** (this is the longest instruction)

$$M[AR] \leftarrow M[AR] + 1$$

$$\text{If } M[AR] + 1 = 0 \text{ then } PC \leftarrow PC + 1$$

This instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1.

The microoperations needed to execute this instruction are:

$$D_6T_4: DR \leftarrow M[AR]$$

$$D_6T_5: DR \leftarrow DR + 1$$

$$D_5T_6: M[AR] \leftarrow DR, \text{ if } (DR = 0) \text{ then } (PC \leftarrow PC+1), SC \leftarrow 0$$

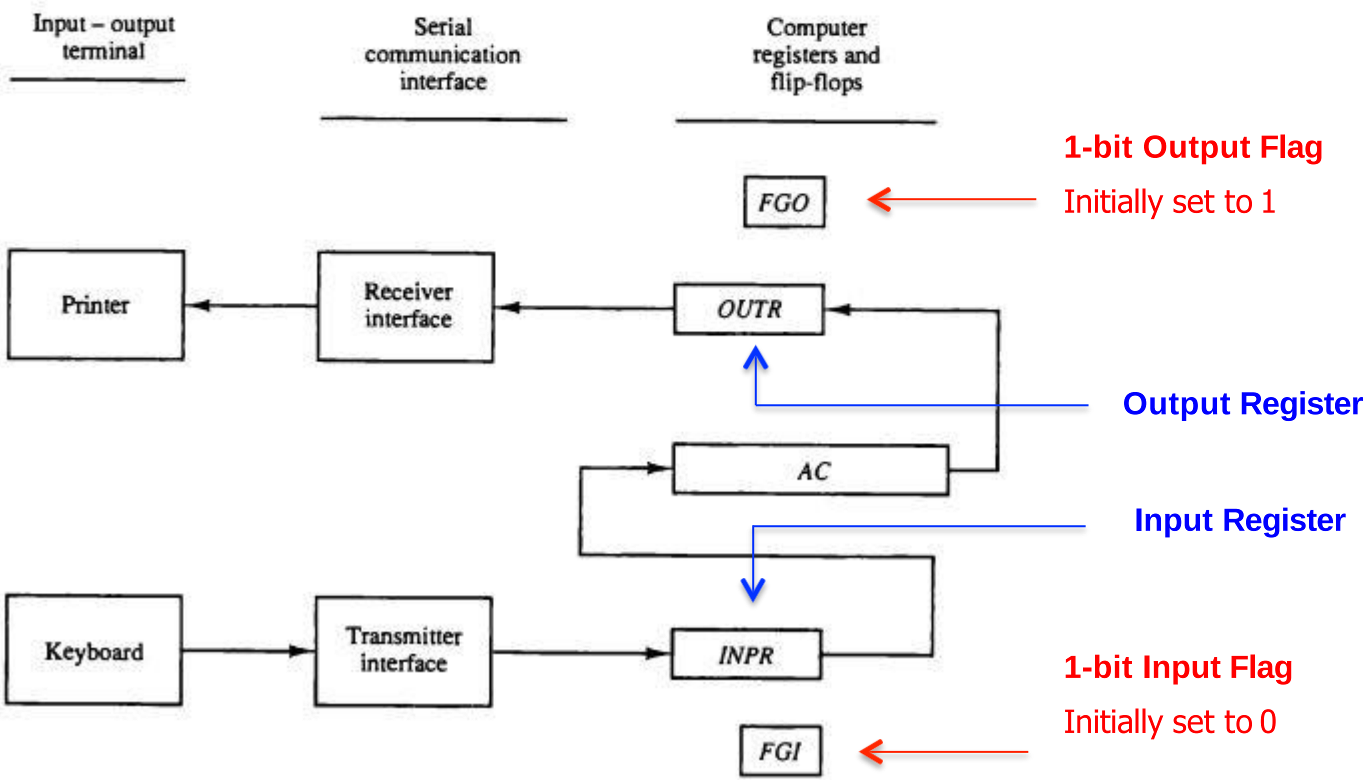
# Input and Output

# Input and Output

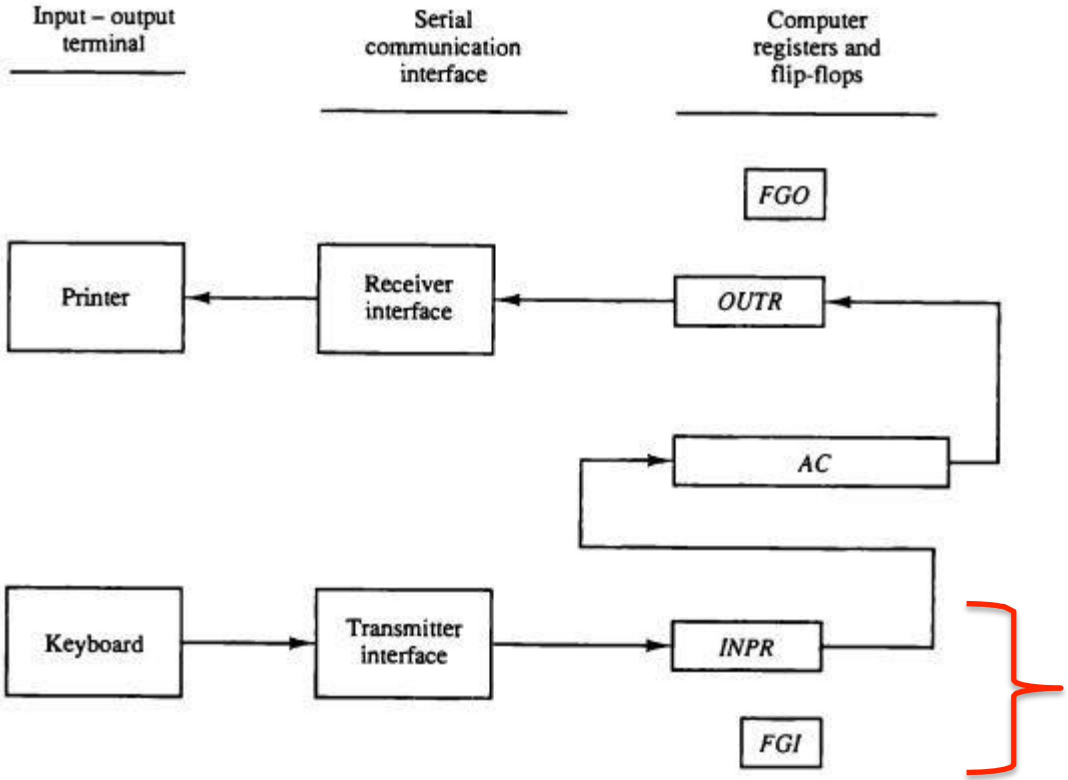
A computer can serve no useful purpose
unless it communicates with the external environment.

Instructions and data stored in the memory
must come from some input device,
and the results of the computations
must be transmitted to the user through some output device.

# Input-Output Configuration

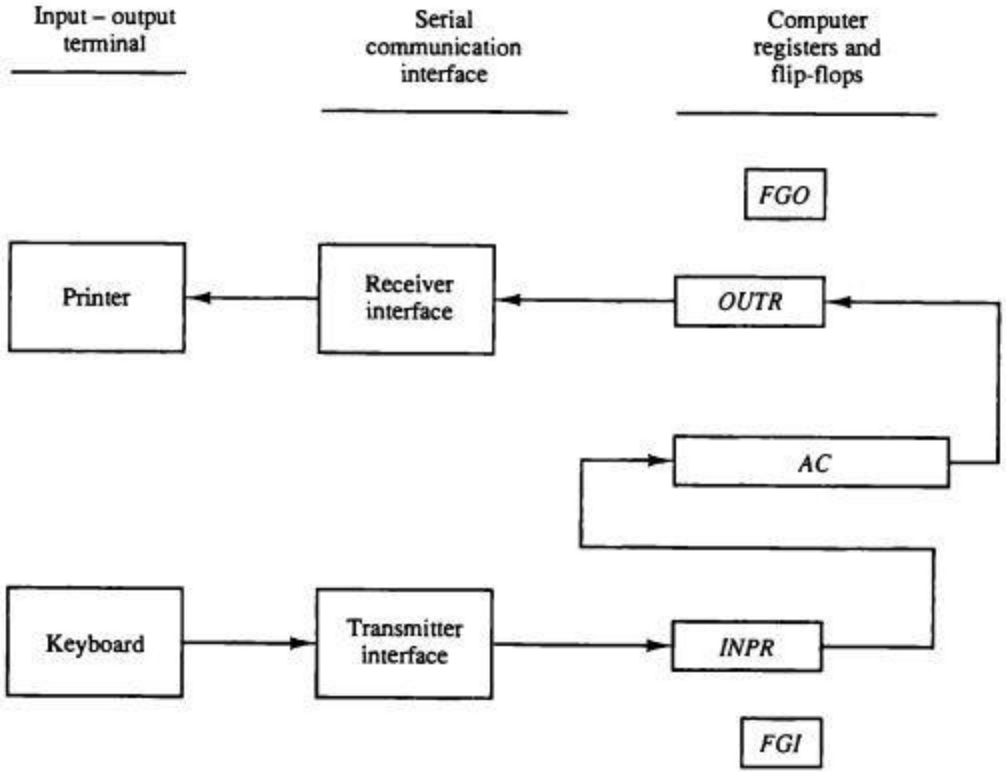


# Input-Output Configuration: Flow of Information Transfer



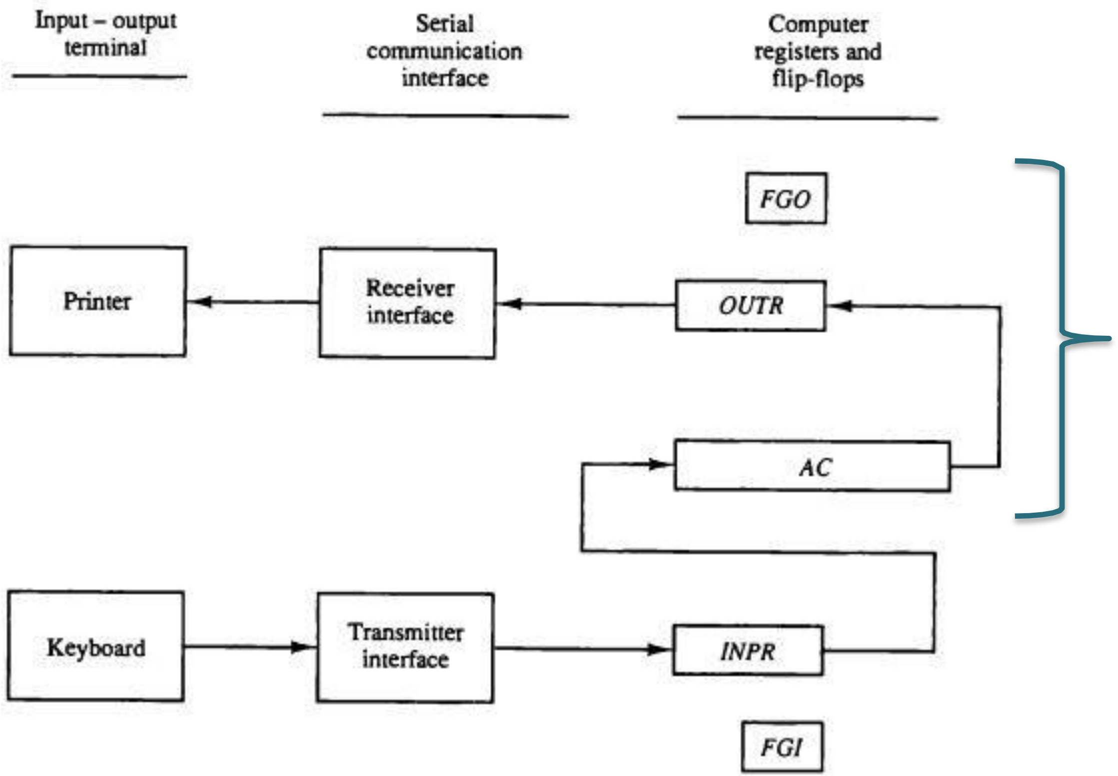

**Initially, FGI is set to 0. When you press a key on the keyboard, an 8-bit alphanumeric code is shifted into INPR and FGI is set to 1. (As long as the flag is set, pressing another key does not change the information in INPR).**

# Input-Output Configuration: Flow of Information Transfer




**The computer checks the flag; if FGI = 1, information from INPR is transferred in parallel to AC and FGI is cleared to 0.**

# Input-Output Configuration: Flow of Information Transfer




Initially, FGO is set to 1. The computer checks the flag; if it is 1, the information from AC is transferred in parallel to OUTR and FGO is cleared to 0. The printer accepts the coded info, prints the corresponding character, and when this is done, FGO is set to 1.

(As long as FGO = 0, no new info goes into OUTR)

# Input/Output Instructions

$D_7 I T_3 = p$ (common to all input-output instructions)
$IR(i) = B_i$ [bit in $IR(6-11)$ that specifies the instruction]

| | | | |
|---|---|---|---|
| INP | $p$: | $SC \leftarrow 0$ | Clear $SC$ |
| | $pB_{11}$: | $AC(0-7) \leftarrow INPR, \quad FGI \leftarrow 0$ | Input character |
| OUT | $pB_{10}$: | $OUTR \leftarrow AC(0-7), \quad FGO \leftarrow 0$ | Output character |
| SKI | $pB_9$: | If $(FGI = 1)$ then $(\mathrm{PC} \leftarrow PC + 1)$ | Skip on input flag |
| SKO | $pB_8$: | If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$ | Skip on output flag |
| ION | $pB_7$: | $IEN \leftarrow 1$ | Interrupt enable on |
| IOF | $pB_6$: | $IEN \leftarrow 0$ | Interrupt enable off |

# Program Interrupt

# Programmed Control Transfer

The process of communication just described is referred to as

PROGRAMMED CONTROL TRANSFER.

In this case, the computer keeps checking the flag bit,

and when it finds that the bit is set,

it initiates an information transfer.

# Programmed Control Transfer

Programmed control transfer is **INEFFICIENT**,
because of the difference in the information flow rates
between the processor and the input/output device.

# Program Interrupt: An Alternative to Programmed Control Transfer

An alternative to programmed control transfer is to **let the input/output device inform the processor when it is ready for data transfer,** and in the meantime, the processor is free to perform other useful tasks.

THIS IS KNOWN AS "PROGRAM INTERRUPT".

# Program Interrupt: How It Works

**Step 1**

When the computer is running a program, it does not check the flags.

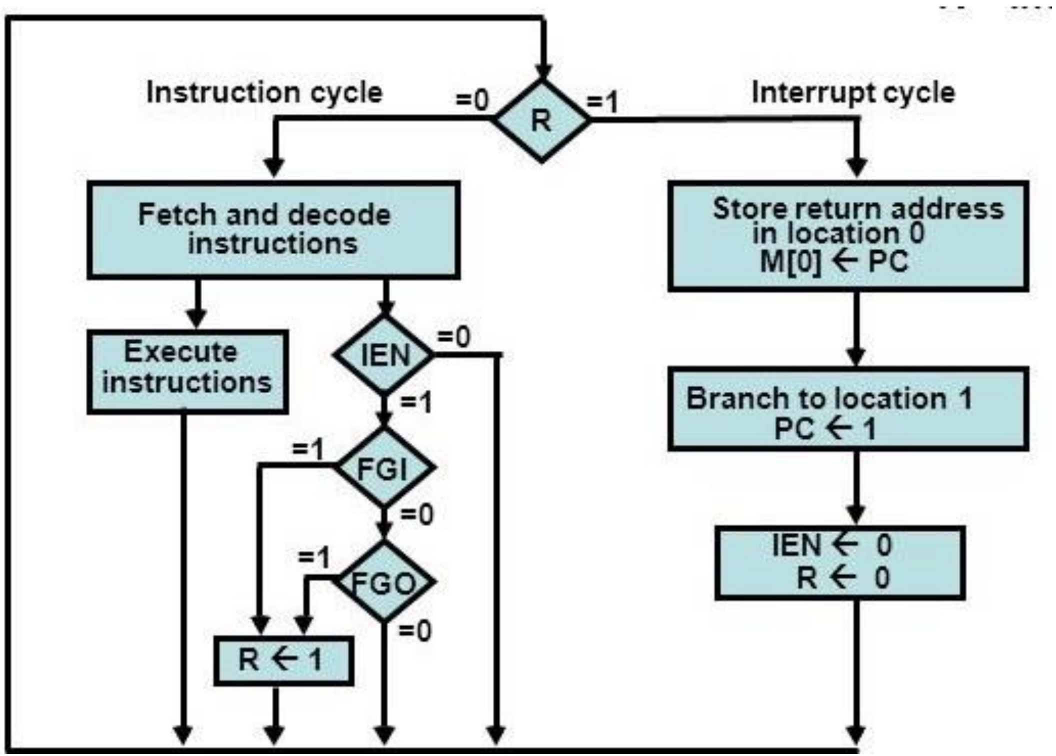
**Step 2**

When a flag is set, the computer is interrupted from its current program, and is informed that a flag has been set.

**Step 3**

The computer momentarily deviates from what it is doing to take care of the input/output operation, before returning to its original task.

# Flowchart of Interrupt Cycle



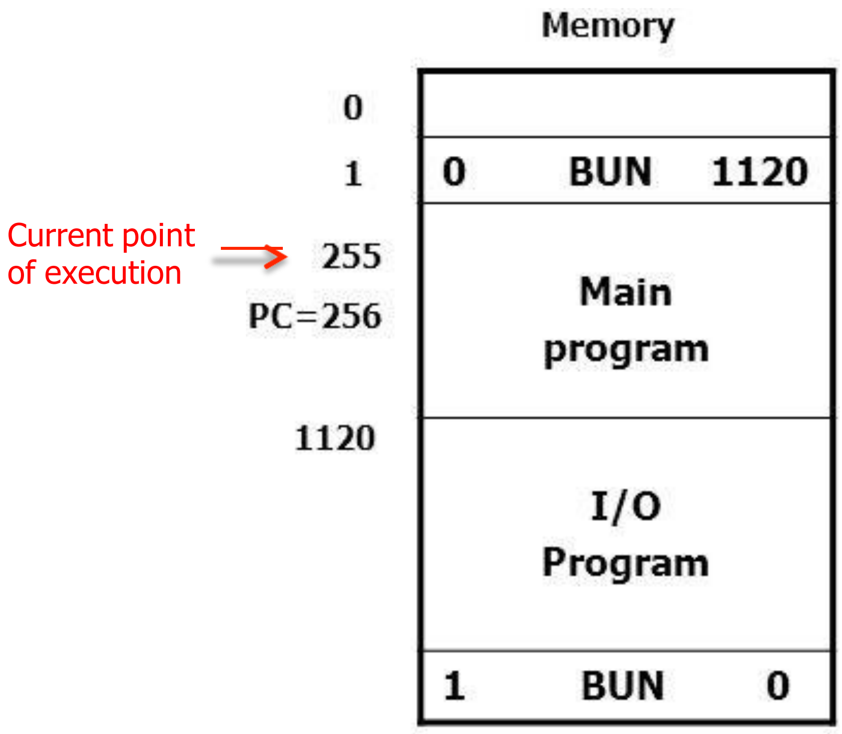

R= Interrupt Flip-Flop

IEN = Interrupt Enable Flip-Flop

When IEN = 0, the flags cannot interrupt the computer.

When IEN = 1, the computer can be interrupted.

# Demonstration of Interrupt Cycle

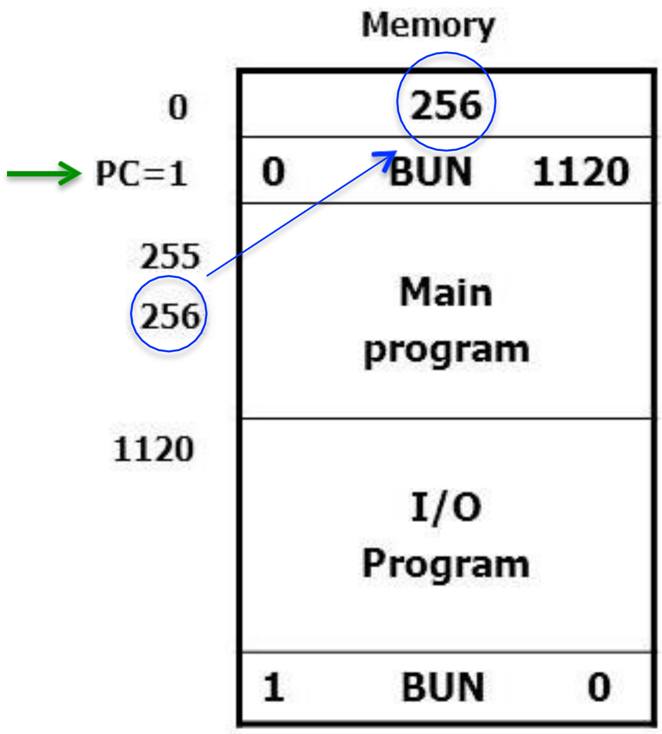



Before Interrupt

Before the interrupt, the main program is being executed (specifically, the instruction at location 255 is being executed).

When R = 1, the interrupt cycle begins…

# Demonstration of Interrupt Cycle




After Interrupt

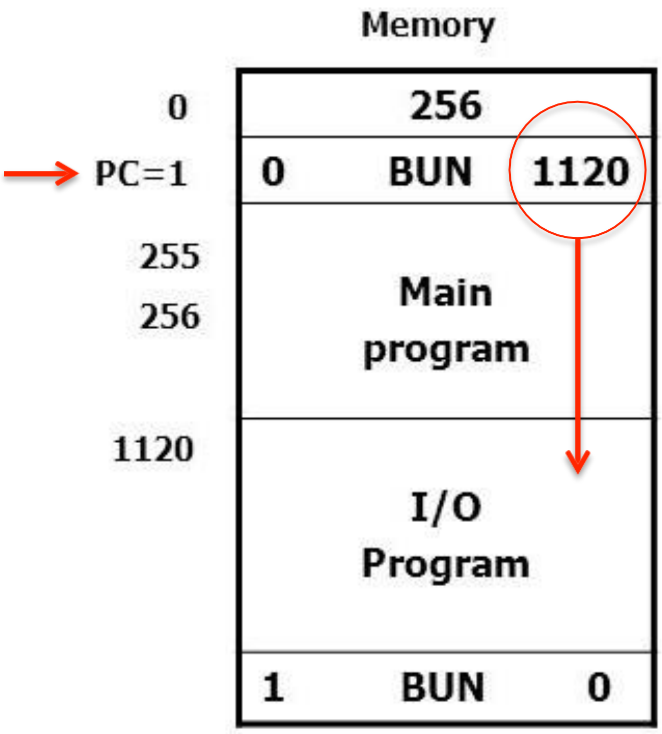
When the interrupt cycle begins…

the content of PC (i.e., 256) is placed in memory location 0

PC is set to 1

and R is cleared to 0

# Demonstration of Interrupt Cycle

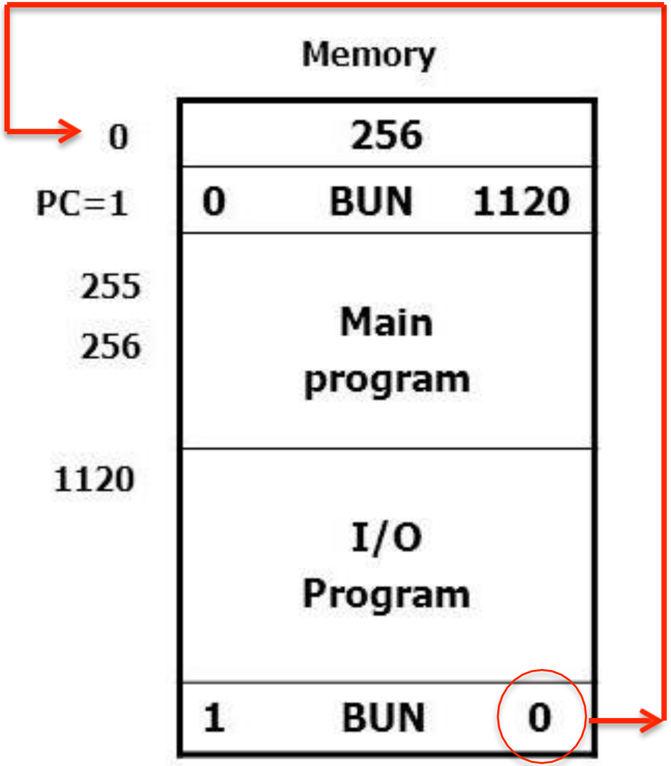



After Interrupt

At the beginning of the next intruction cycle, the instruction whose address is stored in PC is read.

This instruction takes the control to the I/O Program.

# Demonstration of Interrupt Cycle




After Interrupt

When the I/O program is finished, the control goes back to memory location 0, which further takes the control to location 256.

# Microoperations for Interrupt Cycle

1. $RT_0$: $AR \leftarrow 0$, $TR \leftarrow PC$

(During the first timing signal, AR is cleared to 0, and the content of PC is transferred to a temporary register TR.)

2. $RT_1$: $M[AR] \leftarrow TR$, $PC \leftarrow 0$

(During the second timing signal, the return address is stored in the memory location 0, and PC is cleared to 0.)

3. $RT_2$: $PC \leftarrow PC + 1$, $IEN \leftarrow 0$, $R \leftarrow 0$, $SC \leftarrow 0$

(During the third timing signal, PC is incremented, IEN, R, and SC are cleared to 0.)

# Complete Computer Description

# Design of a Basic Computer

# Components of a Basic Computer

A basic components consists of

- A memory unit with 4096 words of 16 bit each
- Nine registers (*PC, AR, DR, AC, IR, TR, OUTR, INPR, SC*)
- Seven flip-flops (*I, S, E, R, IEN, FGI, FGO*)
- Two decoders (3x8 operation decoder, 4x16 timing decoder)
- A 16-bit common bus
- Control logic gates
- Adder and logic circuit connected to the input of AC