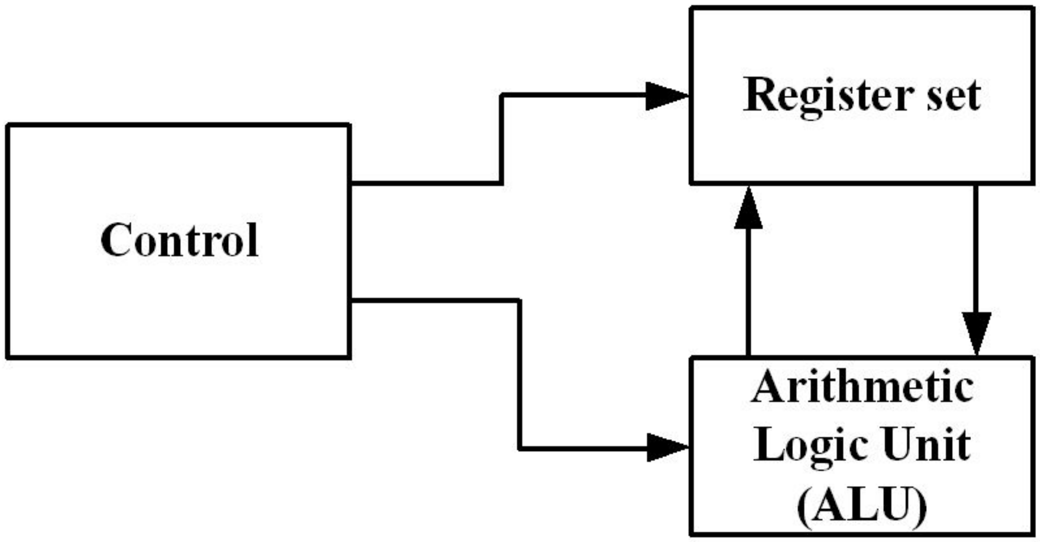
Central Processing Unit

Introduction

- The part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU.

- The CPU is made up of three major parts, as shown in diagram.



Major components of CPU

- The register set stores intermediate data used during the execution of the instructions.

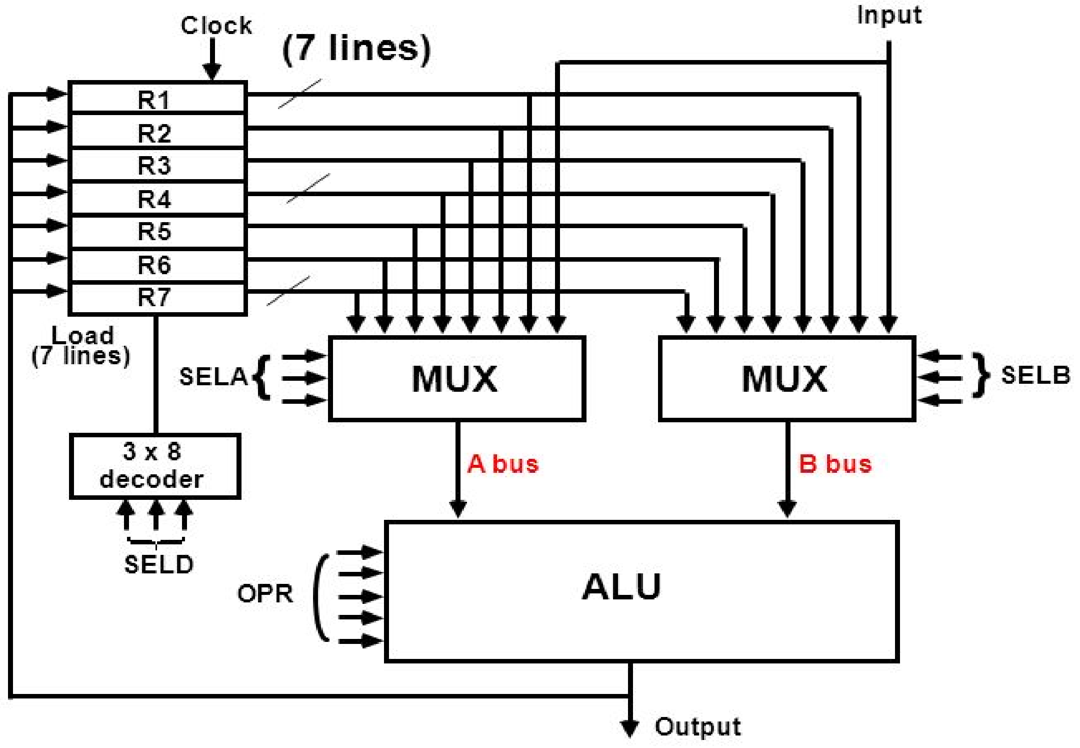
- The arithmetic logic unit (ALU) performs the required micro operations for executing the instructions.

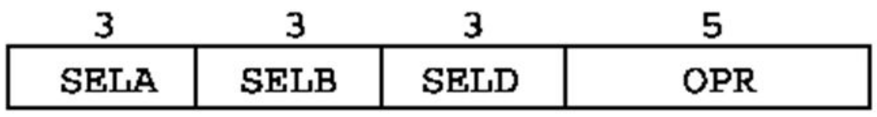
➢ The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

General Register Organization

➢ A bus organization for seven CPU registers is shown in diagram.

BUS: A, B





3	3	3	5
SELA	SELB	SELD	OPR

➢ The output of each register is connected to two multiplexers (MUX) to form the two buses A and B.

- The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses form the inputs to a common arithmetic logic unit (ALU).

- The operation selected in the ALU determines the arithmetic or logic micro operation that is to be performed.

- The result of the micro operation is available for output data and also goes into the inputs of all the registers.

- The register that receives the information from the output bus is selected by a decoder. The decoder activates one of the register load inputs, thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.

- For example, to perform the operation.

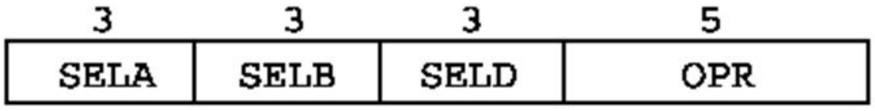
$$R1 \leftarrow R2 + R3$$

- The control must provide binary selection variables to the following selector inputs:

1. MUX A selector (SELA): to place the content of R2 into bus A.
2. MUX B selector (SELB): to place the content of R3 into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition A + B.
4. Decoder destination selector (SELD): to transfer the content of the output bus into R1.

Control word

- There are 14 binary selection inputs in the unit, and their combined value specifies a control word. The 14-bit control word is defined in following diagram.

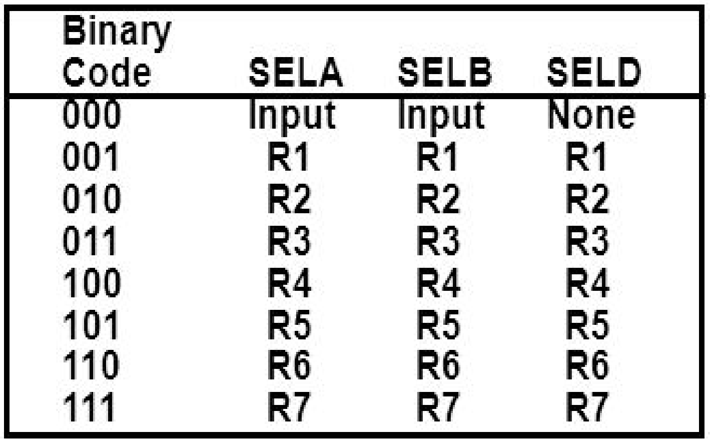


- It consists of four fields. Three fields contain three bits each, and one field has five bits.

- The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU. The three bits of SELD select a destination register using the decoder and its seven load output. The five bits of OPR select one of the operations in the ALU. The 14-bit control word when applied to the selection inputs specify a particular micro operation.

- The encoding of the register selections is specified in following table.

Encoding of register selection fields



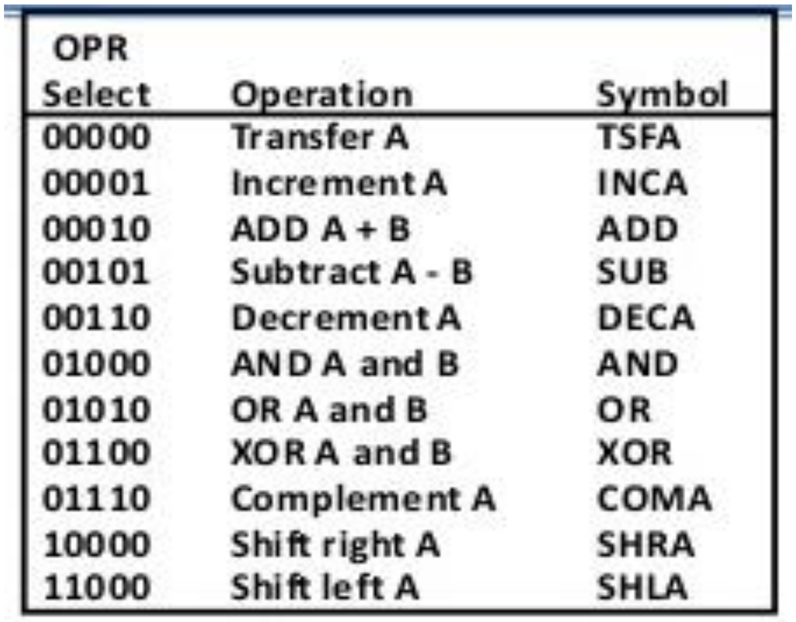
Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

- When SELA or SELB is 000, the corresponding multiplexer selects the external input data.

- When SELD = 000, no destination register is selected but the contents of the output bus are available in the external output.

- The CPU must provide shift operations. The shifter may be placed in the input of the ALU to provide a preshift capability, or at the output of the ALU to provide post shifting capability.

- The encoding of the ALU operations for the CPU is specified in following table. The OPR field has five bits and each operation is designated with a symbolic name.



OPR Select	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	ADD A + B	ADD
00101	Subtract A - B	SUB
00110	Decrement A	DECA
01000	AND A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
01110	Complement A	COMA
10000	Shift right A	SHRA
11000	Shift left A	SHLA

Example of Micro Operations

- For example, the subtract micro operation given by the statement.

  $R1 \leftarrow R2 - R3$
- Specifies R2 for the A input of the ALU, R3 for the B input of the ALU, R1 for the destination register, and an ALU operation to subtract A − B.

Field :	SELA	SELB	SELD	OPR
Symbol :	R2	R3	R1	SUB
Control word:	010	011	001	00101

Stack Organization

- A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved.

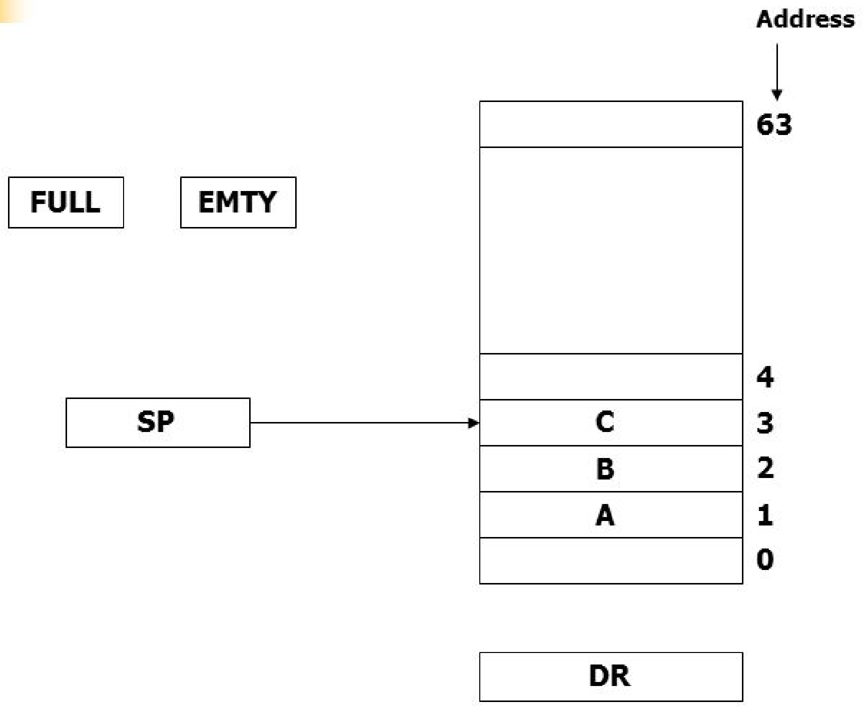
- The register that holds the address for the stack is called a stack pointer (SP) because its value always points at the top item in the stack.

- The two operations of a stack are the insertion and deletion of items. The operation of insertion is called push (or push-down) because it can be through of as the result of pushing a new item on top.

- The operation of deletion is called pop (or pop-up) because it can be thought of as the result of removing one item so that the stack pops up.

Register Stack

- A stack can be placed in portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. Diagram shows the organization of a 64-word register stack.



- Three items are placed in the stack: A, B, and C, in that order. Item C is on top of the stack so that the content of SP is now 3.

- To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of SP. Item B is now on top the stack since SP holds address 2.

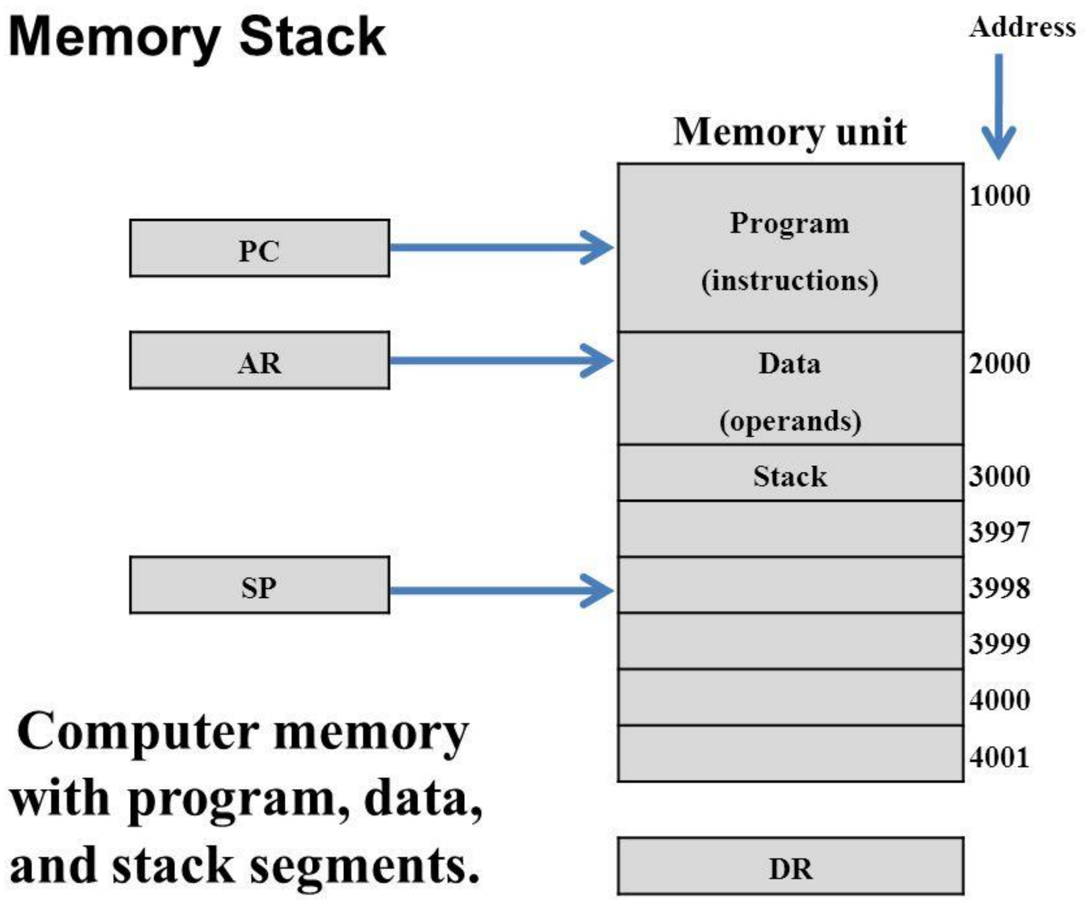
- To insert a new item, the stack is pushed by incrementing SP and writing a word in the next-higher location in the stack.

Memory Stack

- A stack can exist as a stand-alone unit or can be implemented in a random-access memory attached to a CPU.

- The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer.

- Diagram shows a portion of computer memory partitioned into three segments: program, data, and stack.



- The program counter PC points at the address of the next instruction in the program.

- The address register AR points at an array of data. The stack pointer SP points at the top of the stack.

- PC is used during the fetch phase to read an instruction. AR is used during the execute phase to read an operand. SP is used to push or pop items into or from the stack.

- The initial value of SP is 4001 and the stack grows with decreasing addresses. Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack is 3000.

- We assume that the items in the stack communicate with a data register DR. A new item is inserted with the push operation as follow:

$$SP \leftarrow SP - 1$$
$$M[SP] \leftarrow DR$$

- The stack pointer is decremented so that it points at the address of the next word. A memory write operation inserts the word form DR into the top of the stack. A new item is deleted with a pop operation as follows:

$$DR \leftarrow M[SP]$$
$$SP \leftarrow SP + 1$$

- The top is read form the stack into DR. The stack pointer is then incremented to point at the next item in the stack.

- **A stack pointer is loaded with an initial value. This initial value must be the bottom address of an assigned stack in memory. So, SP is automatically decremented or incremented with every push or pop operation.**

- **The advantage of a memory stack is that the CPU can refer to it without having to specify an address, since the address is always available and automatically updated in the stack pointer.**

Reverse Polish Notation

- A stack organization is very effective for evaluating arithmetic expressions.

- The common arithmetic expressions are written in infix notation, with each operation written between the operands. Consider the simple arithmetic expression

  A * B + C * D

- The polish mathematician Lukasiewicz showed that arithmetic expressions can be represented in prefix notation. This representation often referred to as Polish notation, places the operator before the operands.

- The postifix notation, referred to as reverse polish notation (RPN), places the operator after the operands.

- **The following examples demonstrate the three representations:**

  A + B Infix notation
  + AB Prefix or Polish notation
  AB + Postfix or reverse Polish notation

- The reverse Polish notation is in a form suitable for stack manipulation. The expression.

  A * B + C * D

is written is reverse Polish notation as

  AB * CD * +

- **The conversion from infix notation to reverse Polish notation.**

- First perform all arithmetic inside inner parentheses, then inside outer parentheses, and do multiplication and division operations before addition and subtraction operations. Consider the expression

  (A + B) * [C * (D + E) + F]

- The expression can be converted to reverse Polish notation, without the use of parentheses, by taking into consideration the operation hierarchy. The converted expression is

  AB + DE + C * F + *

Evaluation Of Arithmetic Expressions

- The procedure consists of first converting the arithmetic expression into its equivalent reverse Polish notation. The operands are pushed into the stack in the order in which they appear.

- Consider the arithmetic expression

  (3 *4) + (5 * 6)

In reverse Polish notation, it is expressed as

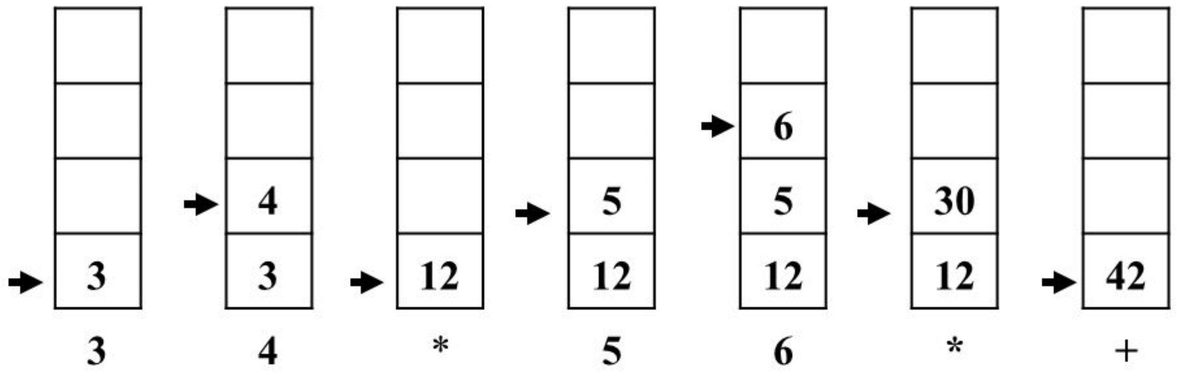
  34 * 56 * +

- Now consider the stack operating shown in diagram. Each box represents one represents one stack operation and the arrow always points to the top of the stack.

Evaluation of Arithmetic Expressions

- **The following numerical example may clarify this procedure.**
- **Consider the arithmetic expression (3 * 4) + (5 * 6)**
- **In reverse Polish notation, it is expressed as**

3 4 * 5 6 * +



Stack operations to evaluate 3 * 4 + 5 * 6

➢ First the number 3 is pushed into the stack, then the number 4. The next symbol is the multiplication operator *. This causes a multiplication of the two top most items in the stack.

➢ The stack is then popped and the product is placed on top of the stack, replacing the two original operands. Next we encounter the two operands 5 and 6, so they are pushed into the stack.

➢ The stack operation that results from the next * replaces these two numbers by their product. The last operation cause.

Instruction Formats

- The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register.

- The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:

  1. An operation code field that specifies the operation to be performed.
  2. An address field that designates a memory address or a processor register.
  3. A mode field that specifies the way the operand or the effective address is determined.

- Computers may have instructions of several different lengths containing varying number of addresses.

- The number of address fields in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organizations:

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

- To illustrate the influence of the number of addresses on computer programs, we will evaluate the arithmetic statement

  X = (A + B) * (C + D)

- Using zero, one, two, or three address instruction. We will use the symbols ADD, SUB, MUL, and DIV for the four arithmetic operations; MOV for the transfer-type operation.

- LOAD and STORE for transfers to and from memory and AC register. We will assume that the operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X.

Three-Address Instructions

- Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand.

- The program in assembly language that evaluates X =(A + B) ∗ (C + D) is shown below

```
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
```

- It is assumed that the computer has two processor registers, R1 and R2. The symbol M [A] denotes the operand at memory address symbolized by A.

Two-Address Instructions

- Two address instructions are the most common in commercial computers. Here again each address field can specify either a processor register or a memory word. 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
```

One-Address Instructions

- One-address instructions use an implied accumulator (AC) register for all data The program to evaluate X = (A + B) * (C + D) is

```
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 operation are done between the AC register and a memory operand. T is the address of a temporary memory location required for storing the intermediate result.

Zero-Address Instructions

- A stack-organized computer does not use an address field for the instructions ADD and MUL.
- The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack. The following program shows how X = (A + B) ∗ (C + D) will be written for a stack organized computer. (TOS stands for top of 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
```

- To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polish notation.

- The name “zero-address” is given to this type of computer because of the absence of an address field in the computational instructions,

Addressing Modes

- The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

- Following are various addressing modes.

1. Implied Mode

- In this mode the operands are specified implicitly in the definition of the instruction. For example, the instruction “complement accumulator” is an implied-mode instruction.

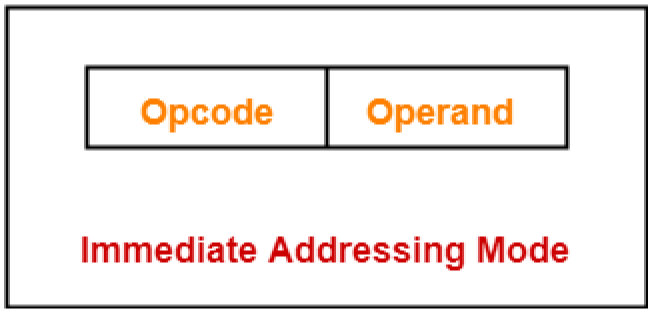
- All register reference instructions that use an accumulator are implied-mode instructions.

- Zero-address instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

2. Immediate Mode

- In this mode the operand is specified in the instruction itself.
- An immediate-mode instruction has an operand field rather than an address field.

- The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction.



Example:

- ADD 10 will increment the value stored in the accumulator by 10.
- MOV R #20 initializes register R to a constant value 20.

3. Register Mode

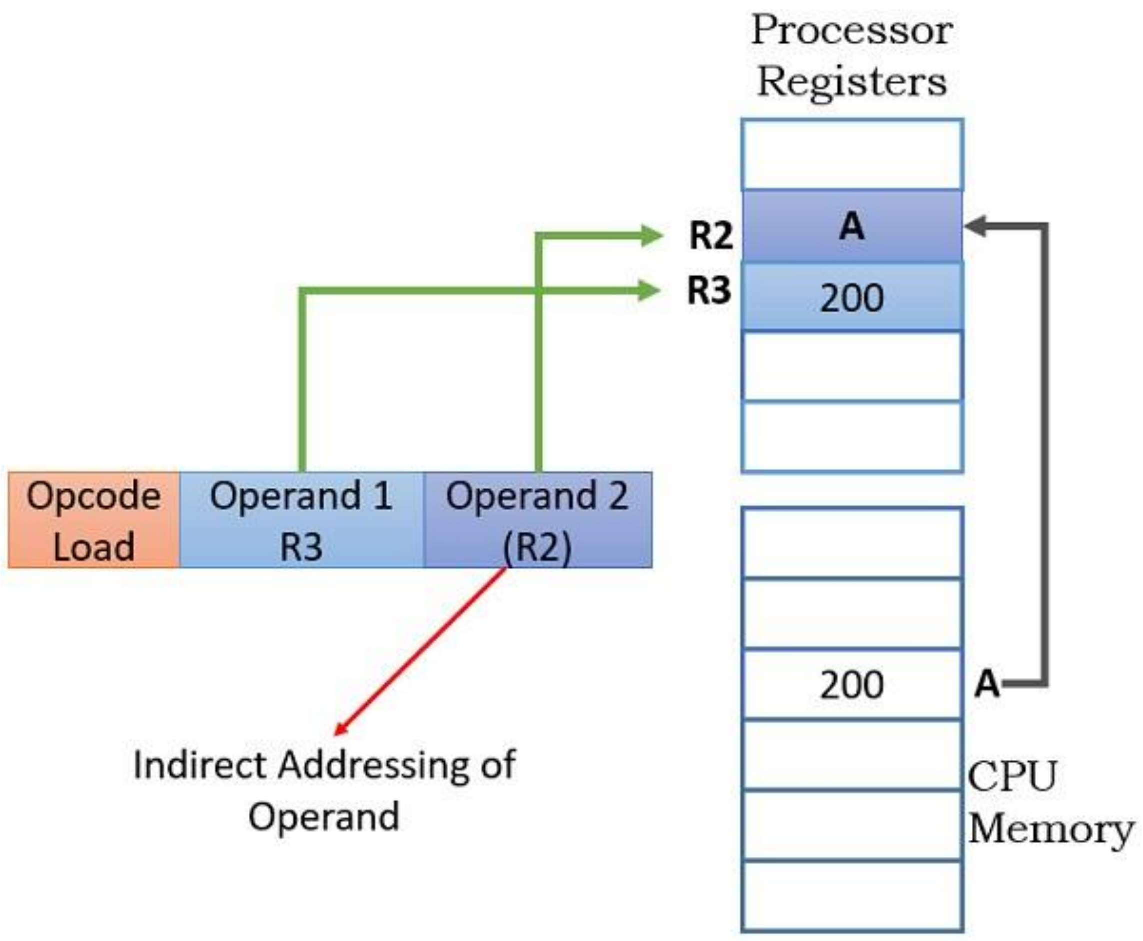
- In this mode the operands are in registers that reside within the CPU.

- The particular register is selected from a register field in the instruction. A k-bit field can specify any one of 2^k registers.

4. Register Indirect Mode

- In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory.

- In other words, the selected register contains the address of the operand rather than the operand itself.



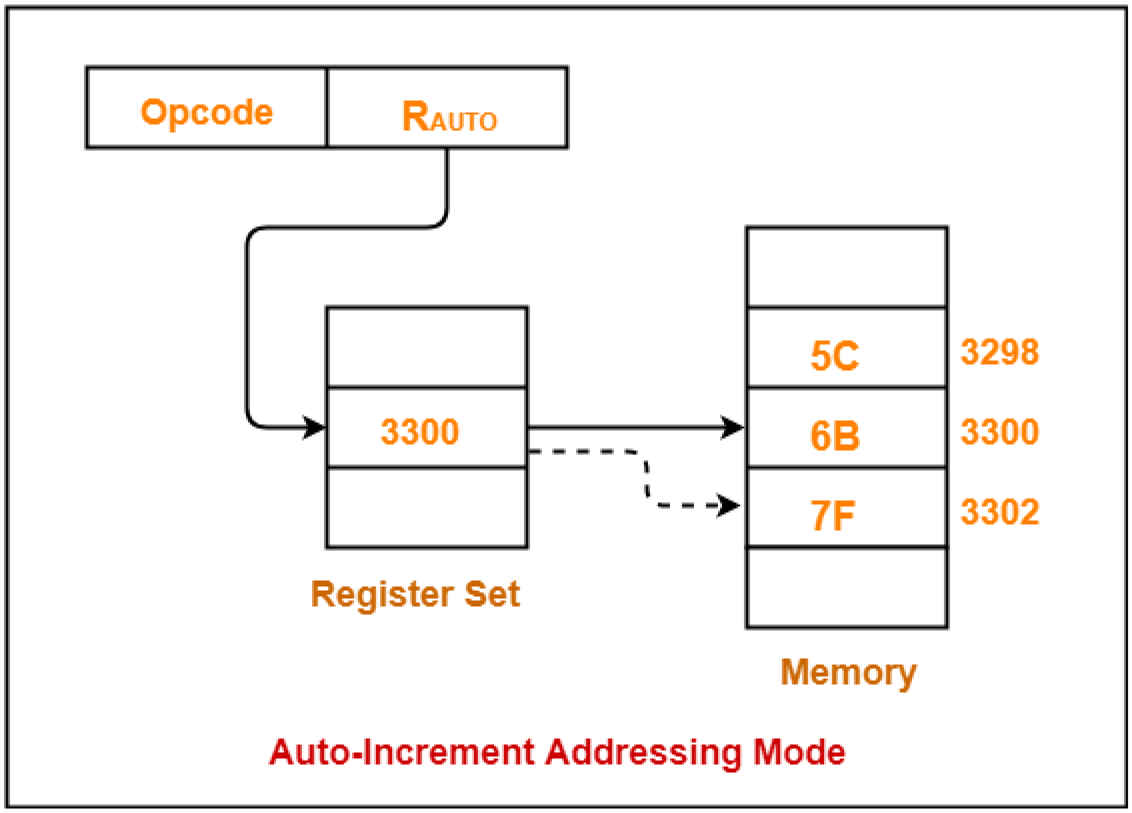
- Before using a register indirect mode instruction, the programmer must ensure that the memory address of the operand is placed in the processor register with a previous instruction.

5. Auto increment Mode:

- The register is incremented or decremented after (or before) its value is used to access memory. When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table.

  Effective Address of the Operand = Content of Register

- This can be achieved by using the increment or decrement instruction or sometimes it is auto incremented.



Assume operand size = 2 bytes.

Here,

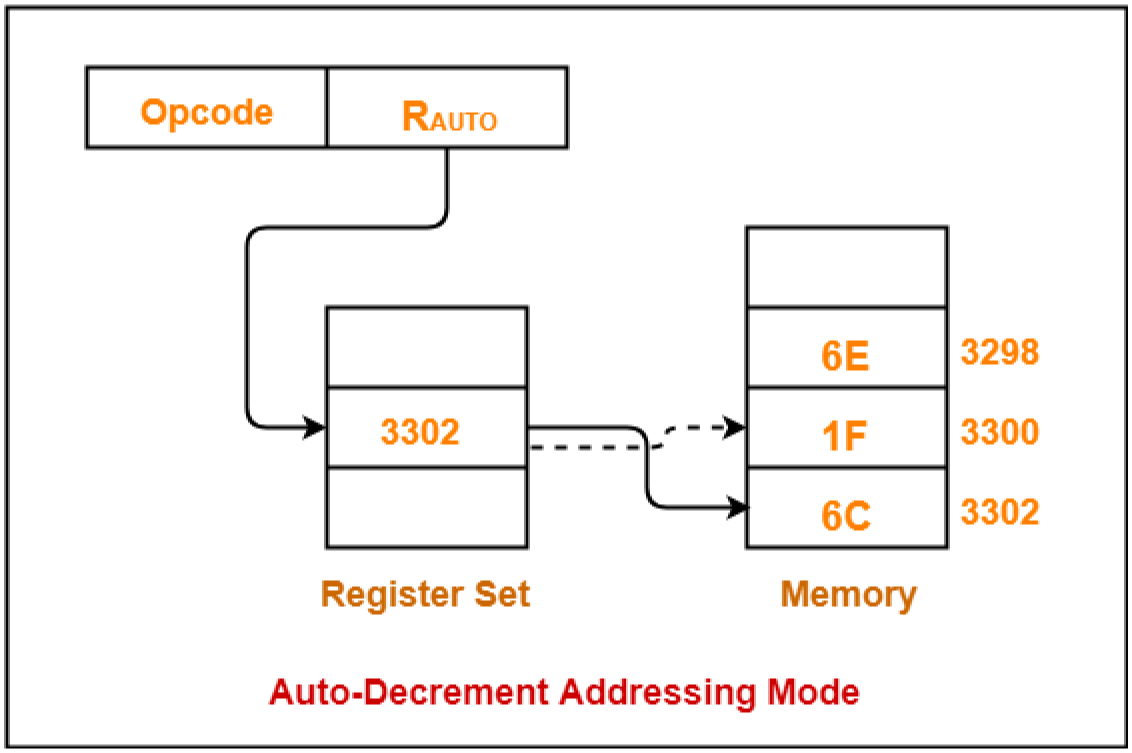
- After fetching the operand 6B, the instruction register RAUTO will be automatically incremented by 2.
- Then, updated value of RAUTO will be 3300 + 2 = 3302.
- At memory address 3302, the next operand will be found.

6. Auto decrement

- This addressing mode is again a special case of Register Indirect Addressing Mode where-

Effective Address of the Operand

= Content of Register – Step Size



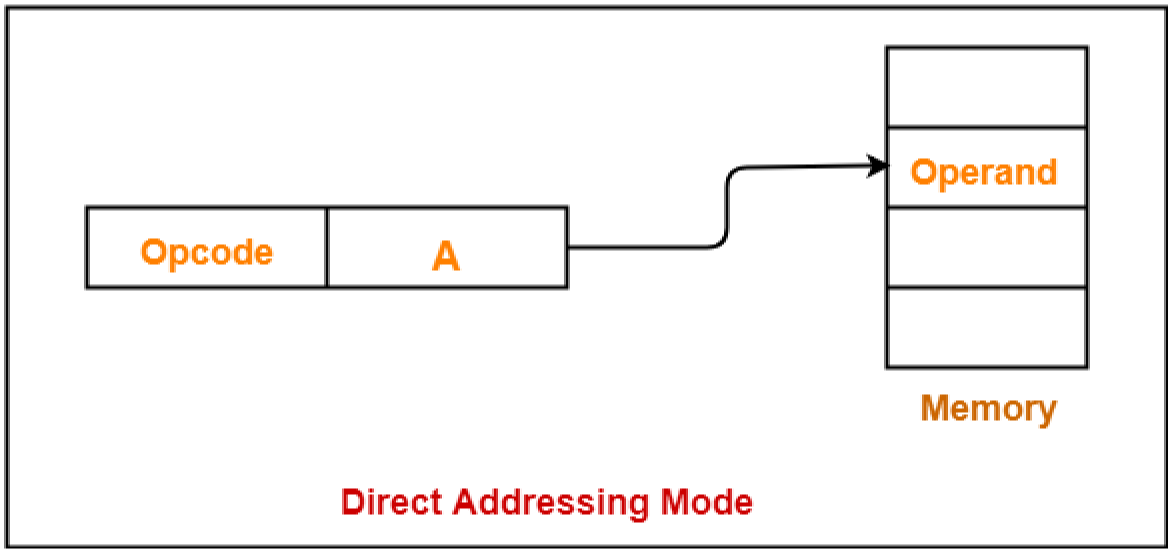
Assume operand size = 2 bytes.

Here,

- First, the instruction register RAUTO will be decremented by 2.
- Then, updated value of RAUTO will be 3302 – 2 = 3300.
- At memory address 3300, the operand will be found.

7. Direct Address Mode

- In this mode the effective address is equal to the address part of the instruction.



- The operand resides in memory and its address is given directly by the address field of the instruction. In a branch-type instruction the address field specifies the actual branch address.

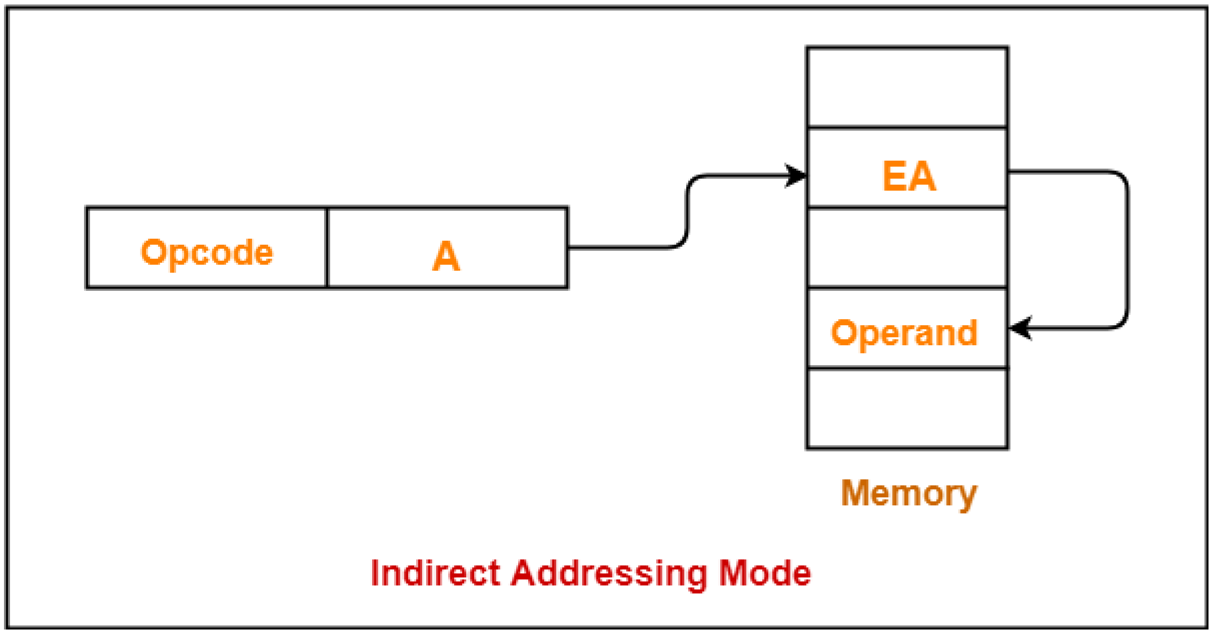
Example

ADD X will increment the value stored in the accumulator by the value stored at memory location X.

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

8. Indirect Address Mode

- In this mode the address field of the instruction gives the address where the effective address is stored in memory.



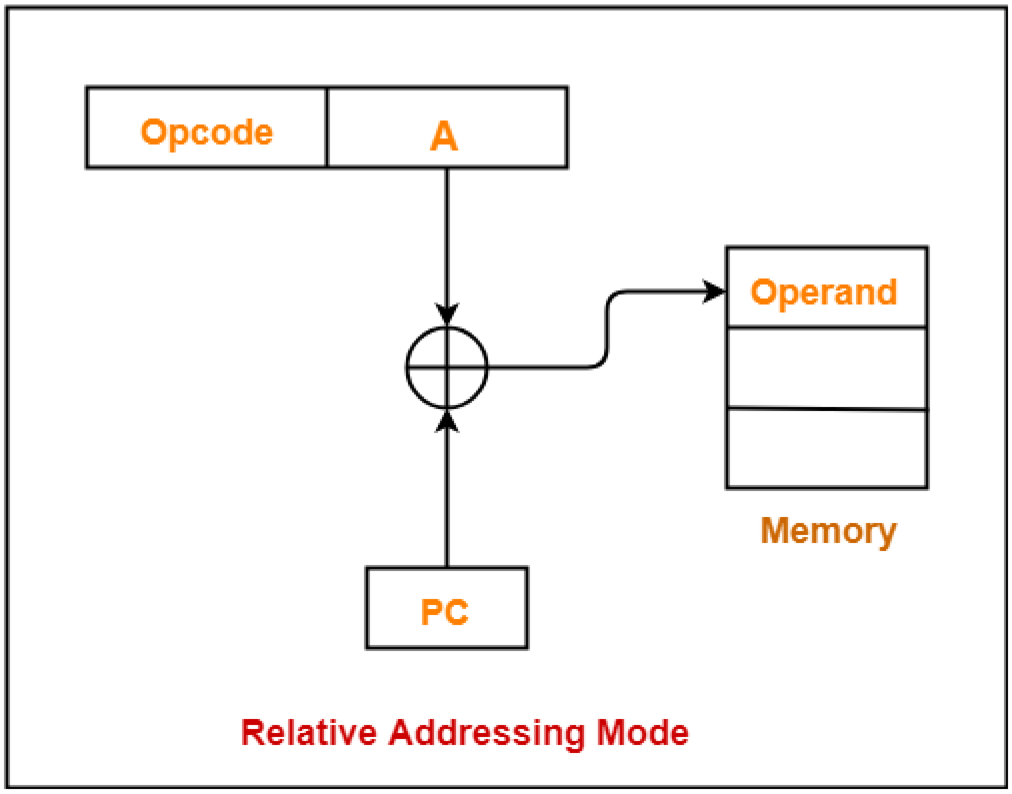
- Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

9. Relative Address Mode

- In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address. The address part of the instruction is usually a signed number (in 2's complement representation) which can be either positive or negative.

- **Effective Address**
  = Content of Program Counter + Address part of the instruction

- When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction.



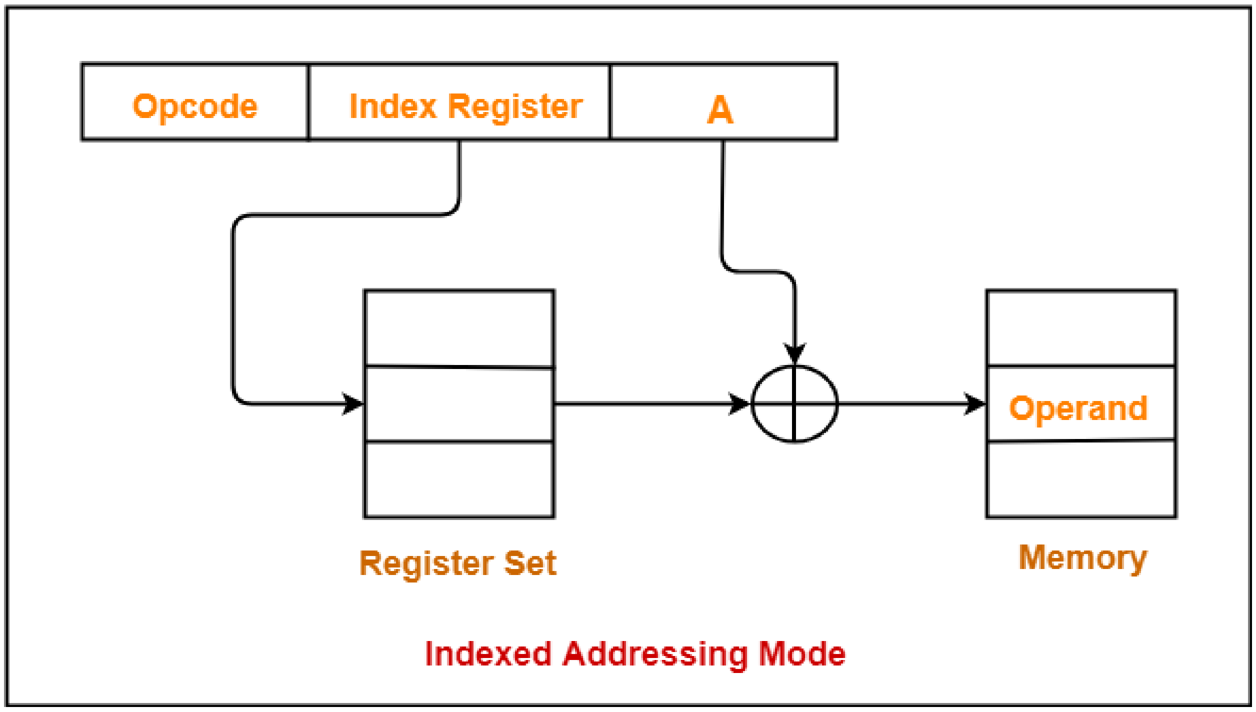
- To clarify **with an example**, assume that the program counter contains the number 825 and the address part of the instruction contains the number 24. The instruction at location 825 is read from memory during the fetch phase and the program counter is then incremented by one to 826 + 24 = 850. This is 24 memory locations forward from the address of the next instruction.

10. Indexed Addressing Mode

- In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.

Effective Address

= Content of Index Register + Address part of the instruction



- The index register is a special CPU register that contains an index value.

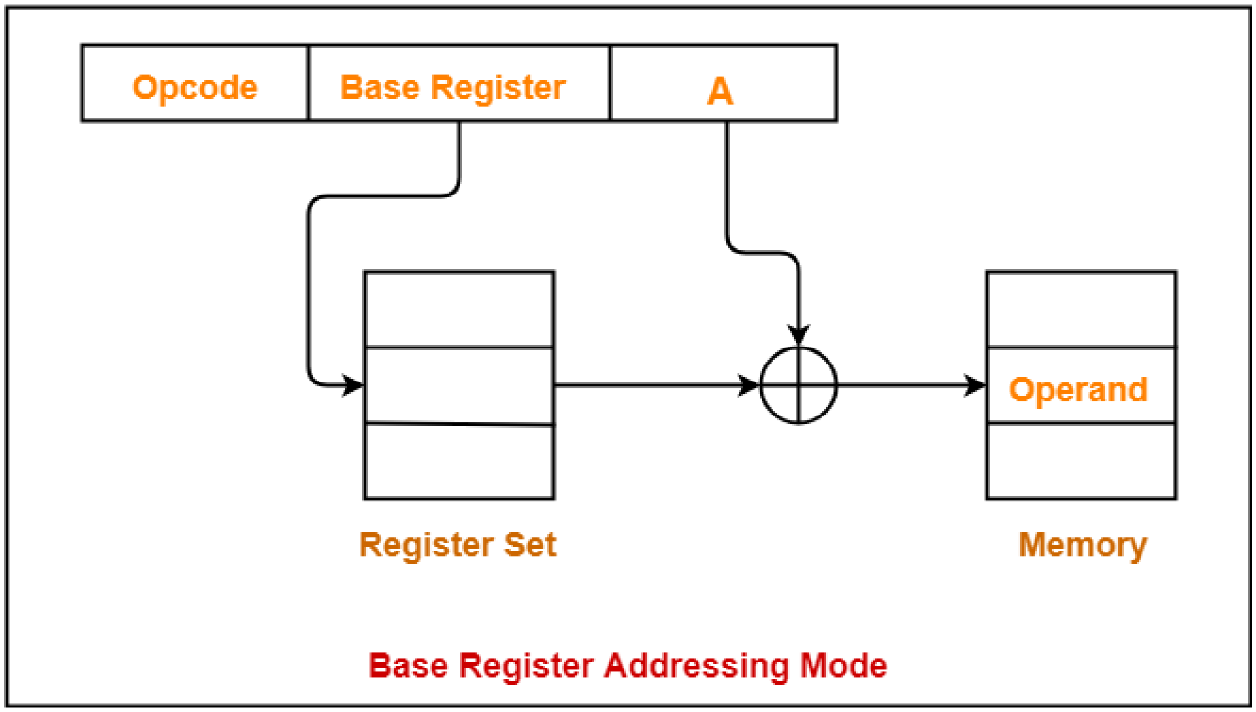
11. Base Register Addressing Mode:

- In this mode the content of a base register is added to the address part of the instruction to obtain the effective address.

Effective Address

= Content of Base Register + Address part of the instruction

- This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register.



- The difference between the two modes is, an index register is assumed to hold an index number that is relative to the address part of the instruction. A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address.

Applications of Addressing Modes

	Addressing Modes	Applications
1	**Immediate Addressing Mode**	• To initialize registers to a constant value
2	**Direct Addressing Mode** **and** **Register Direct Addressing Mode**	• To access static data • To implement variables
3	**Indirect Addressing Mode** **and** **Register Indirect Addressing Mode**	• To implement pointers because pointers are memory locations that store the address of another variable • To pass array as a parameter because array name is the base address and pointer is needed to point the address
4	**Relative Addressing Mode**	• For program relocation at run time i.e. for position independent code

		• To change the normal sequence of execution of instructions • For branch type instructions since it directly updates the program counter
5	**Index Addressing Mode**	• For array implementation or array addressing • For records implementation
6	**Base Register Addressing Mode**	• For writing relocatable code i.e. for relocation of program in memory even at run time • For handling recursive procedures
7	**Auto-increment Addressing Mode and Auto-decrement Addressing Mode**	• For implementing loops • For stepping through arrays in a loop • For implementing a stack as push and pop

Data Transfer and Manipulation

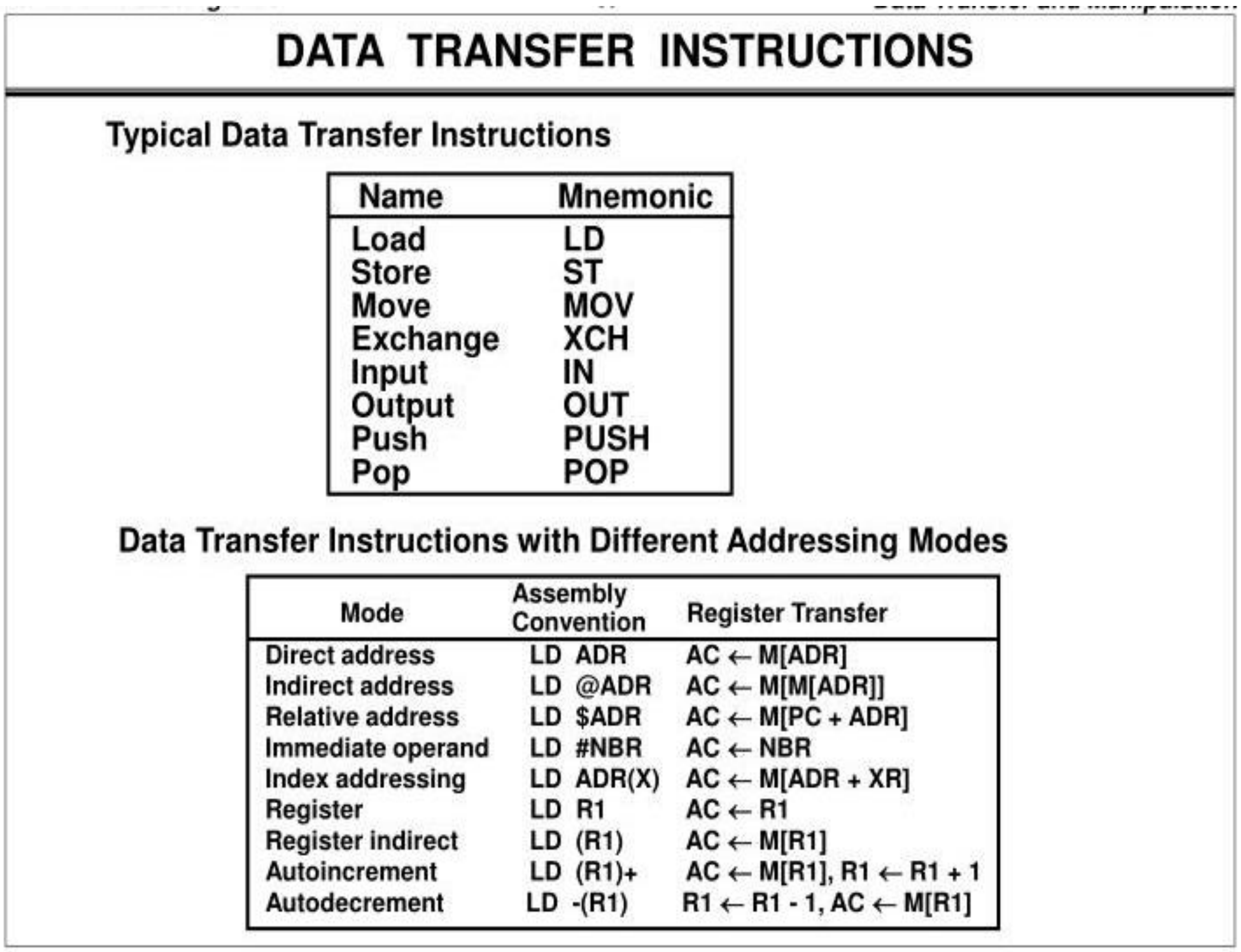
- Most computer instruction can be classified into three categories:

    1. Data transfer instruction
    2. Data manipulation
    3. Program control instructions

1. Data transfer instruction

- Data transfer instruction cause of data from one location to another without changing the binary information content.

- Data manipulation instructions are those that perform arithmetic, logic ,shift operation



DATA TRANSFER INSTRUCTIONS

Typical Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

Data Transfer Instructions with Different Addressing Modes

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD $ADR	$AC \leftarrow M[PC + ADR]$
Immediate operand	LD #NBR	$AC \leftarrow NBR$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR + XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1)+	$AC \leftarrow M[R1], R1 \leftarrow R1 + 1$
Autodecrement	LD -(R1)	$R1 \leftarrow R1 - 1, AC \leftarrow M[R1]$

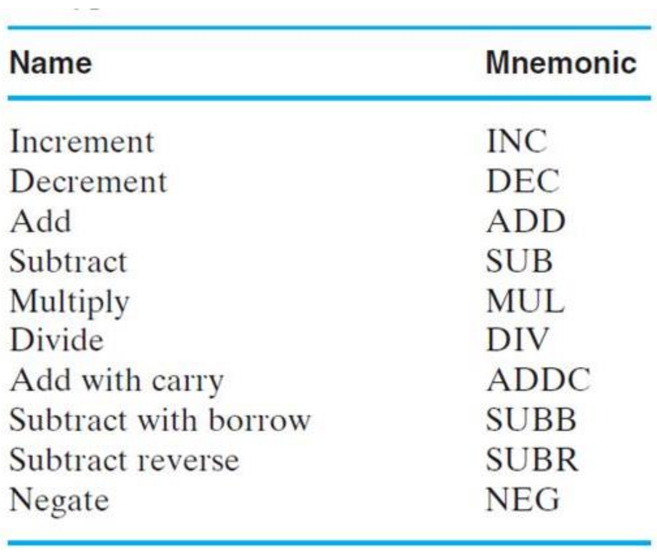
2. Data manipulation Instruction

- Basically data manipulation instructions in a typical computer usually divided into three basic types:

  1. Arithmetic Instructions
  2. Logical and bit manipulation instructions
  3. Shift instructions

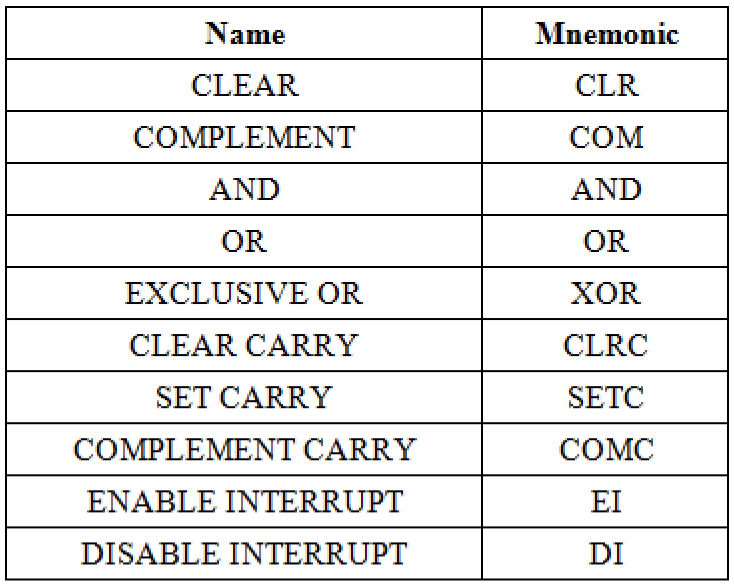
1. Arithmetic Instructions

- The four basic arithmetic instructions are addition, subtraction, multiplication and division.



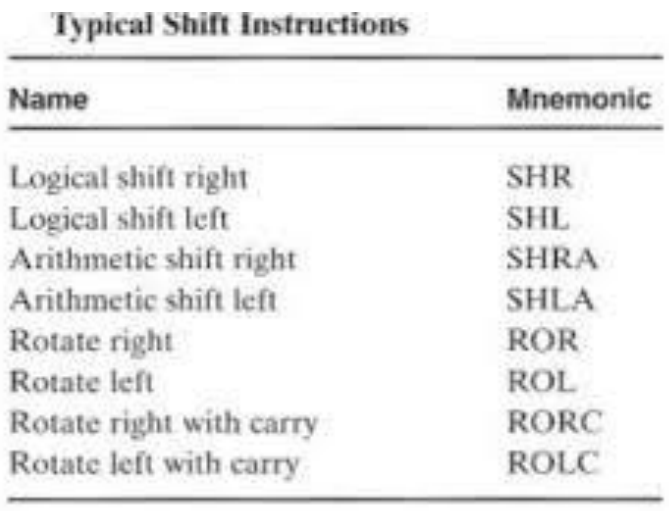
Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Subtract reverse	SUBR
Negate	NEG

2. Logical and bit manipulation instructions



Name	Mnemonic
CLEAR	CLR
COMPLEMENT	COM
AND	AND
OR	OR
EXCLUSIVE OR	XOR
CLEAR CARRY	CLRC
SET CARRY	SETC
COMPLEMENT CARRY	COMC
ENABLE INTERRUPT	EI
DISABLE INTERRUPT	DI

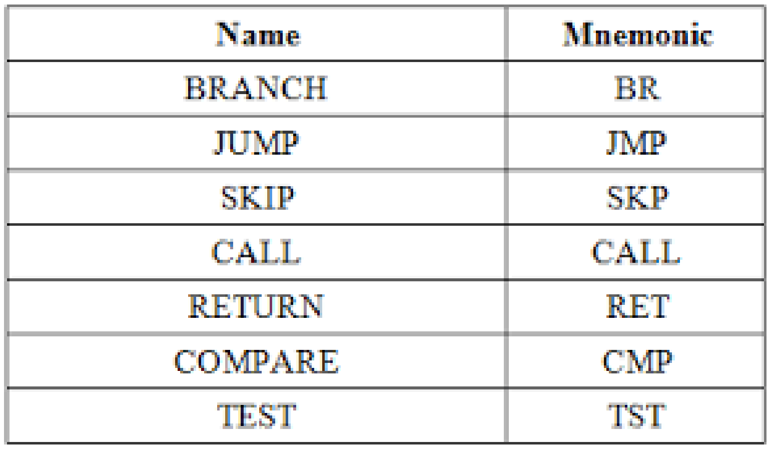
3. Shift instructions


Typical Shift Instructions

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right with carry	RORC
Rotate left with carry	ROLC

3. Program control instructions

- The program control instructions direct the flow of a program and allow the flow of the program to change. A change in flow often occurs when decisions, made with the CMP or TEST instruction, are followed by a conditional jump instruction.

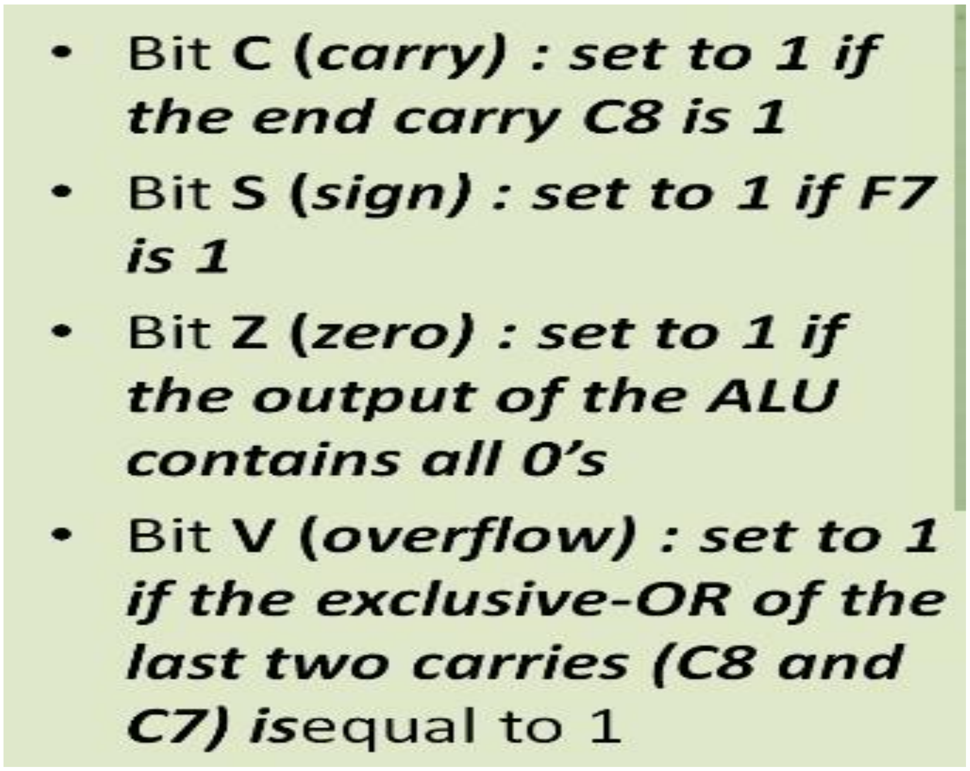


Name	Mnemonic
BRANCH	BR
JUMP	JMP
SKIP	SKP
CALL	CALL
RETURN	RET
COMPARE	CMP
TEST	TST

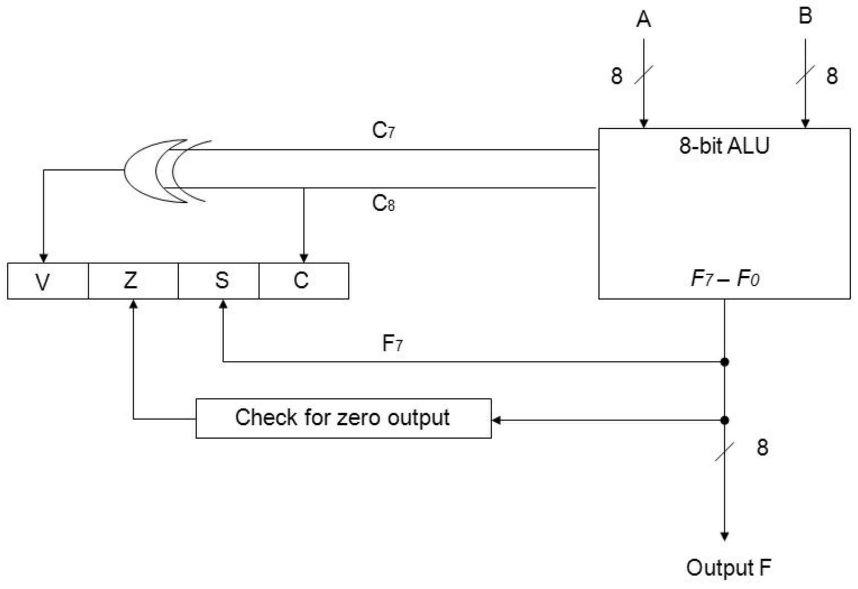
Status Bit Conditions

- The four status bits are symbolized by C, S, Z and V.

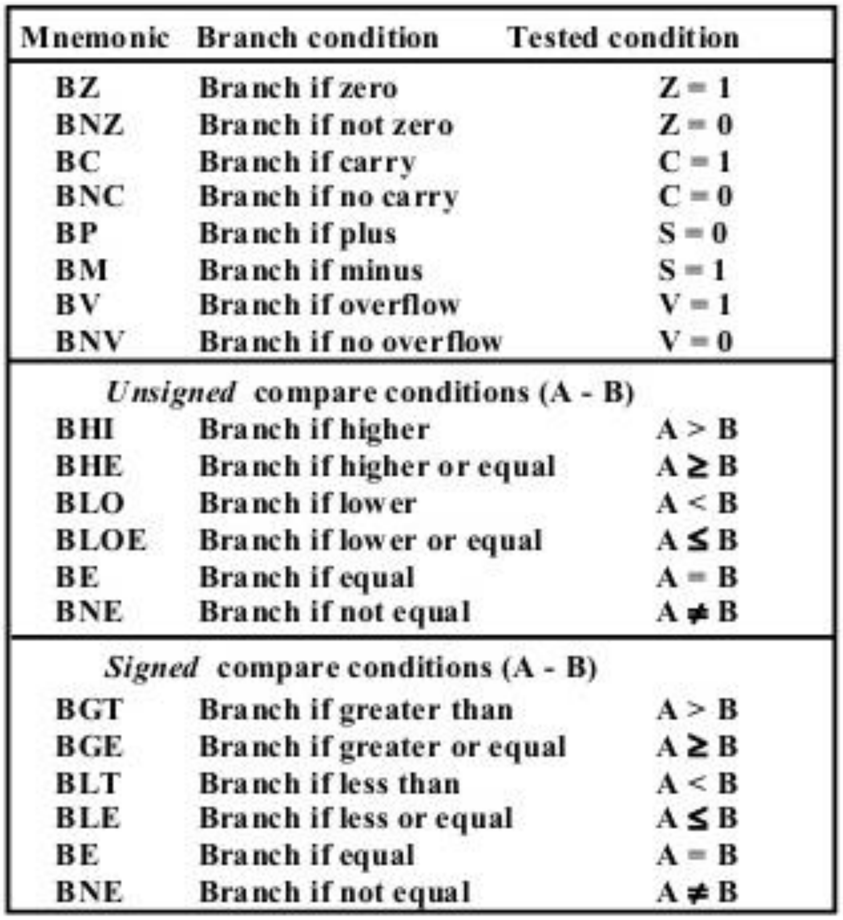
- The bits are set or cleared as a result of an operation performed in the ALU.



- Bit **C** ***(carry) : set to 1 if the end carry C8 is 1***
- Bit **S** ***(sign) : set to 1 if F7 is 1***
- Bit **Z** ***(zero) : set to 1 if the output of the ALU contains all 0's***
- Bit **V** ***(overflow) : set to 1 if the exclusive-OR of the last two carries (C8 and C7) is***equal to 1



Conditional branch Instructions



Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	Z = 1
BNZ	Branch if not zero	Z = 0
BC	Branch if carry	C = 1
BNC	Branch if no carry	C = 0
BP	Branch if plus	S = 0
BM	Branch if minus	S = 1
BV	Branch if overflow	V = 1
BNV	Branch if no overflow	V = 0
	Unsigned compare conditions (A - B)	
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
	Signed compare conditions (A - B)	
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

SUBROUTINES

<u>Subroutine Call and Return</u>

- In a given program, it is often necessary to perform a particular subtask many times on different data-values. Such a subtask is usually **called a subroutine**.

- When a program branches to a subroutine we say that it is calling the subroutine. The instruction that performs this branch operation is named a **Call instruction.**

- After a subroutine has been executed, the calling program must resume execution, the subroutine is said to return to the program that called it by executing a **Return instruction.**

- The Call instruction is just a special branch instruction that performs the following operations

  • Store the contents of the PC in temporary register.

  • Branch to the target address specified by the instruction The Return instruction is a special branch instruction that performs the operation

- Different computers have different temporary location for storing return address. The most efficient way is to store return address i**n memory stack.**

- A subroutine call is implemented with following micro operations.

SP ← SP-1	Decrement stack pointer
M [SP] ← PC	Push content of PC on to stack
PC ← Effective address	Transfer control to subroutine

- The instruction returns from the last subroutine is implemented by following micro operations.

PC ← M [SP]	Pop stack and transfer to PC
SP ← SP+1	Increment stack pointer

<u>**Program Interrupt**</u>

- An event external to the currently executing process that causes a change in the normal flow of instruction execution; usually generated by hardware devices external to the CPU.

- After a program has been interrupted and the service routine been executed, the CPU must return to exactly the same state that it was when the interrupt occurred.

- The state of the CPU at the end of the execute cycle is determined from

    1. The content of the program counter
    2. The content of all processor register
    3. The content of status conditions

- The collection of all status bit conditions in the CPU is called **program status word or PSW.** The PSW is stored in separate register, it includes the status bits from the last ALU operation and it specifies the interrupts that are allowed to occur.

- When CPU is executing a program that is part of the operating system, it is said to be a **supervisor or system mode.**

Types of Interrupt

- Generally there are three types o Interrupts those are Occurred.
    1. Internal Interrupt
    2. Software Interrupt
    3. External Interrupt

External Interrupt

- These types of interrupts generally come from external input / output devices which are connected externally to the processor.
- **For Example** When a Program is executed and when we move the Mouse on the Screen then the CPU will handle this External interrupt first and after that he will resume with his Operation.

Internal Interrupt

- The Internal Interrupts are those which are occurred due to some Problem in the Execution They are also known as traps and their causes could be due to some illegal operation or the erroneous use of data.

- For Example When a user performing any Operation which contains any type of Error.

Software Interrupt

- The interrupts which are caused by the software instructions are called software interrupts.