UNIT- III Syllabus:

Central Processing Unit: General Register Organization, STACK Organization. Instruction Formats, Addressing Modes, Data Transfer and Manipulation, Program Control, Reduced Instruction Set Computer.
Microprogrammed Control: Control Memory, Address Sequencing, Micro Program example, Design of Control 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 Figure (1). The register set stores intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) performs the required microoperations 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.

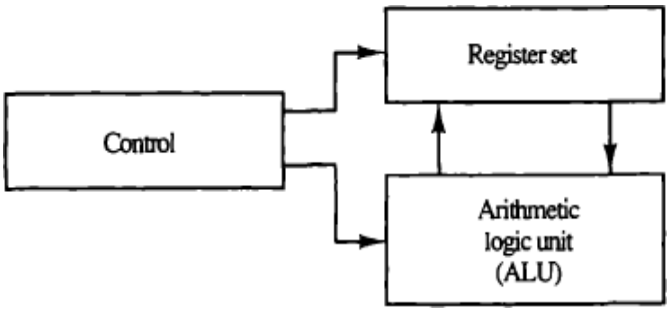


Figure (1): Major components of CPU

One boundary where the computer designer and the computer programmer see the same machine is the part of the CPU associated with the instruction set.

- From the designer's point of view, the computer instruction set provides the specifications for the design of the CPU. The design of a CPU is a task that in large part involves choosing the hardware for implementing the machine instructions.
- The user who programs the computer in machine or assembly language must be aware of the register set, the memory structure, the type of data supported by the instructions, and the function that each instruction performs.

The following sections describe the organization and architecture of the CPU with an emphasis on the user's view of the computer, how the registers communicate with the ALU through buses, explain the operation of the memory stack, the type of instruction formats available, the addressing modes used to retrieve data from memory, and also the concept of reduced instruction set computer (RISC).

General Register Organization:

- We know that the memory locations are needed for storing pointers, counters, return addresses, temporary results, and partial products during multiplication. Having to refer to memory locations for such applications is time consuming because memory access is the most time-consuming operation in a computer.

- It is more convenient and more efficient to store these intermediate values in processor registers. When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system.
- The registers communicate with each other not only for direct data transfers, but also while performing various microoperations. Hence it is necessary to provide a common unit that can perform all the arithmetic, logic, and shift microoperations in the processor.

A bus organization for seven CPU registers is shown in the below figure.

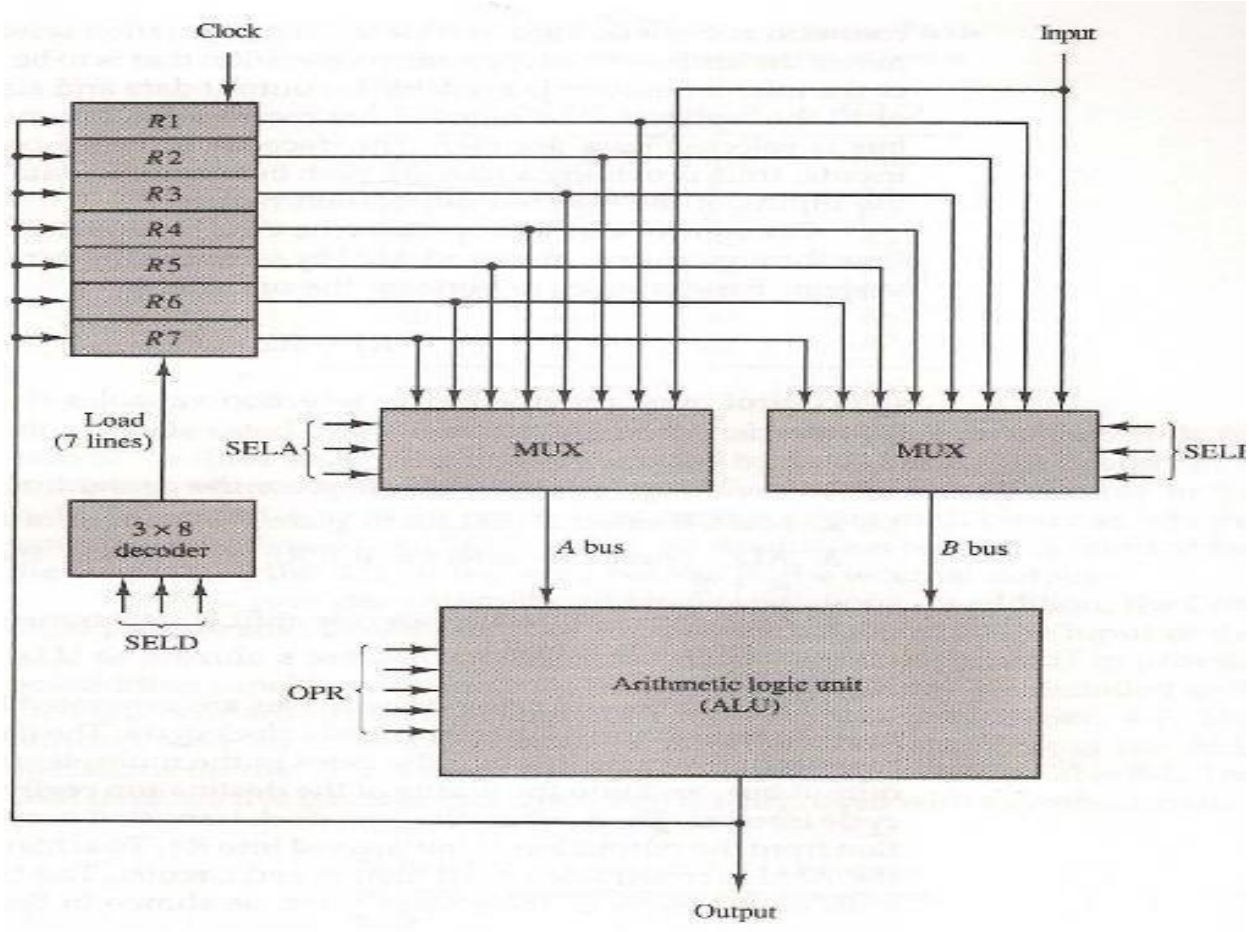


Figure (2): Bus organization for CPU registers

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 microoperation that is to be performed. The result of the microoperation 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.

The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system. For example, to perform the operation

R 1 ← 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 R 3 into bus B.
3 . ALU operation selector (OPR): to provide the arithmetic addition A + B.

4. Decoder destination selector (SELD): to transfer the content o f 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 Figure (3). 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 outputs. 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 microoperation.

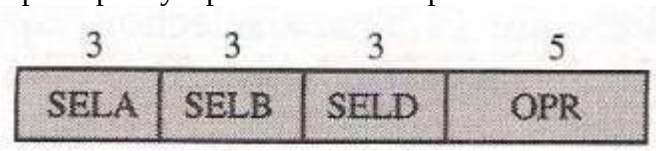


Figure (3): Control word

The encoding of the register selections is specified in the Table (a). The 3-bit binary code listed in the first column of the table specifies the binary code for each of the three fields. The register selected by fields SELA, SELB, and SELD is the one whose decimal number is equivalent to the binary number in the code.

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.

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

Table (1): Encoding of Register Selection Fields

The ALU provides arithmetic and logic operations. The encoding of the ALU Operations are specified in the Table (b). 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

Table (2): Encoding of ALU operations

Examples of Microoperations:

A control word of 14 bits is needed to specify a rnicrooperation in the CPU. The control word for a given microoperation can be derived from the selection variables. For example, the subtract rnicrooperation given by the statement

R 1 ← R 2 – 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. Thus the control word is specified by the four fields and the corresponding binary value for each field is obtained from the encoding listed in Tables (1) and (2). The binary control word for the subtract rnicrooperation is 010 011 001 00101 and is obtained as follows:

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

The control word for this microoperation and a few others are listed in the below table.

	Symbolic Designation				
Microoperation	SELA	SELB	SELD	OPR	Control Word
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \vee R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	—	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	R1	—	R7	TSFA	001 000 111 00000
Output $\leftarrow R2$	R2	—	None	TSFA	010 000 000 00000
Output ← Input	Input	—	None	TSFA	000 000 000 00000
$R4 \leftarrow$ sh1 $R4$	R4	—	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

Table (3): Encoding of ALU operations

STACK Organization:

A useful feature that is included in the CPU of most computers is a stack or last-in, first-out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved.

- The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.

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.

1. Push or push-down (insertion operation)
2. Pop or pop-up (deletion operation)

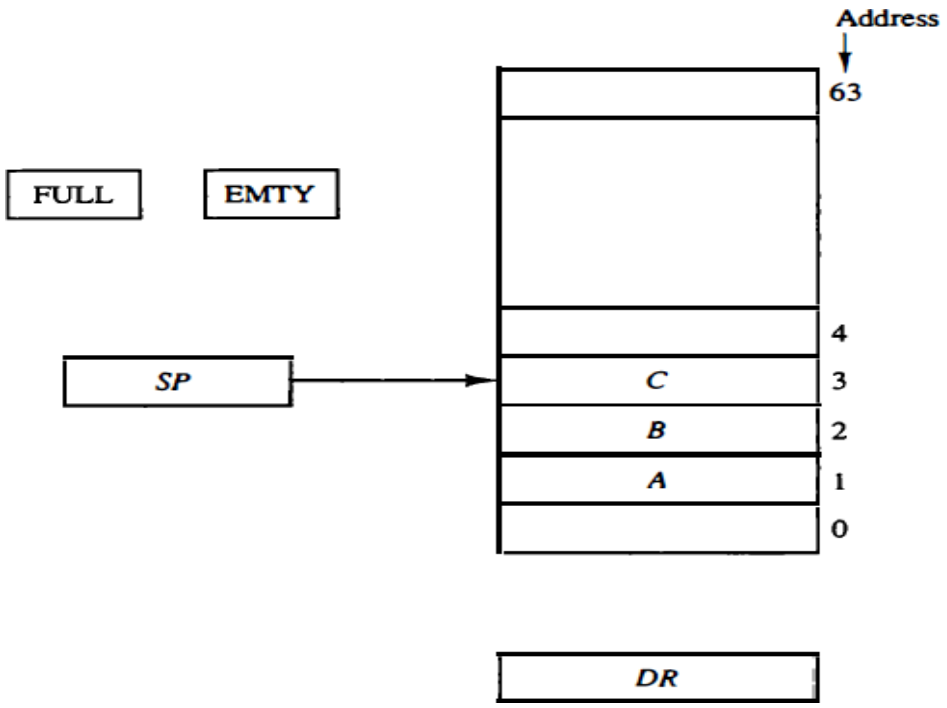


Figure (4): the organization of a 64-word register stack.

Register Stack

A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. Figure (3) shows the organization of a 64-word register stack. The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack.

In a 64-word stack, the stack pointer contains 6 bits because $2^6 = 64$. Since SP has only six bits, it cannot exceed a number greater than 63 (111111 in binary). When 63 is incremented by 1, the result is 0 since 111111 + 1 = 1000000 in binary, but SP can accommodate only the six least significant bits.

Similarly, when 000000 is decremented by 1, the result is 111111. The one-bit register FULL is set to 1 when the stack is full, and the one-bit register EMPTY is set to 1 when the stack is empty of items. DR is the data register that holds the binary data to be written into or read out of the stack.

Push:

Initially, SP is cleared to 0, EMPTY is set to 1, and FULL is cleared to 0, so that SP points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if FULL = 0), a new item is inserted with a push operation. The push operation is implemented with the following sequence of microoperations;

SP ← SP + 1	**Increment stack pointer**
M [SP] ← DR	**Write item on top of the stack**
If (SP = 0) then (FULL ←1)	**Check if stack is full**
EMPTY ← 0	**Mark the stack not empty**

Pop:

A new item is deleted from the stack if the stack is not empty (if EMPTY = 0). The pop operation consists of the following sequence of microoperations:

DR ← M [SP]	**Read item from the top of stack**
SP ← SP – 1	**Decrement stack pointer**
If (SP = 0) then (EMTY ←1)	**Check if stack is empty**
FULL ← 0	**Mark the stack not full**

Memory Stack

A stack can exist as a stand-alone unit as in Figure (3) 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. Figure (4) shows a portion of computer memory partitioned into three segments: program, data, and stack.

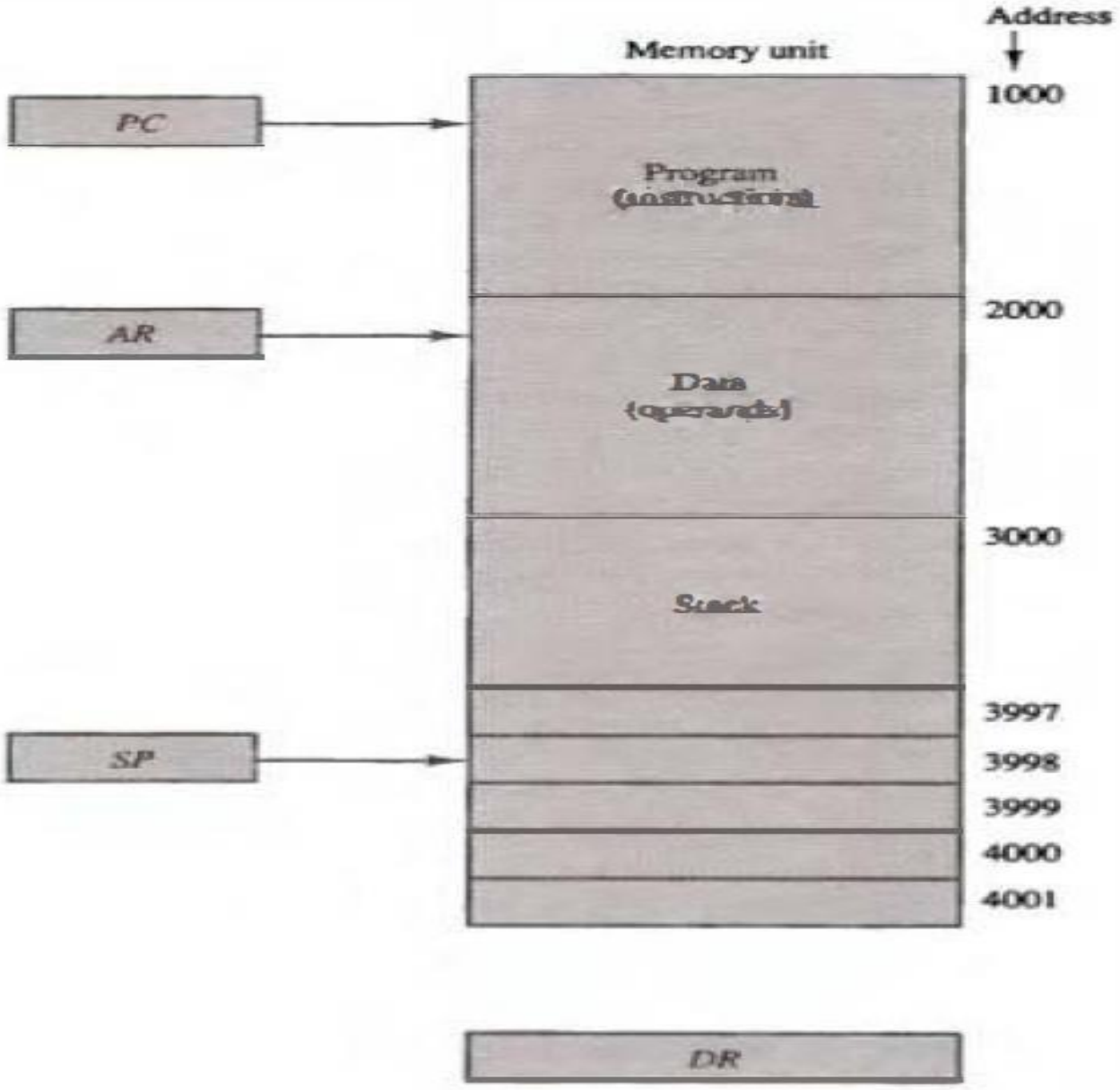


Figure (5): Computer memory with program, data and stack segments

Reverse Polish Notation

A stack organization is very effective for evaluating arithmetic expressions. The common mathematical method of writing arithmetic expressions imposes difficulties when evaluated by a computer.

A * B + C * D ← infix notation

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 postfix 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 in reverse Polish notation as

AB * CD * +

Conversion to Reverse Polish Notation

The conversion from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for infix notation.

- **This hierarchy dictates that we first perform all arithmetic inside inner parentheses, then inside outer parentheses, and do multiplication and division operations before addition and subtraction operations.**

Let I be an algebraic expression written in infix notation. I may contain parentheses, operands, and operators. For simplicity of the algorithm we will use only +, –, *, /, % operators. The precedence of these operators can be given as follows:

Higher priority *, /, %
Lower priority +, –

No doubt, the order of evaluation of these operators can be changed by making use of parentheses. For example, if we have an expression A + B * C, then first B * C will be done and the result will be added to A. But the same expression if written as, (A + B) * C, will evaluate A + B first and then the result will be multiplied with C. Consider the expression

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

The converted expression is

AB + CDE + * F + *

Step 1: Add ")" to the end of the infix expression

Step 2: Push "(" on to the stack

Step 3: Repeat until each character in the infix notation is scanned

IF a "(" is encountered, push it onto the stack

IF an operand (whether a digit or a character) is encountered, add it to the postfix expression.

IF a ")" is encountered, then

a. Repeatedly pop from stack and add it to the postfix expression until a "(" is encountered.
b. Discard the "(". That is, remove the (from stack and do not add it to the postfix expression

IF an operator 'O' is encountered, then

a. Repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression which has the same precedence or a higher precedence than 'O'
b. Push the operator 'O' to the stack

Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty

Example 1: A*B+C*D, first add ")" to the given expression i.e., A*B+C*D) and also push "(" onto the stack.

InfixCharacter Scanned	Stack	Postfix Expression
	(	
A	(	A
*	(*	A
B	(*	AB
+	(+	AB*
C	(+	AB*C
*	(+*	AB*C
D	(+*	AB*CD
)	(+*	AB*CD*+

Example 2: (A + B) * (C * (D + E) + F)

➔ First add ")" to the given expression i.e., (A + B) * (C * (D + E) + F)) and also push "(" onto the stack.

Infix Character Scanned	Stack	Postfix Expression
	(	
(	((	
A	((	A
+	((+	A
B	((+	AB
)	(	AB+
*	(*	AB+
(	(*(	AB+
C	(*(	AB+C
*	(*(*	AB+C
(	(*(*(	AB+C
D	(*(*(	AB+CD
+	(*(*(+	AB+CD
E	(*(*(+	AB+CDE

)	(*(*	AB+CDE+
+	(*(+	AB+CDE+*
F	(*(+	AB+CDE+*F
)	(*	AB+CDE+*F+
)		AB+CDE+*F+*

Evaluation of Arithmetic Expressions

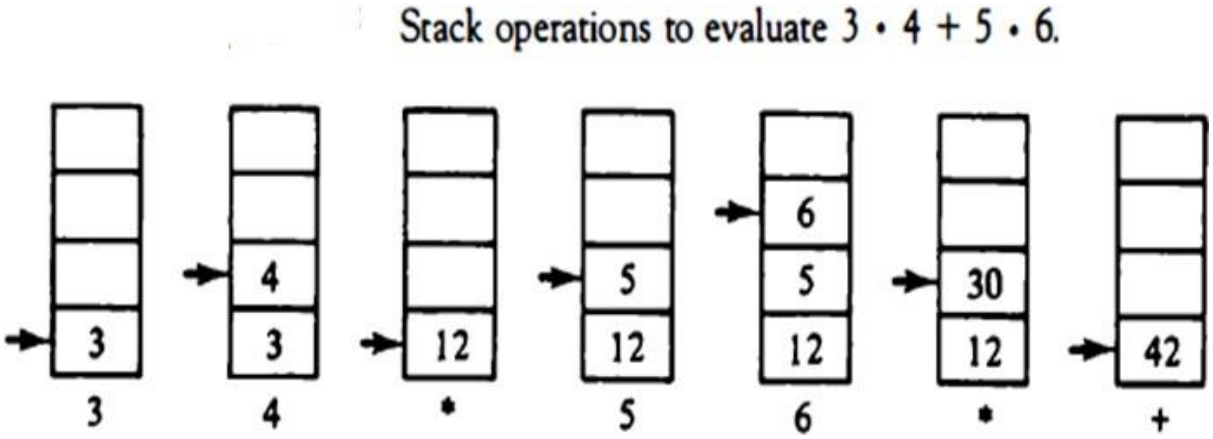
(1) Push the operands into the stack until an operator is reached
(2) Pop the top two operands from the stack, compute the result and also push the result back into the stack.
(3) Continue this process until there are no more operators in the RPN and the final result is in the stack.

The following numerical example may clarify this procedure. Consider the arithmetic expression

(3 * 4) + (5 * 6)

In reverse Polish notation, it is expressed as

34 * 56 * +



Instruction Formats:

A computer will usually have a variety of instruction code formats. It is the function of the control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction.

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.

Other special fields are sometimes employed under certain circumstances, as for example a field that gives the number of shifts in a shift-type instruction.

- The operation code field of an instruction is a group of bits that define various processor operations, such as add, subtract, complement, and shift.

- The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address.

In this section we are concerned with the address field of an instruction format and consider the effect of including multiple address fields in an instruction.

Operations specified by computer instructions are executed on some data stored in **memory** or **processor registers**. Operands residing in memory are specified by their **memory address**. Operands residing in processor registers are specified with a **register address**. A register address is a binary number of k bits that defines one of 2^kregisters in the CPU.

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.

1. An accumulator-type organization:

All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field. For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as:

ADD X

where X is the address of the operand. The ADD instruction in this case results in the operation **AC ← AC + M[X].** AC is the accumulator register and M [X] symbolizes the memory word located at address X.

2. A general register type of organization:

The instruction format in this type of computer needs three register address fields. Thus the instruction for an arithmetic addition may be written in an assembly language as

ADD R1 , R2 , R3

to denote the operation R 1 ← R2 + R 3 . The number o f address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus the instruction

ADD R1 , R2

would denote the operation R1 ← R1 + R2. Only register addresses for R1 and R2 need be specified in this instruction.

General register-type computers employ two or three address fields in their instruction format. Each address field may specify a **processor register** or **a memory word**. An instruction symbolized by

ADD R1 , X

would specify the operation R1 ← R1 + M[X]. It has two address fields, one for register R1 and the other for the memory address X.

3. A stack organization:

Computers with stack organization would have PUSH and POP instructions which require an address field. Thus the instruction

PUSH X

will push the word at address X to the top of the stack. The stack pointer is updated automatically. **Operation-type instructions do not need an address field in stack-organized computers**. This is because the operation is performed on the two items that are on top of the stack. The instruction

ADD

in a stack computer consists of an operation code only with no address field. This operation has the effect of popping the **two top numbers from the stack**, **adding the numbers**, and **pushing the sum into the stack**. There is no need to specify operands with an address field since all operands are implied to be in the stack.

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

$$X = (A + B) \cdot (C + D)$$

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

```
ADD   R1, A, B    R1←M[A] + M[B]
ADD   R2, C, D    R2←M[C] + M[D]
MUL   X, R1, R2   M[X]←R1*R2
```

Two-Address Instructions:

```
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:

```
LOAD     A    AC←M[A]
ADD      B    AC←AC + 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
```

Zero-Address Instructions:

```
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
```

RISC Instructions:

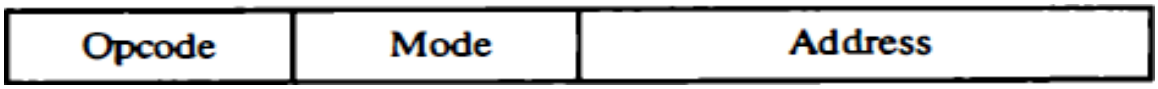
```
LOAD    R1, A          R1←M[A]
LOAD    R2, B          R2←M[B]
LOAD    R3, C          R3←M[C]
LOAD    R4, D          R4←M[D]
ADD     R1, R1, R2     R1←R1 + R2
ADD     R3, R3, R2     R3←R3 + R4
MUL     R1, R1, R3     R1←R1*R3
STORE   X, R1          M[X]←R1
```

Addressing Modes:

The **operation field** of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer registers or memory words. The way the operands are chosen during program execution is dependent on the addressing mode of the instruction. The **addressing mode** specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

In simple terms, **Addressing mode** is the way in which the location of an operand can be specified in an instruction. It generates an effective address (the actual address of the operand).

Instruction format with mode field



Types of Addressing Modes:

1. Implied Mode
2. Immediate Mode
3. Register Mode
4. Register Indirect Mode:
5. Autoincrement or Autodecrement Mode
6. Direct Address Mode
7. Indirect Address Mode
8. Relative Address Mode
9. Indexed Addressing Mode
10. Base Register Addressing Mode

There are two modes that need **no address field at all**. These are the implied and immediate modes.

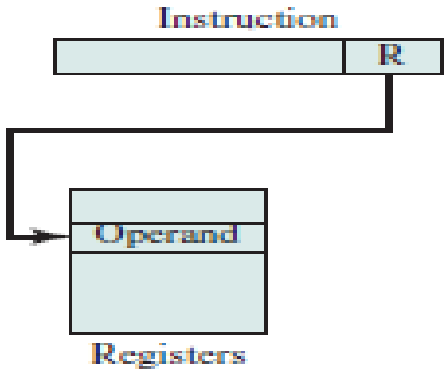
1. **Implied Mode:** In this mode the operands are specified implicitly in the definition of the instruction.
   For example, the instruction "complement accumulator (CMA)" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, 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. In other words, 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.

EX: LDAC #34H
LDAC loads data from memory to accumulator.
Therefore, AC=00110100.

When the address field specifies a processor register, the instruction is said to be in the register mode.

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.



E.g : LDAC R

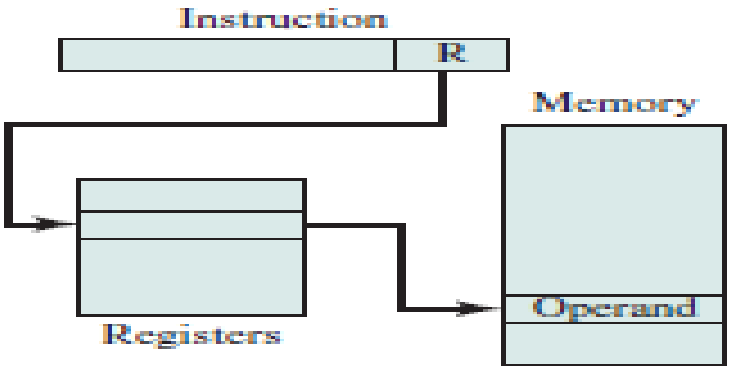
instruction reads data from register R

LDAC R

R : 5 → store value in Ac

∴ AC = 5

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.



EX: **LDAC (R1)**

If R1cotains the address of an operand in the memory, for example: address of an operand is 2000 which contains a value 350. Result: 350 is stored in the AC.

5. **Autoincrement Mode:** This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory. The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of the register are automatically incremented to the next value.

E.g : LDAC (R)

R : 5

instruction reads address from register R.

instruction reads data from memory location

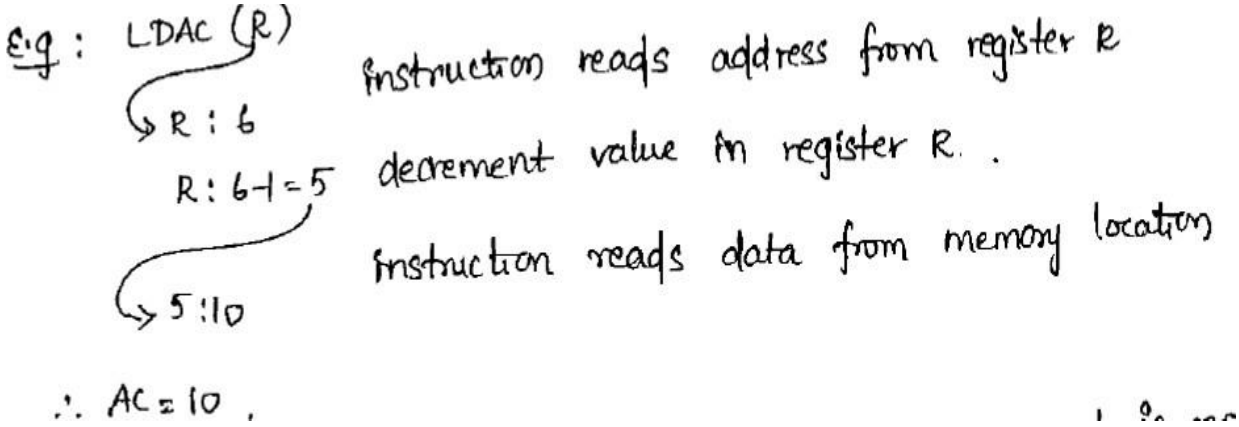
5 : 10 → store in AC

AC = 10

R : 5+1 = 6 .

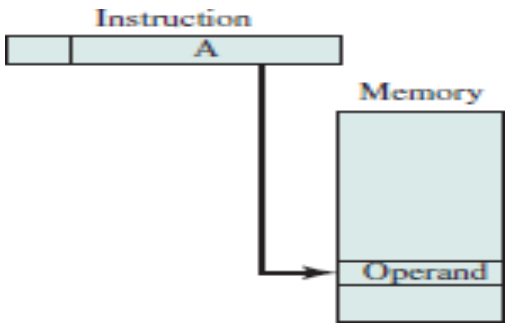
6. Autodecrement Mode

The effective address of the operand is the contents of a register specified in the instruction. Before accessing the operand, the contents of this register are automatically decremented and then the value is accessed.



Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated.

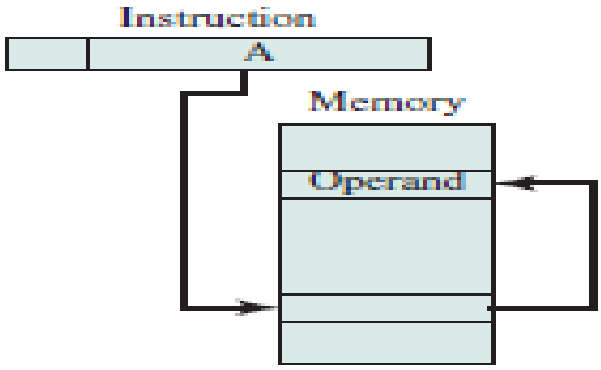
7. Direct Address Mode: In this mode the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of the instruction. In a branch-type instruction the address field specifies the actual branch address.



Ex**: LDAC 5000**

This instruction reads the operand from the Memory location 5000. if the memory location 5000 contains a value 250, then it will be stored in AC.

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.



EX: ADD (A), R1

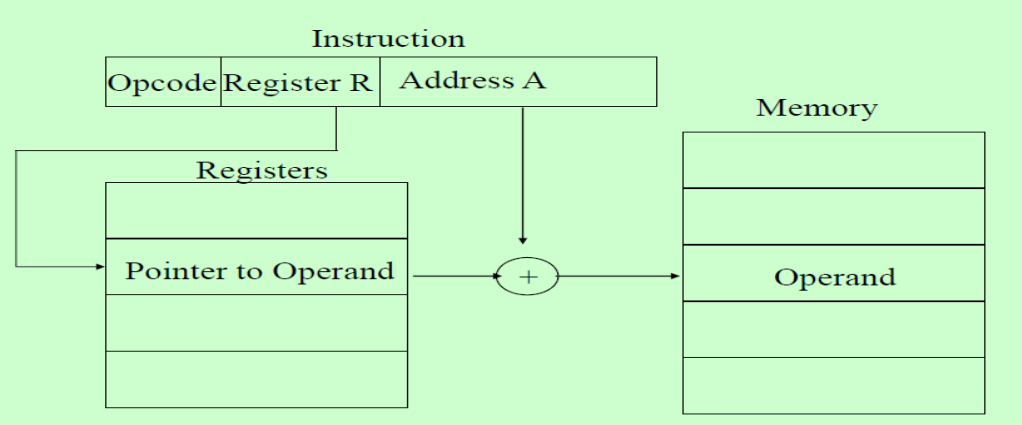
➔ If A is address of EA. For example: address of A is 1000 which contains 3000, 3000 is an address of an operand (EA).

➔ This instruction reads an operand from the location address 3000 and adds its contents to R1.

A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU. The effective address in these modes is obtained from the following computation:

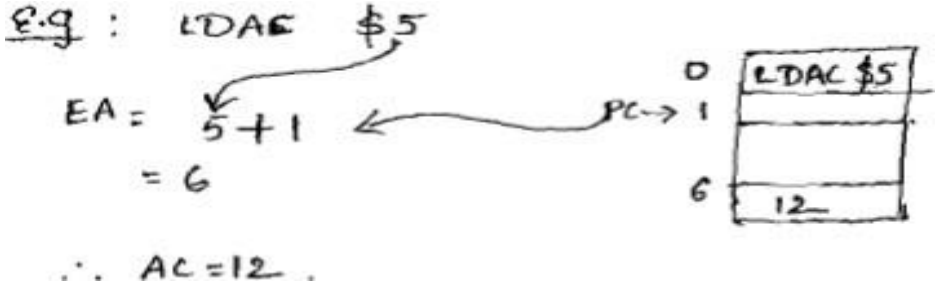
➔ **effective address = address part of instruction + content of CPU register**

The CPU register used in the computation may be the program counter, an index register, or a base register. In either case we have a different addressing mode which is used for a different application.



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.



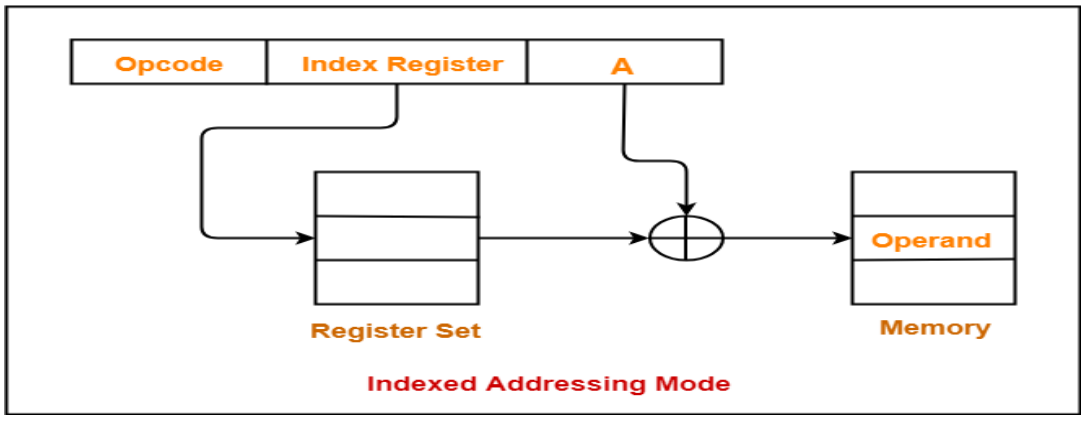
EX:

PC= address of next instruction , i.e., 1. The address given in the instruction is 5

Then EA= 5 + 1= 6 which contains a value 12. Finally AC contains 12.

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.



Ex: LDAC A(XR)

Assume XR=100, A=500

This instruction reads the operand from the effective address (600)

i.e., EA= XR contents (index register) + 500

= 100 + 500=600

If memory location at 600 contains a value 55 (assume), This 55 will be stored in AC.

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.

Ex: LDAC A(R)

Assume R=1000, A=50

This instruction reads the operand from the effective address (1050)

i.e., EA= R contents (Base register) + 50

= 1000 + 50=1050

If memory location at 1050 contains a value 255 (assume), this 255 will be stored in AC.

Numerical Example:

PC = 200
R1 = 400
XR = 100
AC

Address	Memory	
200	Load to *AC*	Mode
201	Address = 500	
202	Next instruction	
399	450	
400	700	
500	800	
600	900	
702	325	
800	300	

Addressing Mode	Effective Address	Content of AC
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

Table (4): Tabular list of some addressing modes of numerical example.

Data Transfer and Manipulation:

Computers provide an extensive set of instructions to give the user the flexibility to carry out various computational tasks. The instruction set of different computers differ from each other mostly in the way the operands are determined from the address and mode fields.

Most computer instructions can be classified into three categories:

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

Data transfer instructions cause transfer of data from one location to another without changing the binary information content.
Data manipulation instructions are those that perform arithmetic, logic, and shift operations.
Program control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer.

The instruction set of a particular computer determines the register transfer operations and control decisions that are available to the user.

1. Data transfer instructions

Data transfer instructions **move data from one place in the computer to another without changing the data content.** The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves. Table (5) gives a list of eight data transfer instructions used in many computers.

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

Table (5): Data Transfer Instructions

- The **load** instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator.
- The **store** instruction designates a transfer from a processor register into memory.
- The **move** instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another. It has also been used for data transfers between CPU registers and memory or between two memory words.
- The **exchange** instruction swaps information between two registers or a register and a memory word.
- The **input and output** instructions transfer data among processor registers and input or output terminals.
- The **push and pop** instructions transfer data between processor registers and a memory stack.

2. Data Manipulation Instructions

Data manipulation instructions perform operations on data and provide the computational capabilities for the computer. The data manipulation instructions in a typical computer are usually divided into three basic types:

i. Arithmetic instructions
ii. Logical and bit manipulation instructions
iii. Shift instructions

i. Arithmetic instructions

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

Table (6): Arithmetic Instructions

- A special carry flip-flop is used to store the carry from an operation. The instruction "**add with carry**" performs the addition on two operands plus the value of the carry from the previous computation.
- Similarly, the "**subtract with borrow**" instruction subtracts two words and a borrow which may have resulted from a previous subtract operation.
- The **negate instruction** forms the 2' s complement of a number.

ii. Logical and Bit Manipulation Instructions

Logical instructions perform binary operations on strings of bits stored in registers. They are useful for manipulating individual bits or a group of bits that represent binary-coded information. The logical instructions consider each bit of the operand separately and treat it as a Boolean variable.

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

Table (7): Logic and Bit Manipulation Instructions

iii. Shift Instructions

Instructions to shift the content of an operand are quite useful and are often provided in several variations. Shifts are operations in which the bits of a word are moved to the left or right. The bit shifted in at the end of the word determines the type of shift used. Shift instructions may specify logical shifts, arithmetic shifts, or rotate-type operations. In either case the shift may be to the right or to the left.

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 through carry	RORC
Rotate left through carry	ROLC

Table (8): Shift Instructions

The **rotate through carry** instruction treats a carry bit as an extension of the register whose word is being rotated. Thus a **rotate-left through carry** instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry and at the same time, and shifts the entire register to the left.

A possible instruction code format of a shift instruction may include five fields as follows:

OP REG TYPE RL COUNT

OP- operation code field
REG- a register address that specifies the location of the operand
TYPE- a 2-bit field specifying the four different types of shifts
RL- a 1-bit field specifying a shift right or left
COUNT- a k-bit field specifying up to 2^k - 1 shifts

Program Control:

After the execution of a data transfer or data manipulation instruction, control returns to the fetch cycle with the program counter containing the address of the instruction next in sequence.

On the other hand, a program control type of instruction, when executed, may change the address value in the program counter and cause the flow of control to be altered. In other words, program control instructions specify conditions for altering the content of the program counter, while data transfer and manipulation instructions specify conditions for data-processing operations.

The change in value of the program counter as a result of the execution of a program control instruction causes a break in the sequence of instruction execution. This is an important feature in digital computers, as it provides control over the flow of program execution and a capability for branching to different program segments.

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

Table (9) : Program Control Instructions

Status Bit Conditions:

It is sometimes convenient to supplement the ALU circuit in the CPU with a status register where status bit conditions can be stored for further analysis. Status bits are also called condition-code bits or flag bits. Figure (6) shows the block diagram of an 8-bit ALU with a 4-bit status register. 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.

1. Bit **C (carry)** is set to 1 if the end carry C_8 is 1. It is cleared to 0 if the carry is 0.
2. Bit **S (sign)** is set to 1 if the highest-order bit F_7 is 1. It is set to 0 if the bit is 0.
3. Bit **Z (zero)** is set to 1 if the output of the ALU contains all 0's. It is cleared to 0 otherwise. In other words, Z = 1 if the output is zero and Z = 0 if the output is not zero.

4. Bit **V (overflow**) is set to 1 if the exclusive-OR of the last two carries is equal to 1, and cleared to 0 otherwise. This is the condition for an overflow when negative numbers are in 2's complement. For the 8-bit ALU, V = 1 if the output is greater than + 127 or less than – 128.

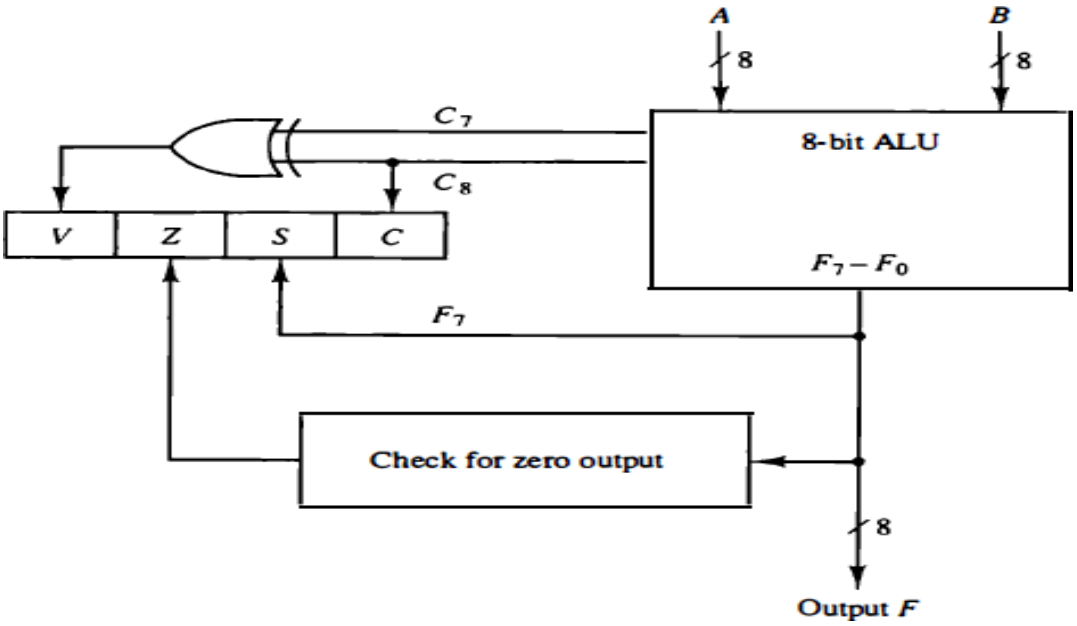


Figure (6): Status register bits

Conditional Branch Instructions:

To change the flow of execution in the program we use some kind of branching instructions which are depending on the some conditions result.

Each mnemonic is constructed with the letter **B** (for branch) and an abbreviation of the condition name. When the opposite condition state is used, the letter **N** (for no) is inserted to define the 0 state. Thus **BC** is Branch on Carry, and **BNC** is Branch on No Carry.

If the stated condition is true, program control is transferred to the address specified by the instruction. If not, control continues with the instruction that follows. The conditional instructions can be associated also with the jump, skip, call, or return type of program control 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	$A \neq B$

Table (10): Conditional Branch Instructions

Example: Consider an 8-bit ALU as shown in Figure (6). The largest unsigned number that can be accommodated in 8 bits is 255. The range of signed numbers is between + 127 and - 128. Let A = 11110000 and B = 00010100. To perform A - B, the ALU takes the 2's complement of B and adds it to A.

$$\begin{array}{rl} A: & 11110000 \\ \bar{B} + 1: & +11101100 \\ \hline A - B: & 11011100 \end{array} \qquad C = 1 \quad S = 1 \quad V = 0 \quad Z = 0$$

The compare instruction updates the status bits as shown. C = 1 because there is a carry out of the last stage. S = 1 because the leftmost bit is 1. V = 0 because the last two carries are both equal to 1, and Z = 0 because the result is not equal to 0.

Subroutine Call and Return:

A subroutine is a self-contained sequence of instructions that performs a given computational task. During the execution of a program, a subroutine may be called to perform its function many times at various points in the main program. Each time a subroutine is called, a branch is executed to the beginning of the subroutine to start executing its set of instructions. After the subroutine has been executed, a branch is made back to the main program.

The instruction that transfers program control to a subroutine is known by different names. The most common names used are **call subroutine**, **jump to subroutine**, **branch to subroutine**, or **branch and save address**.

A call subroutine instruction consists of an operation code together with an address that specifies the beginning of the subroutine. The instruction is executed by performing two operations:

(1) The address of the next instruction available in the program counter (the return address) is stored in a temporary location so the subroutine knows ***where to return***, and

(2) Control is transferred to the beginning of the subroutine. The last instruction of every subroutine, commonly called return from subroutine, transfers the ***return address*** from the temporary location into the program counter. This results in a transfer of program control to the instruction whose address was originally stored in the temporary location.

The most efficient way is to store the return address in a memory stack. The advantage of using a stack for the return address is that when a succession of subroutines is called, the sequential return addresses can be pushed into the stack. The return from subroutine instruction causes the stack to pop and the contents of the top of the stack are transferred to the program counter. In this way, the return is always to the program that last called a subroutine. A subroutine call is implemented with the following microoperations:

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Push content of PC onto the stack
$PC \leftarrow$ effective address	Transfer control to the subroutine

If another subroutine is called by the current subroutine, the new return address is pushed into the stack, and so on. The instruction that returns from the last subroutine is implemented by the microoperations:

$PC \leftarrow M[SP]$	Pop stack and transfer to PC
$SP \leftarrow SP + 1$	Increment stack pointer

By using a subroutine stack, all return addresses are automatically stored by the hardware in one unit. The programmer does not have to be concerned or remember where the return address was stored.

→ A ***recursive subroutine*** is a subroutine that calls itself.

Program interrupt:

Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request. Control returns to the original program after the service program is executed.

The interrupt procedure is, in principle, quite similar to a subroutine call except for three variations:

(1) The interrupt is usually initiated by an internal or external signal rather than from the execution of an instruction (except for software interrupt as explained later);

(2) The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction; and

(3) An interrupt procedure usually stores all the information necessary to define the state of the CPU rather than storing only the program counter.

The state of the CPU at the end of the execute cycle (when the interrupt is recognized) is determined from:
1. The content of the program counter
2. The content of all processor registers
3. The content of certain status conditions

The collection of all status bit conditions in the CPU is sometimes called a **program status word or PSW**. The PSW is stored in a separate hardware register and contains the status information that characterizes the state of the CPU

The last instruction in the service program is a return from interrupt instruction. When this instruction is executed, the stack is popped to retrieve the old PSW and the return address. The PSW is transferred to the status register and the return address to the program counter. Thus the CPU state is restored and the original program can continue executing.

There are three major types of interrupts that cause a break in the normal execution of a program. They can be classified as:
1. External interrupts
2. Internal interrupts
3. Software interrupts

External interrupts come from input-output (l/0) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source. Examples that cause external interrupts are l/0 device requesting transfer of data, l/0 device finished transfer of data etc.

Internal interrupts arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called traps . Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation.

→External and internal interrupts are initiated from signals that occur in the hardware of the CPU.

A software interrupt is initiated by executing an instruction. Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program. The most

common use of software interrupt is associated with a supervisor call instruction. This instruction provides means for switching from a CPU user mode to the supervisor mode.

Reduced Instruction Set Computer (RISC):

- An important aspect of computer architecture is the design of the instruction set for the processor.
- Early computers had small and simple instruction sets, forced mainly by the need to minimize the hardware used to implement them .
- As digital hardware became cheaper with the advent of integrated circuits, computer instructions tended to increase both in number and complexity. Many computers have instruction sets that include more than 100 and sometimes even more than 200 instructions. These computers also employ a variety of data types and a large number of addressing modes.

A computer with a large number of instructions is classified as a ***complex instruction set computer***, abbreviated **CISC**.

In the early 1980s, a number of computer designers recommended that computers use fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often. This type of computer is classified as a ***reduced instruction set computer*** or **RISC**.

CISC Characteristics

The major characteristics of CISC architecture are:

1. A large number of instructions-typically from 100 to 250 instructions
2. Some instructions that perform specialized tasks and are used infrequently
3. A large variety of addressing modes-typically from 5 to 20 different modes
4. Variable-length instruction formats
5. Instructions that manipulate operands in memory

RISC Characteristics

The concept of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer The major characteristics of RISC architecture are:

1. Relatively few instructions
2. Relatively few addressing modes
3. Memory access limited to load and store instructions
4. All operations done within the registers of the CPU
5. Fixed-length, easily decoded instruction format
6. Single-cycle instruction execution
7. Hardwired rather than microprogrammed control

Other characteristics attributed to RISC architecture are:

1. A relatively large number of registers in the processor unit
2. Use of overlapped register windows to speed-up procedure call and return
3. Efficient instruction pipeline
4. Compiler support for efficient translation of high-level language programs into machine language .

Overlapped Register Windows

Procedure call and return occurs quite often in high-level programming languages. When translated into machine language, a procedure call produces a sequence of instructions that save register values, pass parameters needed for the procedure, and then calls a subroutine to execute the body of the procedure. After a procedure return, the program restores the old register values, passes results to the calling program, and returns from the subroutine. Saving and restoring registers and passing of parameters and results involve time consuming operations.

A characteristic of some RISC processors is their use of overlapped register windows to provide the passing of parameters and avoid the need for saving and restoring register values.

Berkeley RISC I

→ One of the first projects intended to show the advantages of RISC architecture was conducted at the University of California, Berkeley.

→ The Berkeley RISC I is a 32-bit integrated circuit CPU. It supports 32-bit addresses and either 8-, 16-, or 32-bit data. It has a 32-bit instruction format and a total of 31 instructions.

→ There are three basic addressing modes: register addressing, immediate operand, and relative to PC addressing for branch instructions. It has a register file of 138 registers arranged into 10 global registers and 8 windows of 32 registers in each.