UNIT 1 INSTRUCTION SETS

Structure

1.0 INTRODUCTION

In the previous block, we have discussed about structure of the computer and the data representation. One term which we have commonly used is the instruction. In this respect, few questions which still need to be answered is : What is an instruction? What are its components? How is the instruction executed by the CPU? This unit is an attempt to answer the first two questions. While, the third question is a complex one and is explained in the later units. In this unit we have discussed in details about the instructions, their types, the operands. We will also discuss about the various addressing schemes which are popular among various PC's and how the effective address is calculated for these schemes. In addition we are also trying to highlight the basic design issues related to the instruction in the unit. We have presented here the instruction set of IBM System/370 as an example. However, you can study the details on the other instruction sets for example VAX machine from the further readings. We have not included discussions on the instruction set of popular INTEL microprocessor. One of this microprocessor instruction set is discussed in the Block 3 of this course. Other related microprocessors instruction set can also be studied from the further readings.

1.1 OBJECTIVES

At the end of this unit a student should be able to:

- discuss about various elements of an instruction
- distinguish various types of instructions
- differentiate various types of operands
- define a classification of computers on the basis of number of addresses in instruction sets
- discuss about various operations which are performed by the instructions
- identify various addressing schemes
- calculate effective address for various schemes
- discuss about the instruction formats design characteristics

1.2 INSTRUCTION SET CHARACTERISTICS

Till now we have discussed about instruction in an abstract way. Now let us discuss in details various characteristics of instructions. But let us first discuss what is the significance of instruction set? One thing which should be kept in mind is that the instruction set is an boundary which is looked upon in a same fashion by a computer designer and the programmer. From the computer designer's point of view, the instruction set provides the functional requirements of the CPU. In fact, for implementing the CPU design, one of the main task is to implement the instruction set for that CPU. However, from the user point of view machine instructions or Assembly instructions are needed for low level programming. In addition, a user should also be aware of registers, the data types supported by the machine and the functioning of the ALU. We will be discussing about the registers in unit 2 and about the ALU in unit 3 of this block.

Explanation on the machine instruction set gives extensive details about the CPU of a machine. In fact, the operations which a CPU can perform can be determined by the machine instructions. Now, let us answer some introductory questions about the instruction set.

What is an instruction set?

An **instruction set** is a collection of all the *instructions* a CPU can execute. Therefore, if we define all the instructions of a computer, we can say we have defined the instruction set. Each instruction consist of several elements. An instruction element is a unit of information required by the CPU for execution.

What are the elements of an instruction?

An instruction have the following elements:

- an operation code also termed as **opcode** which specifies the operation to be performed.
- A reference to the operands on which data processing is to be performed. For example, an address of an operand.
- A reference to the operands which may store the results of data processing operation performed by the instruction.
- A reference for the next instruction, to be fetched and executed.

The next instruction which is to be executed is normally the next instruction following the current instruction in the memory. Therefore, no explicit reference to the next instruction is provided. If, we donot want this normal flow of execution then? You will find the answer to this question in this unit.

An important aspect, of the references to operands and results is: where are those operands located ? In the memory or in the CPU registers or in the I/O device. If the operands are located in the registers then an instruction can be executed faster than that of operands located in the memory. The main reason here is that memory access time is higher in comparison to the register access time.

How is an instruction represented?

Instructions are represented as sequence of bits. An instruction is divided into number of fields. Each of these fields corresponds to a constituent element of instruction. A layout of instruction is termed as **instruction format**. For example, following is the instruction format for IAS computer. It uses four bits for opcode and only two operand references are provided here. No explicit reference is provided for the next instruction to be executed.

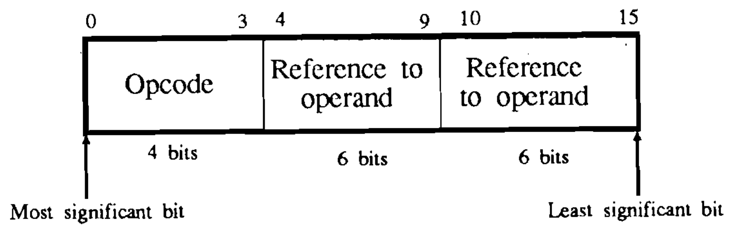


Figure 1 : A sample instruction format

In most instruction sets, many instruction formats are used. An instruction first is read into an instruction register(IR), then the CPU decodes the instruction and extracts the required operands on the basis of references made through the instruction fields and processes it. Since the binary representation of the instruction is difficult to comprehend and is seldom used for representations. We will be using symbolic representation to these instruction in this unit along with the comments wherever desired.

What are the types of Instructions?

The instructions can be categorised under the following categories:

- **Data Processing Instructions:** These instructions are used for arithmetic and logic operations in a machine. Examples of Data Processing Instructions are: Arithmetic, boolean, shift, character and string processing instructions, stack and register, manipulation instructions, vector instructions etc.
- **Data Storage/Retrieval Instructions:** Since the data processing operations are normally performed on the data stored in CPU registers. Thus, we need an instruction to bring data to and from memory to registers. These are called Data storage/retrieval instructions. Examples of data storage and retrieval instructions are: Load and store instructions.
- **Data Movement Instructions:** These are basically input/ output instructions. These are required to bring in programs and data from various devices to memory or to communicate the results to the Input/Output devices. Some of these instructions can be: Start Input/Output, Halt Input/Output, TEST Input/Output etc.
- **Control Instructions:** These instructions are used for testing the status of computation through Processor Status Word (PSW). This register will be discussed in greater details in unit 2. Another of such instruction is the branch instruction used for transfer of control. We will discuss in more details about these instructions.
- **Miscellaneous Instructions:** These instructions does not fit in any of the above category. Some of these instruction are: interrupt or supervisory call, swapping, return from interrupt, Halt instruction or some privileged instruction of operating systems.

What are the factors which play important role for selection/designing of instruction set for a machine?

Instruction set design is the most complex yet interesting and very much analysed aspect of computer design. The instruction set plays important role in design of the CPU as it defines many functions of it. Since instruction sets are the means by which a programmer can control the CPU, therefore, users view must be considered while designing the instruction set. Some of the important design issues relating to instruction design are:

- How many and what operations to be provided?
- What are the operand data types to be provided?
- What should be the Instruction format? This includes issues like: instruction length, number of address, length of various elements of instructions etc.
- What is the number of registers which can be referenced by an instruction and how are they used?
- What are the modes of specifying an operand address?

We will try to analyse these issues in some details in this and subsequent sections. However, we have kept the level of discussion very general. A specific example of instruction set is given at the end of this unit.

1.2.1 Operand Data Types

An operand data type specifies the type of data on which a particular operation can be performed. For example, for an arithmetic operation numbers are to be used as data types. In general the operands which can be used in an instruction can be categorised in four general categories. These are:

- Addresses
- Numbers

- Characters
- Logical data

Addresses: Addresses are treated as a form of data which is used in calculation of actual physical memory address of an operand. In most of the cases, the addresses provided in instruction are operand references and not the actual physical memory addresses.

Numbers: All machines provide numeric data types. One special feature of numbers used in computers is that they are limited in magnitude, and hence the underflow and overflow may occur during arithmetic operations on these numbers. The maximum and minimum magnitude is fixed for an integer number while a limit of precision of number and exponent exist in the floating point numbers. The three numeric data types which are common in computers are:

— Fixed point numbers or Integers (signed or unsigned)

— Floating point numbers

— Decimal numbers

All the machines provide instructions for performing arithmetic operations on fixed point and floating point numbers. Many machines provide arithmetic instructions which perform operations on packed decimal digits.

Characters: Another very common data type is the character or string of characters. The most widely used character representation is ASCII (American National Standard Code of Information Interchange). It has 7 bits for coding data pattern which implies 128 different characters. Some of these characters are control characters which may be used in data communication. The eighth bit of ASCII may be used as a parity bit. One special mention about ASCII which facilitate conversion of a 7 bit ASCII and a 4 bit packed decimal number, is that the last four digits of ASCII number are binary equivalent of digits 0-9.

That is

Decimal	Binary	ASCII
0	0000	011 *0000*
1	0001	011 *0001*
2	0010	011 *0010*
3	0011	011 *0011*
:	:	:
:	:	:
9	1001	011 *1001*

Figure 2 : Decimal digits in ASCII

The other important code is Extended Binary Coded Decimal Interchange Code (EBCDIC). This is a 8 bit code and is compatible with packed decimal in a similar way as that of ASCII. The digits 0 through 9 in this can be represented as 1111 0000 through 1111 1001.

Logical Data: In general a data word or any other addressable unit such as byte, half word etc. are treated as a single unit of data. But can we consider a n-bit data unit consisting of n items of 1 bit each? If we treat each bit of a n-bit data as an item then it can be considered to be logical data. Each of these n items can have a value 0 or 1.

What are the advantages of such a bit oriented view of data? The advantages of such a view will be:

- We can store an array of boolean or binary data items most efficiently.
- We will be in a position to manipulate the bits of any data item.

But where do we need to manipulate bits of a data item? The example of such a case is shifting of significand bits in a floating point operation or for converting ASCII to packed decimals where we need only the 4 right most bits of ASCII's byte.

Please note that for extracting decimal from ASCII, first the data is treated as logical data and then can be used in arithmetic operations as numeric data. Thus, the operation performed on a unit of data determines the type of the unit of data at that instance. This statement may not be true for a high level language, but holds good for machine level language.

Check Your Progress 1

State true or false.

Q1. An instruction set is meant only for the programmer and is not needed at the time of implementation of a machine?

True ☐ False ☐

Q2. Explicit operand references are must for an instruction?

True ☐ False ☐

Q3. You can use only one instruction format for an instruction set of a machine.

True ☐ False ☐

Q4. Data movement instructions are used for bringing in data from the memory to CPU registers.

True ☐ False ☐

Q5. Numbers represented in computers are limited in magnitude.

True ☐ False ☐

Q6. A data value in a machine language can be treated as of one type only.

True ☐ False ☐

1.2.2 Number of Addresses in an Instruction

The fewer number of addresses in an instruction lead to reduced length of instructions, however, it also limits the range of function that can be performed by the instructions. In a sense this implies that a machine instruction set having less number of addresses have longer programs, which means longer execution time. However, more addresses may lead to more complex decoding and processing circuits.

Most of the instructions donot require more than three operand addresses. In instructions having fewer addresses than three, normally some of the operand locations are implicitly defined. Many computers have a range of instructions of different length and number of addresses. The following table gives an example of zero, one, two and three address instruction along with their interpretations.

Number of Addresses	Instruction	Interpretation
3	ADD A,B,C	Operation A = B+C is executed
2	ADD A,B	Two plausible interpretations (i) AC = A + B (ii) A = A + B. In this case the original content of operand location A is lost
1	ADD A	AC = AC+A A is added to accumulator
0	ADD	Top of Stack contains the addition of top two values of the stack.

- AC is a accumulator register.
  A,B,C are operand locations.

Figure 3 : Example of zero, one, two and three address instruction

The register architecture, that is a general classification which is based on register set of the computer, is sometimes classified according to the number of addresses in instructions. These classifications are:

Evaluation-Stack Architecture: These machines are Zero address machines and their operands are taken from top of the stack implicitly. The ALU of such machine directly references a stack which can be implemented in main memory or registers or both. These machine contains instructions like PUSH and POP to load a value on stack and store a value in the memory respectively. Please note that PUSH, POP are not zero address instructions but contain one address.

The main advantages of such an architecture are:

- very short instructions
- since stacks are normally made within CPU in such an architecture machine, the operands are close to the ALU, thus fast execution of instructions.
- excellent support for subroutines.

While, the main disadvantages of this architecture are:

- not very general in nature, in comparison to other architectures.
- difficult to program for applications like text and string processing.

One example of such a machine is Burroughs B6700. However, these machines are not very common today because of the general nature of machines desired. The stack machines uses Polish notations for evaluation of arithmetic expression. Polish notation was introduced by a Polish logician Jan Lukasiewicz. The main theme of this notation is that an arithmetic expression A×B can be expressed as:

either	×AB	(Prefix notation)
or	AB×	(Suffix or reverse polish notation)

In stacks we use suffix or reverse polish notation for evaluating expression. The rule here is to push all the variable values on the top of stack and do the operation with the top elements of stack as an operand is encountered. The priority of operand is kept in mind for generation of reverse polish notation. For example, an expression A×B+C×D/E will be represented in reverse polish notation as:

AB×CD×E/+

Thus, the execution program for evaluation of F = A×B+C×D/E will be:

```
PUSH  A    /Transfer the value of A on to the top of stack /
PUSH  B    /Transfer the value of B on to the top of stack /
MULT       /Multiply. It will remove value of A and B from the stack and multiply A × B /
PUSH  C    /Transfer C on the top of stack /
PUSH  D    /Transfer D on the top of stack /
MULT       /Remove values of C & D from the stack and multiply C × D /
PUSH  E    /Transfer E on the top of stack /
DIV        /C × D/E /
ADD        /Add the top two values on the stack /
POP   F    /Transfer the results back to memory location F /
```

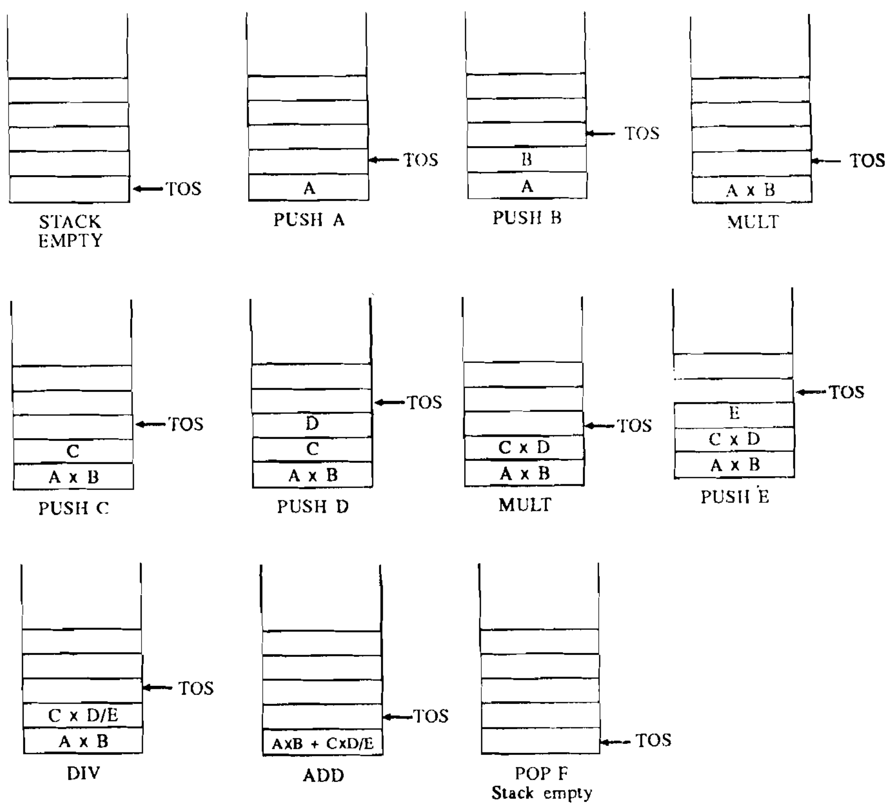
Figure 4 : Sample program for evaluating A×B+C×D/E using zero address instructions

Please note that PUSH and POP are not zero-address instructions.

The execution of the above program using stack is represented diagrammatically in figure 5.

Accumulator Machines: These machines contain a special register called accumulator which holds the results of arithmetic, logical or shift operations. The accumulator is an implicit address. The instructions in such machines are normally one-address instructions. The basic advantage of the one address machines is that the instructions are shorter than 2 and 3 addressed machines. However, the performance of these machines are somewhat slow because these machine require frequent memory accesses. These machines are made by various vendors.

For example, a program for evaluating the expression F=A×B+C×D/E written in this Accumulator based machine is given in figure 6.



Assumption : TOS is top of stack which is pointing to an empty location

Figure 5 : Evaluation using a stack

```
LOAD     A          /Transfer A to Accumulator/
MULT     B          /Multiply Accumulator with B, keep the result of /
                    /multiplication in B/
STORE    T          /Store the intermediate result temporarily in a location T/
LOAD     C          /Transfer C to Accumulator /
MULT     D          /Accumulator = Accumulator X D/
DIV      E          /Accumulator = Accumulator/E /
ADD      T          /Accumulator = Accumulator + T/
STORE    F          /Store the Accumulator value to location F/
```

Figure 6 : Sample program for evaluating A×B+C×D/E using one address instructions

General Purpose Register Set Machines: Many computers have been designed having set of registers termed as general purpose registers. These general-purpose register machines normally have multiple address instructions. In these machines any of the register of the register set can be used as accumulator or an address register or an index register or a stack pointer, or even (in some cases) as a program counter. In such machines, each instruction specifies itself, about the use of register. Some examples of such computers are IBM system 370 & Digital VAX families.

The instructions in such machines can specify as many *Register* operands as desired. Therefore, several operations can be performed solely on registers. This may increase the program execution speed as register references are faster than memory references. For the purpose of flexibility and ability to use multiple registers, these computers may use two and three address instructions. Some of these computers have instructions for Register to memory transfer, and memory to memory transfer. Since number of registers are few, therefore, these instructions are normally shorter.

The program for F = A×B + C×D/E for 2 address and 3 address machines will look like

```
MOVE    T, A    / Move T  A/
MULT    T, B    / Multiply T=TxB /
MOVE    F, C    / Move F  C /
MULT    F, D    / Multiply F=FxD /
DIV     F, E    / Divide F=F/E  /
```

```
MULT   F, A, B    / F=AxB /
MULT   T, C, D    / T=CxD /
DIV    T, T, E    / T=T/E /
ADD    F, F, T    / F=F+T /
ADD        F, T      / Add F=F+T /
```

Figure 7 : Sample program for evaluating F=A×B+C×D/E using two and three address instructions

Special-Purpose Register Set Machines

The main problem with the above type of machine is: too much of generality. In several machines, sets of registers may be used for only special purposes. For example, one set of register may be used as index registers, another set for holding arithmetic operands etc. computers like CDC6000/7000 family fall under this category.

Many a machines fall in between of the above two categories. They have some special purpose and some general purpose register sets. These machines are now a days quite popular.

1.2.3 Operation Types

Different computers use a wide variety of opcodes and number of addresses. Some of the operations which are specified on one machine may not exist on a second machine. However, certain category of operations exist on all the machines. We will try to provide details on some of the typical categories of operation. Broadly the operations specified in instructions irrespective of number of addresses in an instruction, can be categorised as:

— Data Transfer operations

— Arithmetic operations

— Logical and Shift operations

— Conversion operations

— Input/Output operations

— System control operations

— Transfer of control operations

Data Transfer Operations: This is the most fundamental type of operation. A data transfer instruction need to furnish the following information:

- The location of source and destination operands (This could be memory or register or stack-top). This will be more clear after you go through the next subsection.
- The length of data transfer
- The mode of addressing for each operand. These addressing modes are discussed in greater details in the next section.

Some of the common data transfer operations are:

Operation	**Description**
MOVE or TRANSFER	Transfers a word or a block of data from source to destination.
STORE	Transfers a word from the processor to a specified location in the main memory.
LOAD or FETCH	Brings a word from a location of main memory to the processor.
EXCHANGE	Exchanges the contents of the source with the destination.
CLEAR or RESET	Transfers a word containing all 0's to the destination.
SET	Transfers a word containing all 1's to the destination.
PUSH	Transfers a word from a source to top of stack.
POP	Transfer a word from the top of stack to a destination.

Figure 8 : Common data transfer operations

For all these instructions the choice of source or destination can be a location in the main memory or a register or the top of stack. This location can be indicated either in opcode or in the specification of an operand.

Arithmetic Operations: Almost all the machines provide the four basic arithmetic operations on signed fixed point integers. Some machines provide operations on floating point and packed decimal numbers. Some other arithmetic operations such as absolute of a number, negation of a number, incrementing or decrementing a number are also included as instructions in several machines. The execution of an arithmetic instruction requires bringing in the operands in the operational registers such that the data can be processed by the ALU. The arithmetic operations which can be provided in a machine in general are:

ADD, SUBTRACT, MULTIPLY, DIVIDE
ABSOLUTE, NEGATE, INCREMENT, DECREMENT

Logical and Shift Operations: The logical operations are based on boolean operations performed on binary data. Some of the logical operations are: AND, OR, NOT, Exclusive-OR.

In addition to these bitwise logical operations, machines provide a variety of shifting operations. A shift operation is performed either in the left or to the right. In a shift operation, all the bits move towards the left or right as desired. (Please refer to figure 9).

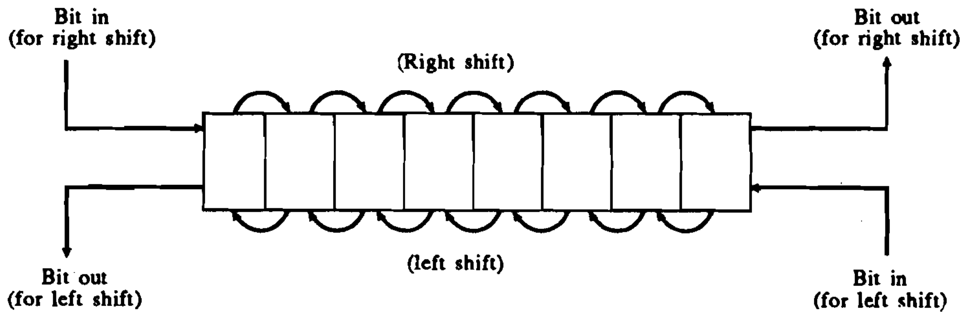


Figure 9 : Shift operations schematic for 8 bit register

The following are some observations about shift operations.

- In *logical left/right shift* the "bit in" is a 0 bit.
- *Arithmetic shift* is same as logical shift except for the sign bit which is not shifted.
- A *circular shift* uses the "bit out" bit as the "bit in" bit.

The main usage of these shifts are:

- The logical shift is used in extracting fields from a word. How?
- The arithmetic shift if performed on numbers represented in signed 2's complement notation cause multiplication by 2 or division by 2 depending on left or right arithmetic shift, provided there is no overflow or underflow.
- The circular shift preserves all the bits.

Conversion Operations: The conversion operations are needed to convert the format of the data. For example changing format from decimal to binary or ASCII to EBCDIC or vice versa. In general the two conversion operation are:

TRANSLATE	This instruction translates a given piece of data depending on a table of correspondence
CONVERT	It converts the contents of a word from one format to other.

Let us give an example of each of these. A common example for TRANSLATE instruction is conversion of ASCII to EBCDIC. This can be achieved by creating a table which is 256 byte long. We can call it as an array of EBCDIC equivalent values. Each index of this array represent the ASCII value, while the content of that location is the equivalent EBCDIC. For example, if 00110011 in ASCII is equivalent to 11001100 then

ARRAY location 00110011 contains 11001100
that is TRANS [00110011] = 11001100

However, the convert instruction converts the format based on certain rule for example, decimal to binary.

Input/Output Operations: There is a lot of variety as far as input/output operations are concerned as they depend on the type of input/output such as programmed I/O, DMA, etc. We have given some I/O operations in the Unit 4 of Block 1. However, some of the common input/output instructions are:

READ (or INPUT)	This command is used for transferring data from input/output module or part to a destination which may be the main memory or processor register.
WRITE (or OUTPUT)	This command transfers data from a specified source to input/output module.
TEST I/O	It transfers the input/output systems status information to a specified destination.

System Control Operations: The system control operations generally come under the category of privileged instructions, that is, these instructions are executed only when the processor is in certain privileged state or the processor is executing a program which is stored in a special privileged area of memory. In general, these instructions are used by the operating system. A typical system control instruction is OSCALL. This instruction causes the interruption of execution of current program and passes the control to the operating system.

Transfer of control Operations: In general, in a program execution the next instruction in sequence is executed next. However, in certain cases such as looping, decision making and subroutine call, the next instruction to be executed may not be the next instruction in sequence. The instructions that disturb the normal flow of instruction execution are called transfer of control instructions. The most common transfer of control instructions which are found in instruction sets are:

— Branch

— Skip

— Subroutine Call

Branch Instruction: A branch instruction causes a jump to the new instruction to be executed. A branch instruction is also known as jump instruction. This instruction has one operand, that is, the address of the instruction to which branch is desired. The branch instruction, in general, is used as a conditional branch instruction that is the branch is made only if a specified condition is satisfied otherwise the next instruction in the normal sequence is executed.

But how is the condition tested? Most of the machines provide a 1 bit or a multiple-bit conditional code, which in certain cases can be treated as a user visible register (we will discuss more about user visible registers in unit 2 of this block). For example, a typical machine performing on an arithmetic operation can set a 2 bit condition code to either zero, positive, negative or overflow condition. On such a machine, we can have the conditional jump instructions as:

Conditional Code (C.C)	Meaning	Instruction	Meaning
00	Resultant is positive	BRP X	Branch to memory location X if the result is positive
01	Resultant is negative	BRN X	Branch to memory location X if the resultant is negative
10	Resultant is Zero	BRZ X	Branch to X if resultant is Zero.
11	Overflow has occurred	BRO X	Branch to X if overflow has occurred.

All these branches may be implemented by assigning Program counter (PC) the address of the location to which branch is desired.

Figure 10 : Example of conditional branch instructions

As the changes in condition code may take place on execution of each instruction, Therefore, the branch instruction will depend on the most recent instruction which have modified the condition code. All the above four branches takes place when a condition, is fulfilled, otherwise the next instruction in sequence is executed. Another type of conditional branch instruction which has the condition in itself can be devised for a three address instruction. For example:

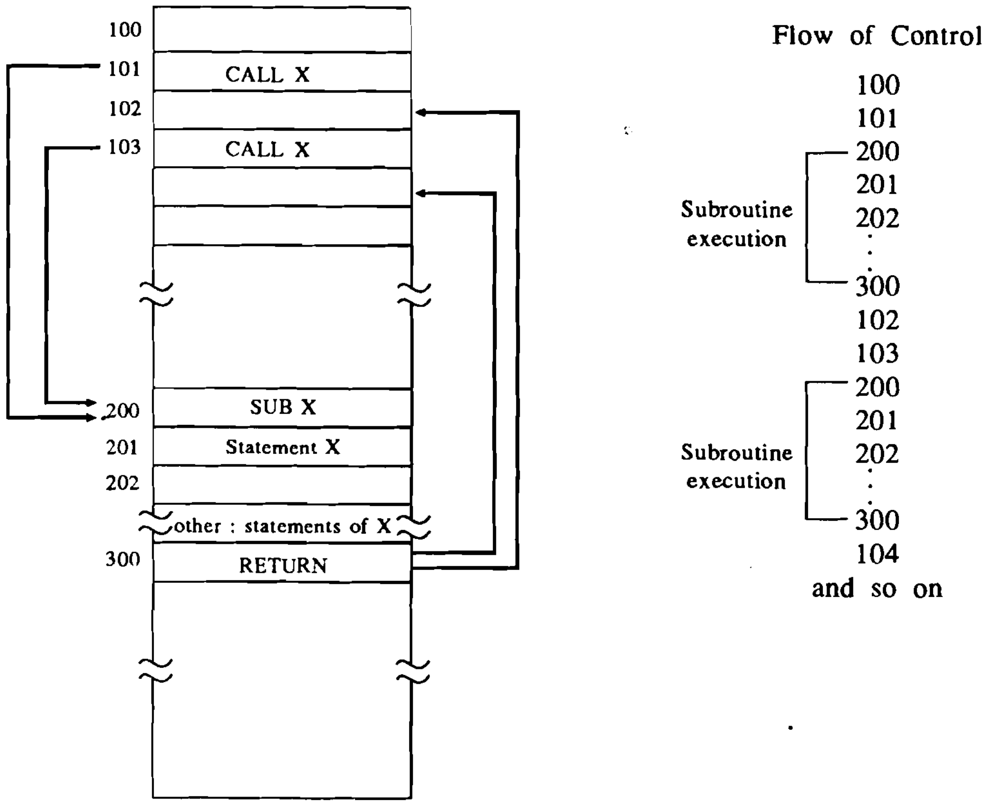
BUN R1, R2, C (Branch to a memory location address 'C' if contents of R1 is not equal to contents of R2.)

Here the condition is tested and branch is determined in a single instruction. Please note that branch can take place to a higher address or to a lower address. How branching can be used for looping is discussed in block 3 of this course.

Skip Instruction: This instruction skips the next instruction to be executed in sequence. In other words this instruction increments the address of "next instruction to be executed" (in many computers it is the program counter) by one instruction length. Skip instruction can also be used with conditions, for example, ISZ instruction skips the next instruction only if the condition code indicates that the resultant of the most recent operation is zero. This instruction along with branch instruction is used for implementing looping structures.

Subroutine Call: A subroutine is a self contained user program which contains the code often used repeatedly in a large program. This program is incorporated in a bigger program. For more details on subroutine, you can refer to Course 2, Block-1, Unit- 1.

A subroutine is called explicitly by a program statement. This is explained in the following figure:



Assumption for the present flow of control: No transfer of control instruction in the subroutine.

Figure 11 : Subroutine Call

Two most important instructions related to subroutine call are CALL and RETURN statements. CALL causes jump to the first instruction of subroutine, however, this jump must remember from where it has started as RETURN brings the control back to the instruction following the CALL instruction. The logic of subroutine call is similar to that of interrupt processing. The subroutine call is implemented in many cases by storing the contents of Program Counter (PC) which in fact is the pointer to the return address.

But where do we store the return address? In general, the return address can be stored in a register or memory location specifically used for this purpose. In such a system the following steps will be followed on encountering a subroutine call.

- Store the content of PC in a predetermined memory location or register. Let us, call this register or memory location as R
- Transfer the starting address of the subroutine which has been called in the PC
- Execute the subroutine
- On encountering RETURN statement of this subroutine load the contents of R into the PC.

A second approach can be to store the return address at the start of the subroutine. For example, a call instruction CALL X will initiate the following steps:

- Store the contents of PC into memory location X.
- Place address X+1 in the PC, as subroutine statements starts from this memory location only, as the Xth location is reserved for storing the return address
- Execute the subroutine
- On encountering RETURN statement, load the contents stored in memory location X to the PC.

However, these two approaches are not valid when more than one person want to execute the same subroutine simultaneously, and also when a subroutine calls itself. The reason is that in both the cases the location where the return address is stored will be rewritten by the new calling address, in turn, canceling the previously stored address.

Thus, a very general approach utilising stacks is used for subroutine call. The return address on a subroutine call is pushed on the top of the stack. On encountering a RETURN statement the POP operation is used to retrieve the most recent return address. This method also works for recursive subroutine calls.

Another important aspect of subroutine call is the parameter passing. With a subroutine call, in general, parameters are passed. There are three approaches for parameter passing

- Parameter passing through registers
- Parameter passing through memory location just after the call instruction location. The return address in such a scheme should be the memory location after these memory based parameters.
- Use of stacks for parameter passing in addition to return address.

The drawback of the first approach is that proper utilisation of registers is to be ensured by the calling program. The second method fails in cases where variable number of parameters are to be passed. The third approach is quite general in nature. Stacks are used for not only storing the return address but are also used for storing the parameters which are to be passed to the subroutine which is called. The stacks are discussed in greater details in Course-4.

Check Your Progress 2

1. Match the following pairs:

(a) Zero address instruction	(i) General purpose register set
(b) One address instruction	(ii) Stacks
(c) Two address instruction	(iii) Accumulator machine

2. What are the advantages and disadvantages of evaluation-stack architecture?

....

3. Match the following:

(i) MOVE	(a) Data transfer operation
(ii) WRITE	(b) I/O operation
(iii) LOAD	(c) Conversion operation
(iv) READ	(d) System control operation
(v) TRANSLATE	

4. How a subroutine call is different from branching?

....

....

5. What are the three methods for calling a subroutine?

....

1.3 ADDRESSING SCHEMES

As discussed earlier, the main function of a computer is to execute instructions. An instruction contains an operation code (opcode) and operand(s) in the coded form. The opcode field specify the operation to be carried out and the operand(s) field specify the location of the operand(s) in the memory. An operand may be specified as the part of the instruction, or the reference of the memory location where the value is stored may be given. The term **'addressing schemes'** refers to the mechanism employed for specifying operands. The arrangement of opcodes and operands and their numbers within the instructions determines the form or the format of an instruction. We will discuss more about addressing formats in the next section.

The choice of addressing schemes and instruction formats are governed by the efficiency, (time as well as space) economy and programming flexibility considerations. All the available machines in the market today, therefore, employ more than one addressing schemes. In the subsequent sections, we describe the most common addressing schemes and show how the contents of operand (address) are mapped to the physical memory location where the operand being addressed is actually stored.

In the description that follows the symbols A, A_1, A_2,...., etc. denote the content of an operand field. Thus A_i may refer to a data or a memory address. In case the operand field is a register address, then the symbols R, R_1, R_2,... etc., are used. If C denotes the contents (either of an operand field or a register or of a memory location), then (C) denotes the content of the memory location whose address is C.

The symbol EA (Effective Address) refers to a physical address in a non-virtual memory environment and refers to a register in a virtual memory address environment. This register address is then mapped to physical memory address. What is a virtual address? Von Neumann had suggested that the execution of a program is possible only if the program and data is residing in memory. In such situation the program length along with data and other space needed for execution cannot exceed the total memory. However, it was found that at the time of execution, the complete portion of data and instruction is not needed as most of the time only few areas of program are being referenced. Keeping this in mind a new idea was put forward where only a required portion is kept on the memory while the rest of the program and data reside in secondary storage. The data or program portion which is stored on secondary storage is brought to memory whenever needed and the portion of memory which is not needed is returned to the secondary storage. Thus, a program size bigger than the actual physical memory can be executed on that machine. This is called virtual memory. The virtual memory has been discussed in greater details in Block-2 of Course-2.

The typicality of virtual addresses are that:

- they are longer than the physical addresses as total addressed memory in virtual memory is more than the actual physical memory
- if a virtual addressed operand is not in the memory then the operating system brings that operand to the memory.

The symbols D, D_1, D_2,..., etc. refer to actual operands to be used by instructions for their execution. Now let us discuss about the various addressing schemes.

1.3.1 Immediate Addressing

Under this addressing scheme, the actual operand D is A, the content of the operand field: i.e.

$$D = A$$

This addressing mode is used to initialise the value of a variable. The advantage of this mode is that no additional memory accesses are required for executing the instruction. However, as the size of instruction and operand field are limited, the type of data specified under this addressing scheme also get restricted.

1.3.2 Direct Addressing

Under this addressing scheme, the content A of the operand field specify EA, the effective address of the operand: i.e.

$$\begin{aligned} EA &= A \text{ and} \\ D &= (EA) \end{aligned}$$

The second statement implies that the data is stored in the memory location specified by effective address. In this addressing scheme only one memory reference is required. This simple addressing scheme provides a limited address space. If the address field has n bits then the address space available is 2^n memory locations.

1.3.3 Indirect Addressing

Under this addressing scheme the effective address EA and the contents of the operand field are related by

$$\begin{aligned} EA &= (A) \text{ and} \\ D &= (EA) \end{aligned}$$

The disadvantage of this addressing scheme is that it requires two memory references to fetch the operand. The first memory reference is used to fetch the effective address from the memory and second for fetching the operand using EA. In this scheme the addressed space is determined by word length. In many machines multiple level of indirection may be used where EA = (... (A) ...) gives the relationship between A and EA. In such a case a bit (generally the most significant bit) in the word address specifies the indirection. If this bit is '1', then the contents of the word represent the address of the address of the operand and if it is '0' then the contents of the field represent the address of the operand. If the size of the address field is n then the address space available under multiple indirection addressing scheme is 2^{n-1}.

1.3.4 Register Addressing

In this addressing scheme, the instruction specifies the address of the register containing the operand:

$$\begin{aligned} EA &= R \\ D &= (EA) \end{aligned}$$

Please note that EA here is a register address and not a memory address. The advantage here is that only a few bits are needed to address the operand. For example, for a machine having 16 general purpose registers only 4 bits are needed to address a register.

In some cases the address of the register containing the operand may not be explicitly specified but is understood implicitly. This is generally the case where one of the operands is in a special register called the Accumulator.

Register access is faster than memory access. So register addressing provides faster instruction execution. However, this statement is valid only if the registers are employed efficiently. For example, if an operand is moved into a register and processed only once and then returned to memory, then no saving occurs, however if an operand is used repeatedly after bringing into register then we have saved few memory references. Thus the task of using register efficiently deals with the task of finding what operand values should be kept in registers such that memory references are minimised. Normally, this task is done by a compiler of a high level language while translating the program to machine language.

1.3.5 Register Indirect Addressing

Under this addressing scheme the operand field specifies the registers which contains the address of the operand.

$$\begin{aligned} EA &= (R) \text{ and} \\ D &= (EA) \end{aligned}$$

The address capability of register indirect addressing scheme is determined by the size of the register.

1.3.6 Displacement Addressing

This is a very powerful addressing scheme. It combines both the direct addressing as well as the register indirect addressing schemes. Here the content A of the operand field is related to EA by

$$EA = A+(R)$$

The register address R may be specified explicitly or implicitly in the instruction. Depending upon the use and the implementation this address scheme may be known as:

Indexed Addressing Scheme: This addressing scheme is generally used to address ue consecutive locations of memory (which may store the elements of an array). The interpretation of the expression EA = A+(R) is as follows:

The contents of the operand field A is taken to be the address of the initial or the reference location (or the first element of array). The contents of register R gives the displacement with respect to the *reference location.* For example, to address of an element B_i of an array B_1, B_2,....B_n, with each element of the array stored in two consecutive locations, and the starting address of the array being 101, the operand field A shall contain the number 101 and the register R will contain the *value of the expression* $(i-1) \times 2$. Thus, for the first element of the array the register will contain 0. For addressing 5th element of the array, the A=101 where as register will contain $(5-1) \times 2 = 8$. Therefore, the address of 5th element of array $B_5=101+8=109$. The B_5, however, is stored in location 109 and 110. To address any other element of the array, changing the content of the register (let us call it index register) will suffice. As the index register are used for iterative applications, therefore, an index register is incremented or decremented after each reference to it. In several systems this operation is performed automatically during the course of an instruction cycle. This feature is known as autoindexing. Autoindexing can be autoincrementing or autodecrementing. The choice of register to be used as an index register differs from machine to machine. Some machine employ general purpose registers for this purpose while other machines may specify special purpose registers referred to as index registers.

Another related addressing scheme which couples the indirect addressing with indexing are also utilised by several systems. Here, there are two possibilities:

Indexing is performed after indirection (postindexing):

In this scheme the memory address specified by opcode address the location that contains a direct address which is to be indexed. That is:

$$\begin{aligned} DA &= (A) \\ EA &= DA + (R) \\ D &= (EA) \end{aligned}$$

(DA is Direct address)

Indexing performed before indirection (Preindexing):

This means that the address generated after doing indexing is the address of the location of operand. That is

$$\begin{aligned} IA &= A + (R) \\ EA &= (IA) \\ D &= (EA) \end{aligned}$$

(IA is indexed address)

In normal circumstances both pre-indexing and post-indexing are not used in an instruction set simultaneously.

Base Addressing Scheme: This addressing scheme is generally employed to relocate the programs in the memory specially in a multiprogramming environment. Relocation is discussed in details in course 2, block 2. Here the register R, referred to as Base Register contains the initial address in the memory (referred to as the base address) of the program segment being relocated. The operand field A contains the displacement of an instruction oı data with respect to the base address. In this case:

$$EA = A + (B) \; ; \quad D = (EA)$$

(B) refers to the contents of a base register B.

The contents of the base register may be changed in the privileged mode only, i.e. in the user mode the contents of the base register cannot be changed.

The base addressing scheme while on one hand provides the enhanced addressable space on the other hand it provides protection of users from one another.

In a base addressing scheme the address of an index addressed element is given by:

EA = A + (B) + (I), where B and I are base register and index register respectively.

Like index register a base register may be a general purpose register or a special register reserved for base addressing.

Relative Addressing Scheme: In this addressing scheme, the register R is the program counter (PC) containing the address of the current instruction being executed. The operand field A contain the displacement (positive or negative) of an instruction or data with respect to the current instruction. This addressing scheme have advantages if the memory references are nearer to the current instruction being executed.

1.3.7 Stack Addressing Scheme

This is not a very common addressing scheme. In this addressing scheme, the address of an operand is not specified explicitly. It is implied. The operand is found on the top of a stack. In some machines the top two elements of stack and top of stack pointer is kept in the CPU registers, while rest of the elements may reside in the memory. Figure 12 sums up the operating principles of all these addressing schemes.

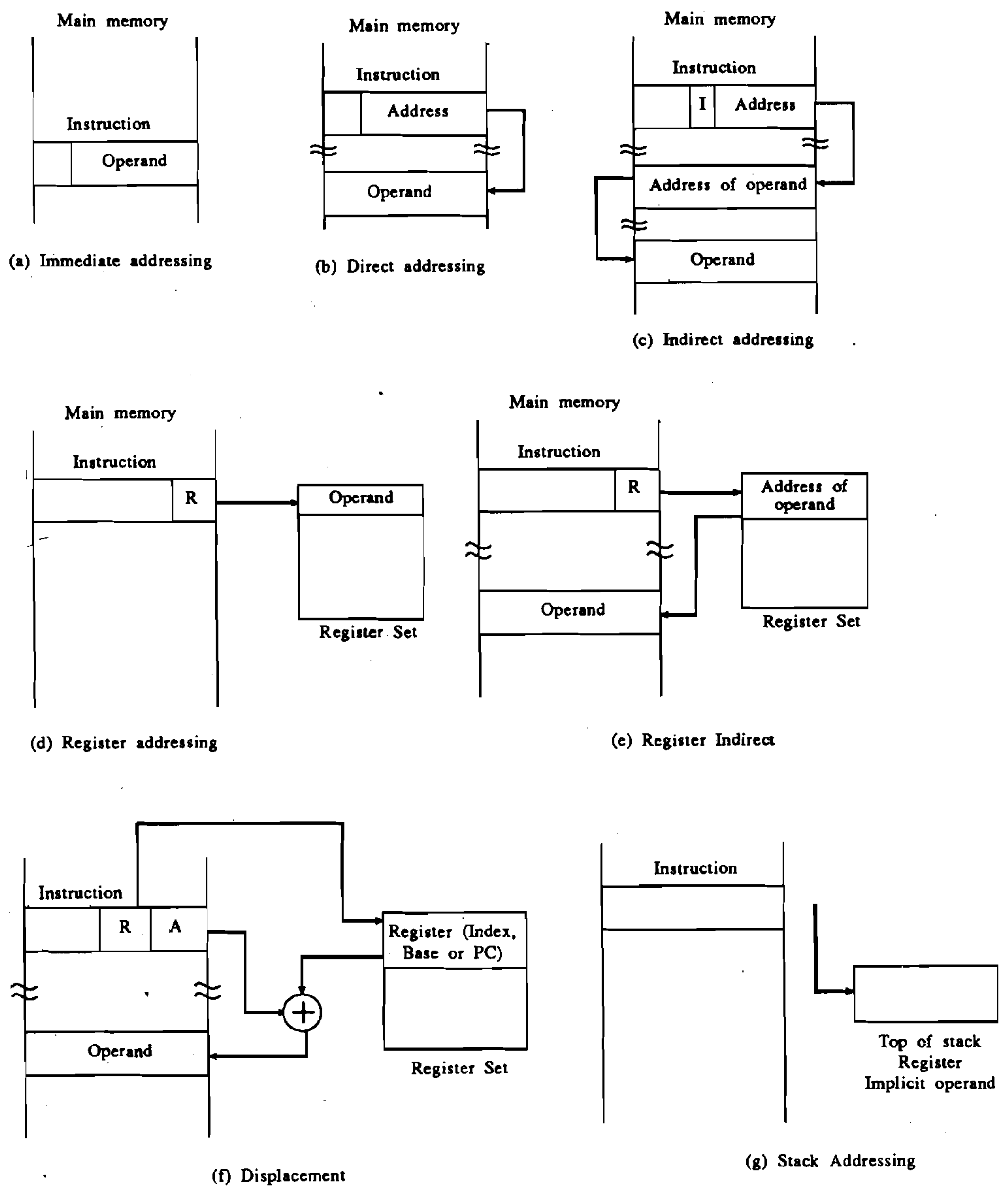


Figure 12 : Basic addressing modes

In general not all of the above modes are used for all applications. However, some of the common areas where compilers of high level languages use them are:

Autoindex mode	→	for pushing or popping the parameters of a procedures.
Direct mode	→	Normally used for global variables (used a little bit less than local variables)
Register	→	For holding local variables of procedures (frequently used)
Index	→	Accessing iterative local variables such as arrays
Immediate	→	For moving constants, initialisation of variables
Register indirect	→	For holding pointers to structure in programming languages such as records in pascal

Check Your Progress 3

1. Find out the memory references required to get the data for the following addressing modes:
   - Direct addressing
   - Indirect addressing
   - Register indirect addressing
   - Immediate addressing

   ....

   ....

2. State true or false

   (a) Immediate addressing is best suited for storing floating point numbers.

   True ☐ False ☐

   (b) Indirect addressing require fewer memory accesses than that of index addressing

   True ☐ False ☐

   (c) Index addressing is used for addressing arrays.

   True ☐ False ☐

3. What are the differences between preindexing and postindexing?

   ....

   ....

4. What are the differences between base and relative addressing schemes?

   ....

1.4 INSTRUCTION FORMAT DESIGN

As discussed earlier an instruction consist of an op-code and one or more operands which are addressed implicitly or explicitly. An instruction format is used to define the layout of the bits allocated to these elements of instructions. In addition, the instruction format explicitly or implicitly indicates the addressing modes used for each operand in that instruction.

As far as the designing of instruction format goes, it is a complex art. The computers have a variety of instructions designed for them in last so many years. We will discuss in this section about the design issues for instruction sets of the machines. We will discuss only point wise details of these issues.

1.4.1 Instruction Length

Significance: It is the most basic issue of the format design. It determines the richness and flexibility of a machine.

Basic Tradeoff: Smaller instruction (less space) Vs desire for more powerful instruction repertoire.

Normally programmer desire:

- more Op-code and operands: as it results in smaller programs.
- more Addressing modes: for greater flexibility in implementing functions like table manipulations, multiple branching.

However, a 32 bit instruction although will occupy double the space and can be fetched at double the rate of a 16 bit instruction, but can not be doubly useful.

Factors which must be considered for deciding about instruction length

Memory Size	:	If larger memory range is to be addressed, then more bits may be required in address field.
Memory organisation	:	If the addressed memory is virtual memory then memory range which is to be addressed by the instruction is larger than physical memory size.
Memory Transfer length (in bus system is equal to the data bus length)	:	Instruction length should normally be equal to data bus length or multiple of it.
Memory transfer	:	The data transfer rate from the memory ideally should be equivalent to the Processor speed. It can become a bottleneck if processor executes instructions faster than the rate of fetching the instructions. One solution for such problem is to use cache memory or another solution can be to keep instruction short.

Normally an instruction length is kept as a multiple of length of a character (that is 8 bits), and equal to the length of fixed point number. The term word is often used in this context. Usually the word size is equal to the length of fixed point number or equal to memory-transfer size. In addition, a word should store integral number of characters. Thus, word size of 16 bit, 32 bit, 64 bit are be coming very common and hence the similar length of instructions are normally being used.

1.4.2 Allocation of bits

The tradeoff here is between the number of opcode versus the addressing capabilities. An interesting development in this regard is the development of variable length opcode. An opcode has a minimum length and then depending on the instructions which require fewer operands or less powerful addressing mode, the opcode can be changed.

Let us discuss the factors which are considered for selection of addressing bits:

- **Number of addressing modes:** The more are the explicit addressing modes the more bits are needed for mode selection. However, some machines have implicit modes of addressing.
- **Number of operands:** Fewer number of operand references in an instruction although require less bits yet result in longer programs. Now a days many of the machines are having two operand references in an instruction. Each of these operands may need a mode indicator field of its own or both the operands may have the same addressing mode indicator field, that is the same addressing mode.
- **Register addressing versus memory addresses:** If more and more registers can be used for operand references then instructions are bound to be smaller as number of registers is far less than the memory size, therefore, they require only few bits in comparison to the bits needed for the memory addresses. In general, the number of user visible registers provided is 8 to 16. This number is found to be an optimum number of registers by several studies.
- **Register Sets:** Even in the case of register address as the trend is moving from a large number of general purpose registers to two or more sets of registers, for specialized data

storage. For example, one special register set can store only the data for calculation, while the other can store only the addresses. This results in further decrease in the size of instruction bits for register addressing. For example, in case a machine has 16 general purpose registers then a register address of it require at least 4 bits, however, if these 16 registers are used as two specialized sets of registers than one of the 8 registers need to be addressed at a time, thus requiring only 3 bits for register addressing mode.

- **Range of address**: The range of main memory addresses which need to be addressed directly or indirectly are involved for having a specific number of bits in instructions. For example, if direct addressing is used then the addressed memory determines directly the number of instruction bits required. However, in displacement index addressing schemes it is the offset which can control the number of bits desired in the instruction.

- **Granularity of address**: As far as memory references are concerned, granularity implies whether an address is referencing a byte or a word at a time. This is more relevant for machines which have 16 bit, 32 bit and higher bits words. Byte addressing although may be better for character manipulation, however, requires more bits in an address. For example, memory of 4K words (1 word = 16 bit) is to be addressed directly then it requires:

WORD Addressing	=	4 K words
	=	2^{12} Words
	⇒	12 bits are required for word addressing.
Byte Addressing	=	2^{12} Words
	=	2^{13} Bytes
	⇒	13 bits are required for byte addressing.

1.4.3 Variable-length of Instructions

With the better understanding of computer instruction sets, the designers came up with the idea of having a variety of instruction formats of different length. What could be the advantages of such a set? The advantages of such a scheme are:

- Large number of operations can be provided which have different lengths of instructions.

- Flexibility in addressing scheme can be provided efficiently and compactly.

However, the basic disadvantage of such a scheme is to have a complex CPU. However with the advances in technology and increase in basic understanding about the CPU designing may reduce the overheads required for added complexity of such a scheme. However, one condition which still holds for length of instructions is that all the possible instruction lengths should be multiple of word size.

An important aspect about these variable length instruction is: " The CPU is not aware about the length of next instruction which is to be fetched". This problem can be handled if each instruction fetch is made equal to the size of the longest instruction. Thus, sometimes in a single fetch multiple instructions can be fetched.

After discussing so much above various concepts, now let us look at the instruction set of IBM S/370 machine in the next section.

Check Your Progress 4

State true or false.

1. A long instruction can be executed faster than short instructions.

   True ☐ False ☐

2. Virtual addresses require more bits for the address part of instruction in comparison to non-virtual addresses.

   True ☐ False ☐

3. The speed gap between processor and memory suggests that instruction size should be as big as possible.

   True ☐ False ☐

4. In general, the instruction length is kept equal to the size of floating point storage format.

   True ☐ False ☐

5. Large number of operand addresses in instruction leads to smaller programs.

   True ☐ False ☐

6 Register addresses for specialised register set machines is smaller than that of the machines having general purpose register set.

   True ☐ False ☐

7. A machine using direct addressing mode having a memory addressing capability of 8 K bytes, require 13 bits for byte addressing.

   True ☐ False ☐

8. In a variable length instruction format, an instruction fetch must fetch the words equal to the size of smallest instruction.

   True ☐ False ☐

1.5 EXAMPLE OF INSTRUCTION SET

In this section we will look at the instruction set of a very important vendor the IBM. We have not included instruction set details of any of the INTEL microprocessor in this unit. One such microprocessor architecture is given in the block 3 of this course. The other instruction sets can be studied by you from the further readings.

IBM 370 Architecture: The IBM S/370 is a popular architecture. Let us discuss about the features provided in this architecture.

Data types: It provides the following data types:

Data type	*Allowed Length*	*Characteristics*
Binary Integers	16 and 32 bits	Unsigned i.e. non-negative numbers or signed binary integer in signed 2's complement notation.
Floating point	32, 64 and 128 bits	A 7 bit exponent with all the formats.
Decimal numbers	1 to 16 bytes	Rightmost 4 bits are used for sign. Thus, 1 to 31 digit decimal number can be represented. Arithmetic on these packed decimal integers is provided.
Binary logical	8, 32 and 64 bits and variable length logic data till 256 byte	Logic operations are defined for the data units of the given length.
Character	8 bits	EBCDIC are used.

Operation Types: Although the IBM/370 principal operation manual classifies the machine instructions in six broad categories, yet for the sake of simplicity they have been separated according to function by Stallings. This classification categories the instruction type in 12 categories:

Instruction Type	**Description**
Fixed point arithmetic Instructions	One register is used for storing one of the operand and the result, while the second operand can be in another register or in memory. It provides operations in either as unsigned 16 or 32 bit integer operands or 16 or 32 bit 2's complemented integer numbers.
Logical Instruction	Provided bitwise AND, OR and XOR operations.
General Register Shifting Instructions	Provides 8 shift instructions pairing the following options:

(1) Left or right shift
(2) Use of single or double register
(3) Signed or logical shift

For example, two of the eight shift instructions will be a left, single register, logical shift instruction and a left, double register, signed shift instruction. A second operand is used in the instruction to specify the amount of shift. The shift instruction effects either a single register or this can effect an even-odd register pair. The arithmetic shift instruction leaves the sign bit intact.

General Register - load and store Instructions	Provided transfer operations as: — Register to register transfer — Register to memory transfer — Memory to register transfer
Compare Instructions	It performs the comparing and testing functions. A condition code of 2 bit is set by the instruction.
Branching Instructions	The value of conditional code determines the branch. A special instruction called Branch and link is used for subroutine call and return. The user need to specify the register to be used for storing return address.
Conversion Instructions	Converts various forms of data for example, from decimal to binary
Decimal Instructions	Provided for providing operations on decimal data. These instructions include arithmetic operations, shifting operations and unpacking operations.
Floating point Instructions	Add, subtract, multiply and divide instructions are provided on 4 byte, 8 byte and 16 byte floating point numbers.
Special purpose control Instructions	These instruction causes a control passing. Example of one such instruction is supervisory call which causes an interrupt that results in passing the control to the operating system.
Privileged I/O Instructions	These instructions are used for starting Input/Output using a I/O channels. One such command is start I/O.
Privileged System Control Instructions	These instructions are issued only by the operating system for controlling registers and data structures which are needed by the operating system.

IBM 370 Addressing

In IBM 370 memory is addressable at byte level. However, all the addresses are virtual addresses. In the system S/370 the length of virtual address is 24 bits whereas, the length in case of IBM S/370 XA it is 31 bits. In this system only three basic addressing modes are supported. These are:

- Immediate Addressing : It is a 1 byte operand, provided within the instruction.
- Register Addressing : In this mode a register operand is addressed. This register operand can be in one of the 16 32-bit general purpose register or 4 64-bit floating point registers. A register address consist of 4 bits as we have 16 general purpose registers. However for the sake of consistency, even if we are referencing floating point register which can be referenced by 2 bits (4 floating point registers), 4 bits are used.
- Displacement Addressing: In this mode the operand is stored in the virtual memory. Here two type of formats are used :

  The base register format: The operand reference in such scheme consist of two components:

  The displacement provided in the instruction

The 4 bit register address, this register is one of the general purpose register and will be used as a base register. However, only the right most 24 bits of a 32 bit register are used for computing address in IBM /360 whereas only rightmost 31 bits are used in IBM 370 XA model. A register address 0 specifies that no register has been used and the displacement becomes the direct address of memory location in that case.

Use of index register along with the base register:

In this scheme the instruction reference consist of the displacement, a four bit reference to a general purpose register to be used as base register and a general purpose register to be used as index register. The feature of autoindexing is also provided.

Let us summarise the addressing modes used in IBM/370.

Addressing Mode	*Calculation of effective address*
Immediate	Operand = A
Register	EA = R
Displacement :	
Base Register	EA = A + (B)
Index on base register	EA = A +(B) + (I)

I → Indicates address of index register. It will be 4 bit long.

B → Indicates address of the base register. It will be 4 bit long.

Figure 13 : Summary of IBM S/370 Addressing Modes

The addressing schemes provided by this system although are simple and easy yet a programmer must understand how to use the segments in IBM system/370. This discussion is beyond the scope of this unit. You can refer to further readings for more details.

Instruction Format

We will present a simplified tabular representation for the instruction format of IBM system/370.

Features:

- Variable length instructions : bytes, 4 bytes, 6 bytes instructions
- Variable op-code length : 1 byte and 2 byte
- Eight different instruction formats are used.
- Mostly two operand instructions, however, one and three operands are also used in some instructions.
- First two bits of instruction specifies:
  length of the instruction and
  format of the instruction

First two bits of opcode	Instruction length (bytes)	instruction opcode format
00	2	R R
01	4	R X
10	4	R R E / R S / S / S I
11	6	S S / S S E

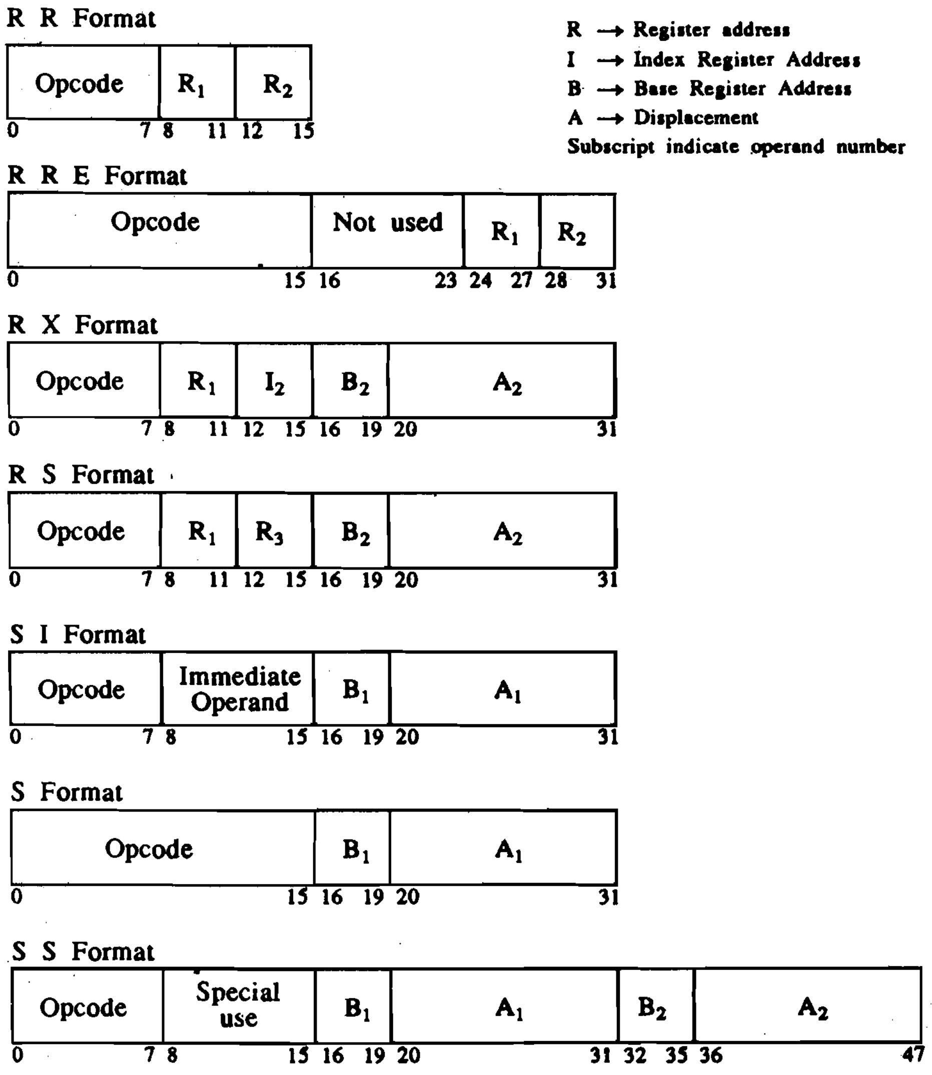
The instruction formats are:

- Register-Register (R R) : A compact representation
  Instruction length : 2 bytes
  opcode : 1 byte
  Register addresses : 2 nos of 4 bit each.
- Register Register Extended : Used for privileged instructions used by the operating system.

(R R E)	Instruction length	4 bytes
	Opcode	2 bytes
	Register addresses	2 of 4 bit each 1 byte is unused.
• Register_Indexed (R X)	Instruction Length	4 bytes
	Opcode	1 byte
	First operand	
	Register operand	One of 4 bit
	Second operand	
	Virtual memory operand	One 4 bit Index Register 4 bit base register 12 bit displacement
• Register_Storage (R S)	Instruction length	4 byte
	Number of operands	3 (the only three operand instruction format)
	opcode	1 byte
	First operand and Third operand	Register operands require 4 bit each for register reference
	Second operand	Virtual memory operand using a base register (4 bit) and displacement (12 bits)
Storage Immediate (S I)	Instruction length	4 bytes
	Opcode	1 byte
	First operand	Virtual memory operand using a base register(4 bits) and displacement (12 bits)
	Second operand	Immediate operand of one byte
Storage (S)	Used for representing privileged instructions for I/O or system control.	
	Instruction length	4 bytes
	Opcode	2 bytes
	Number of operands	one
	Operand address is virtual address using a base register (4 bits) and displacement (12 bits)	
• Storage_Storage (S S)	Instruction length	6 bytes
	opcode	1 byte
	Number of operand	= 2
	both operands are virtual memory operands specified by base register (4 bits) and displacement (12 bits) *Use of remaining 1 byte* : One length format : The remaining byte specifies the number of bytes to be operated upon. Used for moving a block of characters from one location to another. Two length format : two length fields of 4 bit each specifying the size of each of the two operands. Used for operations on BCD. Register specifications : The byte designates two general purpose registers which contains the control information or length specifications. Used for privileged instructions.	
Storage_Storage Extended (S S E)	Used for privileged instructions	
	Instruction length	6 bytes
	Opcode	2 bytes
	Two operands both in the form of base register	

(4 bits) and displacem,ent (12 bits).

Thus, IBM System/370 formats tries to make efficient use of instruction lengths. Figure 14 gives the summary of instruction formats of this system.



Special Use : One field of 8 bit specifying length of Block (One length format)
OR
Two fields of 4 bit each specifying length (Two length format)
OR
Two fields of 4 bit each of register address (Register specification)

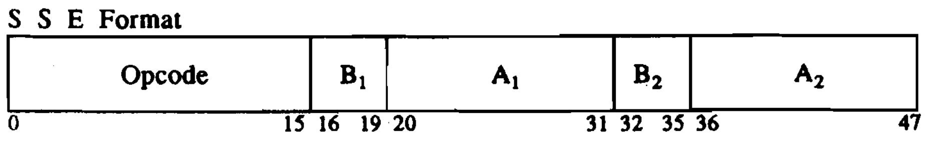


Figure 14 : Summary of IBM System/370 Instruction formats

1.6 SUMMARY

In this unit, we have introduced you to various concepts relating to an instruction. We have discussed about the basic characteristics, the number of addresses, type of operands and operations in instructions and various addressing modes. We have also highlighted the basic issues while designing instruction formats and presented details on the instruction set of IBM

S/370. Please note we have not provided you the detailed instructions which this machine provide. But only the conceptual model behind this instruction set. You can refer to further reading for more details on instruction set for various machines.

1.7 MODEL ANSWERS

Check Your Progress 1

1. False
2. False
3. False
4. False
5. True
6. False

Check Your Progress 2

1. a – (ii), b – (iii), c – (i)
2. See section 1.2.2
3. (a) – (i), (iii); (b) – (ii), (iv); (c)– (v); (d)–NIL
4. The reasons are:
   - → return address is needed
   - → parameter passing
   - → branching and coming back from a small program
5. Refer to sub-section on subroutine calls.

Check Your Progress 3

1. No model answer
2. (a) False, (b) False, (c) True
3. Refer sub-section 1.3.6
4. Refer sub-section 1.3.6

Check Your Progress 4

1. False, 2. True, 3. False, 4. False, 5. True, 6. True,
7. True, 8. False.