3.3 Instruction Set Design

The most significant and complex task in designing a computer is framing its instruction set. For achieving this goal, the following queries must be answered. How many instructions are needed? Which type of instructions have to be included in the instruction set? The early computers had unplanned instruction set. Their weak instruction set design drastically affected the main memory space by lengthy (machine language) programs. Hence, a well-designed, pre-planned instruction set enables the compilers to create a compact object code (machine language program) saving upon the memory space.

A computer architect has to consider the following aspects before finalizing the instruction set:

1. *Programming convenience*: Number of instructions; programmers prefer to have as many instructions as possible so that appropriate operations are carried out by the respective instructions. Also, too many instructions in the instruction set results in a complex control unit design. Instruction decoding requires huge circuitry and time.
2. *Powerful addressing*: Programmers are happy if all possible modes of addressing the operands are present in the architecture. This gives a lot of flexibility to the programmer. Though the control unit design becomes complex. Operand address calculation activity take into account all the possible addressing modes.
3. *Number of general purpose registers (GPRs)*: If the CPU has a large number of registers, the programmer finds data movement and processing faster. But, cost of CPU hardware increases with the number of GPRs.
4. *Target market segment*: The application area aimed for the computer requires certain special operations for efficient data processing. A scientific computer should have floating-point arithmetic without which the precision would be heavily degraded. Whereas, a business computer must support decimal arithmetic, an entertainment computer must have multimedia operations.
5. *System performance*: If a program has less number of instructions, the performance of the system is enhanced since time spent by the CPU in instruction fetching is reduced. For short programs, instructions should be powerful. Thus, a single instruction must be able to perform several micro-operations. Programmers appreciate this as it reduces the program size. But, the flipside of the coin is increase in both control unit complexity and instruction execution time. The modern concept of RISC architecture does not support complex instructions, though all the old CISC based computers use powerful instructions.

Traditionally, the superiority of a computer is decided on the basis of its instruction set. The total number of instructions and their powerfulness is given utmost importance since these two factors contributed to the efficiency of the computer. An efficient program is the one which is short and occupies less memory space. Execution time is also a key factor.

The modern trend follows the use of simple instructions which results in a simplified control unit. The CPU circuitry is more important then the memory size. Thus, the CPU speed is enhanced for instruction processing of a RISC architecture.

The selection of an instruction set for a computer depends on the manner in which the CPU is organized. Traditionally, there are three different CPU organizations with certain specific instructions:

1. Accumulator based CPU
2. Registers based CPU
3. Stack based CPU

3.3.1 Accumulator based CPU

Initially, computers had accumulator based CPUs. It is a simple CPU, in which the accumulator contains an operand for the instruction. Similarly, the instructions leave the result in the accumulator. The contents of the accumulator participate in the arithmetic operations such as addition, subtraction etc. This is also known as one address machine. The PDP-8, is the first minicomputer, which contains this type of CPU and is used for process control and laboratory applications. The Mark I computer cited in Chapter 2 is also a typical accumulator based computer. This style of CPU organization has been totally replaced by the introduction of the new register based CPU.

Example 3.1 Write an assembly language program to derive the expression X = (A + B) – (C + D) for an accumulator based CPU.

LOAD A	—	Loads accumulator with A
ADD B	—	Adds B to accumulator contents and stores the result in accumulator; Accumulator contains A + B
STORE T	—	Stores A + B in T, a temporary memory location
LOAD C	—	Loads C into accumulator
ADD D	—	Adds D to the accumulator contents and stores the result in Accumulator; accumulator contains C + D
SUB T	—	Subtracts accumulator contents from T and stores the result in accumulator
STORE X	—	Stores the accumulator contents in memory location X

The **advantages** of accumulator based CPU are as follows:

1. The accumulator contents is meant to be an operand. Hence, there is no requirement for the operand address field (for one operand) in the instruction. This results in short instructions and less memory space. Due to the absence of the address field, this type of CPU supports zero address instructions. Such CPUs normally have two types of instructions: zero address and single address. The single address instructions have one operand in main memory and the other in the accumulator.
2. Instruction cycle takes less time because it saves time in instruction fetching due to the absence of operand fetch.

The **disadvantages** of accumulator based CPU are as follows:

1. Program size lengthens due to the usage of many instructions in complex expressions. Hence the memory size increases.
2. Program execution time increases due to increase in the number of instructions per program.

3.3.2 Registers based CPU

In this type of CPU, multiple registers are used as accumulator. In other words, there are multiple accumulators. Such a CPU has a GPR organization. The use of the registers result in short programs with limited instructions. IBM System/360 and PDP-11 are some of the typical examples.

Example 3.2 Write an assembly language program to derive the expression X = (A + B) – (C + D) for a registers based CPU.

```
LOAD R1, A  —  Loads A into register R1
ADD R1, B   —  Adds B to R1 contents and stores the result in R1
LOAD R2, C  —  Loads C into R2
ADD R2, D   —  Adds D to R2 contents and stores the result in R2
SUB R1, R2  —  Subtracts R2 contents from R1 contents and stores the result in
               R1
STORE R1, X —  Stores the result in memory location X
```

Comparing with example 3.1, it is observed that the registers based CPU (GPR architecture) results in shorter program size than the accumulator based CPU. Also, the program for the accumulator based CPU requires a memory location for storing partial result. Hence, additional memory accesses are needed during program execution. Thus, increase in the number of registers, increases CPU effeciency. But care should be taken to avoid unnecessary usage of registers. Hence, compilers need to be more efficient in this aspect.

3.3.3 Stack based CPU

The stack is a push down list with a Last In First Out (LIFO) access mechanism. It stores the operands. It is present either inside the CPU or a portion of the memory can be used as a stack. A register (or memory location) is used to point to the address of the top vacant location of the stack. This register is known as the Stack Pointer (SP). When nothing is stored in the stack, the stack is empty and the stack pointer points to the bottom of the stack. When an item is stored in the stack, it is called PUSH operation; the stack pointer is decremented. When the stack is full, the stack pointer points to the top of the stack. When any item is removed from the stack (POP operation), the stack pointer is incremented.

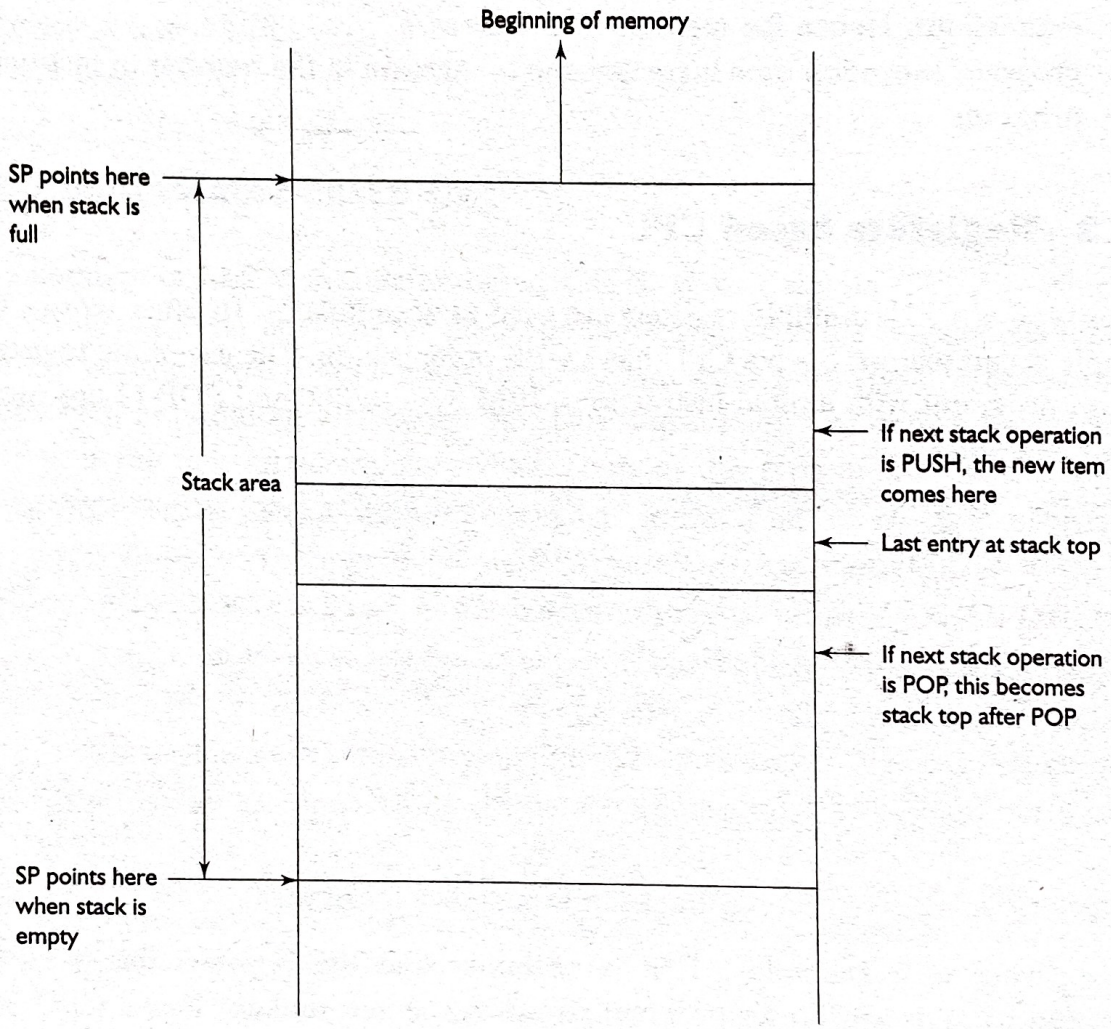


Fig. 3.3 Stack concept

The item which is pushed into the stack last (recently) comes out first in the next POP operation. In a stack based CPU, all operations by the CPU are done on the contents of a

stack. Similarly, the result of an operation is stored in stack. Figures 3.3 and 3.4 illustrate the stack concept and mechanism. On executing an arithmetic instruction such as ADD, the top operands are poped-off the stack. Burroughs B5000 and HP 3000 are some examples of stack computers.

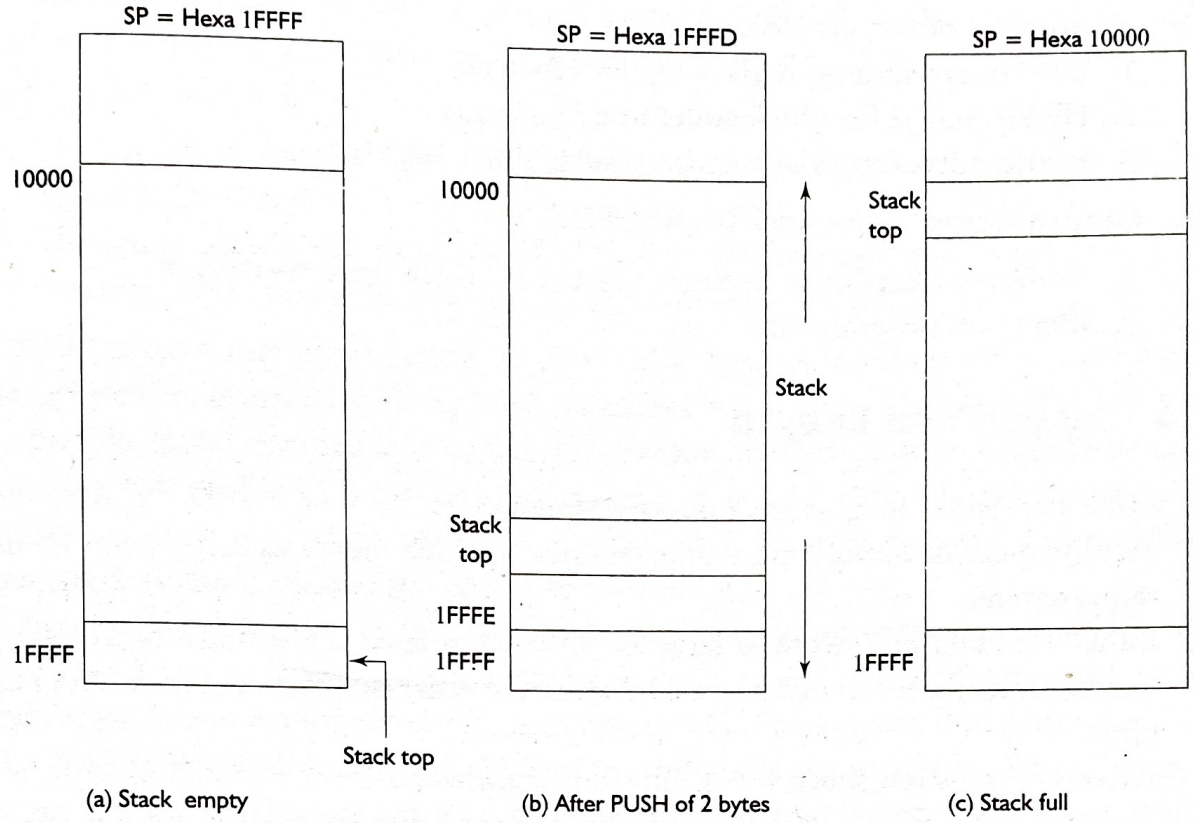


Fig. 3.4 Stack operation

Example 3.3 Write an assembly language program to evaluate the expression X = (A + B) – (C + D) for a stack based computer.

Statement	Stack contents after instruction execution	Stack locations occupied
PUSH A	A	1
PUSH B	A,B	2
ADD	A + B	1
PUSH C	(A + B), C	2
PUSH D	(A + B), C, D	3
ADD	(A + B), (C + D)	2
SUB	(A + B) – (C + D)	1
POP X	Empty (Nil)	0

From the contents of stack, it is observed that the stack grows when some PUSH operation takes place. When an arithmetic instruction is executed, the operands are removed from the stack and the result occupies a position in the stack top. Example 3.3 shows that the program size for the stack computer is more for the registers based CPU.

The **advantages** of the stack based CPU:

1. Easy programming/high compiler efficiency
2. Highly suited for block-structured languages
3. Instructions don't have address field; short instructions

The **disadvantages** of the stack based CPU:

1. Additional hardware circuitry needed for stack implementation
2. Increased program size

3.4 Addressing Modes

The two common options used for locating the operands of an instruction are the main memory locations and the CPU registers. If the operand is located in the main memory, the location address has to be given by the instruction in the operand field. It is not necessary to give the address explicitly in the instruction. Many useful methods are followed to specify the operand address. The different modes for specifying the operand address in the instructions are known as addressing modes. A given computer may not use all the addressing modes. The popular addressing modes are as follows:

1. Immediate addressing
2. Direct (absolute) addressing
3. Indirect addressing
4. Register addressing
5. Register indirect addressing
6. Relative addressing
7. Index addressing
8. Base with index addressing
9. Base with index and offset addressing

Why do we need so many addressing modes? Multiple addressing modes give flexibility to the programmer in writing efficient (short and fast) programs. The following goals influence the computer architect while choosing the addressing modes:

1. Reducing the instruction length by having a short field for the address.
2. Providing powerful aids to the programmer for complex data handling such as indexing of an array, loop control, program relocation etc.

The exact addressing mode used by an instruction is indicated to the control unit, in the following two ways:

1. A separate mode field in the instruction indicates the addressing mode used, as shown in Fig. 3.8.

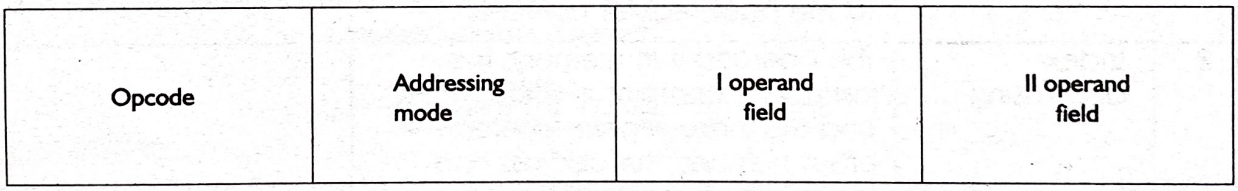


Fig. 3.8 Addressing mode field

2. The opcode itself explicitly specifies the addressing mode used in the instruction.

A given computer is designed to use one of the two techniques. Table 3.2 briefly defines the addressing modes. A detailed description of the addressing modes is given below:

TABLE 3.2 Addressing modes and mechanisms

S. no.	Addressing mode	Mechanism	Remarks
1	Immediate	Operand is present in the instruction itself	Fast operand fetch
2	Direct	Operand is in memory; its location address is given in the instruction	One memory access required for operand fetch
3	Indirect	Operand is in memory; its address is also in memory; address of the location containing operand address is given in the instruction	Additional memory access required for knowing the operand address
4	Register direct	Operand is in a register; the register address (number) is given in the instruction	Quickest access of operand without any memory access; faster than direct addressing
5	Register indirect	Operand is in memory; its address in a register; address (number) of the register containing the operand address is given in the instruction	Faster than indirect addressing
6	Relative addressing	The operand is in memory; its relative position (offset number) with respect to the contents of program counter is given in the instruction	Quick address calculation without memory access
7	Base register addressing	The operand is in memory; its address is specified in two parts; the instruction gives an offset number and also specifies the base register; the offset number is added to the base register contents	
8	Index addressing	The operand is in memory; the instruction contains an address and the index register contains offset number; the address and the offset are added to get the operand address	

3.4.1 Immediate Addressing (Fig. 3.9)

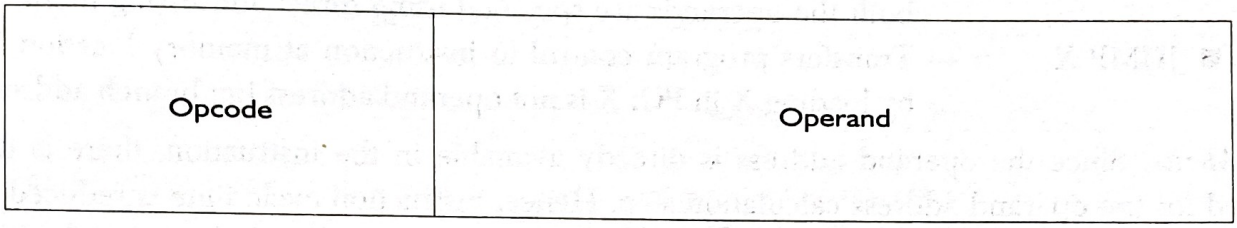


Fig. 3.9 Immediate addressing

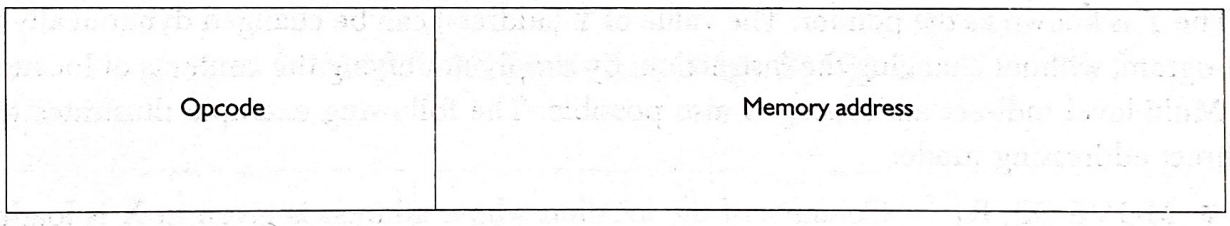
This mode is a simple one with no operand fetch activity. The instruction itself contains the operand. The following examples give assembly language statements for the immediate addressing mode. The (#) sign is used to indicate that the constant following the sign is the immediate operand.

- MOVE #26, R1 or MVI R1, 26—Loads binary equivalent of 26 in register R1
- ADD #26, R1—Adds binary equivalent of 26 to R1 and stores the result in R1
- CMP #26, R1 or CMI R1, 26—Matches R1 contents with binary equivalent of 26

Merits: This operand is available in the instruction as soon as the instruction fetch is over. Hence the instruction cycle winds-off fast.

Demerits: The value of the operand is limited by the length of the operand field in the instruction.

3.4.2 Direct Addressing (Fig. 3.10)



<Memory address> = Operand

Fig. 3.10 Direct addressing

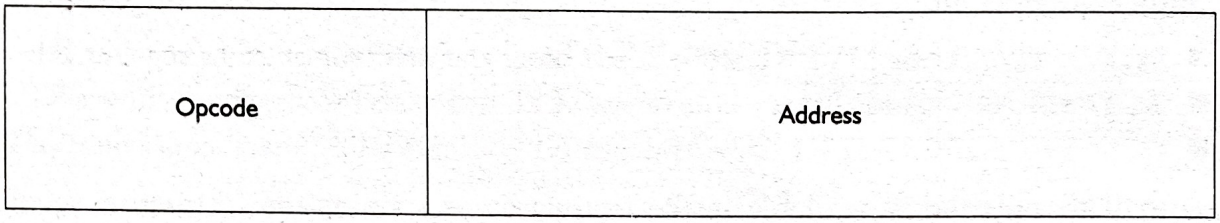
Since the operand address is explicitly given in the instruction, this mode is called direct addressing mode. The following assembly language statements illustrate this mode:

- LOAD R1, X — Loads the contents of memory location X in register R1
- MOV Y, X — Moves the contents of memory location X to the location Y. Here both the operands are specified using direct addressing mode
- JUMP X — Transfers program control to instruction at memory location X by loading X in PC; X is not operand address but branch address

Merits: Since the operand address is directly available in the instruction, there is no need for the operand address calculation step. Hence, instruction cycle time is reduced.

Demerits: The number of bits for the operand address is limited by the operand field in the instruction.

3.4.3 Indirect Addressing (Fig. 3.11)



<Address> = Operand address

Fig. 3.11 Indirect addressing

The instruction gives an address of a location (X) which, in turn contains another location's address (Y), which actually contains the operand. It can be represented as

$$\langle X \rangle = Y, \langle Y \rangle = \text{Operand}$$

The Y is known as the pointer. The value of Y (address) can be changed dynamically in a program, without changing the instruction, by simply modifying the contents of location X. Multi-level indirect addressing is also possible. The following example illustrates the indirect addressing mode:

- MOVE (X), R1 — Contents of the location whose address is given in X is loaded into register R1

Merits: Flexibility in programming; changing the address during program run-time without changing the instruction contents. It is useful for implementing pointers (C language).

Demerits: Instruction cycle time increases since two memory accesses are required even for a single level indirect addressing.

3.4.4 Register (Direct) Addressing (Fig. 3.12)

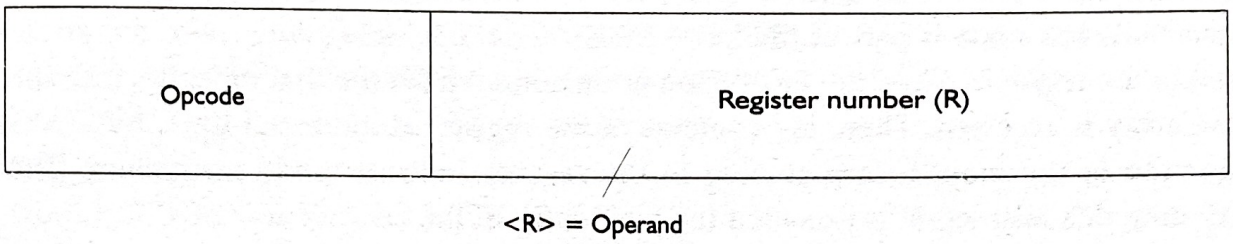


Fig. 3.12 Register addressing

Conceptually, register addressing is similar to direct addressing except that instead of the memory location, the register holds the operand. The instruction contains the register number that has the operand. This mode is very useful to a long program in storing the intermediate results in the registers rather than in memory. The following examples illustrate the register addressing mode:

- ADD R1, R2 — Adds the contents of registers R1 and R2 and stores the result in R1. Both the operands are addressed in register addressing mode
- STORE R1, MEM1 — Contents of register R1 are stored in memory address MEM1; register addressing is used for first operand and direct addressing is used for second operand.

Merits: Faster operand fetch without memory access.

Demerits: Number of registers is limited and hence, effective utilization by the programmer is essential. Otherwise, program execution time will increase unnecessarily.

3.4.5 Register Indirect Addressing (Fig. 3.13)

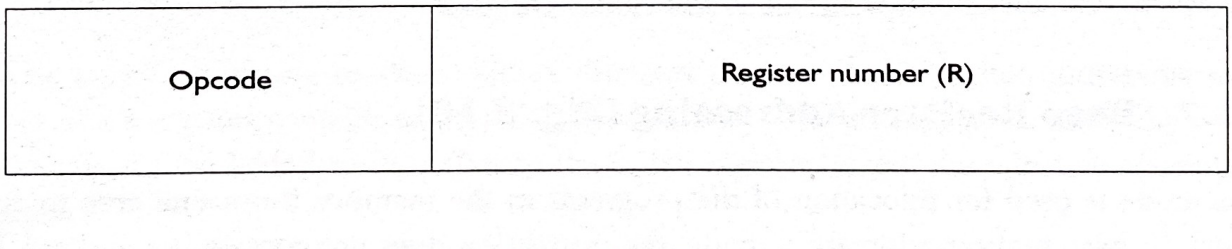
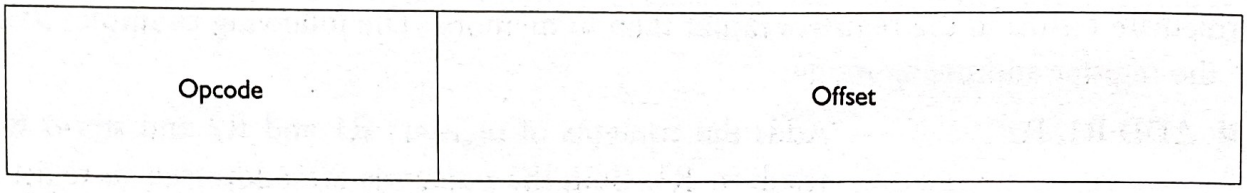


Fig. 3.13 Register indirect addressing

In this mode, a register is used to keep the memory address of the operand rather than the operand itself. Thus, the register acts as the memory address register. This mode is very useful for quick access of the main memory location such as an array. The instruction with register indirect mode is part of the loop. Initially, the beginning address of the array is stored in the register. When the instruction is encountered for the first time, the first entry of the array is accessed. Then, the contents of the register is increased by 1, by another instruction in the loop, before coming to the register indirect mode instruction. Thus, every time this instruction is executed the next entry in the array is accessed.

Merits: Effective utilization of the instruction length since register number is specified with few bits.

3.4.6 Relative Addressing (Fig. 3.14)



Operand address = < PC>+ Offset

Fig. 3.14 Relative addressing

In this mode, the instruction specifies the operand address (memory location) as the relative position of the current instruction address, i.e., the contents of PC. Hence, the operand is situated at 'short distance' from the PC contents. Generally, this mode is used to specify the branch address in the branch instruction, provided the branch address is nearer to the instruction address.

JUMP + 8 (PC)

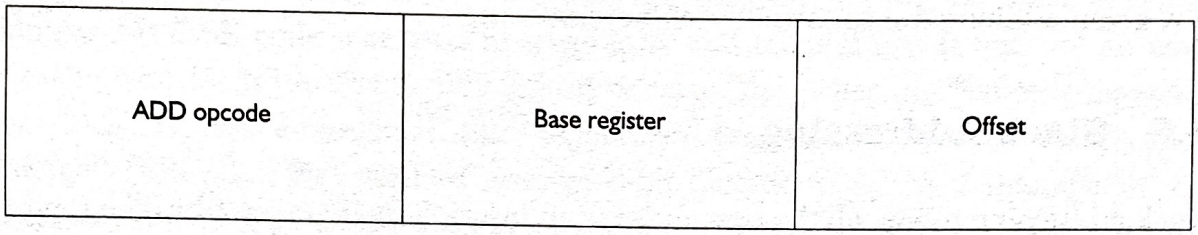
JUMP – 8 (PC)

Demerits: Smaller number of bits in the address field.

3.4.7 Base Register Addressing (Fig. 3.15)

This mode is used for relocation of the programs in the memory (from one area to another). In base register addressing mode, the instruction does not contain the address. It gives the displacement relative to the base address (contents of base register). Hence, to relocate the operand from the current memory area to another memory area, the base

register is loaded with the new base address. The instruction need not be modified. In this way, an entire program or a segment of program can be moved from one area of the memory to another without affecting the instructions, by simply changing the base register contents. This is important for multiprogramming systems since for different times (runs), different area of the memory is available for a program. A CPU may have multiple base registers.

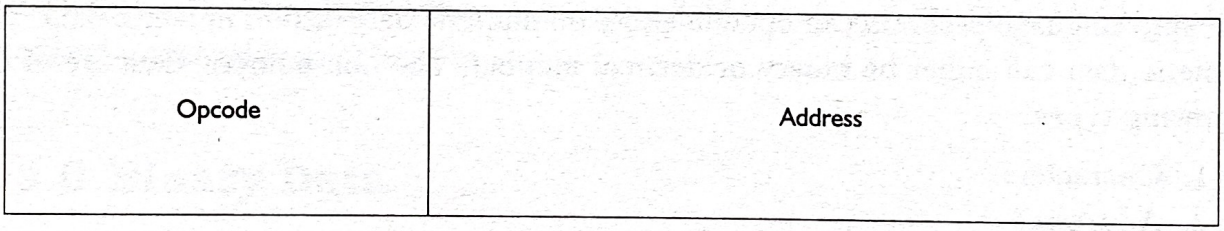


Operand address = < Base register > + Offset

Fig. 3.15 Base register addressing

Merits: The operand address field in the instruction is very short since it only gives offset (displacement); the operand address is calculated without memory access.

3.4.8 Index Addressing (Fig. 3.16)



Operand address = Address + <Index register>

Fig. 3.16 Index addressing

The index addressing mode is slightly different from the base register addressing mode. An index register contains an offset or displacement. The instruction contains the address that should be added to the offset in the index register, to get the effective operand address.

Generally, the address field in the instruction gives the start address of an array in memory. The index register contains the 'index value' for the operand i.e. the difference

between the start address and the operand address. By changing the value of the index register, any operand in the array can be accessed. Usually, the operands (array elements) are in consecutive locations. They are accessed by simply incrementing the index register.

Some CPUs support 'autoindexing' features, which involve automatic incrementing (by hardware) of the index register whenever, an instruction with index addressing is executed. This eliminates the use of a separate instruction to increment the content of the index register. This also, gives faster action as well as shorter program size. However, the control unit is given additional responsibility of 'autoindexing'.

3.4.9 Stack Addressing

In stack addressing mode, all the operands for an instruction are taken from the top of the stack. The instruction does not have any operand field. For example, an ADD instruction gives only the opcode (ADD). Both the operands are in the stack, in consecutive locations. When the ADD instruction is executed, two operands are popped-off the stack one-by-one. After addition, the result is pushed onto the stack.

Merits: No operand fields in the instruction. Hence, the instruction is short.