

STUDY MATERIALS ON COMPUTER ORGANIZATION

(As per the curriculum of Third semester B.Sc. Electronics of Mahatma Gandh Uniiversity)

Compiled by Sam Kollannore U..

Lecturer in Electronics

M.E.S. College, Marampally

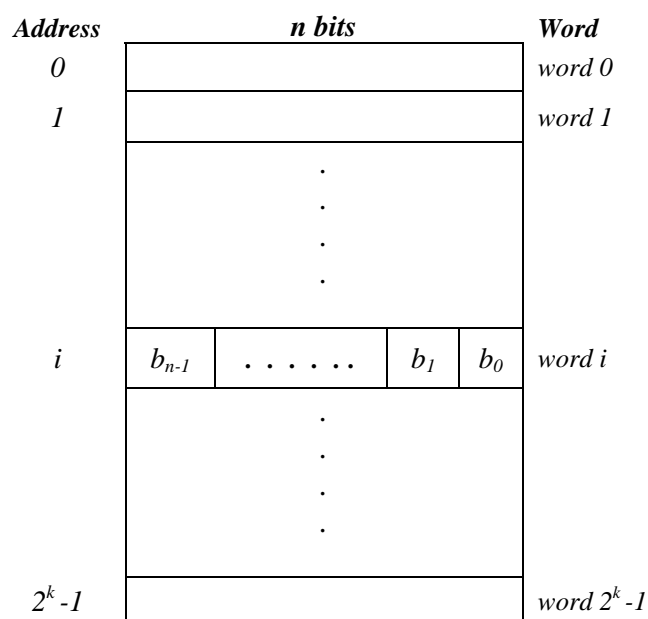
2. ADDRESSING METHODS

2.1 MEMORY LOCATIONS, ADDRESSES & INFORMATION ENCODING

Memory locations consist of millions of storage cells which can store a binary digit or bit having value 0 or 1. Group of n bits, called as a word of information can be stored and retrieved in a single basic operation. Here n is the word length – ranging from 16 to 64 bits

Accessing the main memory requires address for each word location 0 to $2^k - 1$; for some suitable value of k address lines or k address bits. i.e. Main memory of this computer can have up to 2^k words. For example a 24 bit address generates an address space of 2^{24} (16777216) locations $\equiv 16M$ and a 32 bit address creates an address space of 232 or 4G.

Contents of the memory locations can represent either instructions or operands.



Operands can be either numbers or characters.

Fig 2.1 Main Memory Address

2.1.1 Representation of Numbers in main memory

Consider a 32 bit pattern is used to represent a signed integer

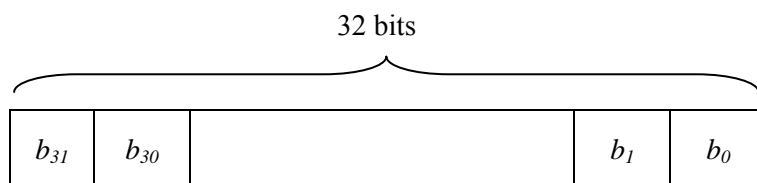


Fig 2.2

Sign bit : $b_{31} = 0$ for positive numbers
 $= 1$ for negative numbers
 Magnitude $= b_{30} \cdot 2^{30} + b_{29} \cdot 2^{29} + \dots + b_1 \cdot 2^1 + b_0 \cdot 2^0$

Magnitude can range from 0 to $2^{31} - 1$ and the numbers are said to be in binary positional notation. The above encoding format is called Signed Magnitude representation. The other two binary representations are 1's compliment and 2's compliment representations. Representation of positive numbers is the same in the three cases. The difference is only in the negative number representation. In all the three methods the left most bit is the sign bit (i.e. 0 represents positive number and 1 represents negative number) 2's compliment method is the most suitable one and is used in all modern computers.

2.1.2 Representation of Characters in main memory

Characters can be letters of the alphabet, decimal digits, punctuation marks etc. They are represented by codes that are usually 6 – 8 bits long.

Fig 2.3 shows how 4 characters in ASCII can be stored in a 32 bit word.

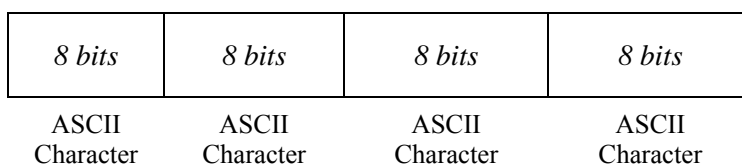


Fig 2.3

2.1.3 Representation of Instructions in main memory

A main memory word can also be used to represent an instruction. One part of the word specifies the operation to be performed and the other part specifies operand address. Each of these parts are called as 'field'.

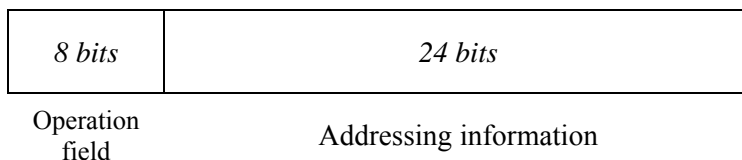


Fig 2.4

- Here the 8 bit operation field can encode 2^8 (256) distinct instructions.
- Addressing information is given in a variety of ways. The different ways in which operands can be named in machine instructions are called addressing modes.
- Memory words whose addresses are specified by the instructions are interpreted as operands. Whether an operand is a character or a numeric data item is determined by the operation field of the instruction.
- An operand may be either shorter or longer than one word. An operand length of 8 bit is convenient, because this size is used to encode character data. An 8 bit data is called a byte.

2.2 BIG-ENDIAN AND LITTLE ENDIAN ASSIGNMENTS

To enable instructions to refer to individual bytes, the smallest addressable unit in most computers is a byte rather than a word and such computers are called byte addressable. Individual bytes can be addressed in a byte addressable computer. If the word length is 32 bits, then 4 bytes can be placed in a word. The two different schemes used are :-

- i) Big-Endian Assignment
- ii) Little-Endian Assignment

In Big-endian assignment, bytes are numbered starting with most significant byte of a word. Word is given the same address as its most significant byte (used in 68000 & Power PC processors)

In Little-endian assignment, bytes are numbered from least significant byte of a word. Word is given the address of its least significant byte.

Both the assignments are used in commercial machines. In both cases, byte addresses 0, 4, 8, . . are taken as the addresses of successive words in the memory. These addresses are used when specifying memory read and write operations for words.

Word Address	Byte Address			
0	0	1	2	3
4	4	5	6	7
	⋮			
$2^k - 4$	$2^k - 4$	$2^k - 3$	$2^k - 2$	$2^k - 1$

Fig 2.5(i) Big-endian assignment

Word Address	Byte Address			
0	3	2	1	0
4	7	6	5	4
	⋮			
$2^k - 4$	$2^k - 1$	$2^k - 2$	$2^k - 3$	$2^k - 4$

Fig 2.5(ii) Little-endian assignment

2.3 MAIN MEMORY OPERATIONS

To execute an instruction the instructions must be transferred from the main memory to the CPU. This is done by the CPU control circuits.

Operands and results must also be moved between the main memory and the CPU. Thus two basic operations involving the main memory are needed namely

- **Load (or Fetch or Read)**
- **Store (or Write)**

Load operation: Transfers a copy of the contents of a specific memory location to the CPU. Word in the main memory remains unchanged. To start a Load or Fetch operation, CPU sends the address of the desired location to the main memory and requests to read its contents. The main memory reads the data stored at that address and sends them to the CPU.

Store operation: Transfers a word of information from the CPU to a specific main memory location, destroying the former contents of that location. Here the CPU sends the address of the desired location to the main memory, together with the data to be written to that location.

2.3 INSTRUCTIONS AND INSTRUCTION SEQUENCING

A computer must have instructions capable of performing four types of operations

1. **Data transfers between the main memory and the CPU registers**
2. **Arithmetic and logic operations on data**
3. **Program sequencing and control**
4. **I/O transfers**

Notations used:-

- a) **Register Transfer Notation (RTN):-** Possible locations involved in transfer of information are memory location, CPU registers or registers in the I/O subsystem. We identify the names for the addresses of memory location as LOC, PLACE, A, VAR2 etc and the names for CPU registers as R0, R5 etc. The contents of a location or a register are denoted by placing the corresponding name between square brackets.

E.g. i) $R1 \leftarrow [LOC]$ means that the contents of memory location LOC are transferred into register R1.
 ii) $R3 \leftarrow [R1] + [R2]$ adds the contents of registers R1 and R2 and then places their sum into register R3.

- b) **Assembly Language Notation:-** The same operations can be represented in assembly language format as shown below.

E.g. i) Move LOC, R1
 ii) Add R1,R2,R3

2.4 BASIC INSTRUCTION TYPES***Three-address instruction***

$C = A + B$ is a high level instruction to add the values of the two variables A and B and to assign the sum to a third variable C. When this statement is compiled, each of these variables is assigned to a location in the memory. The contents of these locations represent the values of the three variables. Hence the above instruction requires the action

$$C \leftarrow [A] + [B]$$

To carry out this instruction, the contents of the memory locations A and B are fetched from the main memory and transferred into the processor – sum is computed – result is sent back to memory and stored in location C. The same action is performed by a single machine instruction (three address instruction)

Add A,B,C

Operands A and B are called the *source operands*, C is called the destination operand, and Add is the operation to be performed on the operands. The general format is

Operation Source1,Source2,Destination

If k bits are needed to specify the memory address of each operand, the encoded form of the above instruction must contain $3k$ bits for addressing purposes + the bits needed to denote the Add operation.

Two-address instruction

An alternative method is to use two address instruction of the form

Operation Source, Destination

E.g. Add A,B which perform the operation

$B \leftarrow [A] + [B]$ Here the sum is calculated and the result is stored in location B replacing the original contents of this location. i.e. operand B acts as source as well as

destination. In the later case (three address instruction) the contents of A and B were not destroyed. But here the contents of B are destroyed. This problem is solved by using another two-address instruction to copy the contents of one memory location into another location.

Now $C \leftarrow [A] + [B]$ is equivalent to

```
Move B,C
Add  A,C
```

Note: In all the above instructions, the source operands are specified first, followed by the destination. But there are many computers in which the order is reversed.

One-address instruction

Instead of mentioning the second operand, it is understood to be in a unique location. A processor register usually called the ***Accumulator*** is used for this purpose.

E.g. i) Add A means that the contents of the memory location A is added to the contents of accumulator and places the sum in the accumulator.

ii) Load A copies the contents of memory location A into accumulator

iii) Store A copies the contents of accumulator to the location A

Depending on the instruction, the operand may be source or destination.

Now the operation $C \leftarrow [A] + [B]$ can be performed by executing the following instructions

```
Load A
Add  B
Store C
```

The above mentioned instructions can also be handled by using general purpose registers. Let R_i represent a general purpose register

Move A, R_i	}	They are generalizations of Load, Store and Add instructions of the single accumulator case in which register R_i performs the functions of accumulator.
Move R_i ,A		
Add A, R_i		

When a processor has several general-purpose registers, then many instructions involve only operands that are in registers.

E.g.	Add R_i, R_j	}	In the first instruction, R_j acts as both source and destination. In the second instruction, R_i & R_j are source and R_k is the destination.
	Add R_i, R_j, R_k		

Advantages of using CPU registers:

- i) Data access from these registers is faster than that of main memory locations; because these registers are inside the processor.

- ii) Only few bits are needed to specify the register; because the number of registers is very less. For example, only 5 bits are needed to specify 32 registers.
- iii) Instructions where only register names are contained, will normally fit into one word of memory

Zero-address instruction

Here locations of all operands are defined implicitly. Such instructions are found in machines that store operands in a structure called a *pushdown stack*. A stack is a list of data elements, usually words or bytes, in which these elements can be added or removed through the top of the stack by following the LIFO (last-in-first-out) storage mechanism. A processor register called *stack pointer (SP)* is used to keep track of the address of the element at the top of the stack at any given time. The terms *push* and *pop* are used to describe placing a new item on the stack and removing the top item from the stack respectively.

2.5 INSTRUCTION EXECUTION AND STRAIGHT LINE SEQUENCING

2.5.1 STRAIGHT LINE SEQUENCING

Let us take the operation $c \leftarrow [A] + [B]$ Figure shows the program segment as it appears in the main memory of a computer that has a two-address instruction format and a number of general purpose CPU registers. The program is

Move A,R0	}	These three instructions are placed in successive memory locations starting at location <i>i</i> as shown
Add B,R0		
Move R0,C		

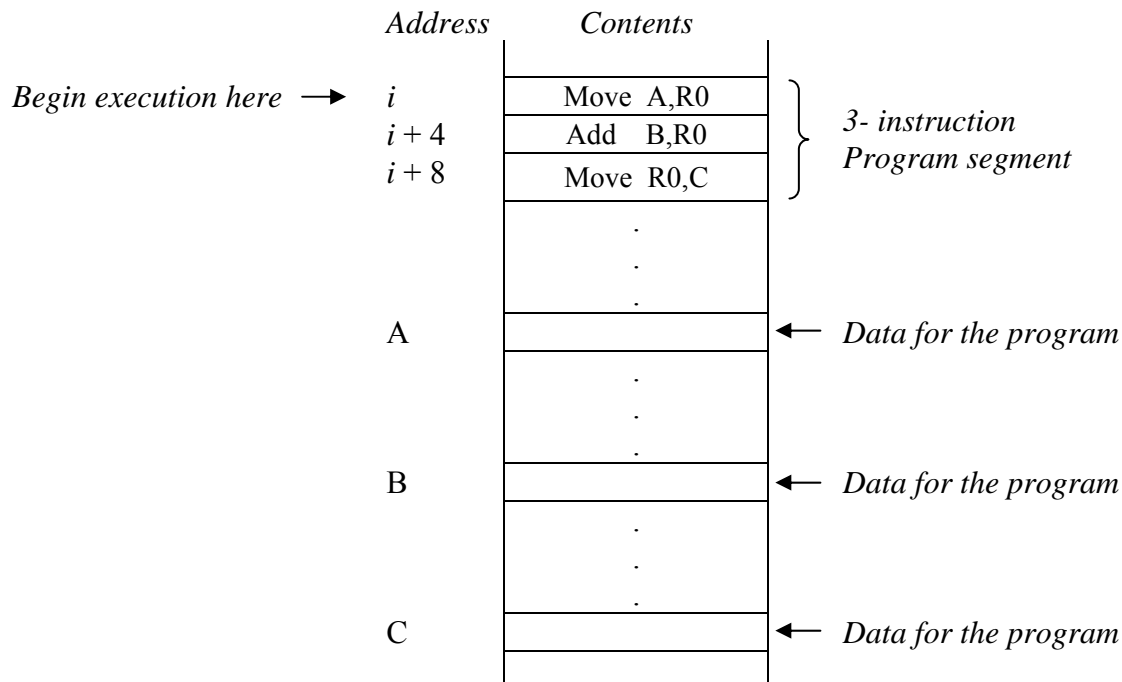


Fig 2.6 A program for $C \leftarrow [A] + [B]$

For executing this program, the following steps are to be performed.

1. CPU contains the register called PC which holds the address of the instruction to be executed next. To begin execution, the address of the first instruction 'i' must be placed in PC.

2. CPU control circuits use the information in the PC to fetch and execute the instructions one at a time in the increasing order of addresses. This is called straight line sequencing.
3. As each instruction is executed, the PC is incremented by 4 to point to the next instruction.

Executing a given instruction is a two-phase procedure.

Ist Phase - Instruction Fetch:- Instruction is fetched from the main memory location whose address is in the PC and is placed in the Instruction Register (IR)

IInd Phase - Instruction Execute:- Instruction in the IR is examined to determine which operation is to be performed. The specified operation is performed by the processor. This may involve fetching operands from main memory (or processor registers), performing an arithmetic or logic operation and storing the result in the destination location. At some point during this two-phase procedure, the contents of the PC are advanced to point to the next instruction. After the execution phase is over, new instruction fetch can begin.

2.5.2 BRANCHING

Consider the task of adding 'n' numbers. Let the address of memory locations containing n numbers are NUM1, NUM2, ... NUMn. Separate Add instruction is used to add each number to the contents of register R0. After all the numbers have been added, the result is placed in the memory location SUM.

<i>Address</i>	<i>Contents</i>
<i>i</i>	Move NUM1,R0
<i>i + 4</i>	Add NUM2,R0
<i>i + 8</i>	Add NUM3,R3
	.
	.
	.
<i>i + 4n - 4</i>	Add NUMn,R0
<i>i + 4n</i>	Move R0,SUM
	.
	.
	.
SUM	
NUM1	
NUM2	
	.
	.
	.
NUMn	

Fig 2.7 A straight-line program for adding *n* numbers

Instead of using long list of Add instruction, it is possible to place a single Ad instruction in a loop as shown in Fig 2.8 This loop causes a straight line sequence of instructions to be executed repeatedly. The loop starts at location LOOP and ends at the instruction Branch>0. During each

pass through this loop, the address of the next entry is determined and that entry is fetched and added to R0. Assume that the number of entries in the list 'n' is stored in location N as shown. Register R1 is used as a counter to determine the number of time the loop is to be executed. Hence the contents of the location N are loaded in register R1 at the beginning of the program. Then within the body of the loop the instruction

Decrement R1

reduces the contents of R1 by 1 each time through the loop. This means that execution of the loop must be repeated as long as the result of the decrement operation is greater than 0

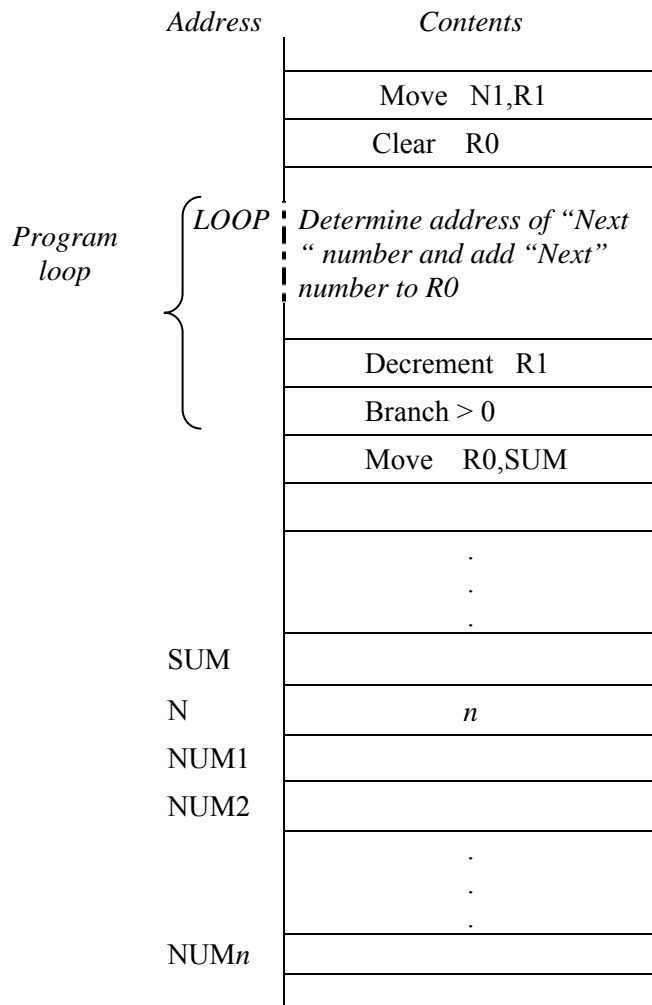


Fig 2.8 Using a loop to add *n* numbers

We now introduce Branch instruction. This type of instruction loads a new value into the program counter. As a result, the processor fetches and executes the instruction at this new address. A conditional branch instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way and the next instruction in sequential address order is fetched and executed.

2.5.3 CONDITION CODES

The processor keeps track of some information about the results of various operations for use by subsequent conditional branch instructions. This is done by recording the required information into

individual bits called as *condition code flags*. In some processors, these flags are grouped together in a special register called the *condition code register* or *status register*.

Four commonly used flags are:-

- N** (negative) Sets to 1 if the result is negative; otherwise, cleared to 0
- Z** (zero) Sets to 1 if the result is 0; otherwise, cleared to 0
- V** (overflow) Sets to 1 if arithmetic overflow occurs; otherwise, cleared to 0
- C** (carry) Sets to 1 if carry-out results from the operation; otherwise, cleared to 0

2.6 ADDRESSING MODES

The term addressing mode refers to the way in which the operand of an instruction is specified.

1. Register mode :- The operand is the contents of a CPU register; the name of register is given in the instruction

E.g. Move R1,R2 The contents of R1 is transferred to R2

2. Absolute mode (Direct mode) :- The operand is in a memory location. The address of the memory location is explicitly given in the instruction

E.g. Add A,B The contents of the memory location A is added to the contents of the memory location B. The addresses of A and B are given in the instruction itself.

3. Immediate mode :- The operand is given explicitly in the instruction. This mode is used in specifying address and data constants in programs

E.g. Move 200_{immediate},R0

This instruction places the value 200 in register R0.

The immediate mode is used to specify the value of a source operand. Using a subscript is not appropriate in assembly language. So we use a pound (#) sign in front of the value of the source operand to indicate that this value is to be used as an immediate operand.

E. g. Move #200,R0

4. Indirect mode :- Here the instruction does not give the operand or its address explicitly. Instead it provides the effective address of the operand. Effective address of the operand is the contents of a register or main memory location, whose address appears in the instruction. We denote indirection by placing the name of the register or the memory address given in the instruction in parenthesis

In the first case (Fig 2.9.a), when the instruction is executed, CPU starts by fetching the contents of location A in the main memory. Since indirect addressing is used, the value B stored in A is not the operand, but the address of the operand. Hence CPU requests another read operation from the main memory and this is to read the operands (contents of location B). The CPU then adds the operand to the contents of R0

In the second case (Fig 2.9.b), the operand is accessed indirectly through register R1 which contains the value B.

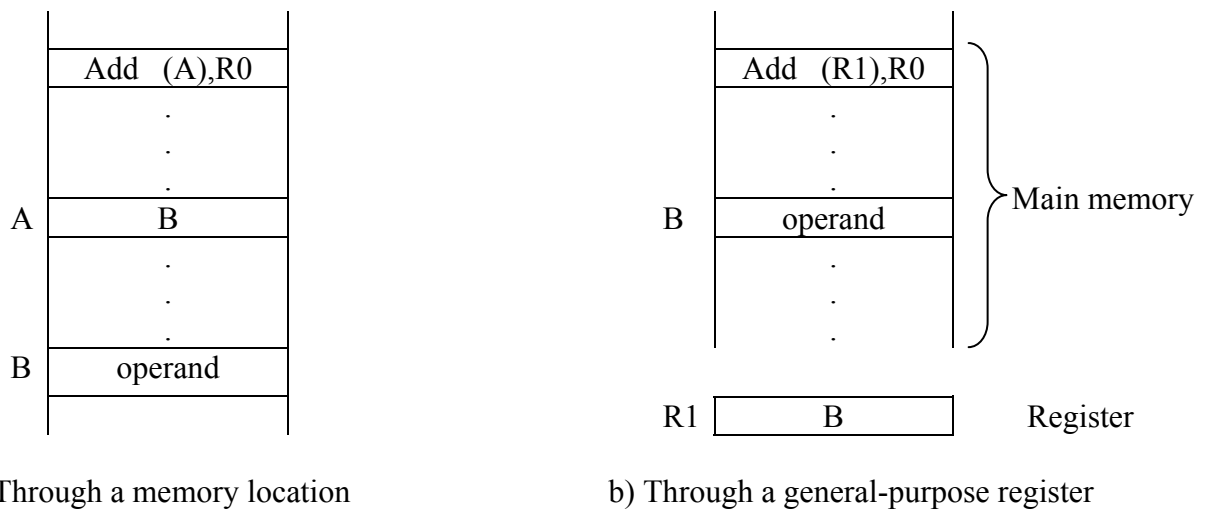


Fig 2.9 Indirect addressing

Note: The register or memory location that contains the address of the operand is called a pointer. Indirection is a powerful concept in programming.

5. Index mode :- In this mode, the effective address of the operand is generated by adding a constant value to the contents of a register. The register used may be a special register provided for this purpose or may be any one of the general purpose register – referred to as an **Index Register**. Index mode is indicated symbolically as $X(R_i)$, where X denotes a constant value contained in the instruction and R_i is the name of the register involved. The effective address of the operand is given by

$$EA \text{ or } A_{\text{eff}} = X + [R_i]$$

In assembly language program, the constant X may be given either as an explicit number or as a name representing a numerical value. Fig 2.10 illustrates the two ways of using the index mode.

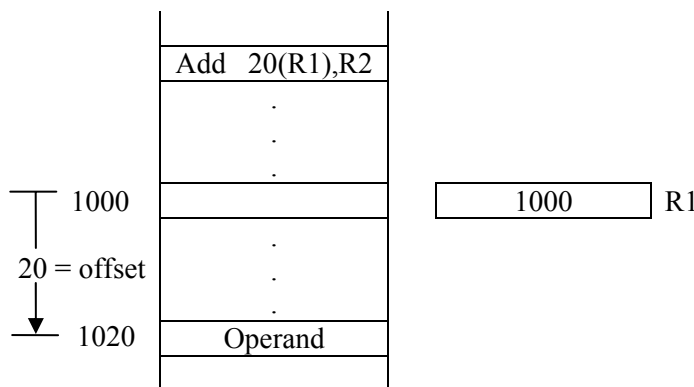


Fig 2.10 (a) Offset is given as a constant

In Fig 2.10(a) index register R1 contains the address of a memory location and the value X defines an offset called displacement.

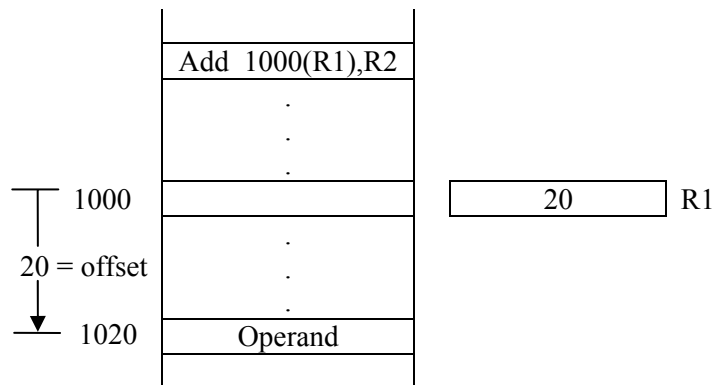


Fig 2.10 (b) Offset is in the index register

In Fig 2.10(b) the constant X corresponds to a memory address and the contents of the index register define the offset to the operand. In either case, the effective address is the sum of two values; one is given explicitly in the instruction and the other is in a register.

Several variations of index addressing mode exist.

- i) Contents of second register may be used as the index constant X ; in which case we can write the index mode as (R_i, R_j) . Here the effective address is the sum of the contents of register R_i and R_j . This provides more flexibility in accessing operands
- ii) The effective address is the sum of the constant X and the contents of registers R_i and R_j which is denoted as $X(R_i, R_j)$

5. Relative mode :- The effective address is determined by the index mode. But here the program counter is used in place of the general purpose register R_i . i.e. $X(PC)$ can be used to address a memory location that is X bytes away from the location presently pointed by the program counter. Since the addressed location is identified “relative” to the program counter, which always identifies the current execution point in a program, this mode is called as Relative mode.

- This mode is used to access data operands.
- Commonly used to specify the target address in branch instruction

6. Autoincrement mode :- The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of the register is automatically incremented to point to the next item in a list. We denote the autoincrement mode by putting the specified register in parenthesis to show that the contents of register is used as the effective address, followed by a plus (+) sign to indicate that these contents are to be incremented after the operand is accessed. Thus the autoincrement mode is written as

$$(R_i) +$$

If we use autoincrement mode, it is possible to eliminate the increment instruction

7. Autodecrement mode :- The contents of a register specified in the instruction are decremented. These contents are then used as the effective address of the operand. We denote the autodecrement mode by putting the specified register in the parenthesis, preceded by a minus (-) sign to indicate that the contents of the register are to be decremented before being used as the effective address. Thus we write

$$-(R_4)$$

This mode allows accessing of operands in the direction of descending address.