G. Pullaiah College of Engineering and Technology: Kurnool

Department Of Electronics and Communication Engineering

LECTURE NOTES ON

COMPUTER ORGANIZATION

DEPARTMENT OF ELCTRONICS AND COMMUNICATION ENGINEERING

G.PULLAIAH COLLEGE OF ENGINEERING AND TECHNOLOGY

Accredited by NAAC with 'A' Grade of UGC, Approved by AICTE, New Delhi
Permanently Affiliated to JNTUA, Ananthapuramu
(Recognized by UGC under 2(f) &12(B) & ISO 9001:2008 Certified
Institution) Nandikotkur Road, Kurnool-518452

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY ANANTAPUR

B. Tech II - II sem (C.S.E)	T	Tu	C
	3	1	3

(15A05402) COMPUTER ORGANIZATION

Course Objectives:

- To learn the fundamentals of computer organization and its relevance to classical and modern problems of computer design
- To make the students understand the structure and behavior of various functional modules of a computer.
- To understand the techniques that computers use to communicate with I/O devices
- To study the concepts of pipelining and the way it can speed up processing.
- To understand the basic characteristics of multiprocessors

Course Outcomes:

- Ability to use memory and I/O devices effectively
- Able to explore the hardware requirements for cache memory and virtual memory
- Ability to design algorithms to exploit pipelining and multiprocessors

Unit I:

Basic Structure of Computer: Computer Types, Functional Units, Basic operational Concepts, Bus Structure, Software, Performance, Multiprocessors and Multicomputer.

Machine Instructions and Programs: Numbers, Arithmetic Operations and Programs, Instructions and Instruction Sequencing, Addressing Modes, Basic Input/output Operations, Stacks and Queues, Subroutines, Additional Instructions.

Unit II:

Arithmetic: Addition and Subtraction of Signed Numbers, Design and Fast Adders, Multiplication of Positive Numbers, Signed-operand Multiplication, Fast Multiplication, Integer Division, Floating-Point Numbers and Operations.

Basic Processing Unit: Fundamental Concepts, Execution of a Complete Instruction, Multiple-Bus Organization, Hardwired Control, Multiprogrammed Control.

Unit III:

The Memory System: Basic Concepts, Semiconductor RAM Memories, Read-Only Memories, Speed, Size and Cost, Cache Memories, Performance Considerations, Virtual Memories, Memory Management Requirements, Secondary Storage.

Unit IV:

Input/output Organization: Accessing I/O Devices, Interrupts, Processor Examples, Direct Memory Access, Buses, Interface Circuits, Standard I/O Interfaces.

Unit V:

Pipelining: Basic Concepts, Data Hazards, Instruction Hazards, Influence on Instruction Sets.

Large Computer Systems: Forms of Parallel Processing, Array Processors, The Structure of General-Purpose, Interconnection Networks.

Textbook:

1) "Computer Organization", Carl Hamacher, Zvonko Vranesic, Safwat Zaky, McGraw Hill Education, 5th Edition, 2013**.**

Reference Textbooks:

1. Computer System Architecture, M.Morris Mano, Pearson Education, 3rd Edition.
2. Computer Organization and Architecture, Themes and Variations, Alan Clements, CENGAGE Learning.
3. Computer Organization and Architecture, Smruti Ranjan Sarangi, McGraw Hill Education.
4. Computer Architecture and Organization, John P.Hayes, McGraw Hill Education.

UNIT-1
BASIC STRUCTURE OF COMPUTERS

1.1 Computer types

A computer can be defined as a fast electronic calculating machine that accepts the (data) digitized input information process it as per the list of internally stored instructions and produces the resulting information.

List of instructions are called programs & internal storage is called computer memory.

The different types of computers are

1. **Personal computers: -** This is the most common type found in homes, schools, Business offices etc., It is the most common type of desk top computers with processing and storage units along with various input and output devices.
2. **Note book computers: -** These are compact and portable versions of PC
3. **Work stations: -** These have high resolution input/output (I/O) graphics capability, but with same dimensions as that of desktop computer. These are used in engineering applications of interactive design work.
4. **Enterprise systems: -** These are used for business data processing in medium to large corporations that require much more computing power and storage capacity than work stations. Internet associated with servers has become a dominant worldwide source of all types of information.
5. **Super computers: -** These are used for large scale numerical calculations required in the applications like weather forecasting etc.,

1.2 Functional unit

A computer consists of five functionally independent main parts input, memory, arithmetic logic unit (ALU), output and control unit.

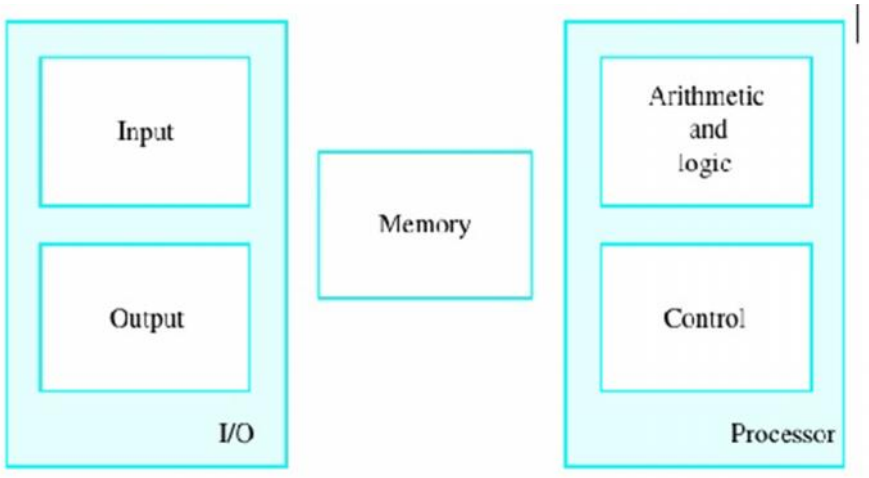


Fig : Functional units of computer

Input device accepts the coded information as source program i.e. high level language. This is either stored in the memory or immediately used by the processor to perform the desired operations. The program stored in the memory determines the processing steps. Basically the computer converts one source program to an object program. i.e. into machine language.

Finally the results are sent to the outside world through output device. All of these actions are coordinated by the control unit.

Input unit: -

The source program/high level language program/coded information/simply data is fed to a computer through input devices keyboard is a most common type. Whenever a key is pressed, one corresponding word or number is translated into its equivalent binary code over a cable & fed either to memory or processor.

Joysticks, trackballs, mouse, scanners etc are other input devices.

Memory unit: -

Its function into store programs and data. It is basically to two types

1. **Primary memory**
2. **Secondary memory**

1. Primary memory: - Is the one exclusively associated with the processor and operates at the electronics speeds programs must be stored in this memory while they are being executed. The memory contains a large number of semiconductors storage cells. Each capable of storing one bit of information. These are processed in a group of fixed site called word.

To provide easy access to a word in memory, a distinct address is associated with each word location. **Addresses are** numbers that identify memory location.

Number of bits in each word is called word length of the computer. Programs must reside in the memory during execution. Instructions and data can be written into the memory or read out under the control of processor.

Memory in which any location can be reached in a short and fixed amount of time after specifying its address is called random-access memory (RAM).

The time required to access one word in called memory access time. Memory which is only readable by the user and contents of which can't be altered is called read only memory (ROM) it contains operating system.

Caches are the small fast RAM units, which are coupled with the processor and are aften contained on the same IC chip to achieve high performance. Although primary storage is essential it tends to be expensive.

2 Secondary memory: - Is used where large amounts of data & programs have to be stored, particularly information that is accessed infrequently.

Examples: - Magnetic disks & tapes, optical disks (ie CD-ROM's), floppies etc.,

Arithmetic logic unit (ALU):-

Most of the computer operators are executed in ALU of the processor like addition, subtraction, division, multiplication, etc. the operands are brought into the ALU from memory and stored in high speed storage elements called register. Then according to the instructions the operation is performed in the required sequence.

The control and the ALU are may times faster than other devices connected to a computer system. This enables a single processor to control a number of external devices such as key boards, displays, magnetic and optical disks, sensors and other mechanical controllers.

Output unit:-

These actually are the counterparts of input unit. Its basic function is to send the processed results to the outside world.

Examples:- Printer, speakers, monitor etc.

Control unit:-

It effectively is the nerve center that sends signals to other units and senses their states. The actual timing signals that govern the transfer of data between input unit, processor, memory and output unit are generated by the control unit.

1.3 Basic operational concepts

To perform a given task an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be stored are also stored in the memory.

Examples: - Add LOCA, R_0

This instruction adds the operand at memory location LOCA, to operand in register R_0 & places the sum into register. This instruction requires the performance of several steps,

1. First the instruction is fetched from the memory into the processor.
2. The operand at LOCA is fetched and added to the contents of R_0
3. Finally the resulting sum is stored in the register R_0

The preceding add instruction combines a memory access operation with an ALU Operations. In some other type of computers, these two types of operations are performed by separate instructions for performance reasons.

Load LOCA, R1
Add R 1 , R0

Transfers between the memory and the processor are started by sending the address of the memory location to be accessed to the memory unit and issuing the appropriate control signals. The data are then transferred to or from the memory.

The fig shows how memory & the processor can be connected. In addition to the ALU & the control circuitry, the processor contains a number of registers used for several different purposes.

The instruction register (IR):- Holds the instructions that is currently being executed. Its output is available for the control circuits which generates the timing signals that control the various processing elements in one execution of instruction.

The program counterPC:-

This is another specialized register that keeps track of execution of a program. It contains the memory address of the next instruction to be fetched and executed.

Besides IR and PC, there are n-general purpose registers R0 through R_{n-1}.

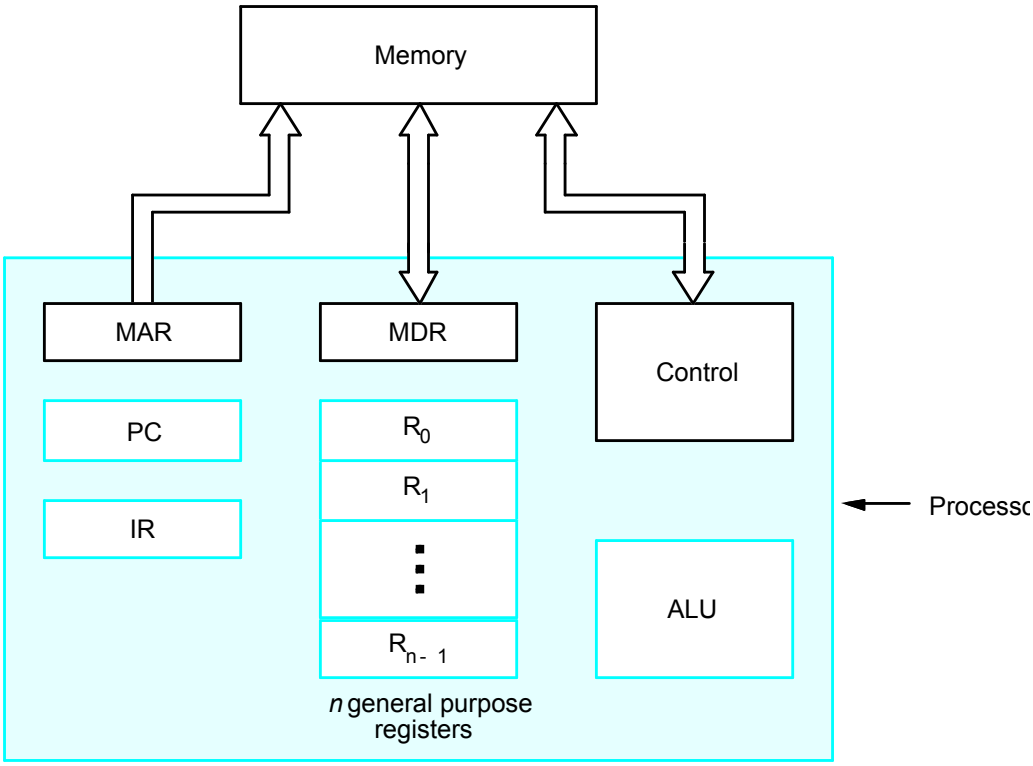


Figure :Connections between the processor and the memory.

The other two registers which facilitate communication with memory are: -

1. **MAR – (Memory Address Register):-** It holds the address of the location to be accessed.
2. **MDR – (Memory Data Register):-** It contains the data to be written into or read out of the address location.

Operating steps are

1. Programs reside in the memory & usually get these through the I/P unit.
2. Execution of the program starts when the PC is set to point at the first instruction of the program.
3. Contents of PC are transferred to MAR and a Read Control Signal is sent to the memory.
4. After the time required to access the memory elapses, the address word is read out of the mmory and loaded into the MDR.
5. Now contents of MDR are transferred to the IR & now the instruction is ready to be decoded and executed.
6. If the instruction involves an operation by the ALU, it is necessary to obtain the required operands.
7. An operand in the memory is fetched by sending its address to MAR & Initiating a read cycle.
8. When the operand has been read from the memory to the MDR, it is transferred from MDR to the ALU.
9. After one or two such repeated cycles, the ALU can perform the desired operation.
10. If the result of this operation is to be stored in the memory, the result is sent to MDR.
11. Address of location where the result is stored is sent to MAR & a write cycle is initiated.
12. The contents of PC are incremented so that PC points to the next instruction that is to be executed.

Normal execution of a program may be preempted (temporarily interrupted) if some devices require urgent servicing, to do this one device raises an Interrupt signal.

An interrupt is a request signal from an I/O device for service by the processor. The processor provides the requested service by executing an appropriate interrupt service routine.

The Diversion may change the internal stage of the processor its state must be saved in the memory location before interruption. When the interrupt-routine service is completed the state of the processor is restored so that the interrupted program may continue.

1.4 Bus structure

The simplest and most common way of interconnecting various parts of the computer. To achieve a reasonable speed of operation, a computer must be organized so that all its units can handle one full word of data at a given time.A group of lines that serve as a connecting port for several devices is called a bus.

In addition to the lines that carry the data, the bus must have lines for address and control purpose. Simplest way to interconnect is to use the single bus as shown

Fig: single bus structure

Since the bus can be used for only one transfer at a time, only two units can actively use the bus at any given time. Bus control lines are used to arbitrate multiple requests for use of one bus.

Single bus structure is

- Low cost
- Very flexible for attaching peripheral devices

Multiple bus structure certainly increases, the performance but also increases the cost significantly.

All the interconnected devices are not of same speed & time, leads to a bit of a problem. This is solved by using cache registers (ie buffer registers). These buffers are electronic registers of small capacity when compared to the main memory but of comparable speed.

The instructions from the processor at once are loaded into these buffers and then the complete transfer of data at a fast rate will take place.

1.5 Performance

The most important measure of the performance of a computer is how quickly it can execute programs. The speed with which a computer executes program is affected by the design of its hardware. For best performance, it is necessary to design the compiles, the machine instruction set, and the hardware in a coordinated way.

The total time required to execute the program is elapsed time is a measure of the performance of the entire computer system. It is affected by the speed of the processor, the disk and the printer. The time needed to execute a instruction is called the processor time.

Just as the elapsed time for the execution of a program depends on all units in a computer system, the processor time depends on the hardware involved in the execution of individual machine instructions. This hardware comprises the processor and the memory which are usually connected by the bus as shown in the fig c.

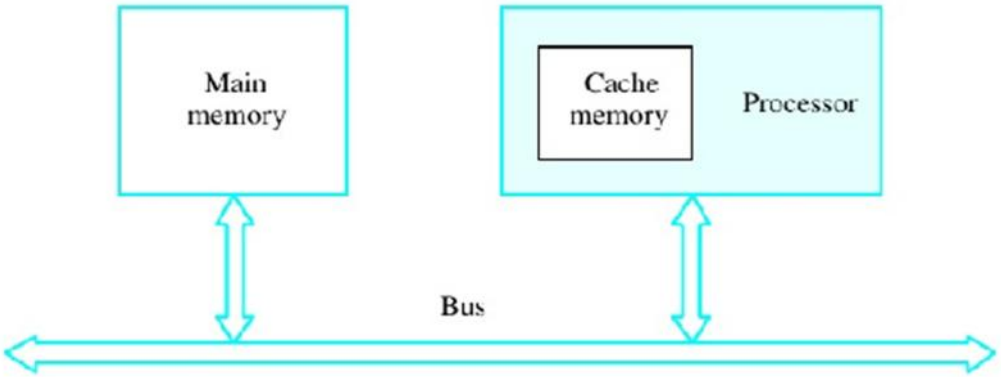


Fig : The processor cache

The pertinent parts of the fig. c are repeated in fig. d which includes the cache memory as part of the processor unit.

Let us examine the flow of program instructions and data between the memory and the processor. At the start of execution, all program instructions and the required data are stored in the main memory. As the execution proceeds, instructions are fetched one by one over the bus into the processor, and a copy is placed in the cache later if the same instruction or data item is needed a second time, it is read directly from the cache.

The processor and relatively small cache memory can be fabricated on a single IC chip. The internal speed of performing the basic steps of instruction processing on chip is very high and is considerably faster than the speed at which the instruction and data can be fetched from the main memory. A program will be executed faster if the movement of instructions and data between the main memory and the processor is minimized, which is achieved by using the cache.

For example:- Suppose a number of instructions are executed repeatedly over a short period of time as happens in a program loop. If these instructions are available in the cache, they can be fetched quickly during the period of repeated use. The same applies to the data that are used repeatedly.

Processor clock: -

Processor circuits are controlled by a timing signal called clock. The clock designer the regular time intervals called clock cycles. To execute a machine instruction the processor divides the action to be performed into a sequence of basic steps that each step can be completed in one clock cycle. The length P of one clock cycle is an important parameter that affects the processor performance.

Processor used in today's personal computer and work station have a clock rates that range from a few hundred million to over a billion cycles per second.

1.6 Basic performance equation

We now focus our attention on the processor time component of the total elapsed time. Let 'T' be the processor time required to execute a program that has been prepared in some high-level language. The compiler generates a machine language object program that corresponds to the source program. Assume that complete execution of the program requires the execution of N machine cycle language instructions. The number N is the actual number of instruction execution and is not necessarily equal to the number of machine cycle instructions in the object program. Some instruction may be executed more than once, which in the case for instructions inside a program loop others may not be executed all, depending on the input data used.

Suppose that the average number of basic steps needed to execute one machine cycle instruction is S, where each basic step is completed in one clock cycle. If clock rate is 'R' cycles per second, the program execution time is given by

$$T=N*S/R$$

this is often referred to as the basic performance equation.

We must emphasize that N, S & R are not independent parameters changing one may affect another. Introducing a new feature in the design of a processor will lead to improved performance only if the overall result is to reduce the value of T.

Pipelining and super scalar operation: -

We assume that instructions are executed one after the other. Hence the value of S is the total number of basic steps, or clock cycles, required to execute one instruction. A substantial improvement in performance can be achieved by overlapping the execution of successive instructions using a technique called pipelining.

Consider Add R_1 R_2 R_3

This adds the contents of R_1 & R_2 and places the sum into R_3.

The contents of R_1 & R_2 are first transferred to the inputs of ALU. After the addition operation is performed, the sum is transferred to R_3. The processor can read the next instruction from the memory, while the addition operation is being performed. Then of that instruction also uses, the ALU, its operand can be transferred to the ALU inputs at the same time that the add instructions is being transferred to R_3.

In the ideal case if all instructions are overlapped to the maximum degree possible the execution proceeds at the rate of one instruction completed in each clock cycle. Individual instructions still require several clock cycles to complete. But for the purpose of computing T, effective value of S is 1.

A higher degree of concurrency can be achieved if multiple instructions pipelines are implemented in the processor. This means that multiple functional units are used creating parallel paths through which different instructions can be executed in parallel with such an arrangement, it becomes possible to start the execution of several instructions in every clock cycle. This mode of operation is called superscalar execution. If it can be sustained for a long time during program execution the effective value of S can be reduced to less than one. But the parallel execution must preserve logical correctness of programs, that is the results produced must be same as those produced by the serial execution of program instructions. Now a days may processor are designed in this manner.

1.7 Clock rate

These are two possibilities for increasing the clock rate 'R'.

1. Improving the IC technology makes logical circuit faster, which reduces the time of execution of basic steps. This allows the clock period P, to be reduced and the clock rate R to be increased.
2. Reducing the amount of processing done in one basic step also makes it possible to reduce the clock period P. however if the actions that have to be performed by an instructions remain the same, the number of basic steps needed may increase.

Increase in the value 'R' that are entirely caused by improvements in IC technology affects all aspects of the processor's operation equally with the exception of the time it takes to access the main memory. In the presence of cache the percentage of accesses to the main memory is small. Hence much of the performance gain excepted from the use of faster technology can be realized.

Instruction set CISC & RISC:-

Simple instructions require a small number of basic steps to execute. Complex instructions involve a large number of steps. For a processor that has only simple instruction a large number of instructions may be needed to perform a given programming task. This could lead to a large value of 'N' and a small

value of 'S' on the other hand if individual instructions perform more complex operations, a fewer instructions will be needed, leading to a lower value of N and a larger value of S. It is not obvious if one choice is better than the other.

But complex instructions combined with pipelining (effective value of S= 1) would achieve one best performance. However, it is much easier to implement efficient pipelining in processors with simple instruction sets.

1.8 Performance measurements

It is very important to be able to access the performance of a computer, comp designers use performance estimates to evaluate the effectiveness of new features.

The previous argument suggests that the performance of a computer is given by the execution time T, for the program of interest.

Inspite of the performance equation being so simple, the evaluation of 'T' is highly complex. Moreover the parameters like the clock speed and various architectural features are not reliable indicators of the expected performance.

Hence measurement of computer performance using bench mark programs is done to make comparisons possible, standardized programs must be used.

The performance measure is the time taken by the computer to execute a given bench mark. Initially some attempts were made to create artificial programs that could be used as bench mark programs. But synthetic programs do not properly predict the performance obtained when real application programs are run.

A non profit organization called SPEC- system performance evaluation corporation selects and publishes bench marks.

The program selected range from game playing, compiler, and data base applications to numerically intensive programs in astrophysics and quantum chemistry. In each case, the program is compiled under test, and the running time on a real computer is measured. The same program is also compiled and run on one computer selected as reference.

The 'SPEC' rating is computed as follows.

$$\text{SPEC rating} = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$

If the SPEC rating = 50

Means that the computer under test is 50 times as fast as the ultra sparc 10. This is repeated for all the programs in the SPEC suit, and the geometric mean of the result is computed.

Let SPECi be the rating for program 'i' in the suite. The overall SPEC rating for the computer is given by

$$\text{SPEC rating} = \left(\prod_{i=1}^{n} SPECi \right)^{\frac{1}{n}}$$

Where 'n' = number of programs in suite.

Since actual execution time is measured the SPEC rating is a measure of the combined effect of all factors affecting performance, including the compiler, the OS, the processor, the memory of comp being tested.

Multiprocessor & microprocessors:-

- Large computers that contain a number of processor units are called multiprocessor system.
- These systems either execute a number of different application tasks in parallel or execute subtasks of a single large task in parallel
- All processors usually have access to all memory locations in such system & hence they are called shared memory multiprocessor systems.
- The high performance of these systems comes with much increased complexity and cost.
- In contrast to multiprocessor systems, it is also possible to use an interconnected group of complete computers to achieve high total computational power. These computers normally have access to their own memory units when the tasks they are executing need to communicate data they do so by exchanging messages over a communication network. This properly distinguishes them from shared memory multiprocessors, leading to name message-passing multi computer.

1.10 Number Representation

We obviously need to represent both positive and negative numbers. Three systems are used for representing such numbers :

- Sign-and-magnitude
- 1's-complement
- 2's-complement

In all three systems, the leftmost bit is 0 for positive numbers and 1 for negative numbers. Fig 2.1 illustrates all three representations using 4-bit numbers. Positive

values have identical representations in al systems, but negative values have different representations. In the sign-and-magnitude systems, negative values are represented by changing the most significant bit (b_3 in figure 2.1) from 0 to 1 in the B vector of the corresponding positive value. For example, +5 is represented by 0101, and -5 is represented by 1101. In 1's-complement representation, negative values are obtained by complementing each bit of the corresponding positive number. Thus, the representation for -3 is obtained by complementing each bit in the vector 0011 to yield 1100. clearly, the same operation, bit complementing, is done in converting a negative number to the corresponding positive value. Converting either way is referred to as forming the 1's-complement of a given number. Finally, in the 2's-complement system, forming the 2's-complement of a number is done by subtracting that number from 2^n.

Hence, the 2's complement of a number is obtained by adding 1 to the 1's complement ofthat number.

Addition of Positive numbers:-

Consider adding two 1-bit numbers. The results are shown in figure 2.2. Note that the sum of 1 and 1 requires the 2-bit vector 10 to represent the value 2. We say that the sum is 0 and the carry-out is 1. In order to add multiple-bit numbers, we use a method analogous to that used for manual computation with decimal numbers. We add bit pairs starting from the low-order (right) and of the bit vectors, propagating carries toward the
high-order (left) end.

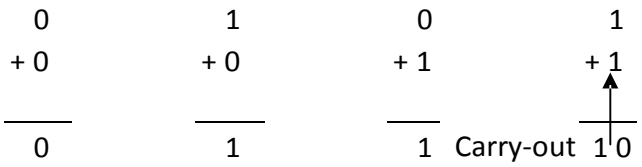


Figure 2.2 Addition of 1-bit numbers.

1.12 Memory locations and addresses

Number and character operands, as well as instructions, are stored in the memory of a computer. The memory consists of many millions of storage cells, each of which can store a bit of information having the value 0 or 1. Because a single bit represents a very small amount of information, bits are seldom handled individually. The usual approach is to deal with them in groups of fixed size. For this purpose, the memory is organized so that a group of n bits can be stored or retrieved in a

single, basic operation. Each group of n bits is referred to as a word of information, and n is called the word length. The memory of a computer can be schematically represented as a collection of words as shown in figure (a).

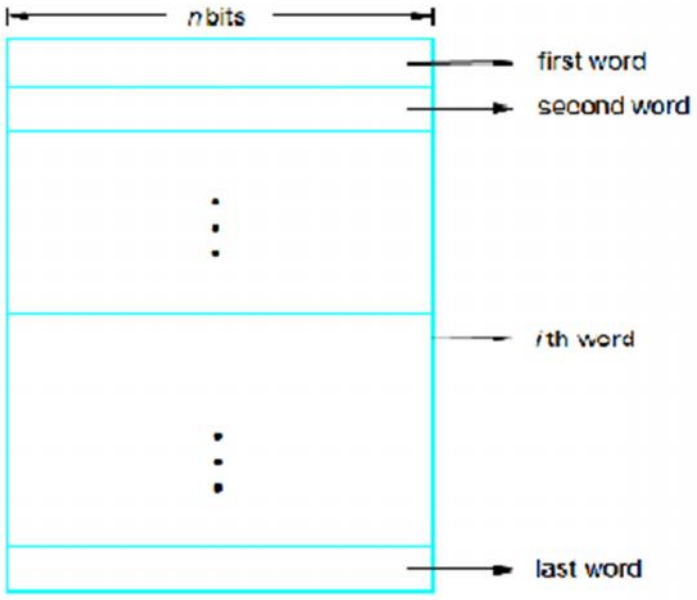
Modern computers have word lengths that typically range from 16 to 64 bits. If the word length of a computer is 32 bits, a single word can store a 32-bit 2's complement number or four ASCII characters, each occupying 8 bits. A unit of 8 bits is called a byte.

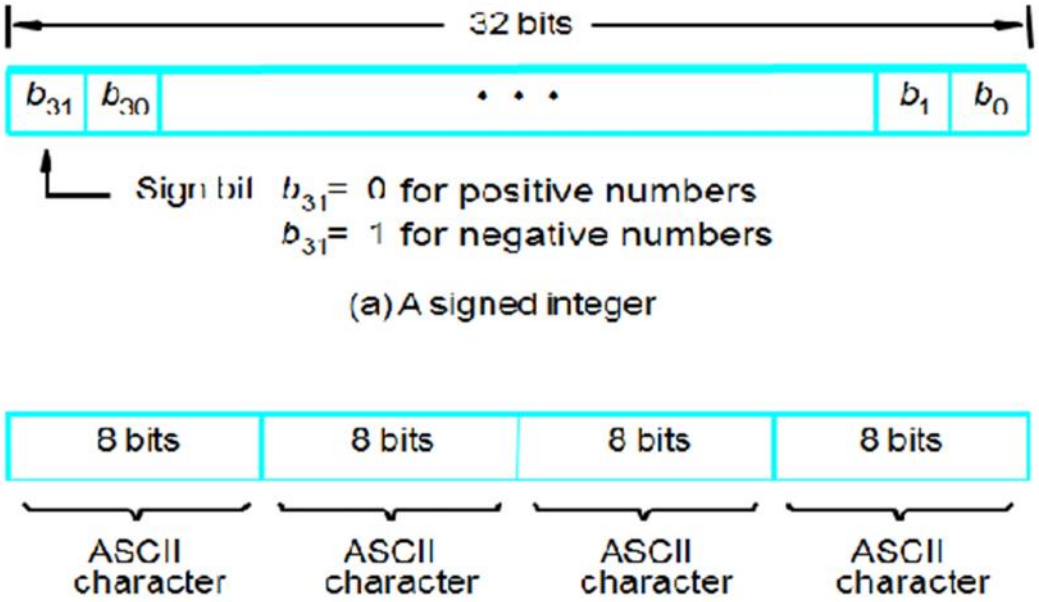
Accessing the memory to store or retrieve a single item of information, either a word or a byte, requires distinct names or addresses for each item location. It is customary to use numbers from 0 through 2^K-1, for some suitable values of k, as the addresses of successive locations in the memory. The 2^k addresses constitute the address space of the computer, and the memory can have up to 2^k addressable locations. 24-bit address generates an address space of 2^{24} (16,777,216) locations. A 32-bit address creates an address space of 2^{32} or 4G (4 giga) locations.

BYTE ADDRESSABILITY:-

We now have three basic information quantities to deal with: the bit, byte and word. A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits. The most practical assignment is to have successive addresses refer to successive byte

Fig a Memory words





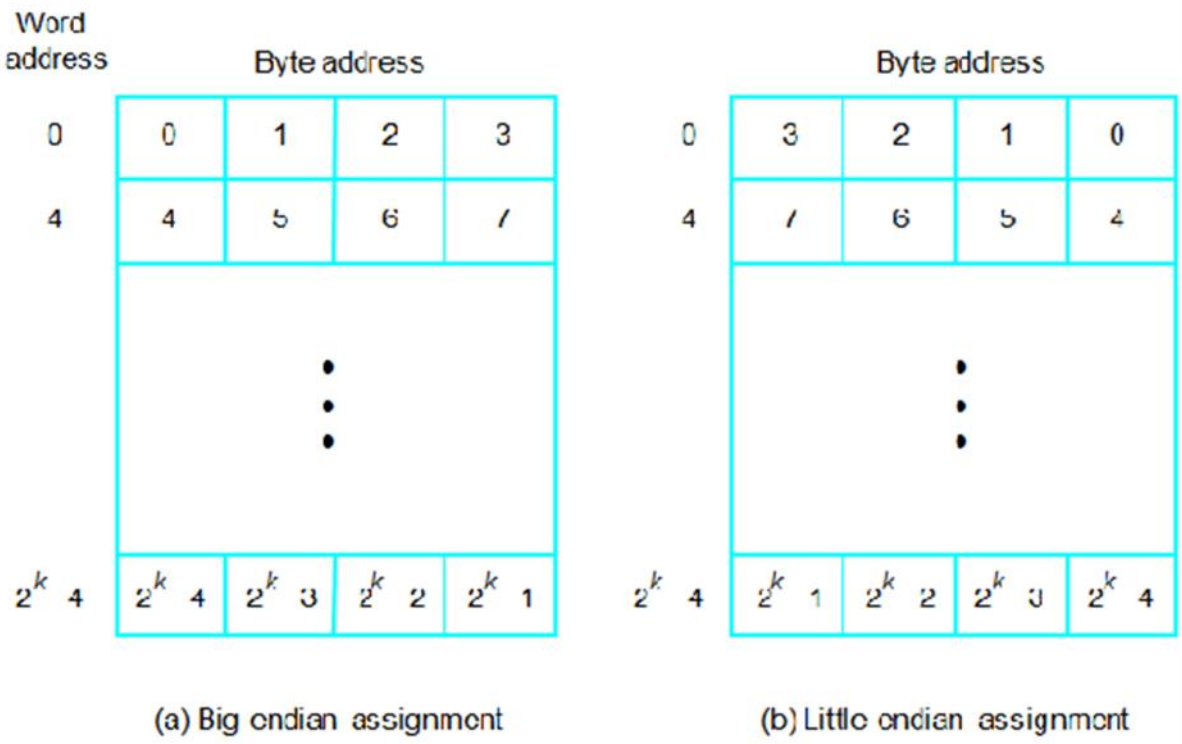
(b) Four characters

Locations in the memory. This is the assignment used in most modern computers, and is the one we will normally use in this book. The term byte-addressable memory is use for this assignment. Byte locations have addresses 0,1,2, …. Thus, if the word length of the machine is 32 bits, successive words are located at addresses 0,4,8,…., with each word consisting of four bytes.

BIG-ENDIAN AND LITTLE-ENDIAN ASIGNMENTS:-

There are two ways that byte addresses can be assigned across words, as shown in fig b. The name big-endian is used when lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word. The name little-endian is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (the rightmost bytes) of the word.

In addition to specifying the address ordering of bytes within a word, it is also necessary to specify the labeling of bits within a byte or a word. The same ordering is also used for labeling bits within a byte, that is, b_7, b_6, …., b_0, from left to right.



(a) Big endian assignment (b) Little endian assignment

Figure 2.7. Byte and word addressing.

WORD ALIGNMENT:-

In the case of a 32-bit word length, natural word boundaries occur at addresses 0,4, 8, …, as shown in above fig. We say that the word locations have aligned addresses . in general, words are said to be aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word. The memory of bytes in a word is a power of 2. Hence, if the word length is 16 (2 bytes), aligned words begin at byte addresses 0,2,4,…, and for a word length of 64 (23 bytes), aligned words begin at bytes addresses 0,8,16 ….

There is no fundamental reason why words cannot begin at an arbitrary byte address. In that case, words are said to have unaligned addresses. While the most common case is to use aligned addresses, some computers allow the use of unaligned word addresses.

ACCESSING NUMBERS, CHARACTERS, AND CHARACTER STRINGS:-

A number usually occupies one word. It can be accessed in the memory by specifying its word address. Similarly, individual characters can be accessed by their byte address.

In many applications, it is necessary to handle character strings of variable length. The beginning of the string is indicated by giving the address of the byte containing its first character. Successive byte locations contain successive

characters of the string. There are two ways to indicate the length of the string. A special control character with the meaning “end of string” can be used as the last character in the string, or a separate memory word location or processor register can contain a number indicating the length of the string in bytes.

1.14 Instructions and instruction sequencing

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

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

REGISTER TRANSFER NOTATION:-

Transfer of information from one location in the computer to another. Possible locations that may be involved in such transfers are memory locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem. Most of the time, we identify a location by a symbolic name standing for its hardware binary address.

Example, names for the addresses of memory locations may be LOC, PLACE, A, VAR2; processor registers names may be R0, R5; and I/O register names may be DATAIN, OUTSTATUS, and so on. The contents of a location are denoted by placing square brackets around the name of the location. Thus, the expression

R1<= [LOC]

Means that the contents of memory location LOC are transferred into processor registerR1.As another example, consider the operation that adds the contents of registers R1 and R2, and then places their sum into register R3. This action is indicated as

R3<= [R1] [R2]

This type of notation is known as Register Transfer Notation (RTN). Note that the right-hand side of an RTN expression always denotes a value, and the left-hand side is the name of a location where the value is to be places, overwriting the old contents of that location.

ASSEMBLY LANGUAGE NOTATION:-

Another type of notation to represent machine instructions and programs. For this, we use an assembly language format. For example, an instruction that causes the transfer described above, from memory location LOC to processor register R1, is specified by the statement

Move LOC, R1

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.

The second example of adding two numbers contained in processor registers R1 and R2 and placing their sum in R3 can be specified by the assembly language statement

Add R1, R2, R3

BASIC INSTRUCTIONS:-

The operation of adding two numbers is a fundamental capability in any computer. The statement

C = A + B

In a high-level language program is a command to the computer to add the current values of the two variables called A and B, and to assign the sum to a third variable, C. When the program containing this statement is compiled, the three variables, A, B, and C, are assigned to distinct locations in the memory. We will use the variable names to refer to the corresponding memory location addresses. The contents of these locations represent the values of the three variables. Hence, the above high-level language statement requires the action.

C <= [A] + [B]

To carry out this action, the contents of memory locations A and B are fetched from the memory and transferred into the processor where their sum is computed. This result is then sent back to the memory and stored in location C.

Let us first assume that this action is to be accomplished by a single machine instruction. Furthermore, assume that this instruction contains the memory addresses of the three operands – A, B, and C. This three-address instruction can be represented symbolically as

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. A general instruction of this type has the format.

Operation Source1, Source 2, Destination

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

An alternative approach is to use a sequence of simpler instructions to perform the same task, with each instruction having only one or two operands. Suppose that two- address instructions of the form

Operation Source, Destination

Are available. An Add instruction of this typeis

Add A, B

Which performs operation B <= [A] + [B].

A single two-address instruction cannot be used to solve our original problem, which is to add the the contents of locations A and B, without destroying either of them, and to place the sum in location C. The problem can be solved by using another two-address instruction that copies the contents of one memory location into another. Such an instruction is

Move B, C

Which performs the operations C< = [B], leaving the contents of location B unchanged.

Using only one-address instructions, the operation C< = [A] + [B] can be performed by executing the sequence of instructions

Load A

Add B

Store C

Some early computers were designed around a single accumulator structure. Most modern computers have a number of general-purpose processor registers – typically 8 to 32, and even considerably more in some cases. Access to data in these registers is much faster than to data stored in memory locations because the registers are inside the processor.

Let R_i represent a general-purpose register. The instructions

Load A, R_i

Store R_i, A and

Add A, R_i

Are generalizations of the Load, Store, and Add instructions for the single-accumulator case, in which register Ri performs the function of the accumulator.

When a processor has several general-purpose registers, many instructions involve only operands that are in the register. In fact, in many modern processors, computations can be performed directly only on data held in processor registers.Instructions such as

Add Ri, Rj

Or

Add Ri, Rj, Rk

In both of these instructions, the source operands are the contents of registers Ri and Rj. In the first instruction, Rj also serves as the destination register, whereas in the second instruction, a third register, Rk, is used as the destination.

It is often necessary to transfer data between different locations. This is achieved with the instruction

Move Source,
Destination

When data are moved to or from a processor register, the Move instruction can be used rather than the Load or Store instructions because the order of the source and destination operands determines which operation is intended.

In processors where arithmetic operations are allowed only on operands that are processor registers, the C = A + B task can be performed by the instruction sequence

Move A, Ri

Move B, R

Add Ri, Rj

Move Rj, C

In processors where one operand may be in the memory but the other must be in register, an instruction sequence for the required task would be

Move A, Ri
Add B, Ri
Move Ri, C

The speed with which a given task is carried out depends on the time it takes to transfer instructions from memory into the processor and to access the operands referenced by these instructions. Transfers that involve the memory are much slower than transfers within the processor.

We have discussed three-, two-, and one-address instructions. It is also possible

to use instructions in which the locations of all operands are defined implicitly. Such instructions are found in machines that store operands in a structure called a pushdown stack. In this case, the instructions are called zero-address instructions.

INSTRUCTION EXECUTION AND STRAIGHT-LINE SEQUENCING:-

In the preceding discussion of instruction formats, we used to task

C<=[A]+[B]

fig 2.8 shows a possible program segment for this task as it appears in the memory of a computer. We have assumed that the computer allows one memory operand

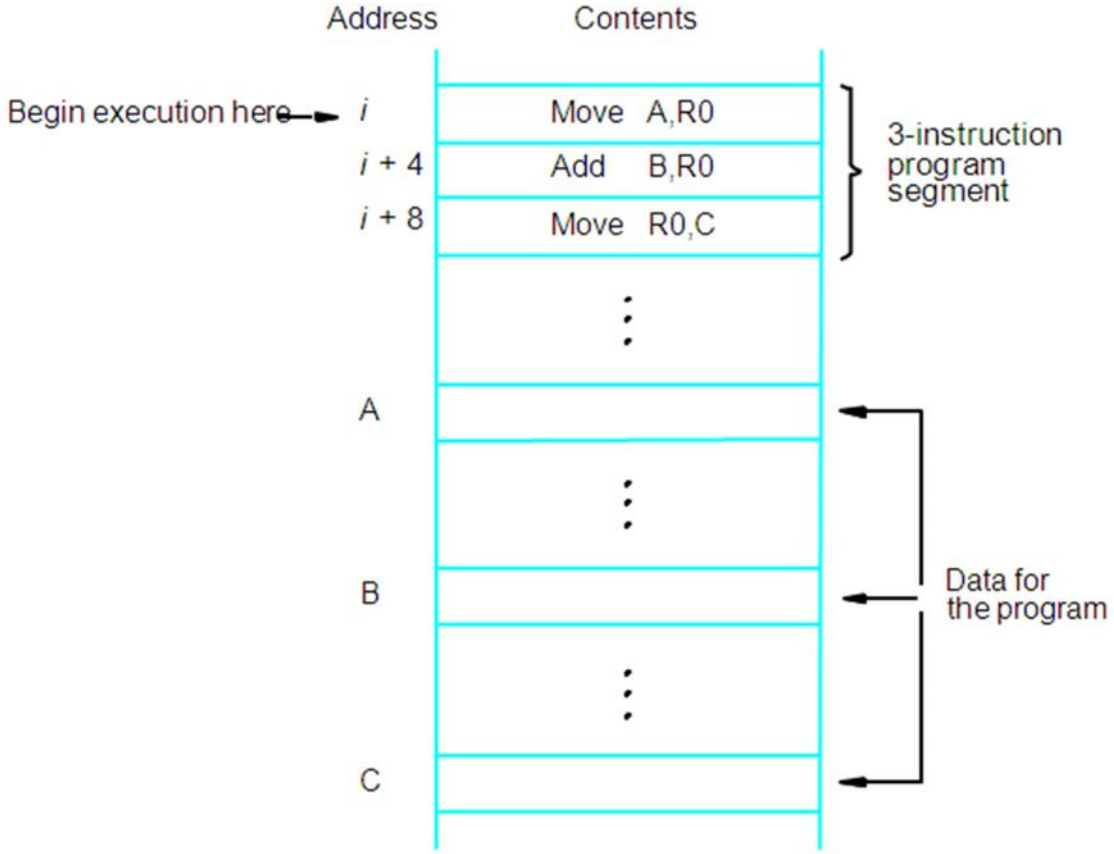


fig:2.8AProgram for C<=A+B

per instruction and has a number of processor registers. The three instructions of the program are in successive word locations, starting at location i. since each instruction is 4 bytes long, the second and third instructions start at addresses i + 4 and i8.

Let us consider how this program is executed. The processor contains a register called the program counter (PC), which holds the address of the instruction to be executed next. To begin executing a program, the address of its first instruction (I in our example) must be placed into the PC. Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called straight-line sequencing. During the execution of each instruction, the PC is incremented by 4 to point to the next instruction. Thus, after the Move instruction at location i + 8 is executed, the PC contains the value i + 12, which is the address of the first instruction of the next program segment.

Executing a given instruction is a two-phase procedure. In the first phase, called instruction fetch, the instruction is fetched from the memory location whose address is in the PC.

This instruction is placed in the instruction register (IR) in the processor. The instruction in IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor. This often involves fetching operands from the memory or from processor registers, performing an arithmetic or logic operation, and storing the result in the destination location.

BRANCHING:-

Consider the task of adding a list of n numbers. Instead of using a long list of add instructions, it is possible to place a single add instruction in a program loop, as shown in fig b. The loop is a straight-line sequence of instructions executed as many times as needed. It starts at location LOOP and ends at the instruction Branch > 0. During each pass through this loop, the address of the next list entry is determined, and that entry is fetched and added to

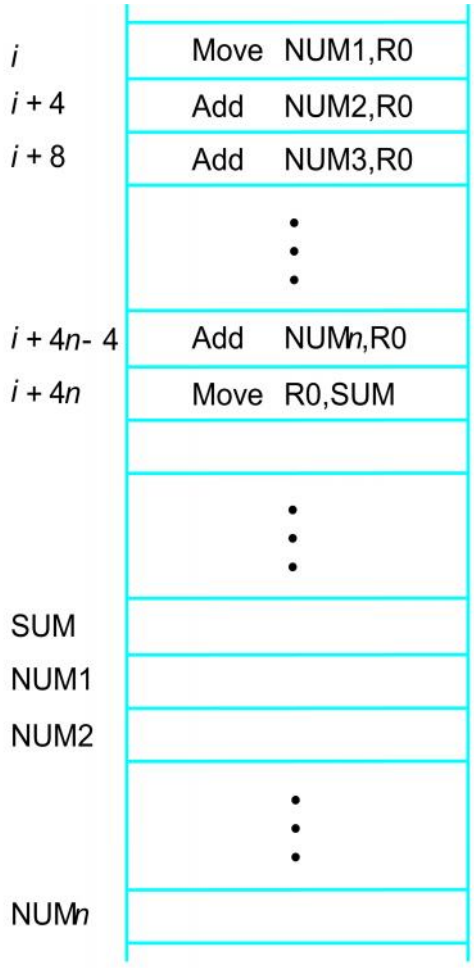


Figure 2.9. A straight-line program for adding *n* numbers.

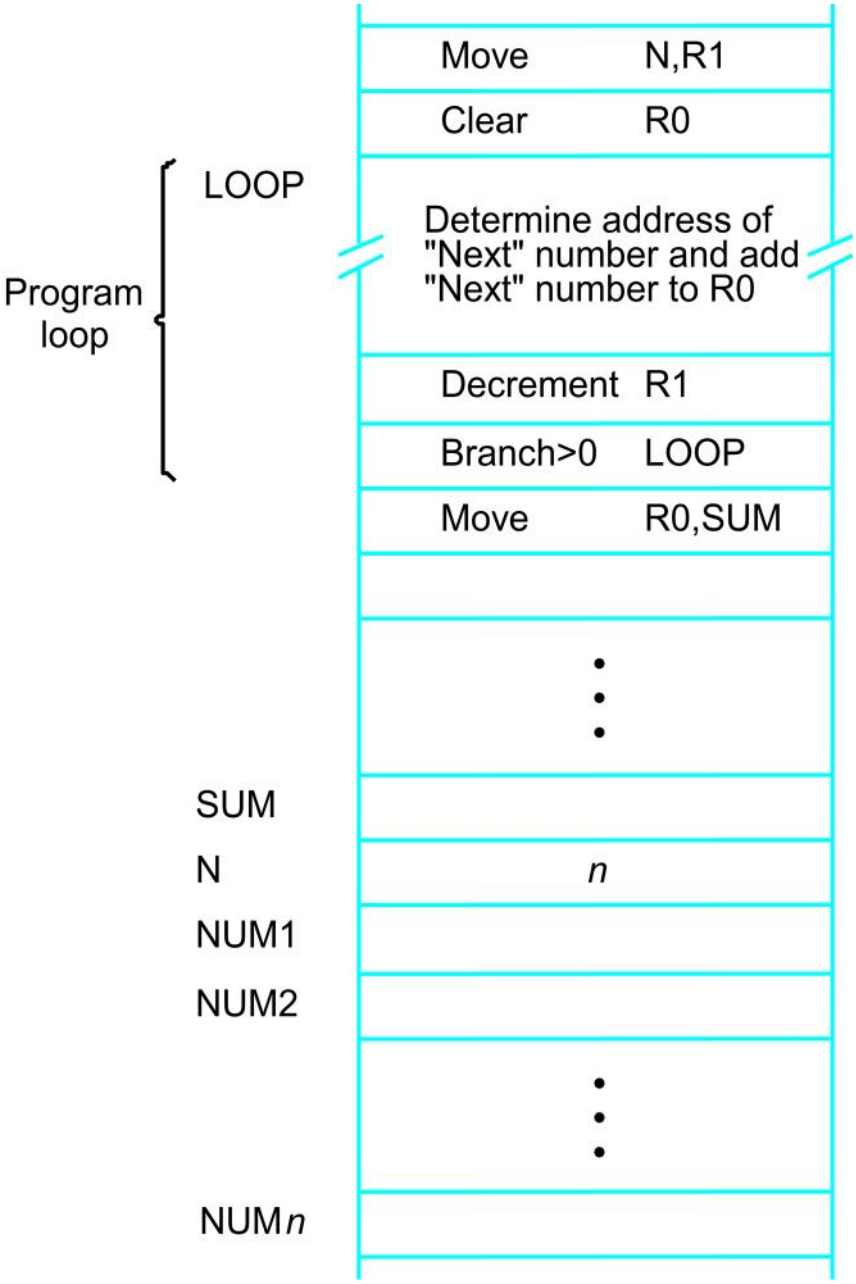


Fig b Using a loop to add n numbers

Assume that the number of entries in the list, n, is stored in memory location N, as shown. Register R1 is used as a counter to determine the number of time the loop is executed. Hence, the contents of location N are loaded into 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 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, called the branch target, instead of the instruction at the location that follows the branch instruction in sequential address order. 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.

Branch > 0 LOOP

(branch if greater that 0) is a conditional branch instruction that causes a branch to location LOOP if the result of the immediately preceding instruction, which is the decremented value in register R1, is greater that zero. This means that the loop is repeated, as long as there are entries in the list that are yet to be added to R0. at the end of the nth pass through the loop, the Decrement instruction produces a value of zero, and hence, branching does not occur.

CONDITION CODES:-

The processor keeps track of information about the results of various operations for use by subsequent conditional branch instructions. This is accomplished by recording the required information in individual bits, often called condition code flags. These flags are usually grouped together in a special processor register called the condition code register or status register. Individual condition code flags are set to 1 or cleared to 0, depending on the outcome of the operation performed.

Four commonly used flags are

N(negative) Set to 1 if the result is negative; otherwise, cleared to 0

Z(zero) Set to 1 if the result is 0; otherwise, cleared to 0

V(overflow) Set ot1 if arithmetic overflow occurs; otherwise, cleared to 0

C(carry) Set to 1 if a carry-out results from the operation; otherwise, cleared to 0

The instruction Branch > 0, discussed in the previous section, is an example of a branch instruction that tests one or more of the condition flags. It causes a branch if the value tested is neither negative nor equal to zero. That is, the branch is taken if neither N nor Z is 1. The conditions are given as logic expressions involving the condition code flags.

In some computers, the condition code flags are affected automatically by instructions that perform arithmetic or logic operations. However, this is not

always the case. A number of computers have two versions of an Add instruction.

GENERATING MEMORY ADDRESSES:-

Let us return to fig b. The purpose of the instruction block at LOOP is to add a different number from the list during each pass through the loop. Hence, the Add instruction in the block must refer to a different address during each pass. How are the addresses to be specified ? The memory operand address cannot be given directly in a single Add instruction in the loop. Otherwise, it would need to be modified on each pass through the loop.

The instruction set of a computer typically provides a number of such methods, called addressing modes. While the details differ from one computer to another, the underlying concepts are the same.

1.15 MACHINE INSTRUCTIONS AND PROGRAMS

1.15.1 Addressing modes:

In general, a program operates on data that reside in the computer's memory. These data can be organized in a variety of ways. If we want to keep track of students' names, we can write them in a list. Programmers use organizations called data structures to represent the data used in computations. These include lists, linked lists, arrays, queues, and so on.

Programs are normally written in a high-level language, which enables the programmer to use constants, local and global variables, pointers, and arrays. The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes.

Name	Assembler syntax	Addressing function
Immediate	# Value	Operand = Value
Register	Ri	EA = Ri
Absolute (Direct)	LOC	EA = LOC
Indirect	(Ri)	EA = [Ri]
	(LOC)	EA = [LOC]
Index	X(Ri)	EA = [Ri] + X
Base with index	(Ri, Rj)	EA = [Ri] + [Rj]
Base with index and offset	X (Ri, Rj)	EA = [Ri] + [Rj] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(Ri)+	EA = [Ri]; Increment Ri
Autodecrement	-(Ri)	Decrement Ri; EA = [Ri]

Table 2.1 Generic addressing modes

EA = effective address

Value = a signed number

IMPLEMENTATION OF VARIABLE AND CONSTANTS:-

Variables and constants are the simplest data types and are found in almost every computer program. In assembly language, a variable is represented by allocating a register or memory location to hold its value. Thus, the value can be changed as needed using appropriate instructions.

Register mode - The operand is the contents of a processor register; the name (address)of the register is given in the instruction.

Absolute mode – The operand is in a memory location; the address of this location is given explicitly in the instruction. (In some assembly languages, this mode is called Direct)

The instruction

Move LOC, R2

Processor registers are used as temporary storage locations where the data is a register are accessed using the Register mode. The Absolute mode can represent global variables in a program. A declaration such as

Integer A, B;

Immediate mode – The operand is given explicitly in the instruction.

For example, the instruction

Move $200_{immediate}$, R0

Places the value 200 in register R0. Clearly, the Immediate mode is only used to specify the value of a source operand. Using a subscript to denote the Immediate mode is not appropriate in assembly languages. A common convention is to use the sharp sign (#) in front of the value to indicate that this value is to be used as an immediate operand. Hence, we write the instruction above in the form

Move #200, R0

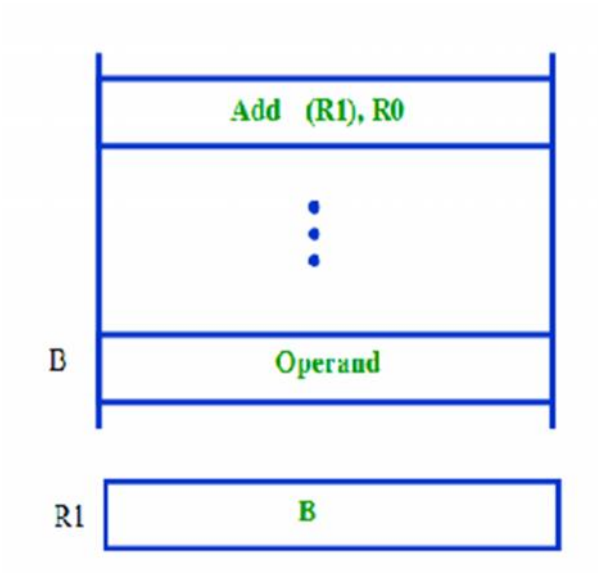
INDIRECTION AND POINTERS:-

In the addressing modes that follow, the instruction does not give the operand or its address explicitly, Instead, it provides information from which the memory address of the operand can be determined. We refer to this address as the effective address (EA) of the operand.

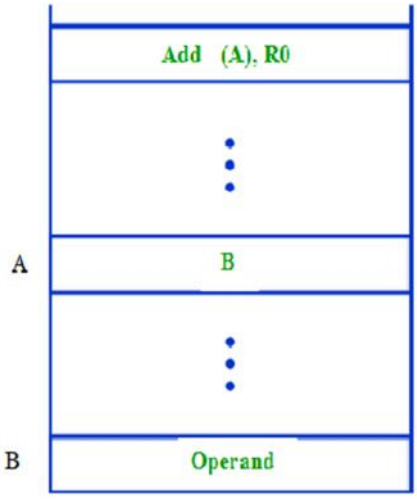
Indirect mode – The effective address of the operand is the contents of a register or memory location whose address appears in the instruction.

To execute the Add instruction in fig (a), the processor uses the value which is in register R1, as the effective address of the operand. It requests a read operation from the memory to read the contents of location B. the value read is the desired operand, which the processor adds to the contents of register R0. Indirect addressing through a memory location is also possible as shown in fig (b). In this case, the processor first reads the contents of memory location A, then requests a second read operation using the value B as an address to obtain the operand

Fig (a) Through a general-purpose register



(b) Through a memory location



```
        Move        N,R1
        Move        #NUM, R2
        Clear       R0
LOOP    ADD         (R2), R0
        ADD         #4, R2
        DECREMENT   R1
        Branch > 0  LOOP
        Move        R0, SUM
```

Fig : use of indirect addressing in the program

The register or memory location that contains the address of an operand is called a pointer. Indirection and the use of pointers are important and powerful concepts in programming.

In the program shown Register R2 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R2. The initialization section of the program loads the counter value n from memory location N into R1 and uses the immediate addressing mode to place the address value NUM1, which is the address of the first number in the list, into R2. Then it clears R0 to 0. The first two instructions in the loop implement the unspecified instruction block starting at LOOP. The first time through the loop, the instruction **Add (R2), R0** fetches the operand at location NUM1 and adds it to R0. The second Add instruction adds 4 to the contents of the pointer R2, so that it will contain the address value NUM2 when the above instruction is executed in the second pass through the loop.

Where B is a pointer variable. This statement may be compiled into

```
Move  B, R1
Move  (R1), A
```

Using indirect addressing through memory, the same action can be achieved with

```
Move  (B), A
```

Indirect addressing through registers is used extensively. The above program shows the flexibility it provides. Also, when absolute addressing is not available, indirect addressing through registers makes it possible to access global variables by first loading the operand's address in a register.

INDEXING AND ARRAYS:-

A different kind of flexibility for accessing operands is useful in dealing with lists and arrays.

Index mode – the effective address of the operand is generated by adding a constant value to the contents of a register.

The register use may be either a special register provided for this purpose, or, more commonly, it may be any one of a set of general-purpose registers in the processor. In either case, it is referred to as index register. We indicate the Index mode symbolically
as

X (Ri)

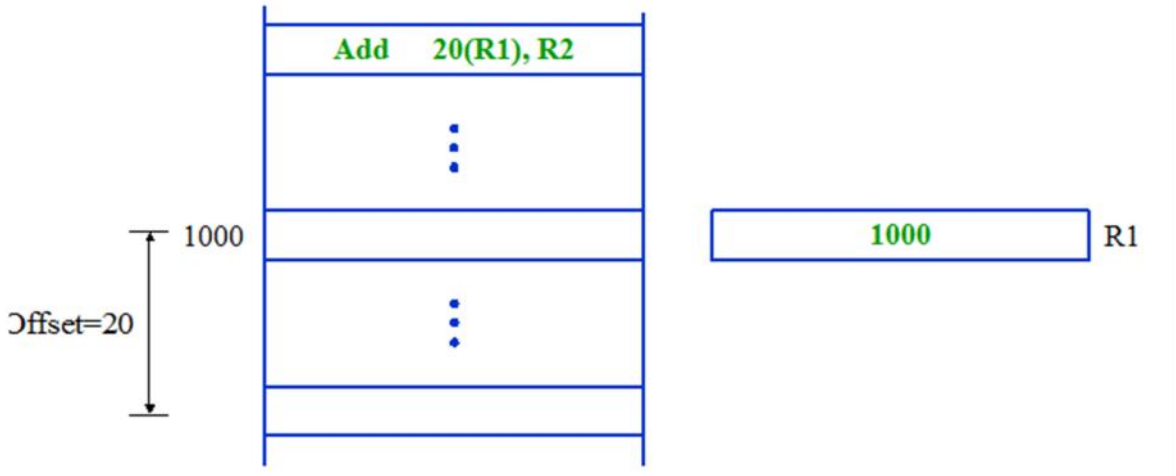
Where X denotes the constant value contained in the instruction and Ri is the name of the register involved. The effective address of the operand is given by

EA = X + [Rj]

The contents of the index register are not changed in the process of generating the effective address. In an assembly language program, the constant X may be given either as an explicit number or as a symbolic name representing a numerical value.

Fig a illustrates two ways of using the Index mode. In fig a, the index register, R1, contains the address of a memory location, and the value X defines an offset (also called a displacement) from this address to the location where the operand is found. An alternative use is illustrated in fig b. Here, 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 stored in a register.

Fig (a) Offset is given as a constant



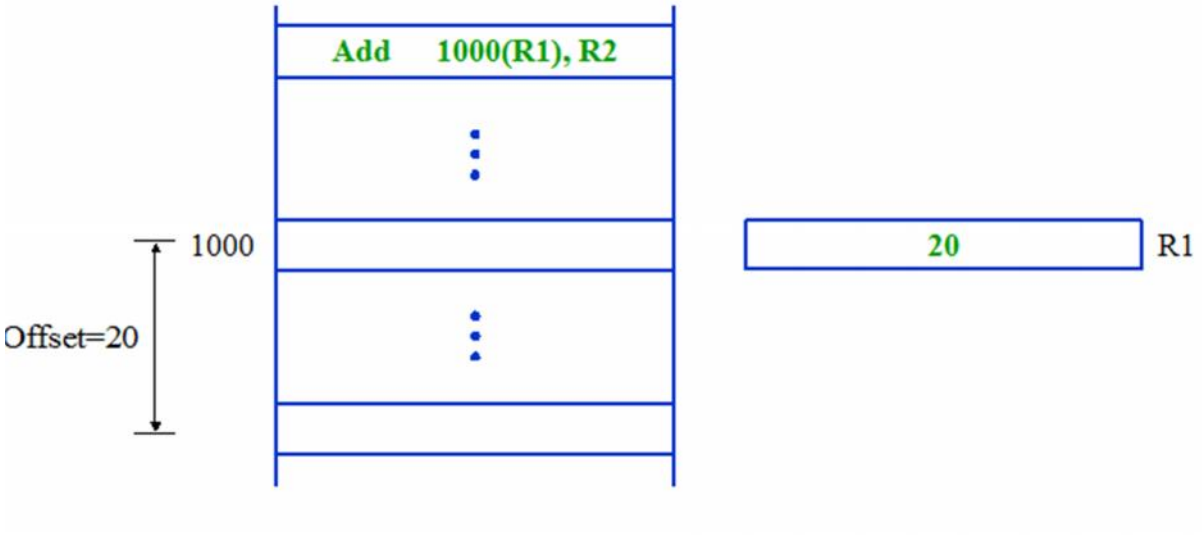


Fig (b) Offset is in the index register

In the most basic form of indexed addressing several variations of this basic form provide a very efficient access to memory operands in practical programming situations. For example, a second register may be used to contain the offset X, in which case we can write the Index mode as

(Ri, Rj)

The effective address is the sum of the contents of registers Ri and Rj. The second register is usually called the base register. This form of indexed addressing provides more flexibility in accessing operands, because both components of the effective address can be changed.

Another version of the Index mode uses two registers plus a constant, which can be denoted as

X(Ri, Rj)

In this case, the effective address is the sum of the constant X and the contents of registers Ri and Rj. This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (Ri, Rj) part of the addressing mode. In other words, this mode implements a three-dimensional array.

RELATIVE ADDRESSING:-

We have defined the Index mode using general-purpose processor registers. A useful version of this mode is obtained if the program counter, PC, is used instead of a general purpose register. Then, X(PC) can be used to address a memory location that is X bytes away from the location presently pointed to by the program counter.

Relative mode – The effective address is determined by the Index mode using the program counter in place of the general-purpose register Ri.

This mode can be used to access data operands. But, its most common use is to specify the target address in branch instructions. An instruction such as

Branch > 0 LOOP

Causes program execution to go to the branch target location identified by the name LOOP if the branch condition is satisfied. This location can be computed by specifying it as an offset from the current value of the program counter. Since the branch target may be either before or after the branch instruction, the offset is given as a signed number.

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 this register are automatically to point to the next item in a list.

(Ri)+

Autodecrement mode – the contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand.

-(Ri)

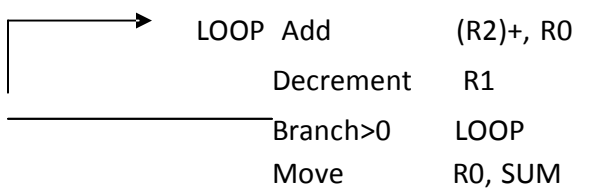


Fig c The Autoincrement addressing mode used in the program

1.16 Basic input/output operations

We now examine the means by which data are transferred between the memory of a computer and the outside world. Input/Output (I/O) operations are essential, and the way they are performed can have a significant effect on the performance of the computer.

Consider a task that reads in character input from a keyboard and produces character output on a display screen. A simple way of performing such I/O tasks is to use a method known as program-controlled I/O. The rate of data transfer from the keyboard to a computer is limited by the typing speed of the user, which is unlikely to exceed a few characters per second. The rate of output transfers from the computer to the display is much higher. It is determined by the rate at which characters can be transmitted over the link between the computer and the display device, typically several thousand characters per second. However, this is still much slower than the speed of a processor that can execute many millions of instructions per second. The difference in speed between the processor and I/O devices creates the need for mechanisms to synchronize the transfer of data between them.

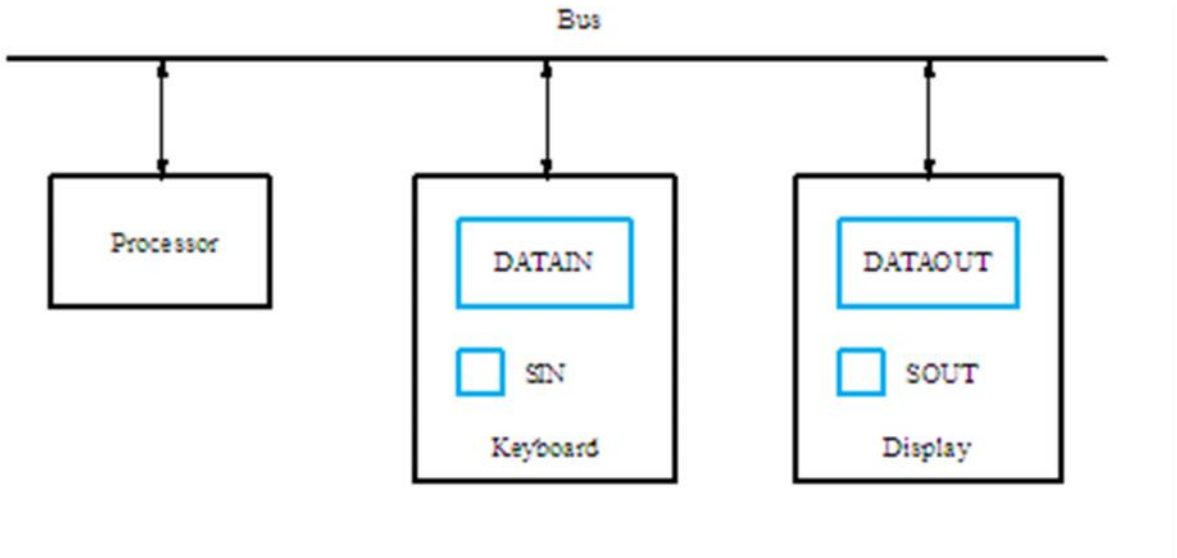


Fig a Bus connection for processor, keyboard, and display

The keyboard and the display are separate device as shown in fig a. the action of striking a key on the keyboard does not automatically cause the corresponding character to be displayed on the screen. One block of instructions in the I/O program transfers the character into the processor, and another associated block of instructions causes the character to be displayed.

Striking a key stores the corresponding character code in an 8-bit buffer register associated with the keyboard. Let us call this register DATAIN, as shown in fig a. To inform the processor that a valid character is in DATAIN, a status control flag, SIN, is set to 1. A program monitors SIN, and when SIN is set to 1, the processor reads the contents of DATAIN. When the character is transferred to the processor, SIN is automatically cleared to 0. If a second character is entered at the keyboard, SIN is again set to 1, and the processor repeats.

An analogous process takes place when characters are transferred from the processor to the display. A buffer register, DATAOUT, and a status control flag, SOUT, are used for this transfer. When SOUT equals 1, the display is ready to receive a character.

In order to perform I/O transfers, we need machine instructions that can check the state of the status flags and transfer data between the processor and the I/O device. These instructions are similar in format to those used for moving data between the processor and the memory. For example, the processor can monitor the keyboard status flag SIN and transfer a character from DATAIN to register R1 by the following sequence of operations.

1.17 Stacks and queues

A computer program often needs to perform a particular subtask using the familiar subroutine structure. In order to organize the control and information linkage between the main program and the subroutine, a data structure called a stack is used. This section will describe stacks, as well as a closely related data structure called a queue.

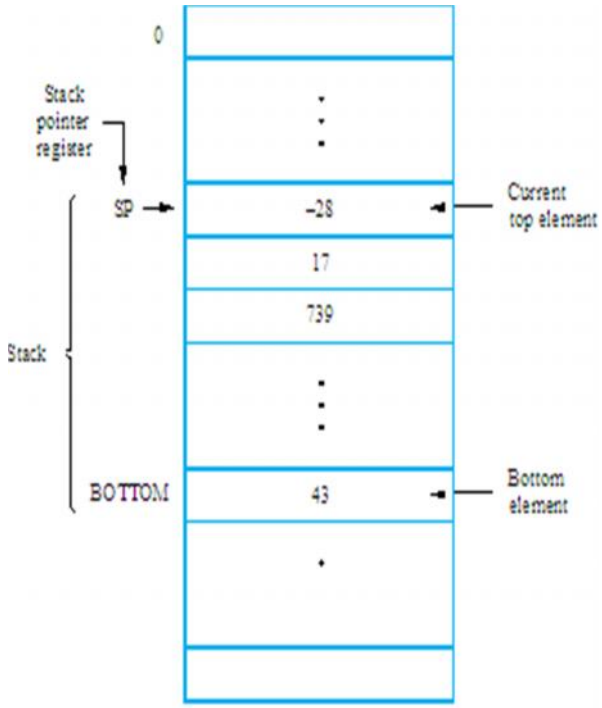
Data operated on by a program can be organized in a variety of ways. We have already encountered data structured as lists. Now, we consider an important data structure known as a stack. A stack is a list of data elements, usually words or bytes, with the accessing restriction that elements can be added or removed at one end of the list only.

This end is called the top of the stack, and the other end is called the bottom. Another descriptive phrase, last-in-first-out (LIFO) stack, is also used to describe this type of storage mechanism; the last data item placed on the stack is the first one removed when retrieval begins. 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

Fig b shows a stack of word data items in the memory ocomputer. It contains numerical values, with 43 at the bottom and -28 at the top

A processor register is used to keep track of the address of the element of the stack that is at the top at any given time. This register is called the stack pointer (SP). It could be one of the general-purpose registers or a register dedicated to this function.

Fig b A stack of words in the memory



Another useful data structure that is similar to the stack is called a queue. Data are stored in and retrieved from a queue on a first-in-first-out (FIFO) basis. Thus, if we assume that the queue grows in the direction of increasing addresses in the memory, which is a common practice, new data are added at the back (high-address end) and retrieved from the front (low-address end) of the queue.

There are two important differences between how a stack and a queue are implemented. One end of the stack is fixed (the bottom), while the other end rises and falls as data are pushed and popped. A single pointer is needed to point to the top of the gher addresses as data are added at the back and removed from the front. So two pointers are needed to keep track of the two ends of the queue.

Another difference between a stack and a queue is that, without further control, a queue would continuously move through the memory of a computer in the direction of higher addresses. One way to limit the queue to a fixed region in memory is to use a circular buffer. Let us assume that memory addresses from BEGINNING to END are assigned to the queue.

The first entry in the queue is entered into location BEGINNING, and successive entries are appended to the queue by entering them at successively higher addresses. By the time the back of the queue reaches END, space will have been created at the beginning if some items have been removed from the queue. Hence, the back pointer is reset to the value BEGINNING and the process continues. As in the case of a stack, care must be taken to detect when the region assigned to the data structure is either completely full or completely empty.

1.18 Subroutines

In a given program, it is often necessary to perform a particular subtask many times on different data-values. Such a subtask is usually called a subroutine. For example, a subroutine may evaluate the sine function or sort a list of values into increasing or decreasing order.

It is possible to include the block of instructions that constitute a subroutine at every place where it is needed in the program. However, to save space, only one copy of the instructions that constitute the subroutine is placed in the memory, and any program that requires the use of the subroutine simply branches to its starting location. When a program branches to a subroutine we say that it is calling the subroutine.

The instruction that performs this branch operation is named a Call instruction.

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

The way in which a computer makes it possible to call and return from subroutines is referred to as its subroutine linkage method. The simplest subroutine linkage method is to save the return address in a specific location, which may be a register dedicated to this function. Such a register is called the link register. When the subroutine completes its task, the Return instruction returns to the calling program by branching indirectly through the link register.

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

- Store the contents of the PC in the link register
- Branch to the target address specified by the instruction

The Return instruction is a special branch instruction that performs the operation

Branch to the address contained in the link register