UNIT 1

TYPES OF COMPUTER:

Computers can be broadly classified by their speed and computing power.

S.no.	Type	Specifications
1	PC (Personal Computer)	It is a single user computer system having moderately powerful microprocessor
2	Workstation	It is also a single user computer system, similar to personal computer however a more powerful microprocessor has.
3	Mini Computer	It is a multi-user computer system, capable of supporting hundreds of users simultaneously.
4	Main Frame	It is a multi-user computer system, capable of supporting hundreds of users simultaneously. Software technology is different from minicomputer.
5	Supercomputer	It is an extremely fast computer, which can execute hundreds of millions of instructions per second.

PC (Personal Computer)

A PC can be defined as a small, relatively inexpensive computer designed for an individual user. PCs are based on the microprocessor technology that enables manufacturers to put an entire CPU on one chip. Businesses use personal computers for word processing, accounting, desktop publishing, and for running spreadsheet and database management applications. At home, the most popular use for personal computers is playing games and surfing the Internet.

Although personal computers are designed as single-user systems, these systems are normally linked together to form a network. In terms of power, now-a-days high-end models of the Macintosh and PC offer the same computing power and graphics capability as low-end workstations by Sun Microsystems, Hewlett-Packard, and Dell.

Workstation

Workstation is a computer used for engineering applications (CAD/CAM), desktop publishing, software development, and other such types of applications which require a moderate amount of computing power and relatively high quality graphics capabilities.

Workstations generally come with a large, high-resolution graphics screen, large amount of RAM, inbuilt network support, and a graphical user interface. Most workstations also have mass storage device such as a disk drive, but a special type of workstation, called diskless workstation, comes without a disk drive.

Common operating systems for workstations are UNIX and Windows NT. Like PC, workstations are also single-user computers like PC but are typically linked together to form a local-area network, although they can also be used as stand-alone systems.

Minicomputer

It is a midsize multi-processing system capable of supporting up to 250 users simultaneously.

Mainframe

Mainframe is very large in size and is an expensive computer capable of supporting hundreds or even thousands of users simultaneously. Mainframe executes many programs concurrently and supports many simultaneous execution of programs.

Supercomputer

Supercomputers are one of the fastest computers currently available. Supercomputers are very expensive and are employed for specialized applications that require immense amount of mathematical calculations (number crunching).

For example, weather forecasting, scientific simulations, (animated) graphics, fluid dynamic calculations, nuclear energy research, electronic design, and analysis of geological data (e.g. in petrochemical prospecting).

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 have 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.,

Functional unit: -

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

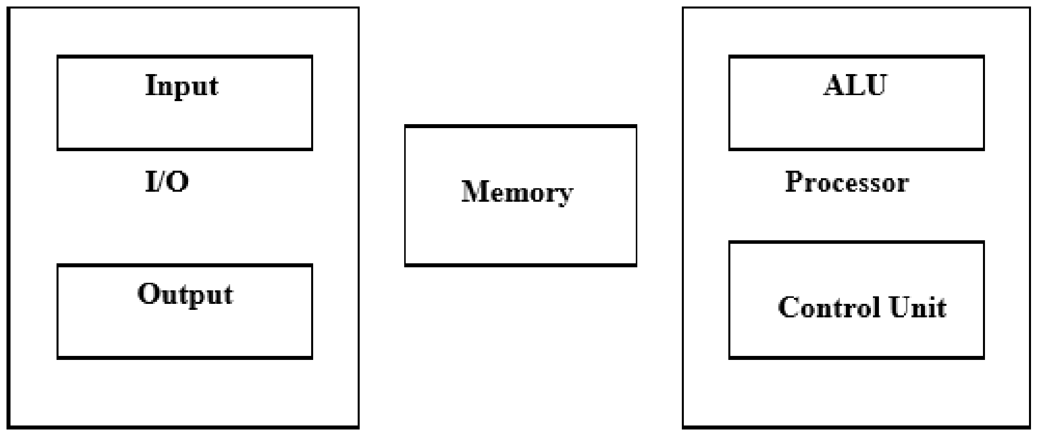


Fig a : Functional units of computer

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 speed 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 often 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 many 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.

<u>BASIC OPERATIONAL CONCEPTS: -</u>

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
R1, 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.

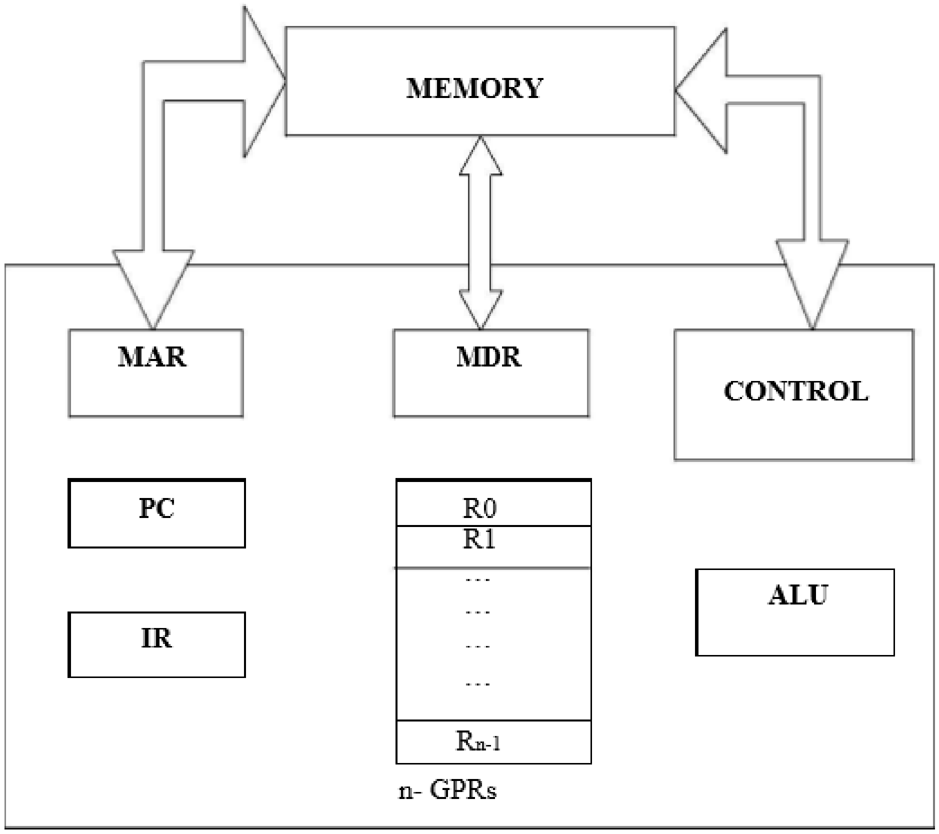


Fig b : Connections between the processor and the memory

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 counter PC: -

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}. 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 memory 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.

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

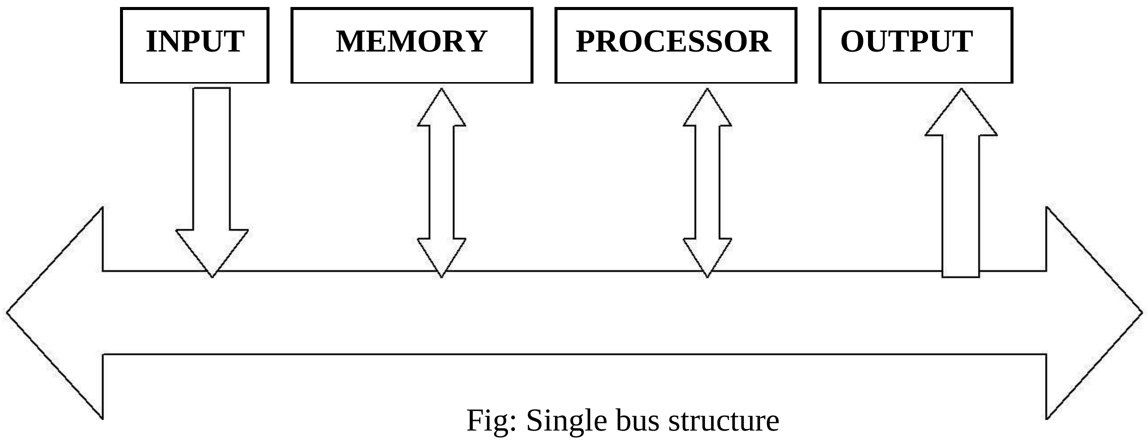


Fig: Single bus structure

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 (i.e. buffer registers). These buffers are electronic registers of small capacity when compared to the main memory but of comparable speed.

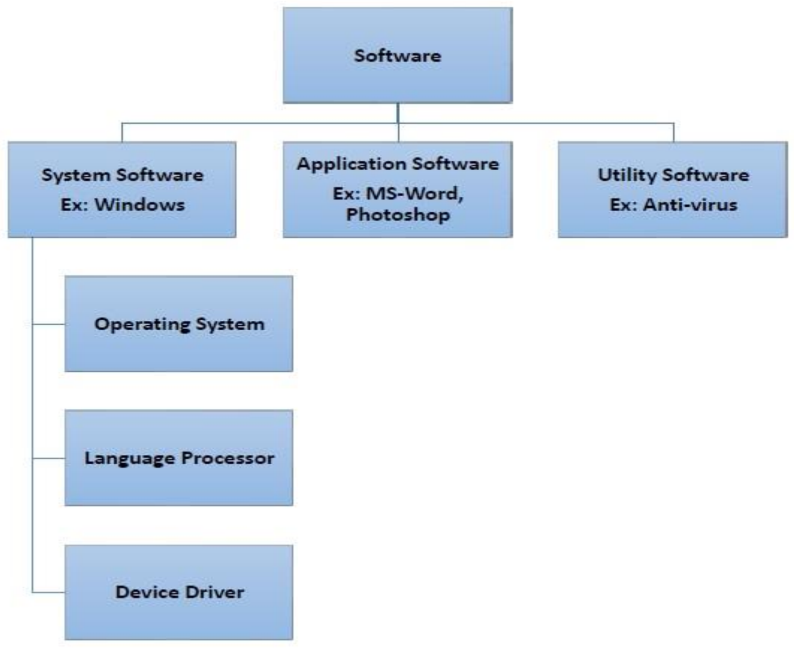
The instructions from the processor at once are loaded into these buffers and then the complete transfer of data at a fast rate will take place.

SOFTWARE

The hardware devices need user instructions to function. A set of instructions that achieve a single outcome are called program or procedure. Many programs functioning together to do tasks make a software.

For example, word-processing software enables the user to create, edit and save documents. A web browser enables the user to view and share web pages and multimedia files. There are two categories of software −

- System Software
- Application Software
- Utility Software



System Software

Software required to run the hardware parts of the computer and other application software are called **system software**. System software acts as **interface** between hardware and user applications. An interface is needed because hardware devices or machines and humans speak in different languages.

Machines understand only binary language i.e. 0 (absence of electric signal) and 1 (presence of electric signal) while humans speak in English, French, German, Tamil, Hindi and many other languages. English is the pre-dominant language of interacting with computers. Software is required to convert all human instructions into machine understandable instructions. And this is exactly what system software does.

Based on its function, system software is of four types −

- Operating System
- Language Processor
- Device Drivers

Operating System

System software that is responsible for functioning of all hardware parts and their interoperability to carry out tasks successfully is called **operating system (OS)**. OS is the first software to be loaded into computer memory when the computer is switched on and this is called **booting**. OS manages a computer's basic functions like storing data in memory, retrieving files from storage devices, scheduling tasks based on priority, etc.

Language Processor:

As discussed earlier, an important function of system software is to convert all user instructions into machine understandable language. When we talk of human machine interactions, languages are of three types −

- **Machine-level language** − this language is nothing but a string of 0s and 1s that the machines can understand. It is completely machine dependent.
- **Assembly-level language** − this language introduces a layer of abstraction by defining **mnemonics**. **Mnemonics** are English like words or symbols used to denote a long string of 0s and 1s. For example, the word "READ" can be defined to mean that computer has to retrieve data from the memory. The complete **instruction** will also tell the memory address. Assembly level language is **machine dependent**.
- **High level language** − this language uses English like statements and is completely independent of machines. Programs written using high level languages are easy to create, read and understand.

Program written in high level programming languages like Java, C++, etc. is called **source code**. Set of instructions in machine readable form is called **object code** or **machine code**. **System software** that converts source code to object code is called **language processor**. There are three types of language interpreters−

- **Assembler** − Converts assembly level program into machine level program.
- **Interpreter** − Converts high level programs into machine level program line by line.
- **Compiler** − Converts high level programs into machine level programs at one go rather than line by line.

Device Drivers

System software that controls and monitors functioning of a specific device on computer is called **device driver**. Each device like printer, scanner, microphone, speaker, etc. that needs to be attached externally to the system has a specific driver associated with it. When you attach a new device, you need to install its driver so that the OS knows how it needs to be managed.

Application Software:

Software that performs a single task and nothing else is called **application software**. Application software is very specialized in their function and approach to solving a problem. So, spreadsheet software can only do operations with numbers and nothing else. Hospital management software will manage hospital activities and nothing else. Here is some commonly used application software −

- Word processing
- Spreadsheet
- Presentation
- Database management
- Multimedia tools

Utility Software:

Application software that assist system software in doing their work is called **utility software**. Thus, utility software is actually a cross between system software and application software. Examples of utility software include −

- Antivirus software
- Disk management tools
- File management tools
- Compression tools
- Backup tools

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.

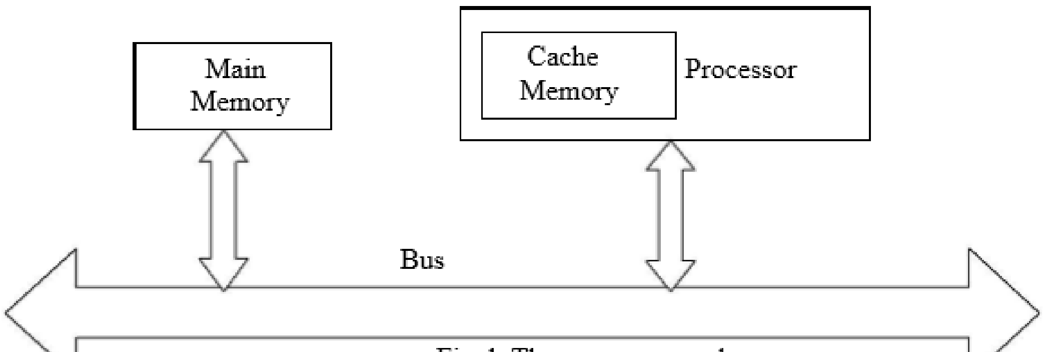


Fig d :The processor cache

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.

<u>Basic performance equation: -</u>

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 other 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

$$T = \frac{N \times S}{R}$$

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.

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 instruction 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.

<u>MUITIPROCESSORS AND MULTICOMPUTERS:</u>

A multiprocessor is a system with two or more CPUs or processors. Multiple processors can execute tasks at the same time. Failure in one processor will not affect the tasks of the other processors. Therefore, a multiprocessor is more reliable.

There are two types of multiprocessors called shared memory multiprocessor and distributed memory multiprocessor. In shared memory multiprocessors, all the CPUs shares the common memory. The processors communicate with each other by reading and writing to the memory. It is also called the **symmetric multiprocessor** system.

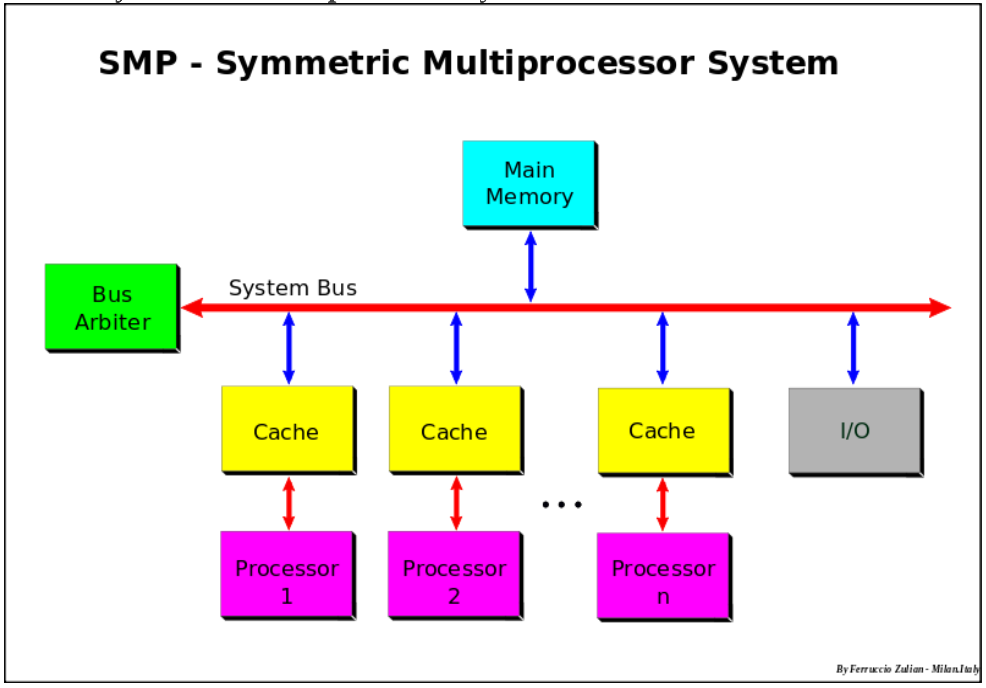


Figure 1: Symmetric Multiprocessor

In a distributed memory multiprocessor, every CPU has its own private memory. If the required data is not available in the private memory, the processor communicates with the main memory or the other processors via the bus. Overall, a multiprocessor provides a high computation speed, high performance and it is more tolerance to failures.

Multicomputer:

A multicomputer is a system with multiple processors that are connected together to solve a problem. Each processor has its own memory and it is accessible only by that particular processor. The processors can communicate with each other via an interconnection network.

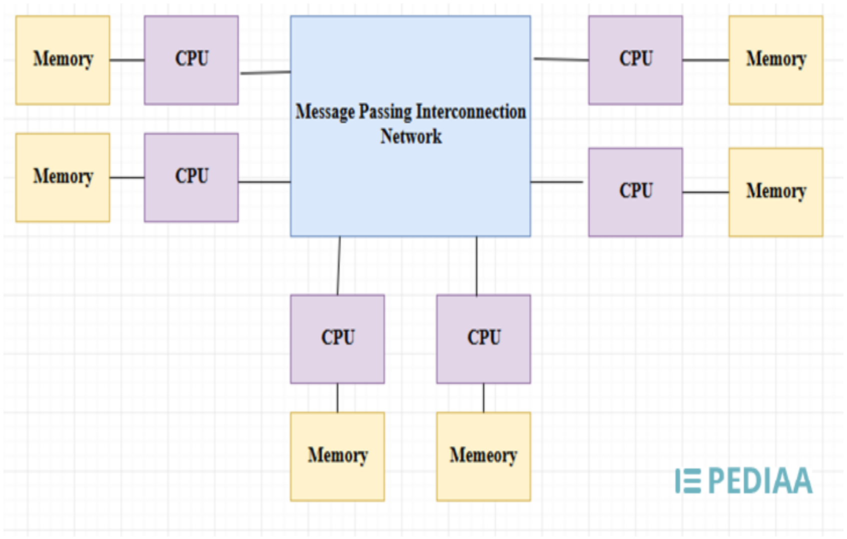


Figure 2: Multicomputer

As the multicomputer is capable of passing messages between the processors, it is possible to divide the task between the processors to complete the task. Therefore, a multicomputer can be used for distributed computing. It is easier and cost effective to build a multicomputer than a multiprocessor. On the other hand, programming a multicomputer is difficult.

Difference Between Multiprocessor and Multicomputer

BASIS FOR COMPARISON	**MULTIPROCESSOR**	**MULTICOMPUTER**
Basic	Multiple processors in a single computer.	Interlinked multiple autonomous computers.
Memory attached to the processing elements	Single shared	Multiple distributed
Communication between processing elements	Mandatory	Not necessary
Type of network	Dynamic network	Static network
Example	Sequent symmetry S-81	Message passing multicomputer

DATATYPES:

The data types found in the registers of digital computers may be classified as being one of the following categories

1. Numbers used in the arithmetic computations.

2. Letters of the alphabet used in data processing.

3. Other discrete symbols used for specific purposes

When we type some letters or words, the computer translates them in numbers as computers can understand only numbers. A computer can understand the positional number system where there are only a few symbols called digits and these symbols represent different values depending on the position they occupy in the number.

The value of each digit in a number can be determined using −

- The digit
- The position of the digit in the number
- The base of the number system (where the base is defined as the total number of digits available in the number system)

NUMBER SYSTEMS:

Radix: A number system of base or radix 'r' is a system that uses distinct symbols for r digits. Numbers are represented by a string of digit symbols. To determine the quantity that the number represents, it is necessary to multiply each digit by an integer power of r and then form sum of all weighted digits.

Decimal Number System:

The number system that we use in our day-to-day life is the decimal number system. Decimal number system has base 10 as it uses 10 digits from 0 to 9. In decimal number system, the successive positions to the left of the decimal point represent units, tens, hundreds, thousands, and so on.

Each position represents a specific power of the base (10). For example, the decimal number 1234 consists of the digit 4 in the unit's position, 3 in the tens position, 2 in the hundreds position, and 1 in the thousands position. Its value can be written as

(1 x 1000) + (2 x 100) + (3 x 10) + (4 x l)
$(1 \times 10^3) + (2 \times 10^2) + (3 \times 10^1) + (4 \times l0^0)$
1000 + 200 + 30 + 4
1234

As a computer programmer or an IT professional, you should understand the following number systems which are frequently used in computers.

S.no.	Number System and Description
1	**Binary Number System** Base 2. Digits used: 0, 1
2	**Octal Number System**

	Base 8. Digits used: 0 to 7
3	**Hexa Decimal Number System** Base 16. Digits used: 0 to 9, Letters used: A- F

Binary Number System:

Characteristics of the binary number system are as follows −

- Uses two digits, 0 and 1
- Also called as base 2 number system
- Each position in a binary number represents a **0** power of the base (2). Example 2^0
- Last position in a binary number represents a **x** power of the base (2). Example 2^x where **x** represents the last position - 1.

Example

Binary Number: 10101_2

Calculating Decimal Equivalent −

Step	Binary Number	Decimal Number
Step 1	10101_2	$((1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0))_{10}$
Step 2	10101_2	$(16 + 0 + 4 + 0 + 1)_{10}$
Step 3	10101_2	21_{10}

Note − 10101_2 is normally written as 10101.

Octal Number System:

Characteristics of the octal number system are as follows −

- Uses eight digits, 0,1,2,3,4,5,6,7
- Also called as base 8 number system
- Each position in an octal number represents a **0** power of the base (8). Example 8^0
- Last position in an octal number represents a **x** power of the base (8). Example 8^x where **x** represents the last position - 1

Example

Octal Number: 12570_8

Calculating Decimal Equivalent −

Step	Octal Number	Decimal Number
Step 1	12570_8	$((1 \times 8^4) + (2 \times 8^3) + (5 \times 8^2) + (7 \times 8^1) + (0 \times 8^0))_{10}$
Step 2	12570_8	$(4096 + 1024 + 320 + 56 + 0)_{10}$
Step 3	12570_8	5496_{10}

Note − 12570_8 is normally written as 12570.

Hexadecimal Number System:

Characteristics of hexadecimal number system are as follows −

- Uses 10 digits and 6 letters, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- Letters represent the numbers starting from 10. A = 10. B = 11, C = 12, D = 13, E = 14, F = 15
- Also called as base 16 number system
- Each position in a hexadecimal number represents a **0** power of the base (16). Example, 16^0
- Last position in a hexadecimal number represents a **x** power of the base (16). Example 16^x where **x** represents the last position - 1

Example

Hexadecimal Number: $19FDE_{16}$

Calculating Decimal Equivalent −

Step	Binary Number	Decimal Number
Step 1	$19FDE_{16}$	$((1 \times 16^4) + (9 \times 16^3) + (F \times 16^2) + (D \times 16^1) + (E \times 16^0))_{10}$

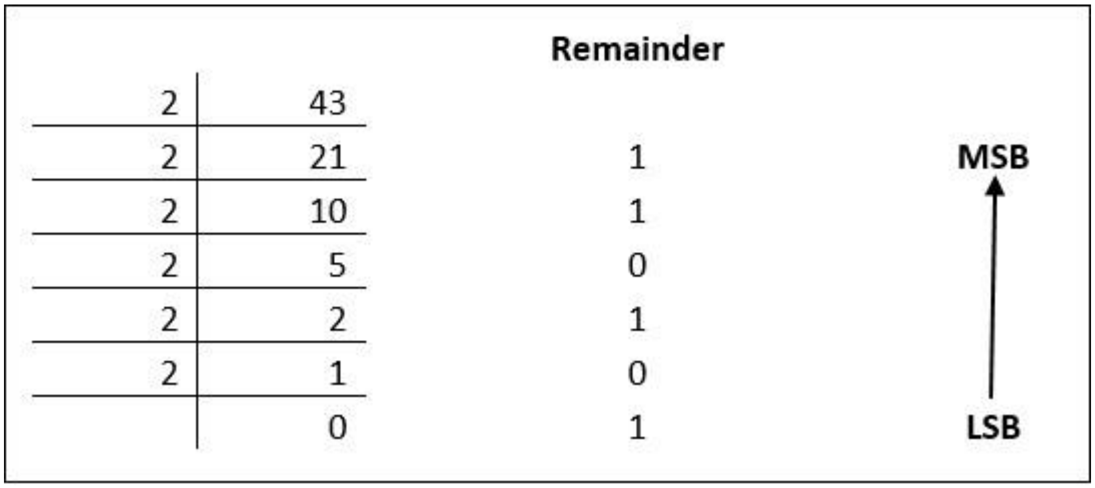
Step 2	$19FDE_{16}$	$((1 \times 16^4) + (9 \times 16^3) + (15 \times 16^2) + (13 \times 16^1) + (14 \times 16^0))_{10}$
Step 3	$19FDE_{16}$	$(65536 + 36864 + 3840 + 208 + 14)_{10}$
Step 4	$19FDE_{16}$	106462_{10}

CONVERSION:

As you know decimal, binary, octal and hexadecimal number systems are positional value number systems. To convert binary, octal and hexadecimal to decimal number, we just need to add the product of each digit with its positional value. Here we are going to learn other conversion among these number systems.

Decimal to Binary

Decimal numbers can be converted to binary by repeated division of the number by 2 while recording the remainder. Let's take an example to see how this happens.

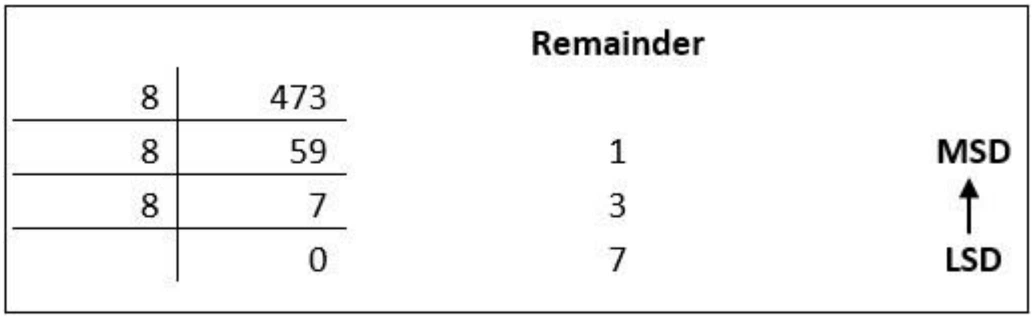


The remainders are to be read from bottom to top to obtain the binary equivalent.

$43_{10} = 101011_2$

Decimal to Octal

Decimal numbers can be converted to octal by repeated division of the number by 8 while recording the remainder. Let's take an example to see how this happens.

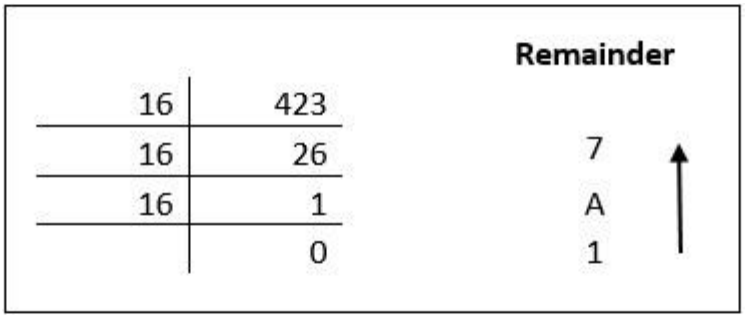


Reading the remainders from bottom to top,

$473_{10} = 731_8$

Decimal to Hexadecimal

Decimal numbers can be converted to octal by repeated division of the number by 16 while recording the remainder. Let’s take an example to see how this happens.



Reading the remainders from bottom to top we get,

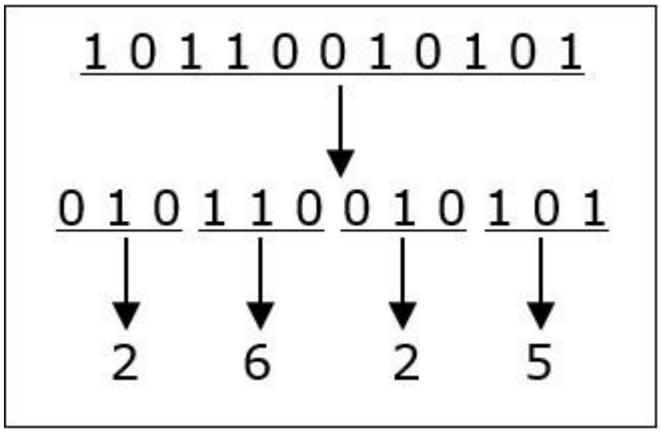
$423_{10} = 1A7_{16}$

Binary to Octal and Vice Versa

To convert a binary number to octal number, these steps are followed −

- Starting from the least significant bit, make groups of three bits.
- If there are one or two bits less in making the groups, 0s can be added after the most significant bit
- Convert each group into its equivalent octal number

Let’s take an example to understand this.



$1011001010_{12} = 2625_8$

To convert an octal number to binary, each octal digit is converted to its 3-bit binary equivalent according to this table.

Octal Digit	0	1	2	3	4	5	6	7
Binary Equivalent	000	001	010	011	100	101	110	111

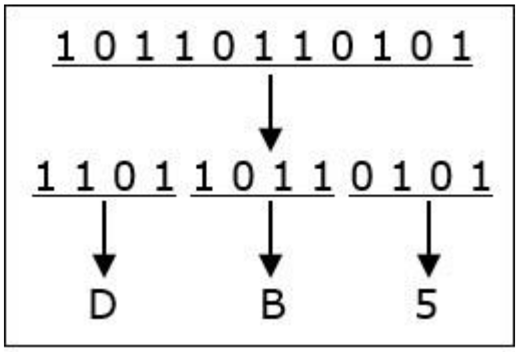
$54673_8 = 101100110111011_2$

Binary to Hexadecimal

To convert a binary number to hexadecimal number, these steps are followed −

- Starting from the least significant bit, make groups of four bits.
- If there are one or two bits less in making the groups, 0s can be added after the most significant bit.
- Convert each group into its equivalent octal number.

Let's take an example to understand this.



$10110110101_2 = DB5_{16}$

To convert an octal number to binary, each octal digit is converted to its 3-bit binary equivalent.

NUMBER REPRESENTATION:

It’s the representation for integers only where the decimal point is always fixed. i.e. at the end of rightmost point. it can be again represented in three ways.

1. Sign and Magnitude Representation: In this system, he most significant (leftmost) bit in the word as a sign bit.

If the sign bit is 0, the number is positive; if the sign bit is 1, the number is negative.

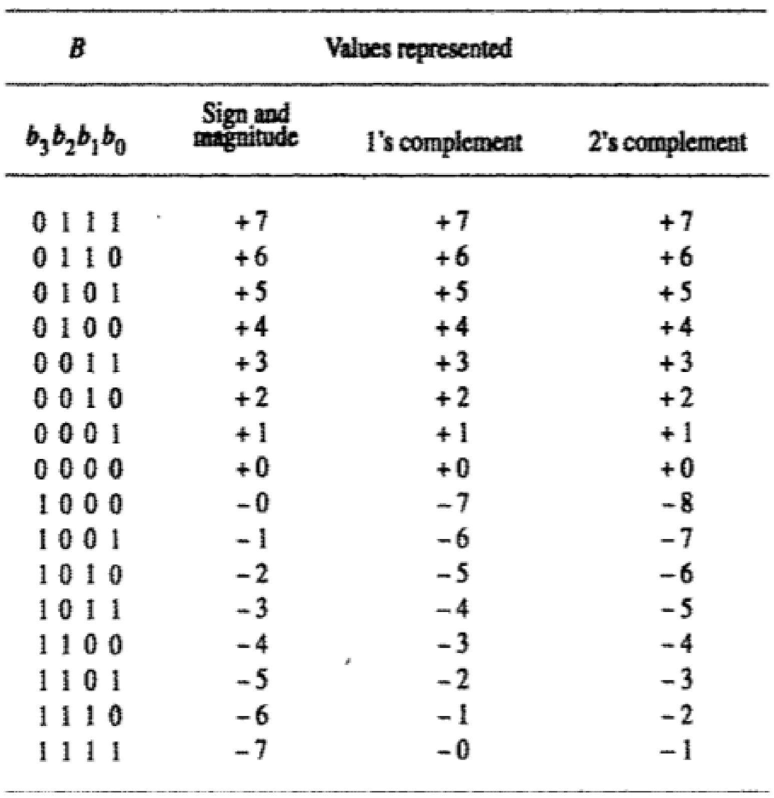
The simplest form of representing sign bit is the sign magnitude representation.

One of the drawback for sign magnitude number is addition and subtraction need to consider both sign of the numbers and their relative magnitude. Another drawback is there are two representation for 0(Zero) i.e +0 and -0.

2.One’s Complement (1’s) Representation; In this representation negative values are obtained by complementing each bit of the corresponding positive number. For example 1s complement of 0101 is 1010 . The process of forming the 1s complement of a given number is equivalent to subtracting that number from 2n -1 i.e from 1111 for 4 bit number.

3.Two’s Complement (2’s) Representation: Forming the 2’s complement of a number is done by subtracting that number from 2n . So 2’s complement of a number is obtained by adding 1 to 1’s complement of that number.

Ex: 2‟s complement of 0101 is 1010 +1 = 1011



B	Values represented		
$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	−0	−7	−8
1 0 0 1	−1	−6	−7
1 0 1 0	−2	−5	−6
1 0 1 1	−3	−4	−5
1 1 0 0	−4	−3	−4
1 1 0 1	−5	−2	−3
1 1 1 0	−6	−1	−2
1 1 1 1	−7	−0	−1

ARITHMETIC OPERATIONS AND PROGRAMS:

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) end of the bit vectors, propagating carries toward the high-order (left) end.

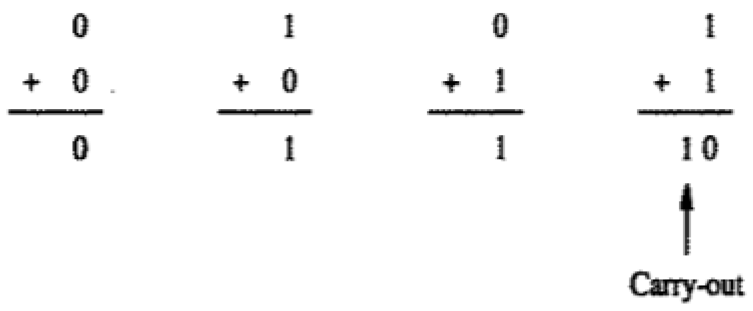


Figure 2.2 Addition of 1-bit numbers.

We now state the rules governing the addition and subtraction of n-bit signed numbers using the 2's-complement representation system.

1. To *add* two numbers, add their n-bit representations, ignoring the carry-out signal from the *most significant bit* (MSB) position. The sum will be the algebraically correct value in the 2's-complement representation as long as the answer is in the range -2^{n-1} through $+2^{n-1}-1$.

2. To *subtract* two numbers X and Y, that is, to perform $X - Y$, form the 2's-complement of Y and then add it to X, as in rule 1. Again, the result will be the algebraically correct value in the 2's-complement representation system if the answer is in the range -2^{n-1} through $+2^{n-1}-1$.

Figure 2.4 shows some examples of addition and subtraction. In all these 4-bit examples, the answers fall into the representable range of -8 through $+7$. When answers do not fall within the representable range, we say that arithmetic overflow has occurred. The next section discusses such situations. The four addition operations (*a*) through (*d*) in Figure 2.4 follow rule 1, and the six subtraction operations (*e*) through (*j*) follow rule 2. The subtraction operation requires the subtrahend (the bottom value) to be

```
(a)      0010    (+2)       (b)      0100    (+4)
       + 0011    (+3)              + 1010    (-6)
       ------    ----              ------    ----
         0101    (+5)                1110    (-2)
```

```
(c)    1011    (-5)          (d)    0111    (+7)
     + 1110    (-2)               + 1101    (-3)
     ------    ----               ------    ----
       1001    (-7)                 0100    (+4)

(e)    1101    (-3)                 1101
     - 1001    (-7)     ==>       + 0111
     ------    ----               ------    ----
                                    0100    (+4)

(f)    0010    (+2)                 0010
     - 0100    (+4)     ==>       + 1100
     ------    ----               ------    ----
                                    1110    (-2)

(g)    0110    (+6)                 0110
     - 0011    (+3)     ==>       + 1101
     ------    ----               ------    ----
                                    0011    (+3)

(h)    1001    (-7)                 1001
     - 1011    (-5)     ==>       + 0101
     ------    ----               ------    ----
                                    1110    (-2)

(i)    1001    (-7)                 1001
     - 0001    (+1)     ==>       + 1111
     ------    ----               ------    ----
                                    1000    (-8)

(j)    0010    (+2)                 0010
     - 1101    (-3)     ==>       + 0011
     ------    ----               ------    ----
                                    0101    (+5)
```

Figure 2.4 2's-complement add and subtract operations.

<u>Basic input output operations:</u>

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

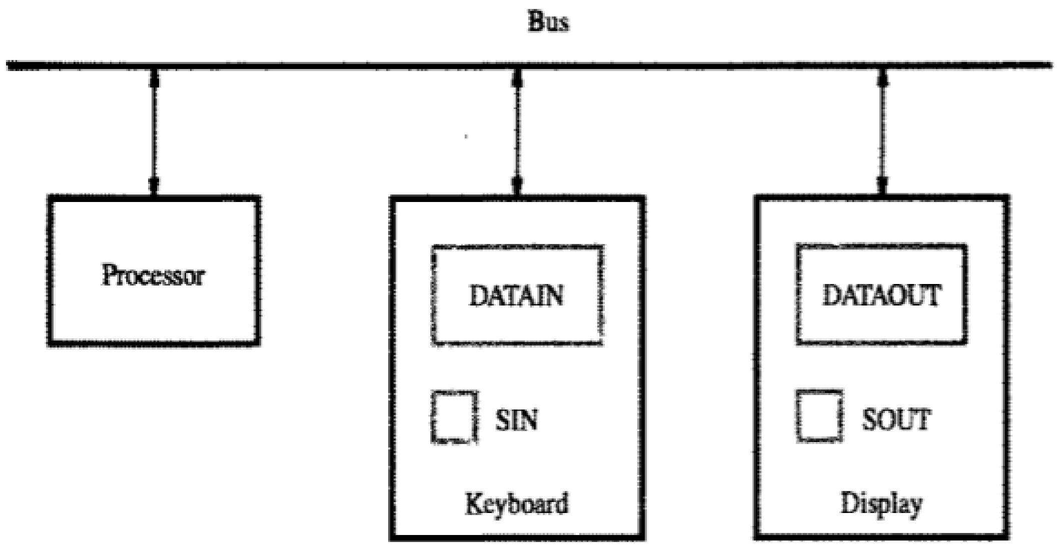


Figure 2.19 Bus connection for processor, keyboard, and display.

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.

A solution to this problem is as follows: On output, the processor sends the first character and then waits for a signal from the display that the character has been received. It then sends the second character, and so on. Input is sent from the keyboard in a similar way; the processor waits for a signal indicating that a character key has been struck and that its code is available in some buffer register associated with the keyboard. Then the processor proceeds to read that code.

The keyboard and the display are separate devices as shown in Figure 2.19. 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.

Consider the problem of moving a character code from the keyboard to the processor. 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 Figure 2.19. 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 process 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. Under program control, the processor monitors SOUT, and when SOUT is set to 1, the processor transfers a character code to DATAOUT. The transfer of a character to DATAOUT clears SOUT to 0; when the display device is ready to receive a second character, SOUT

and SOUT are part of circuitry commonly known as a *device interface*. The circuitry for each device is connected to the processor via a bus, as indicated in Figure 2.19.

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:

```
READWAIT   Branch to READWAIT if SIN = 0
           Input from DATAIN to R1
```

The Branch operation is usually implemented by two machine instructions. The first instruction tests the status flag and the second performs the branch. Although the details vary from computer to computer, the main idea is that the processor monitors the status flag by executing a short *wait loop* and proceeds to transfer the input data when SIN is set to 1 as a result of a key being struck. The Input operation resets SIN to 0.

An analogous sequence of operations is used for transferring output to the display. An example is

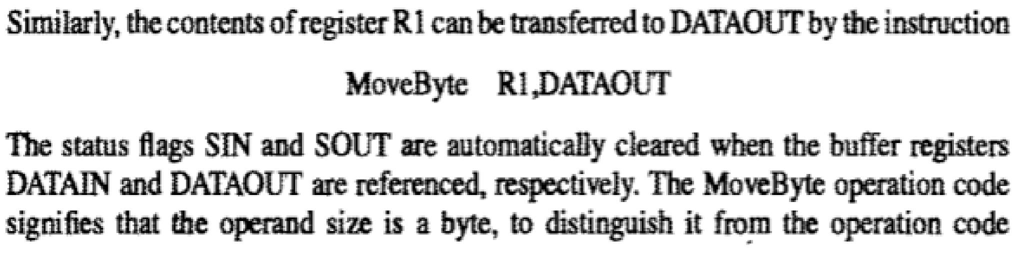
```
WRITEWAIT   Branch to WRITEWAIT if SOUT = 0
            Output from R1 to DATAOUT
```

Again, the Branch operation is normally implemented by two machine instructions. The wait loop is executed repeatedly until the status flag SOUT is set to 1 by the display when it is free to receive a character. The Output operation transfers a character from R1 to DATAOUT to be displayed, and it clears SOUT to 0.

We assume that the initial state of SIN is 0 and the initial state of SOUT is 1. This initialization is normally performed by the device control circuits when the devices are placed under computer control before program execution begins.

Until now, we have assumed that the addresses issued by the processor to access instructions and operands always refer to memory locations. Many computers use an arrangement called *memory-mapped I/O* in which some memory address values are used to refer to peripheral device buffer registers, such as DATAIN and DATAOUT. Thus, no special instructions are needed to access the contents of these registers; data can be transferred between these registers and the processor using instructions that we have already discussed, such as Move, Load, or Store. For example, the contents of the keyboard character buffer DATAIN can be transferred to register R1 in the processor by the instruction

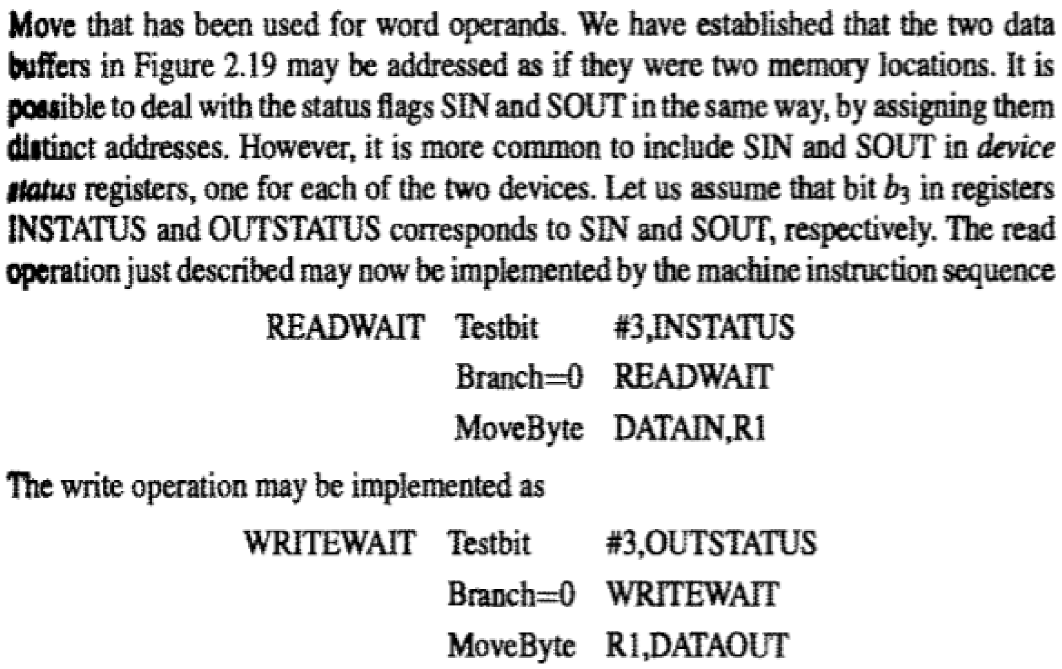
```
MoveByte   DATAIN,R1
```



Similarly, the contents of register R1 can be transferred to DATAOUT by the instruction

```
MoveByte    R1,DATAOUT
```

The status flags SIN and SOUT are automatically cleared when the buffer registers DATAIN and DATAOUT are referenced, respectively. The MoveByte operation code signifies that the operand size is a byte, to distinguish it from the operation code



Move that has been used for word operands. We have established that the two data buffers in Figure 2.19 may be addressed as if they were two memory locations. It is possible to deal with the status flags SIN and SOUT in the same way, by assigning them distinct addresses. However, it is more common to include SIN and SOUT in *device status* registers, one for each of the two devices. Let us assume that bit b_3 in registers INSTATUS and OUTSTATUS corresponds to SIN and SOUT, respectively. The read operation just described may now be implemented by the machine instruction sequence

```
READWAIT   Testbit     #3,INSTATUS
           Branch=0    READWAIT
           MoveByte    DATAIN,R1
```

The write operation may be implemented as

```
WRITEWAIT  Testbit     #3,OUTSTATUS
           Branch=0    WRITEWAIT
           MoveByte    R1,DATAOUT
```

Instruction Sequencing and Execution:

Instruction and Instruction sequencing: A computer program consist of a sequence of small steps, such as adding two numbers,testing for a particular condition, reading a character from the keyboard, or sending a character to be displayed on a display screen. 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: In general, the information is transferred from one location to another in a computer. Possible locations for such transfers are memory locations, processor registers, or registers in the I/O subsystem.
For example, Names for the addresses of memory locations may be LOC, PLACE, A, VAR2; processor register names may be RO, 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

$$R1 \longleftarrow [LOC]$$

means that the contents of memory location LOC are transferred into processor register R1. 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

$$R1 \longleftarrow [R1] + [R2]$$

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 placed, overwriting the old contents of that location.

ASSEMBLY TRANSFER NOTATION: The another type of notation to represent machine instructions and programs is assembly transfer notation. For example, an instruction that causes the transfer described above, from memory location LOC to processor register RI, is specified by the statement.

Move LOC,RI
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 RI and R2 and placing their sum in R3 can be specified by the assembly language statement

Add RI,R2,R3

Instructions in a computer can be of multiple lengths with a variable number of addresses. The various address fields in the instruction format of a computer vary as per the organization of its registers. It depends on the multiple address fields the instruction can be categorized as three address instructions, two address instructions, one address instruction, and zero address instruction.

Three Address Instructions

The general format of a three address instruction is defined as −

operation source 1, source 2, destination

ADD A, B, C

where A, B, and C are the three variables that are authorized to a different area in the memory. ‘ADD’ is the operation that is implemented on operands. ‘A’ and ‘B’ are the source operands and ‘C’ is the destination operand.

Therefore, bits are needed to determine the three operands. n bit is needed to determine one operand (one memory address). Likewise, 3n bits are needed to define three operands (three memory addresses). Bits are also needed to determine the ADD operation.

Two Address Instructions

The general format of a two address instruction is defined as −

operation source, destination

ADD A, B

where A and B are the two variables that are designated to a specific location in the memory. ‘ADD’ is the operation that is implemented on the operands. This instruction adds the content of the variables A and B and saves the result in variable B. Here, ‘A’ is the source operand and ‘B’ is treated as both source and destination operands.

Bits are needed to determine the two operands. n bit is needed to define one operand (one memory address). Likewise, 2n bits are needed to determine two operands (two memory addresses). Bits are also needed to definite the ADD operation.

One Address Instruction

The general format of one address instruction is defined as −

operation source

ADD A

where A is the variable that is authorized to a specific location in the memory. ‘ADD’ is the operation that is implemented on operand A. This instruction adds the content of the variable A into the accumulator and saves the result in the accumulator by restoring the content of the accumulator.

Zero Address Instructions

The locations of the operands in zero address instructions are represented implicitly. These instructions store operands in a structure are known as a pushdown stack.

Instruction execution and sequencing

Execution of instruction is a two-step procedure. In the first step, the instruction is **fetched** from memory. In the second step, the instruction is executed.

First Step-> Instruction Fetch

The processor has a register named **program counter** which has the address of the instruction that has to be executed next. To start the execution of a program, the address of first instruction has to be placed in the program counter.

After the address of first instruction is placed in the program counter, the processor uses this address to fetch the first instruction. As soon as the first instruction is fetched the content in program counter is incremented by the word length (the group of n bits that can store a single, basic instruction) i.e. now, it has the address of next successive instruction to be executed.

After fetching the instruction from memory location it is placed in the **instruction register** of the processor.

Second Step -> Instruction execution

In the second step, the processor examines the instruction present in the **instruction register** and find out which operation has to be performed.

It first fetches the **operands** involved in the instruction from the memory or processor register. Then the processor performs the **arithmetic and logical operation** and **stores** the result in the destination location mention in the instruction.

During this time the content of program counter is incremented so that it could point next instruction for execution. This execution of instructions in increasing order of addresses is **straight line sequencing**.

• The program is executed as follows:
1) Initially, the address of the first instruction is loaded into PC (Figure 2.8).
2) 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.
3) During the execution of each instruction, PC is incremented by 4 to point to next instruction.

• There are 2 phases for Instruction Execution:
1) Fetch Phase: The instruction is fetched from the memory-location and placed in the IR.
2) Execute Phase: The contents of IR is examined to determine which operation is to beperformed. The specified-operation is then performed by the processor.

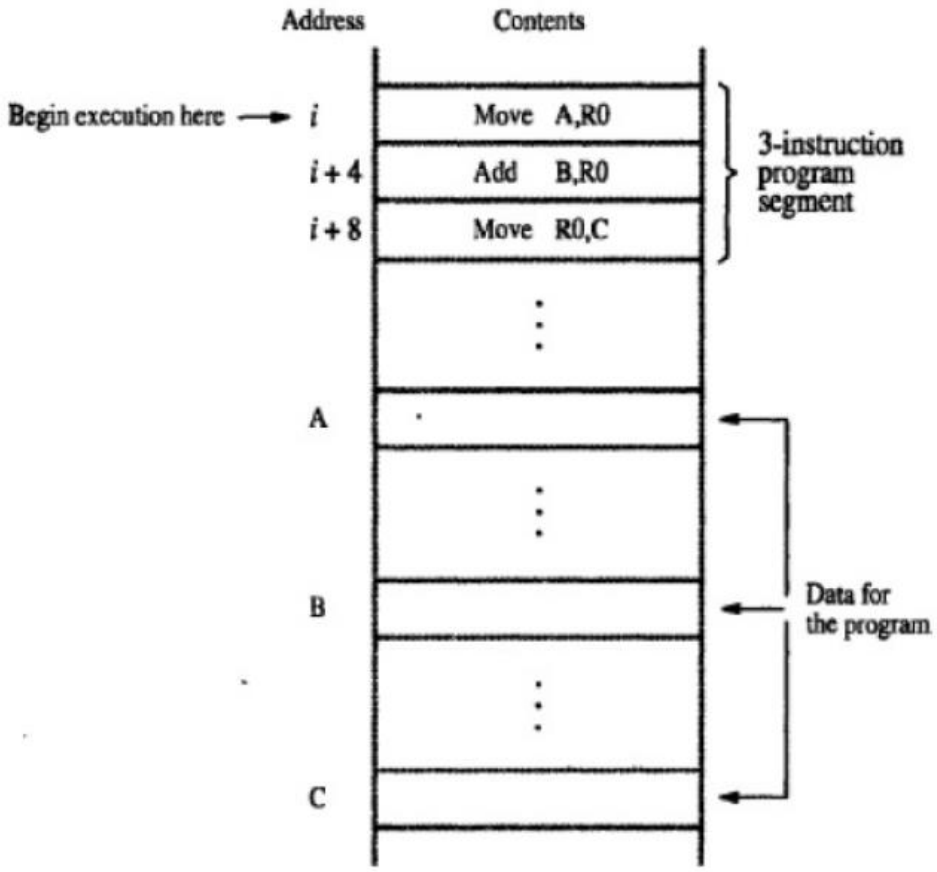


Figure 2.8 A program for C ← [A] + [B].

Program Explanation

- Consider the program for adding a list of n numbers (Figure 2.9).
- The Address of the memory-locations containing the n numbers are symbolically given as 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 memory-location SUM.

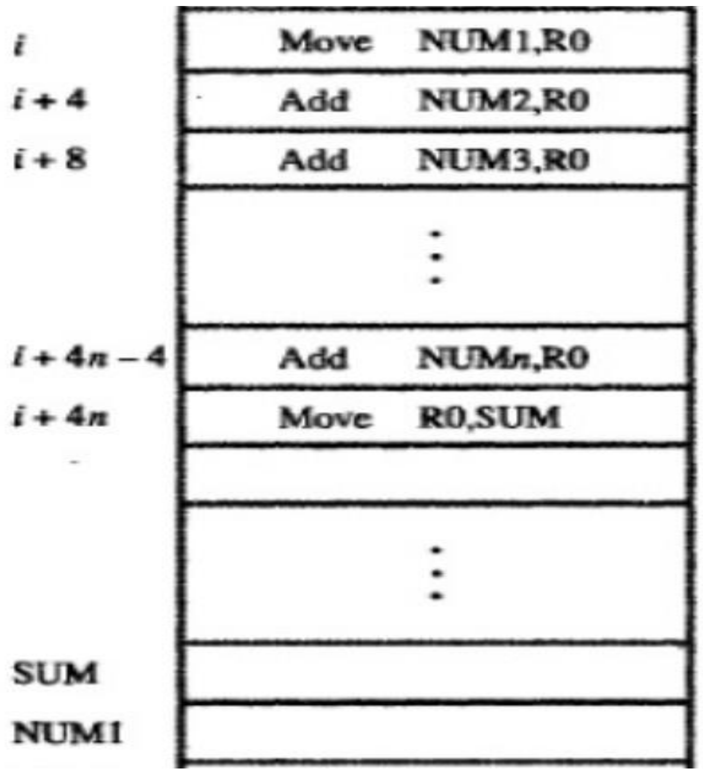


Figure 2.9: a straight line program for adding n numbers

Branching Instruction

Now, suppose we have to add n numbers. Their addresses in the symbolic form are Num1, Num2, Num3,.NumN. So, the instructions for adding these n numbers are as follow:

Load R2, Num1

Load R3, Num2

Add R2, R3, R2

Load R3, Num3

Add R2, R3, R2

Load R3, Num4

Add R2, R3, R2

.

.

.

.

Load R3, NumN

Add R2, R3, R2

Store R2 SUM

Instead of writing this straight-line sequence of instruction we could implement a **loop** to this program. To add n numbers a loop has to be repeated n number of times.

Suppose there are n numbers to be added. So, this count value n is stored in the memory location N. Below is the instruction set specifying the loop to add n numbers

Load R2, N

Clear R3

Determine Next number // *Loop start*

Load R5, Next number

Add R3, R5

Branch if [R2]>0 Loop // *Loop repeat*

Store R3, SUM

In the above code "Branch if [R2]>0 Loop" is a **branch instruction**. This instruction loads the address of the next number in the program counter. The address of the next number to be added is a **branch target**.

The branch instruction we have used is the **conditional branch** instruction as it will perform loop until the value of R2 >0. As the condition R2>0 will become false it will stop performing the loop.

ADDRESSING MODES:

The operation field of an instruction specifies the operation to be performed. This operation will be executed on some data which is stored in computer registers or the main memory. The way any operand is selected during the program execution is dependent on the addressing mode of the instruction. The purpose of using addressing modes is as follows:

1. To give the programming versatility to the user.

2. To reduce the number of bits in addressing field of instruction.

Types of Addressing Modes

Below we have discussed different types of addressing modes one by one:

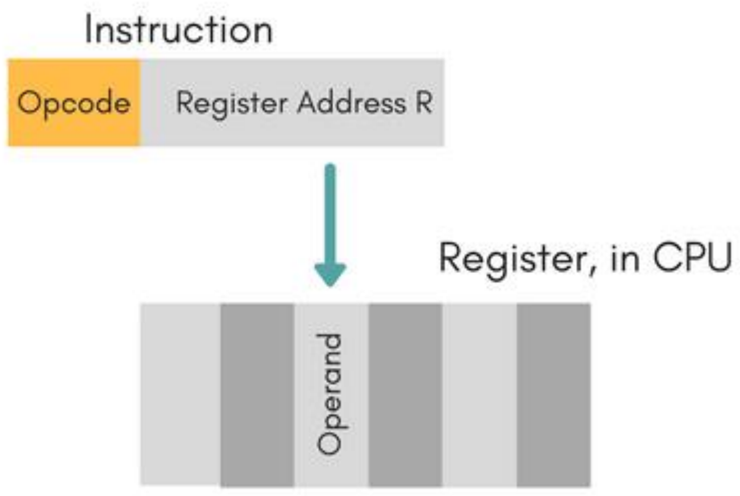
Immediate Mode

In this mode, the operand is specified in the instruction itself. An immediate mode instruction has an operand field rather than the address field.

For example: ADD 7, which say Add 7 to contents of accumulator. 7 is the operand here.

Register Mode

In this mode the operand is stored in the register and this register is present in CPU. The instruction has the address of the Register where the operand is stored.



Advantages

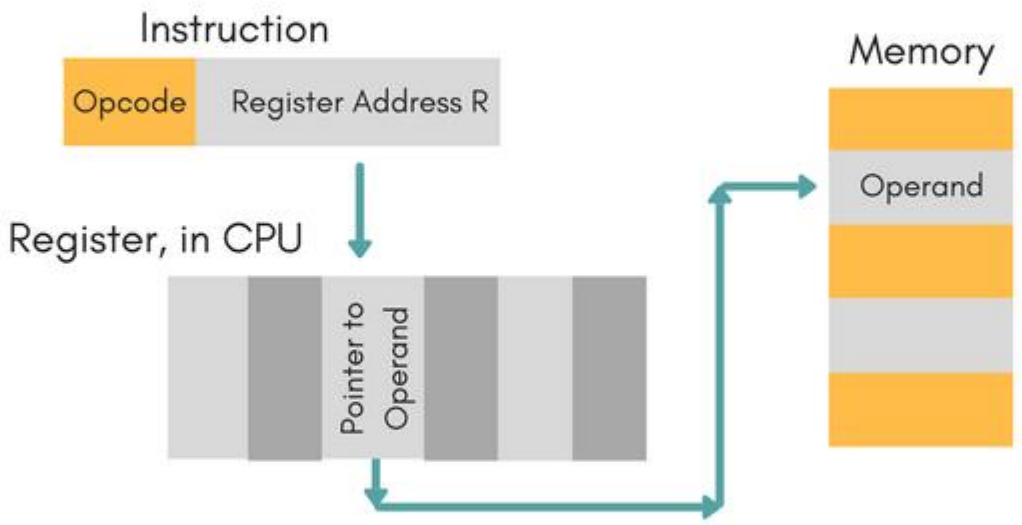
- Shorter instructions and faster instruction fetch.
- Faster memory access to the operand(s)

Disadvantages

- Very limited address space
- Using multiple registers helps performance but it complicates the instructions.

Register Indirect Mode

In this mode, the instruction specifies the register whose contents give us the address of operand which is in memory. Thus, the register contains the address of operand rather than the operand itself.



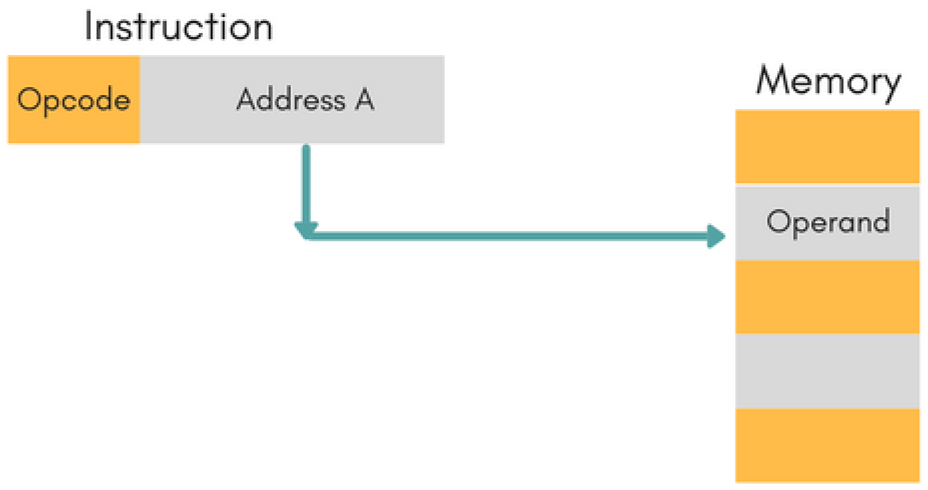
Auto Increment/Decrement Mode

In this the register is incremented or decremented after or before its value is used.

Direct Addressing Mode

In this mode, effective address of operand is present in instruction itself.

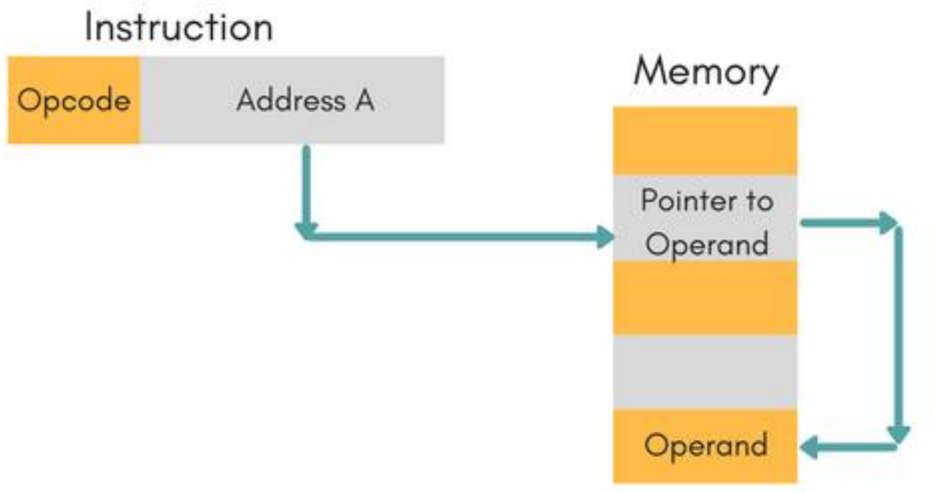
- Single memory reference to access data.
- No additional calculations to find the effective address of the operand.



For Example: ADD R1, 4000 - In this the 4000 is effective address of operand.

Indirect Addressing Mode

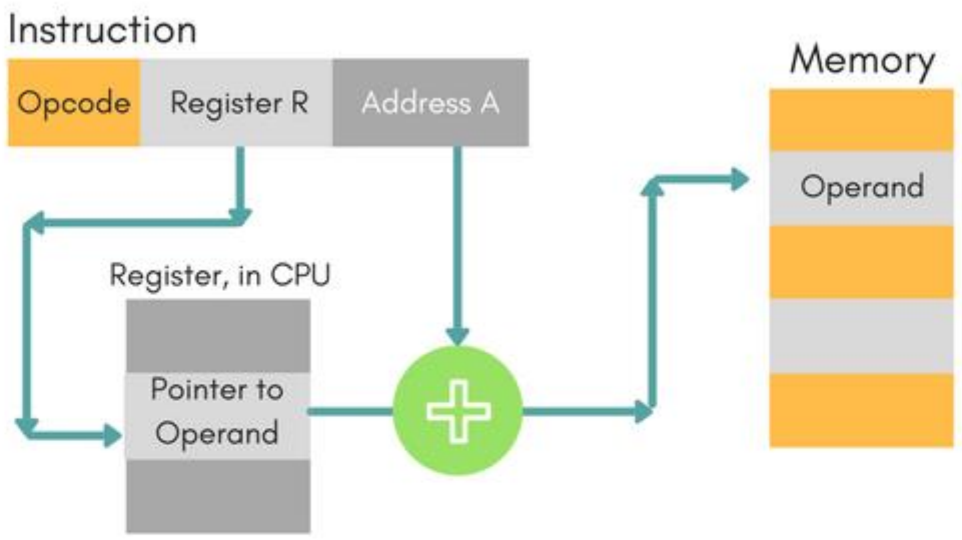
In this, the address field of instruction gives the address where the effective address is stored in memory. This slows down the execution, as this includes multiple memory lookups to find the operand.



Displacement Addressing Mode

In this the contents of the indexed register is added to the Address part of the instruction, to obtain the effective address of operand.

EA = A + (R), In this the address field holds two values, A(which is the base value) and R(that holds the displacement), or vice versa.



Relative Addressing Mode

It is a version of Displacement addressing mode.

In this the contents of PC (Program Counter) is added to address part of instruction to obtain the effective address.

EA = A + (PC), where EA is effective address and PC is program counter.

The operand is A cells away from the current cell (the one pointed to by PC)

Base Register Addressing Mode

It is again a version of Displacement addressing mode. This can be defined as EA = A + (R), where A is displacement and R holds pointer to base address.

Stack Addressing Mode

In this mode, operand is at the top of the stack. For example: ADD, this instruction will *POP* top two items from the stack, add them, and will then *PUSH* the result to the top of the stack.

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.

Fig a illustrates this procedure

Memory Location	Calling program	Memory location	Subroutine SUB
	….		
	….		
200	Call SUB	1000	first instruction
204	next instruction		….
	….		….
	….		Return
	….		

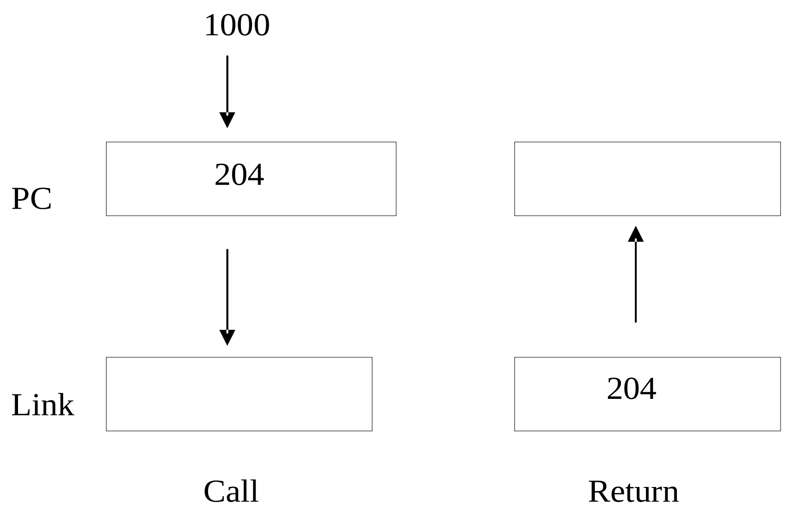


Fig b Subroutine linkage using a link register

SUBROUTINE NESTING AND THE PROCESSOR STACK:-

A common programming practice, called subroutine nesting, is to have one subroutine call another. In this case, the return address of the second call is also stored in the link register, destroying its previous contents. Hence, it is essential to save the contents of the link register in some other location before calling another subroutine. Otherwise, the return address of the first subroutine will be lost.

Subroutine nesting can be carried out to any depth. Eventually, the last subroutine called completes its computations and returns to the subroutine that called it. The return address needed for this first return is the last one generated in the nested call sequence. That is, return addresses are generated and used in a last-in-first-out order. This Suggests that the return addresses associated with subroutine calls should be pushed onto a stack. A particular register is designated as the stack pointer, SP, to be used in this operation. The stack pointer points to a stack called the processor Stack. The Call

Instruction pushes the contents of the PC onto the processor stack and loads the subroutine address into the PC. The Return instruction pops the return address from the processor stack into the PC.

PARAMETER PASSING:-

When calling a subroutine, a program must provide to the subroutine the parameters, that is, the operands or their addresses, to be used in the computation. Later, the subroutine returns other parameters, in this case, the results of the computation. This exchange of information between a calling program and a subroutine is referred to as parameter passing. Parameter passing may be accomplished in several ways. The parameters may be placed in registers or in memory locations, where they can be accessed by the subroutine. Alternatively, the parameters may be placed on the processor stack used for saving the return address.

The purpose of the subroutines is to add a list of numbers. Instead of passing the actual list entries, the calling program passes the address of the first number in the list. This technique is called passing by reference. The second parameter is passed by value, that is, the actual number of entries, n, is passed to the subroutine.

THE STACK FRAME:-

Now, observe how space is used in the stack in the example. During execution of the subroutine, six locations at the top of the stack contain entries that are needed by the subroutine. These locations constitute a private workspace for the subroutine, created at the time the subroutine is entered and freed up when the subroutine returns control to the calling program. Such space is called a stack frame.

Fig a A subroutine stack frame example.

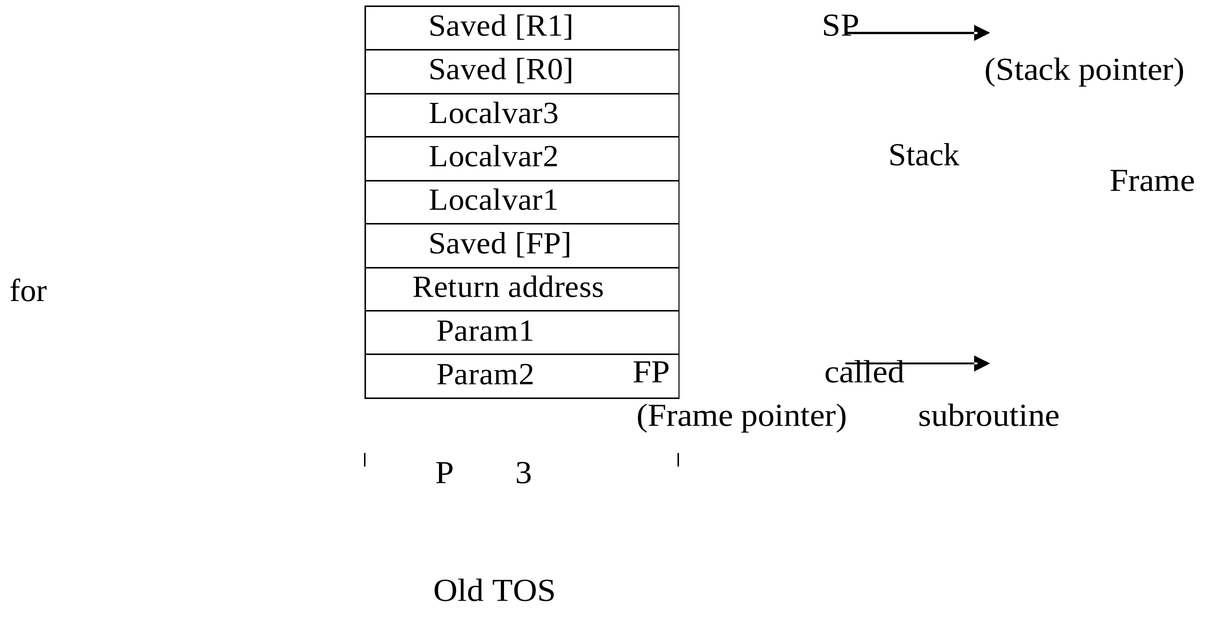


fig b shows an example of a commonly used layout for information in a stack frame. In addition to the stack pointer SP, it is useful to have another pointer register, called the frame pointer (FP), for convenient access to the parameters passed to the subroutine and to the local memory variables used by the subroutine. These local variables are only used within the subroutine, so it is appropriate to allocate space for them in the stack frame associated with the subroutine. We assume that four parameters are passed to the subroutine, three local variables are used within the subroutine, and registers R0 and R1 need to be saved because they will also be used within the subroutine.

The pointers SP and FP are manipulated as the stack frame is built, used, and dismantled for a particular of the subroutine. We begin by assuming that SP point to the old top-of-stack (TOS) element in fig b. Before the subroutine is called, the calling program pushes the four parameters onto the stack. The call instruction is then executed, resulting in the return address being pushed onto the stack. Now, SP points to this return address, and the first instruction of the subroutine is about to be executed. This is the point at which the frame pointer FP is set to contain the proper memory address. Since FP is usually a general-purpose register, it may contain information of use to the Calling program. Therefore, its contents are saved by pushing them onto the stack. Since the SP now points to this position, its contents are copied into FP.

Thus, the first two instructions executed in the subroutine

```
are Move   FP, -(SP)
Move   SP, FP
```

After these instructions are executed, both SP and FP point to the saved FP contents.

```
Subtract   #12, SP
```

Finally, the contents of processor registers R0 and R1 are saved by pushing them onto the stack. At this point, the stack frame has been set up as shown in the fig.

The subroutine now executes its task. When the task is completed, the subroutine pops the saved values of R1 and R0 back into those registers, removes the local variables from the stack frame by executing the instruction.

```
Add    #12, SP
```

And pops the saved old value of FP back into FP. At this point, SP points to the return address, so the Return instruction can be executed, transferring control back to the calling program.

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. The structure is sometimes referred to as a *pushdown* stack. Imagine a pile of trays in a cafeteria; customers pick up new trays from the top of the pile, and clean trays are added to the pile by placing them onto the top of the pile. 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.

Data stored in the memory of a computer can be organized as a stack, with successive elements occupying successive memory locations. Assume that the first element is placed in location BOTTOM, and when new elements are pushed onto the stack, they are placed in successively lower address locations. We use a stack that grows in the direction of decreasing memory addresses in our discussion, because this is a common practice.

Figure 2.21 shows a stack of word data items in the memory of a computer. 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. If we assume a

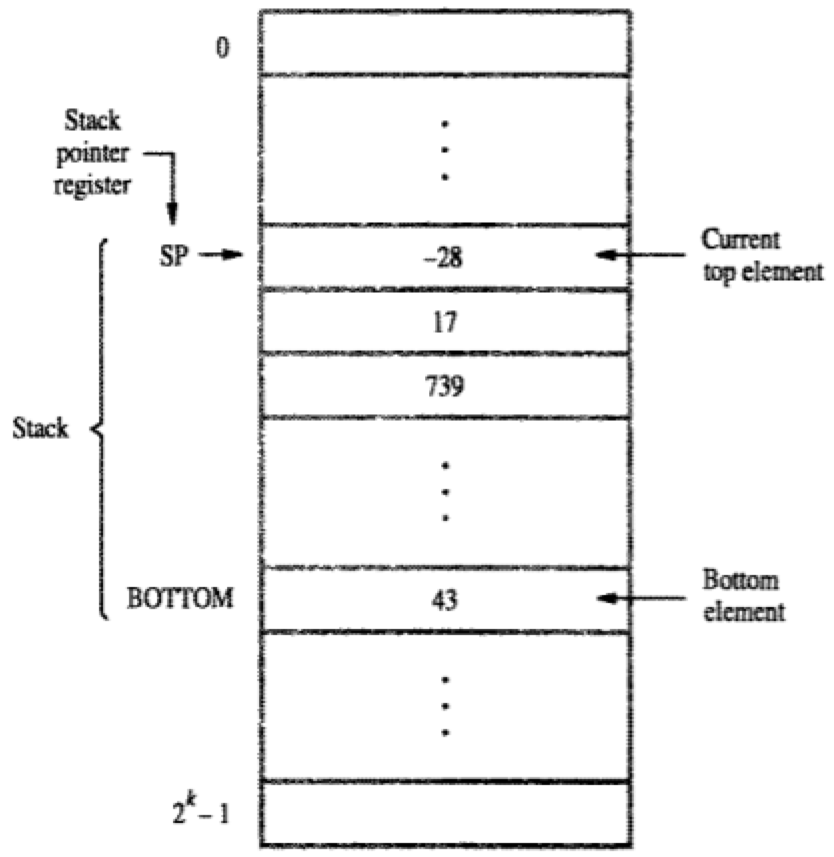


Figure 2.21 A stack of words in the memory.

byte-addressable memory with a 32-bit word length, the push operation can be implemented as

```
Subtract  #4,SP
Move      NEWITEM,(SP)
```

where the Subtract instruction subtracts the source operand 4 from the destination operand contained in SP and places the result in SP. These two instructions move the word from location NEWITEM onto the top of the stack, decrementing the stack pointer by 4 before the move. The pop operation can be implemented as

```
Move  (SP),ITEM
Add   #4,SP
```

These two instructions move the top value from the stack into location ITEM and then increment the stack pointer by 4 so that it points to the new top element. Figure 2.22 shows the effect of each of these operations on the stack in Figure 2.21.

If the processor has the Autoincrement and Autodecrement addressing modes, then the push operation can be performed by the single instruction

```
Move   NEWITEM,-(SP)
```

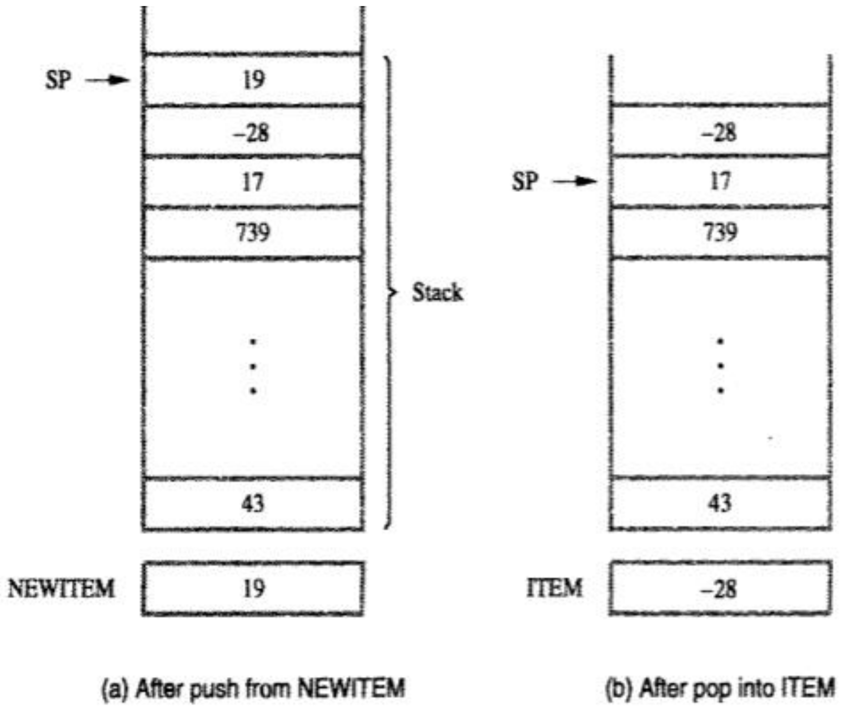


Figure 2.22 Effect of stack operations on the stack in Figure 2.21.

```
Move   (SP)+,ITEM
```

When a stack is used in a program, it is usually allocated a fixed amount of space in the memory. In this case, we must avoid pushing an item onto the stack when the stack has reached its maximum size. Also, we must avoid attempting to pop an item off an empty stack, which could result from a programming error. Suppose that a stack runs from location 2000 (BOTTOM) down no further than location 1500. The stack pointer is loaded initially with the address value 2004. Recall that SP is decremented by 4 before new data are stored on the stack. Hence, an initial value of 2004 means that the first item pushed onto the stack will be at location 2000. To prevent either pushing an item on a full stack or popping an item off an empty stack, the single-instruction push and pop operations can be replaced by the instruction sequences shown in Figure 2.23.

The Compare instruction

```
Compare   src,dst
```

performs the operation

$$[\text{dst}] - [\text{src}]$$

and sets the condition code flags according to the result. It does not change the value of either operand.

```
SAFEPOP   Compare    #2000,SP       Check to see if the stack pointer contains
          Branch>0   EMPTYERROR       an address value greater than 2000. If it
                                      does, the stack is empty. Branch to the
                                      routine EMPTYERROR for appropriate
                                      action.
          Move       (SP)+,ITEM     Otherwise, pop the top of the stack into
                                      memory location ITEM.
```

(a) Routine for a safe pop operation

```
SAFEPUSH  Compare    #1500,SP        Check to see if the stack pointer
          Branch≤0   FULLERROR         contains an address value equal
                                       to or less than 1500. If it does, the
                                       stack is full. Branch to the routine
                                       FULLERROR for appropriate action.
          Move       NEWITEM,−(SP)   Otherwise, push the element in memory
                                       location NEWITEM onto the stack.
```

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 stack at any given time. On the other hand, both ends of a queue move to higher 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

add

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 (see Problems 2.18 and 2.19).