MODULE-3

Digital Design and Computer Organization

Pothi Reddy. K

Functional Units :-

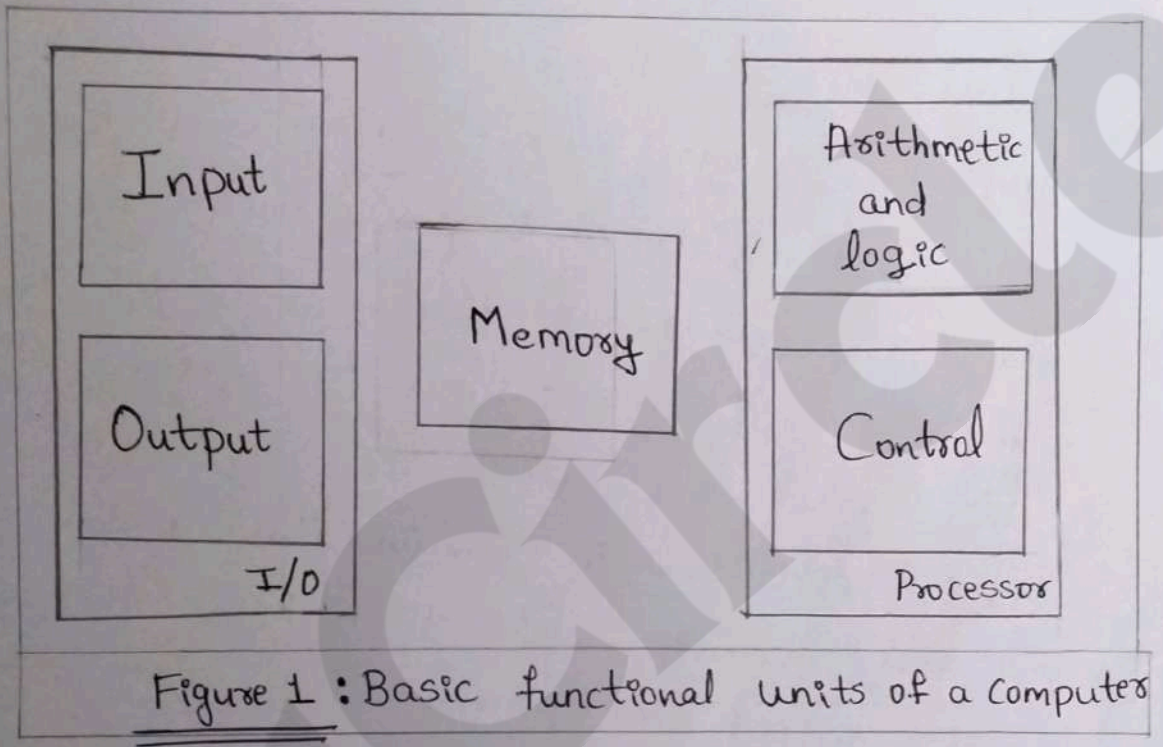


Figure 1 : Basic functional units of a computer

A computer consists of five functionally independent main parts: input, memory, arithmetic and logic, output, and control units, as shown in Figure ①. The input unit accepts coded information from human operators, from electromechanical devices such as key-boards, or from other computers over digital communication lines. The information received is either stored in the computer's memory for later

reference or immediately used by the arithmetic and logic circuitry to perform the desired operations. The processing steps are determined by a program stored in the memory. Finally, the results are sent back to the outside world through the output unit. All of these actions are coordinated by the control unit. Figure ① does not show the connections among the functional units. These connections, which can be made in several ways, are discussed throughout this book. We refer to the arithmetic and logic circuits, in conjunction with the main control circuits, as the processor, and input and output equipment is often collectively referred to as the input - output (I/O) unit.

We now take a closer look at the information handled by a computer. It is convenient to categorize this information as either instructions or data. Instructions or machine instructions, are explicit commands that

- govern the transfer of information within a computer as well as between the computer and its I/O devices.

- Specify the arithmetic and logic operations to be performed.

A list of instructions that performs a task is called 'program'. Usually the program is stored in the memory. The processor then fetches the instructions that make up the program from the memory, one after another, and performs the desired operations. The computer is completely controlled by the stored program, except for possible external interruption by an operator or by I/o devices connected to the machine.

I) Input Unit:-

Computers accept coded information through input units, which read the data. The most well-known input device is the keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted over a cable to either the memory or the processor.

Many other kinds of input devices are available, including joysticks, trackballs, and mouses. These are often used as graphi-c input devices in conjunction with displays.

Microphones can be used to capture audio input which is then sampled and converted into digital codes for storage and processing.

II) Memory Unit:-

The function of the memory unit is to store programs and data. There are two classes of storage, called primary and secondary.

Primary storage is a fast memory that operates at electronic speeds. Programs must be stored in the memory while they are being executed. The memory contains a large number of semiconductor storage cells, each capable of storing one bit of information. These cells are rarely read or written as individual cells but instead are processed in groups of fixed size called words. The memory is organized so that the contents of one word, containing n bits, can be stored or retrived in one basic operation.

The number of bits in each word is often referred to as the word length of the computer. Typical word lengths range from 16 to 64 bits. The capacity of the memory is one factor that characterizes the size of a computer. Small machines typically have only a few tens of millions of words, whereas medium and large

machines normally have many tens or hundreds of millions of words. Data are usually processed within a machine in units of words, multiple of words, or parts of words. When the memory is accessed, usually only one word of data is read or written.

Although primary storage is essential, it tends to be expensive. Thus additional, cheaper, secondary storage is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently. A wide selection of secondary storage devices is available, including magnetic disks and tapes and optical disks (CD-ROMs).

III) Arithmetic And Logic Unit :-

Most computer operations are executed in the arithmetic and logic unit (ALU) of the processor. Consider a typical example: Suppose two numbers located in the memory are to be added. They are brought into the processor, and the actual addition is carried out by the ALU. The sum may then be stored in the memory or retained in the processor for immediate use.

Any other arithmetic or logic operation, for example, multiplication, division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU. When operands are brought into the processor, they are stored in high-speed storage elements called 'registers'. Each register can store one word of data. Access times to registers are somewhat faster than access times to the fastest cache unit in the memory hierarchy.

To control & the arithmetic and logic units 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 keyboards, displays, magnetic and optical disks, sensors, and mechanical controllers.

IV) Output Unit:-

The output unit is the counterpart of the input unit. Its function is to send processed results to the outside world. The most familiar example of such a device is a printer. Printers employ mechanical impact heads, ink jet streams, or photocopying techniques, as in laser printers, to perform the printing. It is possible to produce printers capable of printing as many as 10,000 lines per minute. This is tremendous speed for a

mechanical device but is still very slow compared to the electronic speed of a process -or unit. Some units, such as graphic displays provide both an output function and an input function. The dual role of such units is the reason for using the single name I/o unit in many cases.

v) Control Unit :-

The memory, arithmetic and logic, and input and output units store and process information and perform input and output operations. The operation of these units must be coordinated in some way. This is the task of the control unit. The control unit is effectively the nerve center that sends control signals to other units and senses their states.

The operation of a computer can be summa -rized as follows:

- The computer accepts information in the form of programs and data through an input unit and stores it in the memory.
- Information stored in the memory is fetched, under program control, into an arithmetic and logic unit, where it is processed.
- Processed information leaves the computer through an output unit.
- All activities inside the machine are directed by the control unit.

Basic Operational Concepts

To perform a given task, an appropri-ate program consisting of a list of instruct-ions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be used as operands are also stored in the memory. A typical instruction may be

```
Add LOCA, R0
```

This instruction adds the operand at memory location LOCA to the operand in a register in the processor, R0, and places the sum into register R0. The original contents of location LOCA are preserved, whereas those of R0 are overwritten. This instruction requires the performance of several steps. First, the instruction is fectched from the memory into processor. Next, the operand at LOCA is fetched and added to the contents of R0. Finally the resulting sum is stored in register R0.

The effect of the above instructio-n can be realized by the two-instruction sequence

```
Load LOCA, R1
Add R1, R0
```

The first of these instructions transfers the contents of memory location LOCA into processor register R1, and the second instruction adds the contents of registers R1 and R0 and places the sum into R0. Note that this destroys the former contents of register R1 as well as those of R0, whereas the original contents of memory location LOCA are preserved.

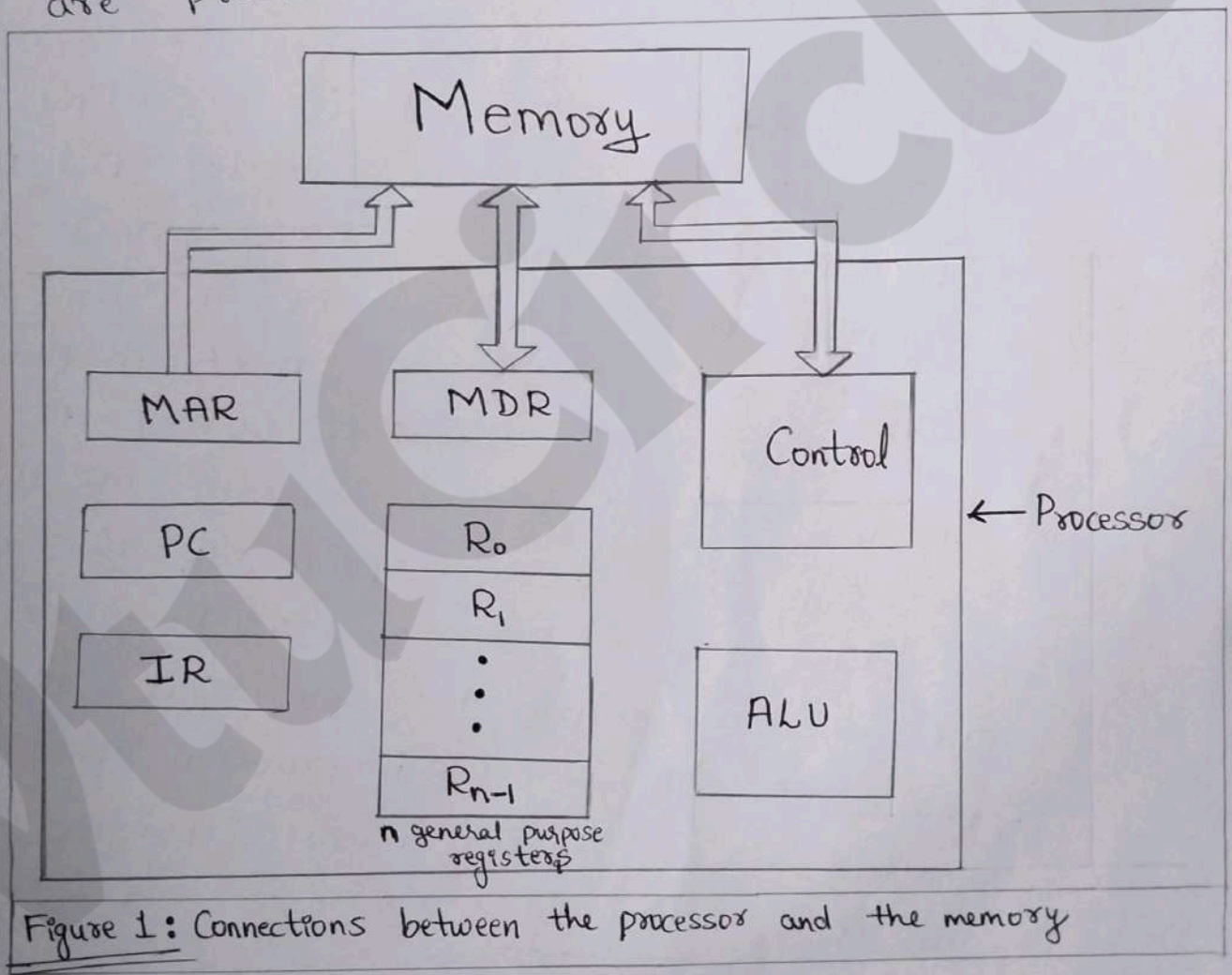


Figure 1: Connections between the processor and the memory

Figure ① shows how the memory and the processor can be connected. It also shows a few essential operational details of the processor that have not been discussed yet. The interconnection pattern for these components is not shown explicity

since here we discuss only their functional characteristics.

In addition to the ALU and the control circuitry, the processor contains a number of registers used for several diff-ent purposes. The instruction register (IR) holds the instruction that is currently being executed. Its output is available to the control circuits, which generate the timing signals that control the various proce-ssing elements involved in executing the instruction. The program counter (PC) is another specialized register. It keeps track of the execution of a program. It contains the memory address of the next instruction to be fetched and executed. During the execution of an instruction, the contents of the PC are updated to correspond to the address of the next instruction to be executed. It is customary to say that the PC points to the next instruction that is to be fetched from the memory. Besides the IR and PC, Figure ① shows n general-purpose registers, R_0 through R_{n-1}.

Finally, two registers facilitate communication with the memory. These are the memory address register (MAR) and the memory data register (MDR). The MAR holds the address of the location to be accessed.

The MDR contains the data to be written into or read out of the addressed location.

Let us now consider some typical operating steps. Programs reside in the memory and usually get there through the input unit. Execution of the program starts when the PC is set to point to the first instruction of the program. The contents of the PC are transferred to the MAR and a Read control signal is sent to the memory. After the time required to access the memory elapses, the addressed word (in this case, the first instruction of the program) is read out of the memory and loaded into the MDR. Next, the contents of the MDR are transferred to the IR. At this point, the instruction is ready to be decoded and executed.

Bus Structures :-

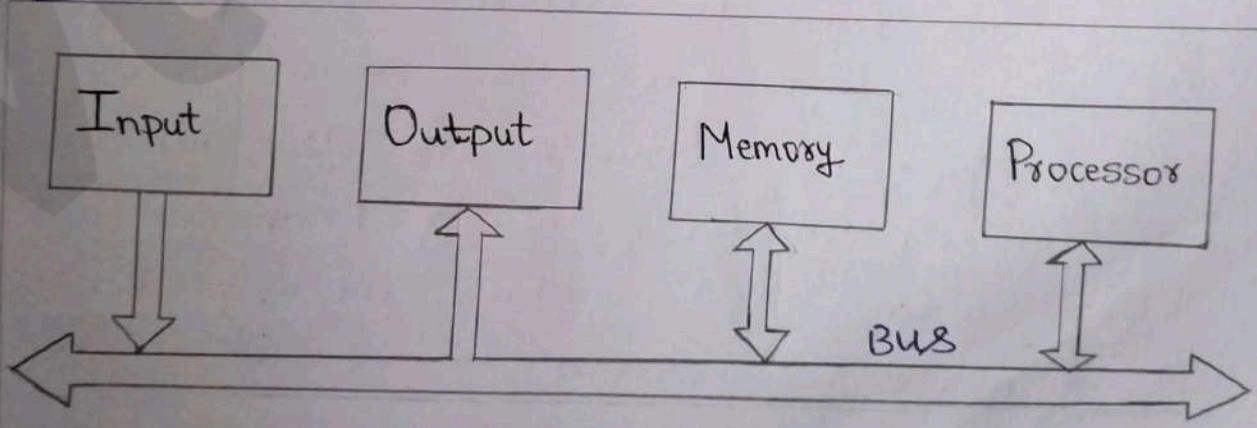


Figure 1 : Single-bus structure

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. When a word of data is transferred between units, all its bits are transferred in parallel, that is, the bits are transferred simultaneously over many wires, or lines, one bit per line. A group of lines that serves as a connecting path 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 purposes.

The simplest way to interconnect functional units is to use a single bus, as shown in Figure ①. All units are connected to this bus. Because 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 the bus. The main virtue of the single-bus structure is its low cost and its flexibility for attaching peripheral devices. Systems that contain multiple buses achieve more concurrency in operations by allowing two or more transfers to be carried out at the same time. This leads to better performance but at an increased cost.

Performance:-

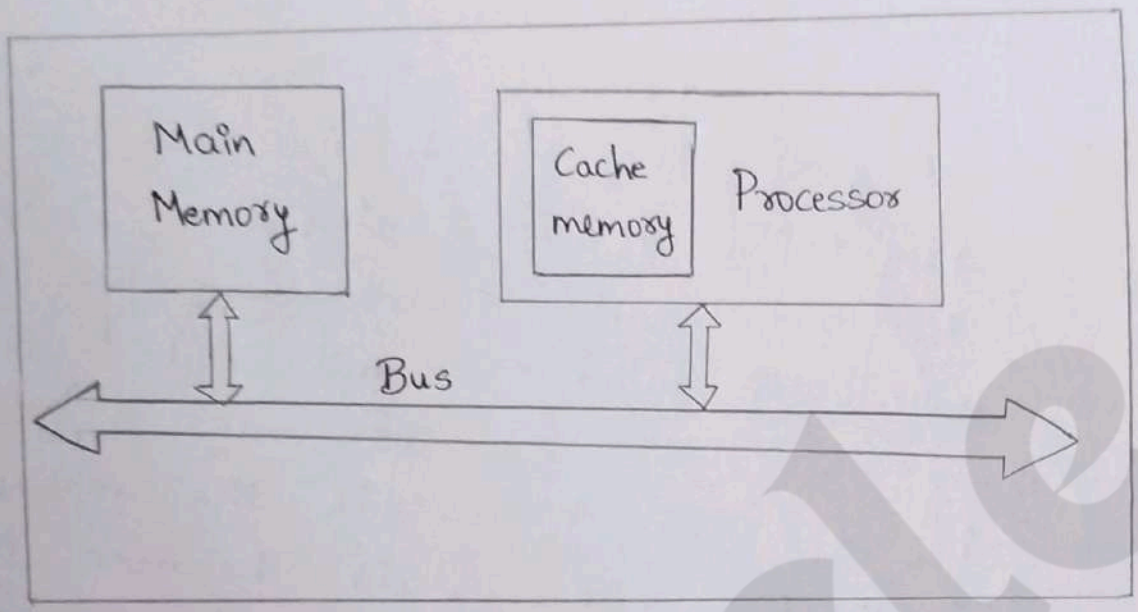


Figure 1: The Processor Cache

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 execution proceeds, instructions are fetched one by one over the bus into the processor, and a copy is placed in the cache. When the execution of an instruction calls for data located in the main memory, the data are fetched 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 a relatively small cache memory can be fabricated on a single integrated circuit chip. The internal speed of performing the basic steps of instruction processing on such chips is very high and is considerably faster than the speed at which instructions 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 happen 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 data that are used repeatedly.

Bus is defined as set of parallel wires used for data communication between different parts of computer. Each wire carries 1 bit of data. There are 3 types of buses, namely

1. Address bus
2. Data bus and
3. Control bus

1. Address bus :-

- It is unidirectional.
- The processor (CPU) sends the address of an I/O device or memory device by means of this bus.

2. Data bus :-

- It is bidirectional bus.
- The CPU sends data from memory to CPU and vice-versa as well as from I/O to CPU and vice-versa by means of this bus.

3. Control bus :-

- This bus carries control signals for memory and I/O devices. It generates control signals for memory namely MEMRD and MEMWR and control signals for I/O devices namely IORD and IOWR.

Features of Single bus organization are

* less Expensive
* Flexible to connect I/O devices.
* Poor Performance due to single bus.

There is a variation in the devices connected to this bus in terms of speed of operation. Few devices like Keyboard, are very slow. Devices like optical disk are faster. Memory and Processor are

faster. but all these devices uses the same bus. Hence to provide the synchroni-zation between two devices, a buffer register is attached to each devices. It holds the data temporarily during the data transfer between two devices.

Processor clock:-

Processor circuits are controlled by a timing signal called a 'clock'. The clock defines regular time intervals, called 'clock cycles'. To execute a machine instruction, the Processor divides the action to be performe-d into a sequence of basic steps, such that each step can be completed in one clock cycle. The length P of one clock cycle is an important Parameter that affects Processor Performance. Its inverse is the clock rate, $\boxed{R = 1/P}$, which is measured in cycles per second. Processors used in today's personal computers and workstations have clock rates that range from a few hundread million to over a billion cycles Per second. In standard electrical engineering terminology, the term "cycles per second" is called hertz (Hz). The term "million" is deno-ted by the prefix Mega (M), and "billion" is denoted by the prefix Giga (G). Hence 500 million cycles per second is usually

abbreviated to 500 Megahertz (MHz), and 1250 million cycles per second is abbreviated to 1.25 Gigahertz (GHz). The corresponding clock Periods are 2 and 0.8 nanoseconds (ns), respectively.

Basic Performance Equation:-

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 language instructions. The number N is the actual number of instruction executions, and is not necessarily equal to the number of machine instructions in the object program. Some instructions may be executed more than once, which is the case for instructions inside a program loop. Others may not be executed at all, depending on the input data used. Suppose that the average number of basic steps needed to execute one machine instruction is S, where each basic step is completed in one clock cycle. If the clock-rate is R cycles per second, the program execution time is,

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

This is often referred to as the basic performance equation.

where 'T' be total time required to execute the program

'N' be the number of instructions contained in the program.

'S' be the average number of steps required to execute one instruction.

'R' be number of clock cycles per second generated by the processor to execute one program.

Clock Rate :-

There are two possibilities for increasing the clock rate, R. First, improving the integrated circuit (IC) technology makes logic circuits faster, which reduces the time needed to complete a basic step. This allows the clock period, P, to be reduced and the clock rate, R, to be increased. Second, 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.

Increases in the value of R that are entirely caused by improvements in IC techno-logy affect 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 expected from the use of faster technology can be realize. The value of T will be reduced by the same factor as R is increased because S and N are not affected. The impact on performance of changing the way in which instructions are divided into basic steps is more difficult to assess.

Performance Measurement:-

It is important to be able to access the performance of a computer. Computer designers use performance estimates to evaluate the effectiveness of new features. Manufacturers use performance indicators in the marketing process. Buyers use such data to choose among many available computer models.

The previous discussion suggests that the only parameter that properly describes the performance of a computer is the execution time, T, for the programs of interest. Despite the conceptual simplicity of eq. $T = \frac{N * S}{R}$

Computing the value of T is not simple. Moreover, Parameters such as the clock speed and various architectural features are not reliable indicators of the expected performance.

For these reasons, the computer community adopted the idea of measuring computer performance using benchmark Programs. To make comparisons possible, Stan-dardized programs must be used. The Performance measure is the time it takes a computer to execute a given benchmark. Initially, some attempt were made to create artificial programs that could be used as Standard benchmarks. But, synthetic programs do not properly predict performance obtained when real application programs are run.

The accepted practice today is to use an agreed-upon selection of real application programs to evaluate performance. A nonprofit organization called System Performance Evalua-tion Corporation (SPEC) selects and publishes representative application programs for different application domains, together with test results for many commercially available computers.

The SPEC rating is computed as follows

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

Thus a SPEC rating of 50 means that the computer under test is 50 times as fast as the UltraSPARC 10 for this particular benchmark. The test is repeated for all the programs in the SPEC suite, and the geometric mean of the results is computed. Let $SPEC_i$ be the rating for program i in the suite. The overall SPEC rating for the computer is given by

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

where n is the number of programs in the suite.

Because the actual execution time is measured, the SPEC rating is a measure of the combined effect of all factors affecting performance, including the compiler, the operating system, the processor and memory of the computer being tested.

Memory Locations and Addresses

Numbers and character operands, as well as instruct-ions, are stored in the Memory of a computer We will now consider how the memory is organ-ized. 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 can be stored or retrieved in a Single, basic opeation. 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 1.

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, as shown in figure 2. A unit representation of 8 bits is called a byte. Machine instructions May require one or more words for their representation.

Byte Addressability

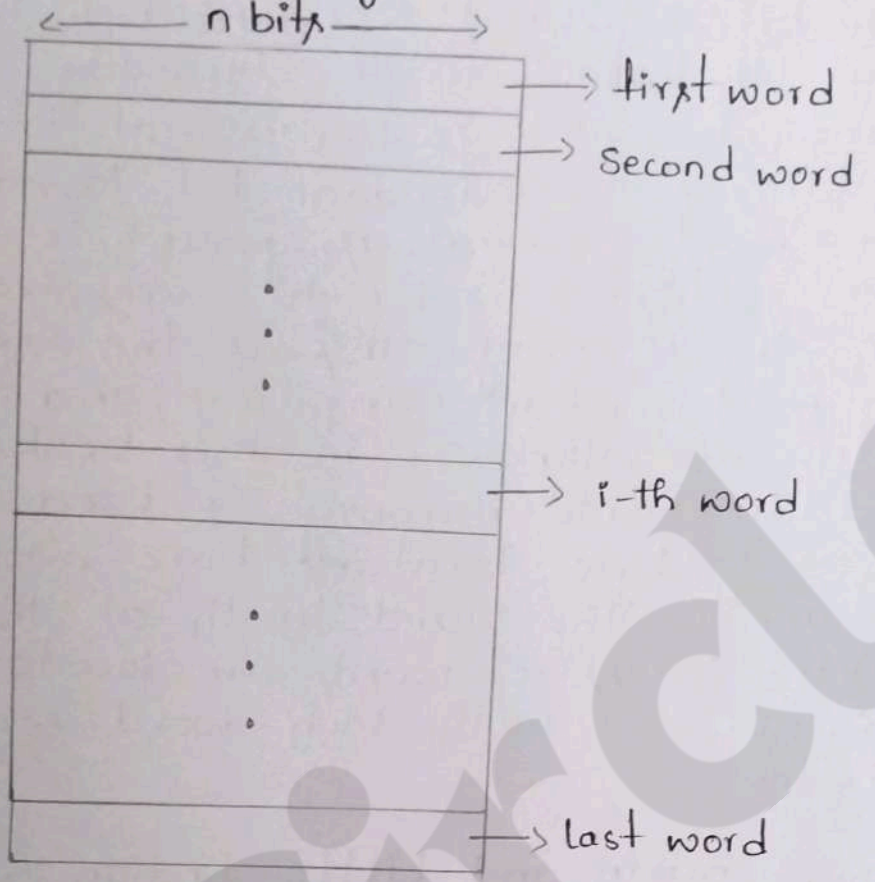
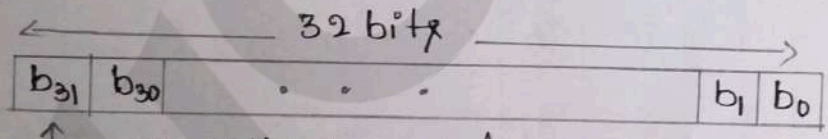
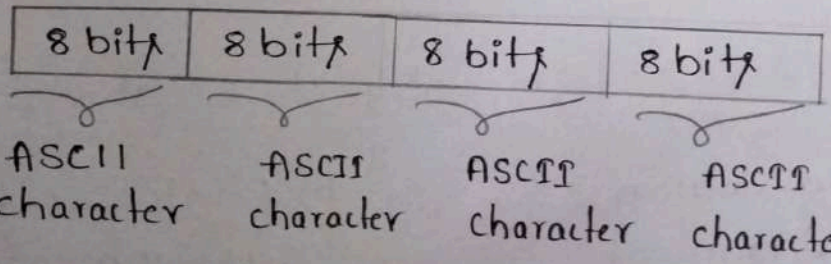


figure 1: Memory words.



Sign bit: $b_{31} = 0$ for positive numbers
$b_{31} = 1$ for negative numbers

(a) A Signed integer



(b) Four character

figure 2: Example of encoded information in a 32-bit word

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. It is impractical to assign distinct addresses to individual bit locations in the memory. The most practial assignment is to have Successive -ve addresses refer to Successive byte 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 used for this assig -ment. 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 Assignments

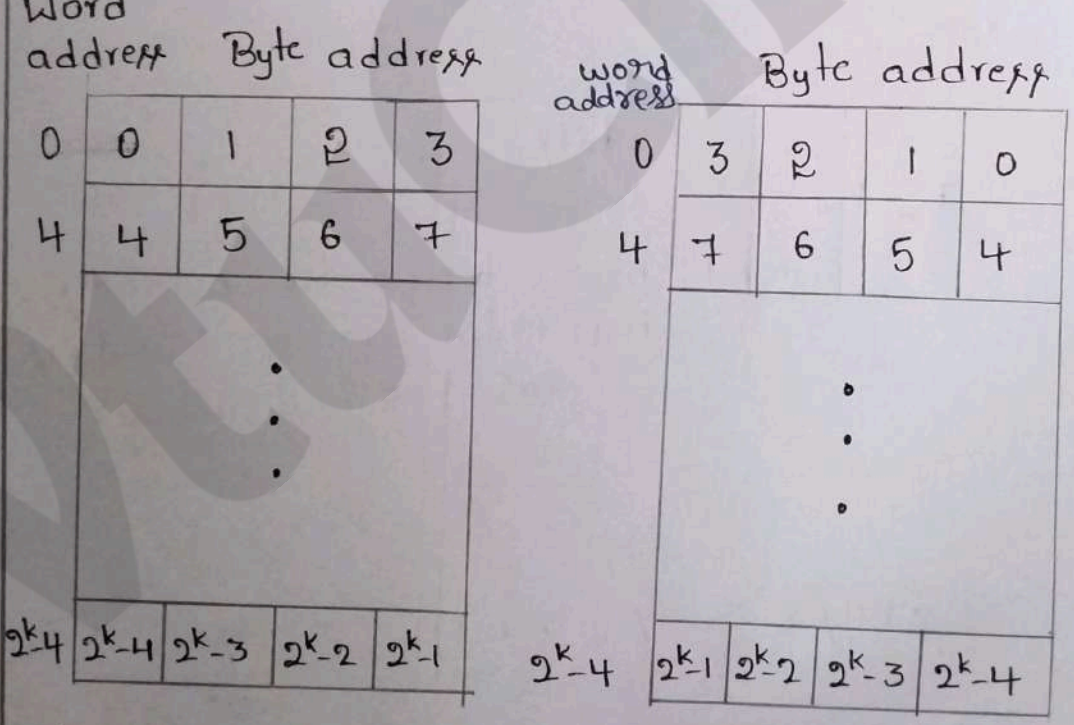


Figure 3- Byte and word addressing

There are two ways that byte addresses can be assigned across words, as shown in figure 3. 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 words. "more Significant" and "less Significant" are used in relation to the weights (powers of 2) assigned to bits when the word represents a number. Both Little-endian and big-endian assignments are used in Commercial machines. In both cases byte addresses 0, 4, 8, ..., are taken as the addresses of Successive words in the memory and are the addresses used when Specifying memory read and write operations for words.

In addition to Specifying the address ordering of bytes within a word, it is also necessary to specify the labeling of bits within a bytes or a word. The most common convention, and the one we will use in this book, is Show in Figure 1.a It is the most natural ordering for the encoding of numerical data. The Same ordering is also used for labeling bits within a byte, that is $b_7, b_6 \cdots b_0$, from left to right. There are Computers, however, that use the reverse ~~ob~~ ordering.

Eg: Store a word "JOHNSENA" in memory starting from word 1000, using Big Endian and Little endian.

Bigendian-

1000	J 1000	O 1001	H 1002	N 1003
1004	S 1004	E 1005	N 1006	A 1007

Little endian-

1000	N 1000	H 1001	O 1002	J 1003
1004	A 1004	N 1005	E 1006	S 1007

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.

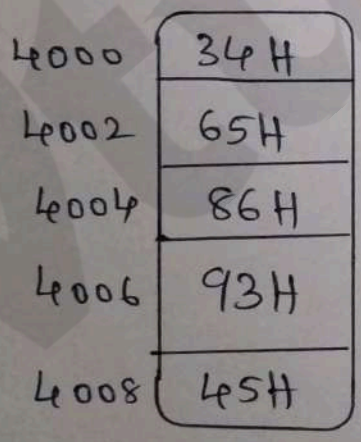
In Many applications, it is necessary to handle character string of variable length. The biginning of the string is indicated by giving the address of the byte containing its first character. Successive byte locations contain successive characters of the string. there are two ways to indicate the length 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 siparate memory word location or process-or register can contain a number indicating the length of the string in bytes.

WORD ALIGNMENT

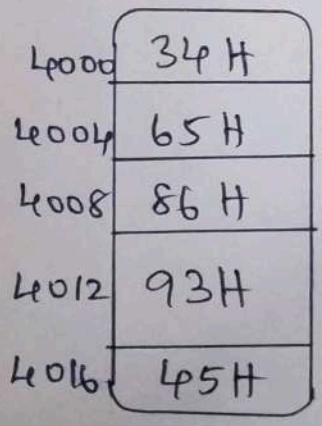
In the case of a 32-bit word length, natural word boundaries occur at addresses 0, 4, 8, ---, as shown in figure 3 We say that the word locations have aligned addresses In General, words are said to be aligned in memory if they bigin at a byte address that is a Multiple of the number of bytes in a word. for practical reasons associ-iated with Manipulating binary-coded addresses, the number of bytes in a word is a power of 2. Hence, if the word length is 16 (2 bytes). aligned words bigin at byte addresses 0, 2, 4--, and for a word length of 64 (2^3 bytes), aligned Words begin at byte addresses 0, 8, 16, ----.

The structure of memory for 16-bit CPU, 32-bit CPU and 64-bit CPU are as shown in the figures.

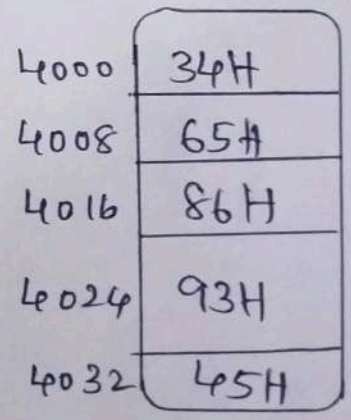
For 16-bit CPU



For 32-bit CPU



For 64-bit CPU



Memory Operations :-

Both program instructions and data operands are stored in the memory. To execute an instruction the processor control circuits must cause the word (or words) containing the instruction to be transferred from the memory to the processor. Operands and results must also be moved between the memory and the processor. Thus, two basic operations involving the memory are needed, namely, Load (or Read or Fetch) and Store (or Write).

The load operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged. To start a Load operation, the processor sends the address of the desired location to the memory and requests that its contents be read. The memory reads the data stored at that address and sends them to the processor.

The store operation transfers an item of information from the processor to a specific memory location, destroying the former contents of that location. The processor sends the address of the desired location to the memory, together with the data to be written into that location.

Instructions and Instruction Sequencing:-

The tasks carried out by a computer progra-m 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:

1. Data transfers between the memory and the processor registers.
2. Arithmetic and logic operations on data
3. Program sequencing and control
4. I/o transfers.

Register Transfer Notation:- (RTN)

The transfer of information from one location in the computer to another. Possible 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. For example, names for the addresses of memory locations may be LOC, PLACE, A, VAR2; Processor register 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 brack-ets around the name of the location. Thus, the expression

$$R1 \leftarrow [LOC]$$

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

$$R3 \leftarrow [R_1] + [R_2]$$

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 Language Notation:-

We need 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 register R1 and R2 and replacing their sum in R3 can be specified by the assembly language statement

```
Add R1, R2, R3
```

Basic Instruction Types :-

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 three variables. Hence, the above high-level language statement requires the action

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

to take place in the computer. 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  Source 1, Source 2, Destination
```

If k bits are needed to specify the memory address of each operand, the encoded from of the above instruction must contain 3k bits for addressing purpose in addition

to the bits needed to denote the Add operation. For a modern processor with a 32-bit address space, a 3-address instruction is too large to fit in one word for a reasonable word length. Thus, a format that allows multiple words to be used for a single instruction would be needed to represent an instruction of this type.

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 type is

`Add A,B`

which performs the operation $B \leftarrow [A] + [B]$. When the sum is calculated, the result is sent to the memory and stored in location B, replacing the original contents of this location. This means that operand B is both a source and a destination.

A single two-address instruction cannot be used to solve our original problem, which is to add 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 operation C ← [B], leaving the contents of location B unchanged. The word "Move" is a misnomer here: it should be "Copy". However, this instruction name is deeply entrenched in computer nomenclature. The operation C ← [A] + [B] can now be performed by the two-instruction sequence

Move B, C
Add A, C

In all instructions given above, the source operands are specified first, followed by the destination. This order is used in the assembly language expressions for machine instructions in many computers. But there are also many computers in which the order of the source and destination operands is reversed. We will see examples of both orderings in chapter 3. It is unfortunate that no single convention has be adopted by all manufacturers. In fact, even for a particular computer, its assembly language may use a different order for different instructions. In this chapter, we will continue to give the source operands first.

we have defined three- and two-address instructions. But, even two-address instructions will not normally fit into one word for usual word lengths and address sizes. Another possibility is to have machine instructions that specify only one memory operand. when a second operand is needed, as in the case of an Add instruction, it is understood implicitly to be in a unique location. A processor register, usually called the accumulator, may be used for this purpose. Thus, the one-address instruction.

Add A

means the following: Add the contents of memory location A to the contents of the accumulator register and place the sum back into the accumulator. Let us also introduce the one-address instructions

Load A

and

Store A

The Load instruction copies the contents of memory location A into the accumulator, and the Store instruction copies the contents of the accumulator into memory location A. Using only one-address instructions, the operation $C \leftarrow [A]+[B]$ can be performed by executing the sequence of instructions

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

Note that the operand specified in the instruction may be a source or a destination, depending on the instruction. In the Load instruction, address A specifies the source operand, and the destination location, the accumulator, is implied. On the other hand, C denotes the destination location in the Store instruction, whereas the source, the accumulator, is implied.

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

```
Move  A, Ri
Move  B, Rj
Add   Ri, Rj
Move  Rj, C
```

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

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

Instruction Execution and Straight-Line Sequencing

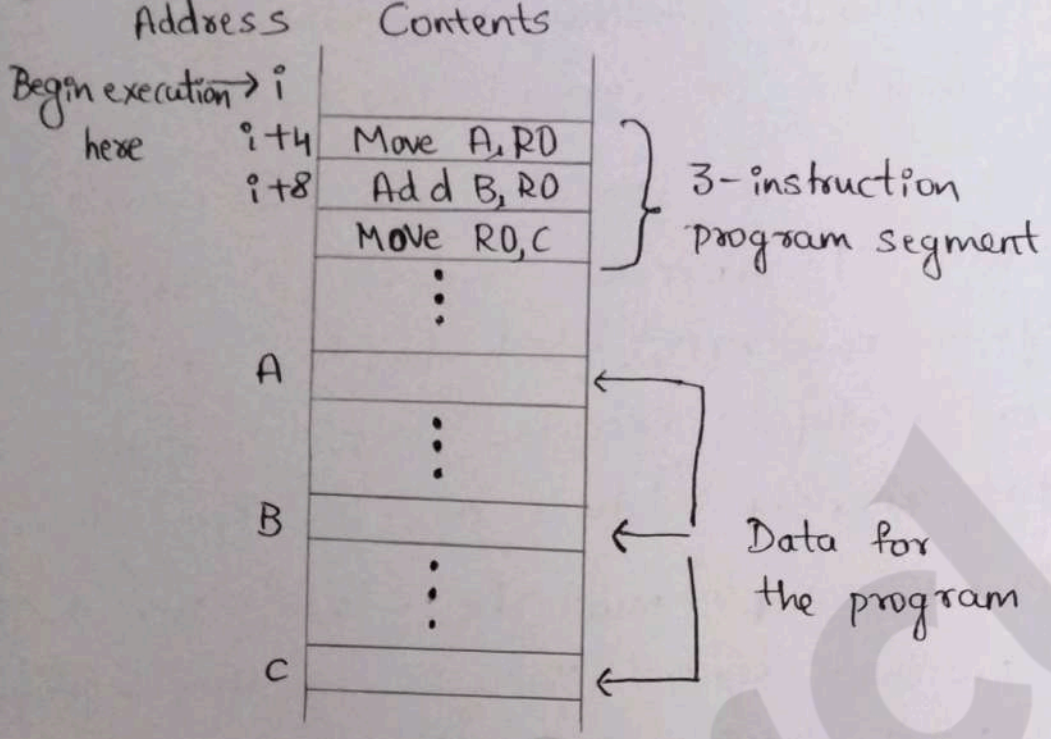


Figure ①: A program for C ← [A] + [B]

In the preceding discussion of instruction formats, we used the task C ← [A] + [B] for illustration. Figure ① 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 per instruction and has a number of processor registers. We assume that the word length is 32 bits and the memory is byte addressable. 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 i+8. For simplicity, we also assume that a full memory address can be directly

specified in a single-word instruction, although this is not usually possible for address space sizes and word lengths of current processors.

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. At the start of the second phase, called instruction execute, 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. At some point during this two-phase procedure, the contents of the PC are advanced to point to the next instruction. When the execute phase of an instruction is completed, the PC contains the address of the next instruction, and a new instruction fetch phase can begin. In most processors, the execute phase itself is divided into a small number of distinct phases corresponding to fetching operands, performing the operation, and storing the result.

Branching :

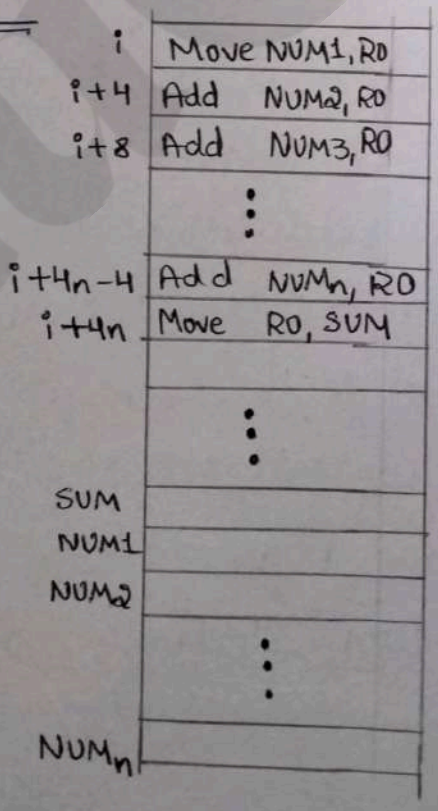


Figure ②: A straight-line program for adding n numbers

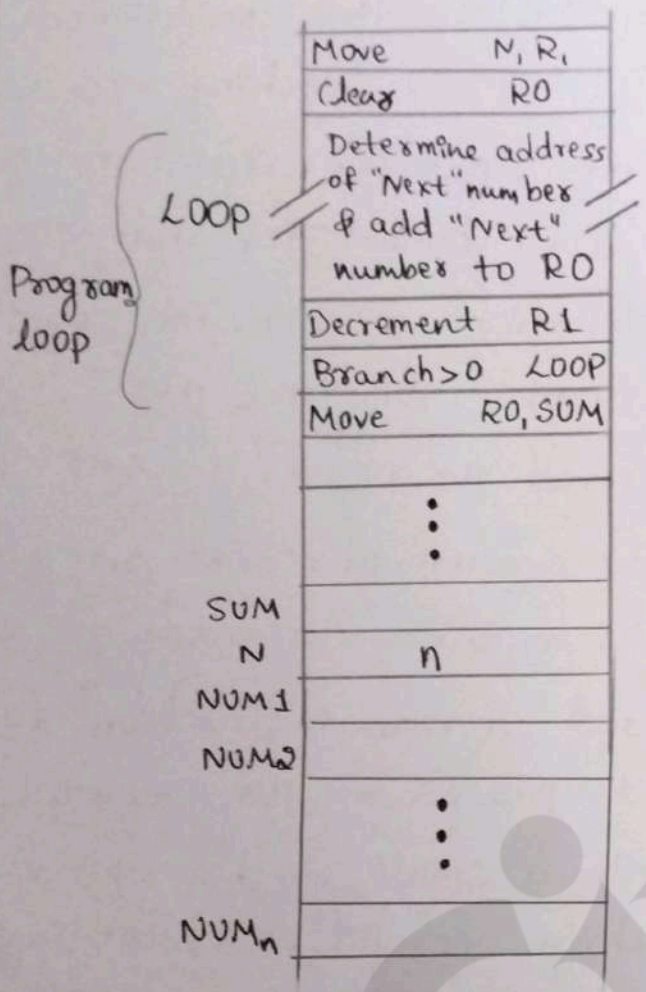


Figure ③ : Using a loop to add n numbers

Consider the task of adding a list of n numbers. The program outlined in figure ② is generalization of the program in figure ①. The addresses of the memory locations containing the n numbers are symbolically given as NUM1, NUM2, ---, NUMn, and a 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.

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 figure (3). 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 R0. The address of an operand can be specified in various ways, as will be described. ~~in~~ For now, we concentrate on how to create and control a program loop.

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 times 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. (A similar type of operation is performed by the Increment instruction, which adds 1 to its operand.) Execution of the loop is repeated as long as the result of the decrement operation is greater

than zero.

We now introduce branch instructions. 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, insted of the instruction at the location that follows the branch instruction is 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.

In the program in figure, the instruction

Branch >0 LOOP

(branch if greater than 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 than 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. Instead, the Move instruction is fetched and executed. It moves the

final result from R0 into memory location SUM.

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 ↘

1. N (negative) → Set to 1 if the result is negative; otherwise, cleared to 0
2. Z (zero) → Set to 1 if the result is 0; otherwise, cleared to 0
3. V (overflow) → Set to 1 arithmetic overflow occurs; otherwise, cleared to 0
4. C (carry) → Set to 1 if a carry-out results from the operation; otherwise, cleared to 0

Addressing Modes

We have now seen some simple examples of assembly language programs. 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 student's names, we can write them in a list. If we want to associate information with each name, for example to record telephone numbers or marks in various courses, we may organize this information in the form of a table. 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. When translating a high-level language program into assembly language, the compiler must be able to implement these constructs using the facilities provided in the instruction set of the computer in which the program will be run. The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes. In this section we present the most important addressing modes found in modern processors.

Table : Generic addressing modes

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

EA = effective address

Value = a signed number