UNIT 1

N. SIVA KUMAR
Dept Of CSE
MRCET

① Computer Types:

A Digital computer or simply computer is a fast electronic calculating machine that accepts digitized input information, processes it according to a list of internally stored instructions and produces the resulting output information.

* The list of instructions is called a computer program.
* The internal storage is called computer memory.

Many types of computers exist that differ widely in size, cost, computational power, and intended use.

The four basic types of computers are,

i) Personal computers.
ii) Work stations.
iii) Enterprise systems and servers
iv) Super computers.

i) Personal computers: The most common computer is a personal computer, which is used widely in homes, schools and business offices

* It is most common form of desktop computers.
* Desktop computers have processing and storage units, visual display, audio output units and a keyboard.
* The storage media includes hard disks, CD-ROM's, and diskettes.

ii) Work stations:

* Work-Stations is another type of a computer with high-resolution graphics, input/output capability.
* These Workstations have more computational power than personal computers.

* Because of these features, workstations are used in Engineering applications, especially for interactive design work.

iii) Enterprise Systems and Servers:

* Beyond workstations, a range of large and very powerful computer systems exist, which are called as enterprise systems and Servers.

* These Enterprise systems are used at low end of range.

* Enterprise systems are also called as mainframes.

* These Enterprise systems are used for business data processing in medium to large firms that require more computing power and storage capacity than workstations can provide.

* Servers contains the storage units which are capable of handling large volumes of requests to access the data.

* Servers are widely used in education, business and personal user communities.

* These Servers can handle thousands of requests and responses that are usually transported over internet.

iv) Supercomputers:

* Supercomputers have the capability of handling large scale numerical calculations.

* Thus, supercomputers are used in applications such as weather forecasting, aircraft design and simulations.

* These Supercomputers are used at the high end of range.

* These Supercomputers may have a number of separate or large functional units. Thus these occupies more space.

② Functional Units:

A computer consists of five functionally independent main parts called as Functional units.

These are as follows.

i) Input unit
ii) memory unit
iii) Arithmetic and logic unit
iv) output unit and
v) Control unit.

* The input unit accepts coded information from human operators from electromechanical devices such as keyboards, 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 circuit 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 and maintained by the control unit.
* The following figure shows the Basic functional units of a computer.
* It does not show any connections between functional units, because these connections can be made in several ways.

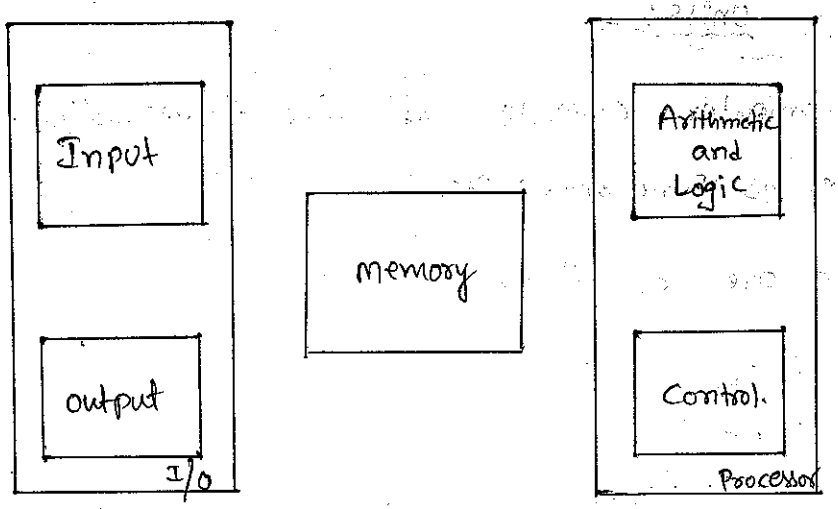


fig: Basic functional units of a computer.

Illustration of functional units:

i) Input unit:

* Computer accepts coded information through input units, which read the data.
* The most well-known input device is a keyboard.
* Whenever a key is pressed, the corresponding letter or digit is automatically translated over a cable to either the memory or the processor.
* Various other Input units include,
  - (a) Joysticks
  - (b) Track balls
  - (c) mouses
  - (d) microphones (used to capture audio input).

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:

* Primary storage is a fast memory that operates at electronic speeds.
* The memory contains a large number of semiconductor storage cells, each capable of storing one bit of information.
* These cells are read and written in groups of fixed size called words.
* To provide easy access to any word in the memory, a distinct address is associated with each word location.
* Address are numbers that identify successive locations.
* A given word is Accessed by specifying its address and issuing a control command that starts the storage or retrieval process.
* The Number of bits in each word is reffered as 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 the computer.
* 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 is called the memory Access time.

ⓑ Secondary Storage:

* AS primary is expensive, cheaper Secondary Storage

is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently.

* we have various secondary storage devices available, which includes magnetic disks, tapes and optical disks (CD-ROM's).

iii) Arithmetic and logic unit:

* most computer operations are executed in the Arithmetic and logic unit (ALU) of the processor.

* for Example,

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 memory or retained in the processor for immediate use.

* Any other Arithmetic and logic operations like, multiplication, division or comparision of numbers is initiated by bringing the required operands in to the processor, where the operation is performed by the ALU.

* when operands are brought in to the processor, they are stored in the highspeed storage elements called "Registers".

* The control and ALU are manytimes faster than other devices connected to a computer system.

Thus enables a single processor to control a number of External devices such as keyboards, displays, magnetic and optical disks.

iv) Output unit:

* The output unit's function is to send processed results to the outside world.

* most familiar example for output unit is a printer.

* It is possible to produce printers capable of printing as many as 10,000 lines per minute.

* Graphic displays provides both output function and input functions, that's why we call as I/O unit in some cases.

v) Control unit:

* The operations performed by memory, ALU, Input and output units must be coordinated in some way, this is the task of the control unit.

* The timing signals which transfers the instructions and data are generated by the control signals.

* Timing signals are signals that determine, when a given action is to take place.

* Data transfers between the processor and the memory are also controlled by the control unit through timing signals.

→ The operation of a computer can be summarized as follows:

① The computer accepts information in the form of programs and data through an input unit and stores it in the memory.

② Information stored in the memory is fetched, under program control, in to an arithmetic and logic unit, where it is processed.

③ processed information leaves the computer through an output unit.

④ All Activities inside the machine are directed by the control unit.

③ Basic operational Concepts:

→ The activity in a computer is governed by instructions.

→ 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 in to 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

→ The above instruction adds the operand at memory location, LOCA to the operand in the register in the processor, R0 and Places the Sum in to Register R0.

→ The original Contents of location LOCA are preserved, where as the Contents of R0 are overwritten.

→ Steps involved in this procedure:

i) First, the instruction is fetched from the memory in to the processor.

ii) Next, the operand at LOCA is fetched and added to the contents of R0.

iii) Finally, the Resulting Sum is Stored in register R0.

→ For Performance reasons, the above instruction can be realized by the two-instruction sequence, as given below.

Load LOCA, R1

Add R1, R0

→ First instruction transfers the contents of memory location LOCA in to processor register R1.

→ The Second instruction adds the contents of Registers R1 and R0, and places the sum in to R0.

→ In this case, contents of LOCA are preserved, whereas the contents of Registers R1 and R0 are destroyed.

→ 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 & from the memory.

→ The below figure shows how the memory and the processor can be comected.

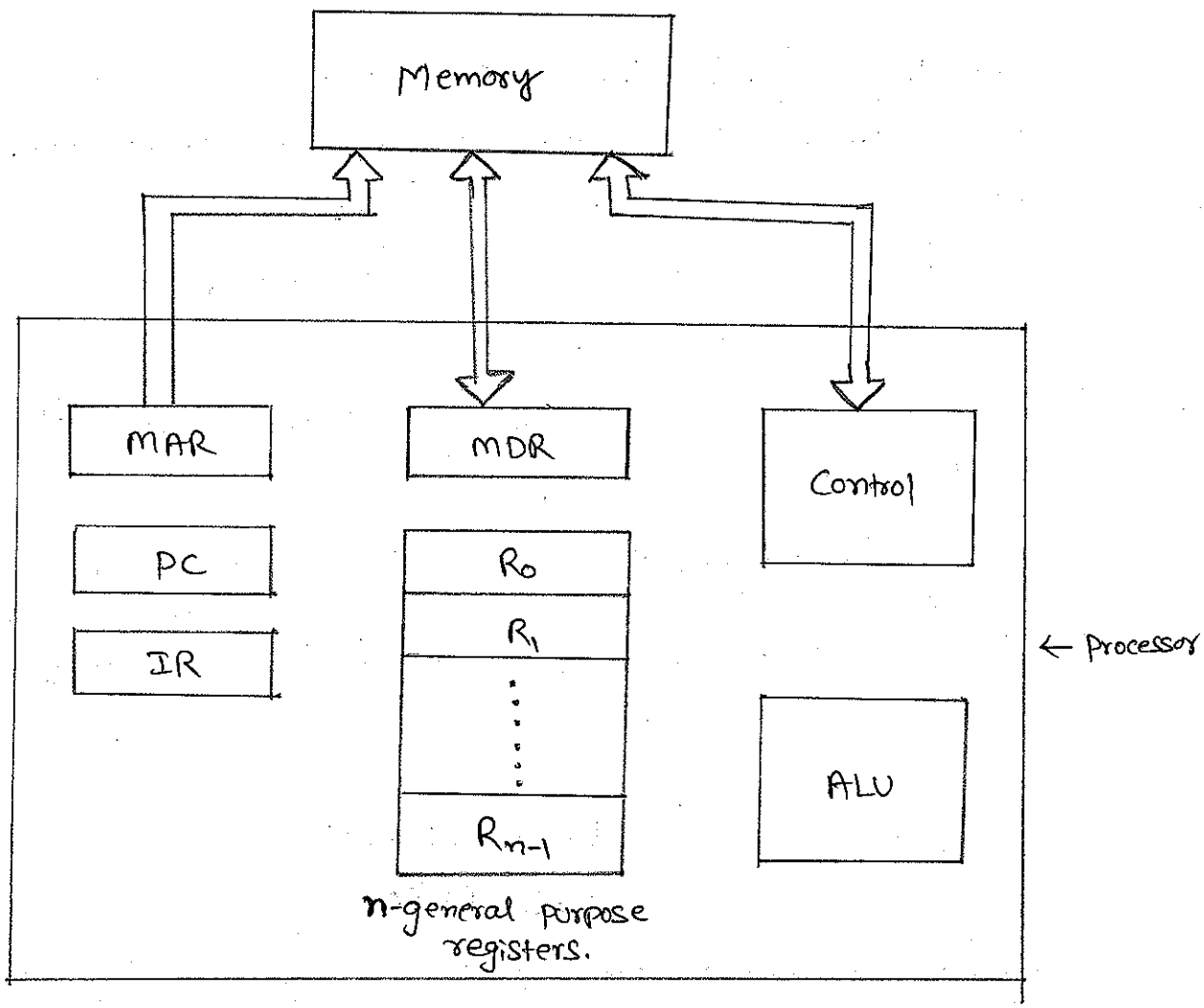


fig: Connections between the processor and the memory.

→ In addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes.

→ The Instruction Register (IR) holds the instruction that is currently being executed. Its output is available to the control circuits, which generates the timing signals that control the various processing elements involved in executing the instruction.

→ The program counter (PC), keeps track of the execution of a program. It contains the memory address of the next instruction to be fetched and executed.

→ Besides IR and PC, we have n-general purpose Registers.

→ Two registers communicate with the memory, MAR and MDR.

→ Memory address register (MAR) holds the address of the location to be accessed.

→ Memory Data register (MDR) contains the data to be written in to (or) read out of the addressed location.

<u>list of some operating steps</u>:

* 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 set to the memory.

* The addressed word (or) instruction is read out of the memory and loaded in to the MDR.

* Next the contents of MDR are transferred to IR.

* At this point, the instruction is ready to be decoded & executed.
* If the instruction involves an operation to be performed by the ALU, it is necessary to obtain the required operands.
* If the operand resides in the memory, it has to be fetched, by sending its address to MAR.
* When the operand has been read from the memory in to the MDR, it is transferred from the MDR to the ALU.
* After performing the desired operation, if the result is to be Stored in the memory, then the result is sent to MDR.
* At some point during the execution of the Current instruction, the contents of the PC are incremented so that the PC points to the next instruction to be executed.

(4) Bus Structure:

* To form an operational system, all the functional units of a computer must be connected in some organized way.
* This can be done in many ways. let's consider the simplest among them.
* A group of lines that serves as a connecting path for several devices is called a "bus".
* 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. i.e., the bits are transferred simultaneously over many wires & lines as one bit per line.
* The simplest way to interconnect all the functional units is to use a single bus as shown below.

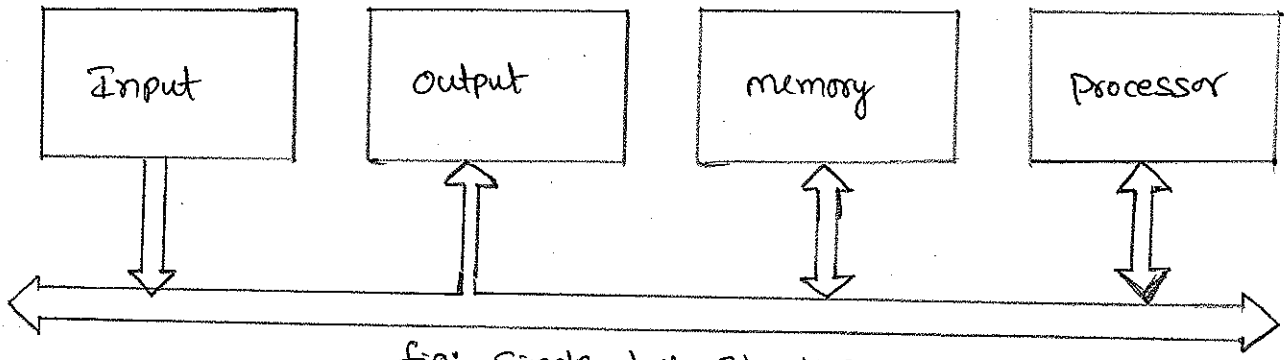


fig: Single-bus structure.

* All units are comected to the BUS.
* The main advantage of the Single-Bus structure is its low cost and its flexibility for attaching peripheral devices.
* Systems that contain multiple buses achieve more operations at the same time. This leads to better performance but at an increased cost.

* The devices connected to a Bus differs widely in their speed of operations.
* Keyboards and printers are relatively slow when compared to magnetic or optical disks.
* An efficient transfer mechanism is needed to reduce timing difference among processors, memories and external devices.
* A common approach is to include buffer registers with the devices to hold the information during transfers.
* for example, consider the transfer of an encoded character from a processor to a character printer.
* The processor sends the character over the bus to the printer buffer.
* Since the buffer is an electronic register, this transfer requires relatively little time.
* once the buffer is loaded, the printer can start printing, without further action by the processor.
* The bus and the processor are no longer needed and can be released for other activity.
* The printer continues printing the character in its buffer and is not available for further transfers until this process is completed.
* Thus buffer registers smooth out time differences among processors, memories and I/o devices.

Software

* In order to enter and run an application program, the computer must already contain some system software in its memory.

* System software is a collection of programs that are Executed as needed to perform functions such as

  i) Receiving and interpreting user commands.
  ii) Entering and Editing application programs and storing them as files in secondary storage devices.
  iii) managing the storage and retrieval of files in secondaly storage devices.
  iv) Running standard application programs.
  v) controlling I/O to receive input information and Produce output results etc.

* System software is thus responsible for the coordination of all activities in a computing system.

* Application programs are usually written in high level programming language, such as C, C++, Java or Fortran, in which the programmer specifies mathematical or text-processing operations.

* These operations are described in a format that is independent of the particular computer, used to Execute the program.

* The system software program called a compiler translates the high-level language program in to a suitable machine-language program containing instructions such as the Add and load instructions.

* Another important System program that all programmers use, is a text editor.

Examples for text editors: Emacs
Notepad developed by microsoft
Leafpad developed by Linux. etc.

* It is used for entering and editing application programs.
* Computer will have a key System software component called the operating System (OS).
* This is a large program, or actually a collection of routines that is used to control the interaction among various Computer units as they Execute application programs.
* In order to understand the basics of operating Systems, letus Consider a System with one processor, one disk and one printer.

Steps involved in running one Application program:

* Assume that the application program has been Compiled from a high level language form in to a machine language form and Stored on the disk.
* The first Step is to transfer the file in to memory.
* when the transfer is complete, Execution of the program is started
* Assume, program's task involves reading the data file from the disk to memory, perform computations and printing the results.

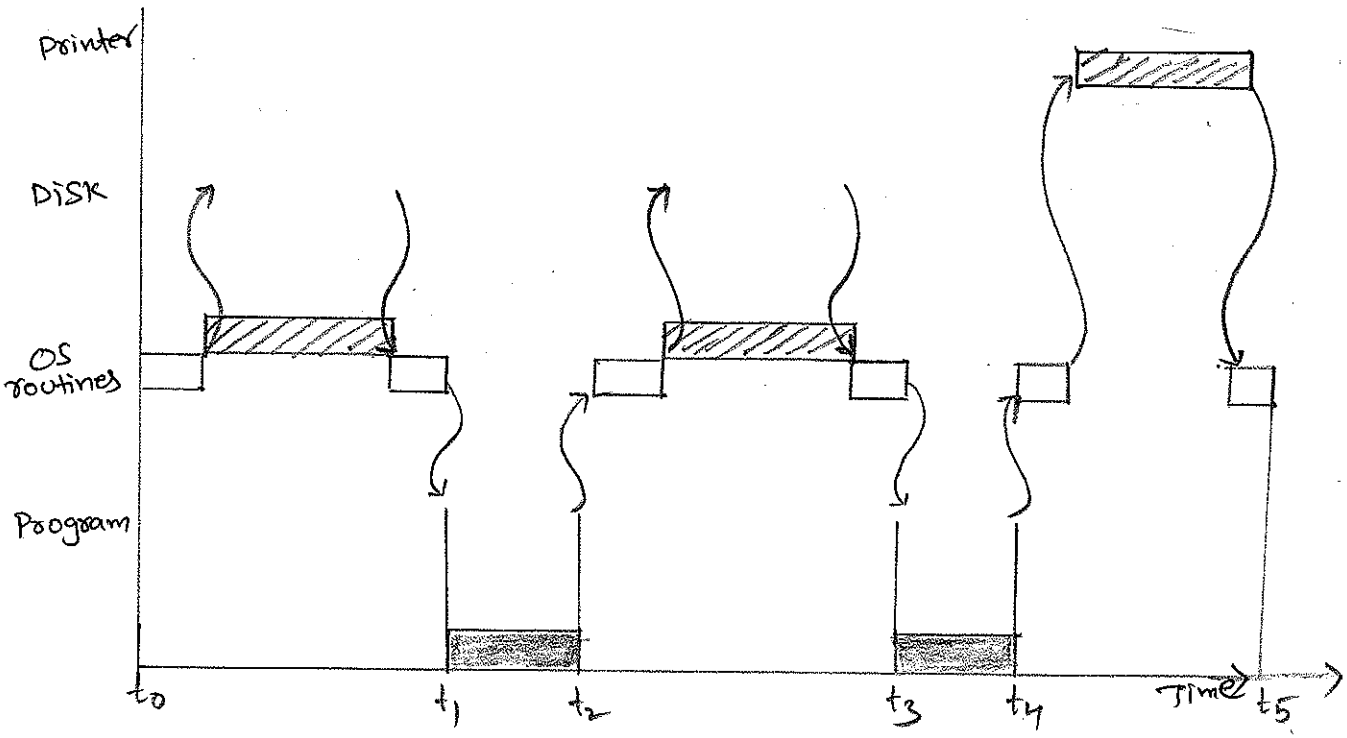
i) when the Execution process reaches the point where the data file is needed, the program requests the operating System to transfer the data file from the disk to the memory.

ii) The OS performs this task, and the application program Performs the required computations.

* when the computations are completed and the results are ready to be printed, the application program again sends a request to the operating system.

* An OS routine is then executed to cause the printer to print the results.

Illustration of sharing of the processor execution time by the time-line diagram:



* During the time period to to t_1, an OS routine initiates loading the application program from disk to memory, waits until the transfer is completed and then passes execution control to the application program.

* A similar pattern occurs during t_2 to t_3 and t_4 to t_5, when the OS transfers the data file from the disk and prints the results.

* At t_5, the operating system may load and execute another application program.

Running Several application programs.

* Notice that the disk and the printer are idle during most of the time period t_4 to t_5.
* The OS can load the Next program to be executed in to the memory from the disk while the printer is operating.
* Similarly, during to to t_1, the operating system can arrange to print the previous programs results while the current program is being loaded from the disk.
* This pattern of Concurrent Execution is called multiprogramming or multitasking.

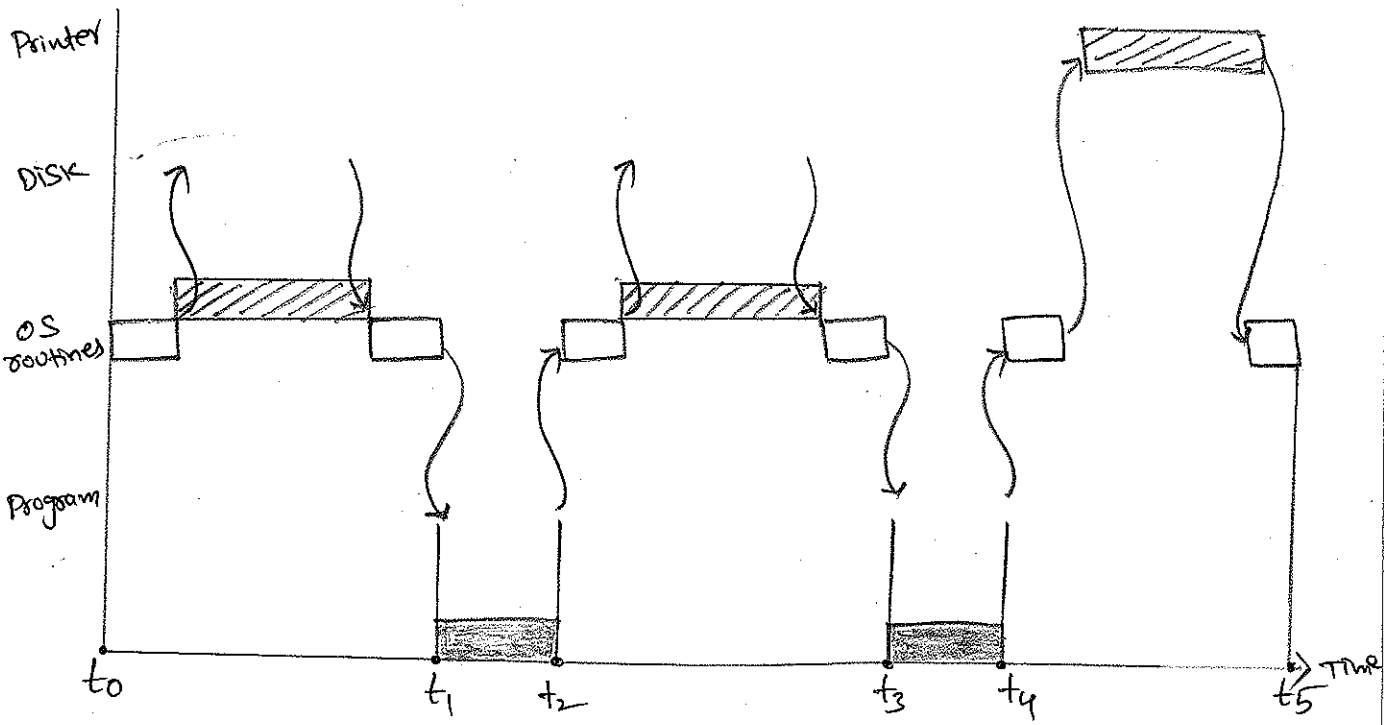
⑥ Performance:

* The most important measure of the performance of a computer is how quickly it can execute programs.

* The speed with which a computer executes programs is affected by the design of its hardware and its machine language instructions.

* Because programs are usually written in a high-level language, Performance is also affected by the compiler which translates Programs in to machine language.

* for Best performance, it is necessary to design the compiler, the machine instruction set, and the hardware in a coordinated way.



* As shown in the above diagram, the total time required to execute the program is $t_5 - t_0$. This elapsed time is a measure of the performance of the entire computer system.

* It is affected by the speed of the processor, disk and the printer.

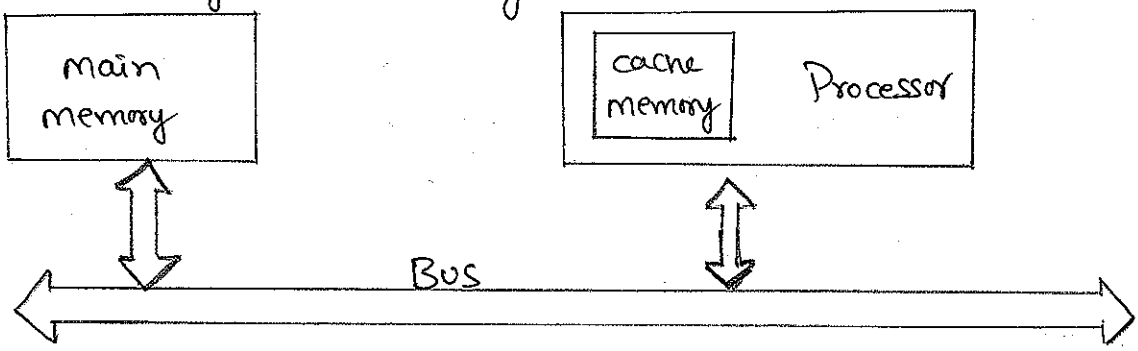
* TO discuss the performance of the processor, we should consider

only the periods during which the processor is active.

* Sum of these periods is reffered as processor time, needed to execute the program.

* 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 involves, the processor and the memory, which are usually connected by a bus.



* Let us see the flow of program instructions and data between the memory and processor.

* At the start of execution all program instructions and the required data are stored in the main memory.

* When execution starts, Instructions are fetched one by one over the bus in to the processor, and a copy is placed in the cache.

* Later if the same instruction or data item is needed for 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 such chip is very high and is faster than the speed at which instructions can be fetched from the main memory.

* A program will be executed faster, if the movement of

instructions or data between the memory and the processor is minimized, which is achieved by using cache.

Ex: loop instructions.

Various factors that illustrates the Performance of a Computer:

Processor clock:

* Processor circuits are controlled by a timing signal called Clock.
* The clock defines regular time intervals, called clock cycles.
* To execute a machine instruction, the processor divides the action to be performed in to 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, $R = 1/P$, which is measured in cycles per second.
* Processors used for personal computers, workstations will have Clockrates from a few hundred millions to billion cycles per second.
* In General terminology "cycles per second" is called Hertz (Hz)
* Therefore, 500 million cycles per second is written as 500 megahertz (MHz) Similarly, 1250 million cycles per second as 1.25 Gigahertz (GHz)

Basic performance equation:

* Let us focus 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 program execution time 'T' is given by,

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

where, N = No. of machine language instructions required for the execution of complete program.

* 'N' may not be equal to the No. of machine instructions in the object program, but it is actual no. of instructions executed.

* for example instructions may be executed more than once as in loops.

S = Average no. of basic steps needed to execute one machine instruction, where each basic step is completed in one clock cycle.

R = Clock rate.

* This equation is referred to as the Basic performance equation.
* Performance parameter 'T' is more important than individual N, S, & R.
* To Achieve high performance, the computer designer must find the ways to reduce the value of 'T' i.e, Reducing N & S and increasing R.
* 'N' is reduced, if the source program is compiled in to fewer machine instructions.
* 'S' is reduced, if instructions have a smaller number of basic steps (or) if the execution of instructions is overlapped.

Pipelining and Superscalar operation:

* Till now, we assumed that instructions are executed one after another. Hence the value of 'S' is the total no.of basic steps, or clock cycles.
* An improvement in performance can be achieved by overlapping the execution of successive instructions, using a technique called Pipelining.
* for example, consider the instruction.

Add R1, R2, R3.

which adds the contents of R1, R2 and places the sum in to R3. The contents of R1 and R2 are first transferred to the input of the ALU.

After the Add operation is performed, the sum is transferred to R3.

* The processor can read the next instruction from the memory while the addition operation is being performed.
* Then, if that instruction also uses the ALU, its operands can be transferred to the ALU inputs at the same time that the result of the Add instruction is being transferred to R3.
* In the ideal case, if all instructions are overlapped to the maximum degree possible, execution proceeds at the rate of one instruction completed in each clock cycle.

Then the effective value of 'S' is 1.

* But practically, the ideal value S=1 cannot be attained for a variety of reasons.

* However, pipelining increases the rate of executing instructions significantly and causes the effective value of S to approach 1.

* A higher degree of concurrency can be achieved, if multiple instruction pipelines are implemented in the processor.

  i.e., The multiple functional units are used, creating parallel paths through which different instructions can be executed in parallel.

* With this arrangement, it becomes possible to start the execution of several instructions in every clock cycle.

* This mode of operation is called Super scalar Execution.

* If it can be sustained for a long time during program execution, the effective value of 'S' can be reduced to less than one.

* Also parallel execution must preserve the logical correctness of programs, i.e, the results produced must be same as those produced by serial execution of program instructions.

* Many of todays high performance processors are designed to operate in this manner.

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'.
* In the presence of cache, the percentage of access to the main memory is small. Hence, much of the performance gain is expected.

Instruction set: CISC and RISC

* Simple instructions require a small number of basic steps to execute.
* Complex instructions involve a large number of steps.
* for a processor that has only simple instructions, a large no. of instructions may be needed to perform a given programming task. This leads to a large value of 'N' and small value for 'S'.
* on the other hand, if individual instructions perform more complex operations, fewer instructions will be needed, which leads to lower value of 'N' and a large value of 'S'.
* we cannot decide which choice is better than the other.
* A key consideration in comparing the two choices is the use of pipelining.

* processors with simple instructions are referred to "Reduced Instruction set computers (RISC)".
* processors with more complex instructions are referred to "Complex instruction set computers (CISC)".

Compiler

* A compiler translates a high-level language program into a sequence of machine instructions.
* An optimizing compiler can reduce the product N X S, which is the total number of clock cycles needed to execute a program.
* The no. of cycles is dependent not only on the choice of instructions, but also on the order in which they appear in the program.
* The compiler may rearrange program instructions to achieve better performance.
* The ultimate objective is to reduce the total number of clock cycles needed to perform a required programming task.

Performance measurement:

* we know that the only parameter that properly describes the performance of a computer is the Execution time, 'T'.
* But computing the value of 'T' is not simple, as the parameters such as clock speed and various architectural features are not reliable indicators of the expected performance.

* For this reasons, the Computer community adopted the idea of measuring computer performance using Benchmark programs.

* To make comparisions possible, standard programs called Benchmark programs must be used.

* The performance measure is the time taken by a computer to execute a given benchmark.

* An organization called System Performance Evaluation Corporation (SPEC) publishes representative application programs.

* The program is compiled for the computer under test, and the running time on a real computer is measured. The same program is also compiled and run on a computer selected for a reference.

* According to SPEC,

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

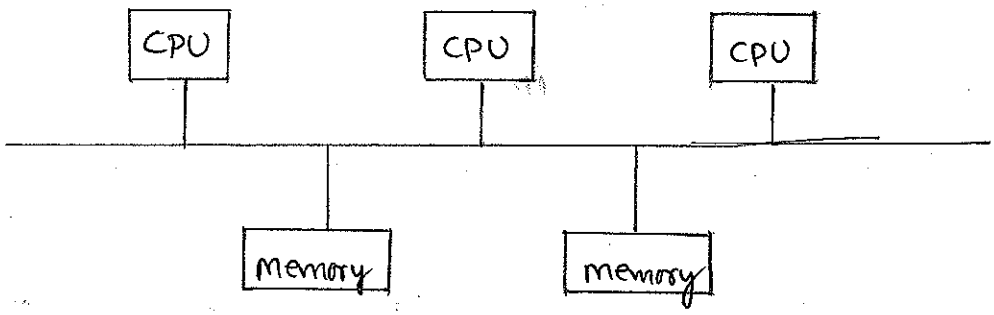
* overall SPEC Rating for a computer is given by,

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

where 'n' is the number of programs in the suit.

(7) Multiprocessors and Multicomputers:

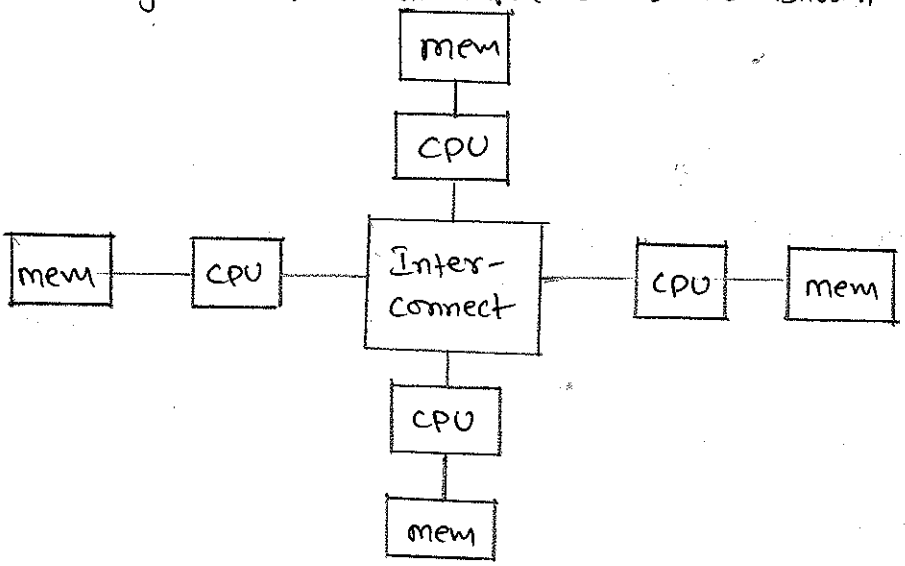
* Large computer systems may contain more number of processor units, called as multiprocessor systems.
* The Block diagram for multiprocessors is as shown below.



* These systems either execute a number of different application tasks in parallel, or they execute sub tasks of a single large task in parallel.
* In such systems, all processors usually have access to all of the memory units. and thus these systems are called as "Shared memory multiprocessor systems".
* use of such systems gives high performance but with much increased complexity and cost.
* In contrast to multiprocessor systems, it is also possible to use an interconnected group of complete computers to achieve high total computational power, which are known

as multicomputers.

* The Block diagram for multicomputers is as shown below.



* This computers normally have access only to their ocon memory units.

* TO communicate data, messages are exchanged over a Communication network.

* Thus these systems are known as "message - Passing multicomputers".

Data types: (Data Representation Basics)

* Binary information is stored in memory (or) processor registers.
* Registers contain either data (or) control information.
* Data types found in the registers of digital computers are classified in to following categories.
  i) Numbers used for arithmetic computations.
  ii) Letters of the alphabet for Data processing.
  iii) Discrete symbols used for specific purposes.

Radix: A number system of base (or) radix 'r' is a system that uses distinct symbols for 'r' digits.

Number Systems:

1. Binary
2. Decimal
3. Octal
4. Hexadecimal.

	Binary	Decimal	Octal	Hexadecimal.
Radix	Uses Radix-2	Uses Radix-10	Uses radix-8	Uses radix-16.
Symbols used.	0, 1	0, 1, 2, 3, ... 9	0, 1, 2, 3, ... 7	0, 1, 2, 3, ... 9, A, B, C, D, E, F.
example.	101101 $1\times2^5+0\times2^4+1\times2^3+1\times2^2+0\times2^1+1\times2^0$	724.5 $7\times10^2+2\times10^1+4\times10^0+5\times10^{-1}$	736.4 $7\times8^2+3\times8^1+6\times8^0+4\times8^{-1}$	F3 $15\times16^1+3\times16^0$
Decimal value.	$(45)_{10}$	$(724.5)_{10}$	$(478.5)_{10}$	$(243)_{10}$.

Complements:

9's complement

Ex: 546700

999999
546700

9's ⇒ 453299

10's complement

999 999
546 700

453299
1

10's ⇒ 453300

1's complement

Ex: 1011001

1's ⇒ 0100110

2's complement

0100110
1

2's ⇒ 0100111

(8) Data Representation:

* In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign.
* Because of the hardware limitations, computer must represent everything with 1's and 0's, including the sign of a number.
* So, to represent the sign, a bit placed in the left most position of the number.
* Generally the sign bit is equal to '0' for positive numbers and '1' for negative numbers.
* In addition to the sign, a number may have a binary (or decimal) point.
* The position of the binary point is needed to represent fractions, integers or mixed-integer fraction numbers.
* There are two ways of specifying the position of the binary point in a register.
    i) Fixed-Point Representation. and.
    ii) Floating-Point Representation.

Fixed-point Representation:

The fixed point method assumes that the binary point is always fixed in one position.

The two positions most widely used are,

i) A binary point in the extreme left of the registers to make the stored number a fraction. and

ii) A binary point in the extreme right of the register to make the stored number an integer.

Integer Representation:

* when an integer binary number is positive the sign is represented by '0' and the magnitude by a positive binary number.

* when the number is negative, the sign is represented by '1'. but the rest of the number may be represented in one of three possible ways.

i) Signed-magnitude representation.
ii) Signed - 1's Complement representation. and.
iii) signed - 2's Complement representation

i) Signed - magnitude representation :-

* In Signed magnitude representation, the Signed bit is inserted at the most Significant position. i.e, the '1' is inserted for negative numbers and '0' is inserted for the positive numbers at the most significant position.

*It Contains the magnitude of its normal binary value.

Example:

+14 [0] 0 0 0 1 1 1 0

-14 [1] 0 0 0 1 1 1 0

↓
Sign bit

ii) Signed 1's complement Representation:

* In this Signed 1's complement representation, the negative number is represented in 1's complement of its Positive Value. i.e., the most significant bit represents a negative number by '1'. and the magnitude of the number is written in 1's complement form.

* Consider the Signed number '-14' stored in 8-bit register.

* In Signed 1's Complement representation '-14' is written as follows.

Ex: +14 : 00001110

-14 : 1 1110001

Sign bit. 1's complement of '+14'

ii) Signed 2's Complement Representation:

* In this 2's complement representation, the negative numbers are represented in such a way that, its magnitude is reposesented by 2's complement of its positive value and the first bit represents the negative sign by '1'.

* Consider the Signed number '-14' stored in an 8-bit register.

* In Signed 2's Complement representation '-14' is written as follows.

Ex: -14 : 1 1110010

Sign bit. 2's complement of '+14' (00001110)

i.e, The only way to represent +14 is, 00001110.

* But a negative number (-14 in our example) can be represented in three different forms. i.e,

In Signed-magnitude representation, Signed 1's complement representation and Signed 2's complement representation.

Arithmetic addition

* The addition of two numbers in the Signed-magnitude System follows the rules of ordinary arithmetic.

* If the signs are same, we add the two magnitudes and give the common sign to the sum.

* If the signs are different, we subtract the smaller magnitude from the larger and give the result, the sign of the larger magnitude.

* for example,

$$(+25) + (-37)$$

$$\Rightarrow -(37-25)$$

$$\Rightarrow -12.$$

* This process requires comparision of the signs and the magnitudes and then performing either addition or subtraction.

* The rule for adding numbers in the Signed 2's Complement System does not require a comparision or Substraction. but only requires addition and complementation.

* The procedure is very simple, which is, add the two

numbers, including their sign bits and discard any carry out of the sign (left most) bit position.

* Numerical Examples for addition are shown below.

+6	00000110	−6	11111010
+13	00001101	+13	00001101
+19	00010011	+7	00000111

+6	00000110	−6	11111010
−13	11110011	−13	11110011
−7	11111001	−19	11101101

* Note that negative numbers must initially be in 2's Complement and if the sum obtained after the addition is negative, it is also in 2's complement form.
* In each of the four cases, the operation performed is always addition, including the sign bits.
* Any carry out of the sign bit position is discarded, and negative results are automatically in 2's complement form.

Arithmetic Subtraction:

* Subtraction of two signed binary numbers when negative numbers are in 2's complement form is very simple.
* The procedure is, take the 2's Complement of the Subtrahend and add it to the minuend.
* A carry out of the sign bit position is discarded.

* This procedure shows that a subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed.

* This is demonstrated by the following relationship

$$(\pm A) - (+B) = (\pm A) + (-B)$$
$$(\pm A) - (-B) = (\pm A) + (+B)$$

Overflow:

* When two numbers of 'n' digits each are added and the sum occupies 'n+1' digits, then that case is said to be an overflow occured.

* An overflow is a problem in digital computers. because the width of register is finite.

* A result that contains 'n+1' bits cannot be accomodated in a register with a standard length of 'n' bits.

* For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set which can be checked by the user.

* The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned.

* when two unsigned numbers are added, an overflow is

detected from the end carry out of the most significant position.

* When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.
* An overflow cannot occur after an addition, if one number is positive and the other is negative.

Decimal fixed point Representation:

* The representation of decimal numbers in registers is a function of the binary code used to represent a decimal digit.
* A 4-bit decimal code requires four flip-flops for each decimal digit.
* The Representation of 4385 in BCD requires 16 flip-flops, four flip-flops for each digit.
* The number will be represented in a register with 16 flip-flops as follows.

0100 0011 1000 0101

* By representing numbers in decimal we are wasting a considerable amount of storage space, since the no. of bits needed to store a decimal number in a binary code is greaterthan the number of bits needed for its equivalent binary representation.
* And also the circuits required to perform decimal arithmetic are more complex.
* However, there are some advantages in the use of decimal

representation because computer input and output data are generated by people who use the decimal system.

* Some applications, such as business data processing, require small amounts of arithmetic computations compared to the amount required for input and output of decimal data.

* For this reason, some computers and all electronic calculators perform arithmetic operations directly with the decimal data (in a binary code) and thus eliminate the need for conversion.

* The Representation of signed decimal numbers in BCD is similar to the representation of signed numbers in binary.

* we can either use the familiar signed-magnitude system or the signed-complement system.

* The sign of a decimal number is usually represented with four bits i.e, to represent plus, four 0's are used. and to represent minus, BCD equivalent of '9' is used, which is 1001.

* The signed magnitude system is difficult to use with computers.

* The signed complement system is either the 9's or 10's complement.

* The 10's complement is most often used system.

* To obtain the 10's complement, we first take the 9's complement and then add one to the least significant digit.

* Addition is done by adding all digits, including the sign digit, and discarding the end carry.

Ex: $(+375) + (-240) \Longrightarrow +135$, addition done in signed 10's complement system.

$$\begin{array}{r} 0375 \\ +\ 9760 \\ \hline 0135 \end{array} \quad \begin{array}{l} (0000\ 0011\ 0111\ 0101)_{BCD} \\ (1001\ 0111\ 0110\ 0000)_{BCD} \\ \hline (0000\ 0001\ 0011\ 0101)_{BCD} \end{array}$$

Floating Point Representation:

* The floating point representation of a number has two parts.
* The first part represents a signed, fixed-point number called the mantissa.
* The second part designates the position of the decimal (or binary) point and is called as the exponent.
* The fixed-point mantissa may be a fraction or an integer.

Example: The decimal number +6132.789 is represented in floating-point with a Fraction and an exponent as follows.

Fraction	Exponent
+0.6132789	+04

* The value of the exponent indicates that the actual position of the decimal point is four positions to the right of the indicated decimal point in the fraction.
* This representation is equivalent to the scientific notation,

$$+0.6132789 \times 10^{+4}$$

* Floating point is always interpreted to represent a numbers in the following form.

$$m \times r^{e}$$

where, m = mantissa
r = radix
e = exponent

* A Floating-point binary number is represented in a similar manner Except that it uses base 2 for the exponent.

* For example, the binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as follows.

Fraction	Exponent
01001110	000100

* The Fraction has a '0' in the leftmost position to denote positive.
* The exponent has the equivalent binary number +4.
* The floating-point number is equivalent to,

$$m \times 2^e \Longrightarrow +(.1001110)_2 \times 2^{+4}$$

* A Floating-point number is said to be normalized if the most significant digit of the mantissa is non zero.
* for example, decimal number 350 is normalized but, 000350 is not normalized.
* The number is normalized only if its leftmost digit is nonzero.
* For example, the 8-bit binary number 00011010 is not normalized because of the three leading 0's.
* The number can be normalized by shifting it three positions to the left and discarding the leading 0's to obtain 11010000.
* The three shifts multiply the number by $2^3 = 8$.
* To keep the same value for the floating-point number, the exponent must be subtracted by 3.
* Normalized numbers provide the maximum possible precision for the floating-point number.

Computer Arithmetic

Addition and Subtraction

→ Arithmetic instructions in digital computers manipulate data to produce results necessary for the solution of computational problems.

→ The four basic arithmetic operations are addition, subtraction, multiplication and division. From these basic operations, it is possible to formulate other arithmetic functions and solve scientific problems.

→ The solution to any problem that is stated by a finite number of well-defined procedural steps is called an algorithm.

→ Usually an algorithm will contain a number of procedural steps which are dependent on results of previous steps.

→ A convenient method for presenting algoritums is a flowchart.

→ we consider addition, subtraction, multiplication and division for the following types of data.

1. Fixed point binary data in signed magnitude representation.
2. Fixed point binary data in signed 2's complement Representation
3. Floating point binary data.
4. Binary Coded decimal (BCD) data.

Signed magnitude Addition and Subtraction.

→ We designate the magnitudes of the two numbers by A and B.

→ When the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed.

→ These conditions are listed in the first column of the table.

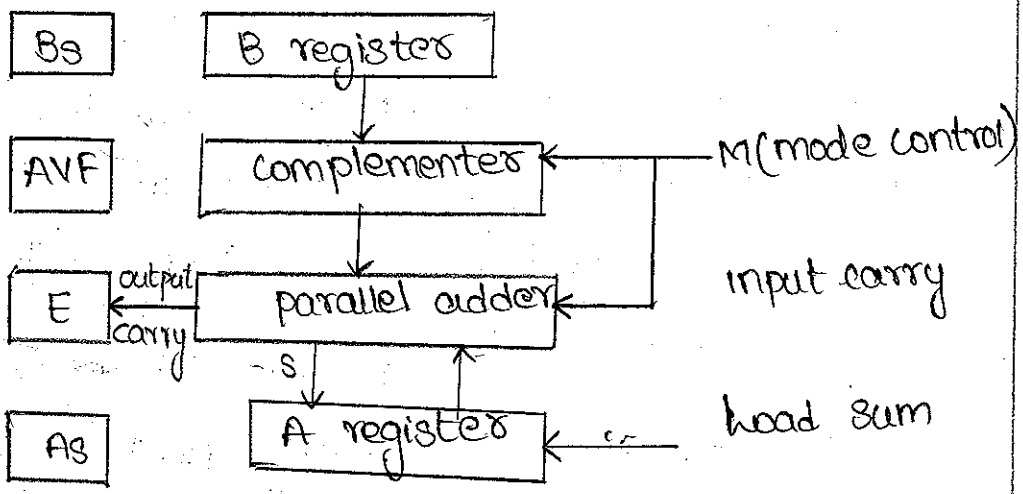
→ The other columns in the table shows the actual operation to be performed with the magnitude of the numbers.

337

Operation	Add Magnitudes	Subtract Magnitudes		
		When A>B	When A<B	When A=B
(+A)+(+B)	+(A+B)			
(+A)+(-B)		+(A-B)	-(B-A)	+(A-B)
(-A)+(+B)		-(A-B)	+(B-A)	+(A-B)
(-A)+(-B)	-(A+B)			
(+A)-(+B)		+(A-B)	-(B-A)	+(A-B)
(+A)-(-B)	+(A+B)			
(-A)-(+B)	-(A+B)			
(-A)-(-B)		-(A-B)	+(B-A)	+(A-B)

Hardware Implementation

→ The Block diagram of the hardware implementation for addition and subtraction operations is as shown below.



→ It consists of registers A and B and sign Flip-flops As and Bs.

→ Subtraction is done by adding A to the 2's complement of B.

→ The output carry is transferred to flip-flop E, where it can be checked to determine the relative magnitudes of the two numbers.

→ The add-overflow flipflop AVF holds the overflow bit when A and B are added.

→ The A Register provides other microoperations that may be needed when we specify the sequence of steps in the algorithm.

→ The addition of A plus B is done through parallel adder.

→ The S(sum) output of the adder is applied to the input of the A Register.

→ The Complementer provides an output of B or the complement of B depending on the state of the mode control 'M'.

→ The Complementer Consists of Exclusive OR gates and the parallel adder Consists of full adder circuits.

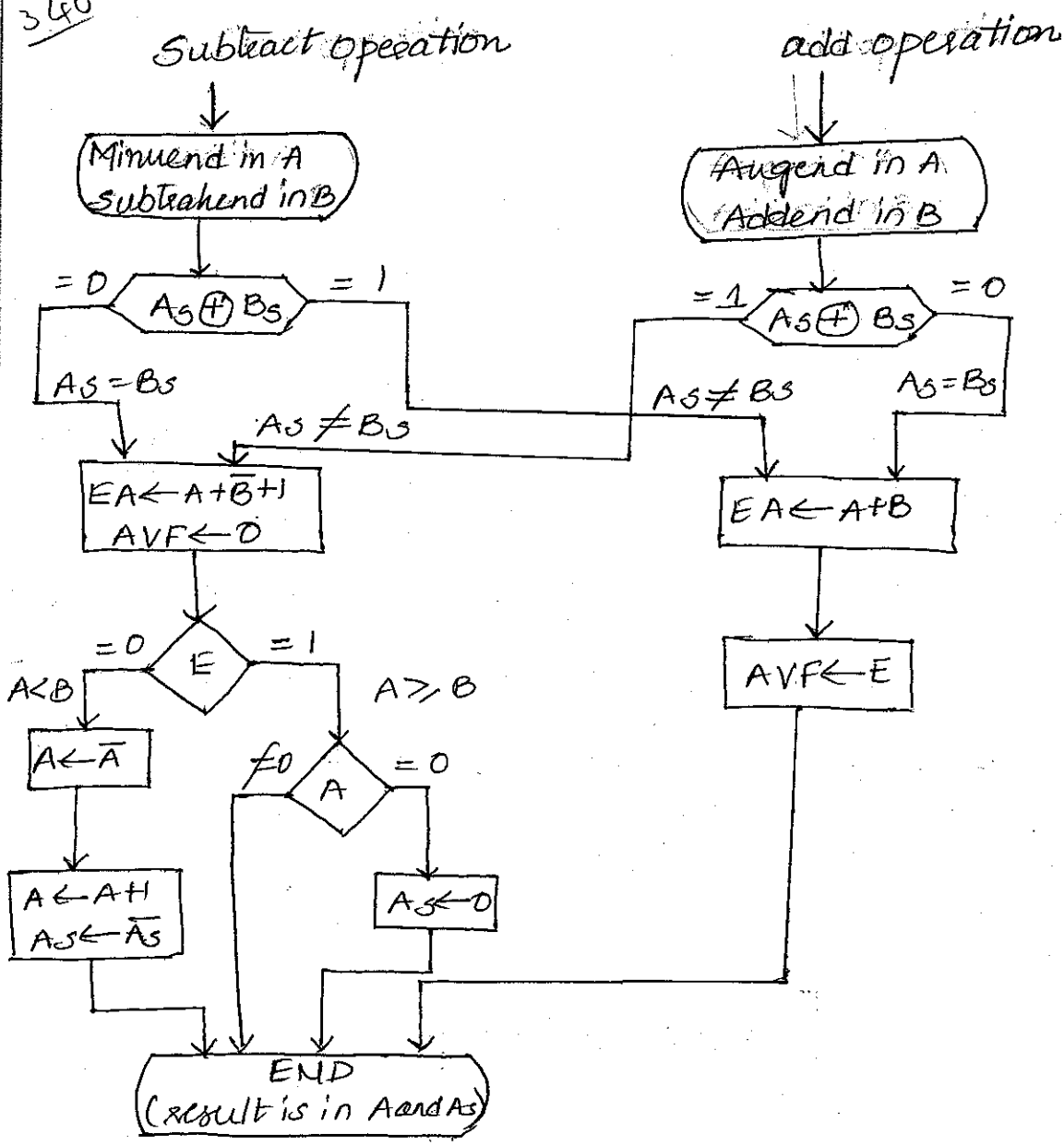
→ When M=0, the output of B is transferred to the adder, the input carry is '0'. and the output of the adder is equal to the sum A+B.

→ When M=1, the is Complement of 'B' is applied to the adder, the input carry is 1, and the output S= A+$\bar{B}$+1. This is equivalent to the Subtraction, A-B.

(P.T.O)

Hardware Algorithm for Addition and Subtraction for Signed-magnitude data.

340



Addition and Subtraction with Signed 2's Complement.

→ when two numbers of 'n' digits each are added and the sum occupies n+1 digits, we say that an overflow occured.

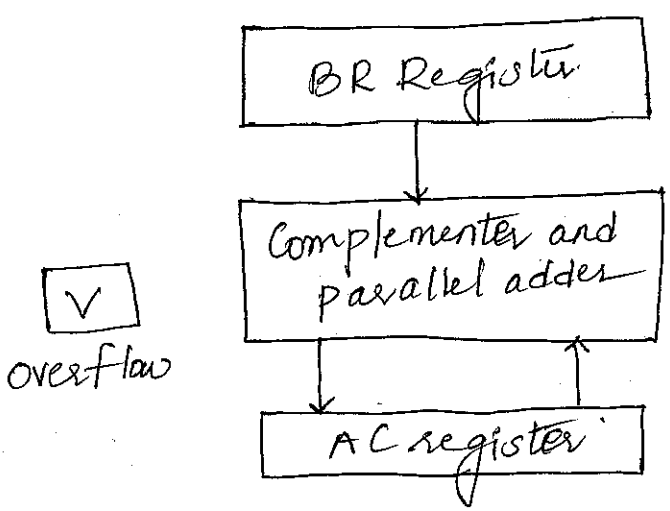
→ The effect of an overflow on the sum of two signed 2's complement numbers is to be considered.

→ An overflow can be detected by inspecting the last two carries out of the addition.

→ when the two carries are applied to an Exclusive 'OR' Gate, the overflow is detected when the output of the gate is equal to 1.

Hardware Implementation.

349



→ The Register configuration for the hardware implementation is as shown in the figure.

→ This is similar to the signed-magnitude, Except that the sign bits are not separated from the rest of the registers.

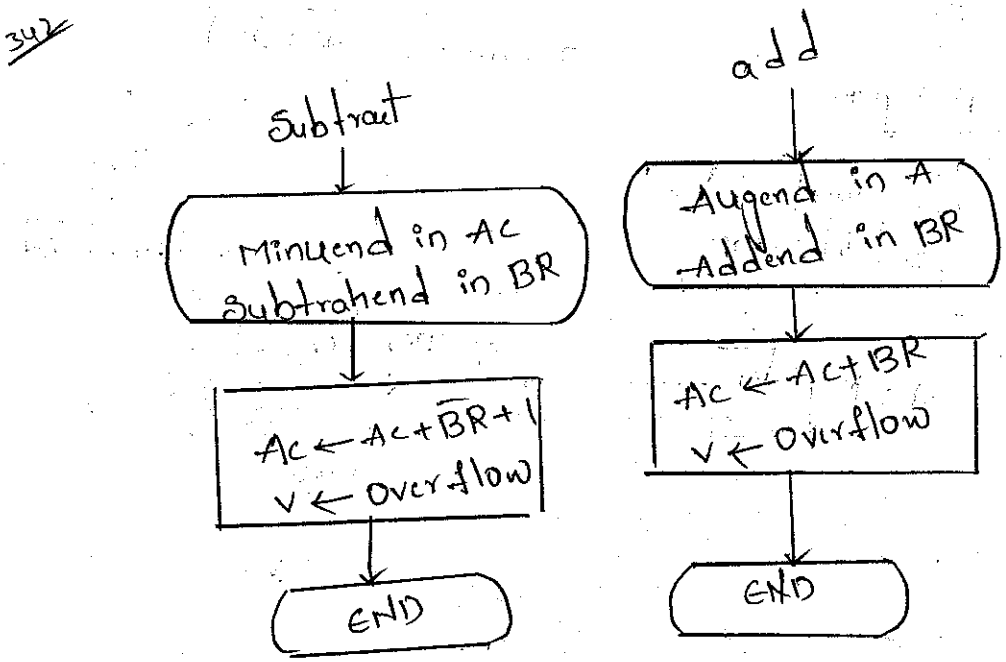
→ we name the A Register as AC (accumulator) and the B Register as BR.

→ The leftmost bit in AC and BR Represent the sign bits of the numbers.

→ The two sign bits are added or subtracted together with the other bits in complementer and parallel adder.

→ The overflow flip-flop 'V' is set to 1, if there is an overflow.

Hardware Algorithm:



Multiplication Algorithm.

Consider an example for general multiplication.

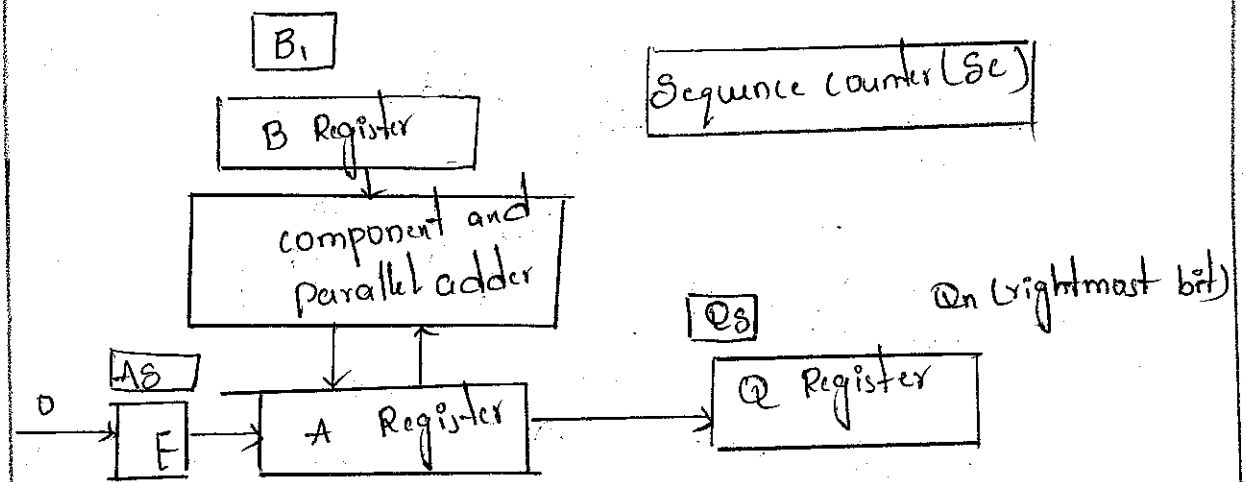
```
23      10111     multiplicand
19   X  10011     multiplier
-----------------

         10111
        10111
       00000
      00000
     10111
-----------------
437   111011010l
-----------------
```

Multiplication algorithm for Signed magnitude data.

Hardware implementation.

3u3



→ The Hardware for multiplication consists of the equipment as shown in figure.

→ The multiplier is stored in the Q Register and its sign in Q_s.

→ The Sequence counter SC is initially set to a number equal to the number of bits in the multiplier.

→ The counter is decremented by 1 after forming each partial product.

→ When the content of the counter reaches zero, the product is formed and the process stops.

→ Initially, the multiplicand is in Register B and the multiplier in Q.

→ The Sum of A and B forms a partial product which is transferred to EA Register.

→ Both partial product and multiplier are shifted to the right.

→ This 'shift is denoted as shr EAQ.

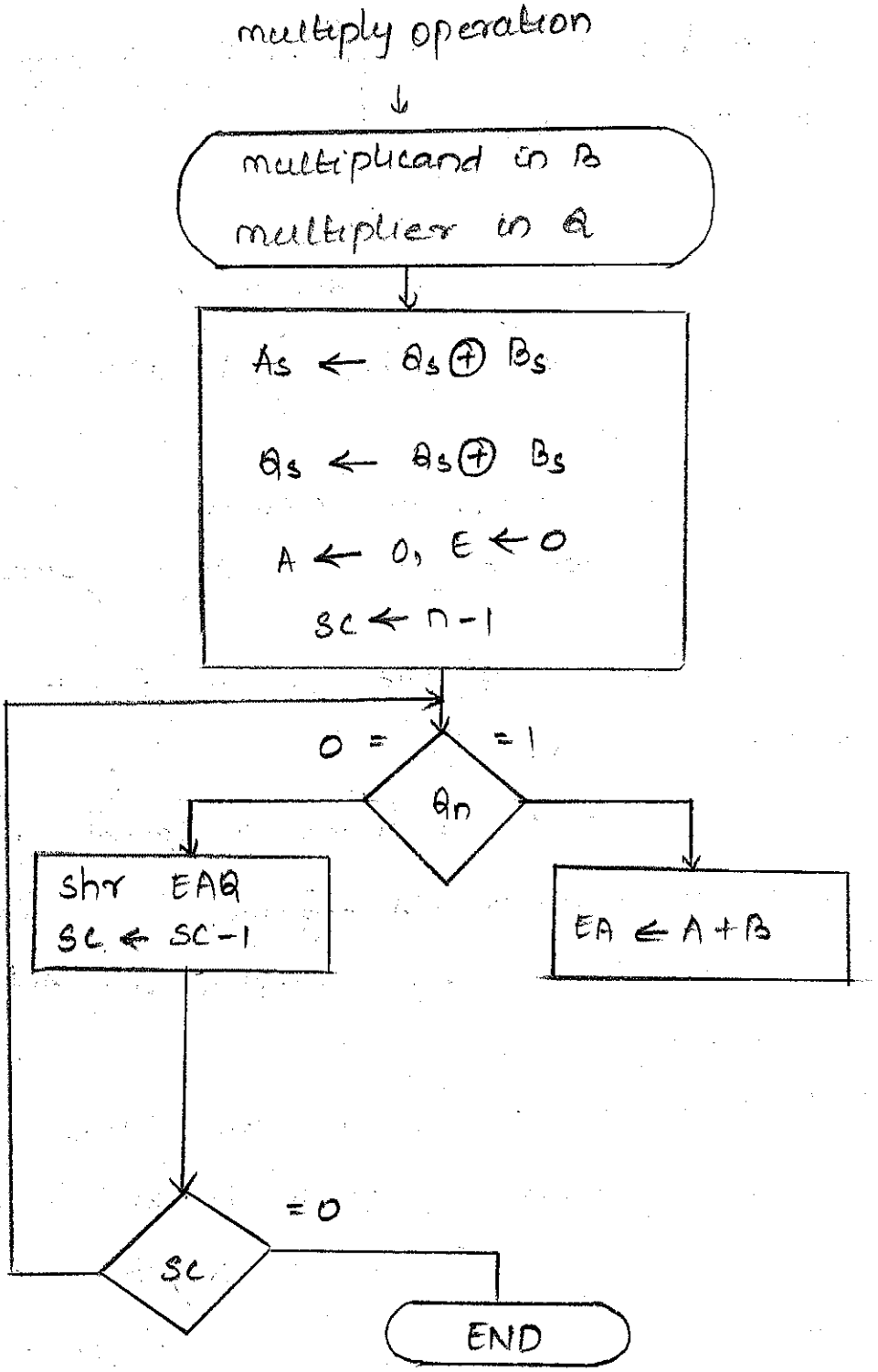
→ The least Significant bit of A is shifted in to the most significant position of Q., the bit from E is shifted in to the most Significant position of A, and '0' is shifted in to E.

→ After the shift, one bit of the partial product is shifted in to Q, pushing the multiplier bits one position to the right.

→ Finally, the right most bit of Q, i.e, Q_n must be inspected next.

Hardware Algorithm

magnitude → multiplication



Numerical Example:

305 Numerical example for Binary Multiplier

Multiplicand B = 10111	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		10111		
first partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
$Q_n = 0$; Shift right EAQ	0	01000	10110	010
$Q_n = 0$; Shift right EAQ	0	00100	01011	001
$Q_n = 1$; add B		10111		
fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000

final product in

AQ = 0110110101

Booth Multiplication Algorithm. (or)

Multiplication algorithm for signed 2's Complement data:-

→ Booth algorithm gives a procedure for multiplying binary integers in signed 2's complement representation.

Booth Algorithm:

Step 1: The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.

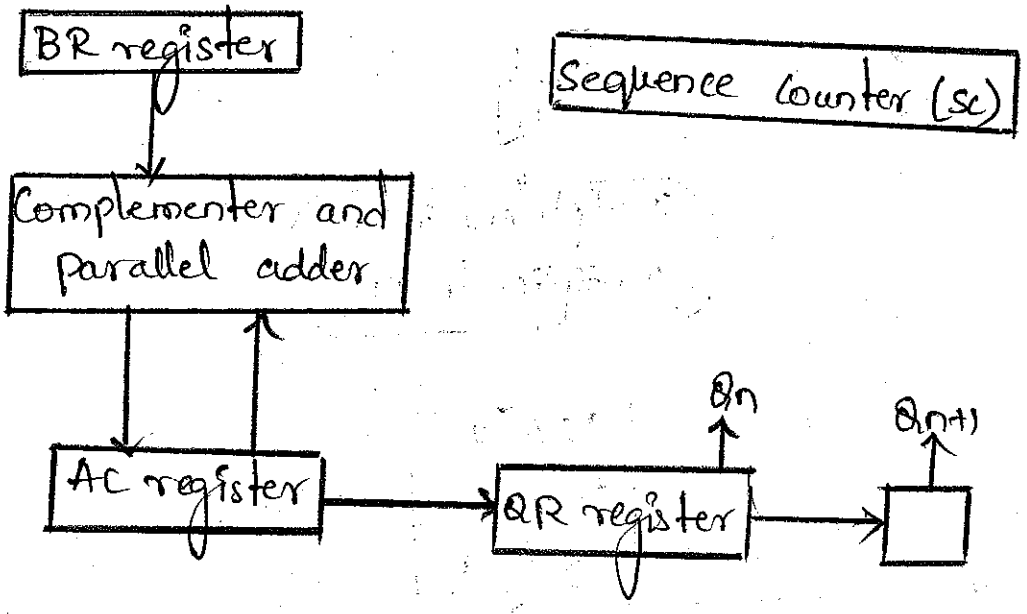
Step 2: The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1) in a string of 0's in the multiplier.

Step 3: The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

→ This algorithm works for positive or negative multipliers in 2's complement representation.

Hardware Implementation.

346



→ The Hardware implementation of Booth algorithm requires the register configuration as shown in figure..

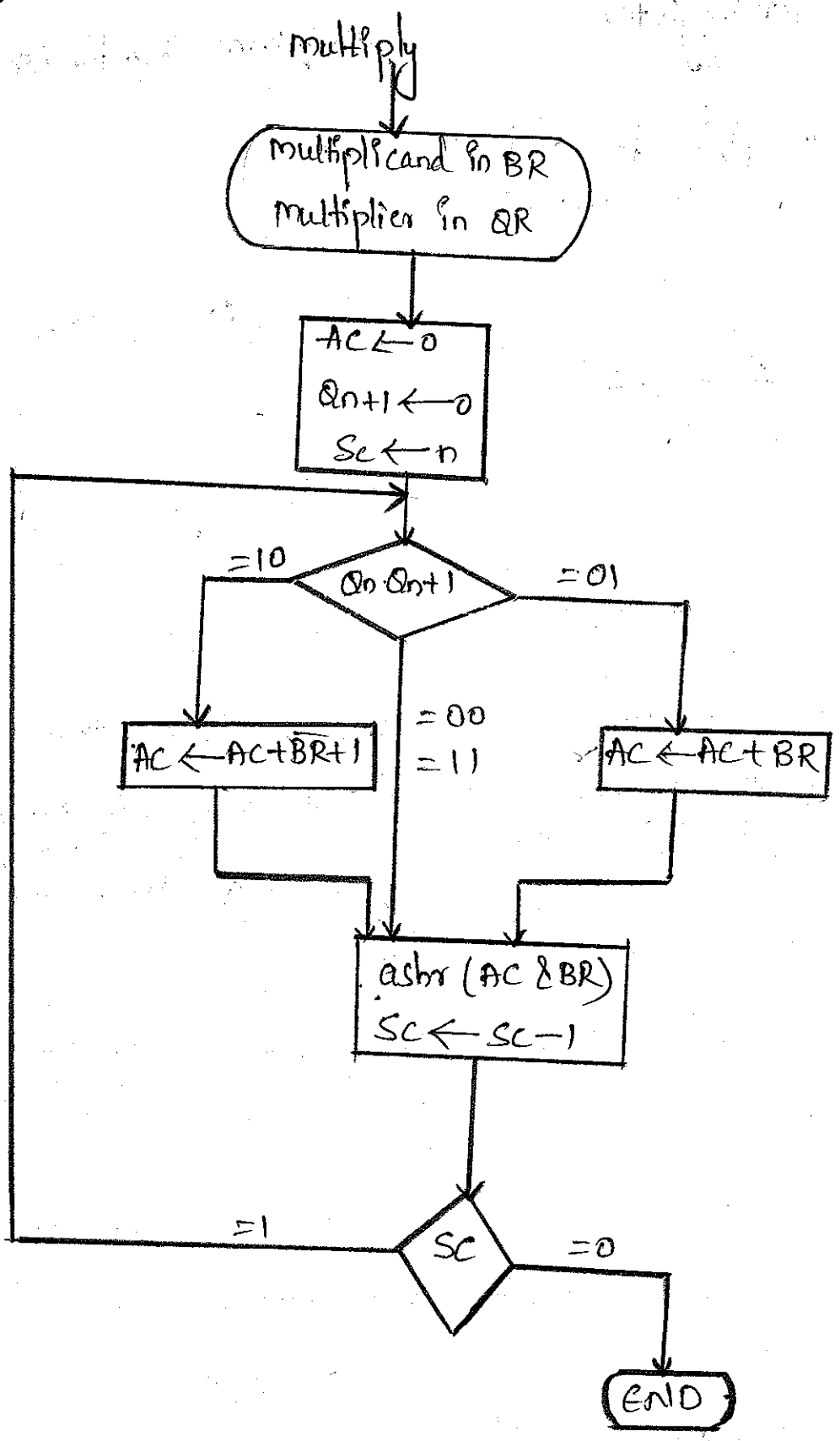
→ In this we Rename registers A, B and Q as, AC, BR and QR respectively.

→ Qn designates the least Significant bit of the multiplier in register QR.

→ An Extra flip-flop Q_{n+1} is appended to QR to facilitate a double bit inspection of the multiplier.

→ The flowchart of Booth algorithm is as shown below.

Flow chart – Booth algorithm for multiplication of Signed 2's complement numbers.



Numerical Example:

TABLE: EXAMPLE OF MULTIPLICATION WTH BOOTH ALGORITHM

Q_n	Q_{n+1}	BR = 10111 $\overline{BR}+1$ = 01001	AC	QR	Q_{n+1}	SC
		Initial	00000	10011	0	101
1	0	Substract BR	01001			
			01001			
		ashr	00100	11001	1	100
1	1	ashr	00010	01100	1	011
		Add BR	10111			
			11001			
		ashr	11100	10110	0	010
0	0	ashr	11110	01011	0	001
1	0	Substract BR	01001			
			00111			
		ashr	00011	10101	1	000

UNIT 2

Register Transfer language

* A Digital System is an interconnection of digital hardware modules that accomplish a specific task.
* Digital Systems vary in size and complexity from a few integrated circuits to a complex interacting digital computers.
* Digital System design Generally uses a modular approach.
* The modules are constructed from such digital Components as registers,
  Decoders,
  Arithmetic elements, and
  Control logic.
* The various modules are inter-connected with common data and control paths to form a digital computer System.
* Digital modules are best defined by the registers they Contain and the operations that are performed on the data Stored in them.
* The operations executed on data Stored in registers are called as microoperations.
* The microoperation is an elementary operation performed on the information Stored in one or more registers.
* The Result of the operation may replace the previous binary information of a register or may be transferred to another register.

Examples of microoperations: (a) Shift
(b) Count
(c) Clear and
(d) Load etc.

* The internal Hardware organization of a digital computer is best defined by specifying,
  i) The set of registers it contain and their functions
  ii) The sequence of microoperations performed on the binary information stored in the registers.
  iii) The control that initiates the sequence of microoperations.

* It is possible to specify the sequence of microoperations in a computer by explaining every operation in words, but this procedure involves a lengthy descriptive explanation.

* For example to add two numbers (or) operands the following sequence of microoperations has to be performed.
  i) Load first operand in register A
  ii) Load second operand in Register B
  iii) Perform addition microoperation.
  iv) Store the result in the destination register.

As described above, it is possible to specify the sequence of microoperations in words, but it involves a lengthy descriptive explanation.

* Therefore, It is more convenient to select a suitable symbology to describe the sequence of transfers between registers and the various arithmetic and logic microoperations associated with the transfers.

* The Symbolic notation used to describe the microoperation transfers among registers is called a "Register Transfer language".

* The term "Register transfer" implies the availability of hardware logic circuits that can perform a stated micro-operation and transfer the result of the operation to the same or another register.

* The word "language" is borrowed from programmers, who apply this term to programming languages.

* Similarly, A Register Transfer language is a system for expressing in symbolic form of the microoperation sequences among the registers of a digital module.

* The Register transfer language is a kind of intermediate representation that is very close to assembly language, which is used in a compiler.

Register transfer:

Registers: Computer registers are designated by capital letters and sometimes followed by numerals, to denote the function of the register.

* For example, the registers that holds an address for the memory unit is usually called a memory address register and is designated by the name MAR.

* Some other designations are,

'PC' for program counter

'IR' for Instruction Register

'R1' for processor register.

* The individual flip-flops in an n-bit register are numbered in sequence from 0 to n-1, starting from 0 in the rightmost position and increasing the numbers towards left.

Block diagram of Registers:

→ The most common way to represent a register is by a rectangular box with the name of the register inside it.

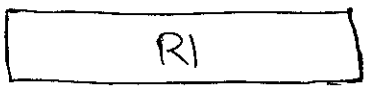


fig: Register R1

→ The Individual bits can be distinguished in an n-bit register are numbered in sequence from 0 to n-1, starting from 0 in the rightmost position and increasing the numbers towards left.

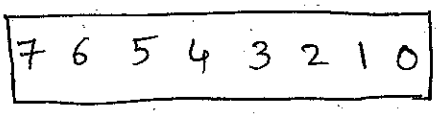


fig: Showing individual bits.

→ The numbering of bits in a 16-bit register can be marked on top of the box.

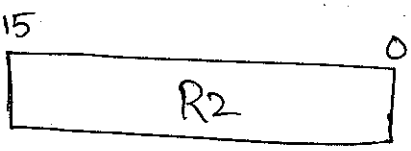


fig: Numbering of bits

→ A 16-bit register can be partitioned in to two parts. bits 0-7 are assigned the symbol 'L' (for low byte) and bits 8-15 are assigned the symbol 'H' (for high byte).

* If the name of the 16-bit register is 'PC', then the symbol PC(0-7) (or) PC(L) refers to low-order byte and PC(8-15) (or) PC(H) refers to High-order byte.

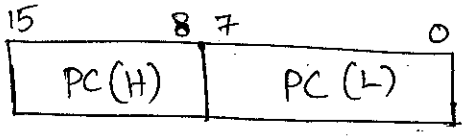
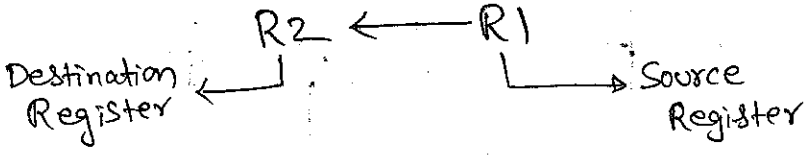


fig: Divided in to two parts.

Implementation of Register transfer:

* Information transfer from one register to another is represented in symbolic form by means of a replacement operator.



The above Statement denotes a transfer of the contents

of register R1 in to register R2. i.e, It replaces the contents of R2 by the contents of R1.

* The contents of the source register, R1 does not change after the transfer.

* Normally transfer is occured only under predetermined control condition.

* This can be shown by means of an if then statement.

$$\text{if}(P=1)\ \text{then}\ (R2 \leftarrow R1)$$

where 'p' is a control signal generated in the control section.

* It is convenient to separate the control variables from the register transfer operation by specifying a control function.

* A control variable is a Boolean variable that is equal to either '1' or '0'.

* The control function can be included in the statement as follows.

$$P: R2 \leftarrow R1$$

Here, 'P' is a control variable and is basically a boolean variable having a value 1 or 0.

* This statement indicates that, the operation is performed only when P=1. otherwise operation is not performed.

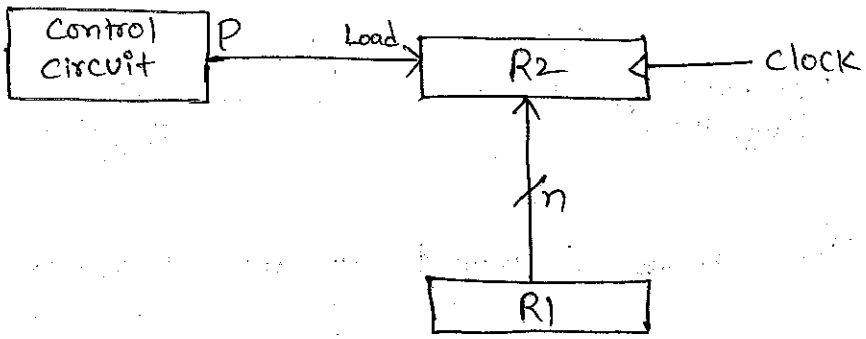


fig: Block diagram that represents the transfer from R1 to R2.

* The 'n' outputs of R1 are connected to the 'n' inputs of register R2.
* The letter 'n' will be used to indicate any number of bits for the register.
* Register R2 has a load input, that is activated by the control variable 'P'.
* The clock for the control variable must be synchronized with the same clock as applied to the register R2.

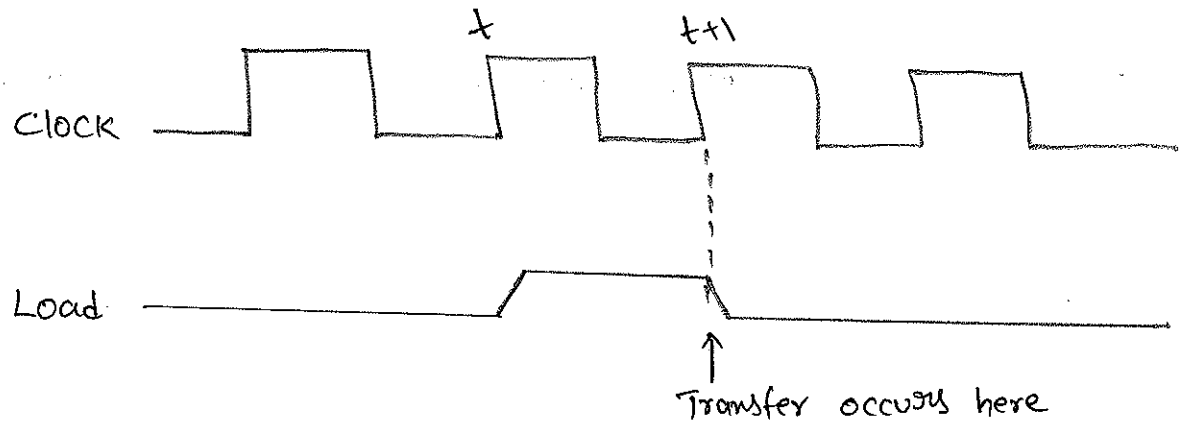


fig: Timing diagram.

* Both registers 'R2' and control variable 'P' are synchronized with the same clock pulse. Therefore, when the register transfer is implemented, at the same time, the control variable 'P' is also enabled.
* In the timing diagram, at time 't', 'P' is enabled which is represented by the rising edge of a clock pulse. (ie, 1). and at time t+1, the load input is activated and the inputs of R1 are transferred in to register R2 simultaneously.

* Now, at time t+1 the control variable 'p' may go back to its original state.

Basic Symbols for register transfers

Symbol	Description	Examples.
① Letters (and numerals)	Denote a register	MAR, R2
② Parenthesis ( )	Denotes a part of a register	R2(0-7), R2(L)
③ Arrow ←	Denotes transfer of Info	R2 ← R1
④ comma ,	Separates two microoperations	R2 ← R1, R_4 ← R_3

Bus and memory Transfers:

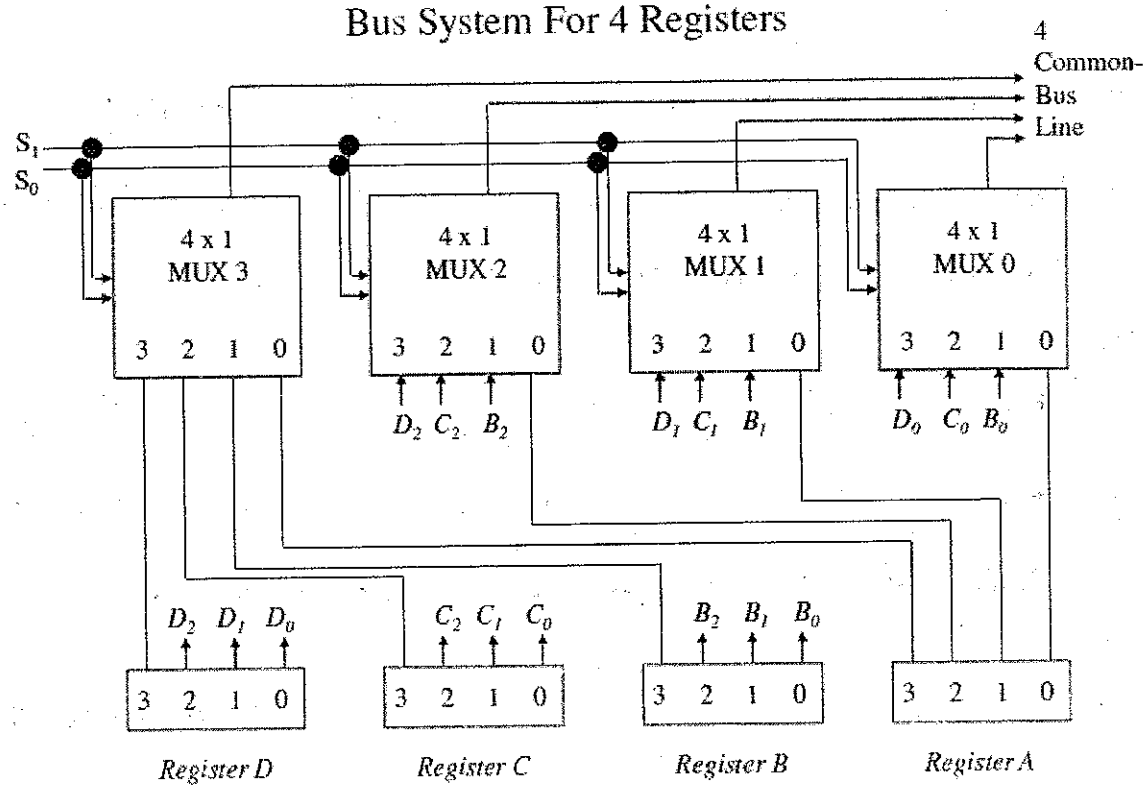
* A Digital computer has many registers, and paths must be provided to transfer information from one register to another.
* The number of wires will be excessive if separate lines are used between each register and all other registers in the system.
* The most efficient scheme for transferring information between registers in a multiple-register configuration is by using a common bus system.
* A Bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time.
* A common bus system can be constructed in two ways
  - i) With multiplexers and
  - ii) With Three-State Bus Buffers.

i) With multiplexers:

The multiplexers select the source register whose binary information is then placed on the bus.

→ The construction of a bus system for four registers is shown in the following figure.

Bus System For 4 Registers



* Each register has four bits, numbered 0 to 3.
* The Bus consists of four 4X1 multiplexers, each having four data inputs, 0 to 3.
* And Bus System has two selection inputs, S1 and S0.
* In order to, not to complicate the diagram with 16 lines crossing each other, we use labels to show the connections from the outputs of the registers to the inputs of the multiplexers.
* The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer, to form one line of the bus.
* Thus,

  MUX 0 multiplexes the four 0 bits of the registers.

  MUX 1 multiplexes the four 1 bits of the registers.

  MUX 2 multiplexes the four 2 bits of the registers

  MUX 3 multiplexes the four 3 bits of the registers
* The two selection lines S1 and S0 are connected to the selection inputs of all four multiplexers.
* The selection lines choose the four bits of one register and transfer them in to the four line common bus.
* When S1S0=00, then 0 data inputs of all multiplexers are selected and applied to the outputs that form the Bus.
* This causes the bus lines to receive the content of register 'A'.

* Similarly, Register B is selected if S1S0 = 01 and so on.

* The following table shows, the register that is selected by the bus for each of the four possible binary value of the selection lines.

S1	S0	Register selected.
0	0	A
0	1	B
1	0	C
1	1	D

* In general, a bus system will multiplex 'k' Registers of 'n' bits each to produce an 'n' line common bus.

* The no. of multiplexers needed to construct the bus is equal to n, the no. of bits in each register.

* The size of each multiplexer must be kx1, since it multiplexes 'k' data lines.

* for Example:

K ⇒ registers = 4
n → Bits = 4

Then, No. of multiplexers needed = n ⟹ 4.

and size of multiplexer = K x 1
⟹ 4 x 1

* The Symbolic statement for a bus transfer may mention the bus or its presence may be implied in the statement.

* When the bus is included in the statement, the register transfer is Symbolized as follows.

BUS ← C , R1 ← BUS

* for convenience the above transfer is generally written as follows, which shows the direct transfer.

$$R1 \leftarrow C$$

Three-state Bus Buffer:

* A Digital circuit that exhibits 3 states is known as three-state gate.
* Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a high-impedance state.
* The high-impedance state behaves like an open circuit, which means that, the o/p is disconnected and does not have a logic significance.
* The most commonly used conventional logic, in design of a bus system is the buffer gate.
* This buffer gate consists of Normal input and control input.
* The control input determines the output state.
* when control input=1, the o/p is enabled and is equal to the normal input.
* when the control input=0, the output is disabled and the gate goes to a high-impedance state.

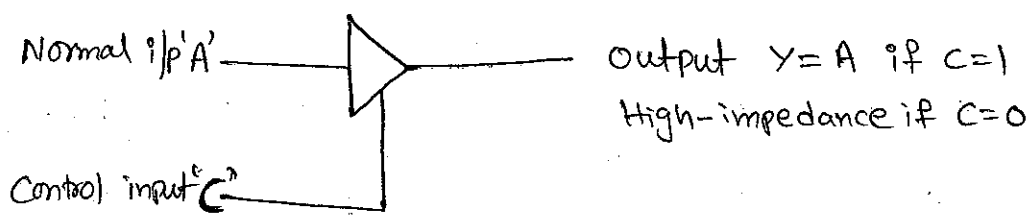
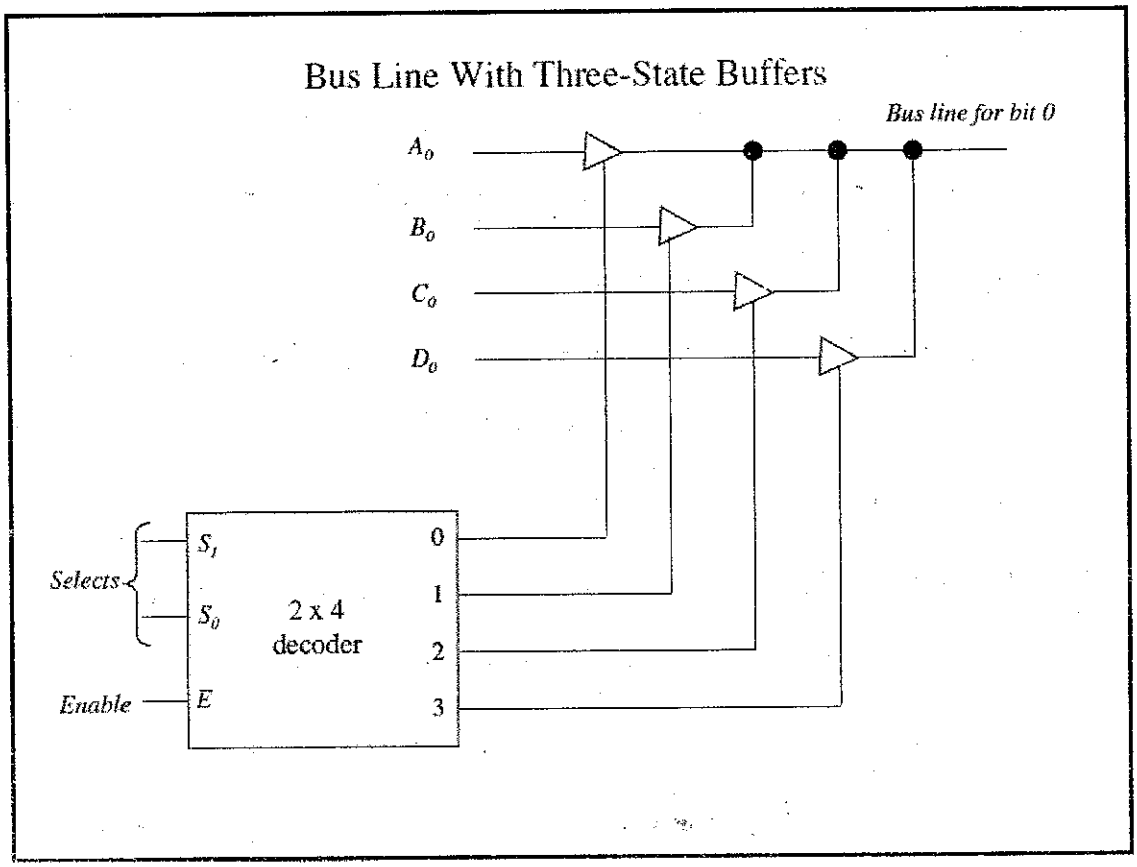


fig: Three state Buffer

→ The construction of a bus system with three-state buffers is as given below.



* The outputs of four buffers are connected together to form a single bus line.

* The control inputs to the buffers determines which of the four normal inputs will communicate with the bus line.

* only one three-state buffers has access to the bus line and all other buffers are maintained in a high-impedance state.
* To ensure that no more than one control input is active at any given time, a decoder is used.
* When the enable input of the decoder is 0, all of its outputs are 0.
* When the enable input is active, one of the three-state buffers is active, depending on the binary value in the selected inputs of the decoder.
* To construct a common bus for four registers of 'n' bits using three-state buffers, we need 'n' circuits with four buffers in each.
* if Registers = 4
  No. of Bits = 'n', then it requires n → circuits.
* only one decoder is necessary to select between the four registers.

Memory Transfer:

Memory Read: The transfer of information from a memory word to the outside environment is called a Read operation.

Read : DR ← M[AR]

where, DR → is a data register and AR → Address register.
M → memory word.

Memory Write:

Memory write:

The transfer of new information to be stored in the memory is called as write operation.

Write: $M[AR] \longleftarrow R1$

This causes a transfer of information from R1 into the memory word 'M' selected by the address in AR.

Microoperations:

* A microoperation is an elementary operation performed with the data stored in registers.

* Microoperations in digital computers are classified in to four categories.

i) Register transfer microoperations transfer binary information from one register to another.

ii) Arithmetic microoperations perform arithmetic operation on numeric data stored in registers.

iii) Logic microoperations perform bit manipulation operations on nonnumeric data stored in registers

iv) Shift microoperations perform shift operations on data stored in registers.

Arithmetic microoperations:

* The Basic arithmetic microoperations are, ~~addition~~
  - Addition
  - Subtraction
  - Increment
  - Decrement and
  - Shift

* The arithmetic microoperation defined by the statement,

$$R3 \leftarrow R1 + R2$$

specifies an add microoperation.

* It states that the contents of register R1 are added to the contents of register R2 and the sum transferred to register R3.

* To implement this statement with hardware, we need three registers and a digital component that performs the addition operation.

* Subtraction is implemented through complementation and addition.

* Instead of using the minus operator, we can specify the subtraction by the following statement:

$$R3 \leftarrow R1 + \overline{R2} + 1$$

* Adding '1' to the 1's complement, $\overline{R2}$ produces the 2's complement.

* Adding the contents of R1 to the 2's complement of R2 is equivalent to R1 − R2.

* The other basic arithmetic operations are listed below.

Ex:		Description
$R3 \leftarrow R1 + R2$		contents of R1 plus R2 transferred to R3
$R3 \leftarrow R1 - R2$		Contents of R1 minus R2 transferred to R3
$R2 \leftarrow \overline{R2}$		Complement the contents of R2 (1's complement)

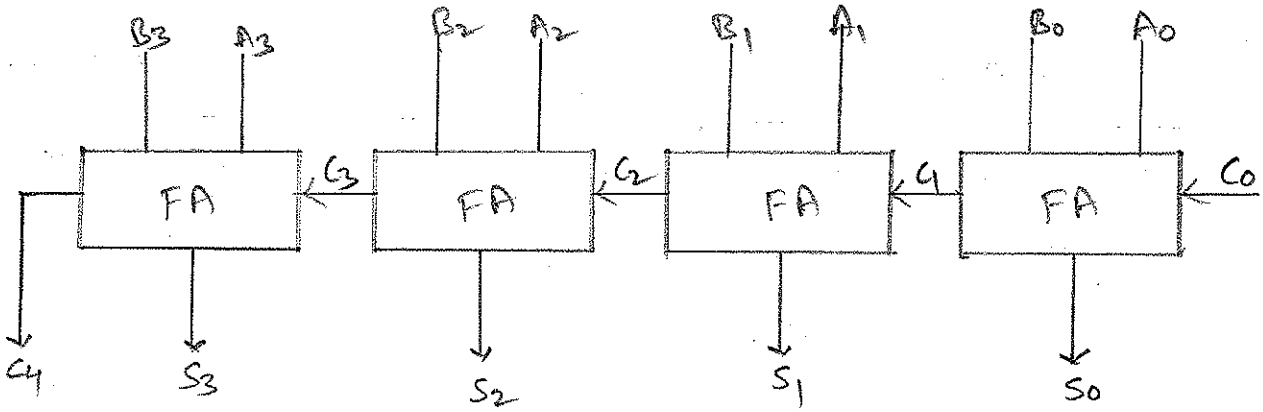
Binary Adder:

* To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the arithmetic addition.

* The digital circuit that forms the arithmetic sum of two bits and a previous carry is called a full adder.

* The digital circuit that generates the arithmetic sum of two binary numbers of any length is called a binary adder.

* The n-bit binary adder requires 'n' full-adders.
* The binary adder is constructed with full-adder circuits connected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder.
* The following figure shows a 4-bit binary adder with four Full-adders.



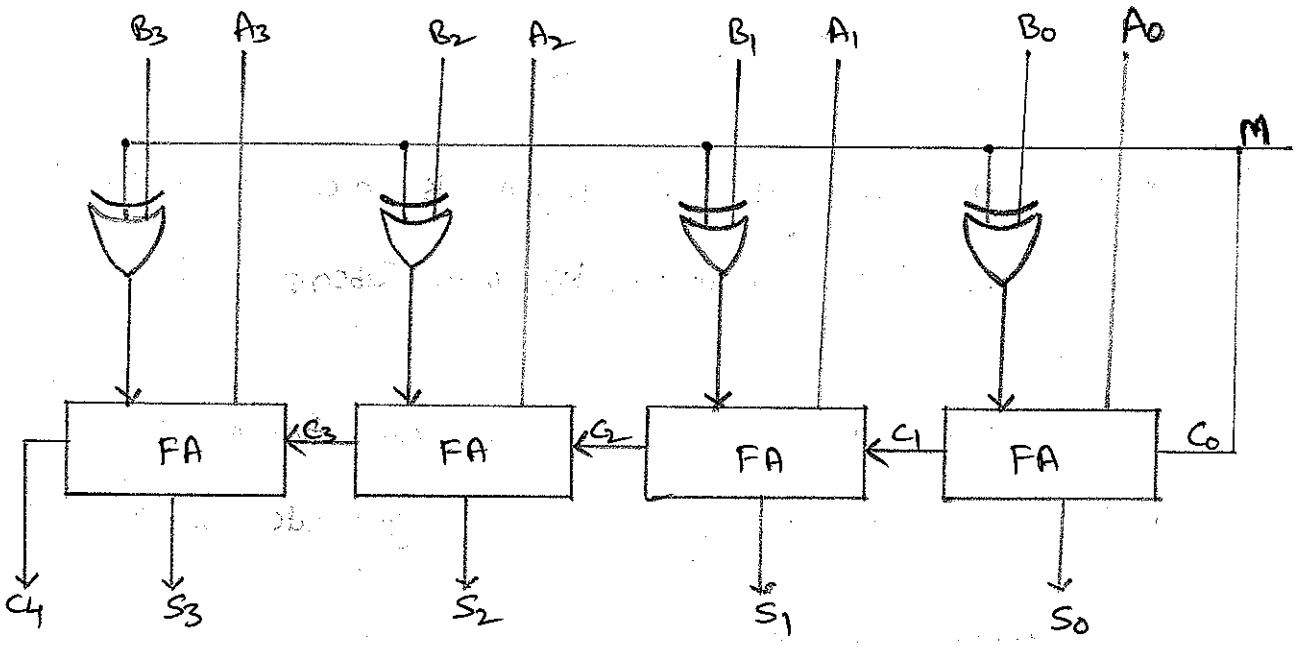
* The n data bits of A and B are designated by subscript numbers from Right to left, with subscript '0' denoting the low-order bit.
* The carries are connected in a chain through the full-adders.
* The S outputs of the full-adders generate the required sum bits.

Binary adder - Subtractor

* The Subtraction of binary numbers can be done most easily by means of complements.
* i.e, the subtraction A-B can be done by taking 2's complement of B and adding it to A.
* The addition and subtraction operations can be combined in to one common circuit by including an exclusive-OR gate

With each full-adder.

* A 4-bit adder-subtractor circuit is shown below.
* The mode input 'M' controls the operation.
* when M=0, the circuit is an adder and when M=1, the circuit becomes a subtractor.
* Each Exclusive-OR gate receives input M and one of the inputs of B.
* when M=0, we have $B \oplus 0 = B$. So the full-adder receives the value B, the input carry is 0, the circuit performs A plus B.
* when M=1, we have $B \oplus 1 = B'$ and $C_0 = 1$, In this case the circuit performs the operation A plus the 2's complement of B. which is nothing but A-B.



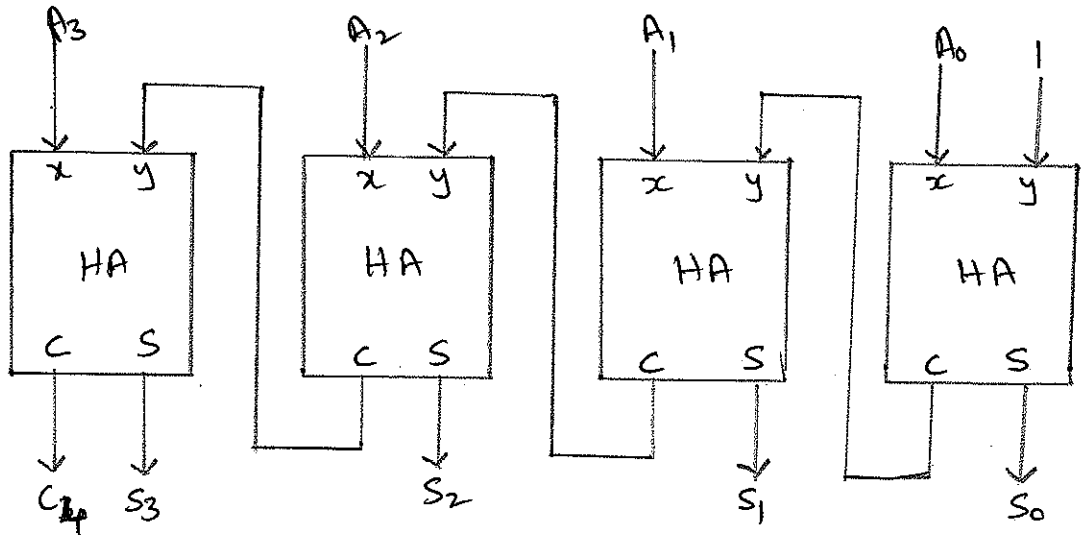
Binary Incrementer

* The Increment microoperation adds one to a number in a register.
* for Example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented.

i.e, $R1 \leftarrow R1+1$,

$$\begin{array}{r} 0110 \\ 1 \\ \hline 0111 \end{array}$$

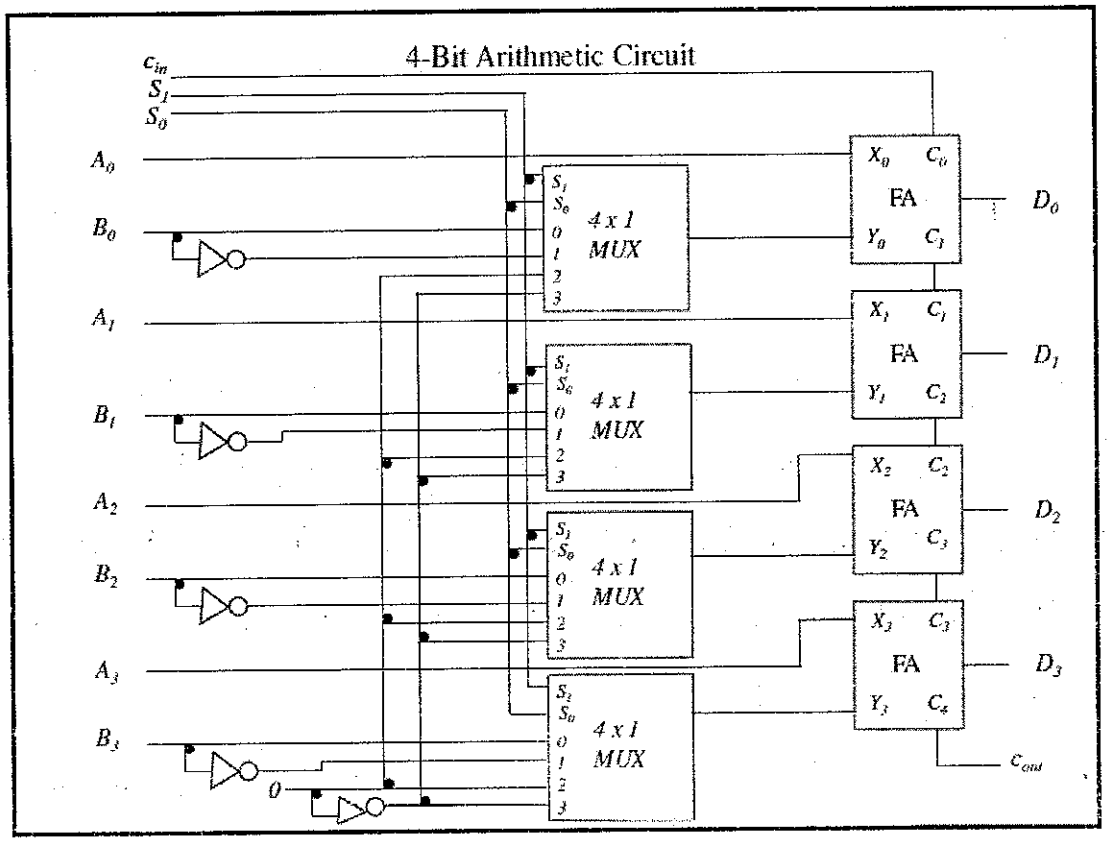
* This can be accomplished by using half-adders connected in cascade.
* A 4-bit combinational circuit incrementer is as shown below.



* one of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented.
* The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder.
* The circuit receives the four bits from A_0 to A_3, adds one to it, and generates the incremented output in S_0 to S_3.
* This circuit can be extended to an n-bit binary incrementer by Extending the diagram to 'n' half-adders.

Arithmetic Circuit:

* Multiple number of arithmetic microoperations can be implemented in one composite arithmetic circuit.
* The basic component of an arithmetic circuit is the parallel adder.
* By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.

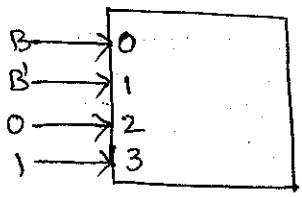

4-Bit Arithmetic Circuit
c_{in}
S_1
S_0
A_0
B_0
A_1
B_1
A_2
B_2
A_3
B_3
0
4 x 1 MUX
4 x 1 MUX
4 x 1 MUX
4 x 1 MUX
FA
FA
FA
FA
D_0
D_1
D_2
D_3
c_{out}

* The circuit consists of 4-Full adders, which forms 4-bit adder and 4-multiplexers for choosing different operations.

* There are two 4-bit inputs A and B and a 4-bit output D.

* The four inputs from A go directly to the X-inputs of the binary adder.

* Each of the four inputs from B are connected to the data inputs of the multiplexers.



* The four multiplexers are controlled by two selection inputs S_1 & S_0.

* The input carry C_{in} goes to the carry input of the Full Adder, in the least significant position. the other carries are connected from one stage to the next.

* The output of the binary adder is calculated from the following arithmetic sum.

$$D = A + Y + C_{in}$$

* So, by controlling the value of Y, with two selection inputs S_1, S_0 and making C_{in} equal to 0 (or) 1, it is possible to generate the eight arithmetic operations, given in the following table.

Select			Input Y	output $D = A+Y+C_{in}$	microoperation.
S_1	S_0	C_{in}			
0	0	0	B	$D = A+B$	ADD
0	0	1	B	$D = A+B+1$	Add with carry
0	1	0	$\overline{B}$	$D = A+\overline{B}$	Subtract with borrow
0	1	1	$\overline{B}$	$D = A+\overline{B}+1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A+1$	Increment A
1	1	0	1	$D = A-1$	Decrement A
1	1	1	1	$D = A$	Transfer A

Logic microoperations:

* Logic microoperations are binary operations performed on corresponding bits of two bit strings.
* These operations consider each bit of the register separately and treat them as binary variables.
* For Example, the exclusive-OR microoperation with the contents of two registers R1 and R2 is symbolized by the statement

$$P: R1 \leftarrow R1 \oplus R2$$

* It specifies a logic microoperation to be executed on the individual bits of the registers provided that the control variable $P=1$
* Let the content of R1 be 1010 and the content of R2 be 1100. Then logic computation of above statement is as follows.

```
1010  content of R1
1100  content of R2
0110  content of R1 after P=1.
```

* After the execution of the microoperation, the contents of R1 is equal to the bit-by-bit Exclusive-OR operation on pairs of bits in R2 and previous values of R1.
* The logic microoperations are used in scientific computations, but they are very useful for bit manipulation of binary data and for making logical decisions.

* Special Symbols

Special symbols will be used for the logic microoperations,

OR, AND and Complement, to distinguish them from the corresponding Symbols used to Express boolean functions.

* Special symbols used for logic microoperations are:

  Symbol '$\vee$' for 'OR' operation.

  Symbol '$\wedge$' for 'AND' operation

  Symbol '$\oplus$' for XOR operation.

List of logic microoperations:

* There are 16 different logic microoperations that can be performed with two binary variables.

* They can be determined from all possible truth tables obtained with two binary variables as shown below.

Table: 16 - logic microoperations.

Boolean function	Microoperation	Name.
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \bar{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \bar{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow (A \oplus B)$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x+y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \bar{B}$	complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \bar{B}$	
$F_{12} = x'$	$F \leftarrow \bar{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \bar{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow$ all 1's	set to all 1's.

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

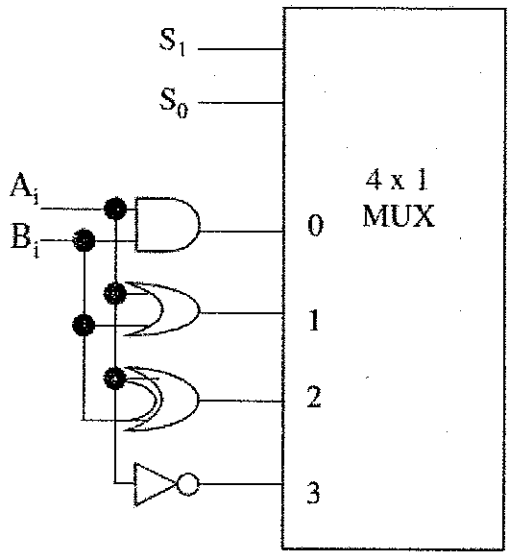
Table: Truth tables for 16 functions.

* The 16 boolean functions of two variables x and y are Expressed in algebraic form in the first column.
* The 16- logic microoperations are derived from these functions by replacing variable x by the binary content of register A and variable 'y' by the binary content of register B.

Hardware implementation

* The Hardware implementation of logic microoperations requires the logic gates to be inserted for each bit or pair of bits in the registers to perform the required logic function.
* Although there are 16 logic microoperations, most computers use only four, given by, AND, OR, XOR and Complement, from which all others can be derived.
* The below diagram shows one stage of a circuit that generates the four basic logic microoperations.

One Stage of Logic Circuit



S_1	S_0	Output	Operation
0	0	E = A ^ B	AND
0	1	E = A ∨ B	OR
1	0	E = A ⊕ B	XOR
1	1	E = A'	Complement

* It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic.
* The outputs of the gates are applied to the data inputs of the multiplexer.
* The two selection inputs S_1 and S_0 choose one of the data inputs of the multiplexer and direct its value to the output.
* The diagram shows one typical stage. For a logic circuit with n bits, the diagram must be repeated n times.

Applications of logic microoperations:

* Logic microoperations are very useful for manipulating individual bits of a register.
* They can be used to, i) change bit values.
  ii) Delete a group of bits.
  iii) Insert new bit values.
* Various Applications are:
  i) selective set
  ii) Selective Complement
  iii) select clear
  iv) mask
  v) insert
  vi) clear.

Numerical Examples:

i) Selective Set: Selective-set sets to 1 the bits in register A where there is corresponding 1 in register B.

Ex:

1010	Contents of A before
1100	Contents of B (logic operand)
1110	Content of A After

→ This is done using the logical-OR operation.
→ Therefore, the OR microoperation can be used to selectively set bits of a register.

ii) Selective complement:

Selective-complement complements the bits in register A where there is a corresponding 1 in register B.

Ex:
```
1010    Content of A Before
1100    Content of B (logical operand)
----
0110    Content of A After
----
```

→ This is done using the Exclusive-OR operation.

→ Therefore, the Exclusive-OR microoperation can be used to selectively complement bits of a register.

iii) Selective-clear:

Selective-clear clears to 0 the bits in register A, where there is a corresponding 1 in register B.

Ex:
```
1010    Contents of A Before
1100    Contents of B (logical operand)
----
0010    Contents of A After.
----
```

→ This is done using the logical-AND operation and $\overline{B}$.

→ If, the logic micro operation is,

$$A \leftarrow A \wedge \overline{B}$$

iv) Mask:

Mask clears to '0' the bits in register A where there is a corresponding '0' in register B.

Ex:
```
1010    Content of A Before
1100    content of B (logical operand)
----
1000    Content of A After.
----
```

→ This is done using the logical-AND operation and B.

v) Insert:

* The insert operation inserts a new value into a group of bits.

* This is done by first masking the bits and ORing them with the required value.

Ex: Suppose A register contains 8 bits 0110 1010, to replace the four leftmost bits by the value 1010,

→ First, we mask out the upper four bits.

```
0110 1010   Content of A Before
0000 1111   content of B (logic operand)
-----------
0000 1010   Content of A after.
-----------
```

→ Second, we insert the new values.

```
0000 1010   Contents of A Before
1001 0000   Contents of B(logic operand)
-----------
1001 1010   Contents of A After.
-----------
```

→ The masking is done using an AND and the insertion is done with an OR.

vi) Clear:

Clear compares A and B and produces all 0's, if the numbers are equal.

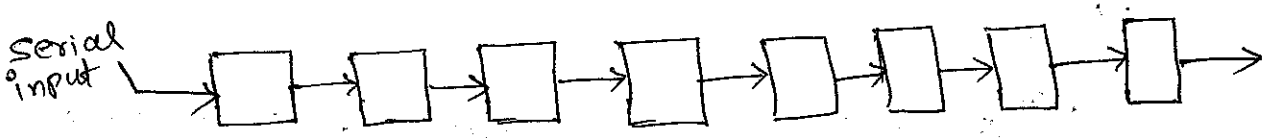
```
Ex:  1010   Content of A Before
     1010   Content of B (logic operand)
     ----
     0000   A ← A ⊕ B
     ----
```

i.e., if A & B are both '1' or both '0', this produces 0. This is done using the logical XOR operation and B.

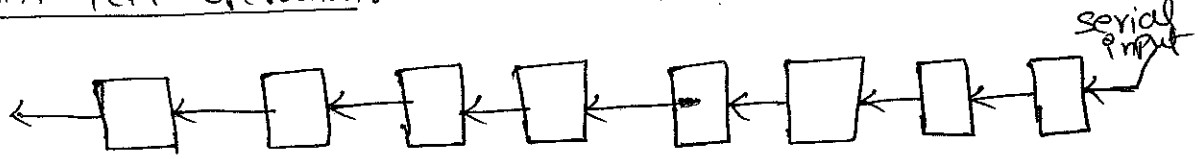
Shift Microoperations:

* Shift microoperations are used for serial transfer of data.
* They are also used in conjunction with arithmetic, logic and other data processing operations.
* The contents of a register can be shifted to the left or right.
* At the same time that the bits are shifted, the first flip-flop receives its binary information from the serial input.
* During a shift-right operation the serial input transfers a bit in to its left most position and during shift-left operation, the serial input transfers a bit in to the right-most position.
* The information transferred through the serial i/p, determines the type of shift.
* There are three types of shifts.
    i) Logical shift
    ii) Arithmetic shift and.
    iii) Circular shift.
* Generally shift right and shift left operations are performed as follows.

A Shift-right operation:



A Shift-left operation:



i) Logical shift:

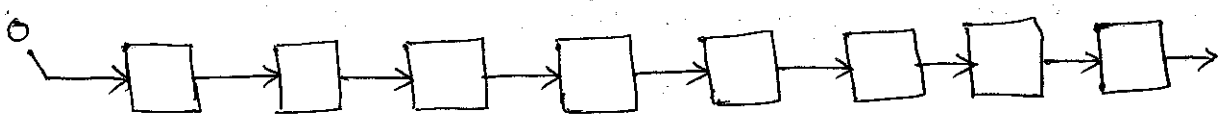
* A Logical shift transfers '0' through the serial input.
* The symbols 'shl' and 'shr' are used for logical shift-left and shift right microoperations respectively.

$$R1 \leftarrow shl\ R1$$
$$R2 \leftarrow shr\ R2$$

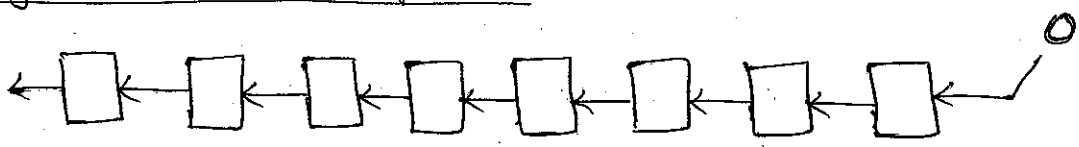
The above two microoperations specify that, 1-bit shift to the left of the content of register R1. and a 1-bit shift to right of the content of register R2.

Representation:

A logical shift right operation:

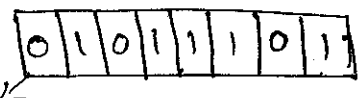


A logical shift left operation:



Ex:

Before logical shift right



After logical shift right.

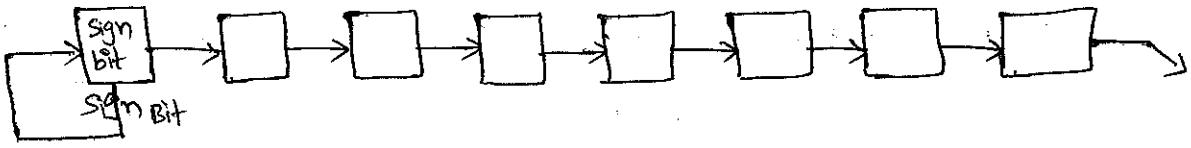
Why A bit '0' is inserted at right most position, when logical shift left operation is performed.

ii) Arithmetic shift:

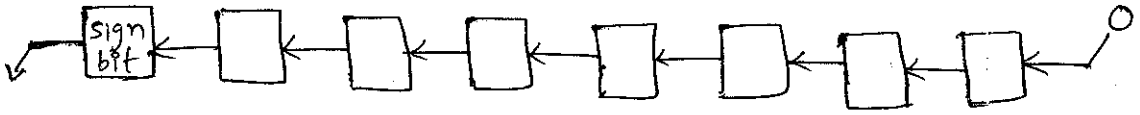
* An Arithmetic shift is a microoperation that shifts a signed binary number to the left or right.
* An Arithmetic shift left multiplies a signed number by two.
* An Arithmetic shift right divides a signed number by two.
* Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains same, when it is multiplied or divided by 2.

Representation:

* An Arithmetic shift right operation.



* An Arithmetic shift left operation.



* Let us assume 'R' is a register with n-bits, from $R_0, R_1, \ldots R_{n-1}$.

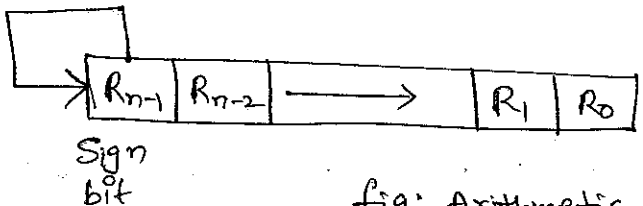


fig: Arithmetic shift right.

* The left most bit in a register holds the sign bit, and the remaining bits hold the number.
* The sign bit is 0 for positive and 1 for Negative.

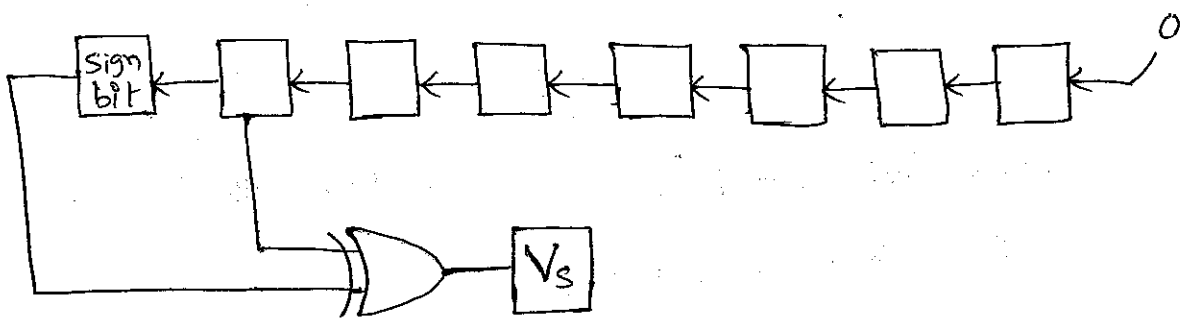
* The Arithmetic shift-right leaves the sign bit unchanged and shifts the number including the sign bit to the right.
* Thus R_{n-1} remains the same, R_{n-2} receives the bit from R_{n-1} and so on, for the other bits in the register.
* The bit in R_0 is lost.
* The arithmetic shift left inserts a '0' in to R_0 and shifts all other bits to the left.
* The initial bit of R_{n-1} is lost and replaced by the bit R_{n-2}.
* A sign reversal occurs if the bit in R_{n-1} changes in value after the shift.
* An overflow occurs after an arithmetic shift left, if initially, before the shift, R_{n-1} is not equal to R_{n-2}.
* An overflow flip-flop V_s can be used to detect an arithmetic shift-left overflow.

$$V_s = R_{n-1} \oplus R_{n-2}$$

if $V_s = 0$, there is no overflow.

if $V_s = 1$, there is an overflow and a sign reversal after the shift.

overflow detection:



Notation's used:

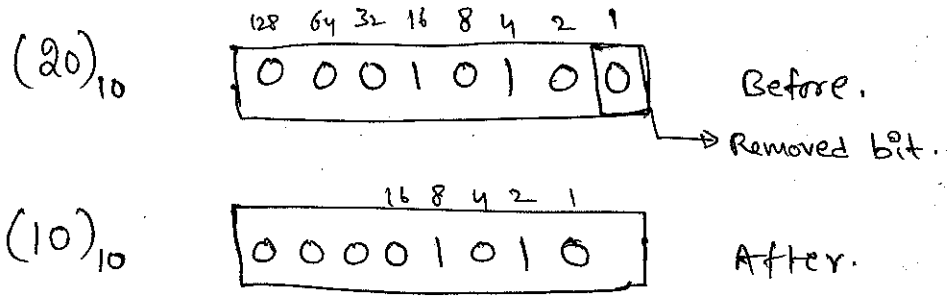
→ ashl for an arithmetic shift left.

→ ashr for an arithmetic shift right

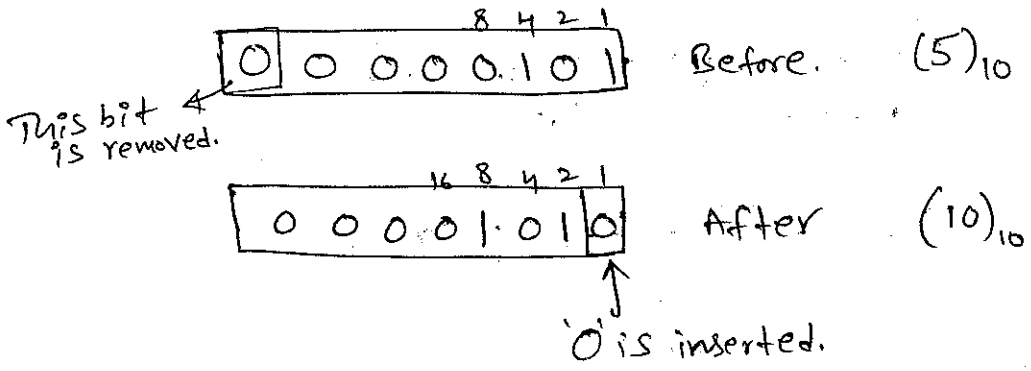
Ex: $R1 \leftarrow$ ashl R1

$R2 \leftarrow$ ashr R2

Arithmetic Shift-right Example:



Arithmetic Shift-left Example:

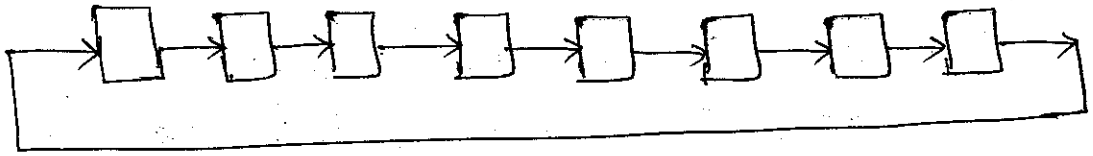


iii) Circular Shift:

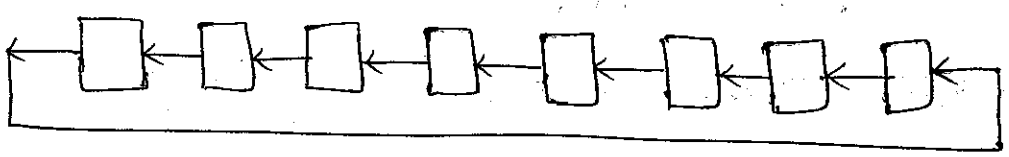
* The circular shift (Rotate operation) circulates the bits of the register around the two ends without loss of information.

* This is accomplished by connecting the serial output of the shift register to its serial input.

* A circular shift right operation.



* A circular shift - left operation.



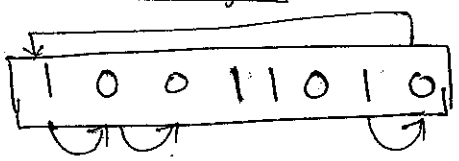
Notation's used

→ Cil for circular shift left
→ Cir for circular shift right

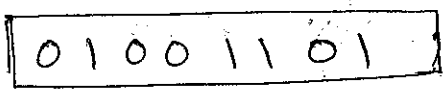
Examples:

R2 ← Cir R2
R3 ← Cil R3

Example for shift right



Before.

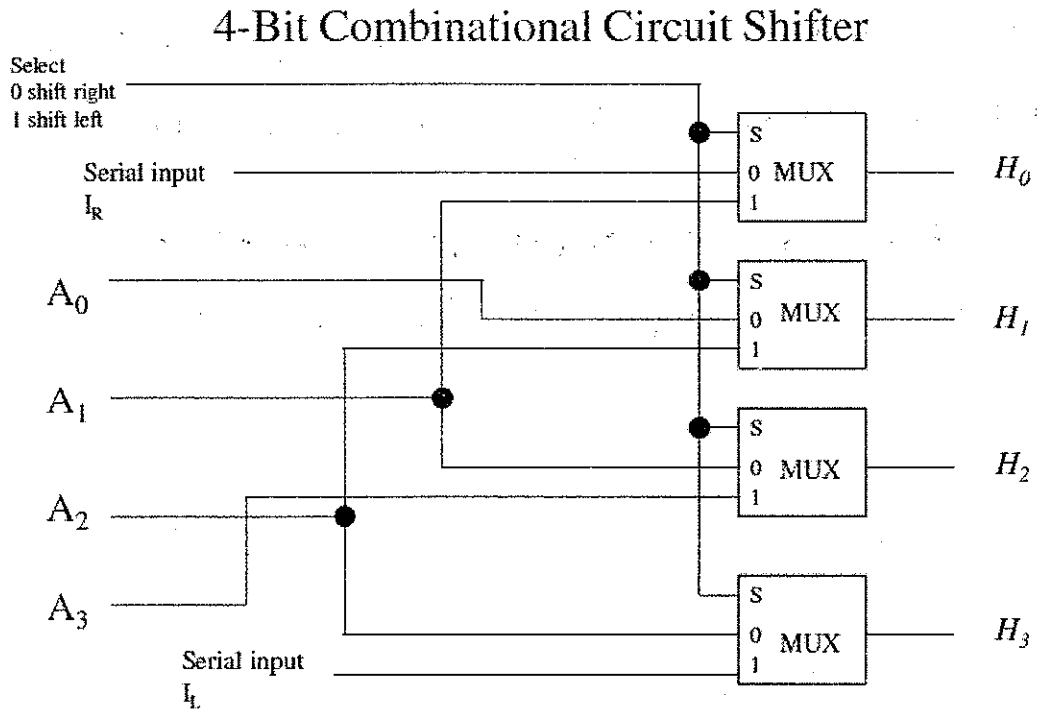


After

Hardware implementation:

* The Best choice for a shift unit would be a bidirectional shift register with parallel load.
* In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit.

* A combinational circuit shifter can be constructed with multiplexers as shown below.



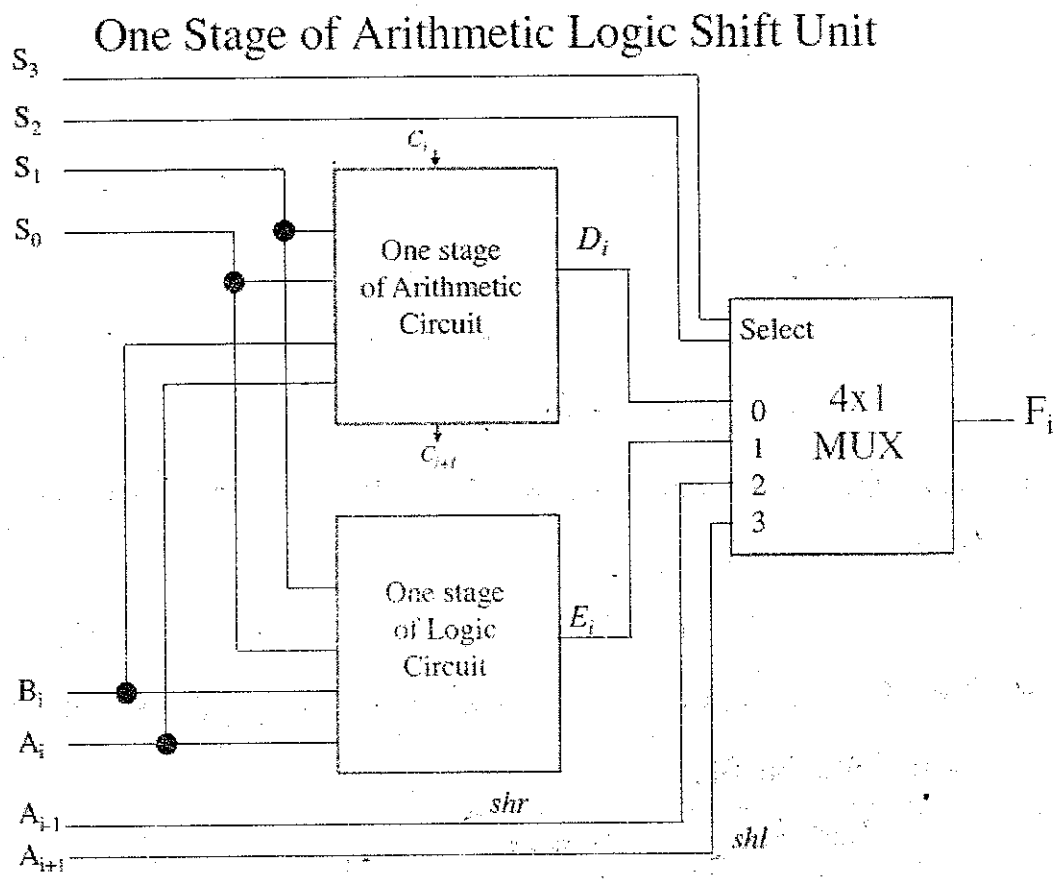
* It has four data inputs, A_0 to A_3, and four data outputs, H_0 to H_3.
* There are two serial inputs, one for shift left (I_L) and the other for shift right (I_R).
* when the selection input S=0, the input data are shifted right. (Down in the diagram).
* when S=1, the input data are shifted left (up in the diagram).
* The function table below shows which input goes to each output after the shift.

Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

Arithmetic logic shift unit:

* Instead of having individual registers performing the microoperations directly, computer systems employ a number of storage registers connected to a common operational unit called an arithmetic logic unit, abbreviated as ALU.

* To perform a microoperation, the contents of specified registers are placed in the inputs of the common ALU.



* The ALU performs an operation and the result of the operation is transferred to a destination register.
* The ALU is a combinational circuit, so that the entire register transfer operation from the source registers in to the destination register through ALU, can be performed during one clock pulse period.
* As shown, the Arithmetic, logic and Shift circuits can be Combined in to one ALU with common selection variables.
* Inputs A_i and B_i are applied to both the arithmetic and logic units.
* The data in the multiplexer are selected with inputs S_3 and S_2.
* The other two data inputs to the multiplexer receive inputs A_{i-1} for shift-right operation and A_{i+1} for the shift-left operation.
* The circuit must be repeated 'n' times for an 'n' bit ALU.
* The output carry C_{i+1} of a given arithmetic stage must be connected to the input carry C_i of the next stage in sequence.
* This circuit provides eight arithmetic operations, four logic operations and two shift operations.
* Each operation is selected with the five variables S_3, S_2, S_1, S_0 and C_{in}.
* Arithmetic operations are selected with $S_3S_2 = 00$
* The logic operations are selected with $S_3S_2 = 01$
  → The i/p carry has no effect during the logic operations and is marked with don't care x's.
* The shift operations are selected with $S_3S_2 = 10$ and $S_3S_2 = 11$

Function table for arithmetic logic shift unit:

Function Table for Arithmetic Logic Shift Unit

Operation Select

S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	F = A	Transfer A
0	0	0	0	1	F = A + 1	Increment A
0	0	0	1	0	F = A + B	Addition
0	0	0	1	1	F = A + B + 1	Add with Carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with Borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	F = A – 1	Decrement A
0	0	1	1	1	F = A	Transfer A

Function Table for Arithmetic Logic Shift Unit

Operation Select

S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	1	0	0	x	$F = A \wedge B$	AND
0	1	0	1	x	$F = A \vee B$	OR
0	1	1	0	x	$F = A \oplus B$	XOR
0	1	1	1	x	F = A	Complement A
1	0	x	x	x	F = shr A	Shift-Right A into F
1	1	x	x	x	F = shl A	Shift-Left A into F

BASIC COMPUTER ORGANIZATION

⊛ Instruction formats :-

→ An Instruction can be represented in a specific format. usually a computer will have a variety of instruction code formats.

→ It is the function of the control unit with in the CPU to find each instruction code and provide the necessary control functions needed to process the instruction.

→ The bits of the instruction are divided in to groups called fields.

The most common fields found in instruction formats are.

1. opcode : opcode is an operation code field that specifies the operation to be performed.
2. Address field : An address field that designates a memory address or a processor register.
3. Mode field : A mode field specifies the way the operand or the effective address is determined.

→ Computers may have instructions of different lengths containing different number of addresses.

→ The number of address fields in the instruction format of a computer depends on the internal organization of its registers.

→ most Computers will have one of this three types of CPU organizations.

1. Single accumulator organization.
2. General register organization.
3. Stack organization.

1. Single Accumulator organization:

→ All operations are performed with an implied accumulator register.

→ The Instruction format in this type of computer uses one address field.

Example: ADD X

where X is the address of the operand.

→ The ADD instruction in this case results in the operation

AC ← AC + M[X].

where AC is the accumulator register and

M[X] is the memory word located at address X.

2. General Register organization:

→ In this type of compoters, the instruction format needs three register address fields.

→ Thus the instruction for an arithmetic addition may be written in an assembly language as,

ADD R1, R2, R3.

→ The above instruction denotes the operation, R1 ← R2 + R3.

→ The number of address fields in the instruction can be reduced from three to two, if the destination register is the same as one of the source registers.

→ Thus, the above instruction can be written as,

ADD R1, R2.

This instruction denotes the operation, $R1 \leftarrow R1 + R2$.

→ This General register-type computers employ two or three address fields in their instruction format.

3. Stack organization :-

→ The stack organized computers would have PUSH and POP operations instructions which requires only one address field.

Example: PUSH X

The above instruction will push the word at address X to the top of the stack.

→ operation-type instructions do not need an address field in stack-organized computers. This is because the operation is performed on the two items that are on top of the stack.

Ex: ADD

The above operation will pop the two top numbers from the stack, adds the numbers and push's the sum into the stack.

→ most computers will be one of the three types of organizations. But some computers like "Intel 8080 microprocessor" combine features from more than one organizational structure.

→ As instructions may have different number of Addresses, they can be classified in to four types.

i) Three address Instructions.
ii) Two address Instructions
iii) one address Instructions.
iv) zero address Instructions

→ To illustrate various instructions with different number of addresses, lets evaluate the arithmetic statement

$X = (A+B) * (C+D)$, using zero, one, two or three address instructions.

→ Assume that operands are in memory addresses A, B, C and D and result must be stored in memory at address X.

i) Three Address Instructions :

→ Computers with three address instruction formats can use each address field to specify either a processor register or a memory operand.

The program in assembly language that evaluates $X = (A+B) * (C+D)$ is shown Below.

ADD	R1, A, B	R1 ← M[A] + M[B].
ADD	R2, C, D	R2 ← M[C] + M[D].
MUL	X, R1, R2	M[X] ← R1 * R2

→ Here we assume that the computer has two processor registers, R1 and R2.

→ The symbol M[A] denotes the operand at memory address symbolized by A.

Advantage :- It results in short programs when Evaluating arithmetic expressions.

Disadvantage :- The Binary coded instructions require too many bits to specify three addresses.

ii) Two Address Instructions:

→ Two-Address instructions are the most common in commercial computers.

→ Here, again each address field can specify either a processor register or a memory word.

→ The program to Evaluate $X = (A+B) * (C+D)$ is as follows.

MOV	R1, A	R1 ← M[A]
ADD	R1, B	R1 ← R1 + M[B].
MOV	R2, C	R2 ← M[C].
ADD	R2, D	R2 ← R2 + M[D].
MUL	R1, R2	R1 ← R1 * R2
MOV	X, R1	M[X] ← R1

→ The "MOV" instruction moves the operands to and from memory and processor registers.

iii) One-Address Instructions:

→ One-Address instructions use an implied accumulator (AC) register for all data manipulations.

→ The program to Evaluate $X = (A+B) * (C+D)$ is,

LOAD	A	AC ← M[A].
ADD	B	AC ← AC + M[B].
STORE	T	M[T] ← AC
LOAD	C	AC ← M[C].
ADD	D	AC ← AC + M[D].
MUL	T	AC ← AC * M[T].
STORE	X	M[X] ← AC.

→ 'T' is the address of a temporary memory location required for storing the intermediate result.

iv) Zero - Address Instructions :-

→ The Stack-organized computers does not use an address field for the instructions ADD and MUL.

→ However, PUSH and POP instructions need an address field to specify the operand.

→ The program for $X = (A+B) * (C+D)$ is given by,

PUSH	A	TOS ← A
PUSH	B	TOS ← B
ADD		TOS ← (A+B)
PUSH	C	TOS ← C
PUSH	D	TOS ← D
ADD		TOS ← (C+D)
MUL		TOS ← (C+D) * (A+B)
POP	X	M[X] ← TOS

→ TOS stands for Top of Stack.

→ The name "zero - Address" is given to this type of Computer, because of the absence of an address field.

RISC Instructions:

→ In Addition to these four types of Computers, There are RISC type of Computers.

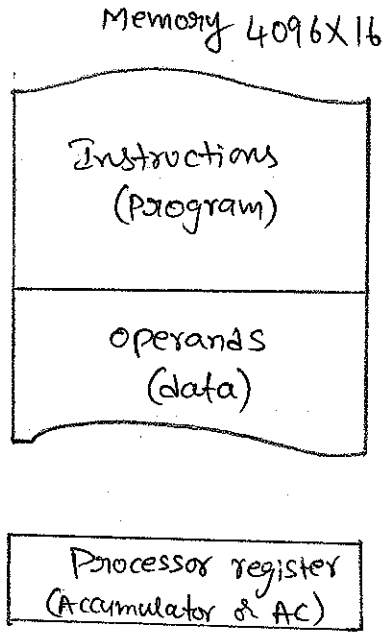
→ In this RISC type, the instruction set is restricted to the use of LOAD and STORE instructions when communicates b/w memory and CPU.

LOAD	R1, A	R1 ← M[A]
LOAD	R2, B	R2 ← M[B]
LOAD	R3, C	R3 ← M[C]
LOAD	R4, D	R4 ← M[D]
ADD	R1, R1, R2	R1 ← R1+R2
ADD	R3, R3, R4	R3 ← R3+R4
MUL	R1, R1, R3	R1 ← R1*R3
STORE	X, R1	M[X] ← R1

⊛ Instruction codes:-

* The internal organization of a digital system is defined by the sequence of microoperations it performs on data stored in its registers.
* Every different processor type has its own design. i.e., different registers, buses, microoperations, machine instructions etc.
* The modern processor is a very complex device, it contains many registers, multiple arithmetic units for both integer and floating point calculations, and the ability to pipeline several consecutive instructions to speed Execution etc.
* Therefore, to understand how processors work, we will use a simple processor model called as Basic computer.
* This Basic computer will be used to illustrate the processor organization and the relationship of the RTL model to the higher level computer processor.
* The Basic Computer has two components, a processor and memory.
* The memory has 4096 words in it.

  $\Rightarrow 4096 = 2^{12}$, so it takes 12-bits to select a word in memory.
* Each word is 16 bit long.



Instruction code: An instruction code is a group of bits that instruct the computer to perform a specific operation.

* A computer instruction is a binary code that specifies a sequence of microoperations for the computer.
* Instruction codes together with data are stored in memory.
* The computer reads each instruction from memory and places it in a control register.
* The control then proceeds to execute the instruction by issuing a sequence of microoperations.

* Instruction format:

* The computer instruction is often divided into two parts, each having its own particular interpretation.

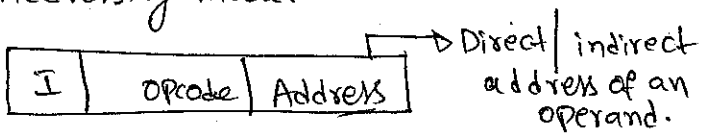
* The two parts of instruction are,

i) opcode
ii) address.

Addressing modes:

* To enlarge the capability and computing power of an instruction, it is desirable to have different types of addressing capabilities in an instruction format.

* we have many number of Addressing modes, but there are two most basic types, given by, i) Direct Addressing mode and
ii) Indirect Addressing mode.



i) Direct addressing mode:

* This Direct addressing mode uses direct addresses of the operands.

* The Address part of the instruction format, is given the address in memory where data is stored.

* The Direct addressing mode is represented to the processor by assigning bit '0' in 'I'-bit of the instruction code.

* The direct addressing mode is demonstrated by the following diagram.

* An opcode, also known as operation code, specifies the operation for that instruction.

* An Address specifies the registers and/or locations in memory to use for that operation.

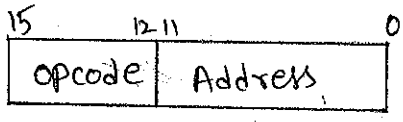


fig: General instruction format.

* In the Basic computer, since the memory contains 4096 ($=2^{12}$) words, we need 12-bits to specify the address used by that instruction.

* In the basic computer, bit_15 of the instruction specifies the addressing mode.

* Addressing modes, '0' for direct addressing mode.
'1' for indirect addressing mode.

* The instruction format in Basic Computer is given as follows

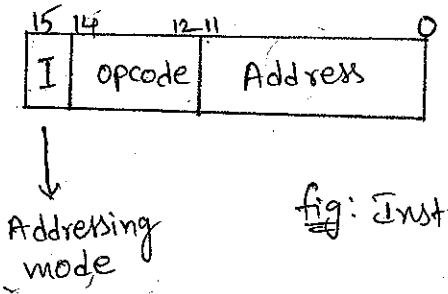


fig: Instruction format.

* Since, in a basic computer, Instruction is 16-bit long,

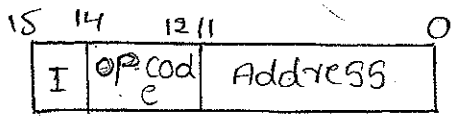
No. of bits for address = 12
No. of bits for opcode = 3
No. of bits for Addressing mode = 1 (Represented by Symbol I).

ii) Indirect Addressing mode:-

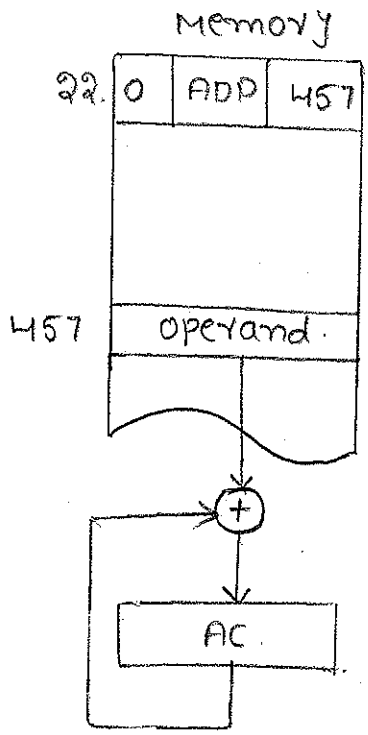
* This Indirect Addressing mode uses the indirect address of the operands.
* The Address part of the Instruction code specifies a memory location, where the address of the operand is located.
* The Indirect Addressing mode is represented to the processor by assigning '1' in the 'I'-bit position of the instruction code format.
* The Indirect addressing mode is demonstrated by the following diagram.

(129)

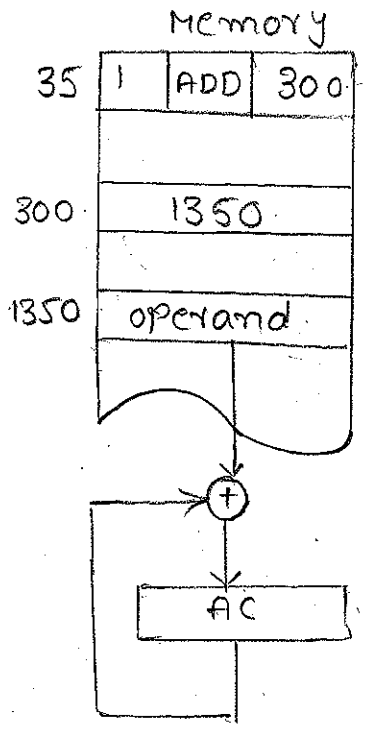
Fig 5-2 Demonstration of direct & indirect address



(a) Instruction format



(b) Direct address



(c) Indirect address

⊛ Computer Registers :-

* Computer instructions are normally stored in consecutive memory locations and are executed sequentially one at a time.
* The control reads an instruction from a specific address in memory and executes it.
* It then continues by reading the next instruction in sequence and executes it, and so on.
* This type of instruction sequencing needs a counter to calculate the address of the next instruction.
* And it is also necessary to provide a register in the control unit for storing the instruction code after it is read from memory.
* Therefore, a computer needs many processor registers which holds instructions, address, data etc.
* The following are the various computer registers used in a basic computers.

a) Program counter
b) Address register
c) Data register
d) Accumulator
e) Temporary register
f) Instruction register
g) Input register
h) output register

a) Program counter: (PC):

* The processor has a register, the program counter (PC) that holds the memory address of the next instruction to be fetched.

* Since the memory in the basic computer has only 4096 locations, the PC only needs 12 bits.

* TO hold the next instruction, PC is incremented by '1'.

b) Address registers: (AR):

* The Address register is a processor register that keeps track of what locations in memory it is addressing.

* The AR, is a 12 bit register in the Basic computer.

c) Data register: (DR):

* when an operand is found, using either direct or indirect addressing, it is placed in the Data register (DR). The processor then uses this value as data for its operations.

* Data registers uses 16-bits of length.

d) Accumulator (AC):

* Accumulator (AC) is a single general purpose register that is used in the basic computers.

* Accumulator (AC) will have 16-bits of information.

e) Temporary register: (TR).

* In Basic computers, some times the processor will need a register to store intermediate results or other temporary data. The Temporary register is used for this purpose.

* A Temporary register holds 16-bits of data.

f) Instruction register (IR):

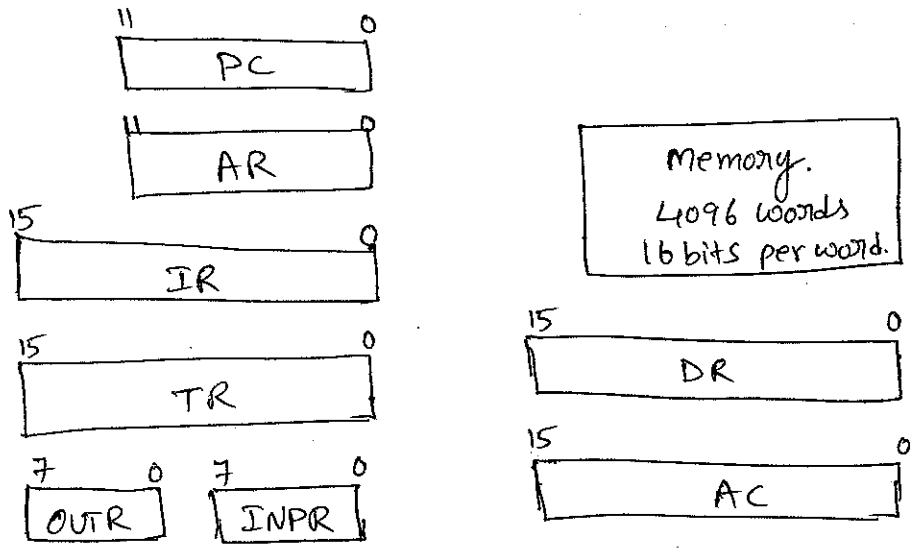
* The Instruction which was read from the memory is placed in the Instruction register.

* An Instruction register holds 16-bits of data.

g) Input and output registers:

* The Basic computer uses a very simple model of input/output operations.
  → The input devices are considered to send 8 bits of character data to the processor
  → The processor can send 8 bits of character data to output devices.
* The Input register (INPR) holds an 8 bit character gotten from an input device.
* The output register (OUTR) holds an 8-bit character to be send to an output device.

The following diagrams shows the memory and various registers of a basic computer.



The following table shows the list of registers and their functions.

Register Symbol	Number of bits	Register Name	Function.
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds Address for memory
AC	16	Accumulator	processor register
IR	16	Instruction register	Holds instruction code
PC	12	program counter	Holds Address of instruction
TR	16	Temporary register	Holds Temporary data
INPR	8	Input register	Holds input character
OUTR	8	output register.	Holds output character.

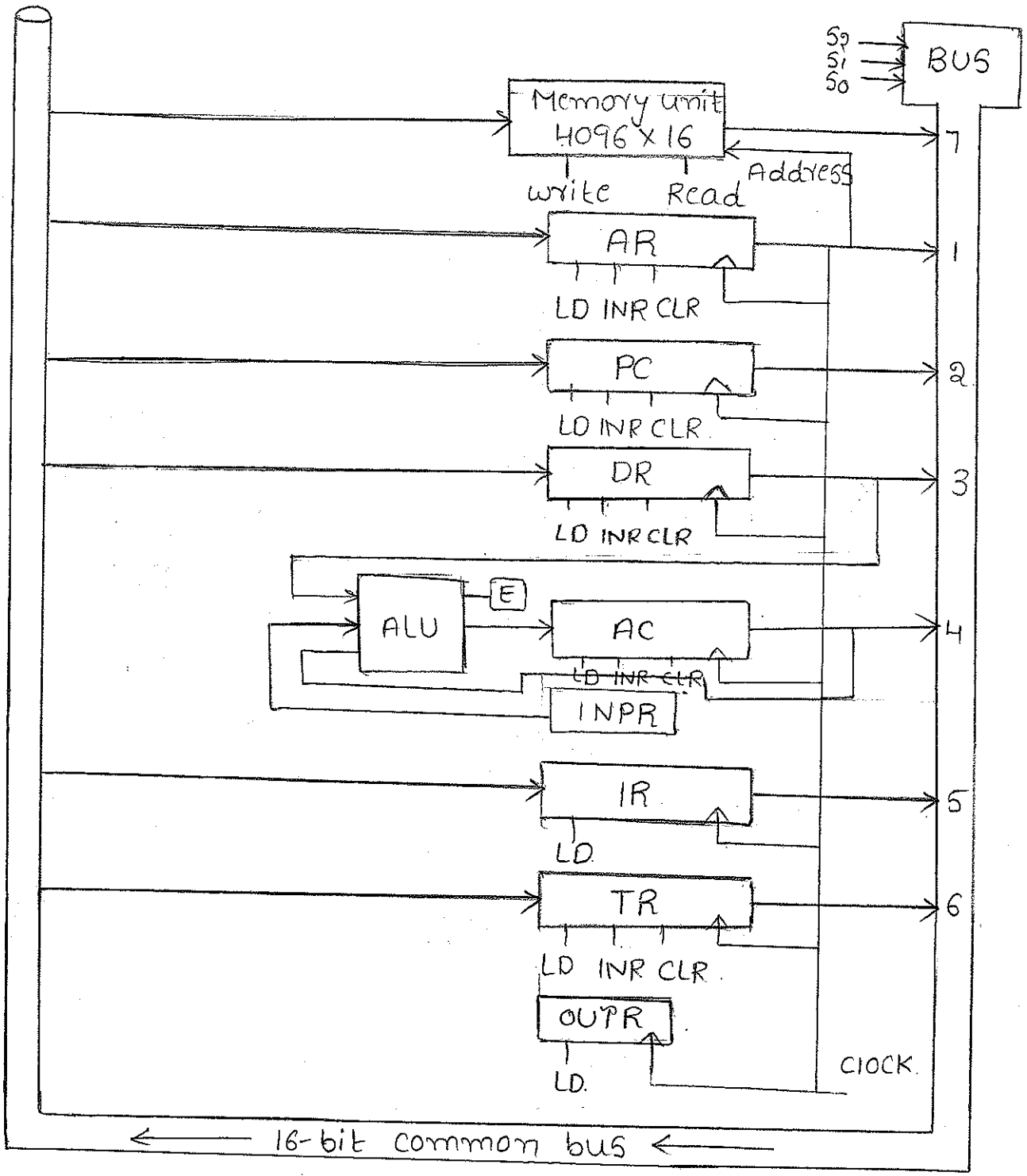
Common Bus System:

* The Basic computer has eight registers, a memory unit, and a control unit.

* So, Paths must be provided to transfer information from one register to another and between memory and registers.
* The no. of wires will be excessive if connections are made between the outputs of each register and the inputs of the other registers.
* A more efficient scheme for transferring information in a system with many registers is to use a common bus.
* The connection of the registers and memory of the basic computer to a common bus system is shown in the figure.

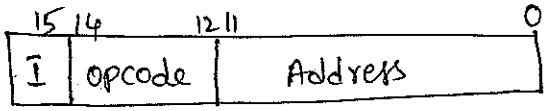
* The output of seven registers and memory are connected to the common bus.
* The specific output is determined from the binary value of the selection variables S_2, S_1 and S_0.
* Five registers have three control inputs LD (Load), INR (Increment) and CLR (Clear).
* Adder and logic circuit has three sets of inputs.
* one set of 16-bit inputs came from the outputs of AC. They are used to implement register microoperations such as complement AC and Shift AC.
* Another set of 16-bit inputs come from the data register DR.
* The inputs from DR and AC are used for arithmetic and logic microoperations such as ADD DR to AC etc.
* The Result of an addition is transferred to AC and the end carryout of the addition is transferred to flip-flop 'E'.
* A third set of 8-bit inputs come from the input register INPR.

COMMON BUS SYSTEM



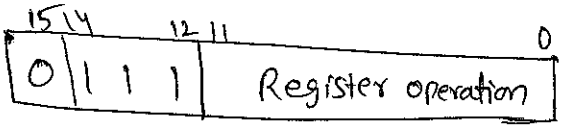
Computer Instructions:

* The Basic computer has three instruction code formats, as shown below.
* Each format has 16 bits.



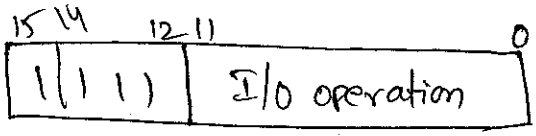
(Opcode = 000 through 110)

(a) Memory reference instruction.



(Opcode = 111, I = 0)

(b) Register-reference instruction



(Opcode = 111, I = 1)

(c) Input-output Instruction.

(a) Memory-reference instructions:

* The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered.
* A memory-reference instruction uses 12 bits to specify an address and one bit to specify the Addressing mode I.
* I is equal to '0' for direct address and I is equal to '1' for indirect address.

(b) Register-reference instructions:

* The Register reference instructions are recognized by

the operation code 111 with a '0' in the left most bit (bit 15) of the instruction.

* A Register-reference instruction specifies an operation on the AC Register.
* An operand from memory is not needed, therefore, the other 12 bits are used to specify the operation or test to be executed.

(c) Input-output Instructions:

* An input-output instruction does not need a reference to memory and is recognized by the operation code 111 with a '1' in the leftmost bit of the instruction.
* The Remaining 12 bits are used to specify the type of input-output operatim performed.

Recognizing

→ The type of instruction is recognized by the Computer Control from the four bits in positions 12 to 15 of the instruction.

* If the three opcode bits in positions 12 to 14 are not equal to 111, the instruction is a memory reference type. and the bit in position 15 is taken as the addressing mode I.

* If the 3-bit opcode is equal to 111, control then inspects the bit in position 15. If this bit is 0, the instruction is a register-reference type.

* If the bit in position 15 is 1, the instruction is a Input-output type.

→ The instructions for the computer are listed below.

(a) Memory-reference instructions:

Symbol	Hexadecimal Code		Description.
	I=0	I=1	
AND	0XXX	8XXX	AND memory word to AC
ADD	1XXX	9XXX	Add memory word to AC
LDA	2XXX	AXXX	Load memory word to AC
STA	3XXX	BXXX	Store content of AC in memory.
BUN	4XXX	CXXX	Branch unconditionally.
BSA	5XXX	DXXX	Branch and save return address.
ISZ	6XXX	EXXX	Increment and SKIP if zero.

* The hexadecimal code is equal to the equivalent hexadecimal number of the binary code used for the instruction.

* By using the hexadecimal equivalent we reduce the 16 bits of an instruction code to four digits with each hexadecimal digit being equivalent to four bits.

* The address part is denoted by three x's and stand for the three hexadecimal digits corresponding to the 12-bit address.

* The last bit of the instruction is designated by the symbol I.

* When, I=0, the last four bits of an instruction have a hexadecimal digit equivalent from 0 to 6, Since the last bit is '0'.

* When, I=1, the hexadecimal digit equivalent of the last four bits of the instruction ranges from 8 to E, Since the last bit is '1'.

(b) Register_reference instructions.

Symbol	Hexadecimal Code	Description.
CLA	7800	clear AC
CLE	7400	clear E
CMA	7200	Complement AC
CME	7100	Complement E
CIR	7080	circular right AC and E
CIL	7040	circular left AC and E
INC	7020	Increment AC
SPA	7010	Skip next instruction if AC positive
SNA	7008	" " " " " negative
SZA	7004	" " " " " zero
SZE	7002	" " " " " E is 0
HLT	7001	Halt Computer.

* Register reference instructions use 16 bits to specify an operation.

* The leftmost four bits are always 0111, which is equivalent

to hexadecimal 7.

* The other three hexadecimal digits give the binary equivalent of the remaining 12 bits.

(c) Input-output instructions:

Symbol	Hexadecimal code	Description.
INP	F800	Input character to AC
OUT	F400	Output character from AC
SKI	F200	Skip on input flag
SKO	F100	Skip on output flag
ION	F080	Interrupt on
IOF	F040	Interrupt off.

* The Input-output instructions also use all 16 bits to specify an operation.

* The last four bits are always 1111, equivalent to hexadecimal F.

Instruction Set Completeness:

* A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable.

* The set of instructions are said to be complete, if the computer includes a sufficient number of instructions in each of the following categories.

i) Arithmetic, logic and shift instructions.
ii) Instructions for moving information to and from memory.
iii) Program control instructions together with instructions that check status conditions.
iv) Input and output Instructions.

Instruction cycle:

* A program in the memory unit of the computer consists of a sequence of instructions.
* The program is executed in the computer by going through a cycle for each instruction.
* Each instruction cycle inturn is subdivided into a sequence of subcycles or phases.
* In the basic computer each instruction cycle consists of the following phases:
  1. Fetch an instruction from memory.
  2. Decode the instruction.
  3. Read the Effective address from memory if the instruction has an indirect address.
  4. Execute the instruction.
* After the completion of Step 4, the control goes back to Step1 to fetch, decode and Execute the next instruction.

Fetch and decode:

* Initially the program counter PC is loaded with the address of the first instruction in the program.
* The sequence counter 'SC' is cleared to 0, providing a decoded timing signal T_0.

* After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T_0, T_1, T_2 and so on.

* The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

T_0: AR ← PC

T_1: IR ← M[AR], PC ← PC+1

T_2: $D_0 \ldots\ldots D_7$ ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15)

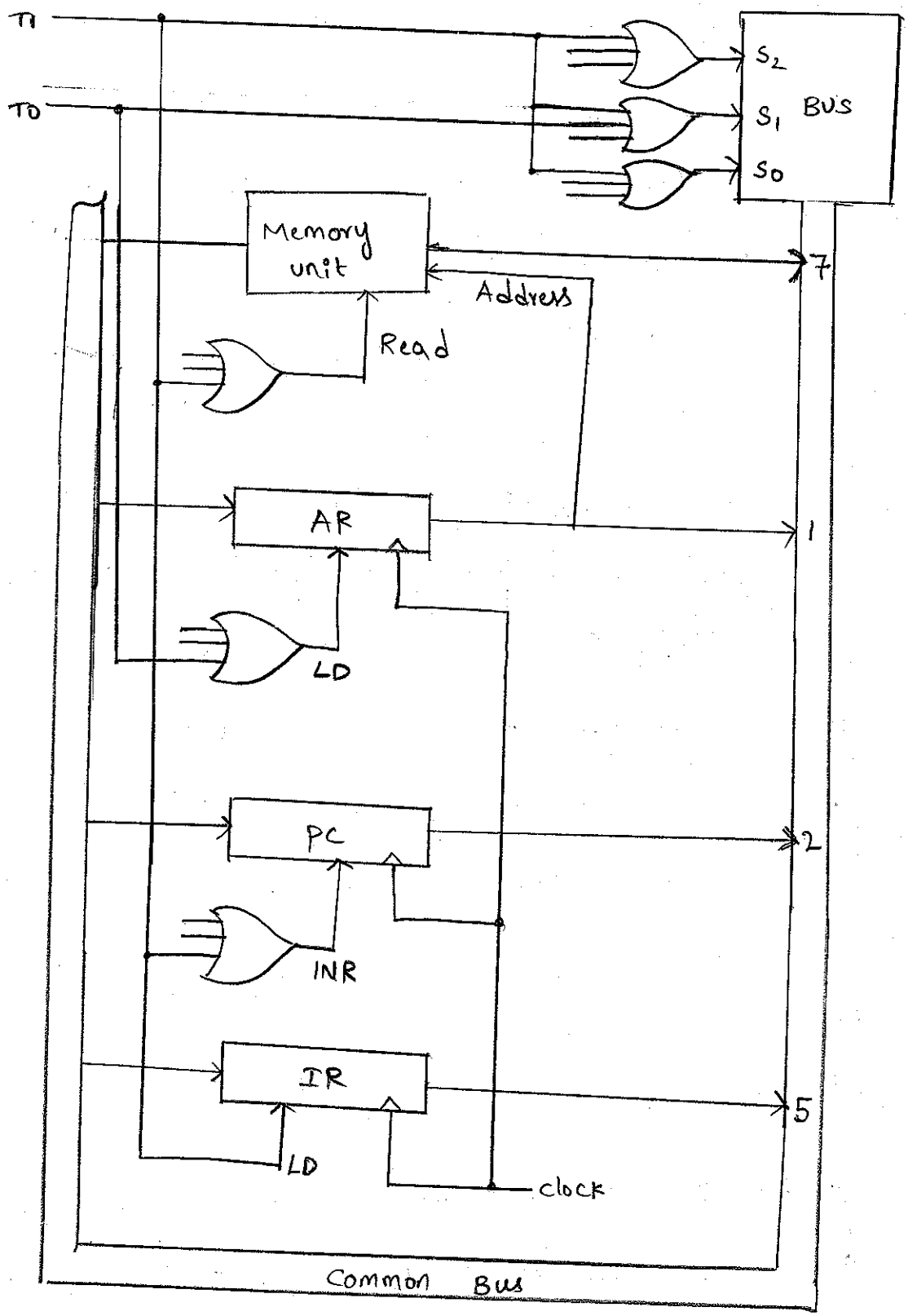
* Since only AR is connected to the address inputs of memory, it is necessary to transfer the address from PC to AR during the clock transition having timing signal T_0.

* The instruction read from the memory is then placed in the instruction register IR with the clock transition associated with timing signal T_1.

At the same time PC is incremented by one, to prepare it for the address of the next instruction in the program.

* At time T_2, the operation code in IR is decoded, the indirect bit is transferred to flip-flop I, and the address part of the instruction is transferred to AR.

* The following figure shows how the first two registers transfer statements are implemented in the bus system.



Register transfers for the fetch phase

* To provide the data path for the transfer of PC to AR we must apply timing signal T_0 to ~~activate~~ achieve the following Connection:

1) Place the content of PC on to the bus by making the bus selection inputs $S_2S_1S_0$ equal to 010

2) Transfer the content of the bus to AR by enabling the LD input of AR.

* The next clock transition initiates the transfer from PC to AR Since $T_0 = 1$. In order to implement the second Statement

$$T_1 : \quad IR \leftarrow M[AR], \quad PC \leftarrow PC+1.$$

* By using timing signal T_1, the following connections are provided in the bus System.

i) Enable the read input of memory.

ii) Place the Content of memory on to the bus by making $S_2S_1S_0 = 111$.

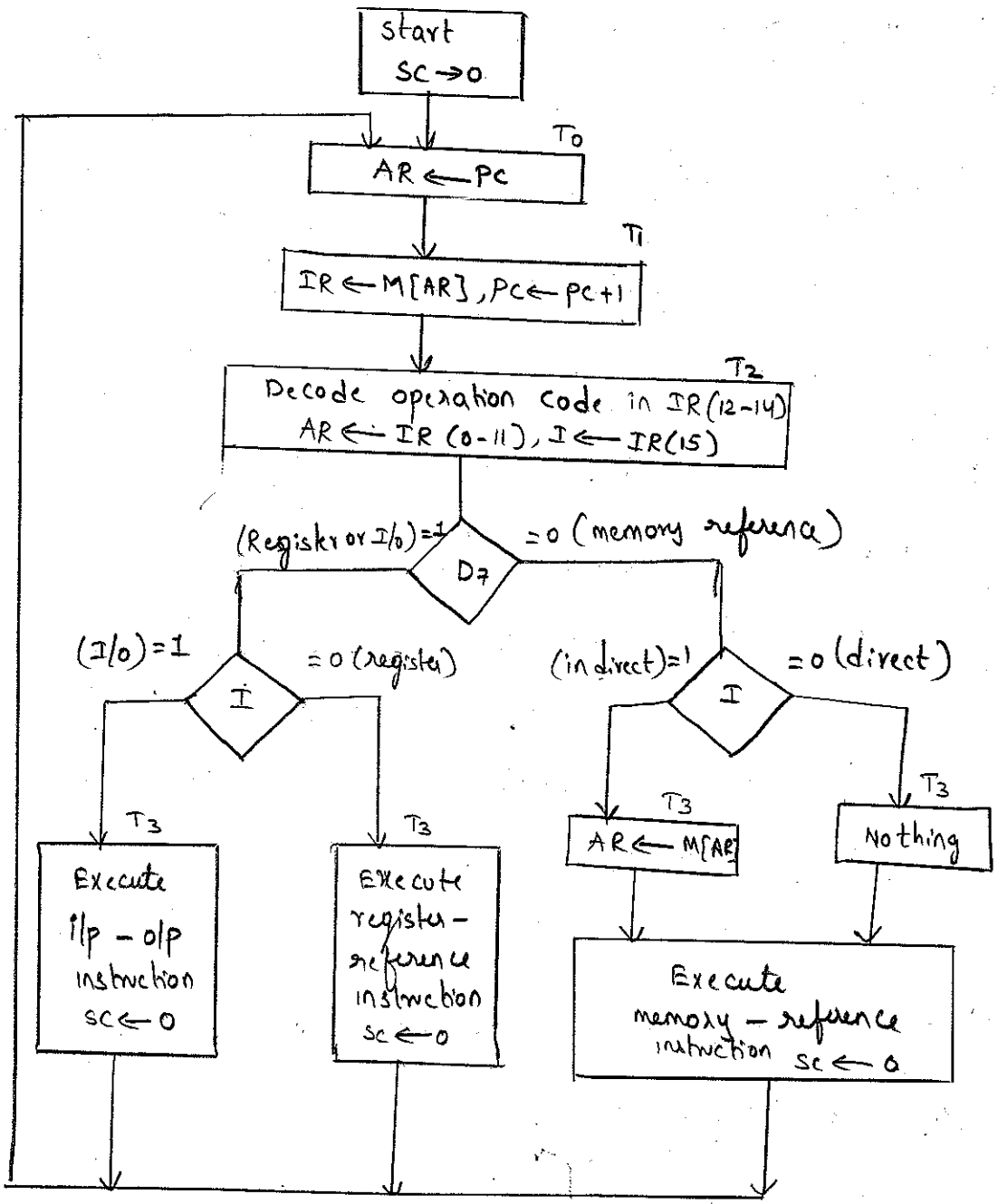
iii) Transfer the content of the bus to IR by enabling the LD input of IR.

iv) Increment PC by enabling the INR input to PC.

* The next clock transition initiates the read and increment operations Since $T_1 = 1$.

Determine the type of instruction.

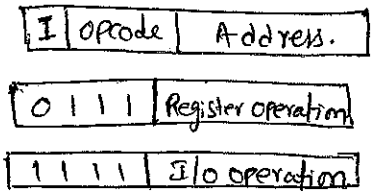
* The timing signal that is active after the decoding is T_3.
* During time T_3, the control unit determines the type of instruction that was just read from memory.
* The flow chart shows how the control determines the instruction type after decoding.



flowchart for Instruction cycle.

The three possible instruction types are,

i) memory-reference instruction
ii) Register-reference instruction
iii) Input-output instruction



* Decoder output D_7 is equal to 1 if the operation code is equal to binary 111.
* we can determine that if $D_7=1$, the instruction must be register-reference (or) Input-output instruction.
* If $D_7=0$, the operation code lies in 000~110, which means that a memory-reference instruction.
* control then inspects the value of first bit of the instruction which is now available in flip-flop I.
* If $D_7=0$ and $I=1$, we have a memory-reference instruction with an indirect addressing mode.
* It is then necessary to read the effective address from memory.
* The microoperation for the indirect address condition can be Symbolized as,

$$AR \leftarrow M[AR].$$

* Initially, AR holds the address part of the instruction. This address is used during the memory read operation.
* The word at the given address AR is read from memory and placed on the common bus.
* The LD input of AR is then enabled to receive the indirect address that resided in the 12 least Significant bits of the memory word.

* The selected operation is activated with the clock transition associated with timing signal T_3.

* This can be symbolized as follows.

$D_7' I T_3$: $AR \leftarrow M[AR]$

$D_7' I' T_3$: Nothing

$D_7 I' T_3$: Execute a register-reference instruction.

$D_7 I T_3$: Executes input-output instruction.

Register-reference instructions:

* Register reference instructions are recognized when $D_7 = 1, I = 0$.

* These instructions use bits 0 to 11 of the instruction code to specify one of 12 instructions.

* These bits are available in IR(0-11), and transferred to AR during T_2.

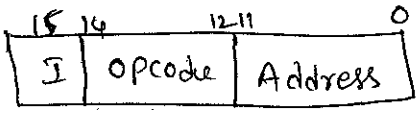
* for Example, the instruction CLA has the hexadecimal code 7800.

$\Rightarrow$ 0111 1000 000 000

(*) Memory-reference Instructions:

* There are 7 memory-reference instructions.
* The decoded output D_i for $i = 0,1,2,3,4,5$ and 6 from the operation decoder that belongs to each instruction is included in the table.

* Instruction format ⟹



* The effective address of the instruction is in the address register 'AR' and was placed there during timing signal T_2, when I=0 (or) during timing signal T_3 when I=1.
* The Execution of this memory reference instructions starts with timing signal T_4.
* The following are the 7-memory reference instructions.

Symbol	Operation Decoder	Symbolic description.
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR]$, $E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC$, $PC \leftarrow AR+1$
ISZ	D_6	$M[AR] \leftarrow M[AR]+1$ If $M[AR]+1=0$ then $PC \leftarrow PC+1$.

Now, lets see the operation of each instruction and the list of control functions and microoperations needed for their Execution.

i) AND to AC:

* This is an instruction that performs AND logic operation on pairs of bits in AC and the memory word specified by the effective address.
* The result of the operation is transferred to AC.
* The microoperations that Executes this instruction are:

D_0T_4 : $DR \leftarrow M[AR]$ Read operation.

D_0T_5 : $AC \leftarrow AC \wedge DR, SC \leftarrow 0$.

* Two timing signals are needed to Execute the instruction.
* The clock transition associated with timing signal T_4 transfers the operand from memory to DR.
* The clock transition associated with the next timing signal T_5 transfers to AC, the result of the AND logic operation between the contents of DR and AC.
* The same clock transition clears SC to 0, to start a new instruction cycle.

ii) ADD to AC:

* This instruction adds the content of the memory word specified by the effective address to the value of AC.
* The sum is transferred in to AC and the output carry Cout is transferred to the E-flip flop.

* The microoperations needed are,

D_1T_4: DR ← M[AR].

D_1T_5: AC ← AC + DR, E ← C_{out}, SC ← 0.

iii) LDA: Load to AC

* This instruction transfers the memory word specified by the Effective address to AC.

* The microoperations needed to Execute this instruction are

D_2T_4: DR ← M[AR].

D_2T_5: AC ← DR, SC ← 0.

* It is necessary to read the memory word in to DR first and then transfer the content of DR in to AC.

iv) STA: Store AC:

* This instruction stores the contents of AC in to the memory word specified by the effective address.

* we can Execute this instruction with one microoperation as follows.

D_3T_4: M[AR] ← AC, SC ← 0.

v) BUN: Branch unconditionally:

* This instruction transfers the program to the instruction specified by the effective address.

* Remember that PC holds the address of the instruction to be read from memory in the next instruction cycle. PC is incremented at time T_1 to prepare it for the address of the next instruction in the program sequence.

* The BUN instruction allows the programmer to specify

an instruction out of sequence and we say that the program branches or jumps unconditionally.

* The instruction is executed with one microoperation as follows.

$$D_4T_4: PC \leftarrow AR, SC \leftarrow 0.$$

vi) BSA: Branch and save return Address:

* This instruction is useful for branching to a portion of the program called a subroutine.
* when executed, the BSA instruction stores the address of the next instruction in sequence (which is available in PC) in to a memory location specified by the effective address.
* The effective Address plus One is then transferred to PC to serve as the address of the first instruction in the subroutine
* This operation is specified with the following register transfer.

$$M[AR] \leftarrow PC, \quad PC \leftarrow AR+1.$$

* The BSA instruction performs the function usually referred to as a "subroutine call".
* The Indirect BUN instruction at the end of the subroutine performs the function referred to as a "subroutine return".

vii) ISZ: Increment and skip if zero:

* This instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1.
* The programmer usually stores a negative number (in 2's complement)

in the memory word.

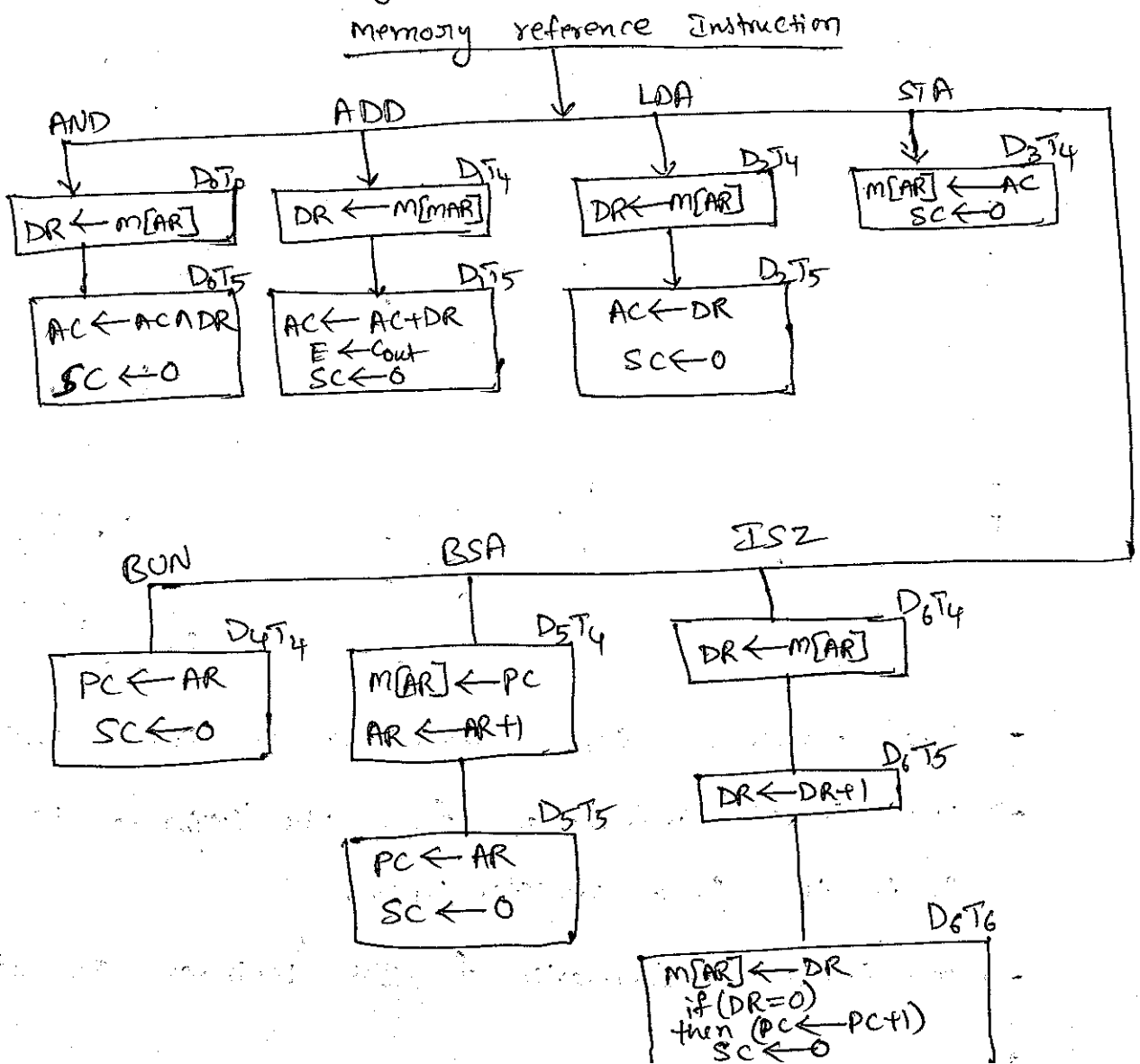
* As this negative number is repeatedly incremented by one, it eventually reaches the value zero.
* At that time PC is incremented by one in order to skip the next instruction in the program.
* Since it is not possible to increment a word inside the memory, it is necessary to read the word into DR, increment DR, and store the word back into memory.
* The microoperations are

D_6T_4: DR ← M[AR]

D_6T_5: DR ← DR+1

D_6T_6: M[AR] ← DR, if (DR=0) then (PC ← PC+1), SC ← 0.

Flowchart for memory-reference instructions:



(*) Input - output and Interrupt:

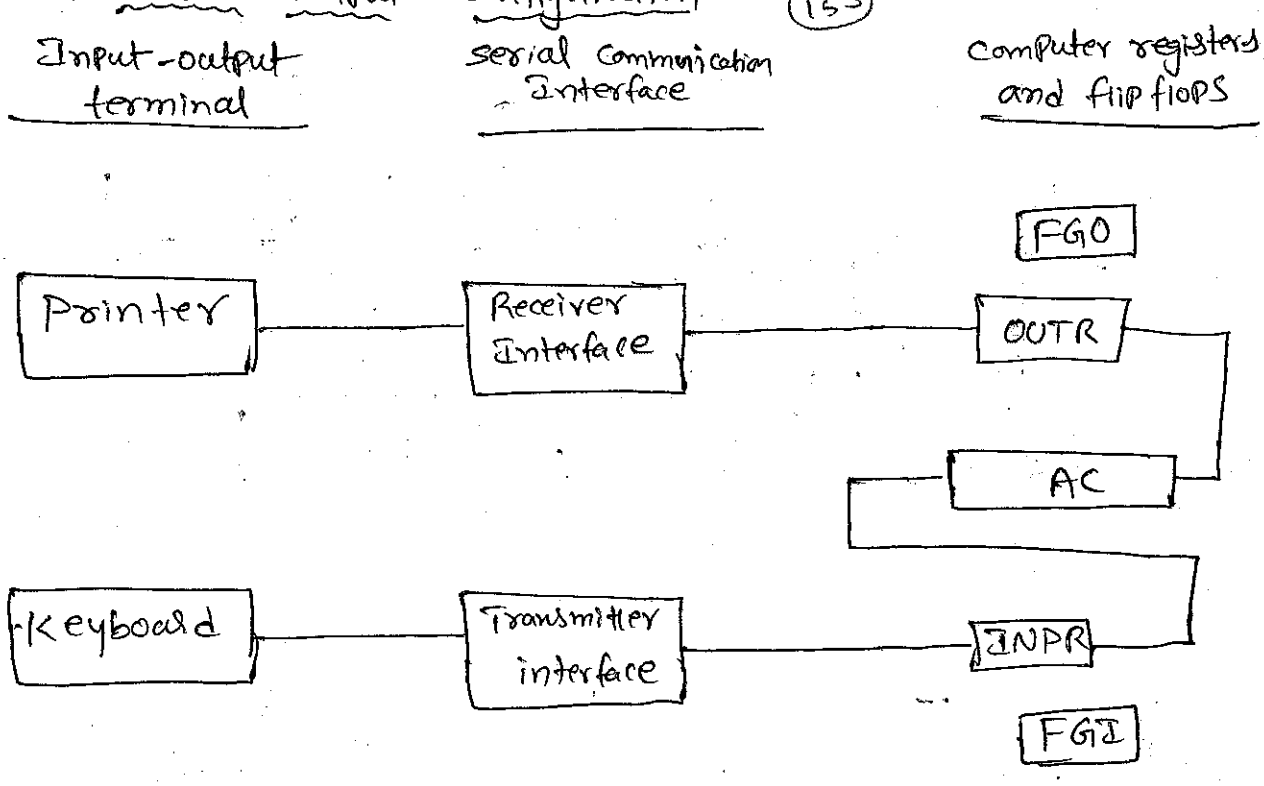
→ Instructions and data stored in memory must come from some input device.

→ Computational results must be transmitted to the user through some output device.

→ Computer have many input and output devices.

→ for this illustration of input - output configuration, lets consider Keyboard and printer

fig: Input - output Configuration (153)



→ The terminal sends and receives serial information.

→ The serial information from the keyboard is shifted in to the input register INPR.

→ The serial information for the printer is stored

in the output register OUTR.

→ These two registers communicate with a communication interface serially and with the AC in parallel.

→ The transmitter interface receives serial information from the keyboard and transmits in to INPR.

→ The receiver interface receives information from OUTR and sends it to the printer, serially.

→ The INPR consists of 8-bits. The 1-bit input flag FGI is a control flip-flop.

→ The flag bit is set to '1' when new info is available in the input device. and is cleared to '0' when the information is accepted by the computer.

→ Initially the FGI is set to '0'. when a key is pressed in a keyboard, the 8-bit alpha numeric code is shifted to INPR. and the flagbit is set to '1'.

→ The computer checks the flag bit. If it is '1', the information from INPR is transferred to AC.

→ The output register works similarly but the direction of information flow is reversed.

→ Initially the output flag FGO is set to '1'. The computer checks the flag bit, if it is '1', the information from AC is transferred in parallel to OUTR and FGO is cleared to '0'.

UNIT 3

Addressing modes :-

→ Instruction format with mode field is given by,

Mode	Opcode	Address.

→ The opcode, i.e, the operation field of an instruction specifies the operation to be performed.

→ This operation must be Executed on some data stored in computer registers or memory words.

→ The way in which the operands are ~~chosen~~ Chosen during program execution is dependent on the addressing mode of the instruction.

The way in which the operand is taken from register/memory is named as "Addressing mode"

→ The availability of various addressing modes gives flexibility of writing programs in assembly language program.

→ Generally, the control unit of a computer is designed to go through an instruction cycle which has three major phases.

1. Fetch the instruction from memory.
2. Decode the instruction
3. Execute the instruction.

→ There is a register called Program counter (PC) that keeps track of the instructions in the program stored in memory.

→ The decoding step determines the operation to be performed.

→ Finally the instruction gets executed and control goes to step 1.

→ So, to perform the Execution of any instruction, we use Addressing mode field which is used to locate the operands needed for the operation.

The Various Addressing modes are,

① Implied mode:

In this mode the operands are specified implicitly in the definition of the instruction.

Ex: i) Complement Accumulator.
ii) zero-Address instructions in a stack organized computers are implied mode instructions.

② Immediate mode:-

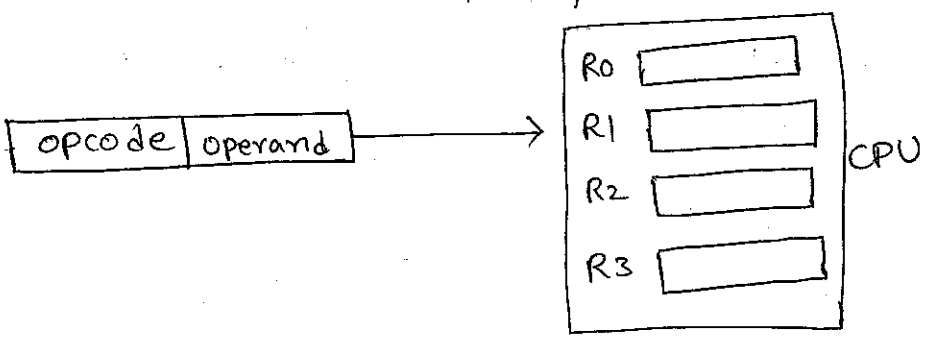
→ In this mode the operand is specified in the instruction itself.

→ In other words, an immediate mode instruction has an operand field rather than an address field.

③ Register mode:-

→ In this mode the operands are in registers that reside with in the CPU.

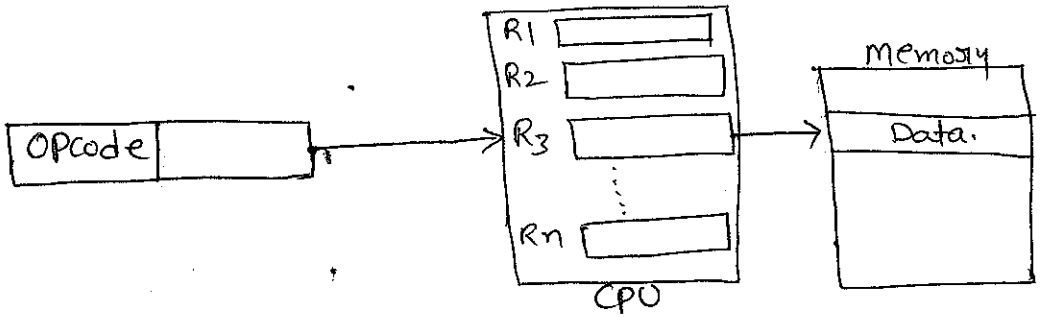
→ A K-bit field can specify any one of 2^K Registers.



④ Register Indirect mode:-

→ In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory.

→ In other words, the selected register contains the address of the operand rather than the operand itself.



⑤ Auto increment / Auto decrement mode :-

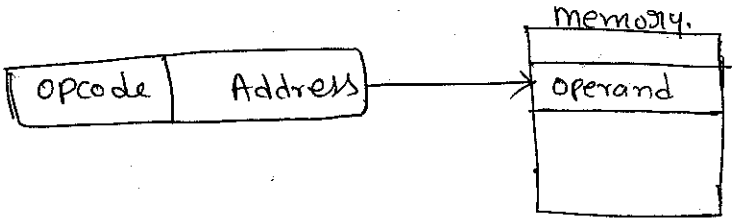
→ This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.

→ When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table.

⑥ Direct addressing mode :-

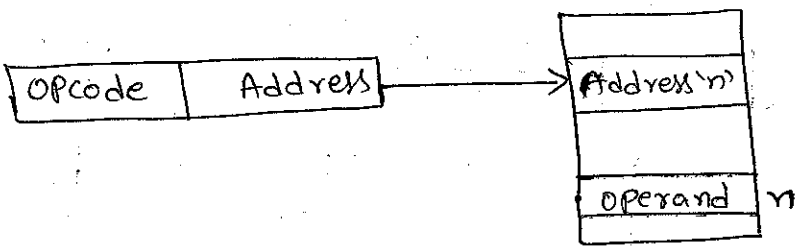
→ In this mode the effective address is equal to the address part of the instruction.

→ The operand present in memory and its address is given directly by the address field of the instruction.



⑦ Indirect addressing mode :-

In this mode the address field of the instruction gives the address, where the effective address is stored in memory.



→ A few addressing modes require that the address field of the instruction is added to the content of a specific register in the CPU.

i.e., Effective address = Address part of instruction + contents of CPU register.

⑧ Relative address mode:-

→ In this mode the content of the program counter is added to the address part of the instruction in order to obtain the Effective address.

Example: Assume that program counter (PC) contains '825' and address part of the instruction contains the number '24'.

So, During fetch phase PC will be incremented by one and becomes '826'.

∴ Effective address = 826 + 24
= 850

⑨ Indexed addressing mode:-

→ In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.

→ The index register is a special CPU register that contains an Index value.

(10) Base register addressing mode:-

In this mode the content of the base register is added to the address part of the instruction to obtain the effective address.

Numerical Example

The following numerical example shows the effect of the various addressing modes.

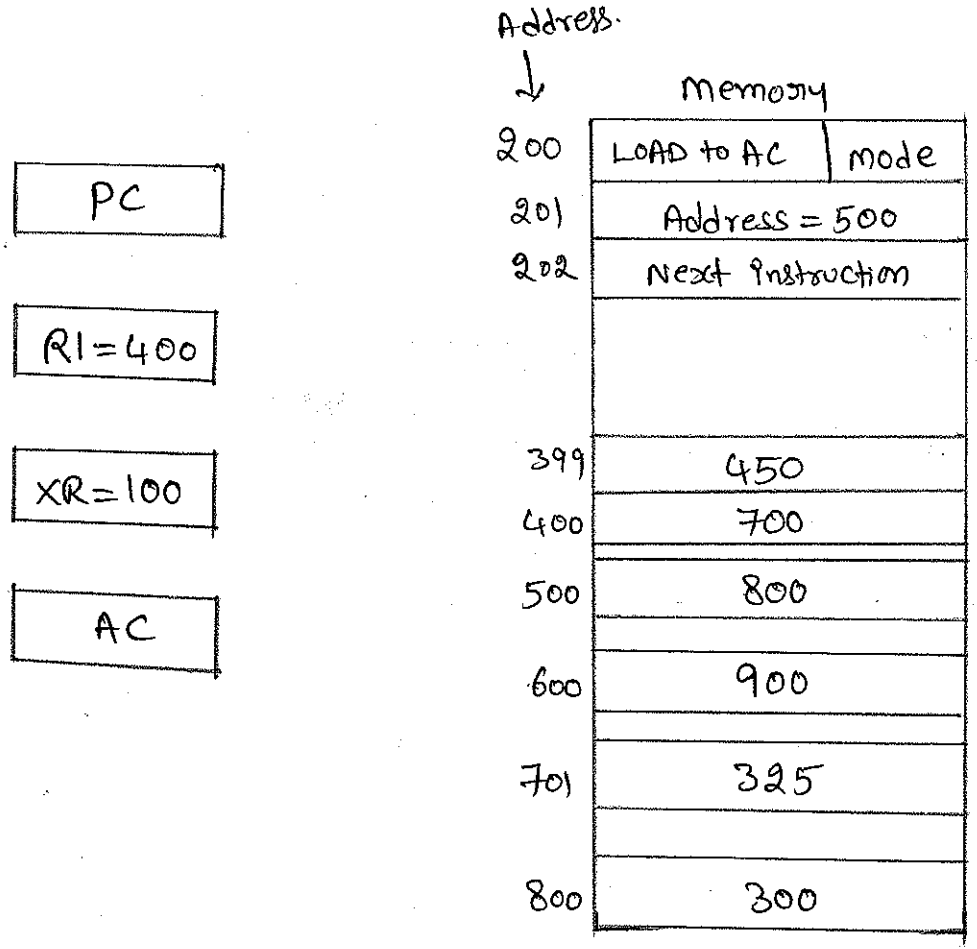


fig: Numerical Example for addressing modes.

→ The two-word instruction at address 200 and 201 is a "Load to AC" instruction with an address field equals to 500.

→ The first word of the instruction specifies the operation code and mode, and the second word specifies the address part.

→ PC has the value 200 for fetching this instruction.

→ The contents of processor register R1 is 400.

→ The contents of an index register XR is 100.

→ AC Receives the operand after the instruction is executed.

The following table shows the values of the effective address and the operand loaded into AC for the nine addressing modes.

Addressing mode	Effective address	Content of AC
① Direct Address	500	800
② Immediate Operand	201	500
③ Indirect Address	800	300
④ Relative Address	701	325
⑤ Indexed Address	600	900
⑥ Register mode	—	400
⑦ Register Indirect	400	700
⑧ Auto increment	400	700
⑨ Auto decrement	399	450

Stack organization:

→ A useful feature that is included in the CPU of most computers is a stack or Last-in, First-out (LIFO) list.

→ A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved.

→ The stack is a memory unit with an address register that can count only after an initial value is loaded in to it.

→ The register that holds the address for the stack is called stack pointer (SP).

→ Its value always points to the top item in the stack.

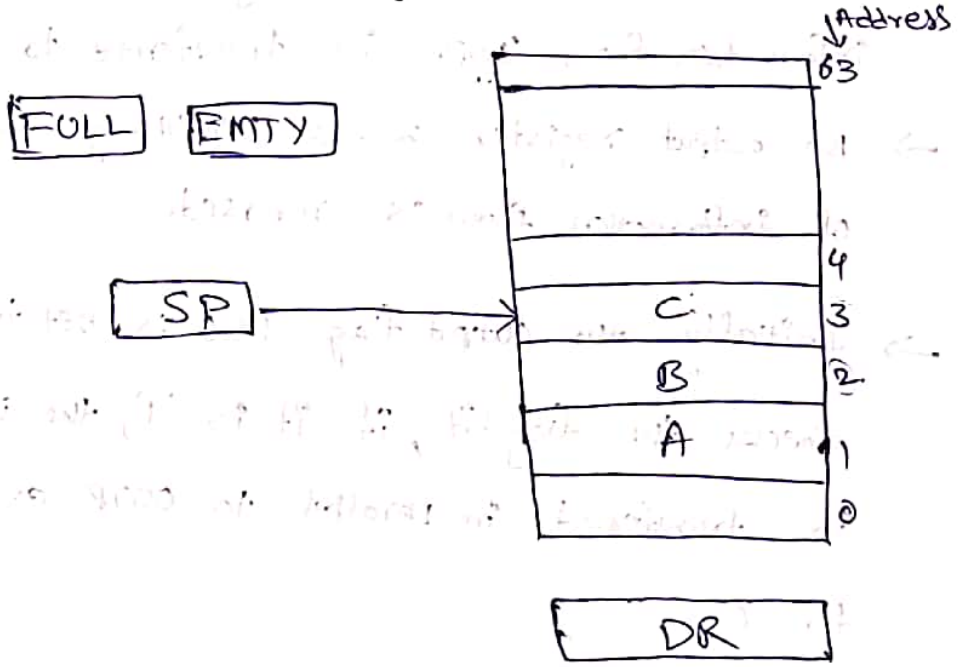
→ The two operations of a stack are the insertion and deletion.

→ The operation of insertion is called 'push', and deletion is called as 'pop'.

Register stack :-

→ A stack can be placed in a portion of a large memory or it can be organized as a collection of registers.

(250)



→ The stack pointer register (SP) contains a binary number whose value is equals to the address of the word that is currently on top of the stack.

→ A, B, C are placed in stack in same order.

→ Item 'C' is on top of the stack so that the content of SP is now 3.

→ To remove the top item the stack is popped by reading the memory word at address 3 and decrementing the content of SP.

→ Now item B is on top, since SP holds address 2.

→ To insert a new item, the stack is pushed by incrementing SP and writing a word in the next-higher location in the stack.

→ In a 64 word stack, the stack pointer (SP) contains 6-bits because $2^6 = 64$.

→ When 63 is incremented by 1, the result is '0'.

since 111111 + 1 ⇒ 1000000

⇒ 000000

→ When 000000 is decremented by 1, the result is 111111.

→ The one-bit register 'FULL' is set to 1 when the stack is full.

→ The one-bit register 'EMTY' is set to 1, when the stack is Empty.

→ DR is the data register that holds the binary data to be written in to (or) read out of the stack.

Push: → Initially, SP is cleared to '0', EMTY is set to '1', FULL is cleared to 0.

→ Then SP points the word at address 0.

→ If the stack is not full (i.e, FULL=0), an item is inserted with a push operation.

→ The microoperations involved are.

SP ← SP+1 , Increment stack pointer

M[SP] ← DR , write item on top of the stack.

If (SP=0) then (FULL ← 1) , Check if stack is full

EMTY ← 0 , mask the stack as Empty.

Pop: → A Next item is deleted from the stack if the stack is not Empty. (i.e, EMTY=0)

→ The POP operation consists of these microoperations.

DR ← M[SP] , read item from top of stack.

SP ← SP-1 , Decrement stack pointer.

If (SP=0) then (EMTY ← 1) , Check if stack is Empty.

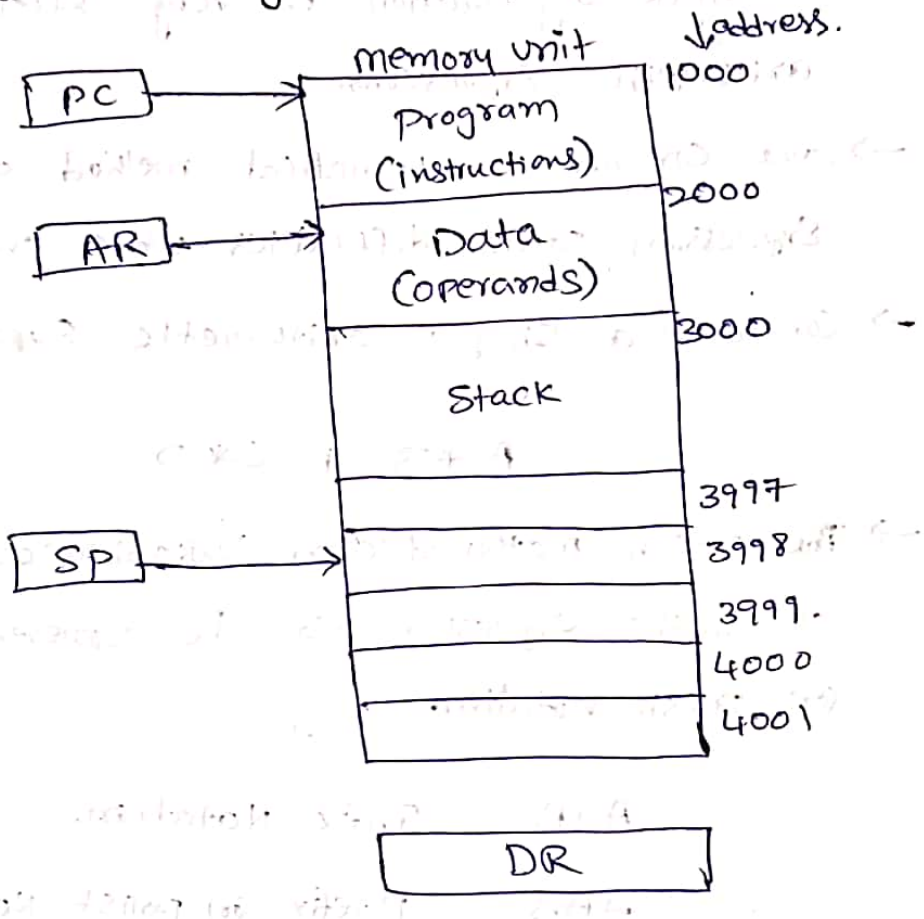
FULL ← 0 , mask the stack not full.

Memory stack :-

→ A stack can be implemented in a random-access memory attached to a CPU.

→ The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer.

The following figure shows a portion of computer memory partitioned into three segments; program, data and stack.



→ The program counter (PC) points at the address of the next instruction in the program.

→ The Address register (AR) points at an array of data.

→ The Stack Pointer (SP) points at the top of the stack.

→ The three registers are connected to a common address bus and either one can provide an address for memory.

→ PC is used during Fetch phase, AR is used during Execution phase, and SP is used to Push and Pop items.

Push:- $SP \leftarrow SP-1$
$M[SP] \leftarrow DR$

Pop:- $DR \leftarrow M[SP]$
$SP \leftarrow SP+1$

→ SP is updated by incrementing or decrementing depending on the organization of the stack.

Reverse Polish Notation (RPN)

→ A stack organization is very effective for evaluating arithmetic Expressions.

→ The common mathematical method of writing arithmetic Expression causes difficulties while evaluated by a computer.

→ Consider a simple arithmetic Expression.

$$A * B + C * D$$

→ The polish mathematecian Lukasiewicz showed that arithmetic Expression can be represented in prefix Notation (or) polish notation.

A+B	Infix Notation.
+AB	Prefix (or) polish Notation
AB+	Postfix (or) Reverse polish Notation

→ The reverse polish Notation is in a form suitable for stack manipulation.

$$A * B + C * D \xrightarrow[\text{Scan}]{} AB * CD * +$$

Steps for Evaluation:

→ Scan the Expression from left to Right.

→ when an operator is found, perform the operation with the two operands found on the left side.

→ Remove the operands and operator and replace them by the result.

→ Continue the scan and do the same procedure till the End.

Conversion to RPN

→ The conversion from infix notation to Reverse Polish notation must take into consideration the operational hierarchy adopted for infix notation.

→ This hierarchy dictates that we first perform all arithmetic inside parentheses, then inside outer parentheses, and do multiplication and division operations before addition and subtraction operations.

Consider the Expression,

$$(A+B) * [C * (D+E) + F]$$

The converted Expression is.

$$AB+DE+C*F+*$$

Proceeding from left to right, we first add A and B, then add D and E.

$$(A+B)(D+E)C*F+*$$

where (A+B) and (D+E) are each a single number obtained from the sum. The two operands for the next '*' are C and (D+E).

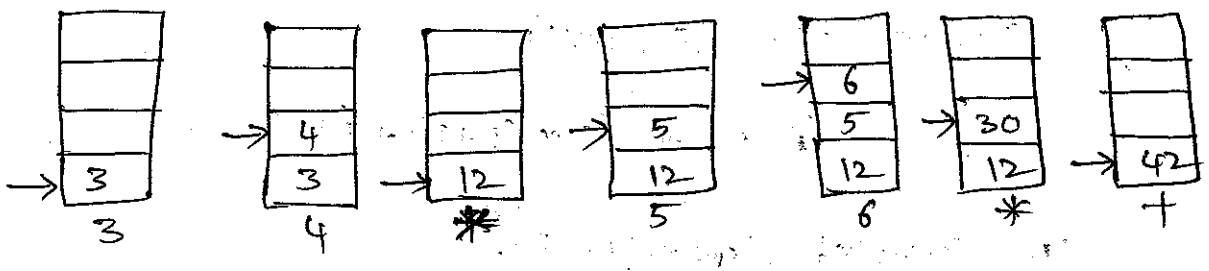
These two numbers are multiplied and the product added to F. The final * causes the multiplication of the two terms.

Evaluation of Arithmetic Expressions:

Reverse polish Notation, Combined with a stack arrangement of registers, is the most efficient way known for evaluating arithmetic Expressions.

Consider the arithmetic Expression (3*4) + (5*6).

Stack operations to Evaluate 3*4 + 5*6. are,



⊛ Instruction Formats:-

→ An Instruction can be represented in a specific format. The Bits present in the instruction are classified in to several groups to form fields.

There are three basic fields that are present in an instruction

1. Opcode → operation code field
2. Address field → Memory address & processor register specified
3. Mode field → Determines the mode of addressing.

→ The length of an instruction differs in different computers based on the number of addresses used within an instruction.

→ The three different types of CPU organizations are

1. Single accumulator organization.
2. General register organization.
3. Stack organization.

1. Single accumulator organization:

only one address field is used within the instruction format.

Ex: ADD B

This instruction will add the value present at address B with the value present in the accumulator and stores the result back in to the accumulator.

2. General register organization:-

In this, There are two to three register address fields within the instruction format.

Ex: ADD R1, R2

This instruction will add the data present in R1 with the data present in R2 and stores result in R1 Register.

3. Stack organization: It uses PUSH or POP instructions which use only one address field.

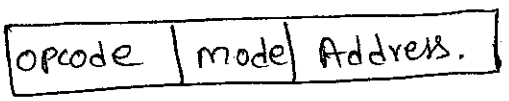
Ex: PUSH C

This instruction will push the word present at address C to the top of the stack.

(*) Addressing modes

In order to enlarge the capability and Computing power of an instruction, it is desirable to have different types of addressing capabilities in an instruction format.

Instruction format with 'mode' field is given by,



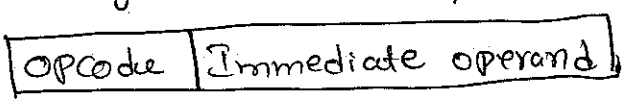
Types of addressing modes:

① Implied addressing mode:-

when the instruction's definition contains the operands that are implicitly provided on which an operation specified by the opcode is to be performed, then the addressing mode is called as implied addressing mode.

② Immediate addressing mode:-

In this mode, operand is part of the instruction instead of the contents of a register or memory location.

Instruction format ⟹ 

③ Register mode:-

As we know, the address of an instruction may specify either a memory word or a processor register.

So, In this mode, the operands are in registers that reside with in the CPU. a k-bit field can specify any one of 2^k registers.

④ Register indirect mode:-

In this mode, the instruction specifies a register in the CPU, whose contents give the address of the operand in memory. i.e., the selected register contains the address of the operand rather than the operand itself.

⑤ Autoincrement (or) Autodecrement mode:-

This is similar to the register indirect mode, except that the register is incremented or decremented after (or) before its value is used to access memory.

⑥ Direct address mode:-

The operand in this mode is a memory address where the data is stored.

⑦ Indirect addressing mode:-

In this mode, the address field of the instruction gives the address where the effective address is stored in memory.

⑧ Relative address mode:-

In this mode, the contents of the program counter is added to the address part of the instruction in order to obtain the effective address.

Ex: If PC contains 825, and address part contains number 24. during fetch phase, the instruction at location 825 is read. the PC, then incremented by 1 to 826.

Then the effective address computation for the relative address mode is $826 + 24 = 850$. i.e, 24 instructions forward from the address of the next instruction.

⑨ Indexed addressing mode:-

In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.

This indexed addressing mode is used to access consecutive operands.

⑩ Base register addressing mode:-

In this mode, the content of the base register is added to the address part of the instruction to obtain the effective address.

Data transfer and manipulation:

* Computers provide an extensive set of instructions to the user to carryout various computational tasks.

* Now, we will discuss the basic set of operations available in a typical computer.

→ most computer instructions can be classified into three categories.

1) Data transfer instructions
2) Data manipulation instructions and
3) Program control instructions.

→ Data transfer instructions cause transfer of data from one location to another, without changing the binary information content.

→ Data manipulation instructions are those that perform arithmetic, logic and shift operations.

→ Program control instructions provide decision making capabilities. and change the path taken by the program when executed in the computer.

(1) Data transfer instructions.

Name	mnemonic/ Symbol used	Description.
Load	LD	→ Loads or transfers data from a given memory location to the processor for computations.
Store	ST	→ Transfer from a processor register to memory.
move	MOV	→ Transfer of data from source register to destination register
Exchange	XCH	→ swaps information between two registers.
Input	IN	→ Transfers data among processor registers and
output	OUT	input (or) output terminals.
Push	PUSH	→ Transfers data between processor registers and memory
POP	POP	stack.

② Data manipulation instructions:

→ Data manipulation instructions Perform operations on data and Provide the Computational capabilities for the computer.

→ The data manipulation instructions are usually divided in to three basic types.

i) Arithmetic instructions.
ii) Logical and bit manipulation instructions
iii) Shift instructions.

i) Arithmetic instructions:

→ The four basic arithmetic operations are addition, subtraction, multiplication and division.

→ A list of Arithmetic instructions are given below.

Name	mnemonic
INCrement	INC → increments the operand by 1
Decrement	DEC → Decrements the operand by 1
Add	ADD → Adds the two operand values
Subtract	SUB → Subtracts the two operand values.
Multiply	MUL → multiplies " " " "
Divide	DIV → Divides " " " "
Add with Carry	ADDC → Performs addition along with Carry bit
Subtract with borrow	SUBB → " Subtraction " " Borrow bit
Negate (2's complement)	NEG → Changes the sign of the Operand.

ii) Logical and bit manipulation instructions:

→ Logical instructions perform binary operations on strings of bits stored in registers.

→ Some logical and bit manipulation instructions are listed below.

Name	Mnemonic
Clear	CLR → clears the register i.e, makes it empty.
Complement	COM → Complements the bits in a register.
AND	AND → Performs logical AND operation.
OR	OR → " " OR operation.
Exclusive-OR	XOR → " " XOR "
Clear carry	CLRC → Clears the register maintaining the carry bit
Set carry	SETC → Enables the carry register.
Complement carry	COMC → Complements the carry bit.
Enable interrupt	EI → Enables the interrupt
Disable interrupt	DI → Disables the interrupt.

iii) Shift instructions:

→ Shift instructions shifts the contents of the registers to either left or right.

→ Various shift instructions are given as follows.

Name	mnemonic
Logical Shift left	SHL
Logical Shift Right	SHR
Arithmetic " Right	SHRA
" " Left	SHLA
Rotate Right	ROR
Rotate left	ROL
Rotate Right throughcarry	RORC
" left " carry	ROLC

3) Program Control instructions

→ These program control instructions makes the control to switch to other different locations.

→ Various program control instructions are given by.

Name	mnemonic
Branch	BR → control goes to the instruction specified in Branch Condition
Jump	JMP → program switches to another location.
SKIP	SKP → Skips the next instruction.
Call	CALL → Calls the specific procedure to be executed.
return	RET → It returns the program control from activated Call instruction
Compare (by subtraction)	CMP → Comparision is made b/w two operands.
Test (By ANDing)	TST → Tests the given value based on the given condition.

Reduced instruction set computers:

RISC: processors with simple instructions are referred to "Reduced Instruction set Computers" (RISC).

CISC: processors with more complex instructions are referred to "complex instruction set computers".

Types of interrupts:

Internal Interrupts: These are the interrupts generated by the program itself.
Ex: register overflow, divided by zero.

External interrupts: External Interrupts are occured from the hardware units.
Ex: I/O devices, power sensing circuits.

Software Interrupts: Software Interrupts is initiated by executing an instruction of type INT. S/w interrupts are controlled by programmers.

Control Memory:

* The major functional parts in a digital computer are CPU, Memory and Input-output.

* The main digital hardware functional units of CPU are Control unit, arithmetic and logic unit and registers.

* The function of the Control unit in a digital computer is to initiate sequences of microoperations.

* Two methods of implementing Control unit are,
  i) Hardwired Control and
  ii) microprogrammed Control.

i) Hardwired control:

* The design of Hardwired Control involves the use of fixed instructions, fixed logic blocks of arrays, encoders, decoders etc.

* The key characteristics of hardwired control logic are high speed operation, expensive, relatively complex and no flexibility of adding new instructions.

* When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired.

Example: CPU's with hardwired control are Intel 8085, Motorola 6802, Zilog 80 and any RISC CPU's.

ii) Microprogrammed control:

* microprogramming is a second alternative for designing the control unit of a digital computer.

* The principle of microprogramming is a systematic method for controlling the microoperation sequences, in a digital computer.

Example: cpu's with microprogrammed control unit are, Intel 8080, Motorola 68000, any clse cpu's.

* The control function that specifies a microoperation is a binary variable.

* When it is in '1' binary state, the corresponding microoperation is executed.

* The active state of a control variable may be either the '1' state or the 0 state, depending on the application.

* In a bus-organized systems the control signals that specify microoperations are groups of bits that select the paths in multiplexers, decoders and arithmetic logic units.

* The control variables at any given time can be represented by a string of 1's and 0's called a control word.

* These control words can be programmed to perform various operations on the components of the system.

* A Control unit whose binary control variables are stored in memory is called a microprogrammed Control unit.
* Each word in control memory contains within a microinstruction.
* The microinstruction specifies one or more microoperations for the System.
* A sequence of microinstructions is called a microprogram.
* Since alterations of the microprogram are not needed once the control unit is in operation, the control memory can be a read-only memory (ROM).
* ROM words are made permanent during the hardware production of the unit.
* The contents of the word in ROM at a given address specifies a microinstruction.
* Dynamic microprogramming permits a microprogram to be loaded initially from a magnetic disk.
* Control units that use dynamic programming employ a writable control memory.
* A memory that is part of a control unit is referred to as a control memory.

* A Computer with Microprogrammed Control unit will have two separate memories.

  ⓐ main memory.

  ⓑ Control memory.

* The main memory is available to the user for storing the programs.

* The contents of main memory can be altered when the data is manipulated. and every time the program is changed.

* But, the control memory holds a fixed microprogram that cannot be altered by the user.

* Each machine instruction initiates a series of micro-instructions in control memory.

* These micro instructions generate the microoperations,

  to fetch the instruction from main memory,

  to evaluate the effective address.

  to Execute the operation specified by the instruction and

  to return control to the fetch phase, in order to repeat the cycle for the next instruction.

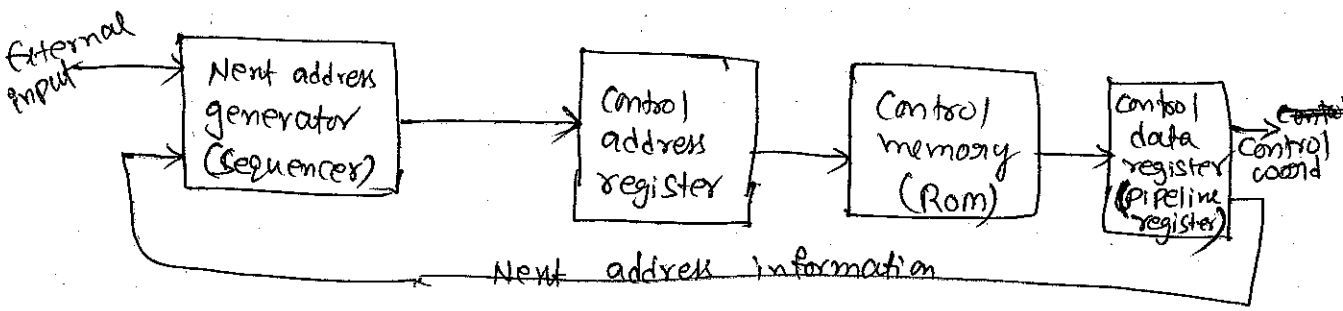


fig: Micro programmed Control organization.

* The Control address register specifies the address of the microinstruction.
* The Control data register holds the microinstruction read from the memory. This microinstruction contains a control word which specifies one or more microoperations.
* After these operations are executed, the control must determines the next address.
* Next address, may be the one next in sequence (or)
  may be located somewhere else in control memory (or)
  may be a function of external i/p condition.
* The next address generator is sometimes called a microprogram sequencer.
* The control data register allows the execution of the microoperations simultaneously with the generation of the next microoperation.
* Therefore this data register is sometimes called a pipeline register.

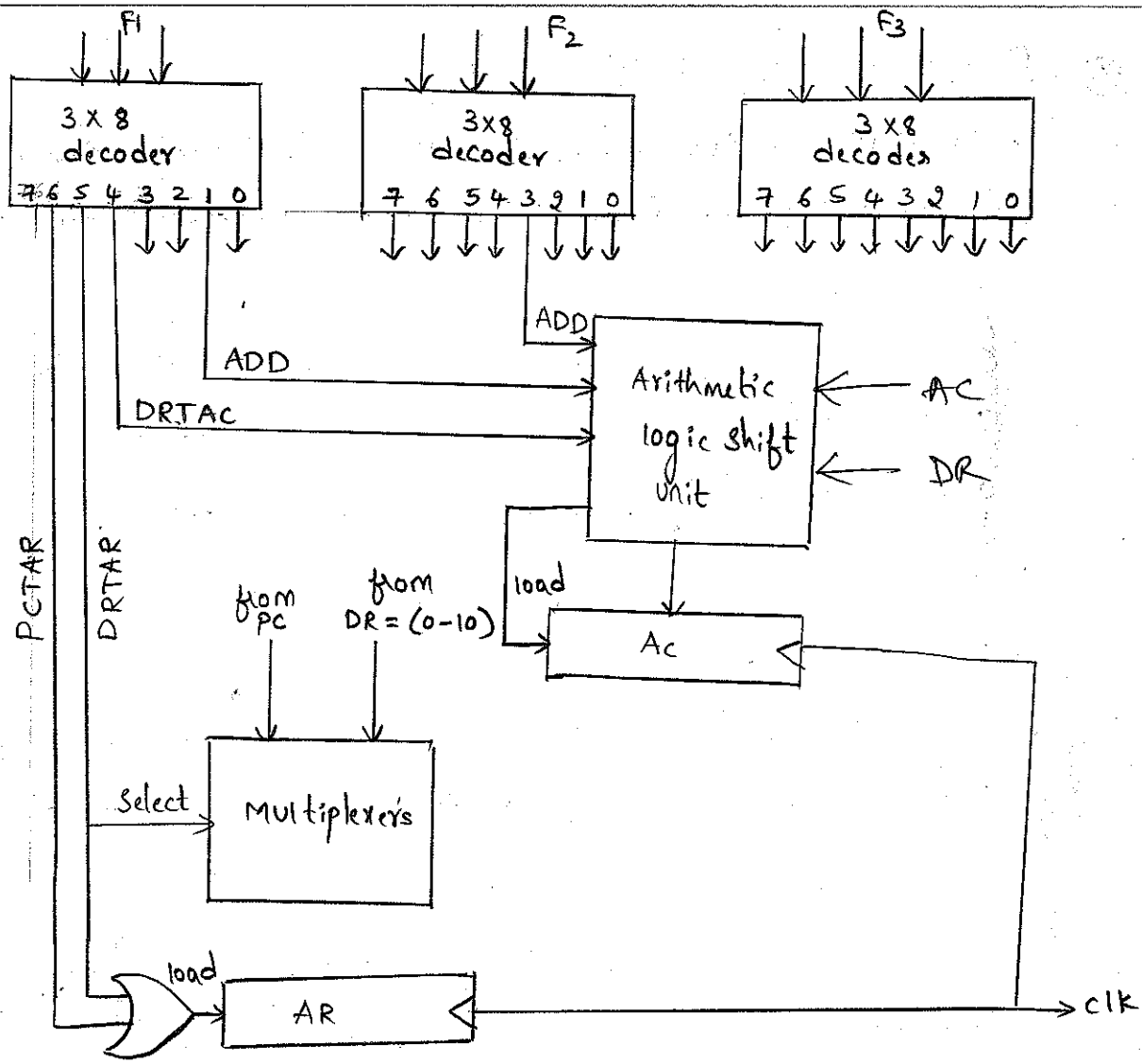
Design of control unit:

* The bits of the microinstruction are usually divided into fields, with each field defining a distinct, separate function.
* The various fields encountered in instruction formats provide control bits to initiate microoperations in the system, special bits to specify the way that the next address is to be evaluated, and an address field for branching.
* Each field requires a decoder to produce the corresponding control signals.
* This method reduces the size of the microinstruction bits but requires additional hardware, external to the control memory.
* The nine bits of the microoperation field are divided into three subfields of three bits each.
* The control memory output of each subfield must be decoded to provide the distinct microoperations.
* The outputs of the decoders are connected to the appropriate inputs in the processor unit.

Decoding of f fields:

* The following figure shows the three decoders and some of the connections that must be made from their outputs.

* Each of the three fields of the microinstruction presently available in the output of control memory are decoded with a 3x8 decoder to provide eight outputs as shown in fig.



* Each of these outputs must be connected to the proper circuit to initiate the corresponding microoperation.

* For Example, when F1 = 101 (binary 5), the next clock pulse transition transfers the content of DR(0-10) to AR, symbolized by DRTAR.

* As shown in fig, outputs 5 and 6 of decoder F1 are connected to the load input of AR so that when either one of these outputs is active, information from the multiplexers is transferred to AR.

* The multiplexers select the information from DR when output 5 is active and from PC when output 5 is inactive.

Arithmetic logic shift unit:

* Instead of using gates to generate the control signals like AND, ADD etc, these inputs to arithmetic logic shift unit will come from the outputs of the decoders associated with the symbols AND, ADD and DRTAC as shown in the figure.

* The other outputs of the decoders that are associated with an AC operation must also be connected to the arithmetic logic shift unit.

(*) Microprogram Sequencer:

* The basic components of a microprogrammed control unit are the control memory and the circuits that select the next address.

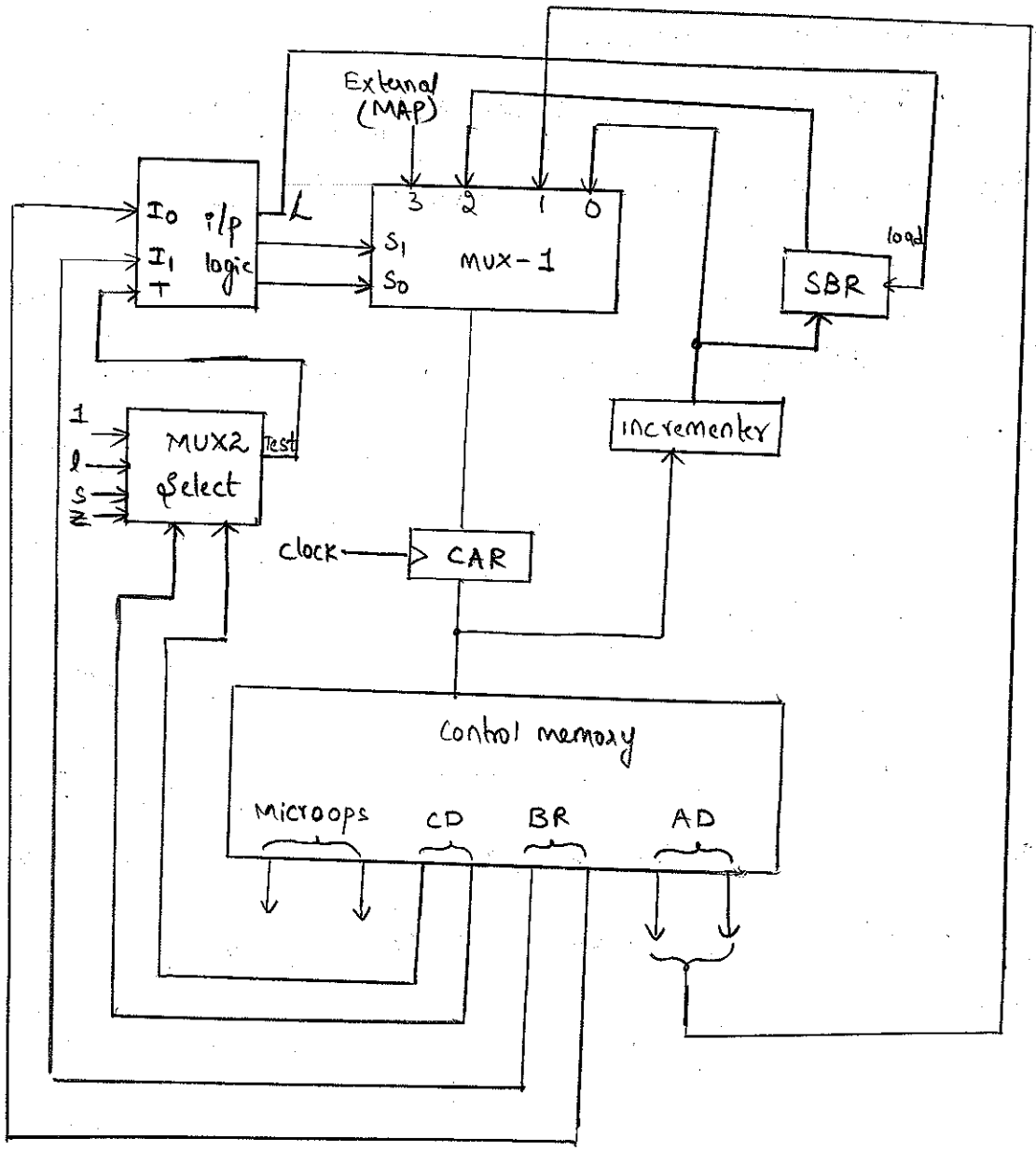
* The address selection part is called a microprogram sequencer.

* The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed.

* The next-address logic of the sequencer determines the specific address source to be loaded in to the control address register.

* The choice of the address source is guided by the next address information bits that the sequencer receives from the present microinstruction.

* The Block diagram of the microprogram sequencer is as shown below.



* The Control memory is included in the diagram to show the interaction between the Sequencer and the memory attached to it.
* There are two multiplexers in the circuit.
* The first multiplexer selects an address from one of four sources and routes it into a control address register CAR.
* The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit.

* The output from CAR provides the address for the control memory.
* The contents of CAR is incremented and applied to one of the multiplexer inputs and to the Sub-routine register SBR.
* The other three inputs to multiplexer number '1' come from the address field of the present microinstruction, from the output of SBR, and from an external source that maps the instruction.
* The CD (condition) Field of the microinstruction selects one of the status bits in the second multiplexer.
* If the bit selected is equal to 1, then T (test) variable is equal to 1; otherwise, it is equal to 0.
* The T value together with the two bits from the BR (Branch) field go to an input logic circuit.
* The input logic in a particular Sequencer will determine the type of operations that are available in the unit.
* Typical Sequencer operations are: increment, branch (or) Jump, Call and return from subroutine, load an external address, Push or pop the stack, and other address sequencing operations.
* With three inputs, the Sequencer can provide upto eight address sequencing operation.

Design of input logic:

* The input logic circuit in Fig: has three inputs I_0, I_1 and T, and three outputs, S_0, S_1 and L.

* Variables S_0 and S_1 select one of the source addresses for CAR. ~~Variable~~

* Variable L enables the load input in SBR.

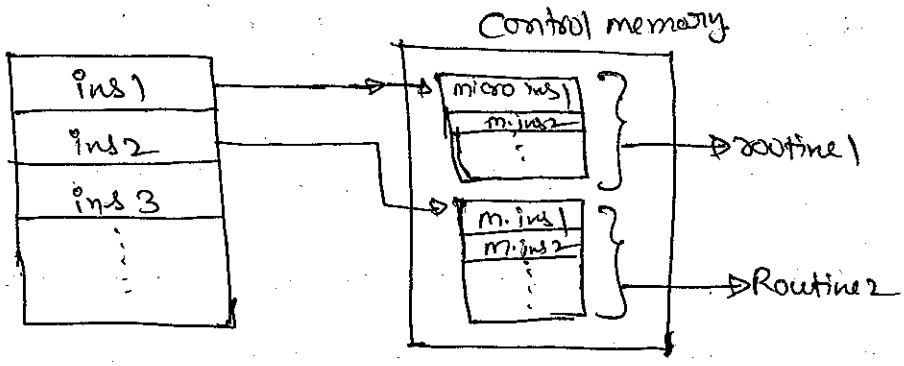
* The binary values of the two selection variables determine the path in the multiplexer.

* For example, with $S_1, S_0 = 10$, multiplexer input number 2 is selected and establishes a transfer path from SBR to CAR.

(*) Address Sequencing:-

→ Microinstructions are usually stored in groups where each group specifies how to carry out an instruction.

→ Each routine must be able to branch to the next routine in the sequence.



→ An initial address is loaded into the car when power is turned on. This is usually the address of the first microinstruction in the instruction Fetch routine.

→ Next the control unit must determine the effective address of the instruction.

→ Address Sequencing capabilities of control memory include,

i) Incrementing of the control address register.

ii) Unconditional branch or conditional branch depending on Status bit Conditions.

iii) A mapping process from the bits of the instruction to an address for control memory.

iv) A facility for Subroutine call and return.

Conditional Branching:

→ Control the conditional branch decisions made by the

branch logic together with the field in the microinstruction that specifies a branch address.

Mapping: The next step is to generate the microoperations that executed the instruction.

This involves taking the instructions opcode and transforming it, in to an address for the instructions microprogram in control memory.

This process is called mapping.

→ The mapping scheme shown in the below figure allows four microinstructions as well as overflow space from 1000000 to 1111111

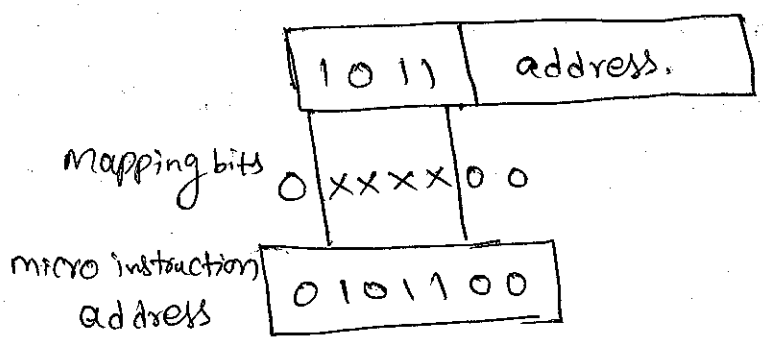


fig: mapping from instruction code to microoperation address.

Subroutines:

→ Subroutines calls are a special type of branch where we return to one instruction below the calling instruction

→ provision must be made to save the return address since it cannot be written in to ROM.