KTU NOTES

MODULE 1
COMPUTER ARITHMETIC AND PROCESSOR BASICS

Syllabus:

Algorithms for binary multiplication and division. Fixed and floating-point number representation. Functional units of a computer, Von Neumann and Harvard computer architectures, CISC and RISC architectures. Processor Architecture – General internal architecture, Address bus, Data bus, control bus. Register set – status register, accumulator, program counter, stack pointer, general purpose registers. Processor operation – instruction cycle, instruction fetch, instruction decode, instruction execute, timing response, instruction sequencing and execution (basic concepts, data-path).

1.1 FUNCTIONAL UNITS

A computer consists of five functionally independent main parts: input, memory, arithmetic and logic, output, and control units, as shown in Figure 1.1.

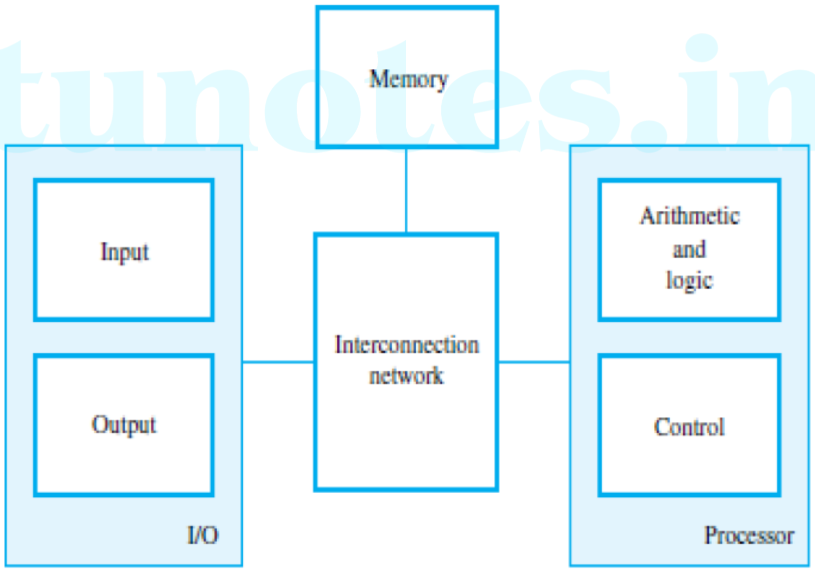


Fig 1.1 Basic Functional units of a computer

The input unit accepts coded information from human operators using devices such as keyboards, or from other computers over digital communication lines. The information received is stored in the computer‟s memory, either for later use or to be processed immediately by the arithmetic and logic unit. The processing steps are specified by a program that is also stored in the memory. Finally, the results are sent back to the outside world through the output unit. All of these actions are coordinated by the control unit. An interconnection network provides the means for the functional units to exchange information and coordinate their actions. The arithmetic and logic circuits, in conjunction with the main control circuits, is the processor. Input and output equipment is often collectively referred to as the input-output (I/O) unit. A program is a list of instructions which performs a task. Programs are stored in the memory. The processor fetches the program instructions from the memory, one after another, and performs the desired operations. The computer is controlled by the stored program, except for possible external interruption by an operator or by I/O devices connected to it. Data are numbers and characters that are used as operands by the instructions. Data are also stored in the memory. The instructions and data handled by a computer must be encoded in a suitable format. Each instruction, number, or character is encoded as a string of binary digits called bits, each having one of two possible values, 0 or 1, represented by the two stable states.

Input Unit:

Computers accept coded information through input units. The most common input device is the keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor. Many other kinds of input devices for human-computer interaction are available, including the touchpad, mouse, joystick, and trackball. These are often used as graphic input devices in conjunction with displays. Microphones can be used to capture audio input which is then sampled and converted into digital codes for storage and processing. Similarly, cameras can be used to capture video input. Digital communication facilities, such as the Internet, can also provide input to a computer from other computers and database servers. **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 Memory** Primary memory, also called main memory, is a fast memory that operates at electronic speeds. Programs must be stored in this memory while they are being executed. The memory consists of a large number of semiconductor storage cells, each capable of storing one bit of information. These cells are rarely read or written individually. Instead, they are handled in groups of fixed size called words. The memory is organized so that one word can be stored or retrieved in one basic operation. The number of bits in each word is referred to as the word length of the computer, typically 16, 32, or 64 bits. To provide easy access to any word in the memory, a distinct address is associated with each word location. Addresses are consecutive numbers, starting from 0, that identify successive locations. Instructions and data can be written into or read from the memory under the control of the processor. A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called a random-access memory (RAM). The time required to access one word is called the memory access time. This time is independent of the location of the word being accessed. It typically ranges from a few nanoseconds (ns) to about 100 ns for current RAM units **Cache Memory** As an adjunct to the main memory, a smaller, faster RAM unit, called a cache, is used to hold sections of a program that are currently being executed, along with any associated data. The cache is tightly coupled with the processor and is usually contained on the same integrated-circuit chip. The purpose of the cache is to facilitate high instruction execution rates. At the start of program execution, the cache is empty. As execution proceeds, instructions are fetched into the processor chip, and a copy of each is placed in the cache. When the execution of an instruction requires data, located in the main memory, the data are fetched and copies are also placed in the cache. If these instructions are available in the cache, they can be fetched quickly during the period of repeated use. **Secondary Storage** Although primary memory is essential, it tends to be expensive and does not retain information when power is turned off. Thus additional, less expensive, permanent secondarystorage is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently. Access times for secondary storage are longer than for primary memory. The devices available are including magnetic disks, optical disks (DVD and CD), and flash memory devices.

Arithmetic and Logic Unit Most computer operations are executed in the arithmetic and logic unit (ALU) of the processor. Any arithmetic or logic operation, such as addition, subtraction, multiplication division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU. When operands are brought into the processor, they are stored in high-speed storage elements called registers. Each register can store one word of data. Access times to registers are even shorter than access times to the cache unit on the processor chip. **Output Unit** Output unit function is to send processed results to the outside world. A familiar example of such a device is a printer. Most printers employ either photocopying techniques, as in laser printers, or ink jet streams. Such printers may generate output at speeds of 20 or more pages per minute. However, printers are mechanical devices, and as such are quite slow compared to the electronic speed of a processor. Some units, such as graphic displays, provide both an output function, showing text and graphics, and an input function, through touchscreen capability. The dual role of such units is the reason for using the single name input/output (I/O) unit in many cases. **Control Unit** The memory, arithmetic and logic, and I/O units store and process information and perform input and output operations. The operation of these units must be coordinated in some way. This is the responsibility of the control unit. The control unit is effectively the nerve center that sends control signals to other units and senses their states. I/O transfers, consisting of input and output operations, are controlled by program instructions that identify the devices involved and the information to be transferred. Control circuits are responsible for generating the timing signals that govern the transfers. They determine when a given action is to take place. Data transfers between the processor and the memory are also managed by the control unit through timing signals. A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units. The operation of a computer can be summarized as follows: • The computer accepts information in the form of programs and data through an input unit and stores it in the memory. • Information stored in the memory is fetched under program control into an arithmetic and logic unit, where it is processed. • Processed information leaves the computer through an output unit. • All activities in the computer are directed by the control unit. **Von Neumann architecture** In the 1940s, a mathematician called John Von Neumann described the basic arrangement (or architecture) of a computer. Most computers today follow the concept that he described although there are other types of architecture. A Von Neumann-based computer is a computer that: Uses a single processor. Uses one memory for both instructions and data. A von Neumann computer cannot distinguish between data and instructions in a memory location! It „knows" only because of the location of a particular bit pattern in RAM. Executes programs by doing one instruction after the next in a serial manner using a fetch-decode-execute cycle.

1.2 COMPUTER ARCHITECTURE

Computer Architecture refers to the internal design of a computer with its CPU, which includes:

- ❖ Arithmetic and logic unit,
- ❖ Control unit,
- ❖ Registers,
- ❖ Memory for data and instructions,
- ❖ Input/output interface and
- ❖ External storage functions.

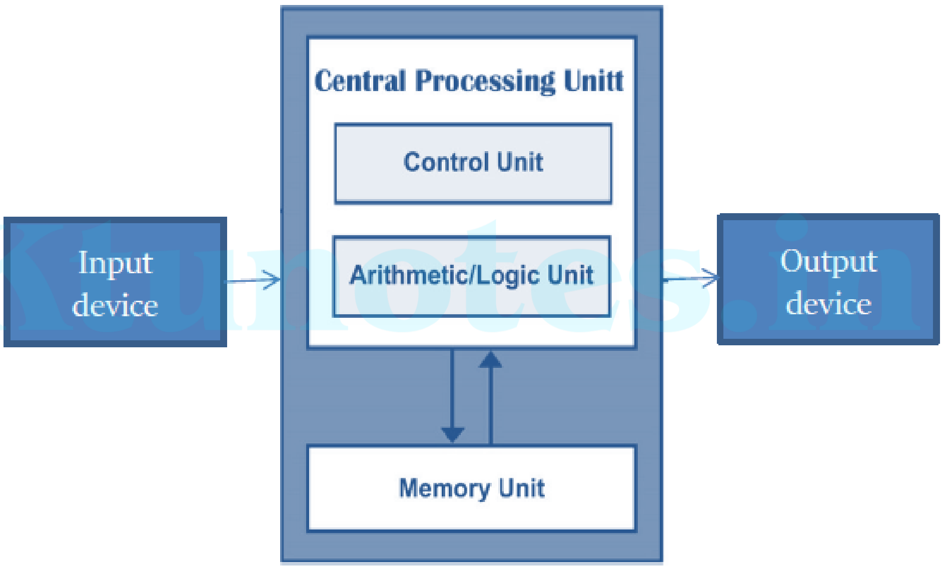


Fig 1.2 : General Architecture

VON-NEUMANN ARCHITECTURE:

The same memory and bus are used to store both Data and Instructions.

The main drawback:

CPU is unable to access program memory and data memory simultaneously. This case is called the "**bottleneck**" that affects system performance.

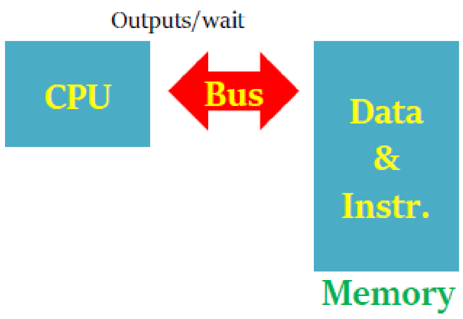


Fig 1.3: Von-Neumann architecture

The bottleneck

- If a Von-Neumann machine wants to perform an instruction (already fetched from the memory) on some data in memory, it has to move the data across the bus into the CPU.

- When the computation is done, it needs to move outputs of the computation to memory across the same bus; this operation will be completed if the bus is not used by another operation to fetch another instruction or data from the shared memory; otherwise the outputs of the computation has to wait.

HARVARD ARCHITECTURE:

The Harvard architecture stores machine instructions and data in separate memory units using different buses.

The main advantage:

- Computers designed with the Harvard architecture are able to run a program and access data independently, and therefore simultaneously.

Harvard architecture is more complicated but separate pipelines remove the **bottleneck** that Von-Neumann creates.

MODIFIED HARVARD ARCHITECTURE

The majority of modern computers have no physical separation between the memory spaces used by both data and instructions, therefore could be described technically as Von-Neumann. But as they have two separate address spaces, different buses and special instructions that keep data from being mistaken for code, this architecture is called "Modified Harvard Architecture".

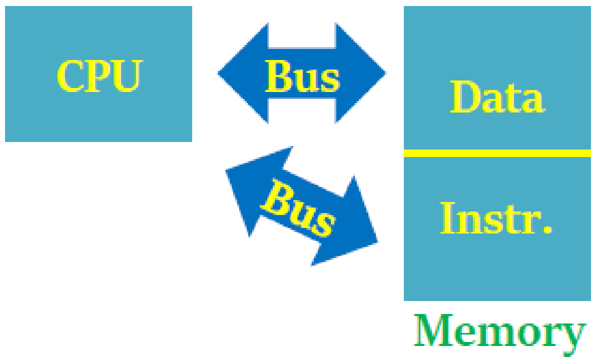
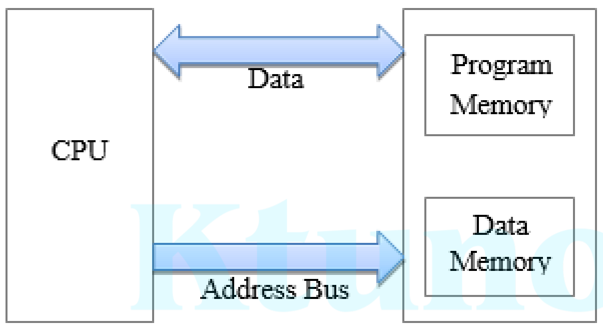
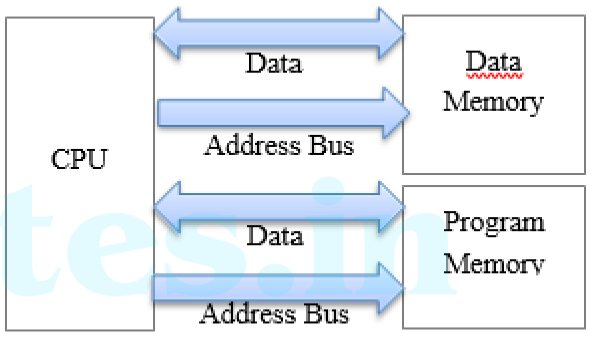


Fig 1.4: Harvard Architecture

Ex. some initial data values or constants can be accessed by the running program directly from instruction memory without taking up space in data memory.

HARVARD & VON- NEUMANN CPU ARCHITECTURE

Von-Neumann (Princeton architecture)	**Harvard architecture**
	 CPU Data Address Bus Data Memory Data Address Bus Program Memory
Von-Neumann (Princeton architecture)	**Harvard architecture**
It uses single memory space for both instructions and data.	It has separate program memory and data memory
It is not possible to fetch instruction code and data	Instruction code and data can be fetched simultaneously
Execution of instruction takes more machine cycle	Execution of instruction takes less machine cycle
Uses CISC architecture	Uses RISC architecture
Instruction pre-fetching is a main feature	Instruction parallelism is a main feature
Also known as control flow or control driven computers	Also known as data flow or data driven computers
Simplifies the chip design because of single memory space	Chip design is complex due to separate memory space
Eg. 8085, 8086, MC6800	Eg. General purpose microcontrollers, special DSP chips etc.

1.3 RISC AND CISC ARCHITECTURES

General

The dominant architecture in the PC market belongs to the Complex Instruction Set Computer (CISC) design. The obvious reason for this classification is the “complex” nature of its Instruction Set Architecture (ISA). The motivation for designing such complex instruction sets is to provide an instruction set that closely supports the operations and data structures used by Higher-Level Languages (HLLs). However, the side effects of this design effort are far too serious to ignore.

Addressing Modes in CISC

The decision of CISC processor designers to provide a variety of addressing modes leads to variable-length instructions. For example, instruction length increases if an operand is in memory as opposed to in a register.

- This is because we have to specify the memory address as part of instruction encoding, which takes many more bits.
- This complicates instruction decoding and scheduling. The side effect of providing a wide range of instruction types is that the number of clocks required to execute instructions varies widely.
- This again leads to problems in instruction scheduling and pipelining.

Evolution of RISC

For these and other reasons, in the early 1980s designers started looking at simple ISAs. Because these ISAs tend to produce instruction sets with far fewer instructions, they coined the term Reduced Instruction Set Computer (RISC). Even though the main goal was not to reduce the number of instructions, but the complexity, the term has stuck.

There is no precise definition of what constitutes a RISC design. However, we can identify certain characteristics that are present in most RISC systems.

- We identify these RISC design principles after looking at why the designers took the route of CISC in the first place.
- Because CISC and RISC have their advantages and disadvantages, modern processors take features from both classes. For example, the PowerPC, which follows the RISC philosophy, has quite a few complex instructions.

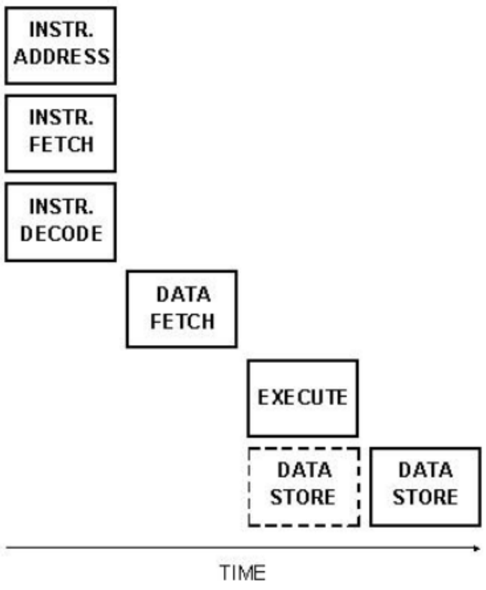


Fig 1.5: **Typical RISC Architecture based Machine - Instruction phase overlapping**

Definition of RISC

RISC, or Reduced Instruction Set Computer is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures.

Evolution/History.

The first RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s and early 80s. The IBM 801, Stanford MIPS, and Berkeley RISC 1 and 2 were all designed with a similar philosophy which has become known as RISC. Certain design features have been characteristic of most RISC processors

- **One Cycle Execution Time**. RISC processors have a CPI (clock per instruction) of one cycle. This is due to the optimization of each instruction on the CPU and a technique called ;
- **Pipelining**. A technique that allows for simultaneous execution of parts, or stages, of instructions to more efficiently process instructions;
- **Large Number of Registers**. The RISC design philosophy generally incorporates a larger number of registers to prevent in large amounts of interactions with memory

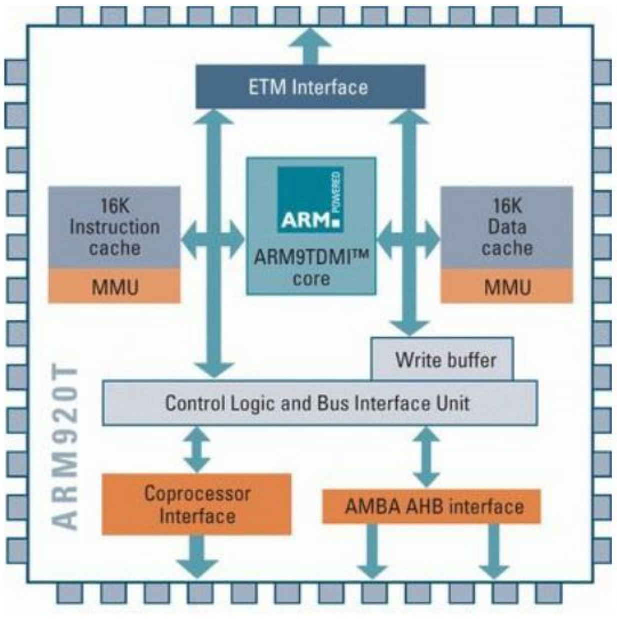


Fig 1.6 **Advanced RISC Machine (ARM)**

Non RISC Design or Pre RISC Design

In the early days of the computer industry, programming was done in assembly language or machine code, which encouraged powerful and easy to use instructions. CPU designers therefore tried to make instructions that would do as much work as possible. With the advent of higher level languages, computer architects also started to create dedicated instructions to directly implement certain central mechanisms of such languages.

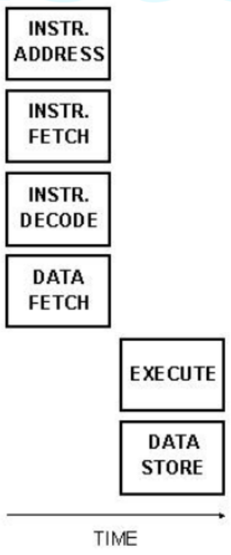


Fig 1.7: **Typical CISC Architecture** – Stack Design

Another general goal was to provide every possible addressing mode for every instruction, known as orthogonality, to ease compiler implementation. Arithmetic operations could therefore often have results as well as operands directly in memory (in addition to register or immediate).

The attitude at the time was that hardware design was more mature than compiler design so this was in itself also a reason to implement parts of the functionality in hardware and/or microcode rather than in a memory constrained compiler (or its generated code) alone. This design philosophy became retroactively termed Complex Instruction Set Computer (CISC) after the RISC philosophy came onto the scene.
An important force encouraging complexity was very limited main memories (on the order of kilobytes). It was therefore advantageous for the density of information held in computer programs to be high, leading to features such as highly encoded, variable length instructions, doing data loading as well as. These issues were of higher priority than the ease of decoding such instructions.

An equally important reason was that main memories were quite slow (a common type was ferrite core memory); by using dense information packing, one could reduce the frequency with which the CPU had to access this slow resource. Modern computers face similar limiting factors: main memories are slow compared to the CPU and the fast cache memories employed to overcome this are instead limited in size. This may partly explain why highly encoded instruction sets have proven to be as useful as RISC designs in modern computers.

TYPICAL CHARACTERISTICS OF RISC ARCHITECTURE

Designers make choices based on the available technology. As the technology, both hardware and software, evolves, design choices also evolve. Furthermore, as we get more experience in designing processors, we can design better systems. The RISC proposal was a response to the changing technology and the accumulation of knowledge from the CISC designs. CISC processors were designed to simplify compilers and to improve performance under constraints such as small and slow memories. The important observations that motivated designers to consider alternatives to CISC designs were

- **Simple Instructions**. The designers of CISC architectures anticipated extensive use of complex instructions because they close the semantic gap. In reality, it turns out that compilers mostly ignore these instructions. Several empirical studies have shown that this is the case. One reason for this is that different high-level languages use different semantics. For example, the semantics of the C for loop is not exactly the same as that in other languages. Thus, compilers tend to synthesize the code using simpler instructions.
- **Few Data Types**. CISC ISA tends to support a variety of data structures, from simple data types such as integers and characters to complex data structures such as records and structures. Empirical data suggest that complex data structures are used relatively infrequently. Thus, it is beneficial to design a system that supports a few simple data types efficiently and from which the missing complex data types can be synthesized.
- **Simple Addressing Modes**. CISC designs provide a large number of addressing modes. The main motivations are

  (1) To support complex data structures and

  (2) To provide flexibility to access operands.

(a) Problems Caused. Although this allows flexibility, it also introduces problems. First, it causes variable instruction execution times, depending on the location of the operands.

(b) Second, it leads to variable-length instructions. For example, the IA-32 instruction length can range from 1 to 12 bytes. Variable instruction lengths lead to inefficient instruction decoding and scheduling.

- **Identical General Purpose Registers**. Allowing any register to be used in any context, simplifying compiler design (although normally there are separate floating point registers).
- **Harvard Architecture Based**. RISC designs are also more likely to feature a Harvard memory model, where the instruction stream and the data stream are conceptually separated; this means that modifying the memory where code is held might not have any effect on the instructions executed by the processor (because the CPU has a separate instruction and data cache), at least until a special synchronization instruction is issued. On the upside, this allows both caches to be accessed simultaneously, which can often improve performance.

RISC VS CISC – AN EXAMPLE

The simplest way to examine the advantages and disadvantages of RISC architecture is by contrasting it with its predecessor, CISC (Complex Instruction Set Computers) architecture.

Multiplying Two Numbers in Memory.

The main memory is divided into locations numbered from (row) 1: (column) 1 to (row) 6: (column) 4. The execution unit is responsible for carrying out all computations. However, the execution unit can only operate on data that has been loaded into one of the six registers (A, B, C, D, E, or F). Let's say we want to find the product of two numbers - one stored in location 2:3 and another stored in location 5:2 - and then store the product back in the location 2:3

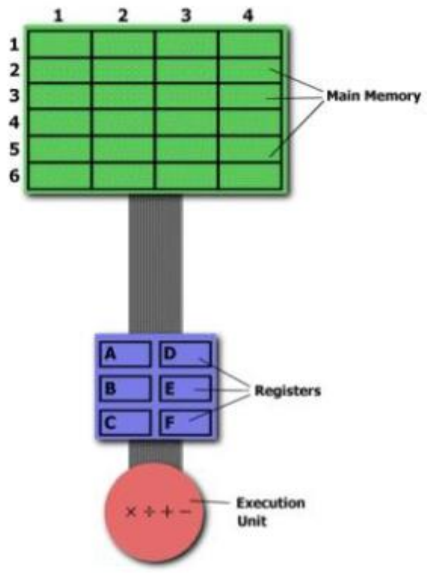


Fig 1.8: : Representation **of Storage Scheme for a Generic Computer**

The CISC Approach. The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations. For this particular task, a CISC processor would come prepared with a specific instruction (say "MUL").

- When executed, this instruction loads the two values into separate registers, multiplies the operands in the execution unit, and then stores the product in the appropriate register.
- Thus, the entire task of multiplying two numbers can be completed with one instruction:

MUL 2:3, 5:2

- MUL is what is known as a "complex instruction."
- It operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions.
- It closely resembles a command in a higher level language. For instance, if we let "a" represent the value of 2:3 and "b" represent the value of 5:2, then this command is identical to the C statement "a = a x b."

Advantage.

One of the primary advantages of this system is that the compiler has to do very little work to translate a high-level language statement into assembly. Because the length of the code is relatively short, very little RAM is required to store instructions. The emphasis is put on building complex instructions directly into the hardware.

The RISC Approach. RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the "MUL" command described above could be divided into three separate commands:

- "LOAD," which moves data from the memory bank to a register,
- "PROD," which finds the product of two operands located within the registers, and
- "STORE," which moves data from a register to the memory banks.

In order to perform the exact series of steps described in the CISC approach, a programmer would need to code four lines of assembly:

LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A

Analysis. At first, this may seem like a much less efficient way of completing the operation. Because there are more lines of code, more RAM is needed to store the assembly level instructions. The compiler must also perform more work to convert a high-level language statement into code of this form.

- ❖ Advantage of RISC. However, the RISC strategy also brings some very important advantages. Because each instruction requires only one clock cycle to execute, the entire program will execute in approximately the same amount of time as the multi-cycle "MUL" command. These RISC "reduced instructions" require less transistors of hardware space than the complex instructions, leaving more room for general purpose registers. Because all of the instructions execute in a uniform amount of time (i.e. one clock), pipelining is possible.

a) Separating the "LOAD" and "STORE" instructions actually reduces the amount of work that the computer must perform.

b) After a CISC-style "MUL" command is executed, the processor automatically erases the registers. If one of the operands needs to be used for another computation, the processor must re-load the data from the memory bank into a register. In RISC, the operand will remain in the register until another value is loaded in its place.

The following table will differentiate both the architectures and based on the analysis the overall advantage will be discussed.

RISC	**CISC**
Instruction takes one or two cycles	Instruction takes multiple cycles
Only load/store instructions are used to access memory	In additions to load and store instructions, memory access is possible with other instructions also.
Instructions executed by hardware	Instructions executed by the micro program
Fixed format instruction	Variable format instructions
Few addressing modes	Many addressing modes
Few instructions	Complex instruction set
Most of the have multiple register banks	Single register bank
Highly pipelined	Less pipelined
Complexity is in the compiler	Complexity in the microprogram

Table **Comparison of CISC and RISC Architectures**

- ❖ CISC Approach. The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction.
- ❖ RISC Approach. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program.

1.4 PROCESSOR ARCHITECTURE

<u>GENERAL INTERNAL ARCHITECTURE</u>

The general internal architecture of any processor is shown in Figure. Apart from the arithmetic logic unit (ALU), which performs all arithmetic and logical operations, every processor offers a set of general purpose registers for various storage and operations. Its status register accommodates the status of different arithmetic and logical operations, which might be necessary for conditional branching. Its program counter holds the address of next instruction word to be fetched from external memory. The stack pointer indicates the address of the stack-top.

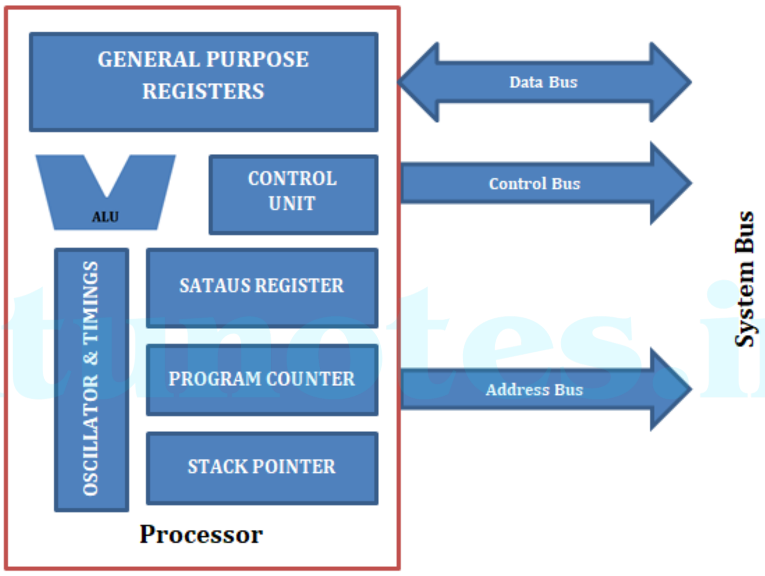


Fig 1.9 : General Internal Architecture of a processor

Two more architectural features of any processor are indicated in Figure. They are control unit and, oscillator and timing module. The control unit is responsible for generating all control signals and general working of the processor. This is achieved by the instructions with the help of internal clock, which is maintained by the oscillator unit.

BASIC FUNCTION

These architectural details of a processor are meant for executing a software. We should always remember that hardware and software must be dealt concurrently for any computer or any processor. Only hardware or only software would not be able to achieve any tangible outcome. ***The basic duty of any processor is to fetch, decode and execute instructions as long as it is powered on***. Unless it is a microcontroller, these instructions are available outside the physical boundary of the processor, within memory chips (ICs). These memory chips are electrically connected with the processor through a bunch of wires, designated as the bus.

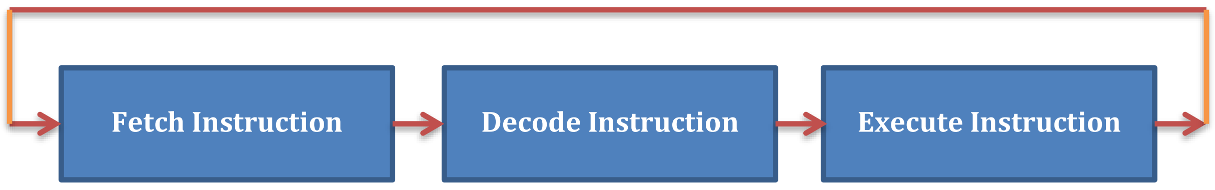


Fig 1.10: Basic Function of any Processor

Apart from fetching the opcode of executable instructions, sometimes it might be necessary for the processor to load or store operands in the external memory, if indicated so by the ongoing instruction Generally, the operation of a processor is sequential, which is diverted to another sequence due to conditional branching, subroutine calls and returning from subroutines, or to respond against any eventual external interrupt signal.

PERIPHERAL DEVICES AND EXTERNAL COMMUNICATION

Memory devices are not the only category of peripheral devices necessary for a processor to be functional. Later in this chapter, we shall see that a large number of peripheral devices are used to support different duties, as per the system requirements. These devices (non-memory devices) are, generally, referred as Input/output devices or I/O devices. Note that, just like memory devices, these I/O devices are also connected (or interfaced) with the processor through the bus.

Every processor offers three major types of bus.

- Address bus
- Data bus
- Control bus.

Out of these three bus, data bus is bi-directional, as data must come in and also go out of the processor, depending upon the specific requirements. Address bus is always unidirectional and it carries address signals from the processor to all external devices around it, memory and I/O. Most of the control signals also move out of the processor. Schematically, a generic processor’s external signals are presented in Figure.

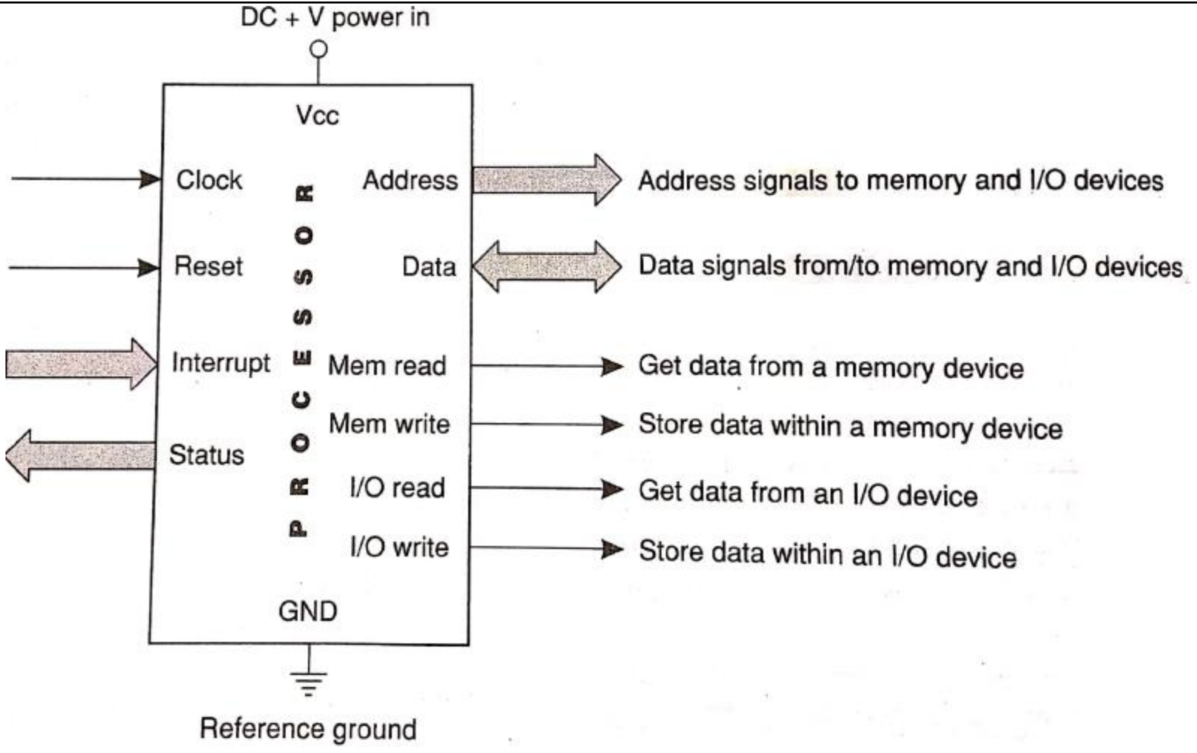

DC + V power in
Vcc
PROCESSOR
Clock
Reset
Interrupt
Status
Address
Data
Mem read
Mem write
I/O read
I/O write
GND
Reference ground
Address signals to memory and I/O devices
Data signals from/to memory and I/O devices
Get data from a memory device
Store data within a memory device
Get data from an I/O device
Store data within an I/O device

Fig 1.11; External signals of a generic processor

ADDRESS BUS AND ADDRESSING

The width of address bus or number of address lines available from any processor indicates its maximum memory size handling capability. The number of memory locations (bytes or words as the case might be) addressable by ***n*** address lines is 2^n. Therefore, if the processor offers 16 address lines then it can address 2^{16} or 64K locations (1 K = 2^{10} = 1,024). If it is offering 20 address lines then it can address 2^{20} or 1M locations and so on.

We have already discussed how the address bus helps in locating any desired data with any memory or I/O device. These address signals are decoded by a decoder inside the memory or I/O device to target the desired location.

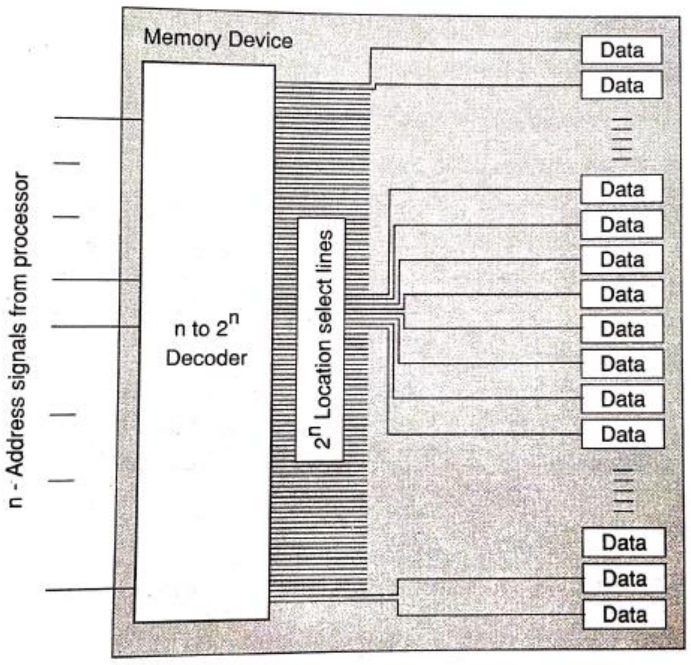


Fig 1.12 : Addressing of memory location by the processor

DATA BUS AND DATA FLOW CONTROL

The width of data bus of any processor indicates its simultaneous handling capability of the maximum number of bits. Generally, a processor is designated by its data bus width. For example, an 8-bit processor is capable of communicating 8-bit of data at the same time or having an 8-bit data bus. Similarly, a 16-bit processor has 16 parallel data lines for data communications.

The flow of data is bi-directional, depending upon whether the processor is interested in reading from or writing into the device (memory or I/O). This intension of the processor is expressed through its control signals (read and write). Depending upon this indication (read or write), the device (memory or I/O) enables the appropriate 3-state buffer to allow the flow of data signals from the data location already selected by address signals. The identical type of 3-state buffers is also present at the processor end in its data bus. This is illustrated through Figure, using 1-byte (8-bits) of storage area. Note that D0-D7 represents the data bus, interfaced with the processor. Eight flip-flops are for storage of data (8-bits) and at the input and output of each flip-flop, tri-state buffers are provided, whose control inputs are connected in parallel. The location-select signal from the decoder within the memory IC along with memory read or memory write signal enable these buffers. The clock signal acts in conjunction with memory write and select signals for the storage operation.

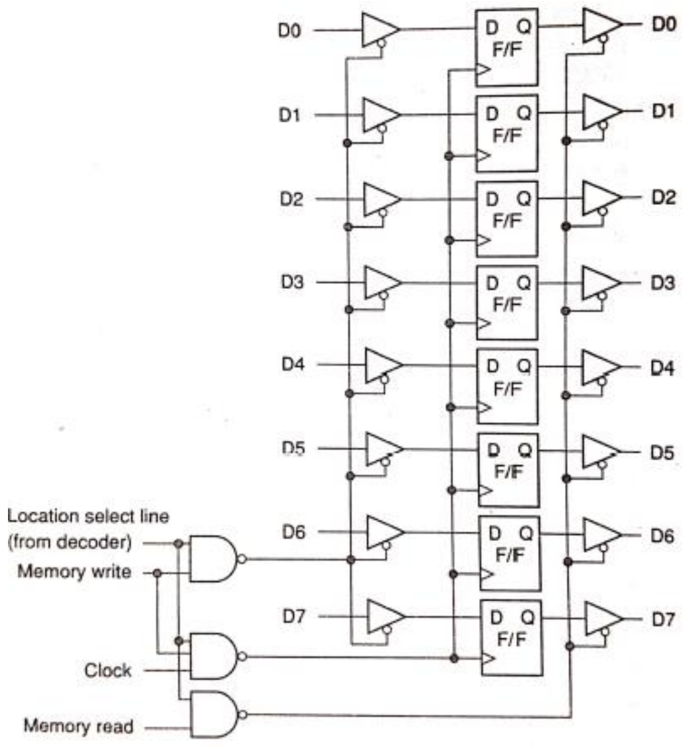


Fig 1.13 : Data flow mechanism between memory & processor

In Figure , the reader should note that although separate lines indicate input-to and output-from the tri-state buffers, the designation for any pair of buffers is the same, i.e., either both are DO or both are D1. Externally, these data line pairs are connected together to form a bus of 8 data lines, i.e., DO-D7.

CONTROL BUS

Number and functions of control signals, constituting the control bus varies widely with the processor itself. However, two of its important signals are READ and WRITE. Condition of these control signals indicate whether the present operation, intended by the processor is expecting the data in (READ) or sending the data out (WRITE). As the processor is to interact with two types of devices, memory and I/O, in general, four read/write signals are offered, as shown in Figure 1.11 . A few status signals are also available from the processor, apart from power input signals. Two more input signals are essential for all processors, namely clock and reset. Apart from these, a few external interrupt input signals are also provided in all processors.

The purpose of all these signals is to execute any program. The programs are composed of individual instructions. We shall now discuss how this program execution is implemented by the processor during its operational stage.

1.5 PROCESSOR OPERATION

The job of the processor is to execute programs, which are composed of multiple instructions. At this point, we should remember that instructions executable by the processor, are always in the machine code. Programs developed with high level language (HLL) instructions are first changed to this machine code, understandable by the processor. In general, the machine code instructions are extremely primitive and simple, e.g.,

- Copy a data byte from external memory to internal register or vice versa.
- Add two numbers available within the processor registers.
- If the result of subtraction is zero, then skip next three (or three thousand three hundred thirty three) instructions.

These instructions must be present in binary form within the memory of the system.

INSTRUCTION CYCLE

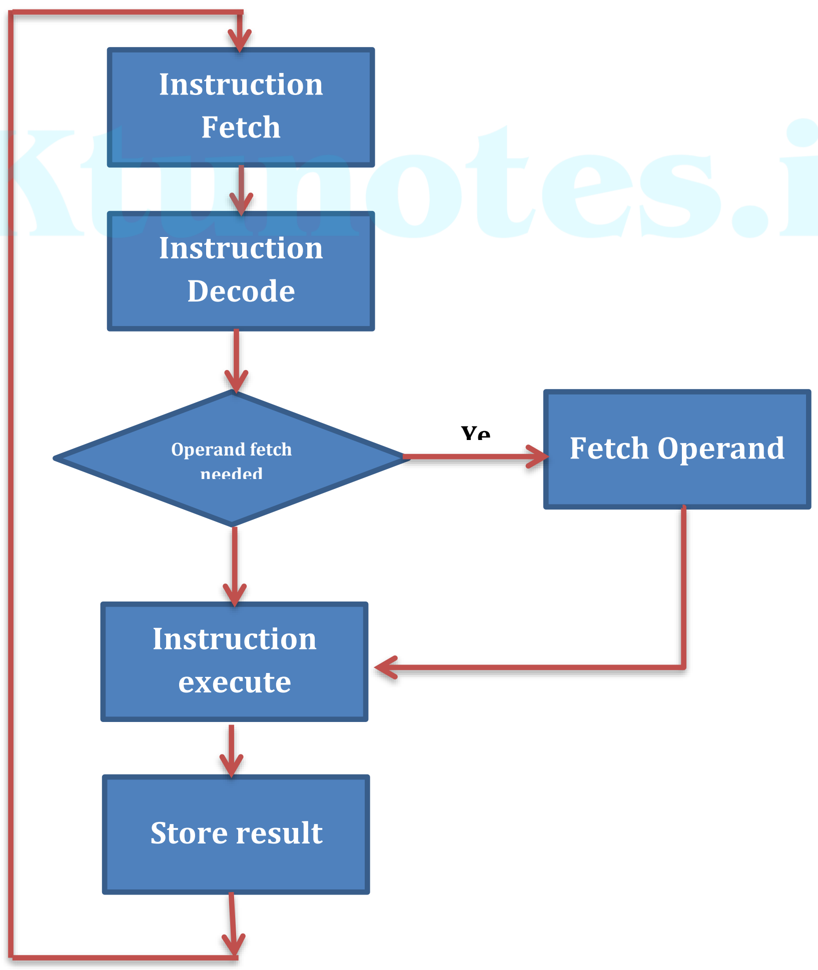


Fig 1.14 : Flow chart for simplified instruction cycle

To execute any type of instruction including those that are cited above, the processor should perform the following steps

- Fetch
- Decode
- Execute

Combination of these three steps is known as Instruction cycle. A flow chart of simplified form of instruction cycle of a generic processor is shown in figure

It may be observed from the flowchart that after fetching the instruction in the form of its opcode and decoding it, the processor checks for any eventual operand fetch, which might be necessary for some (not for all) instructions. If found necessary, then the operand is fetched from memory and then the instruction is executed. Finally, the result of the instruction is stored and the whole cycle is repeated. At this point, the reader may ask a question why this is designated as a simplified instruction cycle ? The answer is, we are avoiding many other details related with the instruction cycle, e.g., checking for any interrupt signal or looking for any direct memory access (DMA) request and so on. At a later stage, we shall consider all these details of the instruction cycle. We shall now discuss about the details of these three stages and some more related aspects.

INSTRUCTION FETCH

The first step, as indicated before, is to fetch the instruction byte(s) from external memory. This external memory is a vast area containing many bytes of instructions. Therefore, the processor must pin-point the correct location of this large memory area to extract the target byte.

It was already indicated that every memory location (byte in majority of cases) has a unique binary address. After receiving this address, the duty of the memory device is to decode the address to locate the target byte and place it on the data bus, so that the content of that address is available for the processor

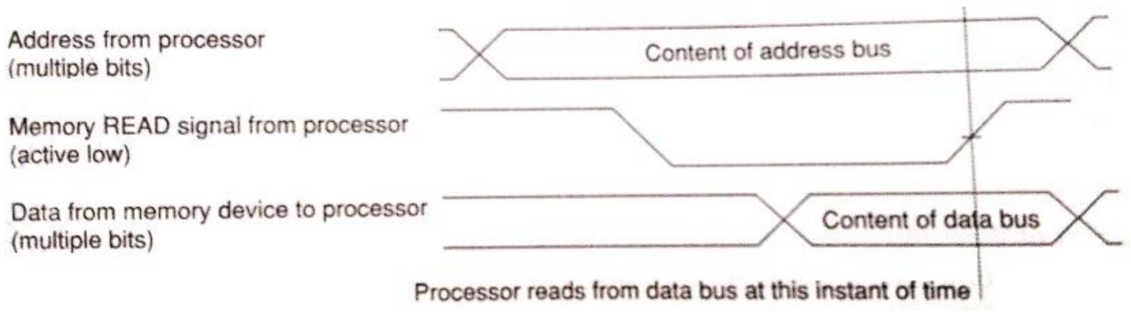

Address from processor
(multiple bits)
Content of address bus
Memory READ signal from processor
(active low)
Data from memory device to processor
(multiple bits)
Content of data bus
Processor reads from data bus at this instant of time

Fig 1.15 : Timing diagram for Instruction Fetch

Therefore, for the purpose of instruction fetch, the processor places an address, composed of multiple bits of binary information, on the address bus. Simultaneously, the processor also sends a memory read signal through its control bus. When these signals reach the memory device, the data are sent to the processor automat: by the memory device. Schematically, this transaction is depicted in Figure, which is known as timing diagram. Observe from Figure that data must be valid (stable) when the memory read signal goes from low to high. Address signals, generated by the processor, are stable at this stage to ensure a valid data transact

One question may arise here that how, out of so many devices interfaced with address, data and control bus, the correct device would pay attention to the processor's demand and that the other devices would remain silent? The answer is, every device has a chip select input (generally, designated as CE or CS) and if this input 1s not activated, the device does not react with the system bus communications, Using a part of the address lines and a suitable decoder (we have studied this in Chapter 3), the processor activates only one device during any communication and that solves the problem. This technique is known as address decoding and device selection. A processor is assisted with a memory decoder and an VO decoder to target the correct device, which is of current interest.

INSTRUCTION DECODE

After receiving the instruction code byte within itself, the processor becomes busy in understanding it (what to do?). This part is known as instruction decode, carried out within the processor itself. After the completion of instruction decoding, the processor knows whether to fetch operands from external memory or to increment a register by one or to store a register content in external memory location.

This instruction decoding may be implemented through hardware. Instruction decoding may also be implemented through software, known as micro-programming. This demands a miniature processor within the processor itself, completely devoted for instruction decoding and its execution.

INSTRUCTION EXECUTE

This is the last and final phase of an instruction's execution. Depending upon the instruction, one or Several operations are implemented by the processor. Once this part is complete, the processor looks forward for the next instruction fetch-decode-execute, and the process continues.

1.6 MACHINE CYCLE AND T-STATES

An instruction cycle has one or more machine cycles and every machine cycle is composed of several T-States. These points need some elaboration. A machine cycle is the step or time-slice during which 1-byte (or one word) of data are transacted between the processor and some external device. Generally, this external device is the memory device. However, in exceptional cases it might be an I/O device also. To transact 1-byte of information,
information, one machine cycle must be executed by the processor. In Figure 1.15 , we have illustrated such a machine cycle. Note, that instead of reading, it might be a writing operation also. Each machine cycle is composed of several T-states. One complete oscillation of the processor clock is designated as one T-state. Depending upon the

processor, the number of T-states necessary to complete one machine cycle must be known. For example, Intel 8085 processor needs four to six T-states to complete one machine cycle. The correlation of T-states, machine cycle and instruction cycle is shown in Figure 1.16 . Figure

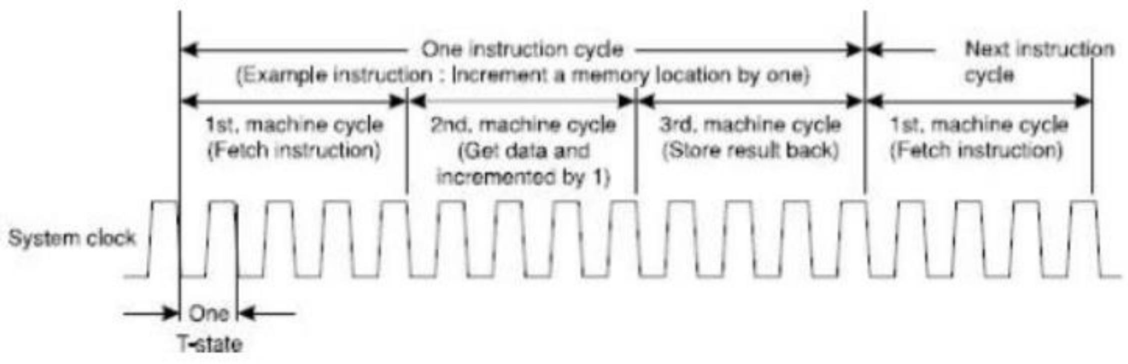

One instruction cycle
(Example instruction : Increment a memory location by one)
Next instruction cycle
1st. machine cycle (Fetch instruction)
2nd. machine cycle (Get data and incremented by 1)
3rd. machine cycle (Store result back)
1st. machine cycle (Fetch instruction)
System clock
One T-state

Fig 1.16 ***Example of instruction cycle, machine cycle and T-state correlation***

For the sake of example, execution of an instruction increment a memory location by one is illustrated through Figure 1.16. It is assumed that it is a 1-byte instruction, which is fetched by the first machine cycle. As the data, to be incremented by one, are available in external memory location, the next machine cycle reads this operand from memory

(brings the data byte within the processor). The data are then incremented by one by the processor and are stored back in the same memory location in the third machine cycle. Two questions may arise after this explanation of Figure 1.16, as follows

When the instruction was decoded?
When the data were incremented by one?

To answer the first question, it must be pointed out that the instruction must be decoded before the beginning of the second machine cycle, as the processor must know by that time what to do. As a rule, instruction decoding is carried out immediately after receiving the instruction byte within the processor. In other words, for every case, instruction decoding is done at the end of the first machine cycle. Does it not demand any extra time? Well it depends. If it is a hardware-based decoding, then it does not need any extra time. However, in case of micro-programming, it would consume one or two extra T-states. For example, Intel 8085 spends four T-states for fetching the first instruction byte during its first machine cycle, while for subsequent machine cycles it spends only three T-states. As a matter of fact, in its first machine cycle first three T-states are sufficient for fetching the first byte of instruction. Next T-state of the first machine cycle is devoted for instruction decoding. As the answer of the second question, we can say that the data would be incremented either at the end of second machine cycle or at the beginning beginning of the third machine cycle, depending upon the processor. Here also, the adopted technique plays an important role.

1.7 TIMINGS, CONTROL AND RESPONSE

Through the above discussions, it must be clear to the reader that timing and control play very important roles in smooth and efficient functioning of any processor. To further explain this concept, we may take up the example of interrupt. Although we shall have a detailed discussions on interrupt, it may be introduced here as an external asynchronous signal, which forces the processor to carry out something special for it by branching to a pre-defined address and, thus, executing a special program segment, known as interrupt service routine (ISR). As this is an asynchronous signal, it may be activated at any time during the execution of any instruction by the processor. However, the processor cannot leave an instruction's execution half-way to start doing something else for the sake of such an interrupting signal. To solve this problem, processors reserve a particular time-slot for checking the existence of any interrupt input signal during the execution of each and every instruction. For example, Intel 8085 processor had reserved the penultimate T-state of the last machine cycle of any instruction for this interrupt signal checking. If it is present, then the next instruction would not be executed immediately and the processor would start executing from the interrupt's ISR. However, the modified flowchart of the instruction cycle is presented in Figure 1.17, where the previously explained portion is shaded.

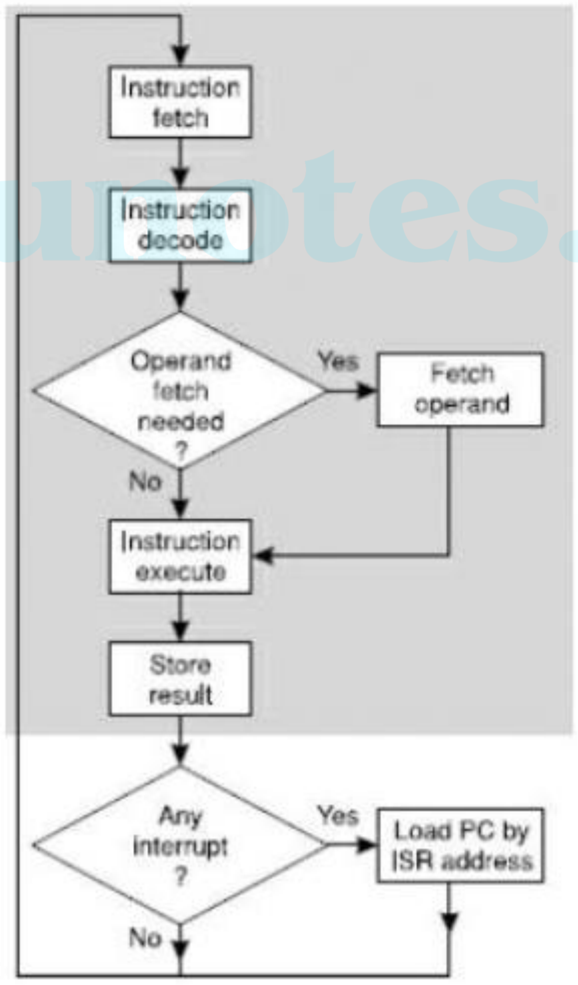


Fig 1.17: Modified Flow chart for simplified instruction cycle

1.8 REGISTER SET

To perform internal operations, all processors offer some internal registers, which can store temporary information or some operands. Similar to read/write memory, these registers are nothing but a combination of several flip-flops. Most of these registers are user (programmer) accessible and a few are not. The number of user accessible registers varies from processor to processor. Those processors that are memory oriented (e.g., Motorola 6800) offers lesser number of internal registers as it expects the data or operands would mainly be stored and manipulated within the read/write memory (RAM) of the system. On the other hand, some processors are register oriented (e.g., Zilog Z80), which offers a larger number of internal registers for the user. It may be noted that the program execution time for a processor would be less if the data are available within itself rather than looking outside for them. However, more internal registers means more complexity in instruction decoding as each register would demand a separate instruction to be provided by the instruction set of the processor. So far, we have been discussing about the general purpose registers. However, other types of registers are also available within the processors. They are accumulator or result register, status register, stack pointer, program

counter, interrupt register and so on. Most of these registers, in most processors, would be user accessible. Apart from these, there are some registers that are purely for processor's own use, e.g., temporary registers. We shall now have a brief discussion about some of these special purpose registers.

STATUS REGISTER

Every processor performs some arithmetic or logical operations generating some results. Depending upon whether the result is zero or negative or produced a carry or odd/even parity, some additional actions might have to be taken by the programmer. Status register solves this problem by offering the result status of the last performed arithmetic or logical operation through its pre-assigned bits. Generally, each bit of this status register is assigned for one particular indication, e.g., carry, parity, zero, overflow and so on. These bits act as flags and their conditions (true or false) help the program to decide further course of actions and dictate the conditional program branching.

ACCUMULATOR

In earlier processors, result of all arithmetic or logical operations were made available only in the accumulator. In more recent register-to-register architecture, all relevant registers available within the processor may contain the result of similar operations.

PROGRAM COUNTER

This is one of the most important registers within any processor as it is responsible for holding the address of the memory location for next instruction byte/word to be fetched by the processor. After fetching every instruction byte, this is automatically incremented by one to point to the next byte. The only exception for this auto-increment of the program counter is in the case of program branching, when it is reloaded by a new value. This counter is always initialized during system reset so that the first executable instruction byte is fetched from a pre-defined location of the memory.

STACK POINTER

System stack is a RAM area, which is earmarked by the programmer to accommodate important information, e.g., return address or register values, in last-in-first-out (LIFO) sequence. Stack pointer always points to the top of the stack area.

GENERAL PURPOSE REGISTERS

These registers are available within the processor for temporary data storage and manipulation. For arithmetic or logical operations, one of the two operands must be within these registers (the other one should be in the accumulator). As already indicated, number of these registers vary, depending upon the processor.

STACK ORGANIZATION

Stack is an area within the system RAM earmarked for some special storage by the program or programmer. In other words, the stack consists of several bytes of read–write memory where some special data may be stored in and restored from, as per the program's requirements. Why this cannot be accomplished by using the available registers within the processor? This is because, the number of registers is very limited and they have their other specific purpose rather than storing return addresses. Stack is, generally, used to store some important address and data sets. The particular location within the stack, where the next such information to be stored, is known as the stack-top. Generally, the address of this stack-top is available in the register designated for this specific purpose and known as Stack Pointer. Figure 1.18(a) illustrates a sample stack area within the address space between FFF0H and FFFFH (16-bytes) and the stack pointer. It is assumed that some stack locations are already occupied (used for storage) and the stack pointer has the address of the next free location of stack, i.e., FFF9H.

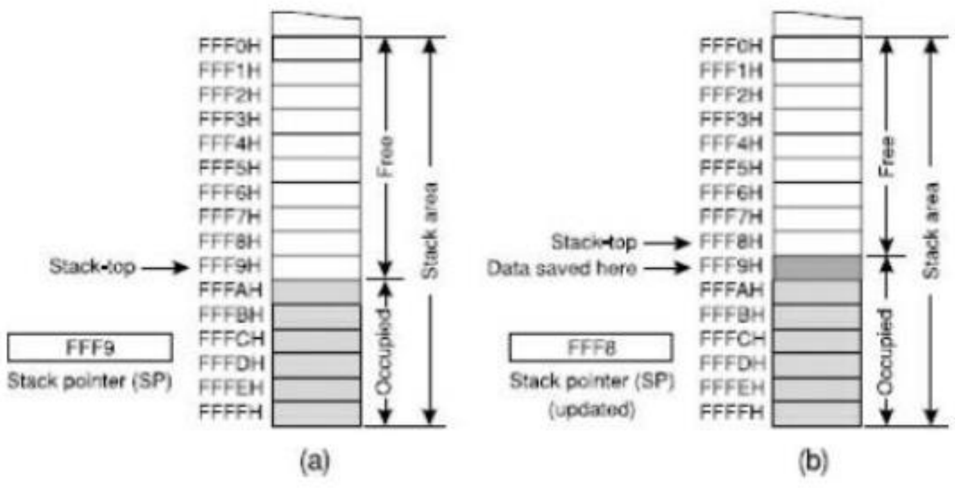


Fig 1.18 Stack and its operation

If any new data to be stored within the stack are included, it must be stored in the address pointed by the stack pointer, i.e., FFF9H, and in that case the stack pointer would show the next available free location for storage, i.e., FFF8H, as shown in Figure 1.18(b). Stack follows the last-in-first-out (LIFO) data movement technique. In other words, the data that is placed last on the stack-top must be retrieved first.

STACK AS STORAGE AREA

In general, every processor offers two instructions to handle the stack directly. These two instructions are PUSH and POP. PUSH instruction places the data on the stack-top and POP instruction takes it out from the stack-top. These data might be originally available within a general purpose register or some other data. It is already mentioned that the register within the processor, which holds the current stack-top address, is designated as stack pointer (SP). Whenever any data are placed on the stack-top or taken out from it, SP is also automatically changed by the processor itself. Here, we use the term 'changed' as, for some processors, a PUSH operation increments the SP while for other processors the SP is decremented for a PUSH instruction.

SUBROUTINES AND STACK

Stacks are widely used for subroutine calls. In these cases, the return address from the subroutine is placed on the stack-top before branching to the subroutine. As subroutines are always terminated by a RETURN instruction, once the execution of
the subroutine is complete, this RETURN instruction forces the processor to load the program counter from the stack-top, producing an effect of returning to the original part of the program that was left to branch to the subroutine. As an example case of flow of program control during execution of subroutine call and return instructions, a portion of a program is shown in Figure 1.19, which includes a call to one of its subroutines (Get Average). When this Call Get Average instruction is executed, the processor stores the address of the next instruction (18F3H) on stack-top and loads the program counter by the address of the subroutine 224BH. The reader may ask how does the processor know about the address of the subroutine? Well, the address of the subroutine is included within the call instruction itself. The control is then transferred automatically to the subroutine, which terminates with a Return instruction. During the execution of this Return instruction, the processor always reloads the program counter from stack-top and, in this case, it loads the program counter by the value 18F3H, which it had saved over the stack. Therefore, the control is now automatically transferred to the next instruction after the call instruction, and the execution proceeds sequentially thereafter. Stack is also essential for service interrupts, which we are about to discuss now.

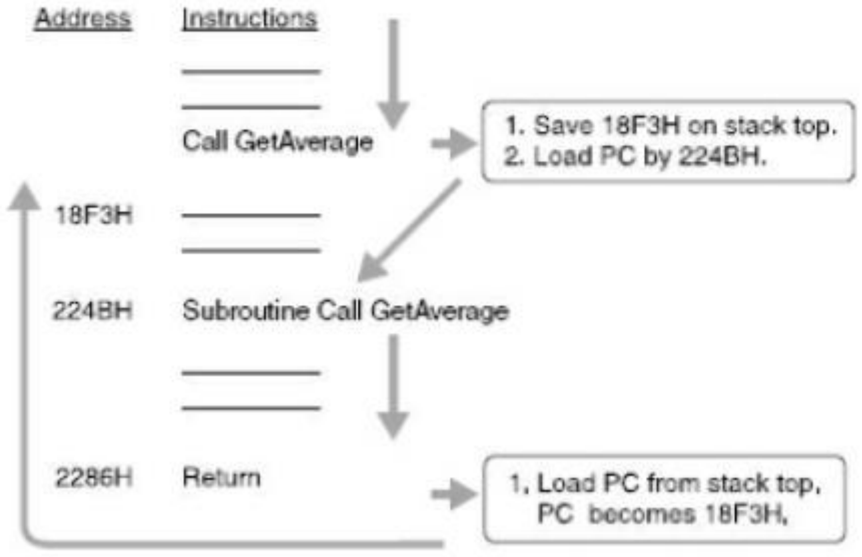


Fig 1.19: Functioning of stack during subroutine call and return

1.20 ALGORITHMS FOR BINARY MULTIPLICATION AND DIVISION

1.21 MULTIPLICATION ALGORITHMS

Apart from addition and subtraction, multiplication is another frequently used arithmetic operation. Several algorithms are available to implement it with binary numbers, both unsigned as well as signed. We shall show only a few of those in this section.

Paper and Pencil Method

The paper and pencil method that we adopt to perform multiplication of decimal numbers is also applicable for binary numbers, as illustrated in Figure 1.20 . Note its similarity with ANDing operation of Boolean algebra. To illustrate binary multiplication, we have selected two integers, 2 and 3 and shown their multiplication details by interchanging the multiplier and multiplicand to confirm that the order does not affect the result.

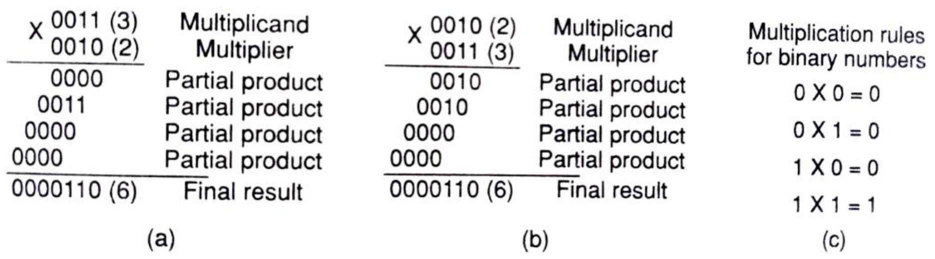


Fig 1.20 Example of multiplication method with binary numbers

In the first case 3 is multiplied by 2. As 3 in decimal is represented by 0011 and 2 by 0010 in binary, these binary values are written one below the other. Following the basic rule of multiplication as shown in Figure (c), four partial products are obtained. Note the way each partial product is placed in offset with the previous partial product, which we are familiar in our decimal multiplication. These partial products are finally added together to generate the result. The most important point may be noted here that the product of two 4-bit numbers may be as long as 8-bit. The same is applicable for 8-bit or 16-bit numbers, which may generate their products as 16-bit or 32-bit respectively.

Method of Repeated Additions

In computers, multiplication may be implemented in various methods. Another method is to perform repeated additions. With our example numbers, we may develop a program to add 2 three times to get the result 6. However, this method is time consuming in comparison to another method known as Booth's algorithm, which we are about to discuss now.

BOOTH'S ALGORITHM

Booth's algorithm (by Andrew D. Booth) for multiplication uses two's complement representation of binary numbers and is applicable for both positive and negative integers. Before discussing this algorithm, we should be familiar with one technique used as an important part in this algorithm, which is known as arithmetic right-shift.

Arithmetic right-shift

In normal right-shift operations of any microprocessor, all bits of the container, i.e., a register, are shifted uniformly one-bit towards right, making the present value of bit $_{(n-1)}$, = the old value of bit, In this process, the least significant bit is thrown out of the register, generally pushed into the carry flag, and a new bit, the content of the carry flag is inserted in the place of most significant bit. This procedure is usually designated as rotate-right-through-carry. In another variation of this right-shift, the carry flag does not come into the picture and the least significant bit is shifted into the most significant bit. This technique is designated as rotate-right-circular.

In case of arithmetic right-shift, all bits are shifted one bit right and the least significant bit is thrown out, identical as what happens in the case of a normal right-shift operation. The difference is in the condition of the most significant bit, which remains unchanged even though it is shifted to its right, after an arithmetic right-shift. It looks like as if after a normal right-shift, the old content of the most significant bit is copied back to its original place, keeping it unchanged. Therefore, in arithmetic right-shift, the most significant bit or the original sign-bit of the number remains unchanged. Two examples of arithmetic right-shift, one with a positive and another with a negative number are illustrated in Figure 1.21 using 4-bit format. Note that the same principle is applicable for 8-bit, 16-bit and all other bit formats in the same manner.

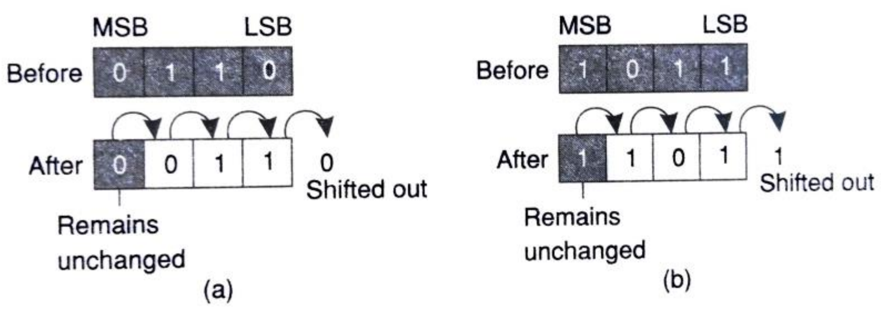


Fig 1.21: Example of 4-bit arithmetic right-shift

Locations and Counters

Booth's algorithm performs this arithmetic right-shift as many times as the number of bits involved. In other words, for a 4-bit representation of data, four such arithmetic right-shifts are performed. For 8-bit data sets, eight right-shifts are necessary and for 16-bit data, 16 right-shift operations have to be implemented.

Generally, a location (register) is used as counter for this purpose, which is initially loaded with the number of bits represented (a known and constant value for a computer system) and is decremented by one after every shift. When this counter becomes zero, the multiplication operation is considered to be completed as per Booth's algorithm. The register usage for Booth's algorithm is presented in Figure 1.22, which would be referred for explanation purpose. Instead of registers, we shall designate these as locations, which is a general designation, and this would be more appropriate in the present context.

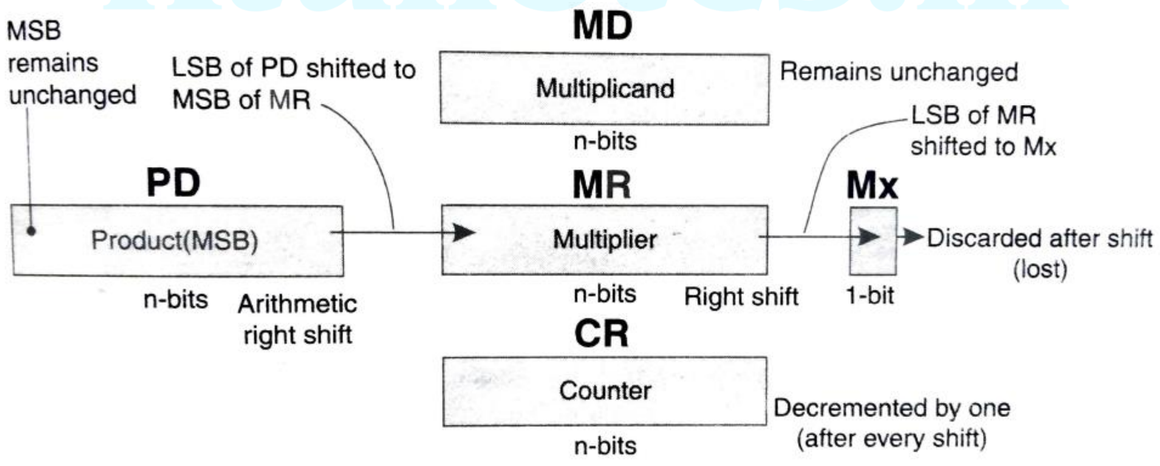


Fig 1.22 Locations involved for Booth's algorithm

- **MD** (n-bit) is for n-bit multiplicand
- **MR** (n-bit) is for n-bit multiplier (initially) and LS n-bit of product (finally)
- **CR** (n-bit) is for counter (from n to 0)
- **PD** (n-bit) is for MS n-bit of product (finally)
- **Mx** (1-bit) is for shift out from MR.

Details of Shift Operation

Now let us consider the shift operation which must be performed n times (till CR = 0). For the location PD ,it would be arithmetic right-shift. The LS bit of PD, coming out by this process, would be inserted within MS bit of MR and all original (old) bits MR are to be shifted one-bit right. The LS bit of MR, which would be coming out by this process, would be accommodated within Mx and the old (previous) content of Mx would be lost (discarded). Through Figure 1.23, all these operations are explained and may be correlated by the readers.

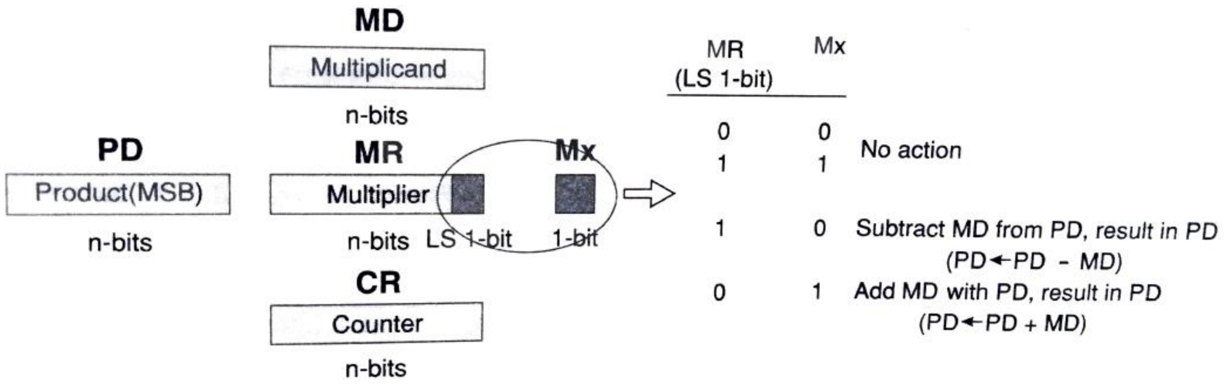


Fig 1.23 : Condition check and actions to be taken before every shift.

Finally, before implementing every shift operation, we are to ensure certain conditions and perform addition, subtraction or no such operation accordingly. The condition we are to check is the pattern generated by the two bits, the LS bit of location MR and 1-bit location Mx (both encircled in Figure 4.8). If these two bits are either 00 or 11, then we do not implement any addition or subtraction and directly proceed to the shift-right operation. However, if the pattern is 10 (LS bit of MR is 1 and Mx containing 0) then we are to subtract content of MD from the present content of PD and the result would be placed in PD. Note that the original content of PD would be lost. Any eventual borrowing during this subtraction would be neglected. It is needless to indicate that two's complement addition may be performed in place of subtraction. On the other hand, if the pattern is 01 (LS bit of MR being 0 and Mx having 1), then we are to add MD with the current content of PD and the result would be stored in PD overwriting PD's old content. Any carry generated by this subtraction or addition would be neglected. All four conditions and actions to be taken against each are shown at right side of Figure 1.23.

Algorithm and Flowchart

Having discussed all basic techniques related to Booth's algorithm, we are now in a position to discuss the steps involving it. The algorithm is presented through the following steps and also presented as flowchart through Figure 1.24.

- ***Step I:*** Load multiplicand in MD, multiplier in MR. For negative numbers, two's complement format to be used.
- ***Step 2:*** Initialize the down counter CR by the number of bits involved.

- ***Step 3:*** Clear locations PD (n-bits) and Mx (1-bit).
- ***Step 4:*** Check LS bit of MR and Mx jointly. If the pattern is 00 or 11 then go to Step 5. If 10, then PD = PD — MD. If 01, then PD = PD + MD.
- ***Step 5:*** Perform arithmetic right-shift with PD, MR and Mx. LS of PD goes to MS of MR and LS of MR goes to Mx. Old content of Mx is discarded.
- ***Step 6:*** Decrement CR by one. If CR is not zero then go to Step 4.
- ***Step 7***: Final result (or the product) is available in PD (higher part) and MR (lower part).

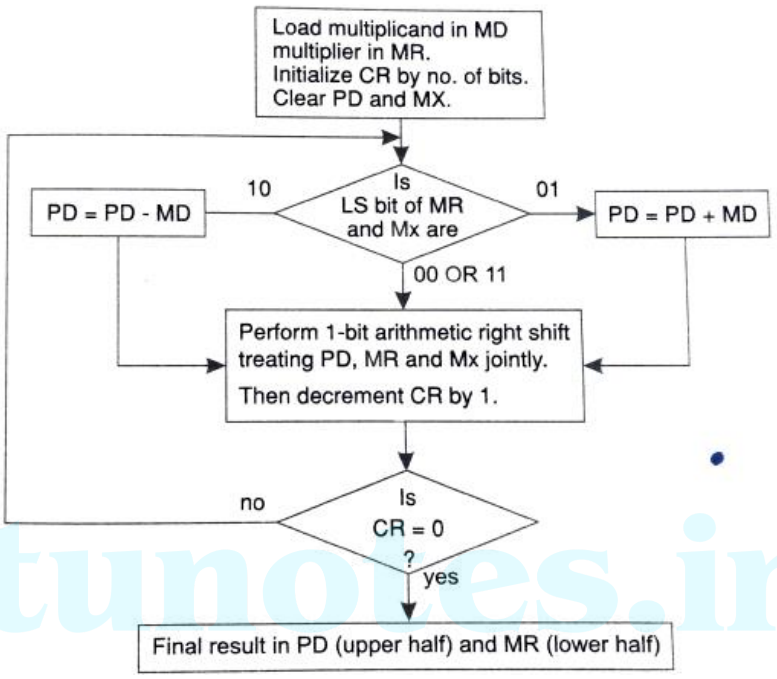

Load multiplicand in MD
multiplier in MR.
Initialize CR by no. of bits.
Clear PD and MX.
PD = PD - MD
10
Is LS bit of MR and Mx are
01
PD = PD + MD
00 OR 11
Perform 1-bit arithmetic right shift treating PD, MR and Mx jointly.
Then decrement CR by 1.
no
Is CR = 0 ?
yes
Final result in PD (upper half) and MR (lower half)

Fig 1.24 : Flow chart for Boot's Algorithm

Example 1 : 2 X 3

All locations are used except Mx, would be 4-bit locations as the maximum range of numbers covered is less than 7, in our case. We initialize MD by loading 2 (0010) and MR by 3 (0011). PD and Mx are cleared to 0000 and 0, respectively, and CR 1s loaded by 4, to count the number of cycles of iteration.
To start the first cycle, we check the pattern formed by LS bit of MR and Mx (indicated by an underline in the sketch). As they are 10, we subtract MD from PD (conditions indicated at right side of Figure 1.23), For subtraction, we calculate two's complement of 0010 in MD, which is 1110. This is added with PD (presently 0000) and the result 1110, is placed in PD. The next step is to perform one-bit arithmetic right-shift, with PD, MR and Mx together. After the shift, we get PD as 1111, MR as 0001 and Mx as 1. Note that LS bit from PD is shifted to MS bit of MR and LS bit of MR goes to Mx, as we have discussed before. The next step is to decrement the counter CR by 1, which becomes 3. This completes the

first cycle of iteration and three more cycles remain. The reader may note that in Figure 1.25, those locations which do not contribute in an operational phase are lightly shaded.

As the counter CR is not 0, we start the second cycle by checking two bits, LS of MR and Mx. This is because, they appear as 11, there is no need for any addition or subtraction and we may perform only the arithmetic right-shift operation using PD, MR and Mx. After shifting right, PD becomes 1111, MR . becomes 1000 and Mx becomes 1. By decrementing CR by one, we complete the second cycle, and then two more are remaining.

We then check two bits of MR and Mx (underlined) at the starting of the third cycle and as they are 01, we perform an addition of PD (presently 1111) with MD (having 0010). We place the result of this addition, 0001, in PD and then perform one-bit arithmetic nght-shift as before. This shifting makes PD as 0000, MR as 1100 and Mx as 0. Counting down the location CR, we get 1 and now we may start our last cycle.

As the two bits to be checked (LS of MR and Mx) are presently 00, we skip any addition or subtraction and simply perform the arithmetic right-shift of all the three locations (PD, MR and Mx) by one bit. This would leave 0000 in PD, 0110 in MR and 1 in Mx. Lastly, we decrement CR by 1 and as it is 0, we terminate the whole process. Note that the final product is now available as an 8-bit value within locations PD and MR (thick underlined), placed side by side. Location PD is to contain the most significant half and MR accommodates the least significant half of the product. In our example case, it contains 0000 0110, which is the correct answer (6 1n decimal).

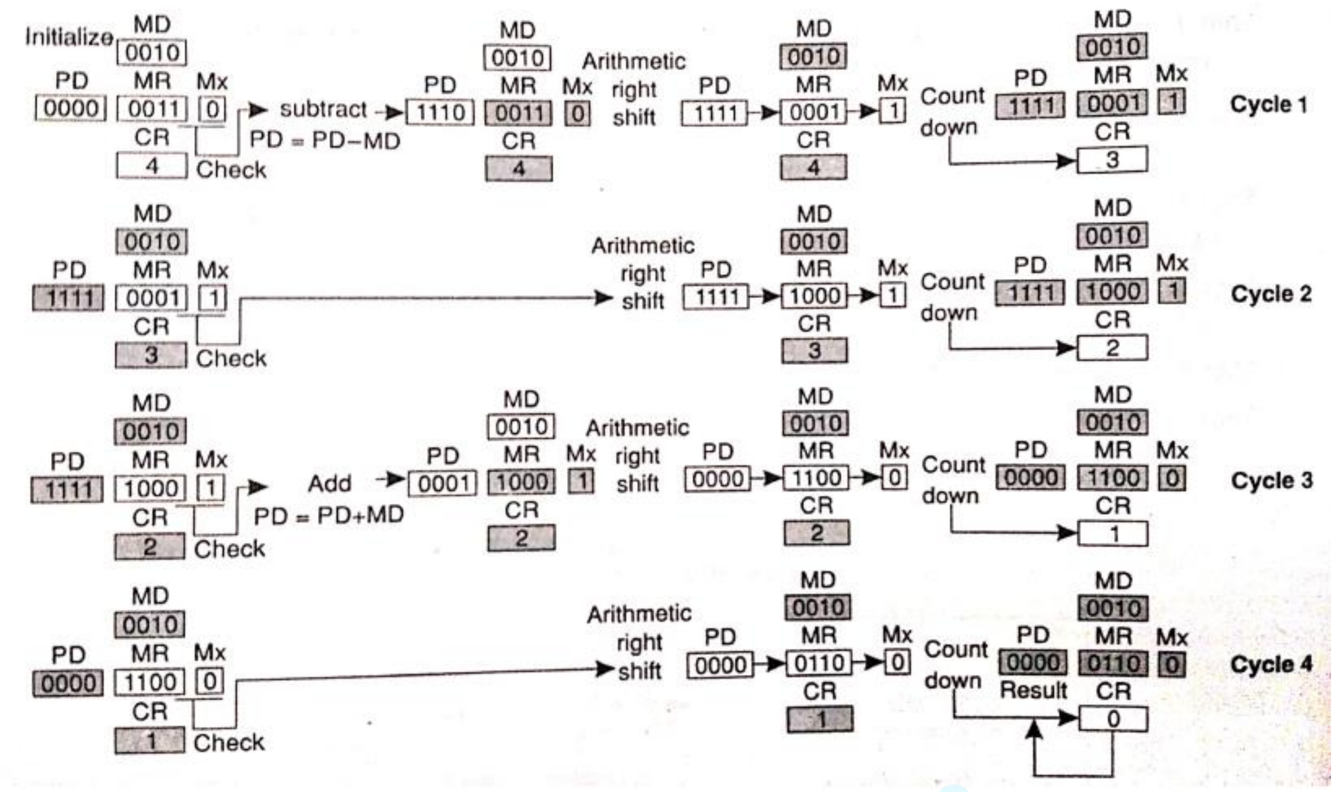


Fig 1.25: Illustration of Booth's algorithm for 2X3

Example 2 :   2 x (-3)

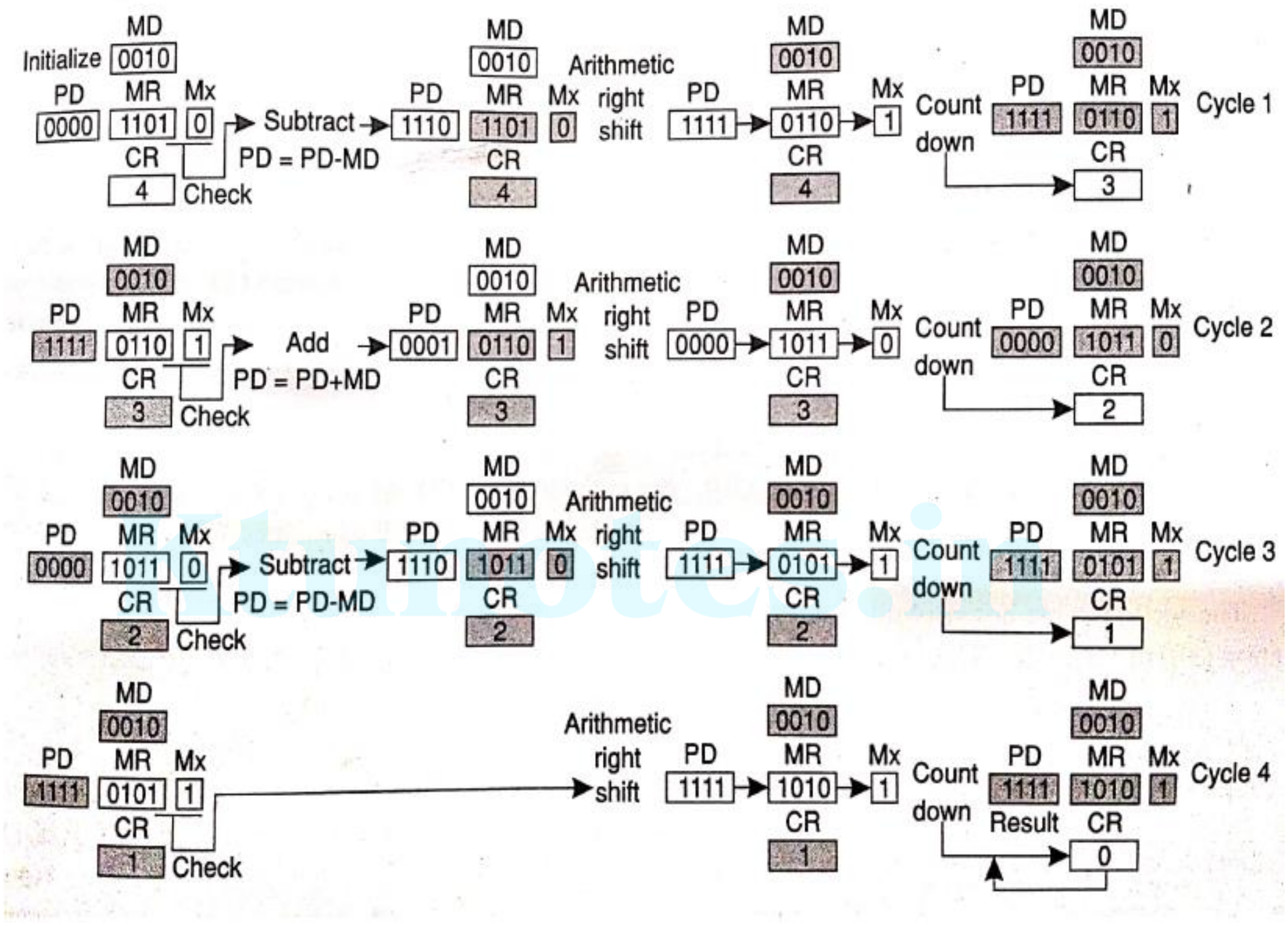


Fig 1.26: Illustration of Booth's algorithm for 2X-3

Example 2 : (- 2) x 3

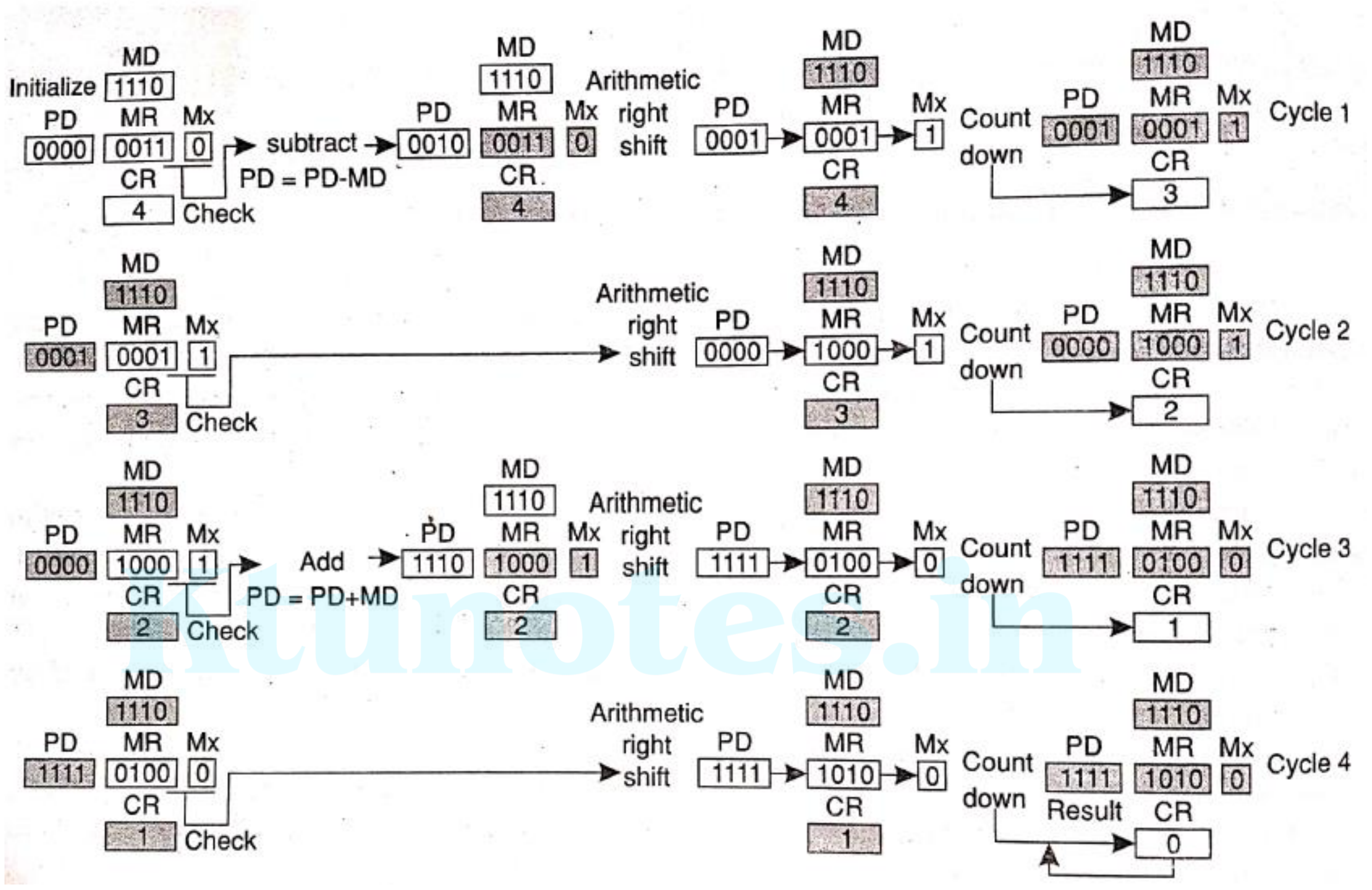


Fig 1.27: Illustration of Booth's algorithm for- 2X3

Example 2 : (- 2) x (- 3)

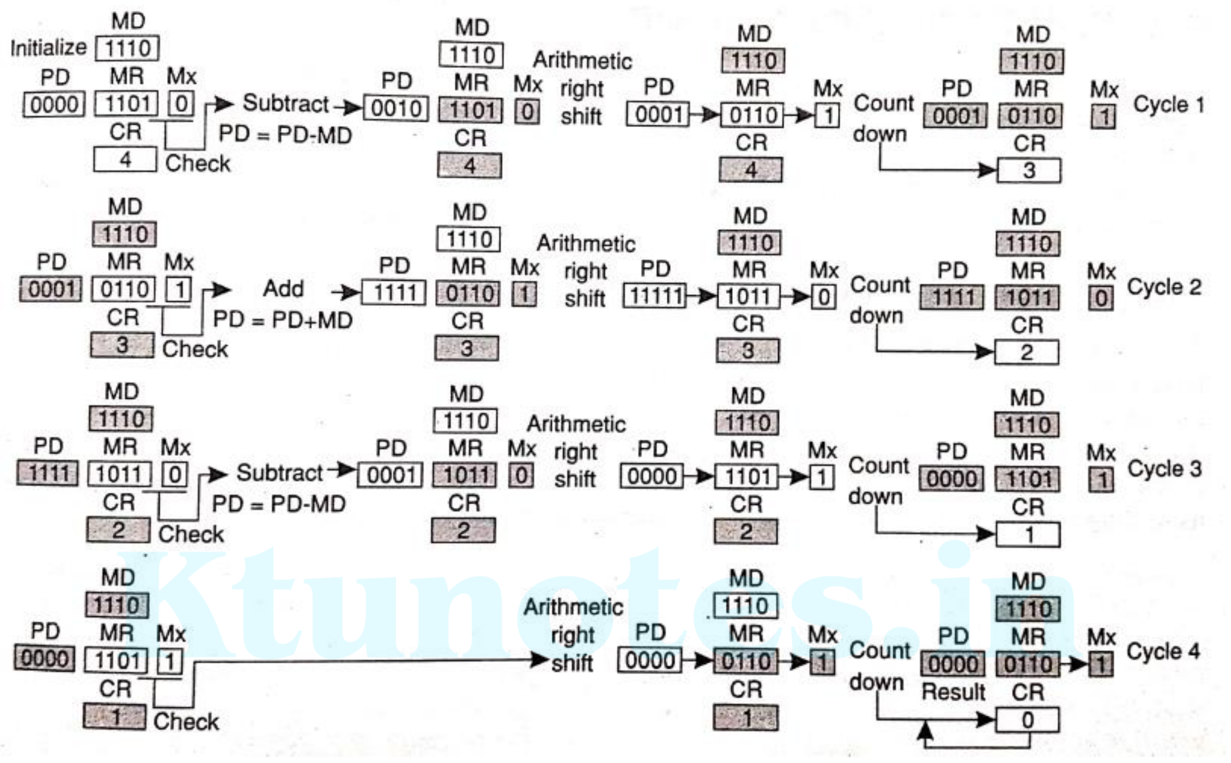

Initialize
MD 1110
PD 0000 MR 1101 Mx 0
CR 4
Check
Subtract
PD = PD-MD
PD 0010 MR 1101 Mx 0 CR 4
Arithmetic right shift
PD 0001 MR 0110 Mx 1 CR 4
Count down
PD 0001 MR 0110 Mx 1 CR 3
Cycle 1
PD 0001 MR 0110 Mx 1 CR 3
Add
PD = PD+MD
PD 1111 MR 0110 Mx 1 CR 3
PD 11111 MR 1011 Mx 0 CR 3
PD 1111 MR 1011 Mx 0 CR 2
Cycle 2
PD 1111 MR 1011 Mx 0 CR 2
Subtract
PD = PD-MD
PD 0001 MR 1011 Mx 0 CR 2
PD 0000 MR 1101 Mx 1 CR 2
PD 0000 MR 1101 Mx 1 CR 1
Cycle 3
PD 0000 MR 1101 Mx 1 CR 1
PD 0000 MR 0110 Mx 1 CR 1
Result
PD 0000 MR 0110 Mx 1 CR 0
Cycle 4

Fig 1.28: Illustration of Booth's algorithm for -2X-3

1.22 DIVISION ALGORITHMS

Just like multiplication may be carried out by repeated additions, division also may be completed by repeated subtraction till a negative reminder is encountered. However, there are other algorithms also for performing divisions with integers. We may also adopt the method used by us to perform division in longhand method (paper and pencil method). One such sample calculation is shown in Figure 1.29 . We spend some time on this as that would give us some insight related to the methods of unsigned binary division.

Paper and Pencil Method

To illustrate paper and pencil method of division, we take 5 as dividend and 2 as divisor. In 4-bit format, these numbers may be written as 0101 and 0010, respectively. This is shown in Figure 1.29(a). To initiate the process of division, the dividend has to be scanned from left to right, one digit at a time, and if not divisible, a zero to be placed in the place of quotient. Therefore, we start our scanning and first encounter with a zero [underlined in Figure 1.29 (b)]. As it is less than the divisor (10, we may neglect its leading zeros in this case), the first zero is inserted at the quotient [Figure 1.29 (b)].

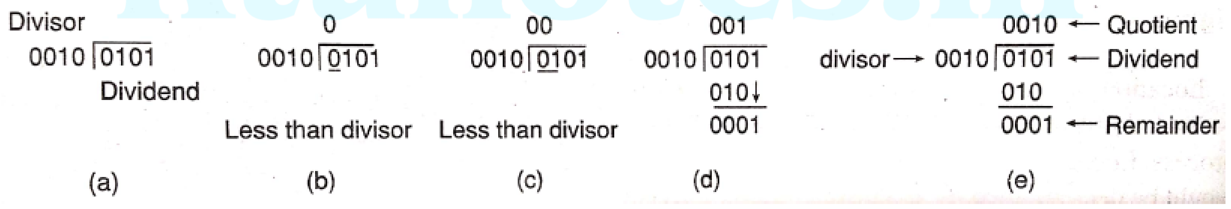


Fig 1.29 : Example of division by paper & pencil Method

We continue our scanning from left to right and next encounter with the left most two digits of the dividend, i.e., 01 (underlined). Again it is found to be less than the divisor, which forces us to add another zero in the quotient [Figure 1.29(c)]. As we continue our scanning, we next meet with 010 part of the dividend (its three leading digits) and find that it is equal (even greater would do) to the divisor. At this point, we may add a | in the place of quotient, making it 001, write the divisor below the dividend and perform a subtraction. The remainder in this case is 000. We then write the next considerable digit of the dividend, i.e., 1 at the right side of the remainder [shown by a vertical downward arrow in Figure 1.29(d)].

As the last step, we find that the present remainder 0001 is less than divisor. Therefore, we add another zero with the quotient making it 0010 and complete our process. Thus, finally, we get a quotient of 0010 (2 in decimal) and a remainder 0001 (1 in decimal) by our process, matching with our expectations.

An algorithm may be developed for the above method to divide unsigned integers. However, this method would not be applicable for signed integers, expressed in two's complement form. For that purpose, we have to adopt another algorithm (William Stallings, 2009), as described below.

Locations and Counters

To implement this algorithm for division of signed integers, we would need four locations; all should be n-bit wide, where 'n' is the number of bits being considered for divisor and dividend. In other words, for 4-bit numbers, we shall need 4-bit wide locations, for 8-bit numbers, we shall need 8-bit wide locations and so on. We designate these four locations as V, R, D and C, as indicated in Figure 1.30.

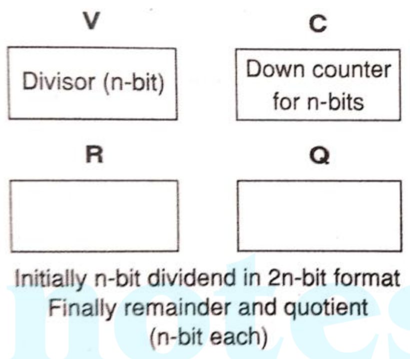

V
Divisor (n-bit)
C
Down counter for n-bits
R
Q
Initially n-bit dividend in 2n-bit format
Finally remainder and quotient (n-bit each)

Fig 1.30: Locations involved for division algorithm.

Location V, is to contain n-bit divisor. If the divisor is negative, its two's complement form should be loaded in V. We shall only use its content, which would remain unchanged till the completion of the process. Location C is the n-bit down counter and initially to be loaded by n. At the end of every cycle, C would be decremented by 1. The operation of division would be complete when C becomes 0. The n-bit dividend must be expanded to 2n-bit form and be loaded in locations R and Q, where R would contain the most significant part. If negative, then the dividend should be changed to its two's complement form and expanded from n-bit to 2-bit format. In other words, if 2 is 0100, then it should be expanded as 00000100 and if 7 is 1101, then it should be expanded to 11111101. However, at the end of operation, R would contain n-bit remainder and Q would contain n-bit quotient

One bit Left-Shift

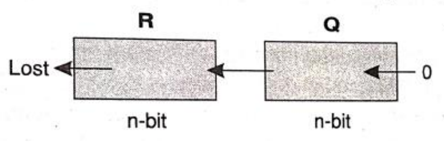


Fig 1.31: Detail of one-bit left shift operation for division algorithm

Let us consider the shifting technique to be implemented at the beginning of each iteration. In this case it would be a one bit left-shift considering R and Q, simultaneously. A zero has to be inserted at the LS bit of Q, and the MS bit of Q should be shifted to LS bit of R. The MS bit of R would be moved out and discarded.

Algorithm for Division of Signed Integers

The algorithm for the division of signed integers is explained through the following steps.

- **Step I:** Load V by n-bit divisor using two's complement form for negative numbers.
- **Step 2:** Load n-bit dividend within R and Q after expanding it to 27-bit format maintaining its sign as it is.
- **Step 3:** Load C by the number of bits being considered, i.e., n.
- **Step 4:** Perform 1-bit left-shift with R-Q, inserting a 0 at the LS bit of Q.
- **Step 5:** If the divisor (in V) and R have same sign, then replace R by R — V, otherwise replace R by R+V.
- **Step 6:** If the sign of R remains unchanged after Step 5, or R becomes 0, then set LS bit of Q as 1. Otherwise, if the sign of R changes after Step 5 and R becomes non-zero, then restore the value of R as it was after Step 4.
- **Step 7:** Decrement C by 1 and if it is not 0, then go to Step 4.
- **Step 8:** Find the remainder in R. If the divisor and dividend have same signs, then Q indicates the quotient. Otherwise, its two's complement would be the correct quotient.

Note that steps 4 to 7 are involved in the iteration process. Steps 1 to 3 are for initialization and the Step 8 is for getting the correct result. We next discuss two example cases for implementing this division algorithm.

Example : **5÷2**

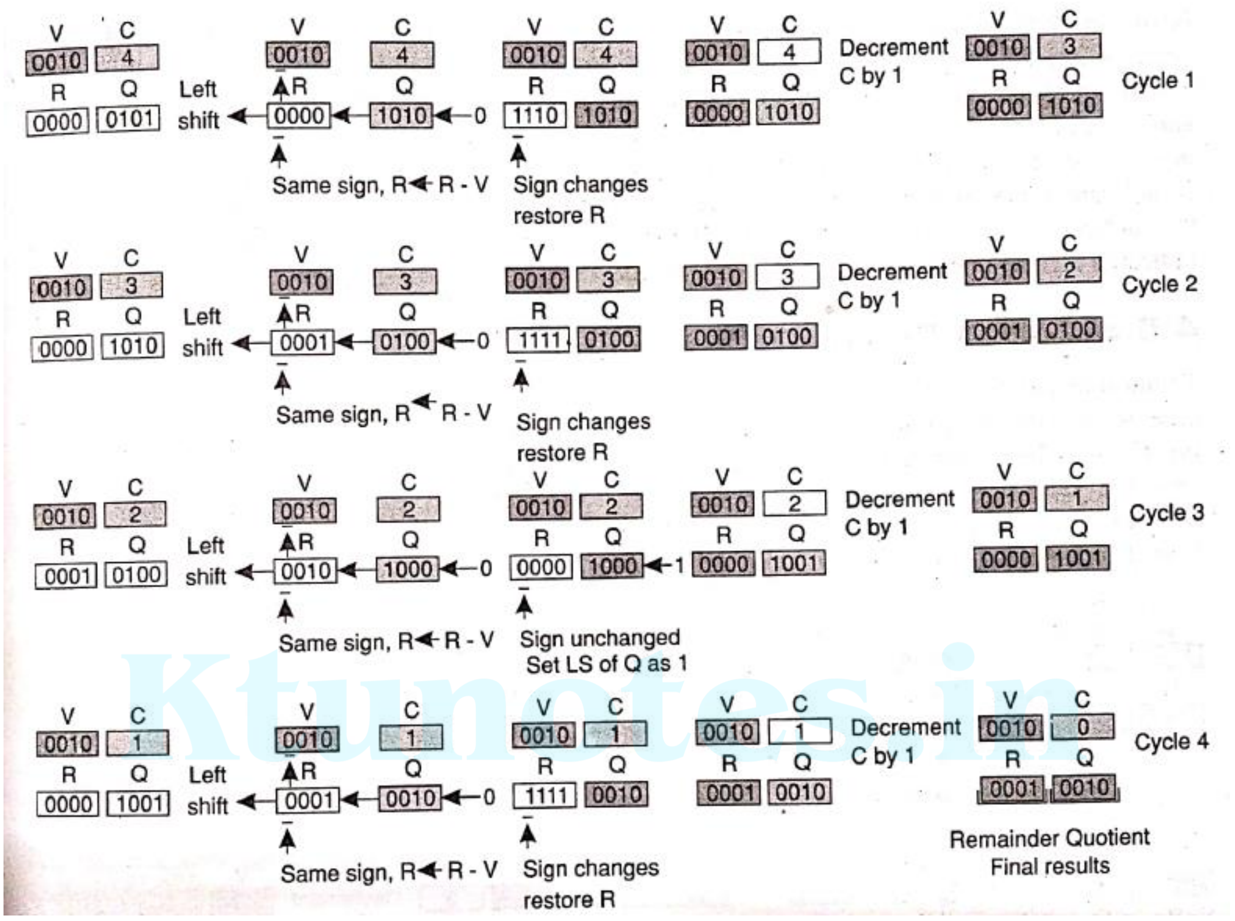


Fig 1.32 : Illustration of division algorithm for 5 divided by 2

We start with initialization by loading the divisor (2 in decimal) in location V in its binary form of 4-bit, i.e., 0010. The counter C is initialized as 4 to represent 4-bit operation. The dividend (5 in decimal) is converted to its 8-bit representation, i.e., 00000101 and its lower half 0101 is loaded within location Q and the upper half 0000 1s loaded in location R.

The first cycle begins with a one-bit left-shift of R and Q and a 0 is inserted at the least significant position of Q. This changes Q to 1010 and R remains as 0000 after left-shift. As the MS bit of R and MS bit of V are both 0, a subtraction operation is to be performed to replace R by R — V. So, two's complement form of V, 1.e., 1110 is added with R, i.e., 0000 to get 1110, which becomes the value in R at this point. As there is a sign change between the present content of R (1110) and its previous content (0000), the previous value of R is restored and R becomes 0000 again. By decrementing C from 4 to 3, we complete the first cycle and three more cycles are pending.

The second cycle begins with a left-shift of R and Q together changing R to 0001 and Q to 0100. As the present signs of V and R are identical (both are 0), V is subtracted from R and the result (1111) is placed in R. This is because of the sign change of R in this process, R is restored to its original value of 0001. The counter C is decremented by | to make it 2.

In the third cycle, locations R and Q are jointly shifted one-bit left, making R as 0010 and Q as 1000. As V and R are having same sign at this point, therefore, R is replaced by the result of R ~ V, which is 0000.Note that the sign of R remains unchanged forcing us to set the least significant bit of Q as 1.This changes Q to 1001 and R remains as 0000. Finally, C is decremented to 1, completing the third cycle.

The fourth cycle starts with the left-shift of R as well as Q, changing R to 0001 and Q to 0010. As R and V are of same sign, R is replaced by R — V, which is 1111. This change in sign of R with respect to is previous sign forces us to replace R by its old value of 0001. C is now decremented to 0 indicating the completion of the division operation. The result of the division is now available at Q and R. Q contains the quotient, i.e., 0010 or 2 in decimal and R contains the remainder 0001 or 1 in decimal. These results indicated that the operation is correctly performed.

Example : **-5 ÷ -2**

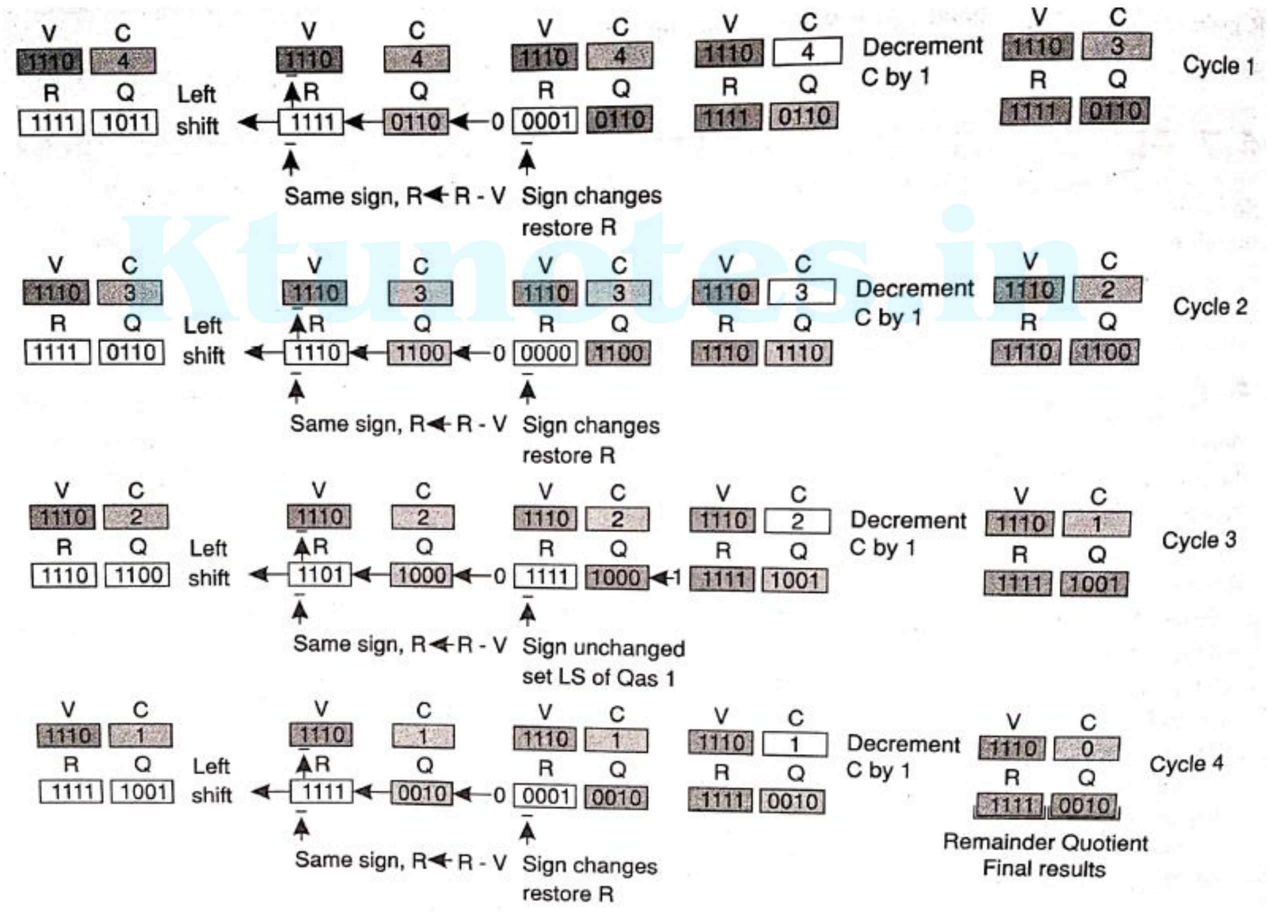


Fig 1.33 : Illustration of division algorithm for -5 divided by -2

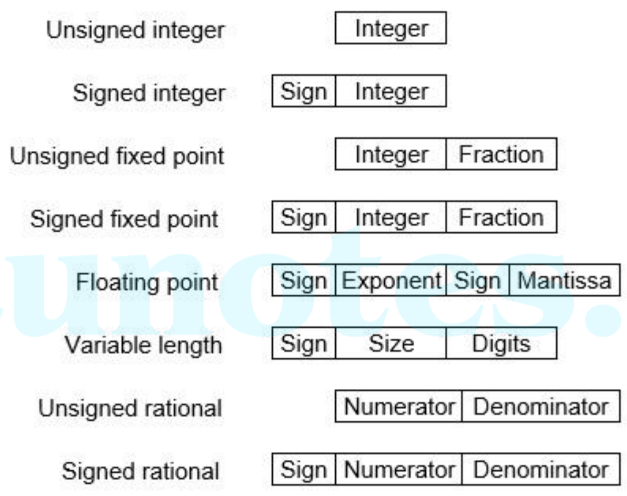
FIXED POINT AND FLOATING POINT NUMBER REPRESENTATIONS

Digital Computers use Binary number system to represent all types of information inside the computers. Alphanumeric characters are represented using binary bits (i.e., 0 and 1). Digital representations are easier to design, storage is easy, accuracy and precision are greater.

There are various types of number representation techniques for digital number representation, for example: Binary number system, octal number system, decimal number system, and hexadecimal number system etc. But Binary number system is most relevant and popular for representing numbers in digital computer system.

Storing Real Number

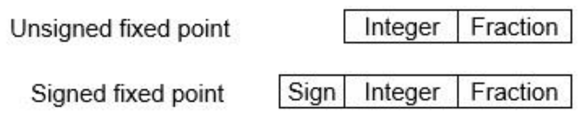
These are structures as following below −



There are two major approaches to store real numbers (i.e., numbers with fractional component) in modern computing. These are (i) Fixed Point Notation and (ii) Floating Point Notation. In fixed point notation, there are a fixed number of digits after the decimal point, whereas floating point number allows for a varying number of digits after the decimal point.

Fixed-Point Representation

This representation has fixed number of bits for integer part and for fractional part. For example, if given fixed-point representation is IIII.FFFF, then you can store minimum value is 0000.0001 and maximum value is 9999.9999. There are three parts of a fixed-point number representation: the sign field, integer field, and fractional field.



We can represent these numbers using:

- ❖ Signed representation: range from $-(2^{(k-1)}-1)$ to $(2^{(k-1)}-1)$, for k bits.
- ❖ 1's complement representation: range from $-(2^{(k-1)}-1)$ to $(2^{(k-1)}-1)$, for k bits.
- ❖ 2's complementation representation: range from $-(2^{(k-1)})$ to $(2^{(k-1)}-1)$, for k bits.

2's complementation representation is preferred in computer system because of unambiguous property and easier for arithmetic operations.

Example –Assume number is using 32-bit format which reserve 1 bit for the sign, 15 bits for the integer part and 16 bits for the fractional part.

Then, -43.625 is represented as following:

1	000000000101011	1010000000000000
Sign bit	Integer part	Fractional part

Where, 0 is used to represent + and 1 is used to represent **–.** 000000000101011 is 15 bit binary value for decimal 43 and 1010000000000000 is 16 bit binary value for fractional 0.625.

The advantage of using a fixed-point representation is performance and disadvantage is relatively limited range of values that they can represent. So, it is usually inadequate for numerical analysis as it does not allow enough numbers and accuracy. A number whose representation exceeds 32 bits would have to be stored inexactly.

Smallest	0	000000000000000	0000000000000001
	Sign bit	Integer part	Fractional part
Largest	0	111111111111111	1111111111111111
	Sign bit	Integer part	Fractional part

These are above smallest positive number and largest positive number which can be store in 32-bit representation as given above format. Therefore, the smallest positive number is $2^{-16} \approx$ 0.000015 approximate and the largest positive number is $(2^{15}-1)+(1-2^{-16})=2^{15}(1-2^{-16})$ =32768, and gap between these numbers is 2^{-16}.

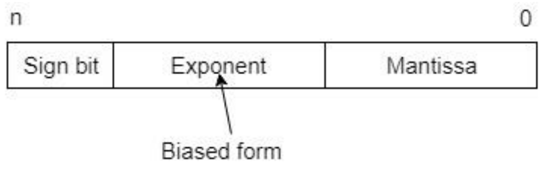
We can move the radix point either left or right with the help of only integer field is 1.

Floating-Point Representation

This representation does not reserve a specific number of bits for the integer part or the fractional part. Instead it reserves a certain number of bits for the number (called the mantissa or significand) and a certain number of bits to say where within that number the decimal place sits (called the exponent).

The floating number representation of a number has two part: the first part represents a signed fixed point number called mantissa. The second part of designates the position of the decimal (or binary) point and is called the exponent. The fixed point mantissa may be fraction or an integer. Floating -point is always interpreted to represent a number in the following form: $M \times r^e$.

Only the mantissa m and the exponent e are physically represented in the register (including their sign). A floating-point binary number is represented in a similar manner except that is uses base 2 for the exponent. A floating-point number is said to be normalized if the most significant digit of the mantissa is 1.



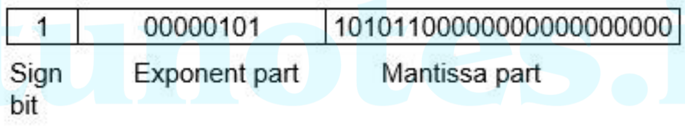
So, actual number is $(-1)^s(1+m)x2^{(e\text{-}Bias)}$, where *s* is the sign bit, *m* is the mantissa, *e* is the exponent value, and *Bias* is the bias number.

Note that signed integers and exponent are represented by either sign representation, or one's complement representation, or two's complement representation.

The floating point representation is more flexible. Any non-zero number can be represented in the normalized form of $\pm(1.b_1b_2b_3 \ldots)_2x2^n$ This is normalized form of a number x.

Example –Suppose number is using 32-bit format: the 1 bit sign bit, 8 bits for signed exponent, and 23 bits for the fractional part. The leading bit 1 is not stored (as it is always 1 for a normalized number) and is referred to as a *"hidden bit*".

Then −53.5 is normalized as $-53.5=(-110101.1)_2=(-1.101011)x2^5$, which is represented as following below,

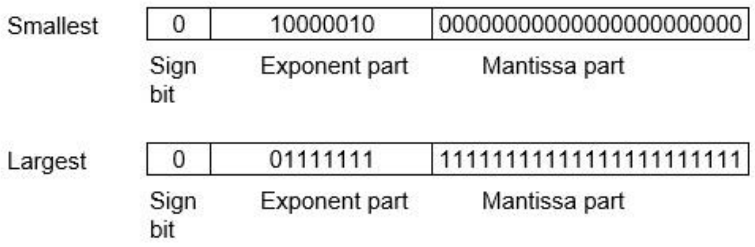


1	00000101	10101100000000000000000
Sign bit	Exponent part	Mantissa part

Where 00000101 is the 8-bit binary value of exponent value +5.

Note that 8-bit exponent field is used to store integer exponents $-126 \leq n \leq 127$.

The smallest normalized positive number that fits into 32 bits is $(1.00000000000000000000000)_2x2^{-126}=2^{-126}\approx1.18x10^{-38}$, and largest normalized positive number that fits into 32 bits is $(1.11111111111111111111111)_2x2^{127}=(2^{24}-1)x2^{104} \approx 3.40x10^{38}$. These numbers are represented as following below,



Smallest	0	10000010	00000000000000000000000
	Sign bit	Exponent part	Mantissa part
Largest	0	01111111	11111111111111111111111
	Sign bit	Exponent part	Mantissa part

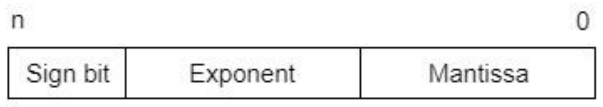
The precision of a floating-point format is the number of positions reserved for binary digits plus one (for the hidden bit). In the examples considered here the precision is 23+1=24.

The gap between 1 and the next normalized floating-point number is known as machine epsilon. the gap is $(1+2^{-23})-1=2^{-23}$for above example, but this is same as the smallest positive floating-point number because of non-uniform spacing unlike in the fixed-point scenario.

Note that non-terminating binary numbers can be represented in floating point representation, e.g., 1/3 = $(0.010101 \ldots)_2$ cannot be a floating-point number as its binary representation is non-terminating.

IEEE Floating point Number Representation –

IEEE (Institute of Electrical and Electronics Engineers) has standardized Floating-Point Representation as following diagram.



So, actual number is $(-1)^s(1+m)\text{x}2^{(e\text{-Bias})}$, where *s* is the sign bit, *m* is the mantissa, *e* is the exponent value, and *Bias* is the bias number. The sign bit is 0 for positive number and 1 for negative number. Exponents are represented by or two's complement representation.

According to IEEE 754 standard, the floating-point number is represented in following ways:

- Half Precision (16 bit): 1 sign bit, 5 bit exponent, and 10 bit mantissa
- Single Precision (32 bit): 1 sign bit, 8 bit exponent, and 23 bit mantissa
- Double Precision (64 bit): 1 sign bit, 11 bit exponent, and 52 bit mantissa
- Quadruple Precision (128 bit): 1 sign bit, 15 bit exponent, and 112 bit mantissa

Special Value Representation –

There are some special values depended upon different values of the exponent and mantissa in the IEEE 754 standard.

- All the exponent bits 0 with all mantissa bits 0 represents 0. If sign bit is 0, then +0, else -0.
- All the exponent bits 1 with all mantissa bits 0 represents infinity. If sign bit is 0, then +∞, else -∞.
- All the exponent bits 0 and mantissa bits non-zero represents denormalized number.
- All the exponent bits 1 and mantissa bits non-zero represents error.