Lecture Notes
ON
COMPUTER ORGANISATION
AND
ARCHITECHTURE

Namita Das
TITE

→ A Computer system is subdivided into two functional entities: ① Hardware
② Software.

Hardware The hardware of the computer consists of all the electronic components and electromechanical devices that compreises the physical entities of the device i.e. computer.

Software consists of the instructions and data that the computer manipulates to perform various data processing task.

→ A sequence of instructions for the computer is called program.

→ The data that are manipulated by the program constitute the data base.

* In this paper we will discuss the hardware operation
* of the computer system. So we will first understand
* what is Computer organization & computer architecture.

Computer Organization (user bound)

Computer organization means the way the hardware components operates and the way they are connected together to form the computer system.

Computer Architecture (system bound)

Computer architecture means the structure and behavior of the computer seen by the user. It includes the information

formats, the instruction set and techniques for addressing memory.

* The architectural design of a computer system is deals with the specification of various functional modules, such as processor and memory, and structuring them together into a computer system.

Computer is a fast electronic calculating machine that accepts digitized input information, process it according to a list of internally stored instruction and produces the resulting output information.

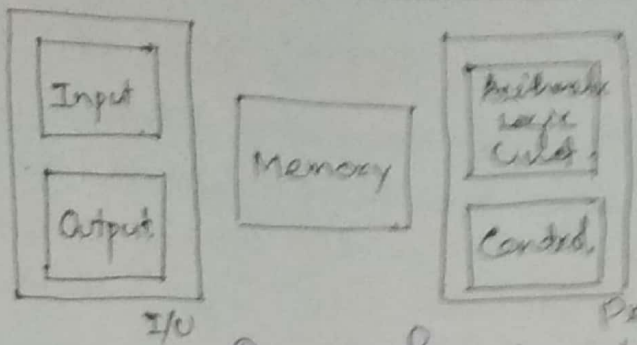
A Computer Consists of five functionally independent main parts: Input.
Memory
Arithmetic & Logic unit.
output
control unit.

→ The input unit accepts coded information from human operators, from electromechanical devices (keyboard of a computer or from other Computer) through digital communication lines.

→ The information received is either stored in memory for later reference or immediately used by the arithmatic and logic circuit to perform desired operations.

→ Finally the result is sent back to the outside world through output unit.

→ All of these actions are coordinated by control unit.



Basic functional Units of a computer

→ Now we will analyze what happens when the information fed to the computer,

Information is of two types
- instruction
- data

INSTRUCTION or M/C instruction are commands that
- Govern the transfer of information within a computer as well as between computer and I/O Devices.
- Specify the arithmetic and logic information to be performed.
- A set of instruction that perform a task is called a program. Usually the program is stored in memory.
- When processor fetches the instruction to execute the program, it brings the program from memory, one at a time, and perform the desired operation.
- The computer is completely controlled by the stored program, except for possible external interruption.

DATA are numbers and encoded characters that are used as operands by the instruction.

COMPIER TRANSLATES THE SOURCE PROGRM IN TO A MACHINE LANGUAGE PROGRAM.

Basic Operational Concepts

→ The activity of a computer is governed by instructions.

→ To perform a given task, a appropriate program consisting of a set of instruction is stored in the memory. Individual instruction is brought from the memory into the processor, which executes the specified operations.

→ Data to be used as operands are also stored in memory.

→ A typical instruction may be

Add LOC A, R0

⇒ This instruction add the operand at memory location LOCA to the operand in a register inside the processor, R0, and place the sum into R0.

→ The original content of LOCA are preserved whereas those of R0 is overwritten.

→ The process of execution of the instruction can be subdivided into several steps.

↳ First the instruction is fetched from the main memory into the processor.

↳ Next the operand at LOCA is fetched and add to the content of R0.

↳ Then finally the result of sum is stored in R0.

⇒ In the above instruction combines a memory access instruction with an ALU operation.

↳ In many modern computer these two types of operation are performed by separate instruction for performance reason.

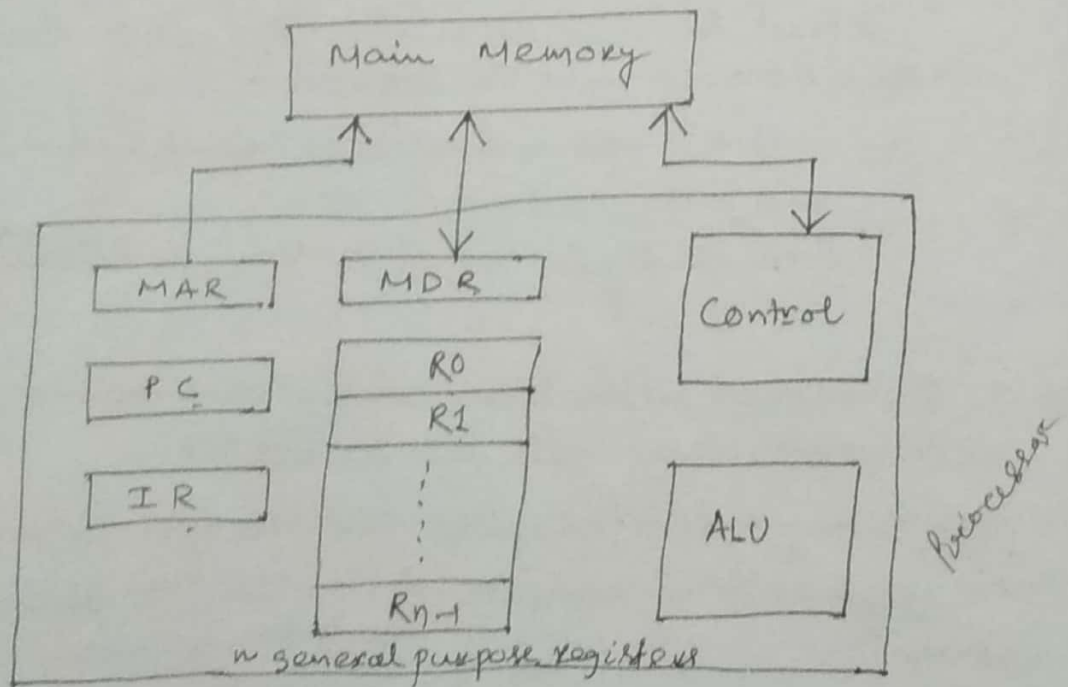
→ The same instruction can be written in two instructions as follows.

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

→ The first instruction transfers the contents of main memory LOCA into processor register R1.
Second instruction adds the contents of R0 & R_1 and place the sum in R0.

⇒ Transfer between the main 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 signal.
Then the data are transferred to or from the memory.

Here we will discuss ~~with~~ how the main memory and the processor are connected, with necessary figure below. It also shows a few operational details of the processor.



Here the interconnection part is not shown explicitly, because here we only discuss the functional characteristics of the components.

→ In addition to the ALU and the control circuitry, the processor contains a number of registers used for temporary storage of data.

→ 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) register keeps track of the execution of a program. It contains the memory address of the instruction currently being executed.

↳ During the execution of an instruction, the contents of the PC are updated to the address of the next instruction to be processed.

→ WE CAN SAY THAT PC POINTS TO THE ADDRESS OF NEXT INSTRUCTION THAT IS TO BE FETCHED FROM THE MEMORY.

→ There are n general-purpose registers, R0 to Rn-1.

→ Another two registers are there to perform the communication with the main memory.

MAR (Memory Address Register)

The MAR holds the address of the location to or from which data are to be transferred.

MDR (Memory Data Register)
The MDR contains the data to be written into or read out of the addressed location.

Now we will analyze a complete operating steps.

→ Programs reside in the main memory, and usually get there through the input unit.

→ Execution of the progm starts when the PC set to point to the first instruction of the program

→ The contents of the PC is transferred into the MAR and a read control signal is sent to the memory.

→ After the time required to access the memory, the first instruction of the program is read out of the memory and loaded into the MDR.

→ Next the contents of the MDR is transferred into IR. At this point the instruction is ready to be decoded and executed.

⇒ If the instruction involves an operation to be performed by ALU, it is first necessary to obtain the required operand.

→ If the operand resides in the memory, its address is sent to the MAR, and initiating a read cycle.

→ Then the operand has been read from the memory and placed inside MDR, and it may be transferred from the MDR to ALU.

→ Then the ALU can perform the desired operation.

→ If the result is to be stored in memory, then the result is sent to the MDR, and the address of the location where the ~~data~~ result is to be stored is sent to the MAR and the write process initiated.

→ While an instruction is being executed, the content of the PC are incremented so that the PC contains the address of the next instruction to be executed.

~~In addition to transfer data from~~

→ Sometime the computer accepts data from input device and sends data to output devices. Thus we need some machine instruction with the ability to handle I/O transfers are provided.

INTERUPT

→ Normal execution of a program may be pre-empted if some device requires urgent servicing. To do this, the device raises an interrupt signal. (An Interrupt is a request from an I/O device for service by the processor).

→ The processor provide the required service by executing an appropriate interrupt service routine.

→ Because such Interrupt alter the internal state of the processor, its state must be saved in memory location before serving the interrupt.

→ Normally the content of PC, the general registers, and some Control information are stored in memory.

→ After the interrupt service routine is completed, the previous state of the processor is restored.

→ The total processor unit can be implemented with the use of VLSI chip.

Bus Structures

Word is the natural unit of organization used by processor. A word consists of 8, 16, 24, 32 or 64 bits. A modern general purpose computer usually has 32 or 64 bits.

Individual parts of a computer have their respective functions, but to form an operational system, these parts must be connected in some organized way.

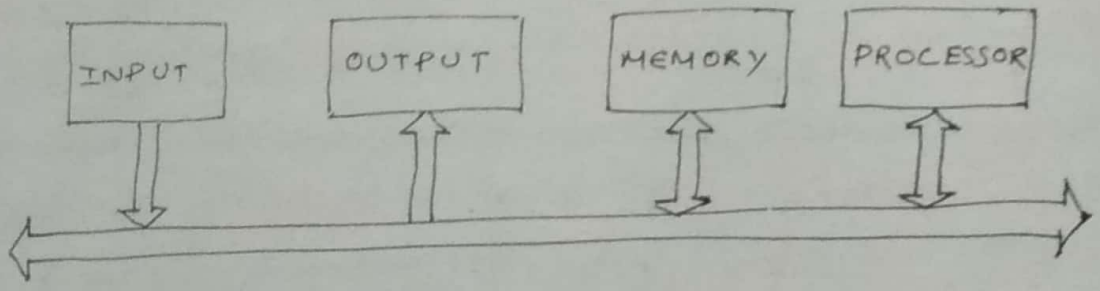
To achieve a reasonable speed of operation, a computer must be organized so that all its units can handle one full word of data at a given time.

→ When a word of data is transferred between units, all its bits should transferred in parallel, to do this a group of different wires are necessary.

→ A group of wires that connects several devices is called a bus.

→ Inside a computer along with the data bus, it has some bus for carrying address and control signals.

→ The simplest way to interconnect functional units, is to use a single bus as shown in figure bellow.



All units are connected to the main bus.

→ Because the bus can be used for only one transfer at a time, only two units can actively use the bus at a given instant.

→ So the only advantage of the single bus structure is its low cost and its flexibility for attaching peripheral devices.

→ But the system that contains multiple bus achieve more parallelism, so it gives a better performence but at an increased cost.

Buffer Register.

The different devices connected to a bus are varying with respect to their speed of operation. Like, the speed of keyboard, printer is much slower than memory and processor.

→ But these devices must communicate with each other over a bus, an efficient transfer mechanism is required.

→ So common solution is to include buffer-register with the slower devices to store the information during transfers.

example

Suppose we want to transfer information from processor to printer. The processor sends the characters to printer through bus. Though a buffer register attached to the printer and it is an electronic device, this transfer requires relatively less time (processor to buffer register). Once the buffer is loaded, the printer can start printing without further interaction with processor. So the bus and processor is no longer needed by the printer and can be engaged for other activity. By this, a high speed processor is being prevented from being locked to a slow I/O device during a sequence of data transfer. This allows the processor to switch rapidly from one device to another.

Software

When a user wants to enter and run an application program, the computer must already have some system software.

→ System software is a collection of programs that are executed as needed to perform function such as:

- Receiving and interpreting user command.
- Entering and editing application programs and storing them as file in secondary storage.
- Managing the retrival & storage of files in secondary storage.
- Running standard application programs such as spreadsheets or games with data supplied by the user.
- Controlling I/O units to receive input information and produce output results.
- Translating programs from source code prepared by the user into object code consisting of machine instructions.
- Linking user written application programs with existing standard library routines. (Such as numerical computation package).

Some basic aspects of system Software

→ Application progᵐ are written in high level language (such as C, C++, Pascal, Fortan). This type of progᵐ is not understandable by the computer. Simultaneously the user uses the high level language progᵐ need not know the machine level progᵐ.

A system software prog^m called Compiler translates the high level language prog^m into a suitable m/c language program.

→ Another system program is there which is use full for textentry and editing program. The user of the program interactively executes commands that allows statements of a source prog^m enter at a keyboard to be accumulated in a file. A file is simply a sequence of alphanumeric characters or binary data i.e. stored in memory or in secondary storage.

→ Most important system software is operating system.

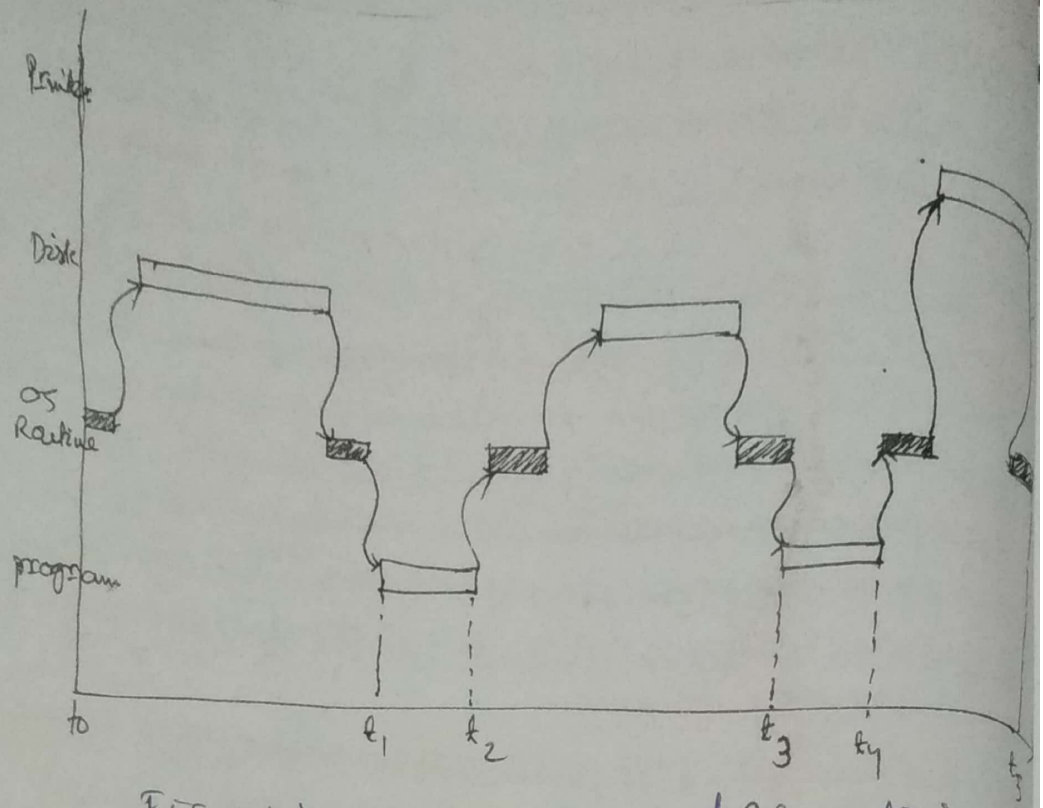


Fig: User program and OS routine sharing of the processor

To understand basic of OS, consider the system with one processor one disk & one printer.

→ Running one application program.

→ Assume that the appl.ⁿ program has been compiled from a high level language into a machine lang. form & stored on the disk.

→ First step to transfer this file into memory

→ When the transfer is complete, execution of prog. is started

→ When the datafile is needed for execution of a prog., the prog. request the OS to transfer the datafile from the disk to the memory.

→ The OS perform this task & passes execution control back to the application program which then proceed comp.

→ After completed of computation the result is ready to be printed.

→ The appl.ⁿ prog again send a request to the OS

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

Explanation :-

Performance

The way, the performance of a computer can be measured is how quickly it can executes programs.

→ The speed with which a computer executes programs is affected by the design of hardware and its machine language instructions. Though the programs are written in high level language, performance is also highly affected by the compiler that translates prog^m into machine level language.

→ For best performance, it is necessary to design a compiler, the machine instruction set, and the hardware in a coordinate way.

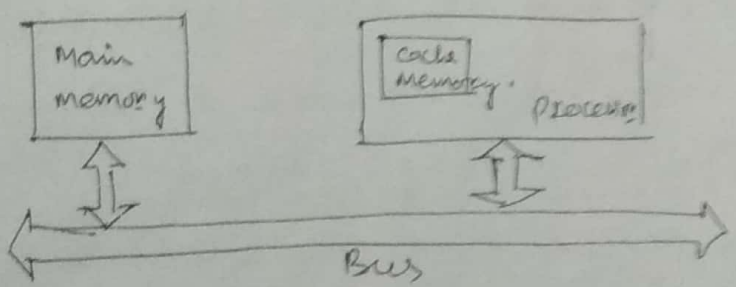
(Here we discuss only about s/w and instruction set)

→ One of the main thing which affect the performance of the system is the processing speed (processor). Here we will discuss some key parameters which affects the processor time.

→ Cache memory.

The processing time depends on the hardware involved in the execution of each individual instructions. The hardwares are processor and the memory, which are connected by bus.

Here a new memory is included inside the processor to increase the performance, is called the cache memory.



Let us consider the flow of instruction and data inbetween memory and processor. At the start of

execution, all the program instructions and data are stored inside the main memory. At the time of execution instruction are fetched one by one over the bus into the processor, and a copy is placed in the cache, same for the data. When some instruction or data needed for the second time, it is read directly from the cache.

The processor and the cache are fabricated on a single chip, so the speed in which the processor can communicate with the cache is very fast and also considerably very ~~large~~ faster than the speed in which CPU can communicate with the main memory. A program execution can be faster if the communication between the processor and the main memory can be reduced, which is possible by using cache.

① Processor Clock.

Processor circuits are controlled by timing signals called as clock. The clock defines regular time intervals, called clock cycle. ②

→ To perform a machine instruction, the processor devides the m/c instruction in to sequence of some steps, and each step can be completed in one clock cycle.

→ The length P of one clock cycle is an important parameter that affects processor performance.

→ The inverse is the clock rate, $R = 1/P$, which is measured in cycles per second.

→ Today PC having clock rate few hundred million to over a billion cycles per second.

→ cycles per second called as hertz (Hz).

→ million denoted by mega (M) so MHz
→ billion " " Giga (G) so GHz

So 500 million cycles per second = 500 MHz. 1 MHz
& 1250 million cycles per second = 1.25 GHz. 1 GHz

So corresponding clock period are 2 or 0.8 nanosec.

② Basic Performance Equations.

Let (T) be the processor time required to execute a programme.

After compilation lets assume that the complete execution of the programme requires the execution of (N) machine language instructions.

Suppose the average no of basic steps needed to execute one machine instructions is (S), where each basic step is completed in one clock cycle.

The clock rate is (R) cycles per second,

The program execution time is $T = \frac{N \times S}{R}$

(This is called Basic Performance Equation)

The performance parameter T is much important for an user, than that of N, S & R. So the user always want to reduce the value of T, which means reducing the value of N and S and increasing the value of R.

→ N reduces if source prog^m compiled into few no of m/c codes.
→ S reduces if instruction has a small no of basic steps to perform.
→ R increases using higher frequency clock.

③ Pipelining and Superscalar operations.

Till now we discussed that instructions are executed one after another, so the value of S is the total no of basic steps, or clock cycles required to execute an instruction.

A substantial improvement in performance can be achieved by overlapping the execution of the successive instructions, using a technique called pipelining.

Let us consider the instruction Add R_1, R_2, R_3 ⇒ which adds the content of R_1 & R_2 & place the sum into R_3 (content of R_1, R_2 fed to ALU for the add operation performed, the sum fed to R_3)

(ideal case) After overlapping all instructions to the maximum degree possible, execution proceeds at the rate of one instruction complete in each clock cycle. Though individual instruction takes several clock cycle to complete, but for the purpose of Computing T, the effective value of S = 1.

Clock Rate

Clock rate is cycle per second

2 ways to increase clock rate R,
First, is to improve the Ic tech. to make Logic circuit faster.
Second, Reducing the no. of basic steps executed in P

But increases in the value of R entirely caused by improvements in Ic tech. It again affects the reading of data from main memory. Again that can be handled by using cache.

Complex inst" set computers · Reduced inst" set Computers

(5) Instruction Set CISC & RISC

(CISC)	(RISC)
1. Complexity found in hardware.	1. Complexity on Software Side.
2. Load & Store (memory to memory) functionality found in a single instruction.	2. Register-to-register: Load & store are separate instructions.
3. Less line of code needed to provide same functionality.	3. More instructions necessary to provide same functionality.
4. Instruction are not always the same size.	4. All instructions are of uniform size.
5. Instruction are difficult to decode because instructions are not uniform.	5. Instruction are easier to decode because of how they are set up: Opcode will always be in the same place.
6. Difficult to use pipelining, because instruction needs to be broken down to smaller components at process level.	6. Capable of using pipelining by design efficiently.
* Intel x86 has CISC Arch.	7. IBM power PC use RISC Arch. (Games, Cell phones, Embedded chips for car etc)

Compiler. A compiler translates a high level language program into a sequence of instruction. The compiler & the processor are designed at the same time to achieve best results. An optimizing compiler takes advantages of various feature to reduce the product NXS i.e to reduce clock cycle.

Performance Measurement

→ PM is important to assess the performance of a computer. Computer designers use performance estimate to evaluate the effectiveness of new features. The performance measure is the time takes a comp. to execute a given benchmark. 1st attempt to make AI as standard benchmark for general purpose comp. a suite of Benchmark prog. was selected in 1989, was modified in 95 then in 2000.

(Benchmark is the act of running a comp prog or a set of prog or other operation.
Benchmark are provide a method of comparing performance of various sub systems.)

Performance Measurement

(SPEC) rating

System Performance Evaluation Corporation (SPEC) Ex:- SPEC rating so

Non profit organization

the comp. under test is ____ running time on the reference computer

means → 50 time is running time on the computer as ULtra SPARC 10. for this particular bench mark. under test.

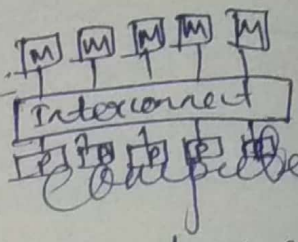
for programs in the suite. Let $SPEC_i$ be the

The overall SPEC rating $= \left(\prod_{i=1}^{n} SPEC_i\right)^{1/n}$ where n is the no of prog. in the suit

SPEC rating is a measure of combined effect of all factor effecting performance including the compiler, the OS, the processor & the memory

computer organization
Architecture

Multiprocessor & Multicomputer



P-18

Multiprocessor :- Large computer system may contain a number of processer units. So they are called multiprocessor System.

→ These systems either execute a number of application tasks in parallel
or → They execute subtasks of a single large task in parallel.

→ All the processor usually access all of the memory. In such systems. So called shared memory - multiprocessor system.

Adv → high performance
disadv. → Increased complexity & cost (due to more complex inter connection system)

Multicomputers

An interconnected group of complete computers known as multicomputers. which achieve high total computation power.

→ Access their own memory units.
→ Tasks are executed need to communicate data.
→ They exchanging messages over a communication network, so called message-passing multicomputers.

History
1st, second, third, fourth
They key concept of a stored program was introduced by John Von Neumann.

Chapter-2
Carl Hamacher, Zaky.

Machine Instructions & Programs

mod 2

Introduction.

Computers are built using logic circuit that operate on information represented by two valued electrical signal 0 & 1. We define that signal as bit of info where bit stands for binary digit.

Representation of a no. in computer system by string of bits called Binary no. is

A text character can also be represented by a string of bits called a character code.

Number Representation

Consider n-vector $B = b_{n-1} \cdots b_1 b_0$ where $b = 0$ or 1 for $0 \le i \le n$

This vector can represent unsigned integer values V in the range 0 to $2^n - 1$ where

$$V(B) = b_{n-1} \times 2^{n-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

we obviously need to represent both +ve & -ve nos.

Three systems are used for represent such system
- sign & magnitude
- 1's complement
- 2's complement ⊕ ...

Memory Location & Addresses

Main memory of a computer consists of million of storage cells, in each cell we can store a bit of information having value 0 or 1.

→ Because a bit represent a very small amount, we don't handle a bit individually.

→ Normally we are dealing with them in groups of fixed size and for this the main memory is organised so that a group of n ~~digits~~ bits can be stored or retrieved in a single operation.

→ That group of n bits can be called as a word of information and 'n' is called as the word length (ranges from 16 to 64 bits)

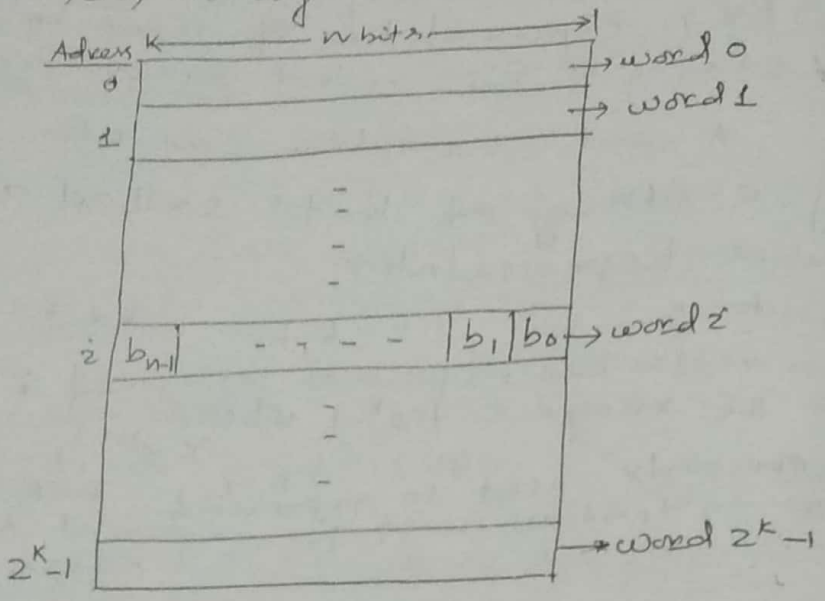
→ To access the main memory we need distinct address for each word location.

→ So we can use adress 0 to $2^K - 1$, where K is the bit carried by the adress bus.

A word is 2 bytes (16 bit), a doubled word is 4 bytes (32 bit), and a quad word is 8 bytes (64 bits)

→ 2^K adress constitutes the adress space of the computer and the main memory of the computer can have upto 2^K words.

Example A 24 bit adress generates an adress space of 2^{24} (16,777,216) memory locations.

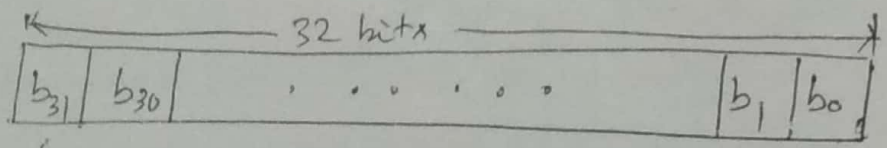


main memory adresses

The contents of memory locations can either aginstruction or an operand, again operand may be either number or characters.

Number.

Suppose the word length is 32. So the figure bellow shows how a number is represented having 32 bits word length.



↳ sign bit $b_{31}=0$, for +ve number.

$b_{31}=1$, for -ve number.

magnitude = $b_{30}\times 2^{30} + \cdots + b_1\times 2^1 + b_0\times 2^0$

→ The magnitude that can be represented this way range from 0 to $2^{31}-1$, and the numbers are said to be in binary positional notation.

→ The encoding format is called sign-and-magnitude representation.

↳ another two other representation are also used called 1's complement & 2's complement representation.

↳ In both these schemes, the repr^n of +ve number is same as sign magnitude repr^n, but differs in the way in which -ve numbers are represented.

↳ but the sign bit (b_{31}) is same for all the three schemes.

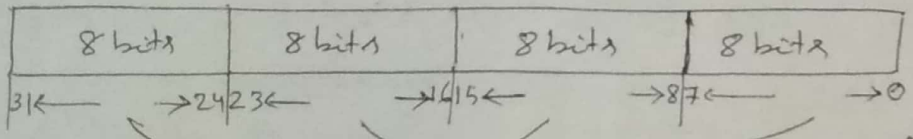
→ The 2's complement repr^n is the most suitable one.

Characters.

Character can be letters or the alphabet, decimal digit, punctuation mark, all other symbols, and so on..

→ They are represented by codes that are usualy 6 to 8 bit long.

→ Bill now we saw how 4 characters in the ASCII code can be stored in a 32 bit word.



In an ASCII file each alphabetic, numeric or special character is represented with a 7-bit binary no. (a string of 7 bits can represent 128 possible characters are defined)

ASCII CHARACTERS → is the most common format for text files in most computers & on the Internet

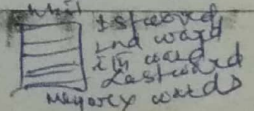
(American Standard Code for Information Interchange)

Appl^s → UNIX & DOS based os use ASCII for text files

ASCII was developed by the American National Standard Institute (ANSI)

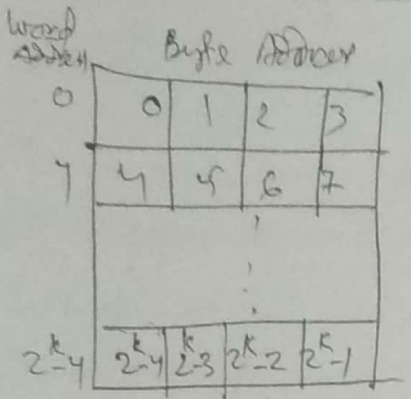
BYTE ADDRESSABILITY

A Byte is always 8 bits, but the word ranges typically between 16 to 64 bits. It is difficult to assign distinct addresses to individual bit locations in the memory. The most practical assignment is to have successive addresses refer to successive bytes.

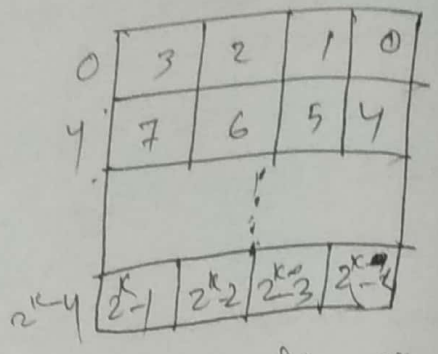


There are two ways that byte addresses can be assigned across words. The Big-endian used where lower ~~Instructions~~ byte addresses are used for more significant byte of the word. The Little-endian where the lower is used for opposite ordering

BIG-ENDIAN & LITTLE-ENDIAN ASSIGNMENTS.



Big-endian assignment.



Little-endian assignment.

byte addresses are used for less significant bytes of the word. The word "more significant & less significant are used in relation to the weights (power of 2) assigned to bits when the word represents a numerical data

Appl.ⁿ - Commercial M/C → For encoding characters with a byte
→ Labelling bits from left to right $b_7, b_6 \ldots b_0$

Word Alignment

of the word lengths

For an instance if is 32 bit, then the natural word boundaries occure at addresses 0, 4, 8 .....

If it is the case then we say the word locations have aligned addresses.

And it is not mandatory to begin the words ~~at aligned addr~~ as done in the previous case. So if the words begin at an arbitrary byte address, then we say the words have unaligned addresses.

Memory Operations -

2 Basic operations -
(i) LOAD (or Read or Fetch)
(ii) STORE (or Write)

Load op^n transfer a copy of the contents of a specific memory location to the processor.

store op^n transfers an item of information from the processor to a specific memory locations.

→ Tasks are a sequence of small steps such as adding two no.s, testing for a particular cond^n or reading char. from keyboard or sending a char to display

Instruction & Instruction Sequence

4 types of operations performed by instructions of a computer system

↳ Data transfer between the memory and processor register.

↳ Arithmatic & Logical operation on data

↳ Program sequence & control.

↳ I/O transfers.

To discuss the different types of instructions we need to know some notations used to represent the instructions.

Register Transfer Notation - To describe the transfer of data from one location to another in a system we use Register Transfer Notation.

e.g. $R1 \leftarrow [LOC]$ or $R3 \leftarrow [R1] + [R2]$

'[ ]' is used to denote the content.

Most of the comp. have 3 type of C(x) organization
→ Single accumulator organization
→ General register organization
→ Stack organization

Assembly language Notation.

The way we represent m/c language instruction
→ Assembly language

Example

```
MOVE  LOC, R1
ADD   R1, R2, R3
```

Three Instruction Type.

① Three-Address Instruction

Suppose we want to perform C ← [A]+[B]

```
Add A, B, C
```

(Source operand, Source operand, Destination operand)

operation source1, source2, destination.

$X = (A+B) * (C+D)$

```
ADD A,B,R1
ADD C,D,R2
MUL R1,R2,X
```

② Two address Instruction

```
MOV A,R1
ADD B,R1
MOV C,R2
ADD D,R2
MUL R1,R2
MOV R2,X
```

operation source, destination

Add A, B ⇒ Move B, C // C ← [B]

which performs Add A, C // C ← [A]+[B]

[B] ← [A]+[B]

$X = (A+B) * (C+D)$

R1 ← M[A]
R1 ← [R1]+M[B]
R2 ← M[C]
R2 ← [R2]+M[D]
R2 ← [R1]*[R2]
X ← [R2]

③ One address Instruction

when one operand is needed

m/c inst" that specify 1 operand & needed a unique location which is already defined used by default, e.g. called as Accumulator

Add A1. (A1 with accumulator is stored in Accu)

Store A1 → (copy accumulator content to memory loc")

Load A1 → (copy content of memory loc" A1 to Accu")

C ← [A]+[B]. →
LOAD A (A is source)
ADD B
Store C (C is destination)

Load A
Add B
Store C
..
two numbers

ADD of

Zero address Instruction.

It is possible to use instructions in which the locations of all operands are defined implicitly.

→ Such instructions are found in machine that store operands in a structure called a pushdown stack. In this case the instructions are called zero-address instruction. Because the ADD, MUL, etc. instructions doesn't need an address field to specify.

Example Suppose we have to solve the following exp —

$(a+b) * (c+d)$

→ its postfix expression is $ab+cd+*$

The process of evaluating this expression is to use a stack.

Zero address Instruction

Instruction	Stack
Push a	Top ← a
Push b	Top ← b
Add	Top ← (a+b) (pop top two elements and add)
Push c	Top ← c
Push d	Top ← d
Add	Top ← (c+d)
Mul	Top ← (c+d) * (a+b) (result)
POP X	X ← Top

There are zero address instruction.

One address Instruction

X = (A+B) * (C+D)

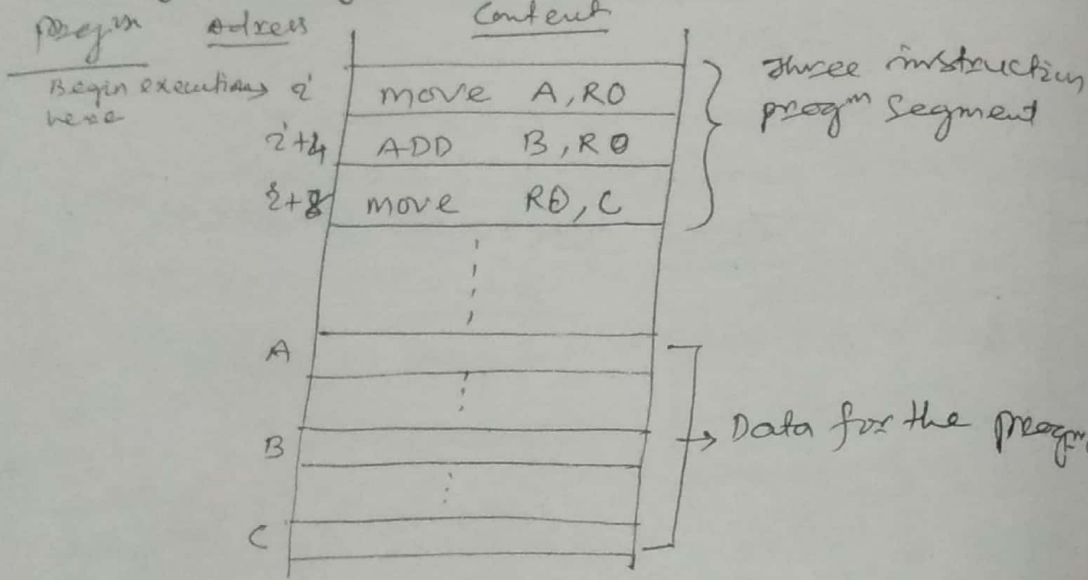
LOAD A	AC ← M[A]
ADD B	AC ← [AC] + M[B]
STORE T	M[T] ← [AC]
LOAD C	AC ← M[C]
ADD D	AC ← [AC] + M[D]
MUL T	AC ← [AC] * M[T]
STORE X	X ← [AC]

Instruction Execution

Straight-line Sequencing

Suppose we have to perform an addition operation ($C \leftarrow [A] + [B]$) in the main processor.

Suppose the formal of the writing of the program is two address instruction, and we have general purpose registers in the CPU. we also assume the word length is 32 bit and word addresses are aligned.



Suppose the three instructions of the program is placed in successive memory locations, whose address value increases according to the order in which instruction are to be performed, starting at locat

→ Lets discuss the execution of the program.

↳ CPU Contains a register called the PC, which hold the address of the instruction to be executed next.

→ In the begining of execution the ~~first~~ address of

the first instruction must be their inside PC.

→ Then CPU control circuit use the information in the PC to fetch instruction & then execute that same instruction, one at a time, in the order of increasing address.

This is called straight-line sequencing.

→ At the time of executing each instruction the PC is advanced to point to the next instruction.

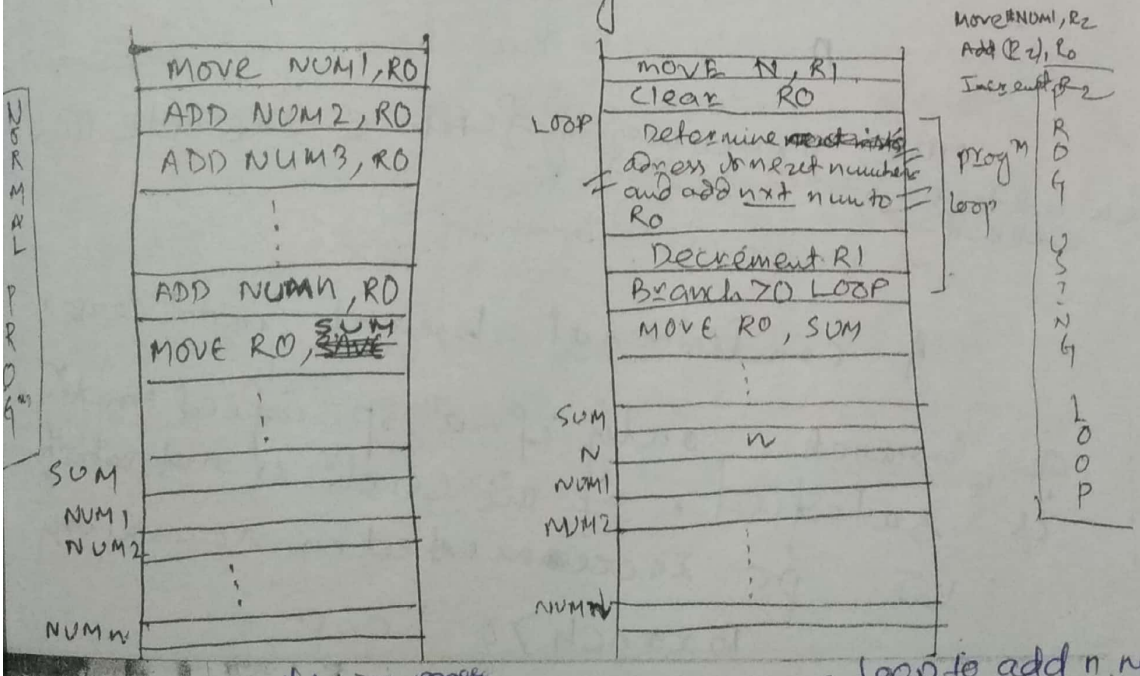
Branching

Suppose we have to add a list of n-numbers.

↳ Let the address of the n numbers in the memory location are symbolically represented as NUM1, NUM2, NUM3 . . . NUMn.

↳ And a seperate Add instruction is used to add each number to the content of register R0.

↳ After all the instruction is being added the result is placed in memory location SUM.



A - Straight-Line prog for adding n-no.

Using a Loop to add n, nos.

→ Instead of using a long list of add instructions, it is possible to put a single add instⁿ in a program LOOP.

→ This LOOP causes a straight line sequence of instⁿ to be executed repeatedly, as many times as needed.

→ The LOOP starts at location LOOP and ends at the instruction Branch >0.

→ During each pass through the loop the address of next line entry is decremented, and the entry is fetched and added to R0.

→ Register R1 used as a counter to decrement the number of times the loop is executed. (So 'n' is loaded to R1 in the beginning of the progᵐ).

→ The execution of the loop must be repeated as long as the result of the decrement operation is greater than zero.

→ Lets discuss what a Branch Instruction is ???

↳ This is an instruction that load a new value to the Progᵐ Counter, The processor fetch & execute this new address called → branch target ??.

→ conditional branch ?!

A conditional branch instⁿ causes a branch only if a specified condⁿ is satisfied. If the condⁿ is not satisfied the PC is incremented in a normal way.

Branch >0 LOOP

Branch Target Buffer -

A Branch target predictor is the part of a processor that predicts the target of taken conditional branch or an unconditional branch instr. before the target of branch instr. is computed by the execution unit of the processor.

Conditional branch - It is a programming instruction that directs the computer to another part of the prog. based on the results of a compare. High-Level language statements, such as IF-THEN-ELSE & CASE, are used to express the compare & conditional branch.

Condition Codes (or Status Register)

The processor keeps track of some information about the result of various operations for use by subsequent conditional branch instruction. This is accomplished by recording the required information into individual bits, often called the condition code flags.

→ In some processor the flags are grouped together in a special register called the condition code or status register.

→ Alternatively, an instruction that set the condition flag, may specify that one of the general purpose registers of the m/c should be used to record the relevant flag.

→ In either case, individual condition code flags are set to 1 or clear 0, depending upon the outcome of the operation performed.

Four common FLAGS

N (Negative) → Set to 1 if result is -ve, otherwise clear to 0.

Z (Zero) → Set to 1 if the result is 0, otherwise clear to 0.

V (overflow) → Set to 1 if the arithmatic overflow occurs, otherwise clear to 0.

C (Carry) → Set to 1, if a carry out results from the operation, otherwise clear to zero.

Addressing Mode

These are the rules for interpreting the address field of instr^{n}.

10 add. Mode

1) Implicit mode / implied Add. mode

operands are specified implicitly

CMA eg: Complement Accumulator.

Means the accumulator is containing the operand. It specify the instr^{n} itself. that's why it is implicit mode.

2) Immediate Mode

Actual operand in any instr^{n} & another field is address. Here is opcode But in,

[opcode | Address]

~~Immediate mode operand field~~

Immediate Add. mode inplace of Address there is operand field

[opcode | operand]

Acc^{n} to this mode the operand is specified the instr^{n} itself

3) Register Mode (Register Direct mode)

In this mode operands are stored in register & the register will hold the value of the operands.

Direct mode means the register contain the operand directly.

4) Register indirect Mode

Register contain the address which can referred to effective address

(that address which contain the operand)

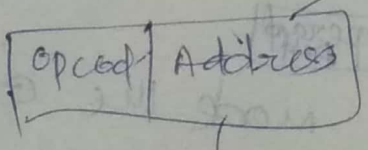
Hence it is reg. indirect mode.

(5) Auto increment & Autodecrement mode

says that the value of the reg. increment by one but after the exec. of instr.

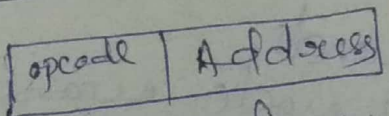
But incase of Auto decrement mode the value of the reg. is decremented by one before the exec. of the instr.,

6) Direct addressing Mode



instr. format contain the address which directly reffer to the operand

7) Indirect Address Mode



the instr. format address

is not the effective address but it will point to another address which can be an add. or effective address refer to the operand

(8) Relative add. Mode

In this case the value of PC is added to the address field value of instr" which will give you the effective address which referred to the operand.

value of PC + Add. → effective add.

9) Indexed add. Mode.

In this case the value of Index Reg. is added to the address field & give the effective address.

Index reg + Add field → effective add.

10) Base Addressmode :-

The value of Base reg is added to the Add. field which give the effective address & effective add. point to operand.

Base add + Address field → effective add.

Data Representation

Binary information in digital comp. is stored in memory or processor registers.

→ Contain either data or control info.

The d-type found in the registers of digital comp. are classified as — (1) numbers used in arithmetic computation.

(2) letters of the alphabet used in data processing &

(3) other discrete symbols used for specific purposes.

All types of data except binary no., are represent in comp. registers in binary coded form. This is because registers are made up of flip flops & flip flops are two-state devices that can store only 1's & 0's.

Number System

A no. system of base or radix r is a system that uses distinct symbol for r digits. It is necessary to multiply each digit by an integer power of r & then formed sum of all weighted digit.

radix Decimal means radix 10

The 10 general no. symbols are 0 ---- 9.

decimal The string of digit 724.5

$\rightarrow 7\times10^2+2\times10^1+4\times10^0+5\times10^{-1}$

simply $(23)_{10} = 2\times10^1+3\times10^0$

binary The string of digit 101101

$1\times2^5+0\times2^4+1\times2^3+1\times2^2+0\times2^1+1\times2^0$

$= 45$

$(101101)_2 = (45)_{10}$

octal The eight symbol of the octal system are 0 — 7.

hexa decimal & The 16 symbols of the hexadecimal system are 0, 1 - - - 9, A, B, C, D, E & F.

A, B, C, D, E, F correspond to the decimal no. 10, 11, 12, 13, 14, 15 resp.

octal to decimal

$(736.4)_8 = 7\times8^2+3\times8^1+6\times8^0+4\times8^{-1}$

$= 7\times64+3\times8+6\times1+4/8$

$= (478.5)_{10}$

The equivalent decimal no. of hexadecimal F_3 is obtained from the following calcul.

$$(F3)_{16} = F \times 16^1 + 3 \times 16^0$$
$$= 15 \times 16 + 3$$
$$= (243)_{10}$$

Octal & Hexadecimal No.

Conversion from & to binary
octal & hexadecimal representation

Since $8^3 = 8$ & $2^4 = 16$

So each octal & hexadecimal corresponds to binary

Conversion of decimal 41.6875 into binary :-

Integer = 41

41	
20	1
10	0
5	0
2	1
1	0
0	1

$(41)_{10} = (101001)_2$

Fraction = 6875

0.6875	
	2
1.3750	
	2
0.7500	
	2
1.5000	
	2
1.0000	

$(0.6875)_{10} = (0.1011)_2$

2 | 41	
2 | 20	1
2 | 10	0
2 | 5	0
2 | 2	1
2 | 1	0
	1

$(41.6875)_{10} = (101001.1011)_2$

1 2 7 5 4 3 → Octal

1 010 111 . 011 000 11 → Binary

A F 6 3 → Hexadecimal

Assignment $(ABC)_{16} = 2749 = 16^2 \times A + 16^1 \times B + 16^0 \times C = 256 \times 10 + 16 \times 11 + 1 \times 12$
$= 2560 + 176 + 12$

Binary coded octal no.

Octal no	Binary-coded octal	Decimal equivalent
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7

code for one octal digit

10	001 000	8
11	001 001	9
12	001 010	10
24	010 100	20
62	110 010	50
143	001 100 011	99
370	011 111 000	248

Binary coded hexadecimal no.

Hexa decimal no.	Binary coded hexadecimal	Decimal equivalent
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15
14	0001 0100	20
32	0011 0010	50
63	0110 0011	99
F8	1111 1000	248

code for one hexadecimal digit

Binary-coded Decimal (BCD) Num

Decimal number	Binary-coded decimal (BCD) number
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Code for one decimal digit

10	0001 0000
20	0010 0000
50	0101 0000
99	1001 1001
248	0010 0100 1000

Few decimal number and their representat in BCD

Alphanumeric Representation

Many applications of digital computers require the handling of data that consist not only of numbers, but also of the letters of the alphabet & certain special characters.

The standard alphanumeric binary code is the ASCII (American Standard Code for Information Interchange)

(American Standard Code for Information Interchange) (ASCII)

Character	7-bit ASCII Binary code
A	100 0001
B	100 0010
C	100 0011
D	100 0100
E	100 0101
F	100 0110
G	100 0111
H	100 1000
I	100 1001
J	100 1010
K	100 1011
L	100 1100
M	100 1101

Character	Binary Code
41 0	011 0000
42 1	011 0001
43 2	011 0010
44 3	

Complement

(r−1)'s complement

9's complement

9's comp. of no = N defined as $(r^n - 1) - N$

9's comp. of $r - 1 = 9 = r = 10$

$10^n - 1 - N$

~~10^n represent as~~

$10^n - 1$ is a no represented by n 9's

Let n = 4

$10^4 - 1 = 9999$

$10^4 = 10000$

1) 9's compl. of 546700

999999 − 546700

= 453299

2) 9's comp. of 12389

is 99999 − 12389 = 87610

1's comp.

r = 2

r − 1 = 1

1's compl. of N is

$(2^n - 1) - N$

n = 4, $2^4 = (10000)_2$

& $2^4 - 1 = (1111)_2$

0001
1110
+ 1

r's complement

(r's) of n-digit no. N in base r is defined as $r^n - N$ for $N \neq 0$ and 0 for $N = 0$

$$r^n - N = [(r^n - 1) - N] + 1$$

2's comp. of binary

10 1100 is 01001 + 1

= 01010 0

10's comp. means find 9's complement then add 1.

Fixed point Representation

→ 3 parts
→ sign field
→ Integer field
→ fractional field

Let us take an example of binary no. with radix pt.

① 001 ⑩ 101
sign int. fraction

Positive integer including 0 can be represented as unsigned 32 bit format numbers. In ordinary arithmetic for -ve no we use — (minus sign) & for +ve no we use + (plus sign)

32 bit: ① sign bit, ⑮ int, ⑯ fraction

Binary fixed pt representation: $x_n x_{n-1} \ldots x_2 x_1 x_0 . x_{-1} x_{-2} \ldots x_{-m}$

sign bit: 0 for +ve, 1 for -ve

Because of h/w limitations, computer must represent everything with 1's and 0's including the sign with a bit placed in the leftmost position of the no. The convention is to make the sign bit equal to 0 for +ve & to 1 for -ve.

There are two ways of specifying the position of the binary point in a register: by giving a fixed position or by employing a floating pt. representation.

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

The two pos^{r} most widely used 1) a binary pt. in the extreme Left of the register to make the stored no. a fraction.

(2) a binary point in the extreme right of the register to make the stored no. an integer.

Integer Representation → unsigned
→ signed

Inter Representation

signed

1. Signed-magnitude representa
2. Signed 1's complement))
3. Signed 2's))

only one way to represent (+0)
+14 = 00001110
three ways

(-14) → 9n

1. — 1 000 1110
2. — 1 111 0001
3. — 1 111 0010

Sign bit of 1 ← in the Left most pos^{r}

+6 00000110
+13 00001101
+19 00010011

-6	11111010	
+13	00001101	
+7	00000111	

+6	00000110	
-13	11110011	
-7	11111001	

-6	11111010	
-13	11110011	
-19	11101101	

(+375) + (-240) = +135

→ Decimal fixed pt representation

The representation of decimal no. in reg. The representation of decimal code used to represent is a func^n of the binary code represent a decimal digit. A 4-bit decimal code represent four flip-flops for each decimal digit.

Ex: The representation of 4385 in BCD required 16 flip-flops, four flip-flop for each digit.

The above no. will be represented in a register with 16 flip-flop as follows

0100 0011 1000 0101.

Sing: The representation of signed decimal no. in BCD is similar to the representation of signed numbers in binary. We can either use familiar sign magnitude or signed complement system.

The signed complement are 9's & 10's ...

Ex: 9's is familiar. After finding 9's compl. we add 1 to it & find 10's comp.

Floating point Representation

principle - It is possible to represent a +ve or -ve integer centred on 0 with a fixed-pt notation by assuming a fixed binary or ~~radix~~ point.

Limitation of fixed point Representation

→ Very large no. can not be represented nor can very small fractions

→ The fractional part of quotient is a division of two large no. could be lost.

→ The floating point representation of a no has two parts

(i) a signed fixed point no (Mantisa)

(ii) decimal or binary point (exponent)

→ For decimal no. one can gets by using scientific notation.

Thus, 976,000,000,000,000 can be represented as $9.76 \times 10^{+14}$ &

0.000 000 000 976 can be represented as 9.76×10^{-14}

This approach can be taken with binary numbers. We can represent a number in the form $\pm S \times 13^{\pm E}$ (Base)

floating point is in the form

This no. can be stored in a binary word with three fields

- Sign : Plus or minus
- Significand S
- Exponent E

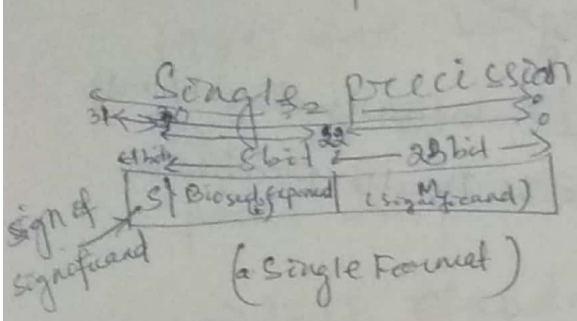
Left most bit stores the sign.

The sign may be either +ve or -ve.

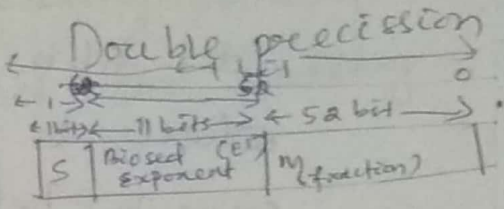
0 is represent as +ve & 1 is for -ve.

(i.e 0 → +ve & 1 → -ve)

* IEEE 754 standard defines both 32-bit single & 64 bit double precession format with 8 bit & 11 bit exponent respectively. The implicit base is 2.



(a. Single Format)



(b. Double format)

$E' = E + bias$

Bias is a no. added in both single & double prec. the bias added is 1023.
In single prec. the bias added is 127 & in double prec.
→ The representation used in floating point representation known as a biosed representation

→ Floating point represent in many ways
Typically the bias equals $(2^{k-1}-1)$ where k is the no. of bits. In this case the 8-bit field yields the nos 0 through 255.

$(2^{8-1}-1) = 2^{7-1} = 128-1 = 127$

The following are equivalent where the significand is expressed in binary form

0.110×2^5
110×2^2
0.0110×2^6

To simplify operations on floating point nos it is typically required that they be normalized.

→ A normalized no. is one in which the most significand digit of the significand is non zero.

Ex- base 2 representation

$$\pm 1.bbb\cdots b \times 2^{\pm E}$$

where b → either binary digit (0 or 1)

solve by both single & Double precision

Ex $(1460.125)_{10}$

$= (10110110100.001)_2$

↓ Normalize

After normalization it will be

$1.0110110100001 \times 2^{10}$ ← exponent part

Single precision

As +ve no. S = 0

E = 10

M = 01101101000001

Mantisa

Now $E' = 10 + 127 = (137)_{10}$

convert 137 into binary again

$(137)_{10} = 10001001$

0	10001001	01101101000001—0

as Mantisa not 23-bit add 0 after 'dot'

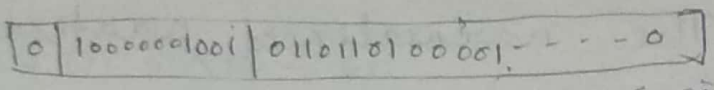
Double Precision

S = 0

E = 10

$E' = 10 + 1023 = 1033$

(convert into binary) = (10000001001)



(Institute of Electrical & Electronics Engineers)

IEEE Standard 754

IEEE Standard 754 adopted in 1985 for binary floating point representation. This is the standard for floating point representation.

→ IEEE standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PCs, MACs, & most unix platforms.

There are several ways to represent floating point number but IEEE 754 is the most efficient in most cases. IEEE 754 has 3 basic components.

1- The sign of Mantisa -
This is simple as the name. 0 represent a positive number while 1 represent a negative number.

2. The Biased exponent -
The exponent field need to represent both positive & negative exponent. A Bias is added to the actual exponent in order to get the stored exponent.

3. The normalized mantissa —

The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits. Here we have only 2 digits, i.e. 0 and 1. So a normalized mantissa is one with only one 1 to the left of the decimal.

IEEE 754 number are divided into two based on the above three components.

→ Single precision & Double precision

Floating-Pt Numbers & Arithmatic op.

$$X = X_S \times B^{X_E}$$

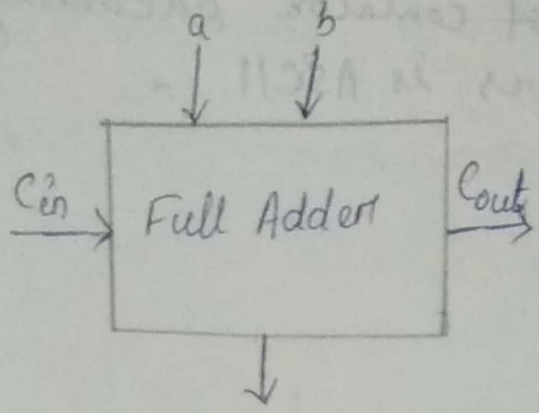
$$Y = Y_S \times B^{Y_E}$$

$$X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E} \quad \left.\begin{matrix} \\ \\ \end{matrix}\right\} X_E \leq Y_E$$

$$X - Y = (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E}$$

$$X * Y = (X_S * Y_S) \times B^{X_E + Y_E}$$

$$\frac{X}{Y} = \frac{X_S}{Y_S} \times B^{X_E - Y_E}$$

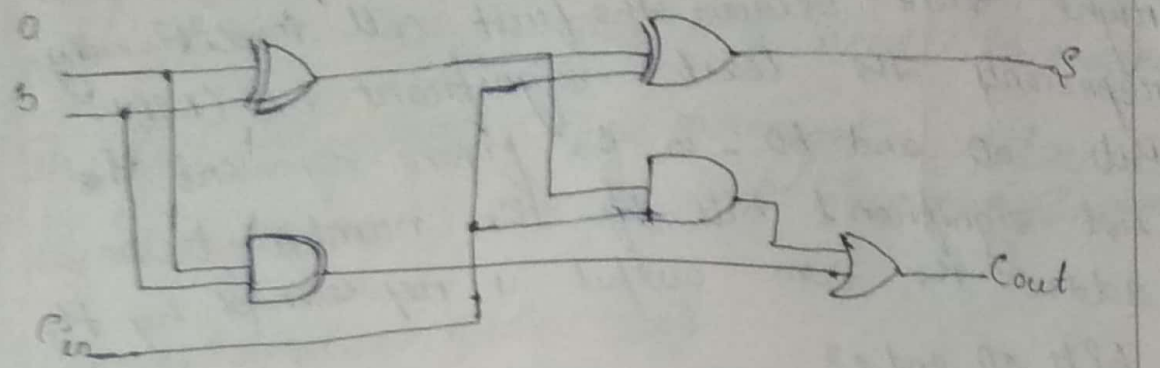


A one bit full adder is a combitional circuit that forms the arithmetic sum of three bits. It consists of three inputs (a, b and C_{in}) and two outputs (S and C_{out}) — as illustrated in figure 1.

Table 1: Full adder truth table

a	b	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The truth table of 1-bit full adder is given in the table



The gate implementation of 1-bit full adder is given in the figure

Example:

$$\begin{array}{rrr} x & 7 & 0\ 1\ 1\ 1 \\ +y & =+6 & =+.0_1\ 1_1\ 1_0\ 0_0\ . \\ \hline z & 13 & 1\ 1\ 0\ 1 \end{array}$$

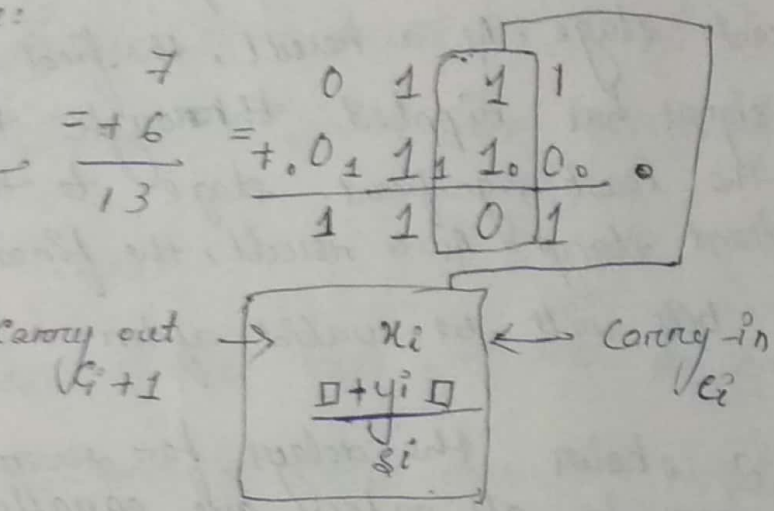


Figure 9.1 Logic specification for a stage of binary addition

Ripple Carry adder

A ripple carry adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascaded with the carry output from each full adder (FA) ~~circuit to provide a 4 bit~~ connected to the carry input of next full adder in the chain. Figure 3 shows the interconnection of four full adder (FA) circuits to

provide a 4-bit ripple carry adder. Notice from Figure 3 that the input is from the right side because the first cell traditionally represents the least significant bit (LSB). Bits a0 and b0 in the figure represent the least significant bits of the numbers to be added. The sum output is represented by the bits s0 and s3.

Ripple carry adder delays

In the ripple carry adder, the output is known after the carry generated by the previous stage is produced. Thus the sum of the most significant stage. As a result, the final sum and carry signal has rippled through the adder from the least significant stage to the most significant stage. As a result, the final sum and carry bits will be valid after a considerable delay.

Table 2 shows the delays for several CMOS gates assuming all gates are equally loaded for simplicity. All delays are normalized normalized relative to the delay of a single inverter. The table also shows the corresponding gate areas normalized to a relative simple minimum-area inverter. The table also shows the corresponding gate areas normalized to a Note from the table that multiple-input gates have to use a different circuit technique compared to simple 2-input gates.

For an n-bit ripple carry adder the sum and carry bits of the most significant bit (MSB) are obtained after a normalized delay of,

Sum $_{n-1}$ delay = $4n+2$ ①

Carry$_n$ delay = $4n+3$ ②

For a 32-bit procession, the carry chain normalized delay would be 131. The ripple carry adder can get slow when many bits need to be added. In fact the carry chain propagation delay is the determining factor in most microprocessor speeds.

Carry Lookahead Adder (CLA)

The Carry lookahead adder (CLA) solves the carry delay problem by calculating the carry signals in advance, based on the input signals. It is based on the fact that a carry signal will be generated in two cases

(1) When both bits a_i and b_i are 1, or

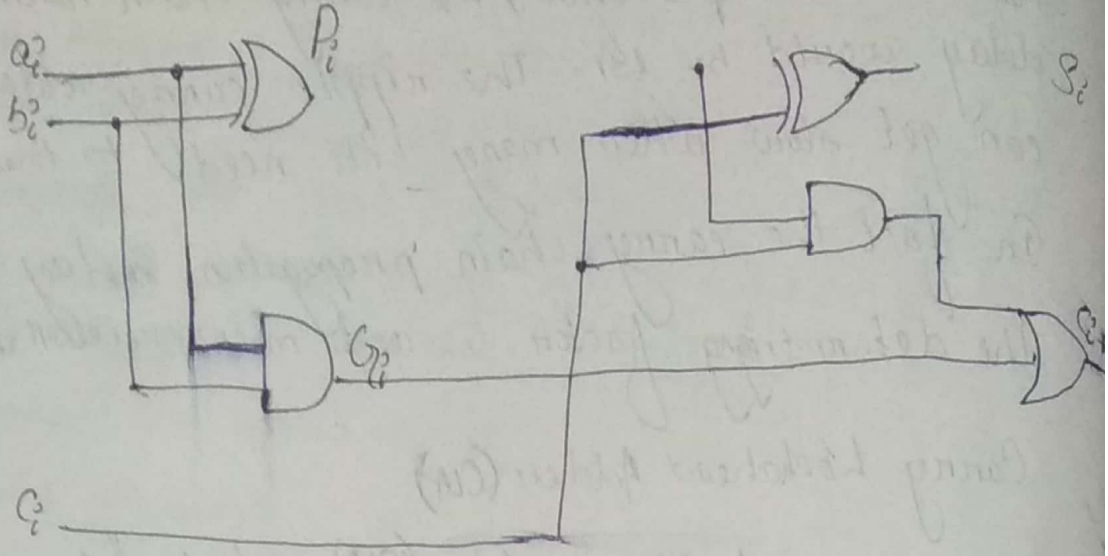
(2) When one of the two bits is 1 and the carry in is 1.

Thus we can write,

$$c_{i+1} = a_i \cdot b_i + (a_i \oplus b_i) \cdot c_i \quad (3)$$

$$s_i = (a_i \oplus b_i) \oplus c_i \quad (4)$$

The above two equations can be written in terms of two new signals P_i and G_i, which are shown in figure 4



Where P_i and G_i are called the carry generate and carry propagate terms respectively. Notice that the generate and propagate terms only depend on the input bits and they will be valid after one and two gate delay, respectively. If one uses the above expression to calculate the carry signals one does not need to wait for the carry to ripple through all the previous stages to find its proper value. Let's apply this to a-bit adder to make it clear.

Notice that the carry-out bit, C_{i+1}, of the last stage will be available after four delay: two gate delays to calculate the propagate signals and two delays as a result of the gates required to implement Equation 13.

$$C_{i+1} = G_i + P_i \cdot C_i \quad (5)$$

$$S_i = P_i \oplus C_i \quad (6)$$

$$G_i = a_i \cdot b_i \quad (7)$$

$$P_i = a_i \oplus b_i \quad (8)$$

Putting $i = 0, 1, 2, 3$ in Equation 5, we get (9)

$$C_1 = G_0 + P_0 \cdot C_0 \quad (10)$$

$$C_2 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0 \quad (11)$$

$$C_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0 \quad (12)$$

$$C_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0 \quad (13)$$

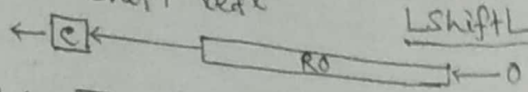
Figure 5 shows that a 4-bit CLA is built using gates to generate the and signals and a logic block to generate the carry out signal according to Equations 10-13.

...puter sc, a Logical shift is a bitwise operation th... shift all the bits of its operand. The two base variants are the logi... left shift & the Logical right shift

Arithmetic shift Microoperations → Logical, circular, Arithmetic

Logical Shift -

(a) Logical shift left LShiftL #2,R0

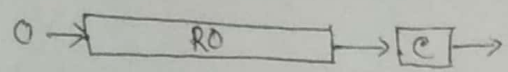


The shift is based on sequential manner.

before : [0] [0 1 1 1 0 ··· 0 1 1]

after : [1] [1 1 0 ··· 0 1 1 0 0] [0] [1 1 1 0 ··· 0 1 1 0]

(b) Logical shift right LShiftR #2,R0

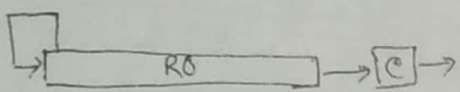


before : [0 1 1 1 0 ··· 0 1 1] [0]

[0 0 1 1 1 0 ··· 0 1] [1]

after : [0 0 0 1 1 1 0 ··· 0] [1]

(c) Arithmetic shift right (sign bit is used as fill in bit)



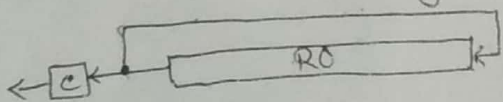
before : [1 0 0 1 1 ··· 0 1 0] [0] AShiftR #2,R0

[1 1 0 0 1 1 ··· 0 1] [0]

after : [1 1 1 0 0 1 1 ··· 0] [1]

Rotate

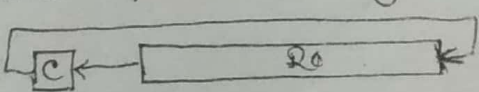
(a) Rotate left without carry RotateL #2, R0



before : [0] [0 1 1 1 0 ··· 0 1 1]

after : [1] [1 1 0 ··· 0 1 1 0 1]

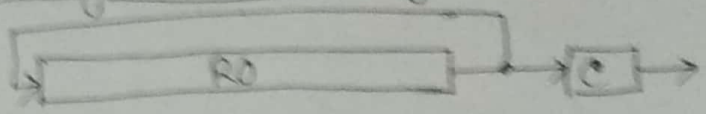
(b) Rotate left with carry RotateLC #2,R0



before : [0] [0 1 1 1 0 ··· 0 1 1] [0] [1 1 1 0 ··· 0 1 1 0]

after : [1] [1 1 0 ··· 0 1 1 0 0]

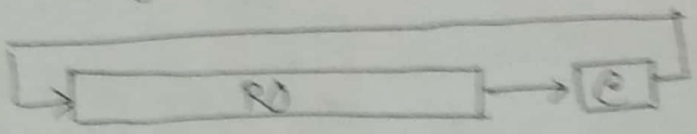
(c) Rotate Right without carry RotateR #2, R0



before : 0 1 1 1 0 . . . 0 1 1 | 0

after : 1 1 0 1 1 1 0 . . . 0 | 1

(d) Rotate right with carry RotateRC #2,



before : 0 1 1 1 0 . . . 0 1 1 | 0

Loop: 0 0 1 1 1 0 . . . 0 1 | 1

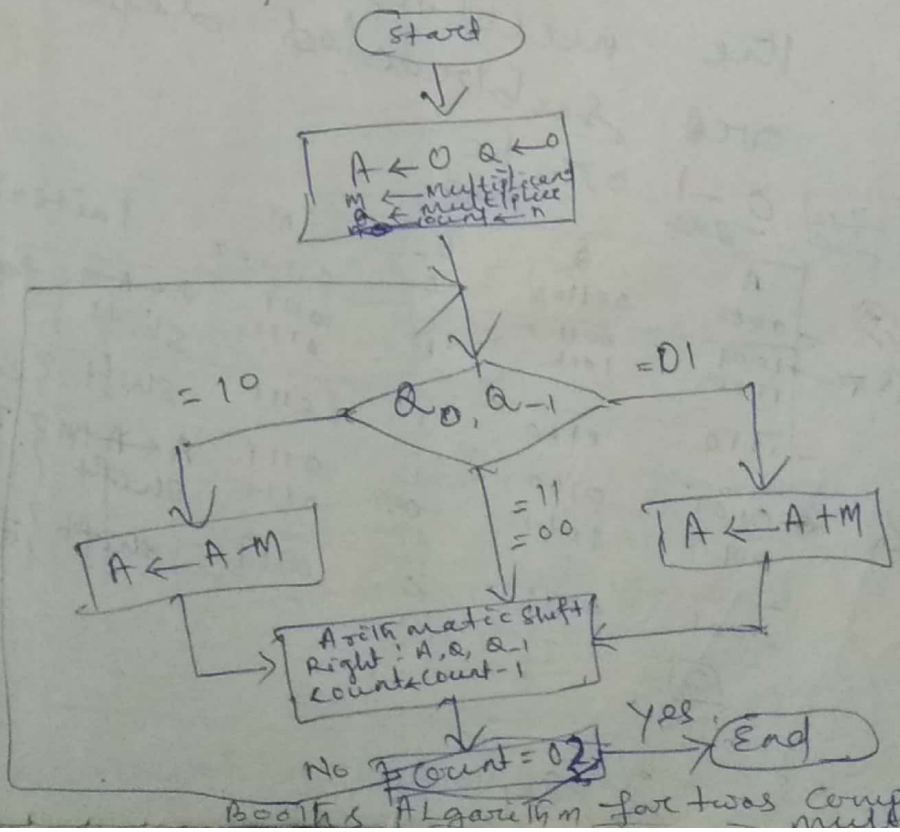
after : 1 0 0 1 1 1 0 . . . 0 | 1

Booth's Multiplication Alg

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

→ It operates on the fact that strings of 0's in the multiplier require no addition but just shifting and a string 1's in the multiplier from bit weight 2^k to weight 2^m can treated as $2^{k+1} - 2^m$

Ex:- (+14) = 001110 has a string of 1's from 2^3 to 2^1.



Booth's Algorithm for twos compl- multi-

The multiplier & multiplicand are placed in Q & M register respectively.

Q_0 → 1-bit register placed Logically at the right end of the Q register designated as Q_{-1}.

Results of multiplication appear in A & Q reg.

A, Q_{-1} are initialized to 0

Control Logic scan the bits of the multiplier one at a time.

If the two bits are 0-0, then all the bits of A, Q, Q_{-1} registers are shifted right to the right 1 bit.

If the two bits differ the multiplicand is added or subtracted, depending on 0-1 or 1-0.

Ex = 7×3

A	Q	Q_{-1}	M	Initial value
0000	0011 = 3	0	0111 = 7	
1001	0011	0	0111	A ← A−M } first cycle
1100	1001	1	0111	shift }
1110	0100	1	0111	shift } 2nd cycle
0101	0100	1	0111	A ← A+M } Third cycle
0010	1010	0	0111	shift }
0001	0101	0	0111	shift } Fourth cycle

(21)

7×3 2×4

Booth's Algorithm

Multiplication of two signed binary no.

Conditions

Consider 4bit $Q_3 Q_2 Q_1 Q_0 | Q_{-1}$

	Q_0	Q_{-1}	
1)	0	0	} only shifting
	1	1	
2)	0	1	} perform → A+M → A then shift
3)	1	0	perform A−M → A; then shift

7 X 3 = 21

7 = M = 0111 , 3 = Q = 0011 , AC = 0000, (for general 1011 = 21)

	A	Q	Q_{-1}	
	0000	0011	0	initially
1)	1001	0011	1	right shift; so → A−M
2)	1100	1001	1	equal shift right
3)	1110	0100	1	A+M
	0101	0100	1	right shift
	0010	1010	0	only shift
4)	0001	0101	0	only shift

A − M ⇒ A + 2's M

0000 + 1001 = 1001

M = 0111 → 1's 1000 → 2's 1001 = −M

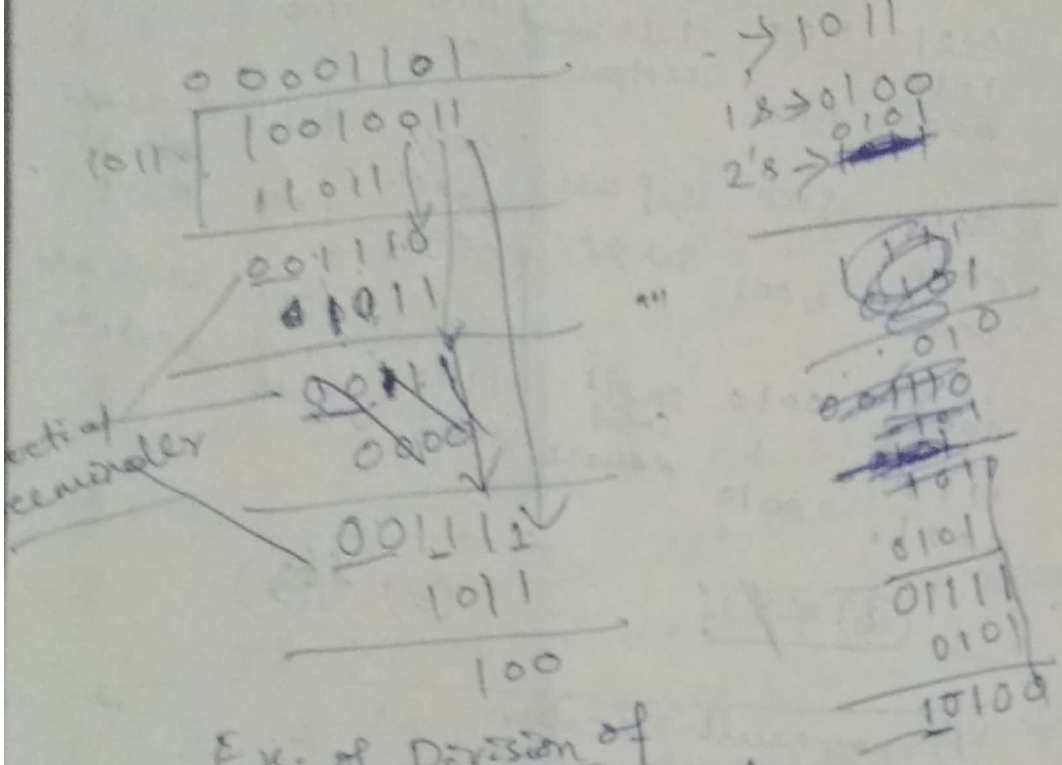
A+M: 1110 + 0111 = 10101

Continuing we get final operation 0001 0101 = 21

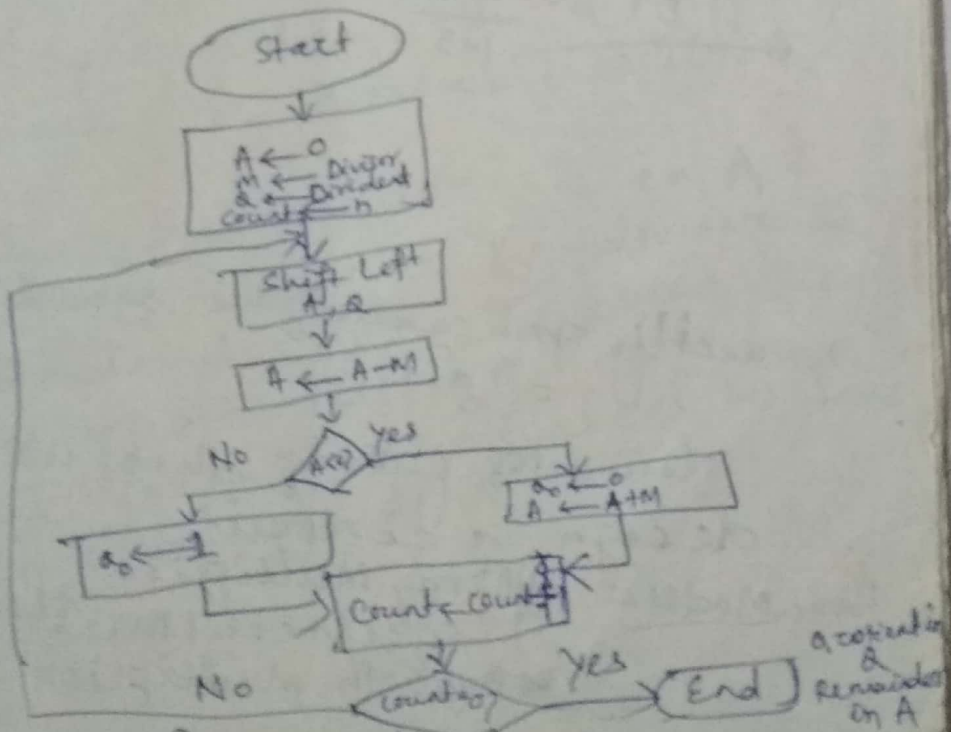
we we get 7 X 3 = 0111 X 0011 = 00010101

Division

Division is some what more complex than multiplication but based on same general principles.



Ex. of Division of Unsigned Binary Int.



: Flowchart for Unsigned Binary division

(a) 7 / 3 — dividend A, Q, load divisor M

A	Q	M = 0011
0000	0111	Initial value
0000	1110	shift
1101		subtract
0000	1110	Restore
0001	1100	shift & subtract
1110		restore
0001	1100	
0011	1000	shift subt
0000		set $Q_0 = 1$
0000	1001	
0001	0010	shift subt.
1110	0010	restore
0001		

(b) 7/(3)

A	Q	M = 1101
0000	0111	Initial value
0000	1110	shift add
1101	1110	restore
0000		
0001	1100	shift add
1110	1100	restore
0001		
0011	1000	shift add
0000	1001	set $Q_0 = 1$
0000		
0001	0010	shift add
1110	0010	restore
0001		

Array multiplication

1011 → 11
1101 → 13
143

exactly you can using the same thing = 2^0 + ... = 143

Now putting this we have design a circuit.

Basic Model –
1) Array Multiplier
2) Sequencial Multiplier
3) Booth Multiplier.

1) Load the divisor into M register and dividend into the A, Q register. The dividend must be expressed as a 2n-bit two's comp. no. Thus 0111 becomes 00000111. The 4 bit 1001 become 11111001.

2) Shift A, Q Left 1 bit pos.

3) If A & M have the same sign of perform $A \leftarrow A - M$; otherwise, $A \leftarrow A + M$.

4) a) If operation is successful or $A = 0$, then set $Q_0 \leftarrow 1$.

b) If the op. is unsuccessful & $A \neq 0$, then set $Q_0 \leftarrow 0$ & restore the previous value of A.

5) Repeat step 2 thru 4 as many time as there are bit pos. in Q.

6) The remainder is in A. If the signs of the divisor & dividend were the same, then the quotient is in Q. Otherwise, the correct quotient is the two's complement of Q.

X86

Introduction to X86 Architecture

The X86 architecture is an instruction set Arch. (ISA) series for computer processor Developed by Intel corporation, x86 Arch. defines how a processor handles & executes different instructions passed from the OS & I/O program.

Microprocessor

- As opposed to mainframes
- All CPU functionality on a single chip.
- Started with popular home computers
  - 8-bit. 6502/6510, Z80
  - 32-bit Motorola 68000 (My time)

Intel Microprocessors

- 4004
  - 1971: First single-chip mp; 740kHz
- 8008
  - 1972: First 8-bit mp; 800 kHz
- 8080
  - 1974: Larger Instruction set; 2 MHz
- 8086/88
  - 1979: ~ 5 MHz 29,000 transistors (today 7B)

∴ Used by NASA at least until 2002 for space shuttle operations.

- IBM PC revolution

Manufactured by:

- Intel
- AMD
- NEC
- Fujitsu
- OKI
- Siemens

X86 Architecture

Designed in 1978, X86 architecture was one of the first ISAs for microprocessor-based computing.

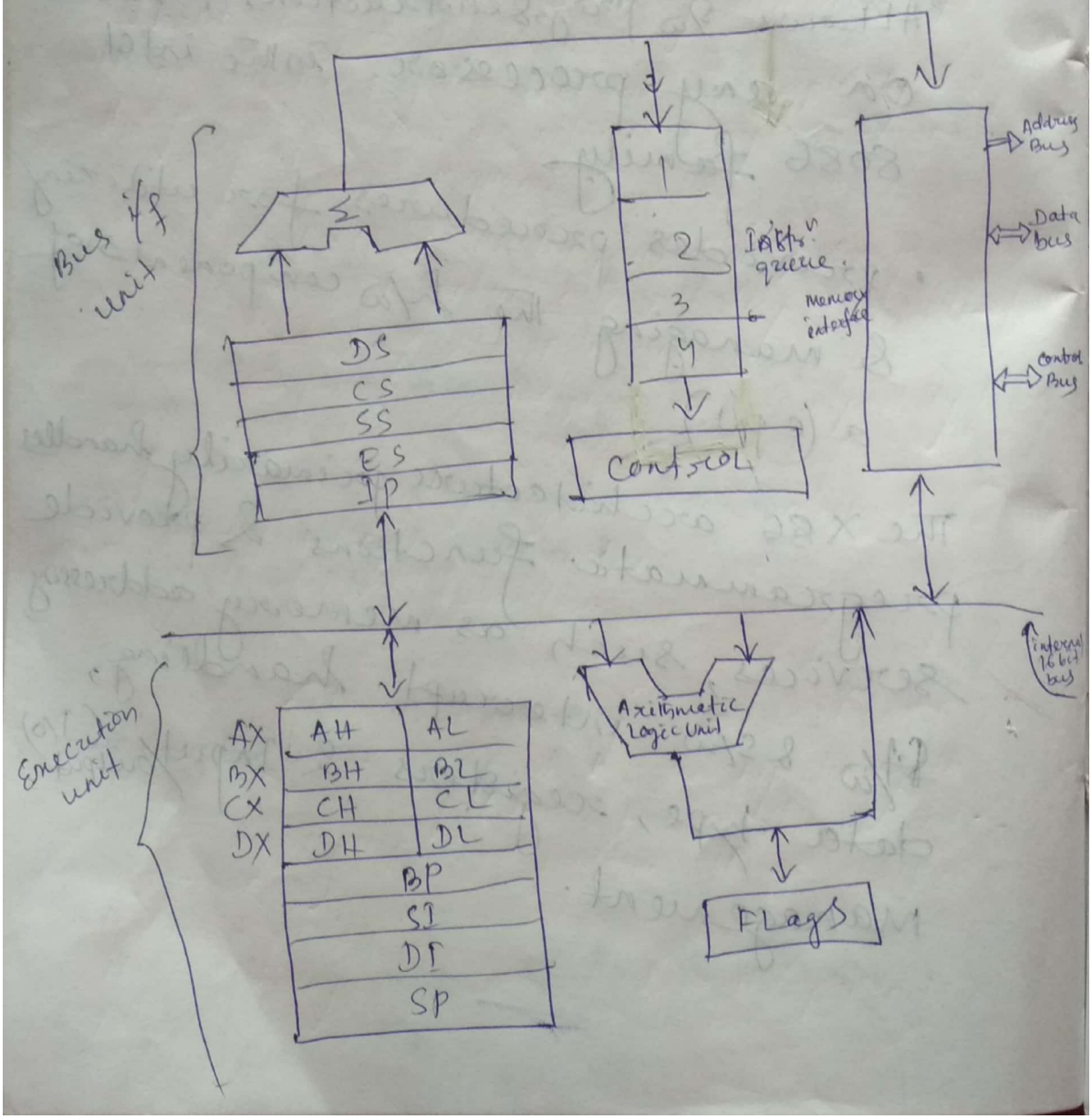
Key feature include:

- provides a logical framework for executing instructions through a processor
- Allows s/w programs/instructions to run on any processor in the Intel 8086 family
- provides procedures for utilizing & managing the h/w components of a (CPU)

The X86 architecture primarily handles programmatic functions & provide services, such as memory addressing, s/w & s/w interrupt handling, data type, registers & Input/output (I/O) management.

Classified by bit amount, the X86 architecture is implemented in multiple mp., including 8086, 80286, 80386, Corez, Atom and the pentium series.
Additionally, other mp manufacturers, like AMD and VIA Technologies have adopted the X86 architecture.

X86 Architecture



Registers: The processor provides 16 registers for use in programming. These registers can be grouped as follows:

- General-purpose data registers: These eight registers are available for storing operands and pointers.
- Segment registers. These registers hold up to six segment selectors.
- Status and Control registers: These registers report and allow modification of the state of the processor and of the program being executed.

General-Purpose Data Registers

The 32 bit general-purpose data registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are provided for holding the following items:

- Operands for logical and arithmetic operations
- Operands for address calculations
- Memory pointers

Although all these registers are available for general storage of operands, results and pointers caution should be used when referencing the ESP register. The ESP register holds the stack pointer and as a general rule should not be used for any other purpose.

Segment Resisters

The 6 segment registers are:

- Stack Segment (SS). Pointer to the stack.
- Code Segment (CS). Pointer to the code.
- Data Segment (DS). Pointer to the data
- Extra Segment (ES). Pointer to the extra data ('E' stands for 'Extra')
- F Segment (FS). Pointer to more extra data ('F' ~~stands~~ comes after 'E')
- G Segment (GS). Pointer to still more extra data ('G' comes after 'F')

Most applications on most modern operating systems (Free BSD, Linux, or Microsoft Windows) use a memory model that points nearly all segment registers to the same place and uses paging instead, effectively disabling their use. Typically the use of FS or GS is an exception to this rule, instead being used to point at thread-specific data.

x86 Processor Registers and Fetch-Execute Cycle

There are 8 registers that can be specified in assembly-language instructions: eax, ebx, ecx, edx, esi, edi, ebp and esp. Register esp points to the "top" word currently in use on the stack (which grows down).

Register ebp is typically used as a pointer to a location in the stack frame of the currently executing function.

Register ecx can be used in binary arithmetic operations to hold the second operand.

There are two registers that are used implicitly in x86 programs and cannot be referenced by name in an assembly language program.

These are eip, the "instruction pointer" or "program counter", and eflags, which contains bits indicating the result of arithmetic and compare instructions.

The basic operation of the processor is to repeatedly fetch and execute instructions, while (running) {

```
fetch instruction beginning at address in eip;
eip <- eip + length of instruction;
execute fetched instruction;
}
```

Execution continues sequentially unless execution of an instruction causes a jump, which is done by storing the target address in eip (this is how conditional and unconditional jumps, and function call and return are implemented)

Addressing modes

The addressing modes indicate how the operand is presented.

Register Addressing
Operand address R is in the adress field.

```
mov ax,bx ; moves contents of register bx into ax.
```

Immidiate
Actual value is in the field.

```
mov ax,1 ; moves contents of register value of 1 into
           register ax
```

Or:

```
mov ax, 010Ch ; moves value of 0x010C into register ax
```

Direct memory addressing
Operand address is in the address field.

```
.data
my_var dw 0abcdh ; my_var = 0xabcd
.code
mov as, [my_var] ; copy my_var content in ax (ax=0xabcd)
```

Direct offset addressing
Uses arithmetics to modify address

```
byte_tbl db 12,15,16,22,..... ; Tables of bytes
mov al [byte_tbl+2]
mov al, byte_tbl[2] ; Same as the former
```

Register Indirect
Field points to a register that contains the operand adress.

```
mov ax, [di]
```

The registers used for indirect addressing are BX, BP, SI, DI
Base-index

```
mov ax [bx+di]
```

For example, if we are talking about an array, BX contains the address of the beginning of the array, and DI contains the index into the array.

Base-index with displacement

```
mov ax [bx+di+10]
```

CPU Operation Modes

Real Mode

Real Mode is a holdover from the original Intel 8086. The Intel 8086 accessed memory using 20-bit addresses. But as the processor itself was 16-bit, Intel invented an addressing scheme that provided a way of mapping a 20-bit addressing space into 16-bits words. Today x86 processors start in the so-called Real Mode, which is an operating mode that mimics the behavior of the 8086, with some very tiny differences, for backwards compability

Protected Mode

Flat memory model

If programming in a modern operating system (such as Linux, Windows), you are basically programming in flat 32-bit mode. Any register ~~instead of~~ can be used in addressing and it is generally more efficient to use a full 32-bit register instead of a 16-bit register part. Additionally, segment registers are generally unused in flat mode, and it is generally a bad idea to touch them.

Multi segment Memory Model

Using a 32-bit register to address memory, the program can access almost (almost) all of the memory in a modern computer. For earlier processors (with only 16-bit registers) the segmented memory model was used.

The 'CS', 'DS' and 'ES' registers are used to point to the different chunks of memory a small program (small model) the CS=DS=ES. For larger memory models these 'Segments' can point to different locations.

Register Transfer Language And Micro Operations:

Register Transfer language:

- Digital system are composed of modules that are constructed from digital components, such as register decoders, arithmetic elements, and control logic.
- The modules are interconnected with common data and control paths to form a digital computer system.
- The operations executed on data stored in registers are called microoperations.
- The microoperation is an elementary operations performed on the information stored in one or more registers.
- Examples are shift, count, clear, and load.
- Some of the digital components from before are registers that implement microoperations
- The internal hardware organization of a digital computer is best by specifieng:
  - The set of registers it contains and their ~~pro~~ functions
  - The sequence of microoperations performed on the binary information stored.
  - The control that indicates the sequence of microoperations.

Use symbols, rather than words to specify the sequence of microoperations

The symbolic notation used is called a register transfer language

A programming language is a procedure for writing symbols to specify a given computation process

Defines symbols for various types of microoperations and describe associated hadware that can implement the microoperations.

Register Transfer

Designate computer registers by capital letters to denote its function.

The register that holds an address for the memory unit is called MAR.

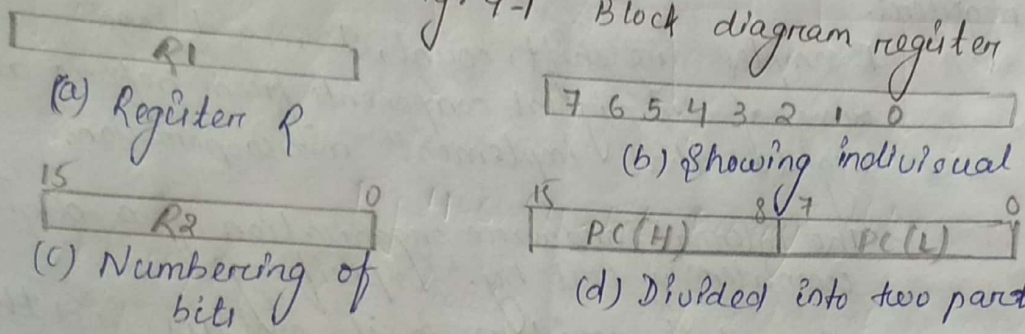
The program counter register is called PC.

IR is the Instruction Register and R1 is a processor register.

The individual flip-flops in an n-bit register are numbered in sequence from 0 to n-1.

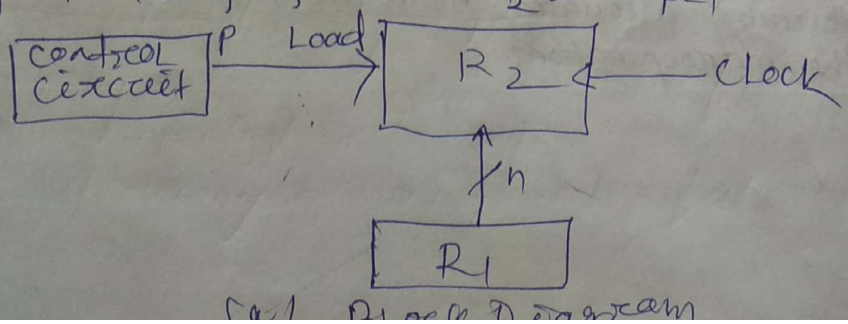
Refer to figure 4-1 for the different representation of a register.

Fig. 4-1 Block diagram register

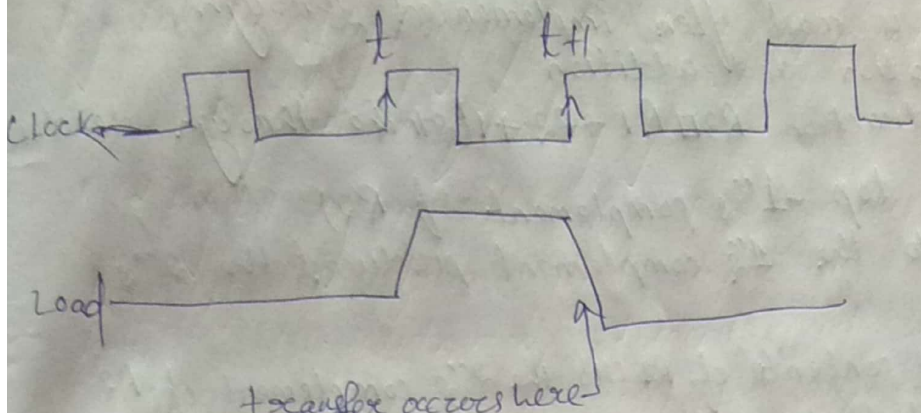


- Designate information transfer from one register to another by R2 ◻ R1
- This statement implies that the hardware is available.
  - The outputs of the source must have a path to inputs of the destination
  - The destination register has a parallel load capability.
- If the transfer is to occur only under a predetermined control condition, designate it by
- If (P=1) then (R2 ◻ R1) or, P: R2 ◻ R1, where P is a control function that can be either 0 or 1.
- Every statement written in register transfer notation implies the presence of the required hardware construction

Fig. - 4.2 Transfer from R1 to R2 when P=1



(a) Block Diagram



(b) Timing Diagram

Table 4.1 Basic symbols for register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	MAR, R2
Parantheses ()	Denotes a part of register	R2(0-7), R2(L)
Arrow ←	Denotes transfer of information	R2 ← R1
Comma ,	Separates two microoperations	R2 ← R1, R1 ← R2

Arithmetic Micro-Operations

There are four categories of the most common micro operations:

Register Transfer: transfer binary information from one register to another.

Arithmetic: perform arithemetic operation on numeric data stored in registers

Logic: Perform bit manipulation operations on non-numeric data stored in registers

Shift: Perform shift operations on data stored in registers

The basic arithemetic micro operation are addition, subtraction, increment, decrement and shift

Eg. of addition: R3 ← R1 + R2

Subtraction is most often implemented through complementation and addition

Eg. of subtraction R3 ← R1 + R2̄ + 1 (strike through denotes bar on the top - 1's complement of R2)

Adding 1 to the 1's complement produces the 2's complement

Adding the contents of R1 to the 2's complement of R2 is equivalent to subtracting

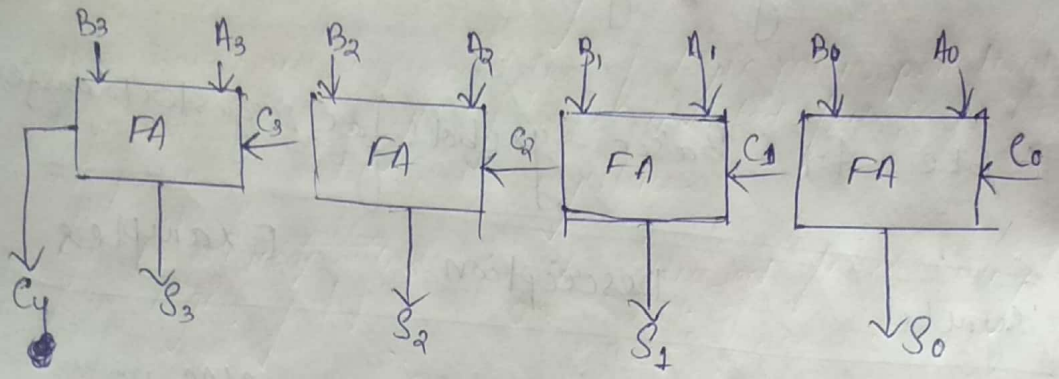


Fig 4-6. 4-bit binary adder

Multiply and divide are not included as micro operations

A micro operation is one that can be executed by one clock pulse.

Multiply (divide) is implemented by a sequence of add and shift micro operations (subtract and shift)

To implement the add micro operation with hardware we need the register that hold the data and the digital component that performs the addition.

A full-adder adds two bits and a previous carry

A binary adder is a digital circuit that generates the arithemetic sum of two binary of any length

~~A binnary number of any~~

~~A binnary added is constructed with full-adder circuits that generates the arithemetic sum of two binary numbers of any length.~~

A binnary added is constructed with full adder circuits connected in cascade.

An n-bit binary adder requires n full adders

The subtraction A-B can be carried out by the following steps: Take the 1's complement of B (invert each bit)

Get the 2's complement by adding 1

Add the results to A.

The addition and subtraction operations can be into one common circuit by including an XOR gate with each full-adder.

The increment micro operation adds one to a number in a register

This can be implemented by using a binary counter - every time the count enable is active, the count is incremented by one

If the increment is to be performed independent of a particular register, then use half adders connected in cascade.

An n-bit binary incrementer requires n half-adders.

Each of the arithmetic micro operations can be implemented in one composite arithmetic circuit

The basic component is the parallel adder

Multiplexers are used to choose between the different operations.

The output of the binary adder is calculated from the following sum: $D = A + Y + C_{in}$

Logic Microoperations

- Logic operations specify binary operations for strings of bit stored in registers and treat each bit, separately
- Example: the XOR of R1 and R2 is symbolized by

  P: R1 ⊡ R2 ⊕ R2

- Example: R1 = 1010 and R2 = 1100

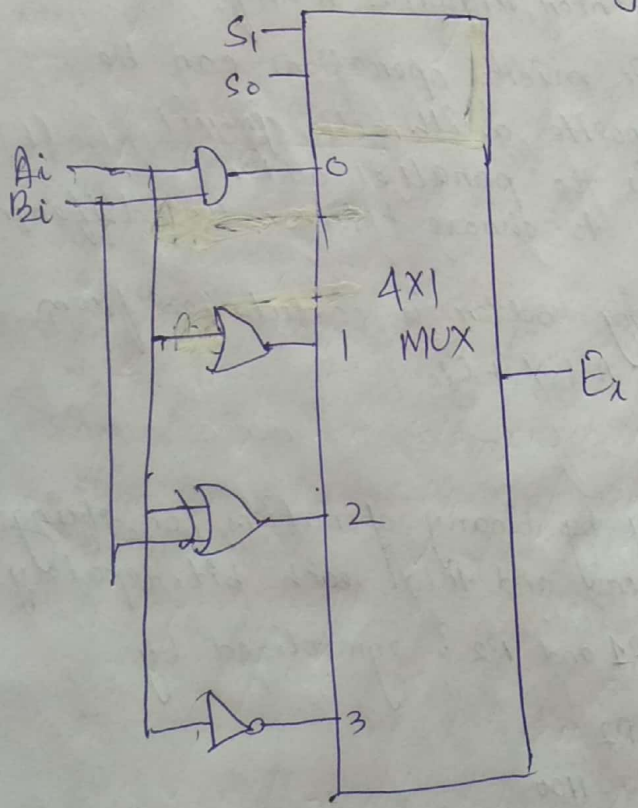
  1010 Content of R1
  1100 Content of R2

  0110 Content of R1 after P = 1

- Symbols used for logical microoperations:
  - OR: ⊡
  - AND: ⊡
  - XOR: ⊕

- The + sign has two different meanings: Logical OR & Summation
- When + is in a microoperation, then summation
- When + is in a control function then OR
- Example:
  P + Q: R1 ◻ R2 + R3, R4 ◻ R5 ◻ R6
- There are 16 different logic operations that can be performed with two binary variables
- The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers.
- All 16 microoperations can be derived from using four logic gates

Fig 4.10 One stage of Logic circuit



(a) Logic diagram

S1	S0	output	operation
0	0	E = A ∧ B	AND
0	1	E = A ∨ B	OR
1	0	E = A ⊕ B	XOR
1	1	E = $\tilde{A}$	complement

(b) Function table

- Logic microoperations can be used to change bit values, delete a group of bits or insert new bit values into a register.
- The selective-set operation sets to 1 the bits in A where there are corresponding 1's in B

1010 A before
1100 B
(Logic operand) 1110 A after A ◻ A ◻ B

- The selective-complement operation complements bits in A where there are corresponding 1's in B

1010 A before
1100 B
(Logic Operand) 0110 A after
A ◻ A ⊕ B

- The selective-clear operation clears to 0 the bits in A only where there are corresponding 1's in B

1010 A before
1100 B (Logic operand) 0010 A
after A ◻ A ◻ B

- The mask operation is similar to the selection-clear operation, ~~is similar~~ except that the bits of A cleared only where there are corresponding 0's in B.

1010 A before
1100 B
(Logic operand) 1000 A
after A ◻ A ◻ B

- The insert operation inserts a new value into a group of bits
- This is done by first masking the bits

to be replaced and then ORing them with the bits to be inserted.

```
01101010    A before
00001111    B (mask)
0000 1010   A after masking
0000 1010   A before
1001 0000   B (inverted)
1001 1010   A after intersection
```

The clear ~~option~~ operation compares the bits in A and B and produces an all 0's result if the two number are equal

```
1010  A
1010  B
0000  A ← A⊕B
```

Shift Microoperations

Shift microoperations are used for serial transfer of data. They are also used in conjuction with arithmetic logic, and other data-processing operations.

There are three types of shift: logical, circular, and arithmetic

A logical shift is one that transfers 0 through the serial input.

The symbols shl and shr are for logical shift-left and shift-right by one position R1 ← shl R.

The circular shift (aka rotate) circulates the bits of the register around the two ends without loss of information.

The symbols cil and cir ~~air~~ circular shift left and right

The arithmetic shift shifts a signed binary number to the left or right.
To the left is multiplying by 2, to the right is dividing by 2.
Arithmetic shifts must leave the sign bit unchanged
The sign ~~bit unchanged~~ reversal occurs if the bit is R_{n-1} changes in value after the shift.
This happens if the multiplication causes an overflow.
An overflow flip-flop V_s can be used to detect the overflow $V_s = R_{n-1} \oplus R_{n-2}$

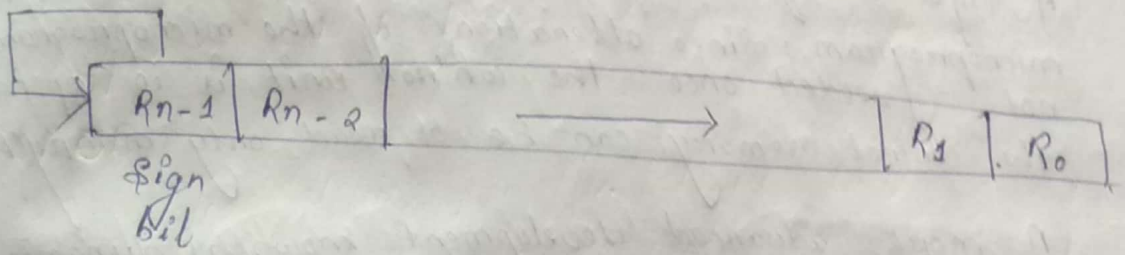


Figure 4-11 Arithmetic shift right

- A bi-directional shift unit with parallel load could be used to implement this
- Two clock pulses are necessary with this configuration. one to load the value and another to shift.
- In a processor unit with many registers it is more effecient to implement the shift operation with a combinational circuit.
- The content of a register to be ~~defined~~ shifted is first placed onto a common bus and the the output is connected to the combinational shifter, the shifted number is then loaded back into the register
- This can be constructed with multiplexers.

Arithmetic Logic unit

- The arithmetic logic unit (ALU) is a common operational unit connected to a number of storage registers
- To perform a microoperations, the contents of specified registers are placed in the inputs of ALU.

- The ALU performs an operation and the result is then transferred to a destination register
- The ALU is a combinational circuit so that the entire register can be performed during one clock pulse period.

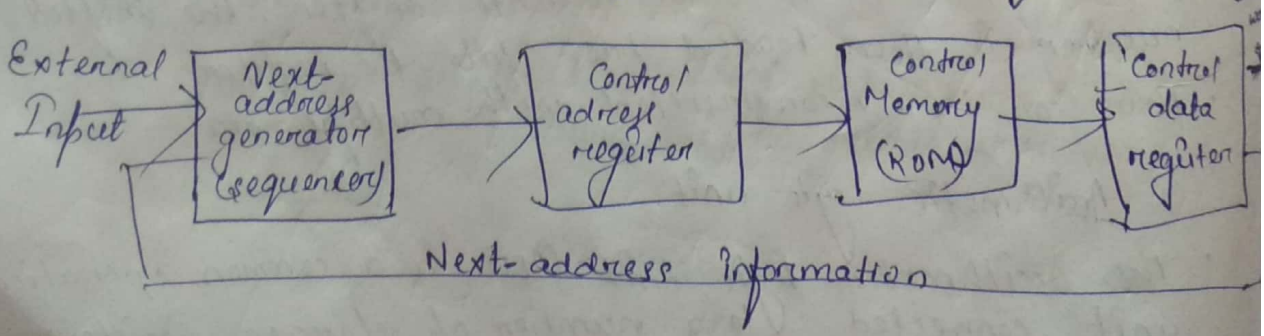
Micro Programmed Control

A control unit whose binary control variables are stored in memory is called a microprogrammed control unit. Each word in control memory contains within it a microinstruction. The microinstruction specifies one or more microoperations for the system. A sequence of microoperations constitutes 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).

A more advanced development known as dynamic microprogramming permits a microprogram to be loaded initially from an auxiliary memory such a magnetic disk.

Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing (to change the microprogram) but is used mostly for reading. A memory that is part of a control unit is referred to as a control memory

Figure 7.1 - Microprogrammed control organization



The next address generator is sometimes called a microprogram sequencer, as it determines the address sequence that is read from control memory. The control data register holds the present microinstruction while the next address is computed and read from memory.

The data register is sometimes called a pipeline register The main advantage of the microprogrammed control is the fact that once the hardware configuration is established there should be no need for further hardware or wiring changes. If we want to establish a different control sequence for the system, all we need to do is specify a different set of microinstructions for control memory. The hardware configuration should not be changed for different operations; the only thing that must be changed is the microprogram residing in control memory. It should be mentioned that most computers based on the reduced instruction set computer (RISC).

Address Sequencing

Microinstructions are stored in control memory in groups, with each group specifying a routine. The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a mapping process A mapping procedure is a rule that transforms the instruction code into a control memory address

1. Incrementing of the control address register.
2. Unconditional branch or conditional branch, depending on status bit conditions.
3. A mapping process from the bits of the instruction to an address for control memory.
4. A facility for subroutine call and return

Conditional Branching

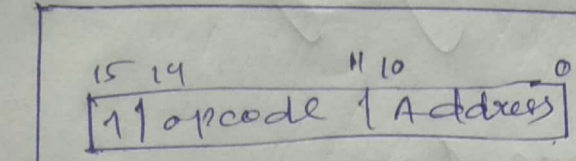
Special bits: The branch logic provides decision-making capabilities in the control unit. The status conditions are special bits in the system that provide parameter information such as the carry out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions

Branch Logic: The branch logic hardware may be implemented in a variety of ways. The simplest way is to test the specified condition and branch to the indicated address if the condition is met; otherwise, the address register is incremented. This can be implemented with a multiplexer

Mapping of Instruction: A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine for an instruction is located. The status bits for this type of branch are the bits in the operation code part of the instruction

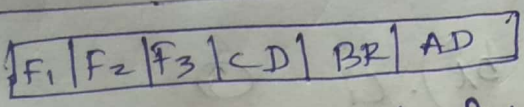
Microinstruction Format

The microinstruction format for the control memory is shown in Figure. The 20 bits of the microinstruction are divided into four functional parts. The three fields F1, F2, F3 specify microoperations for the computer. The CD field selects status bit conditions. The BR field specifies the type of branch to be used. The AD field contains a branch address. The address field is seven bits wide, since the control memory has $128 = 27$ words.



Symbol	Opcode	Description
ADD	0000	AC ← AC + M[EA]
BRANCH	0001	If (AC < 0) then (PC ← EA)
STORE	0010	M[EA] ← AC
EXCHANGE	0011	AC ← M[EA], M[EA] ← AC

EA is the effective Address



F_1, F_2, F_3: Microoperation fields
CD: condition for Branching
BR: Branch field
AD: Address field

a microinstruction can specify two simultaneous microoperations from F2 & F3 & none from F1

DR ← M[AR] with F2 - 100
and PC ← PC + 1 with F3 - 101

Input/Output Organisation

Modes of data transfer:-

There are two modes of data transfer.

(i) Asynchronous mode of data transfer

(ii) Synchronous mode of data transfer.

Asynchronous data transfer Mode

In asynchronous data transmission a start bit is added at both end of the character code.

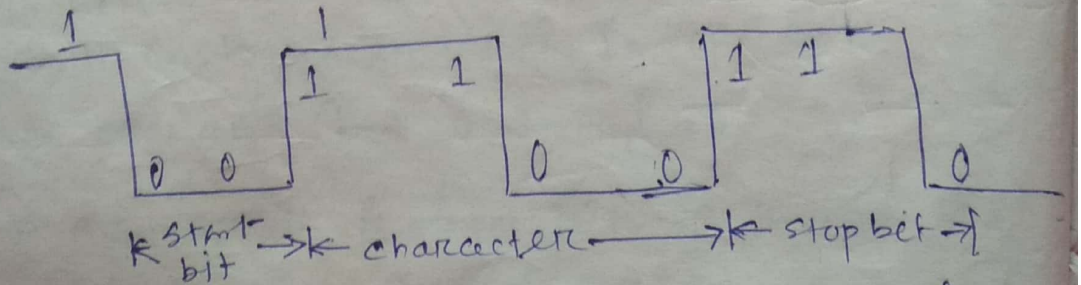
Each character consists of three parts:-

(I) A start bit
(II) character bit
(III) A stop bit.

→ The first bit is called start bit. It is denoted by zero and it indicates the begining of a character.

→ The Last bit of the character is stopbit is always '1' & it always indicates the end of a character.

The figure shows the asynchronous mode of serial comm/n.



In this mode the cpu reg & the interface reg doesn't get clock signal simultaneously

A receiver can detect the transmitted character having the following knowledge.

* When a character is not being sent the line is kept in the one state.

* The initiation of character transmission is always detected by the start bit (0).

* The character bit always follows the start bit.

* After the last bit is transmitted, a stop bit is detected, when the line returns to the one state (for at least one bit time).

Synchronous Mode of data transfer:-

In this mode both transmitter & receiver share a common clock frequency

-> Bits are transmitted from the one end to the other end at the rate dedicated by the clock pulse.

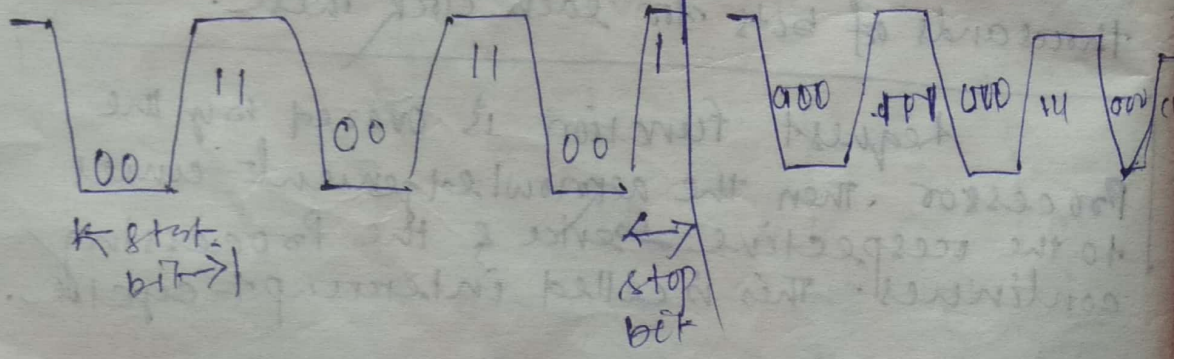
-> Synchronous transmission doesn't use start & stop bit to frame a character word.

MODEMS are used for synchronous data transfer because it transmits thousands of bits for each clock pulse.

Request function is overed by the Processor. Then the acknowledgement comes to the respective device & the Processing continues. This is called interrupt cycle.

Difference between Synchronous & Asynchronous

Asynchronous data transfer	Synchronous data transfer
→ The cpu register & interface register don't get clock signal simultaneously.	→ The two units get clock signal simultaneously.
→ In asynchronous data transfer mode, there is a need of start & stop bit to transfer character.	→ There is no need of start or stop bit.
→ It can be used for short distance transmission	→ It can be used for long distance transmission.
→ It depends upon clock signal.	→ It doesn't depend upon clock signal.
→ Asynchronous transmission is very complex due to use of start & stop bit.	→ Synchronous Mode of data transfer is very simple.
→ It requires ~~more~~ more time for transmission of data.	→ It requires less time for transmission of data.



Isolated I/O VS Memory Mapped I/O

* The memory read and write control lines are enabled during data transfer from the memory or to the i/o devices or in to the memory.

* This configuration of memory isolates all the I/O device address from the I/O mem address which is known as isolated I/O.

* The isolated configuration to the cpu has distinct no. of I/O instruction & each of the instruction is isolated with the address of the interface register.

* The isolated I/O method isolates the memory address and the I/O address, so that the memory address value are not effected by I/O device address value.

The alternative method in which the same memory space is used for both memory address value & I/O device address value is known as Memory mapped I/O.

* In this case the comp uses one set of read, write signal & doesn't isolate bet'n memory address & I/O address. This configuration is referred as 'Memory mapped I/O.

Modes of data transfer

The mode of data transfer betn Perciphercal devices and memory and or Processor can be classified into ~~two~~ 3 types:—

(i) Programmed I/O
(ii) Interrupt initiated I/O
(iii) Direct Memory access.

Programmed I/O:—

Programmed I/O operation is the result of the I/O instruction written in the computer program, so accordingly data transfer takesplace.

→ Usually the data transfer is betn the cpu registers & the peripheral devices.

→ Transferring data under the Program control requires a constant watch of the Peripheral by the cpu.

→ Once the data transfer is initiated the cpu is required to monitor the interface to see when transfer of data is again made.

Interrupt initiated I/O :—

An alternative method for the cpu to constantly monitoring the the interface from the computer, when it is ready to transfer the data.

→ This mode of data transfer uses the interrupt facility.

→ When the interrupt access the cpu momentarily suspends its task & accordingly to the Priority of the device, the data transfer takesplace.

Scanned by CamScanner

Direct Memory Access :-

When Large Volume of data are to be moved, a more efficient technique is required called direct Memory Access (DMA).

→ DMA involves an additional module in the system Bus.

→ DMA uses Buses only, when the Processor doesn't need to interface with the buses, so the DMA module forces the Processor to suspend its operation temporarily.

→ When the Processor wants to read/write a block of data, it issues a command to the DMA module by sending the following information.

(i) Wheather read/write is required.
(ii) The address of the I/O device involved.
(iii) Starting address of the Memory Location to read/write.
(iv) The no. of words to be read/write.

→ After the execution of interrupt, the interrupt acknowledgement signal is fed to the Priority encoder from the cpu.

→ The vector address VAD is Providing

Interfacing of the I/O devices:-

A hardware unit that plays an important role betn the cpu and the peripheral devices to seperate the I/O data transfer is known as interface unit.

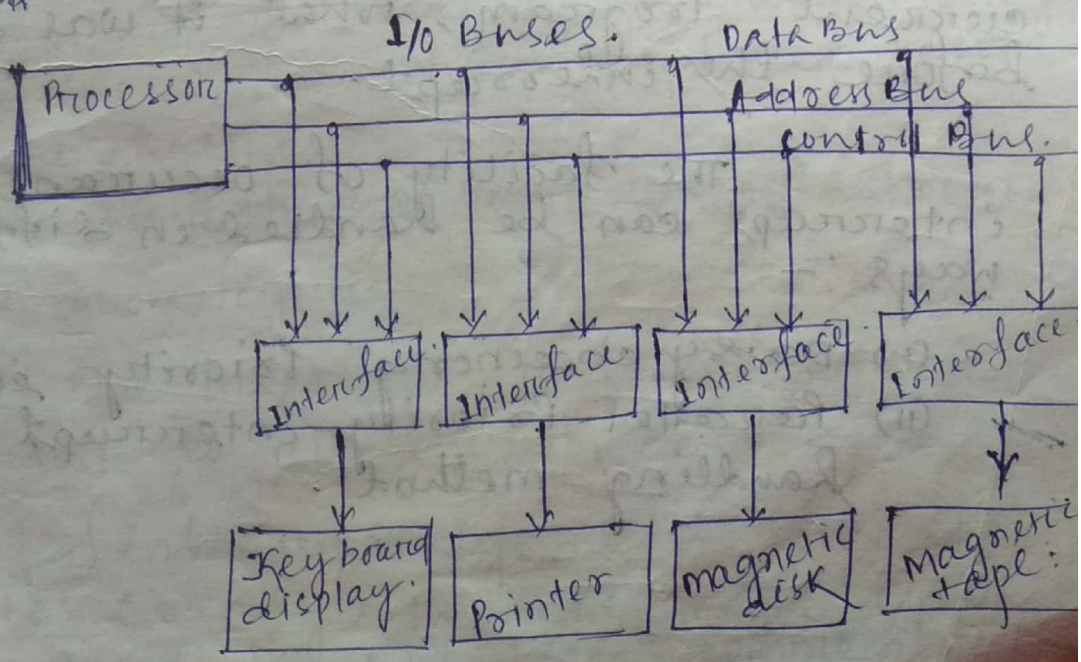
→ The I/O interface provides a method for transfer of information betn internal storage and external I/O devices.

→ The peripheral connected to a comp needs special comm/n links for interfacing them with the cpu.

The computer has no meaning without the ability to receive information from the peripheral devices or transfer the data/result towards the peripheral devices.

The figure shown below shows the comm/n link betn the Processor and several peripheral devices (like key-board, Printer, magnetic disk, etc).

→



→ The different peripheral devices are connected to the processor through I/O bus with different interface ckt.

→ The interface ckt decodes the address ~~and interprete them~~ & control signal is received from the I/O bus & interpriate them for the respective peripheral devices.

→ The control bus provides different control signal for different peripheral devices.

→ During the address decoding system, if the interface ckt detect the address same as the address of the peripheral devices, then the device will be activated.

→ The peripheral device in which the address doesn't match then the device will be inactivated & no operation will takesplace

Interrupt

Interrupt is a system or method in which the computer deviates (suspend the task) momentarily from what it was doing to stop scanning of I/O device data transfer.

After this it returns to the current program, what it was doing before the interrupt.

The facility of occurance of interrupt can be handled in different ways :-

(i) Daisy chaining Priority interrupt
(ii) Parallel Priority interrupt handling method.

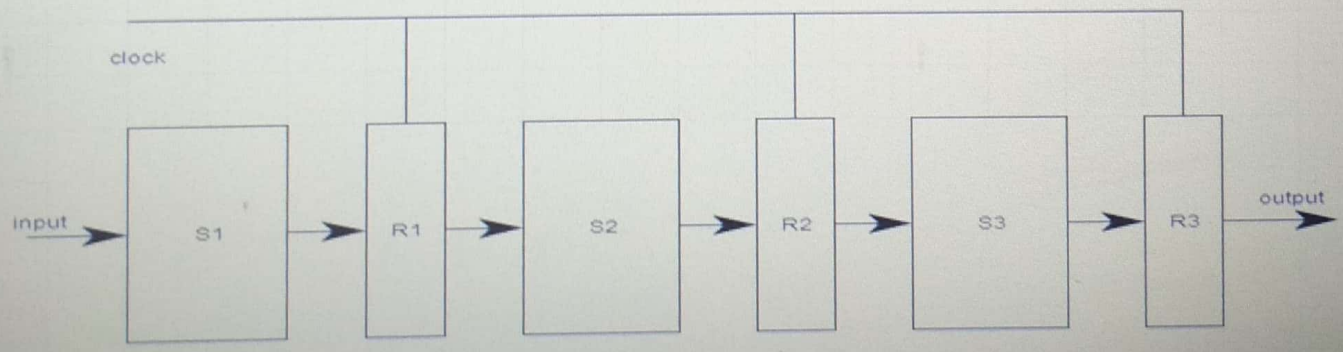
Pipelining

Pipelining is the process of accumulating instruction from the processor through a pipeline. It allows storing and executing instructions in an orderly process. It is also known as **pipeline processing**.

Pipelining is a technique where multiple instructions are overlapped during execution. Pipeline is divided into stages and these stages are connected with one another to form a pipe like structure. Instructions enter from one end and exit from another end.

Pipelining increases the overall instruction throughput.

In pipeline system, each segment consists of an input register followed by a combinational circuit. The register is used to hold data and combinational circuit performs operations on it. The output of combinational circuit is applied to the input register of the next segment.



Pipeline system is like the modern day assembly line setup in factories. For example in a car manufacturing industry, huge assembly lines are setup and at each point, there are robotic arms to perform a certain task, and then the car moves on ahead to the next arm.

Types of Pipeline

It is divided into 2 categories:

1. Arithmetic Pipeline
2. Instruction Pipeline

Arithmetic Pipeline

Arithmetic pipelines are usually found in most of the computers. They are used for floating point operations, multiplication of fixed point numbers etc. For example: The input to the Floating Point Adder pipeline is:

```
X = A*2^a

Y = B*2^b
```

Here A and B are mantissas (significant digit of floating point numbers), while **a** and **b** are exponents.

The floating point addition and subtraction is done in 4 parts:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract mantissas
4. Produce the result.

Registers are used for storing the intermediate results between the above operations.

Instruction Pipeline

In this a stream of instructions can be executed by overlapping *fetch*, *decode* and *execute* phases of an instruction cycle. This type of technique is used to increase the throughput of the computer system.

An instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipeline. Thus we can execute multiple instructions simultaneously. The pipeline will be more efficient if the instruction cycle is divided into segments of equal duration.

Pipeline Conflicts

There are some factors that cause the pipeline to deviate its normal performance. Some of these factors are given below:

1. Timing Variations

All stages cannot take same amount of time. This problem generally occurs in instruction processing where different instructions have different operand requirements and thus different processing time.

2. Data Hazards

When several instructions are in partial execution, and if they reference same data then the problem arises. We must ensure that next instruction does not attempt to access data before the current instruction, because this will lead to incorrect results.

3. Branching

In order to fetch and execute the next instruction, we must know what that instruction is. If the present instruction is a conditional branch, and its result will lead us to the next instruction, then the next instruction may not be known until the current one is processed.

4. Interrupts

Interrupts set unwanted instruction into the instruction stream. Interrupts effect the execution of instruction.

5. Data Dependency

It arises when an instruction depends upon the result of a previous instruction but this result is not yet available.

Advantages of Pipelining

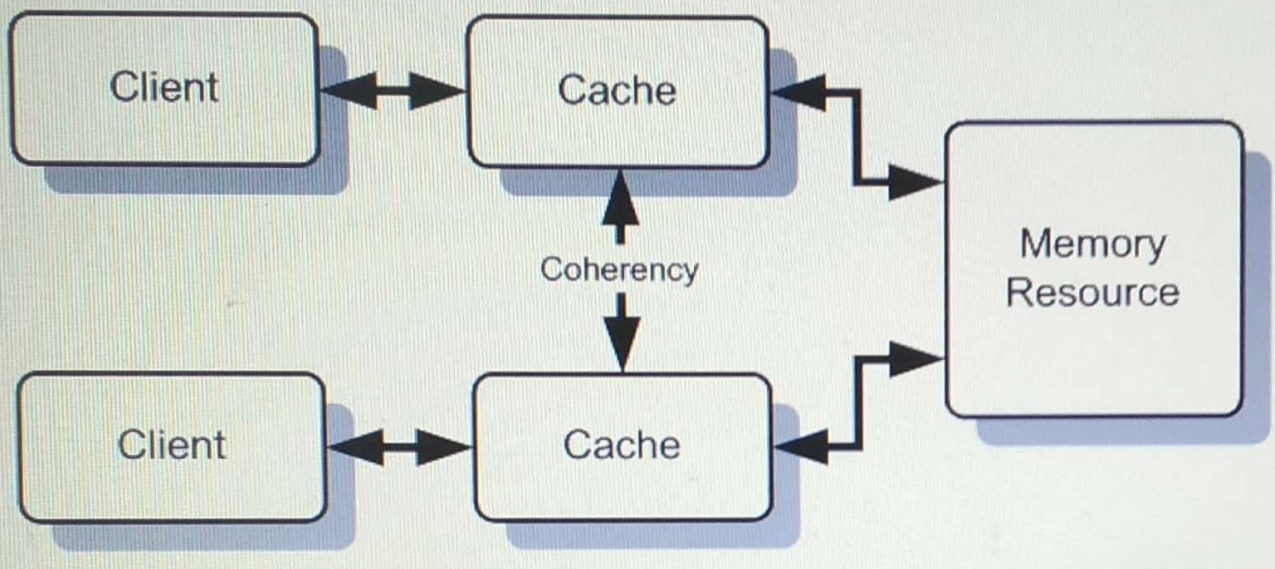
1. The cycle time of the processor is reduced.
2. It increases the throughput of the system
3. It makes the system reliable.

Disadvantages of Pipelining

1. The design of pipelined processor is complex and costly to manufacture.
2. The instruction latency is more.

In computer architecture, **cache coherence** is the uniformity of shared resource data that ends up stored in multiple local caches. When clients in a system maintain caches of a common memory resource, problems may arise with incoherent data, which is particularly the case with CPUs in a multiprocessing system.

In the illustration on the right, consider both the clients have a cached copy of a particular memory block from a previous read. Suppose the client on the bottom updates/changes that memory block, the client on the top could be left with an invalid cache of memory without any notification of the change. Cache coherence is intended to manage such conflicts by maintaining a coherent view of the data values in multiple caches.



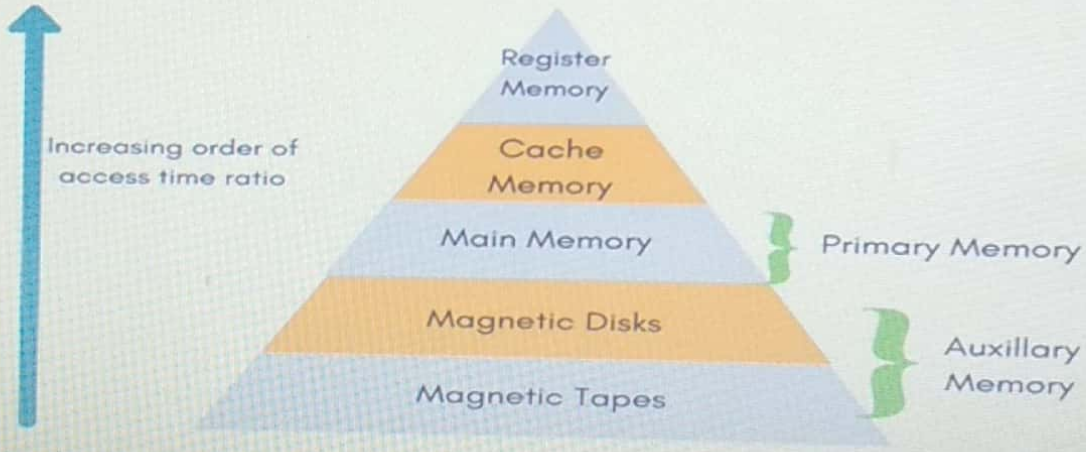
Memory Organization

A memory unit is the collection of storage units or devices together. The memory unit stores the binary information in the form of bits. Generally, memory/storage is classified into 2 categories:

into 2 categories:

- **Volatile Memory**: This loses its data, when power is switched off.
- **Non-Volatile Memory**: This is a permanent storage and does not lose any data when power is switched off

Memory Hierarchy



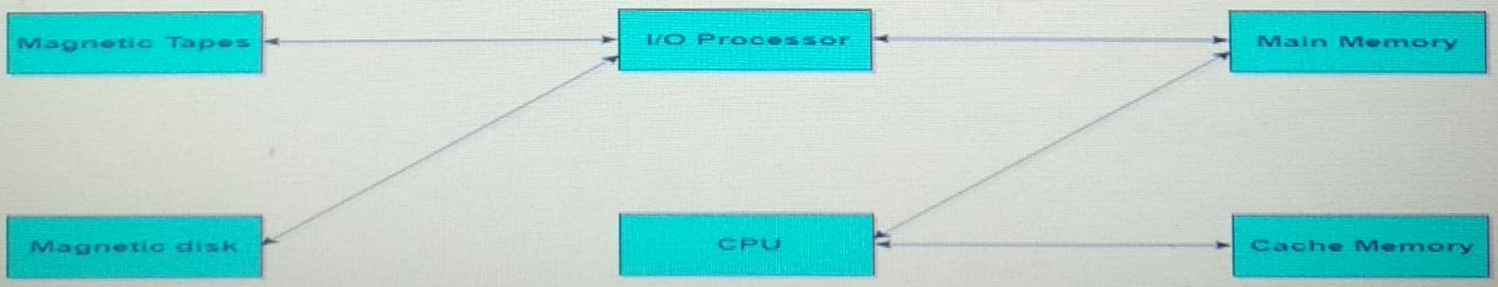
The total memory capacity of a computer can be visualized by hierarchy of components. The memory hierarchy system consists of all storage devices contained in a computer system from the slow Auxiliary Memory to fast Main Memory and to smaller Cache memory.

Auxillary memory access time is generally **1000 times** that of the main memory, hence it is at the bottom of the hierarchy.

The **main memory** occupies the central position because it is equipped to communicate directly with the CPU and with auxiliary memory devices through Input/output processor (I/O).

When the program not residing in main memory is needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space in main memory for other programs that are currently in use.

The **cache memory** is used to store program data which is currently being executed in the CPU. Approximate access time ratio between cache memory and main memory is about **1 to 7~10**



Memory Access Methods

Each memory type, is a collection of numerous memory locations. To access data from any memory, first it must be located and then the data is read from the memory location. Following are the methods to access information from memory locations:

1. **Random Access**: Main memories are random access memories, in which each memory location has a unique address. Using this unique address any memory location can be reached in the same amount of time in any order.
2. **Sequential Access**: This methods allows memory access in a sequence or in order.
3. **Direct Access**: In this mode, information is stored in tracks, with each track having a separate read/write head.

Main Memory

The memory unit that communicates directly within the CPU, Auxillary memory and Cache memory, is called main memory. It is the central storage unit of the computer system. It is a large and fast memory used to store data during computer operations. Main memory is made up of **RAM** and **ROM**, with RAM integrated circuit chips holing the major share.

- RAM: Random Access Memory
  - **DRAM**: Dynamic RAM, is made of capacitors and transistors, and must be refreshed every 10~100 ms. It is slower and cheaper than SRAM.
  - **SRAM**: Static RAM, has a six transistor circuit in each cell and retains data, until powered off.
  - **NVRAM**: Non-Volatile RAM, retains its data, even when turned off. Example: Flash memory.
- ROM: Read Only Memory, is non-volatile and is more like a permanent storage for information. It also stores the **bootstrap loader** program, to load and start the operating system when computer is turned on. **PROM**(Programmable ROM), **EPROM**(Erasable PROM) and **EEPROM**(Electrically Erasable PROM) are some commonly used ROMs.

Auxiliary Memory

Devices that provide backup storage are called auxiliary memory. **For example:**Magnetic disks and tapes are commonly used auxiliary devices. Other devices used as auxiliary memory are magnetic drums, magnetic bubble memory and optical disks.

It is not directly accessible to the CPU, and is accessed using the Input/Output channels.

Cache Memory

The data or contents of the main memory that are used again and again by CPU, are stored in the cache memory so that we can easily access that data in shorter time.

Whenever the CPU needs to access memory, it first checks the cache memory. If the data is not found in cache memory then the CPU moves onto the main memory. It also

transfers block of recent data into the cache and keeps on deleting the old data in cache to accomodate the new one.

Hit Ratio

The performance of cache memory is measured in terms of a quantity called **hit ratio**. When the CPU refers to memory and finds the word in cache it is said to produce a **hit**. If the word is not found in cache, it is in main memory then it counts as a **miss**.

The ratio of the number of hits to the total CPU references to memory is called hit ratio.

```
Hit Ratio = Hit/(Hit + Miss)
```

Associative Memory

It is also known as **content addressable memory (CAM)**. It is a memory chip in which each bit position can be compared. In this the content is compared in each bit cell which allows very fast table lookup. Since the entire chip can be compared, contents are randomly stored without considering addressing scheme. These chips have less storage capacity than regular memory chips.

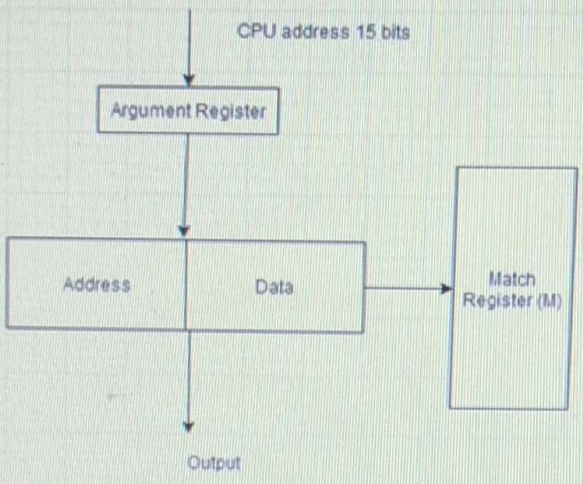
Memory Mapping

The transformation of data from main memory to cache memory is called mapping. There are 3 main types of mapping:

- Associative Mapping
- Direct Mapping
- Set Associative Mapping

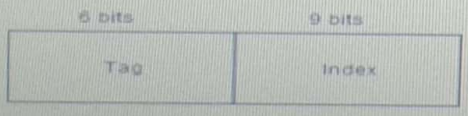
Associative Mapping

The associative memory stores both address and data. The address value of 15 bits is 5 digit octal numbers and data is of 12 bits word in 4 digit octal number. A CPU address of 15 bits is placed in **argument register** and the associative memory is searched for matching address.



Direct Mapping

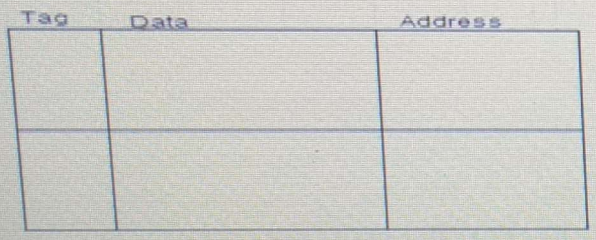
The CPU address of 15 bits is divided into 2 fields. In this the 9 least significant bits constitute the **index** field and the remaining 6 bits constitute the **tag** field. The number of bits in index field is equal to the number of address bits required to access cache memory.



Set Associative Mapping

The disadvantage of direct mapping is that two words with same index address can't reside in cache memory at the same time. This problem can be overcome by set associative mapping.

In this we can store two or more words of memory under the same index address. Each data word is stored together with its tag and this forms a set.



Replacement Algorithms

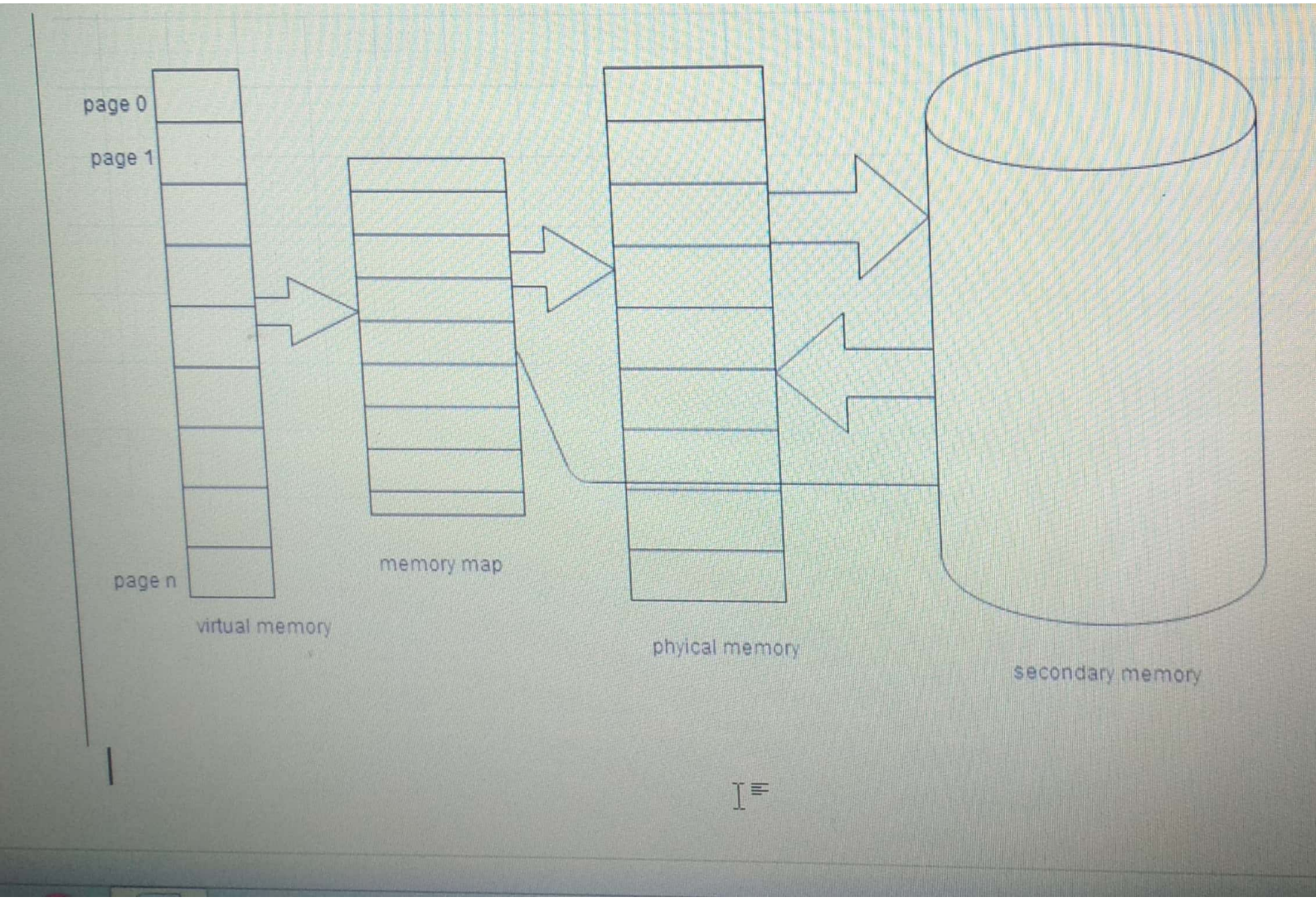
Data is continuously replaced with new data in the cache memory using replacement algorithms. Following are the 2 replacement algorithms used:

- FIFO - First in First out. Oldest item is replaced with the latest item.
- LRU - Least Recently Used. Item which is least recently used by CPU is removed.

Virtual Memory

Virtual memory is the separation of logical memory from physical memory. This separation provides large virtual memory for programmers when only small physical memory is available.

Virtual memory is used to give programmers the illusion that they have a very large memory even though the computer has a small main memory. It makes the task of programming easier because the programmer no longer needs to worry about the amount of physical memory available.



CPU BASICS:

The central processing unit (CPU) in your computer does the computational work—running programs, basically. But modern CPUs offer features like multiple cores and hyper-threading. Some PCs even use multiple CPUs. We're here to help sort it all out.

The clock speed for a CPU used to be enough when comparing performance. Things aren't so simple anymore. A CPU that offers multiple cores or hyper-threading may perform significantly better than a single-core CPU of the same speed that doesn't feature hyper-threading. And PCs with multiple CPUs can have an even bigger advantage. All of these features are designed to allow PCs to more easily run multiple processes at the same time—increasing your performance when multitasking or under the demands of powerful apps like video encoders and modern games. So, let's take a look at each of these features and what they might mean to you.

Hyper-Threading

Hyper-threading was Intel's first attempt to bring parallel computation to consumer PCs. It debuted on desktop CPUs with the Pentium 4 HT back in 2002. The Pentium 4's of the day featured just a single CPU core, so it could really only perform one task at a time—even if it was able to switch between tasks quickly enough that it seemed like multitasking. Hyper-threading attempted to make up for that.

A single physical CPU core with hyper-threading appears as two logical CPUs to an

A single physical CPU core with hyper-threading appears as two logical CPUs to a operating system. The CPU is still a single CPU, so it's a little bit of a cheat. While th operating system sees two CPUs for each core, the actual CPU hardware only has single set of execution resources for each core. The CPU pretends it has more cores than it does, and it uses its own logic to speed up program execution. In other words, the operating system is tricked into seeing two CPUs for each actual CPU core.

Hyper-threading allows the two logical CPU cores to share physical execution resources. This can speed things up somewhat—if one virtual CPU is stalled and waiting, the other virtual CPU can borrow its execution resources. Hyper-threading can help speed your system up, but it's nowhere near as good as having actual additional cores.

Thankfully, hyper-threading is now a "bonus." While the original consumer processors with hyper-threading only had a single core that masqueraded as multiple cores, modern Intel CPUs now have both multiple cores and hyper-threading technology. Your dual-core CPU with hyper-threading appears as four cores to your operating system, while your quad-core CPU with hyper-threading appears as eight cores. Hyper-

threading is no substitute for additional cores, but a dual-core CPU with hyper-threading should perform better than a dual-core CPU without hyper-threading.

Multiple Cores

Originally, CPUs had a single core. That meant the physical CPU had a single central processing unit on it. To increase performance, manufacturers add additional "cores," or central processing units. A dual-core CPU has two central processing units, so it appears to the operating system as two CPUs. A CPU with two cores, for example, could run two different processes at the same time. This speeds up your system, because your computer can do multiple things at once.

Unlike hyper-threading, there are no tricks here — a dual-core CPU literally has two central processing units on the CPU chip. A quad-core CPU has four central processing units, an octa-core CPU has eight central processing units, and so on.

This helps dramatically improve performance while keeping the physical CPU unit small so it fits in a single socket. There only needs to be a single CPU socket with a single CPU unit inserted into it—not four different CPU sockets with four different CPUs, each needing their own power, cooling, and other hardware. There's less latency because the cores can communicate more quickly, as they're all on the same chip.

Windows' Task Manager shows this fairly well. Here, for example, you can see that this system has one actual CPU (socket) and four cores. Hyperthreading makes each core look like two CPUs to the operating system, so it shows 8 logical processors.

Multiple CPUs

Most computers only have a single CPU. That single CPU may have multiple cores or hyper-threading technology—but it's still only one physical CPU unit inserted into a single CPU socket on the motherboard.

Before hyper-threading and multi-core CPUs came around, people attempted to add additional processing power to computers by adding additional CPUs. This requires a motherboard with multiple CPU sockets. The motherboard also needs additional hardware to connect those CPU sockets to the RAM and other resources. There's a lot of overhead in this kind of setup. There's additional latency if the CPUs need to communicate with each other, systems with multiple CPUs consume more power, and the motherboard needs more sockets and hardware.

Systems with multiple CPUs aren't very common among home-user PCs today. Even a high-powered gaming desktop with multiple graphics cards will generally only have a

single CPU. You'll find multiple CPU systems among supercomputers, servers, and similar high-end systems that need as much number-crunching power as they can get.

The more CPUs or cores a computer has, the more things it can do at once, helping improve performance on most tasks.Most computers now have CPUs with multiple cores—the most efficient option we've discussed. You'll even find CPUs with multiple cores on modern smartphones and tablets. Intel CPUs also feature hyper-threading, which is kind of a bonus. Some computers that need a large amount of CPU power may have multiple CPUs, but it's much less efficient than it sounds.

Multiple-Processor Scheduling in Operating System

In multiple-processor scheduling **multiple CPU's** are available and hence **Load Sharing** becomes possible. However multiple processor scheduling is more **complex** as compared to single processor scheduling. In multiple processor scheduling there are cases when the processors are identical i.e. HOMOGENEOUS, in terms of their functionality, we can use any processor available to run any process in the queue.

Approaches to Multiple-Processor Scheduling –

One approach is when all the scheduling decisions and I/O processing are handled by a single processor which is called the **Master Server** and the other processors executes only the **user code**. This is simple and reduces the need of data sharing. This entire scenario is called **Asymmetric Multiprocessing**.
A second approach uses **Symmetric Multiprocessing** where each processor is **self scheduling**. All processes may be in a common ready queue or each processor may have its own private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.

Processor Affinity –

Processor Affinity means a processes has an **affinity** for the processor on which it is currently running.
When a process runs on a specific processor there are certain effects on the cache memory. The data most recently accessed by the process populate the cache for the processor and as a result successive memory access by the process are often satisfied in the cache memory. Now if the process migrates to another processor, the contents of the cache memory must be invalidated for the first processor and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most of the SMP(symmetric multiprocessing) systems try to avoid

migration of processes from one processor to another and try to keep a process running on the same processor. This is known as processor affinity.
There are two types of processor affinity:

1. **Soft Affinity** – When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.
2. **Hard Affinity** – Hard Affinity allows a process to specify a subset of processors on which it may run. Some systems such as Linux implements soft affinity but also provide some system calls like *sched_setaffinity()* that supports hard affinity.

Load Balancing –

Load Balancing is the **phenomena** which keeps the **workload** evenly **distributed** across all processors in an SMP system. Load balancing is necessary only on systems where each processor has its own private queue of process which are eligible to execute. Load balancing is unnecessary because once a processor becomes idle it immediately extracts a runnable process from the common run queue. On SMP(symmetric multiprocessing), it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor else one or more processor will sit idle while other processors have high workloads along with lists of processors awaiting the CPU.

There are two general approaches to load balancing :

1. **Push Migration –** In push migration a task routinely checks the load on each processor and if it finds an imbalance then it evenly distributes load on each processors by moving the processes from overloaded to idle or less busy processors.
2. **Pull Migration –** Pull Migration occurs when an idle processor pulls a waiting task from a busy processor for its execution.

Multicore Processors –

In multicore processors **multiple processor** cores are places on the same physical chip. Each core has a register set to maintain its architectural state and thus appears to the operating system as a separate physical processor. **SMP systems** that use multicore processors are faster and consume **less power** than systems in which each processor has its own physical chip.
However multicore processors may **complicate** the scheduling problems. When processor accesses memory then it spends a significant amount of time waiting for the data to become available. This situation is called **MEMORY STALL**. It occurs for various reasons such as cache miss, which is accessing the data that is not in the cache memory. In such cases the processor can spend upto fifty percent of its time waiting for data to become available from the memory. To solve this problem recent hardware designs have implemented multithreaded processor cores in which two or

more hardware threads are assigned to each core. Therefore if one thread stalls while waiting for the memory, core can switch to another thread.
There are two ways to multithread a processor :

1. **Coarse-Grained Multithreading –** In coarse grained multithreading a thread executes on a processor until a long latency event such as a memory stall occurs, because of the delay caused by the long latency event, the processor must switch to another thread to begin execution. The cost of switching between threads is high as the instruction pipeline must be terminated before the other thread can begin execution on the processor core. Once this new thread begins execution it begins filling the pipeline with its instructions.
2. **Fine-Grained Multithreading –** This multithreading switches between threads at a much finer level mainly at the boundary of an instruction cycle. The architectural design of fine grained systems include logic for thread switching and as a result the cost of switching between threads is small.

Virtualization and Threading –

In this type of **multiple-processor** scheduling even a single CPU system acts like a multiple-processor system. In a system with Virtualization, the virtualization presents one or more virtual CPU to each of virtual machines running on the system and then schedules the use of physical CPU among the virtual machines. Most virtualized environments have one host operating system and many guest operating systems. The host operating system creates and manages the virtual machines. Each virtual machine has a guest operating system installed and applications run within that guest.Each guest operating system may be assigned for specific use cases,applications or users including time sharing or even real-time operation. Any guest operating-system scheduling algorithm that assumes a certain amount of progress in a given amount of time will be negatively impacted by the virtualization. A time sharing operating system tries to allot 100 milliseconds to each time slice to give users a reasonable response time. A given 100 millisecond time slice may take much more than 100 milliseconds of virtual CPU time. Depending on how busy the system is, the time slice may take a second or more which results in a very poor response time for users logged into that virtual machine. The net effect of such scheduling layering is that individual virtualized operating systems receive only a portion of the available CPU cycles, even though they believe they are receiving all cycles and that they are scheduling all of those cycles.Commonly, the time-of-day clocks in virtual machines are incorrect because timers take no longer to trigger than they would on dedicated CPU's.
Virtualizations can thus undo the good scheduling-algorithm efforts of the operating systems within virtual machines.