III and IV Semester

Computer Science Stream B.Tech

Handbook

MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

ENGINEERING MATHEMATICS III AND IV SEMESTER

DISCRETE MATHEMATICS

LATTICE THEORY

Cartesian product: The Cartesian product of two sets A and B denoted $A \times B$ is the set of all ordered pairs of the form (a, b) where $a \in A$ and $b \in B$.

Binary relation: A binary relation from A to B is a subset of $A \times B$.

Reflexive relation: Let R be a binary relation on A. R is said to be reflexive relation if (a, a) is in R for every $a \in A$.

Symmetric relation: A binary relation R on a set A is said to be a symmetric relation if (a, b) in R implies that (b, a) is also in R.

Antisymmetric relation: Let R be a binary relation on A. R is said to be an antisymmetric relation if (a, b) in R implies that (b, a) is not in R unless $a = b$.

Transitive relation: Let R be a binary relation on A. R is said to be a transitive relation if (a, c) is in R whenever both (a, b) and (b, c) are in R.

Equivalence relation: A binary relation is said to an equivalence relation if it is reflexive, symmetric and transitive.

Partial ordering relation: A binary relation is said to be a partial ordering relation if it is reflexive, antisymmetric and transitive.

Partially ordered set (poset): Set A together with a partial ordering relation R on A is called a partially ordered set and is denoted by $(A, \leq)$.

Chain: Let $(A, \leq)$ be a partially ordered set. A subset of A is called a chain if every two elements in the subset are related.

Antichain: Let $(A, \leq)$ be a partially ordered set. A subset of A is called an antichain if no two elements in the subset are related.

Totally ordered set: A partially ordered set $(A, \leq)$ is called a totally ordered set if A is a chain and the binary relation is called a total ordering relation.

Maximal element: Let $(A, \leq)$ be a partially ordered set. An element a in A is called a maximal element if for no b in A, $a \neq b, a \leq b$.

Minimal element: Let $(A, \leq)$ be a partially ordered set. An element a in A is called a minimal element if for no b in A, $a \neq b, b \leq a$.

Upper bound: Let $(A, \leq)$ be a partially ordered set. An element = is said to be an upper bound of a and b if $a \leq c$ and $b \leq c$. An element c is said to be least upper bound of a and b if c is an upper bound of a and b, and if there is no other upper bound d of a and b such that $d \leq c$.

Universal upper bound: An element a in a lattice $(A, \leq)$ is called a universal upper bound if for every element b in A, $b \leq a$. It is unique if it exists and is denoted by 1.

Lower bound: Let $(A, \leq)$ be a partially ordered set. An element c is said to be a lower bound of a and b if $c \leq a$ and $c \leq b$. An element c is said to be greatest lowerbound of a and b if c is a lower bound of a and b, and if there is no other lower bound d of a and b such that $c \leq d$.

Universal lower bound: An element a in a lattice $(A, \leq)$ is called a universal lower bound if for every element b in A, $a \leq b$. It is unique if it exists and is denoted by 0.

Lattice: A partially ordered set is said to be a lattice if every two elements in the set have a unique least upper bound and a unique greatest lower bound.
For any a and b in the lattice $(A, \leq)$, a $\leq$ a $\bigvee$ b and $a \bigwedge b \leq a$
For any a, b, c, d in a lattice $(A, \leq)$, if $a \leq b$ and $c \leq d$ then $a \bigvee c \leq b \bigvee d$ and $a \bigwedge c \leq b \bigwedge d$

Commutative property: For any a and b in a lattice $(A, \leq)$
$a \bigvee b = b \bigvee a$ and $a \bigwedge b = b \bigwedge a$

Associative property: For any a, b and c in a lattice $(A, \leq)$
a$\bigvee(b \bigvee c) = (a \bigvee b) \bigvee c$ and $a \bigwedge(b \bigwedge c) = (a \bigwedge b) \bigwedge c$

Idempotent property: For every a in a lattice $(A, \leq)$ $a \bigvee a = a$ and $a \bigwedge a = a$.

Absorption Property: For any a and b in a lattice $(A, \leq)$, $a \bigvee(a \bigwedge b) = a$ and $a \bigwedge(a \bigvee b) = a$

Cover: Let a and b be two elements in a lattice. Then a is said to cover b if $b < a$ and there is no element c such that $b < c < a$.

Atom: An element is called as an atom if it covers the universal lower bound.

Distributive lattice: A lattice $(A, \vee, \wedge)$ is said to be distributive if for all $a, b, c \in A$,

$$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$$
$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c).$$

Complement of an element: The complement of an element a of a lattice $(A, \vee, \wedge)$ with 0 and 1 is an element $b \in A$ such that $a \vee b = 1$ and $a \wedge b = 0$.

Complemented lattice: A lattice in which every element has a complement is called a complemented lattice.

Boolean lattice: A distributive, complemented lattice is called a Boolean lattice. In a such a lattice, every element a has a unique complement $\bar{a}$, and $^-$ is a unary operation on the lattice.

Boolean algebra: The algebraic structure $(A, \vee, \wedge, ^-)$ formed by a Boolean lattice is called a Boolean algebra.
A Boolean expression over $(\{0,1\}, \vee, \wedge)$ is said to be in **disjunctive normal form** if it is join of minterms.
A Boolean expression over $(\{0,1\}, \vee, \wedge)$ is said to be in **conjunctive normal form** if it is meet of maxterms.

COMBINATORICS

Addition Principle. If there are m ways of doing A and n ways of doing B, with no way of doing both simultaneously, then the number of ways of doing A **or** B is $m + n$.

Multiplication Principle. If there are m ways of doing A and n ways of doing B independently, then there are mn ways of doing A **and** B (or A followed by B).

Permutations and Combinations

The number of permutations of n distinct objects is $n! = n(n-1)(n-2) \times \cdots 3 \times 2 \times 1$.
The number of ways of selecting and arranging r distinct objects from a collection of n distinct objects is ${}^nP_r = \frac{n!}{(n-r)!}$.
The number of ways of selecting r distinct objects from a collection of n distinct objects is ${}^nC_r$ or $\binom{n}{r} = \frac{n!}{r!\,(n-r)!} = \frac{n(n-1)\cdots(n-r+1)}{r!}$.
The number of ways of selecting any number of distinct objects from a collection of n distinct objects is 2^n.
The number of permutations of n objects where n_1 of them are alike of the first kind, n_2 of them are alike of the second kind, …, n_k of them are alike of the k^{th} kind is $\frac{n!}{n_1!n_2!\cdots n_k!}$.
The number of permutations of r objects selected from n types of objects with unlimited repetition of each type is n^r.
The number of selections of r objects from n types of objects with unlimited repetition of each type is $\binom{n+r-1}{r} = \binom{n+r-1}{n-1}$.

Basic identities

1. $n! = n(n-1)!$
2. $\binom{n}{r} = \binom{n}{n-r}$
3. $\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$ for $n > r > 0$
4. $\sum_{r=0}^{n} \binom{n}{r} = 2^n$

Inclusion-Exclusion Principle

Let $a_1, a_2, \ldots, a_n$ be n properties. In a collection of N objects, let $N(a_i)$ denote the number of objects with property a_i, let $N(a_i a_j)$ denote the number of objects with both properties $N(a_i a_j)$, etc. Then the number of objects in the collection that do **not** have any of the properties $a_1, a_2, \ldots, a_n$ is

$$N(\overline{a_1}\,\overline{a_2}\cdots\overline{a_n}) = N - \sum_i N(a_i) + \sum_{i<j} N(a_i a_j) + \cdots + (-1)^k \sum_{i_1<i_2<\cdots<i_k} N(a_{i_1} a_{i_2} \cdots a_{i_k}) + \cdots + (-1)^n N(a_1 a_2 \cdots a_n).$$

Ordering of Permutations

Index sequence for k^{th} permutation of n distinct marks in lexicographical order: $c_{n-1}c_{n-2}\cdots c_1$ where

$$k - 1 = c_{n-1}(n-1)! + c_{n-2}(n-2)! + \cdots + c_1 1!$$

is the factorial base representation of $k - 1$.
Fike's sequence for k^{th} permutation of n distinct marks: $d_1 d_2 \cdots d_{n-1}$, where $d_i = i - c_i$, and

$$k - 1 = c_1 \frac{n!}{2!} + c_2 \frac{n!}{3!} + \cdots + c_{n-1} \frac{n!}{(n-1)!}.$$

Generating Functions

The ordinary generating function for the number of selections of r distinct objects out of n distinct objects is $(1+x)^n = \sum_{r=0}^{n} \binom{n}{r} x^r$.

The ordinary generating function for the number of selections of r objects from n types of objects with unlimited repetition is $(1-x)^{-n} = \sum_{r=0}^{\infty} \binom{n+r-1}{r} x^r$.

The exponential generating function for the number of permutations of n objects is $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$.

Partitions and Compositions

The number of compositions of n into k positive parts is $\binom{n-1}{k-1}$.

The number of compositions of n into any number of positive parts is 2^{n-1}.

The ordinary generating function for the number of unrestricted partitions of n is $(1-x)^{-1}(1-x^2)^{-1}(1-x^3)^{-1}\cdots$.

GRAPH THEORY

A graph G consists of a finite nonempty set $V = V(G)$ whose elements are called 'vertices' of G and a set $E = E(G)$ of unordered pairs of distinct vertices of $V(G)$ whose elements are called the 'edges' of G. A graph with p vertices and q edges is called a (p, q) graph.

The first theorem in graph theory due to Euler, popularly known as 'Hand shaking lemma'. It states that, "the sum of degrees of all the vertices in a graph is twice the number of edges".

There are several types of graphs namely: complete graph, regular graph, cycle graph, path graph, tree, bipartite graph etc.

Some of the preliminary terminologies to be noted are:

Distance: The distance $d(u, v)$ between the two vertices u and v in G is the length of a shortest path joining them if any, otherwise $d(u, v) = \infty$. In a connected graph, distance is a metric. That is, for all the vertices u, v, w

i. $d(u, v) \geq 0$ with $d(u, v) = 0$ if and only if $d(u, u) = 0$
ii. $d(u, v) = d(v, u)$
iii. $d(u, v) + d(v, w) \geq d(u, w)$

Geodesic: A shortest u-v path.

Girth: Girth $g(G)$ of a graph G is the length of the shortest cycle (if any) in G.

Circumference: Circumference $c(G)$ of a graph G is the length of the longest cycle (if any) in G.

Eccentricity: The eccentricity $e(v)$ of a vertex in a connected graph G is the distance from v to the vertex farthest from v in G. That is, $e(v) = \max_{u \in V(G)} \{d(v, u)\}$.

Radius: The radius $r(G)$ or $\mathrm{rad}(G)$ is the minimum eccentricity of the vertices, i.e. $\mathrm{rad}(G) = \min_{v \in V(G)} \{e(v)\}$.

Diameter: The diameter $\mathrm{diam}(G)$ is the maximum eccentricity of the vertices. In other words, the length of any longest geodesic. i.e., $\mathrm{diam}(G) = \max_{v \in V(G)} \{e(v)\}$.

Central vertex: A vertex v is a central vertex if $e(v) = \mathrm{rad}(G)$. And the set of all central vertices is called 'center' of the graph.

GROUP THEORY

Let G be a non-empty set and $*: G \times G \to G$ a binary operation on G. Then

1. Associativity axiom: $(a * b) * c = a * (b * c)$, for all $a, b, c \in G$.
2. Identity axiom: There exists an element $e \in G$ such that $a * e = e * a = a$, for all $a \in G$.
3. Inverse axiom: For $a \in G$, there corresponds an element $b \in G$ such that $a * b = b * a = e$.
4. Commutativity or Abelian axiom: $a * b = b * a$, for all $a, b \in G$.

In the above, if $(G,*)$ satisfies 1 then $(G,*)$ is a **semigroup**.
If $(G,*)$ satisfies 1 and 2 then $(G,*)$ is a **monoid**.
If $(G,*)$ satisfies 1, 2, and 3 then $(G,*)$ is a **group**.
If $(G,*)$ satisfies 1, 2, 3, and 4 then $(G,*)$ is a **commutative** or **Abelian group**.

Definitions
Let $(G,\cdot)$ be a group.
1. A non-empty subset of $H \subseteq G$ is a **subgroup** of G if $(H,\cdot)$ itself is a group. Then we write $H \leq G$.
2. If $H \leq G$, and $a \in G$, then $Ha = \{ha \mid h \in H\}$. Then Ha is a **right coset** of H in G. Similarly, $aH = \{ah \mid h \in H\}$ is a **left coset** of H in G.
3. The number of elements in G is the **order of the group** G, denoted $o(G)$ or $|G|$.
4. Let $a \in G$. The **order of the element** a is the least positive integer m such that $a^m = e$, denoted $o(a)$ or $|a|$.
5. Let $a \in$ G. Then $\langle a \rangle = \{a^i \mid i = 0, \pm 1, \pm 2, \ldots\}$ is the **cyclic subgroup** of G generated by a.
6. A subgroup N of G is a **normal subgroup** of G if for every $g \in G$ and every $n \in N$, $gng^{-1} \in N$.
7. The set $Z(G) = \{z \in G \mid xz = zx, \forall x \in G\}$ is the **center** of G.
8. Let $a \in G$. Then $N(a) = \{x \in G \mid ax = xa\}$ is the **normaliser** of a.
9. Let $(H,\circ)$ also be a group. Then a **group homomorphism** from G to H is a function $f: G \to H$ such that for all $x, y \in G$, $f(xy) = f(x) \circ f(y)$.
10. Let $f: G \to H$ be a group homomorphism. Then the **image** of f is $\text{im}\, f = \{ f(x) \mid x \in G \} \leq H$ and the **kernel** of f is $\ker f = \{x \in G \mid f(x) = e_H\} \leq G$ where e_H is the identity element of H.

Examples of Groups
1. $(\mathbb{Z}, +)$ – Group of integers under addition
2. $(\mathbb{Q}, +)$ – Group of rational numbers under addition
3. $(\mathbb{R}, +)$ – Group of real numbers under addition
4. $(\mathbb{C}, +)$ – Group of complex numbers under addition
5. $\mathbb{Q}^\times$ – Group of non-zero rational numbers under multiplication
6. $\mathbb{R}^\times$ – Group of non-zero real numbers under multiplication
7. $\mathbb{C}^\times$ – Group of non-zero complex numbers under multiplication
8. $\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$ – Group of integers modulo n under addition modulo n
9. $\{1, \omega, \omega^2, \ldots, \omega^{n-1}\}$, where $\omega = e^{\frac{2i\pi}{n}}$ – Group of complex n^{th} roots of unity under multiplication
10. S_n – Group of all permutations of $\{1, 2, \ldots, n\}$ under composition of permutations
11. $\text{GL}_n(\mathbb{R})$ – Group of $n \times n$ invertible real matrices

Basic Results
Let $(G,\cdot)$ be any group.
Uniqueness of identity: G has a unique identity element.
1. **Uniqueness of inverses:** Every element $x \in G$ has a unique inverse $x^{-1} \in G$, and $(x^{-1})^{-1} = x$.
2. **Shoe-sock property:** $\forall x, y \in G, (xy)^{-1} = y^{-1}x^{-1}$.
3. **Cancellation laws**: Let $x, y \in G$. If $\exists a \in G$ such that $ax = ay$, then $x = y$. If $\exists b \in G$ such that $xb = yb$, then $x = y$.
4. If G is finite of order n, then $\forall x \in G, x^n = e$.
5. If $f: G \to H$ is a homomorphism, then $\ker f$ is an normal subgroup of G
6. $Z(G)$ is a normal subgroup of G.

PROPOSITIONAL CALCULUS

Implications

I_1: $P \wedge Q \Rightarrow P$(Simplification)	I_8: $\neg(P \rightarrow Q) \Rightarrow Q$	
I_2: $P \wedge Q \Rightarrow Q$(Simplification)	I_9: $P, Q \Rightarrow P \wedge Q$	
I_3: $P \Rightarrow P \vee Q$(Addition)	I_{10}: $\neg P, P \vee Q \Rightarrow Q$	(Disjunctive syllogism)
I_4: $Q \Rightarrow P \vee Q$(Addition)	I_{11}: $P, P \rightarrow Q \Rightarrow Q$	(Modus ponens)
I_5: $\neg P \Rightarrow P \rightarrow Q$	I_{12}: $\neg Q, P \rightarrow Q \Rightarrow \neg P$	(Modus tollens)
I_6: $Q \Rightarrow P \rightarrow Q$	I_{13}: $P \rightarrow Q, Q \rightarrow R \Rightarrow P \rightarrow R$	(Hypothetical syllogism)
I_7: $\neg(P \rightarrow Q) \Rightarrow P$	I_{14}: $P \vee Q, P \rightarrow R, Q \rightarrow R \Rightarrow R$	(Dilemma)

Equivalences

E_1: $\neg\neg P \Leftrightarrow P$	E_{12}: $R \vee (P \wedge \neg P) \Leftrightarrow R$
E_2: $P \wedge Q \Leftrightarrow Q \wedge P$	E_{13}: $R \wedge (P \vee \neg P) \Leftrightarrow R$
E_3: $P \vee Q \Leftrightarrow Q \vee P$	E_{14}: $R \vee (P \vee \neg P) \Leftrightarrow \mathbf{T}$
E_4: $(P \wedge Q) \wedge R \Leftrightarrow P \wedge (Q \wedge R)$	E_{15}: $R \wedge (P \wedge \neg P) \Leftrightarrow \mathbf{F}$
E_5: $(P \vee Q) \vee R \Leftrightarrow P \vee (Q \vee R)$	E_{16}: $P \rightarrow Q \Leftrightarrow \neg P \vee Q$
E_6: $P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$	E_{17}: $\neg(P \rightarrow Q) \Leftrightarrow P \wedge \neg Q$
E_7: $P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$	E_{18}: $P \rightarrow Q \Leftrightarrow \neg Q \rightarrow \neg P$
E_8: $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$	E_{19}: $P \rightarrow (Q \rightarrow R) \Leftrightarrow (P \wedge Q) \rightarrow R$
E_9: $\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$	E_{20}: $\neg(P \rightleftarrows Q) \Leftrightarrow P \rightleftarrows \neg Q$
E_{10}: $P \vee P \Leftrightarrow P$	E_{21}: $P \rightleftarrows Q \Leftrightarrow (P \rightarrow Q) \wedge (Q \rightarrow P)$
E_{11}: $P \wedge P \Leftrightarrow P$	E_{22}: $P \rightleftarrows Q \Leftrightarrow (P \wedge Q) \vee (\neg P \wedge \neg Q)$

PROBABILITY

Addition rule: If A and B are two events of an experiment having sample space S, then $P(A \cup B) = P(A) + P(B) - P(A \cap B)$.

The conditional probability of an event B, given that the event A already taken place is

$$P(B/A) = \frac{P(A \cap B)}{P(A)}, \quad P(A) > 0.$$

Baye's Theorem:

Let $B_1, B_2, \ldots B_k$ are the partitions of S with $P(B_i) \neq 0, i = 1, 2, \ldots k$ and A be any event of S, then

$$P(B_i/A) = \frac{P(A/B_i)P(B_i)}{\sum_{i=1}^{k} P(A/B_i)P(B_i)}.$$

The multiplicative rule of probability : $P(A \cap B) = \begin{cases} P(A)P(B|A), if\ P(A) \neq 0 \\ P(B)P(A|B), if\ P(B) \neq 0 \end{cases}$

If $P(A \cap B) = P(A)P(B)$, then A and B are independent.

Random Variable: Let S be the sample of space of a random experiment. Suppose with each element s of S, a unique real number X is associated according to some rule then X is called random variable. There are two types of random variable, i) Discrete and ii) Continuous.

Discrete Random Variable: A random variable X is said to be discrete, if the number of possible values of X is finite or countably infinite. The probability distribution function (pdf) is named as probability mass function (PMF). The Probability mass function is defined as, let X be a random variable, hence the range space R_X of consists of atmost a countably infinite number of values. The probability mass function is defined as

$p(x_i) = \Pr\{X = x_i\}$, satisfying the conditions i) $p(x_i) \geq 0\ for\ all\ i$

ii) $\sum_{i=1}^{k} p(x_i) = 1.$

Continuous Random Variable: A random variable X is said to be continuous if it can take all possible values between certain limits, here the range space of X is infinite. Therefore the probability distribution function named for such random variable is Probability density function (PDF), which is defined as the pdf of X is a function $f(x)$ satisfying the following properties i) $f(x) \geq 0$

ii) $\int_{-\infty}^{\infty} f(x)dx = 1$

iii) $\Pr\{a \leq X \leq b\} = \int_a^b f(x)dx$ for any a, b such that $-\infty < a < b < \infty$.

Note: 1. If X is a continuous random variable with pdf f(x), then

$P(a < X < b) = P(a \leq X < b) = P(a < X \leq b) = P(a \leq X \leq b) = \int_a^b f(x)dx.$

2. $P(X = a) = 0$, if X is a continuous random variable.

Cumulative distribution function: Let X be random variable (discrete or continuous), we define F to be the cumulative distribution function of a random variable X given by $F(x) = \Pr\{X \leq x\}$.

Case i) If X is discrete random variable then

$$F(t) = \Pr\{X \leq t\} = P(x_1) + P(x_2) + \cdots + P(t)$$

Case ii) If x is a continuous random variable then $F(x) = \Pr\{X \leq x\} = \int_{-\infty}^{x} f(x)dx$.

Two dimensional random variable: Let E be an experiment and S be a sample space associated with E. Let X=X(s) and Y=Y(s) be two functions each assigning a real number to each outcome s of S. We call (X, Y) to be two dimensional random variable.

Discrete 2D: If the possible values of (X, Y) are finite or countably infinite then (X, Y) is called discrete and it is defined as $P(x_i, y_j)$ satisfying the following condition,

i) $P(x_i, y_j) \geq 0$ and

ii) $\sum_{j=1}^{\infty}\sum_{i=1}^{\infty} P(x_i, y_j) = 1$. The function $P(x_i, y_j)$ defined is called as Joint probability distribution function (Jpdf).

Continuous 2D: If (X, Y) is a continuous random variable assuming all values in some region R of the Euclidean plane, then the Joint probability density function $f(x, y)$ is a function satisfying the following conditions

i) $f(x, y) \geq 0$ for all (x, y)ϵR

ii) $\iint f(x, y)dx\,dy = 1$ over the region R.

Marginal Probability distribution: The marginal probability distribution is defined as

Case i) In the discrete (X, Y), it is defined as $p(x_i) = P\{X = x_i\} = \sum_{j=1}^{\infty} P(x_i, y_j)$ is the marginal probability distribution of X. Similarly $q(y_j) = P\{Y = y_j\} = \sum_{i=1}^{\infty} P(x_i, y_j)$ is the marginal probability distribution of Y.

Case ii) In the continuous (X, Y), it is defined as the marginal probability function of X is defined as $g(x) = \int_{-\infty}^{\infty} f(x, y)dy$ and the marginal probability function of Y is defined as $h(y) = \int_{-\infty}^{\infty} f(x, y)dx$.

To calculate the conditional probability:

Case i) Discrete: Probability of x_i given y_j is defined as $= \frac{P(x_i, y_j)}{q(y_j)}$, $q(y_j) > 0$

Probability of y_j given x_i is defined as $= \frac{P(x_i, y_j)}{p(x_i)}$, $p(x_i) > 0$

Case ii) Continuous: The pdf of X for given Y=y is $= \frac{f(x,y)}{h(y)}$, $h(y) > 0$

The pdf off Y for given X=x is $= \frac{f(x,y)}{g(x)}$, $g(x) > 0$.

Independent Random variable: If X and Y are independent random variable then two dimensional random variable in case of discrete is defined as $P(x_i, y_j) = p(x_i).q(y_j)$ for all the values of i and j. In case of Continuous it is defined as $f(x,y) = g(x).h(y)$.

Mathematical Expectation: If X is a discrete random variable with pmf p(x), then the expectation of X is given by $E(X) = \sum_x xp(x)$, provided the series is absolutely convergent.
If X is continuous with pdf f(x), then the expectation of X is given by $E(X) = \int xf(x)dx$, provided $\int |x|f(x)dx < \infty$.
Variance of X is given by $V(X) = E(X - E(X))^2 = E(X^2) - \big(E(X)\big)^2$.
Chebyshev's inequality: Let x be random variable with mean μ and variance σ^2 then for any positive real number k(k>0)

$p\{|x-\mu| \geq k\} \leq \frac{\sigma^2}{k^2}$ (Upper bound)

$p\{|x-\mu| < k\} > 1 - \frac{\sigma^2}{k^2}$ (Lower bound)

Note: some other forms

1. $p\{|x-\mu| \geq k\sigma\} \leq \frac{1}{k^2}$ and $p\{|x-\mu| < k\sigma\} \geq 1$ (Upper bound)
2. $p\{|x-\mu| \geq \in\} \leq \frac{1}{\in^2} E(x-c)^2$ and $p\{|x-\mu| < \in\} \geq 1$

DISTRIBUTIONS:

Distribution	PMF/PDF	Mean	Variance
Binomial distribution $X \sim B(n,p)$	$P(x) = {}^nC_k p^k (1-p)^{n-k}$, $k = 0,1,2,...,n$	$E(x) = np$	$V(x) = np(1-p)$
Poisson's Distribution $X \sim P(\alpha)$	$P(x) = \frac{e^{-\alpha}\alpha^k}{k!}$, $k = 0,1,2,..., \alpha > 0$	$E(x) = \alpha = np$	$V(x) = \alpha = np$
Uniform Distribution $X \sim U(a,b)$	$f(x) = \begin{cases} \frac{1}{b-a}, & a < x < b \\ 0, & otherwise \end{cases}$	$E(x) = \frac{b+a}{2}$	$V(x) = \frac{(b-a)^2}{12}$
Normal Distribution $X \sim N(\mu, \sigma^2)$	$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-1}{2}\frac{(x-\mu)^2}{\sigma^2}}$, $-\infty < x, \mu < \infty$, $\sigma > 0$	$E(x) = \mu$	$V(x) = \sigma^2$
Exponential Distribution $X \sim E(\lambda)$	$f(x) = \begin{cases} \lambda e^{-\lambda x}, & x > 0 \\ 0, & otherwise \end{cases}$	$E(x) = \frac{1}{\lambda}$	$V(X) = \frac{1}{\lambda^2}$
Gamma Distribution $X \sim G(r,\alpha)$	$f(x) = \begin{cases} \frac{x^{r-1}e^{-\alpha x}\alpha^r}{\Gamma(r)}, & x > 0, \ \alpha, r > 0 \\ 0, & elsewhere \end{cases}$	$E(x) = \frac{r}{\alpha}$	$V(x) = \frac{r}{\alpha^2}$

Chi-square Distribution $X \sim \chi^2(n)$	$f(x) = \begin{cases} \dfrac{x^{\frac{n}{2}-1} e^{-\frac{x}{2}}}{\Gamma(n/2)2^{\frac{n}{2}}}, & x > 0 \\ 0, & elsewhere \end{cases}$	$E(x) = n$	$V(x) = 2n$

Uniform distribution on a two dimensional set: If R is a set in the two-dimensional plane, and R has a finite area, then we may consider the density function equal to the reciprocal of the area of R inside R, and equal to 0 otherwise:

$f(x,y) = \begin{cases} \frac{1}{area\ R}; & if\ (x,y) \in R \\ 0 & Otherwise \end{cases}$.

Covariance: $Cov(x,y) = E(xy) - E(x)E(y)$

Correlation coefficient: $\rho_{xy} = \rho = \dfrac{E(xy) - E(x)E(y)}{\sqrt{V(x)V(y)}}$

Properties:

1. $E(c) = c$, where c is a constant.
2. $V(c) = 0$, where c is a constant.
3. If $E(xy) = 0$ then x and y are orthogonal.
4. $V(Ax+b) = A^2V(x)$ when Ax+B is linear function of x.
5. If $\rho = 0$ then x and y are un correlated.
6. $V(Ax+by) = A^2V(x) + B^2V(y) + 2ABCOV(x,y)$

FUNCTIONS OF ONE DIMENSIONAL RANDOM VARIABLES

Let S be a sample space associated with a random experiment E, then it is known that a random variable X on S is a real valued function, i.e., $X: S \to R$, for each element s ∈ S, there is a real number associated.

Let X be a random variable defined on S. Let $y = H(x)$ is a real valued function of x. Then $Y = H(X)$ is a random variable on S. i.e., for each element s ∈ S, there is a real number associated, say $y = H(X(s))$. Here Y is called a function of the random variable X.

Notations:

1. R_X – the set of all possible values of the function X, called the **range space** of the random variable X.
2. R_Y – the set of all possible values of the function $Y = H(X)$, called the **range space** of the random variable Y.

Equivalent Events: Let C be an event associated with the range space R_Y. Let $B \subset R_X$ defined by $B = \{x \in R_X; H(x) \in C\}$, then B and C are called equivalent events.

Distribution function of functions of random variables:

Case 1: Let X be a discrete random variable with p.m.f. $p(x_i) = P(X = x_i)$ for $i = 1,2,3,...$ Let $Y = H(X)$ then Y is also a discrete random variable. If $Y = H(X)$ is a one to one function then the probability distribution of Y is as follows:

For the possible values of $y_i = H(x_i)$ for $i = 1,2,3,....$ The p.m.f. of $Y = H(X)$ is $q(y_i) = P(Y = y_i) = P(X = x_i) = p(x_i)$ for $i = 1,2,3,....$

Case 2: Let X be a discrete random variable with p.m.f. $p(x_i) = P(X = x_i)$ for $i = 1,2,3,...$ Let $Y = H(X)$ then Y is also a discrete random variable. Suppose tat for one value of $Y = y_i$ there corresponds several values of X say $x_{i_1}, x_{i_2}, ..., x_{i_j}, ...$ then the p.m.f. of of $Y = H(X)$ is

$$q(y_i) = P(Y = y_i) = p(x_{i_1}) + p(x_{i_2}) + \cdots + p\left(x_{i_j}\right) + \cdots$$

Case 3: Let X be a continuous random variable with p.d.f. $f(x)$. Let $Y = H(X)$ be a discrete random variable. Then if the set $\{Y = y_i\}$ is equivalent to an event $B \subseteq R_X$ then the p.m.f. of Y is

$$q(y_i) = P(Y = y_i) = \int_B f(x)\,dx$$

Case 4: Let X be a continuous random variable with p.d.f. $f(x)$. Let $Y = H(X)$ be a continuous random variable. Then the p.d.f. of Y, say g is obtained by the following procedure:

Step 1: Obtain the c.d.f. of Y, $G(y) = P(Y = y)$, by finding the event $A \subseteq R_X$, which is equivalent to the event $\{Y = y_i\}$.

Step 2: Differentiate $G(y)$ with respect to y to get $g(y)$.

Step 3: Determine those values of y in R_Y for which $g(y) > 0$.

Theorem: Let X be a continuous random variable with p.d.f. $f(x)$ where $f(x) > 0$ for $a < x < b$. Suppose that $Y = H(X)$ is strictly monotonic function on $[a, b]$. Then the p.d.f. of the random variable $Y = H(X)$ is given by

$$g(y) = f(x)\left|\frac{dx}{dy}\right|$$

If $Y = H(X)$ is strictly increasing then $g(y) > 0$ for $H(a) < y < H(b)$.
If $Y = H(X)$ is strictly decreasing then $g(y) > 0$ for $H(b) < y < H(a)$.

Theorem: Let X be a continuous random variable with p.d.f. $f(x)$. Let $Y = X^2$ then the p.d.f. of Y is

$$g(y) = \frac{1}{2\sqrt{y}}\left[f\left(\sqrt{y}\right) + f\left(-\sqrt{y}\right)\right]$$

FUNCTIONS OF TWO DIMENSIONAL RANDOM VARIABLES

Let (X, Y) be a two dimensional continuous random variable. Let $Z = H(X, Y)$ be a continuous function of X and Y then $Z = H(X, Y)$ is a continuous one dimensional random variable.

To find the p.d.f. of Z, we introduce another suitable random variable say,
$W = G(X, Y)$ and obtain the joint p.d.f. of the two dimensional random variable (Z, W), say $k(z, w)$. From this distribution, the p.d.f. of Z can be obtained by integrating k with respect to w.

Theorem: Suppose (X, Y) is a two dimensional continuous random variable with joint p.d.f. $f(x, y)$ defined on a region R of the XY-plane. Let $Z = H_1(X, Y)$ and $W = H_2(X, Y)$. Suppose that $H_1$and H_2 satisfies the following conditions;

(i) $z = H_1(x, y)$ and $w = H_2(x, y)$ may be uniquely solved for x, y in terms of z & w say, $x = G_1(z, w)$ and $y = G_2(z, w)$.

(ii) The partial derivatives $\frac{\partial x}{\partial z}, \frac{\partial x}{\partial w}, \frac{\partial y}{\partial z}, \frac{\partial y}{\partial w}$ exist and are continuous

Then the joint p.d.f. of (Z, W) say $k(z, w)$ is given by,

$$k(z, w) = f[G_1(z, w), G_2(z, w)]|J(z, w)|$$

where $J(z, w) = \begin{vmatrix} \frac{\partial x}{\partial z} & \frac{\partial x}{\partial w} \\ \frac{\partial y}{\partial z} & \frac{\partial y}{\partial w} \end{vmatrix}$ is called the Jacobian of the transformation $(x, y) \mapsto (z, w)$. Also, $k(z, w) > 0$ for those values of (z, w) corresponding to the values of (x, y) for which $f(x, y) > 0$.

MOMENT GENERATING FUNCTION (M.G.F.) OF ONE DIMENSIONAL RANDOM VARIABLES

Let X be any one dimensional random variable then the mathematical expectation $E(e^{tX})$ if exists then it is called the moment generating function (m.g.f.) of X. i.e., $M_X(t) = E(e^{tX})$

In particular, if X is discrete then, $M_X(t) = \sum_{i=1}^{i=\infty} e^{tx_i} P(X = x_i)$.

If X is continuous then, $M_X(t) = \int_{-\infty}^{\infty} e^{tx} f(x)\,dx$.

Properties of m.g.f.: Let X be any one dimensional random variable and $M_X(t)$ be the m.g.f. of X then

1. $M_X^n(0) = E(X^n)$ where $M_X^n(0)$ is the n[th] derivative of $M_X(t)$ at $t = 0$.
   i.e.; $M_X'(0) = E(X)$
   $M_X''(0) = E(X^2)$
2. $V(X) = M_X''(0) - \left(M_X'(0)\right)^2$
3. Let X be any one dimensional random variable and $M_X(t)$ be the m.g.f. of X. Let $Y = \alpha X + \beta$. Then the m.g.f. of Y is $M_Y(t) = e^{\beta t} M_X(\alpha t)$.
4. Suppose that X and Y are independent random variables. Let $Z = X + Y$. Let $M_X(t), M_Y(t)$ and $M_Z(t)$ be the m.g.f.'s of the random variables X, Y and Z respectively. Then $M_Z(t) = M_X(t)M_Y(t)$
5. Let $X_1, X_2, \dots, X_n$ be n independent random variables which follows a normal distribution $N(\mu_i, \sigma_i^2)$ for $i = 1,2,3,..,n$. Let $Z = X_1 + X_2 + \cdots + X_n$ then $Z \to N(\mu_1 + \mu_2 + \cdots + \mu_n,\ \sigma_1^2 + \sigma_2^2 + \cdots + \sigma_n^2)$.
6. Let $X_1, X_2, \dots, X_n$ be n independent random variables which follows a Poisson distribution with parameter α_i for $i = 1,2,..,n$. Let $Z = X_1 + X_2 + \cdots + X_n$ then Z has a Poisson distribution with parameter $\alpha = \alpha_1 + \alpha_2 + \cdots + \alpha_n$.
7. Let $X_1, X_2, \dots, X_k$ be k independent random variables which follows a Chi-square distribution with degrees of freedom n_i for $i = 1,2,3,..,k$. Let $Z = X_1 + X_2 + \cdots + X_k$ then Z has a Chi-square distribution with degrees of freedom $n = n_1 + n_2 + \cdots + n_k$.
8. Let $X_1, X_2, \dots, X_k$ be k independent random variables, each having distribution $N(0,1)$. Then $S = X_1^2 + X_2^2 + \cdots + X_k^2$ has a Chi-square distribution with degrees of freedom k.
9. Let $X_1, X_2, \dots, X_r$ be r independent random variables, each having exponential distribution with the same parameter α. Let $Z = X_1 + X_2 + \cdots + X_r$ then Z has a Gamma distribution with parameters α and r.
10. Let $X_1, X_2, \dots, X_n, \dots$ be a sequence of random variable with c.d.f.'s $F_1, F_2, \dots, F_n, \dots$ and m.g.f.'s $M_1, M_2, \dots, M_n, \dots$ Suppose that $\lim_{n\to\infty} M_n(t) = M(t)$, where $M(0) = 1$. Then $M(t)$ is the m.g.f. of the random variable X whose c.d.f is $F = \lim_{n\to\infty} F_n(t)$.

MGF of some standard distributions:

1. **Binomial Distributions:** $\boldsymbol{M_X(t)} = M_X(t) = (pe^t + q)^n$
2. **Poisson Distributions:** $\boldsymbol{M_X(t)} = e^{\propto(e^t - 1)}$
3. **Normal Distributions:** $\boldsymbol{M_X(t)} = e^{t\mu + \frac{\sigma^2 t^2}{2}}$
4. **Exponential Distributions:** $\boldsymbol{M_X(t)} = \frac{\propto}{\propto - t}$
5. **Gamma Distributions:** $\boldsymbol{M_X(t)} = \frac{\propto^r}{(\propto - t)^r}$
6. **Chi square Distributions:** $\boldsymbol{M_X(t)} = (1 - 2t)^{-n/2}$

SAMPLING

In statistical investigation, the characteristics of a large group of individuals (called population) is studied. Sampling is a study of the relationship between a population and samples drawn from it.
The population mean and the population variance are denoted by μ and σ^2 respectively.
Sample mean and sample variance: Let X be the random variable which denotes the population with mean μ and variance σ^2. Let $(X_1, X_2, \dots, X_n)$ be a random sample of size n from X. Then,

Sample mean, $\bar{X} = \frac{\sum_{i=1}^n X_i}{n}$ and

Sample variance, $s^2 = \frac{\sum_{i=1}^n (X_i - \bar{X})^2}{n}$

- If $X \to N(\mu, \sigma^2)$ then $\bar{X}$ and s^2 are independent random variables.
- Let X be the random variable with $E(X) = \mu$ and $V(X) = \sigma^2$. Let $(X_1, X_2, \dots, X_n)$ be a random sample of size n from X. Then, $E(\bar{X}) = \mu$ and $V(\bar{X}) = \frac{\sigma^2}{n}$.
- Let $X \to N(\mu, \sigma^2)$ then $\bar{X} \to N(\mu, \frac{\sigma^2}{n})$ and $s^2 \to \chi^2(n-1)$.

Central Limit Theorem: Let $X_1, X_2, \dots, X_n$ be n independent random variables all of which have the same distribution. Let $\mu = E(X_i)$ and $\sigma^2 = V(X_i)$ be the common expectation and variance. Let $S = \sum_{i=1}^{n} X_i$ then $E(S) = n\mu$ and $V(S) = n\sigma^2$ then for large values of n, the random variable $T_n = \frac{S - E(S)}{\sqrt{V(S)}}$ has approximately the distribution $N(0,1)$.

Testing of Hypothesis:
The central limit theorem is used for testing of hypothesis. The purpose of hypothesis testing is to determine whether there is enough statistical evidence in favor of a certain belief, or hypothesis, about a parameter. For the hypothesis tests, the law of probability is assumed to be known, so the sampling distribution is perfectly known and we take a sample to define a decision criterion which help us to accept or reject the hypothesis.
Using the mean, we test a hypothesis H0. This is referred to as the null hypothesis. It is an assumption made on the probability distribution X. The alternate hypothesis is denoted by H1.
The error of the first kind is called Type I error. The error of the second kind is called Type II error. Thus a Type I error is an error in a statistical test which occurs when a false hypothesis is accepted (a false positive in terms of the null hypothesis) and a Type II error is an error in a statistical test which occurs when a true hypothesis is rejected (a false negative in terms of the null hypothesis). Note that the acceptance of H_1 when H_0 is true is called a Type I error. The probability of committing a Type I error is called the level

	Accept H_0	Reject H_0
H_0 is true	Right decision	Wrong decision (error of the first kind α)
H_0 is false	Wrong decision (error of the second kind β)	Right decision

of significance and is denoted by α. Also, the failure to reject H_0 when H_1 is true is called a Type II error. The probability of committing a Type II error is denoted by β. The probability $1 - \beta$ is called the power of a test; it is the probability of taking the correct action of rejecting the null hypothesis when it is false. By increasing n, we can improve the power of a test. For the same α and the same n, the power of test is also used to choose between different tests; a more powerful test is one that yields the correct action with greater frequency.
A statistical hypothesis test may return a value called the p-value. The p-value is the minimum probability of a Type I error with which H_0 can still be rejected. This is a quantity that we can use to interpret or quantify the result of the test and either reject or fail to reject the null hypothesis. This is done by comparing the p-value to a threshold value chosen beforehand called the significance level α.
If p-value > α: Fail to reject the null hypothesis (not significant result).
If p-value <= α: Reject the null hypothesis (significant result).
Some tests do not return a p-value. Instead, they might return a list of critical values and their associated significance levels, as well as a test statistic. The results are interpreted in a similar way. Instead of

comparing a single p-value to a pre-specified significance level, the test statistic is compared to the critical value at a chosen significance level.
If test statistic < critical value: Fail to reject the null hypothesis.
If test statistic >= critical value: Reject the null hypothesis.
Moving the critical value provides a trade-off between α and β. A reduction in β is always possible by increasing the size of the critical region, but this increases α. Likewise, reducing α is possible by decreasing the critical region. Note that α and β are related in such a way that decreasing one generally increases the other. This problem is solved with the help of sample size. Both α and β can be reduced simultaneously by increasing the sample size.
Consider H0: $\theta = \theta_0$ vs H_1: $\theta > \theta_0$. This is a one-tailed test with the critical region in the right-tail of the test statistic X. Another one-tailed test could have the form, H_0: $\theta = \theta_0$ vs H_1: $\theta < \theta_0$, in which the critical region is in the left-tail. In a two-tailed test we have : H_0: $\theta = \theta_0$ vs H_1: θ not equal to θ_0.
The first type of test is the most basic: testing the mean of a distribution in which we already know the population variance. Let μ and σ be the mean and standard deviation of X. We take from the population a sample of size n large enough. The sample mean is $\bar{x}$. If $\bar{x}$ is between $\mu - t_\infty . (\sigma/\sqrt{n})$ and $\mu + t_\infty . (\sigma/\sqrt{n})$ then H_0 is accepted. However if $\bar{x}$ is outside of those values, the null hypothesis is rejected (two-tailed test).
Our test statistic is

$$z = \frac{(\bar{x} - \mu)}{(\sigma/\sqrt{n})}$$

where n is the number of observations made when collecting the data for the study, and μ is the true mean when we assume the null hypothesis is true. So to test a hypothesis with given significance level α, we calculate the critical value of z (or critical values, if the test is two-tailed) and then check to see whether or not the value of the test statistic in is in our critical region. This is called a z-test. We are most often concerned with tests involving either α = .05 or α = .01. When we construct our critical region, we need to decide whether or not our hypotheses in question are one-tailed or two-tailed. If one-tailed, we reject the null hypothesis if $z \geq z\alpha$ (if the hypothesis is right-handed) or if $z \leq z\alpha$ (if the hypothesis is left-handed). If two-tailed, we reject the null hypothesis if $|z| \geq z\alpha/2$. If we do not know the population variance, and if n is large ($n \geq 30$) it suffices for most distributions commonly encountered to replace the unknown population variance with the modified definition of sample variance.
Critical Region: To construct a critical region of size α, we first examine our alternative hypothesis. If our hypothesis is one-tailed, our critical region is either $z \geq z\alpha$ (if the hypothesis is right-handed) or $z \leq -z\alpha$ (if the hypothesis is left-handed). If our hypothesis is two-tailed, then our critical region is $|z| \geq z\alpha/2$.
Student t distribution:
If we have a sample of size n from a normal distribution with mean μ and unknown variance, we study $t = (x-\mu)/(s/\sqrt{n})$ and compare this to the Student t-distribution with (n –1) degrees of freedom. If $n \geq 30$ then by the Central Limit Theorem we may instead compare it to the standard normal case.
Thus when we have a small sample size ($n < 30$) taken from a normal distribution of unknown variance, we use the t-test with (n – 1) degrees of freedom.
Critical Region**:** To construct a critical region of size α, we first examine our alternative hypothesis. If our hypothesis is one-tailed, our critical region is either $t \geq t\alpha, (n-1)$ (if the hypothesis is right-handed) or $t \leq -t\alpha, (n-1)$ (if the hypothesis is left-handed). If our hypothesis is two-tailed, then our critical region is $|t| \geq t\alpha/2, (n-1)$.
Chi-Square Test:
The chi-square test gives a p-value. The p-value tells us whether the test results are significant or not. The chi-square statistic is given by

$$\chi^2 = \sum_{j=1}^{n} \frac{(O_j - E_j)^2}{E_j}$$

where O_j = each Observed (actual) value, E_j= each Expected value.

Calculating the chi-square statistic and comparing it against a critical value from the chi-square distribution allows us to assess whether the observed cell counts in a table are significantly different from the expected cell counts.

Estimation:

Estimator (or estimate) $\hat{\theta}$ for the unknown parameter θ associated with the distribution of a random variable x is called unbiased estimator (or unbiased estimate) if $E(\hat{\theta}) = \theta$ for all θ.

An unbiased estimate of the variance σ^2 of a random variable based on a sample $x_1, \dots, x_n$ is $\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$.

Maximum likelihood estimate (M.L.E.):- Based on a random sample $x_1, \dots, x_n$ the M.L.E. $\hat{\theta}$ of θ is that value of θ which maximizes $L(x_1, \dots, x_n; \theta) = f(x_1; \theta)f(x_2; \theta) \cdots f(x_n; \theta)$ when $f(x; \theta)$ is either the p.d.f of x.

Confidence intervals:

Once the sample is observed and the sample mean computed to equal $\bar{x}$, this interval $\left[\bar{x} - z\alpha_{/2}\left(\frac{\sigma}{\sqrt{n}}\right), \bar{x} + z\alpha_{/2}\left(\frac{\sigma}{\sqrt{n}}\right)\right]$ is a known interval for the unknown mean μ.

For example $\bar{x} \pm 1.96\left(\frac{\sigma}{\sqrt{n}}\right)$ is 95% confidence interval for μ. The number $100(1-\alpha)\%$ or $1-\alpha$ is confidence coefficient.

Let a and b be constants selected for the random sample $x_1, \dots, x_n$. The confidence interval for the variance σ^2 based on the sample variance $S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$ with $100(1-\alpha)\%$

Confidence is given by $\left[\sqrt{\frac{(n-1)s^2}{b}}, \sqrt{\frac{(n-1)s^2}{a}}\right] = \left[\sqrt{\frac{(n-1)}{b}}s, \sqrt{\frac{(n-1)}{a}}s\right]$, if $P\left(a \leq \frac{(n-1)s^2}{\sigma^2} \leq b\right) = 1-\alpha$.

COMPUTER ORGANIZATION AND ARCHITECTURE

Binary, signed-integer representations

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

Format of typical instructions

RISC Style	**CISC Style**
Move *destination, source*	Move *destination, source*
Operation *destination, source1, source2*	Operation *destination, source*
Clear *destination*	Clear *destination*
Branch *condition, target*	Branch *condition, target*
Load *destination, source*	
Store *source, destination*	

Addressing modes

Name	**Assembler syntax**	**Addressing function**
Immediate	#Value	Operand= Value
Register	R_i	EA= R_i
Absolute	LOC	EA=LOC
Register Indirect	(R_i)	EA= [R_i]
Index	X(R_i)	EA=[R_i]+X
Base with index	(R_i, R_j)	EA=[R_i] + [R_j]
Base index plus constant	X(R_i, R_j)	EA=[R_i] + [R_j] +X
Auto increment	(R_i)+	EA= [R_i] + 1
Auto decrement	-(R_i)	EA= [R_i] -1
Relative addressing	X(PC)	EA=[PC]+X

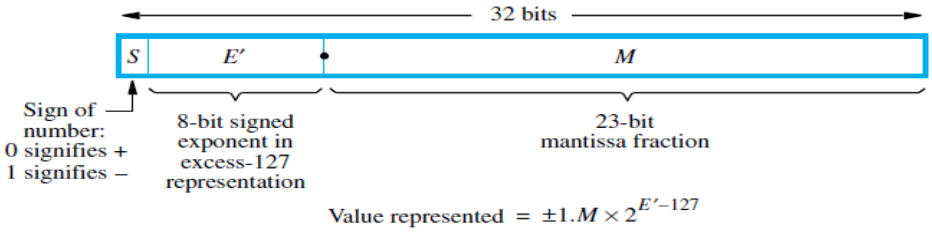
EA=Effective Address
Value= A signed number
X=Index Value

Commonly used flags

N (negative), Z (zero), V (overflow) and C (carry)

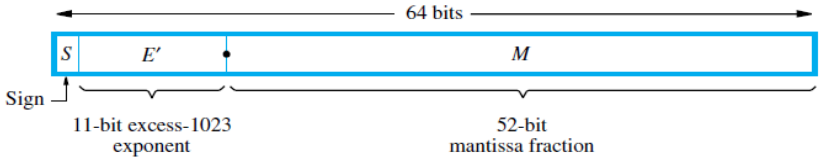
IEEE standard single precision and double precision floating-point formats

Single precision



Value represented $= \pm 1.M \times 2^{E'-127}$

Double precision

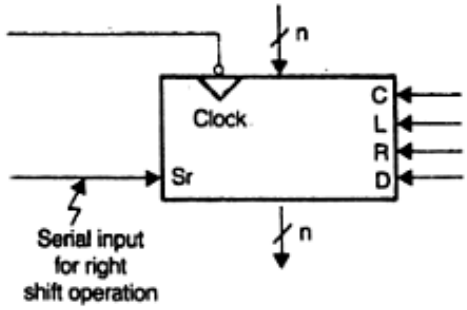


Value represented $= \pm 1.M \times 2^{E'-1023}$

Booth Multiplier Recoding Table

Multiplier		Version of multiplicand selected by bit i
Bit i	Bit $i-1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Characteristics of the components used in the Processing section

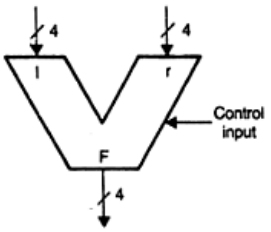


Block Diagram

C	L	R	D	Clock	Action
1	0	0	0	↓	Clear
0	1	0	0	↓	Load external data
0	0	1	0	↓	Right shift
0	0	0	1	↓	Decrement by one
0	0	0	0	↓	No change

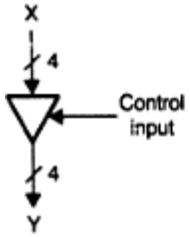
Truth Table

Storage Register



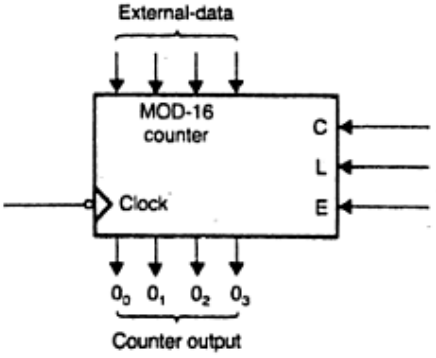
Control input	F
1	I + r
0	I − r

Adder/Subtractor



Control input	Y
1	X
0	High Z

Tristate Buffer



Block Diagram

C	L	E	Clock	Action
1	X	X	X	Clear
0	1	X	↓	Load external data
0	0	1	↓	Count up
0	0	0	↓	No operation

Truth Table

Mod 16 Counter

10 Steps for Hardwired Control

1. Define the task to be performed
2. Propose a trial processing section
3. Provide a Register Transfer Description algorithm based on the processing section outlined
4. Validate the algorithm using trial data
5. Describe the basic characteristics of the hardware elements to be used in the processing section
6. Complete the design of the processing section by establishing necessary control points
7. Propose the block diagram of the controller
8. Specify the state diagram of the controller
9. Specify the characteristics of the hardware elements to be used in the controller
10. Complete the controller design and draw a logic diagram of the final circuit

All microinstructions have 2 fields:

-Control field

-Next address field

Memory Basic Concepts

The maximum size of the memory that can be used in any computer is determined by the addressing scheme.

Eg:

Address	**Memory Locations**
32 Bit	2^{32} = 4G (Giga)

Hit Rate and Miss Penalty

The average access time experienced by the processor in a system with a single cache is

$$t_{ave} = hC + (1 - h)M$$

h → Cache hit ratio
C→ Cache access time
M→ Miss Penalty to transfer information from main memory to cache.
The average access time experienced by the processor in a system with two levels of caches is

$$t_{ave} = h_1C_1 + (1-h_1)\, h_2\, C_2 + (1-h_1)\,(1-h_2)M$$

h1 → Cache L1 hit ratio
h2 → Cache L2 hit ratio
C1→L1 access time
C2→ Miss Penalty to transfer information from L2 to L1
M→ Miss Penalty to transfer information from main memory to L2.

Vector (SIMD) Processing instructions

VectorAdd.S V*i*, V*j*, V*k*
VectorLoad.S V*i*, X(R*j*)
VectorStore.S V*i*, X(R*j*)

Where V*i*, V*j* and V*k* vector registers.
In the instruction X(R*j*) causes vector length *L* elements handled beginning at memory location X + [R*j*].

DATA STRUCTURES

typedef
typedef <existing_name> <alias_name>;
alias_name var_name;

Enumeration(enum)
enum tagname {value1, value2,…… , valueN};
enum tagname var_name;

Structure
```
struct [structure_tag] {
   member definition;
   member definition;
   ...
   member definition;
} [structure variables];
```

Union
```
union [union_tag] {
   member definition;
   member definition;
   ...
   member definition;
} [union variables];
```

Pointers

declaration	*Datatype *pointerVar*;
function returning pointer	*Datatype *func ( )*;
generic pointer type	*void **
null pointer constant	*NULL*
pointer dereferencing	**pointerVar*
address of variable	*&pointerVar*
structure member through pointer	*(*pointerVar).member* or *pointerVar->member*
address of an array element	*address = pointer + (offset * element_size);*

Dynamic Storage Allocation

memory allocation	*void* malloc (size_t size);*
contiguous memory allocation	*void* calloc (size_t count, size_t size);*
reallocation of memory	*void* realloc (void* ptr, size_t newSize);*
releasing memory	*void free (void* ptr);*

Pointers(new and delete):

Syntax:	datatype *pointername;	pointername=new datatype;
For array:	datatype *pointername;	pointername=new datatype[size];
Syntax:	delete pointername;	
For array:	delete[] pointername;	

Stacks

ADT **Stack** is
Objects: a finite ordered list with zero or more elements.

Functions:
for all stack € Stack, item € element, maxStackSize € positive integer

Stack createS(maxStackSize)::=
create an empty stack whose maximum size is maxStackSize

Boolean IsFull(stack, maxStackSize)::=
if (number of elements in stack == maxStackSize)
return TRUE
else return FALSE

stack push (stack,item)::=
if (isFull(stack)) stackFull
else insert item into top of stack and return

Boolean IsEmpty(stack)::=
if(stack==createS(maxStackSize))
return TRUE
else return FALSE

element pop(stack)::=
if (IsEmpty(stack)) return
else remove and return the element at the top of the stack.

Queues

ADT **Queue** is
Objects: a finite ordered list with zero or more elements.
Functions:
for all queue € Queue, item € element, maxQueueSize € positive integer

Queue createQ(maxQueueSize)::=
create an empty queue whose maximum size is maxQueueSize

Boolean IsFull(queue, maxQueueSize)::=
if (number of elements in queue == maxQueueSize)
return TRUE
else return FALSE

Queue Add (queue,item)::=
if (isFull(queue)) queueFull
else insert item into rear of queue and return queue

Boolean IsEmpty(queue)::=
if(queue==CreateQ(maxQueueSize))
return TRUE
else return FALSE

element DeleteQ(queue)::=
    if (IsEmpty(queue)) return
    else remove and return the element at front of queue

Sparse Matrix

ADT **SparseMatrix** is
Objects: a set of triples <row,column,value>, where row and column are integers and form a unique combination, and value comes from the set item.
Functions:
    for all a,b € sparse matrix, x € item, i, ,j, maxCol, maxRow € index

SparseMatrix Create(maxRow, maxCol)::=
    return a sparse matrix that can hold upto max items = maxRow X maxCol

Sparsematrix Transpose(a) ::=
    return the matrix produced by interchanging the row and column value of every triple.

SparseMatrix Add(a,b)::=
    if the dimension of a,b are same return the matrix produced by adding the corresponding items.
    else return error

SparseMatirx Multiply(a,b)::=
    if number of columns in a equals number of rows in b return the matrix D produced by the formula d[i][j]=Σ(a[i][k] . b[k][j]) where d(i,j) is the i,jth element.
    else return error

Linked Lists

Linked List Operations
Suppose head is pointer pointing to the first node in the linked list, some of the basic operations on linked list are:
InsertEnd(head, item) ::=
Inserts an element to the end of the list and **returns** the head pointer.

InsertBeg(head, item) ::=
Inserts an element before the **head** node of the list and **returns** the new head pointer.

InsertBefore(head, pos) ::=
Inserts an element before the **given position** in the list and **returns** the head pointer.

InsertAfter(head, pos) ::=
Inserts an element after the **given position** in the list and **returns** the head pointer.
remove(node) ::=
    if the list is empty
    return error
    else remove the specific node from the list.

Binary Tree
Abstract data type *Binary_Tree*
Structure *Binary_Tree(abbreviated BinTree) is*

Objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree.*

Functions:

for all *bt, bt1, bt2* ∈ *BinTree, item* ∈ *element*

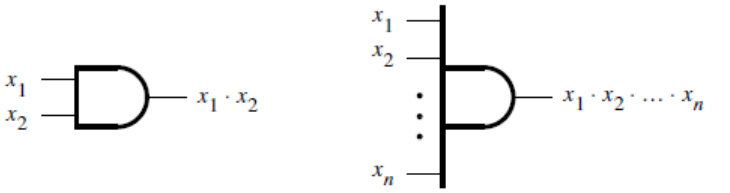
BinTree Create()	::=	creates an empty binary tree.
Boolean IsEmpty(*bt*)	::=	if (*bt* == empty binary tree) **return** *TRUE* **else return** *FALSE*
BinTree MakeBT(*bt1*, *item*, *bt2*)	::=	**return** a binary tree whose left subtree is *bt1*, whose right subtree is *bt2*, and whose root node contains the data *item.*
BinTree Lchild(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the left subtree of *bt*.
element Data*(bt)*	::=	*if (*IsEmpty*(bt))* **return** error **else return** the data in the root node of *bt.*
BinTree Rchild(*bt*)	::=	if (IsEmpty(*bt*)) **return** error **else return** the right subtree of *bt*.

Searching and Sorting

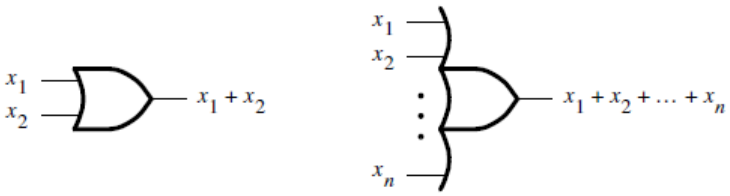
Algorithm	In place	Stable	Best	Average	Worst	Remarks
selection sort	✓		$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	n exchanges; quadratic in best case
insertion sort	✓	✓	n	$\frac{1}{4} n^2$	$\frac{1}{2} n^2$	use for small or partially-sorted arrays
bubble sort	✓	✓	n	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	rarely useful; use insertion sort instead
shellsort	✓		$n \log_3 n$	unknown	$c\, n^{3/2}$	tight code; subquadratic
mergesort		✓	$\frac{1}{2} n \lg n$	$n \lg n$	$n \lg n$	$n \log n$ guarantee; stable
quicksort	✓		$n \lg n$	$2 n \ln n$	$\frac{1}{2} n^2$	$n \log n$ probabilistic guarantee; fastest in practice
heapsort	✓		n †	$2 n \lg n$	$2 n \lg n$	$n \log n$ guarantee; in place

DIGITAL SYSTEM DESIGN

Basic Gates



(a) AND gates



(b) OR gates

(c) NOT gate

Boolean Algebra

x · y = y · x Commutative
x + y = y + x

x · (y · z) = (x · y) · z Associative
x + (y + z) = (x + y) + z

x · (y + z) = x · y + x · z Distributive
x + y · z = (x + y) · (x + z)

x + x · y = x Absorption
x · (x + y) = x

x · y + x · y = x Combining
(x + y) · (x + y) = x

x · y = x + y DeMorgan's theorem
x + y = x · y

The General Form of a Module

```
module module name [(port name{, port name})];
        [parameter declarations]
        [input declarations]
        [output declarations]
        [inout declarations]
        [wire or tri declarations]
        [reg or integer declarations]
        [function or task declarations]
        [assign continuous assignments]
        [initial block]
        [always blocks]
        [gate instantiations]
        [module instantiations]
Endmodule
```

Designing subcircuits in Verilog

```
module_name [#(parameter overrides)] instance_name (
.port_name ( [expression] ) {, .port_name ( [expression] )} );
```

Parameters

```
parameter n = 4;
```

Always Block

```
always @(sensitivity_list)
[begin]
    [procedural assignment statements]
    [if-else statements]
    [case statements]
    [while, repeat, and for loops]
    [task and function calls]
[end]
```

For Loop

```
for (initial_index; terminal_index; increment)
  begin
     statement;
  end
```

The Conditional Operator

```
conditional_expression ?  true_expression : false_expression
```

The if-else Statement

```
if (expression1)
begin
      statement;
end
else if (expression2)
begin
      statement;
end
else
begin
      statement;
end
```

The case Statement

```
case (expression)
            alternative1: begin
                              statement;
                               end
            alternative2: begin
                              statement;
                               end
            [default: begin
                              statement;
                        end]
endcase
```

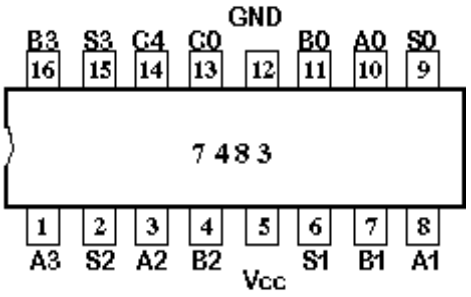
Functions and Tasks

```
function [range | integer] function_name;
      [input declarations]
      [parameter, reg, integer declarations]
      Begin
        statement;
      end
endfunction
```

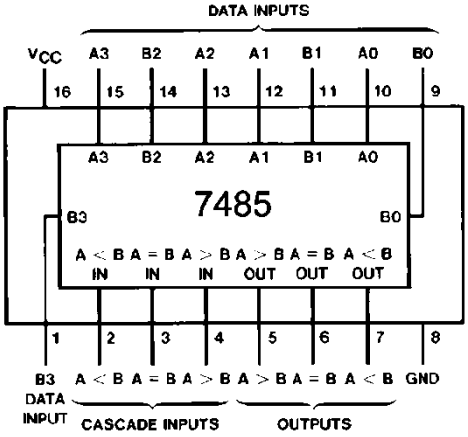
A task is declared by the keyword **task** and it comprises a block of statements that ends with the keyword **endtask**.

IC Details

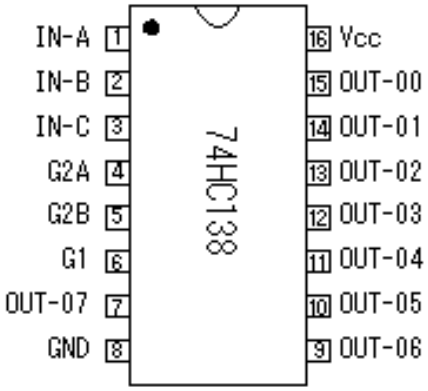
7483 : 4 bit binary full adder



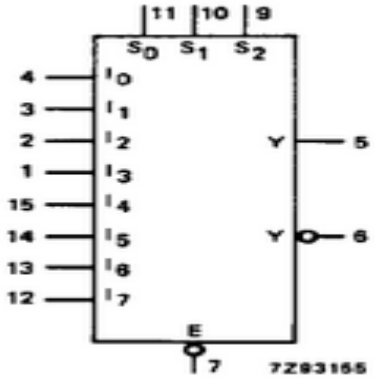
7485:4 bit Magnitude comparator



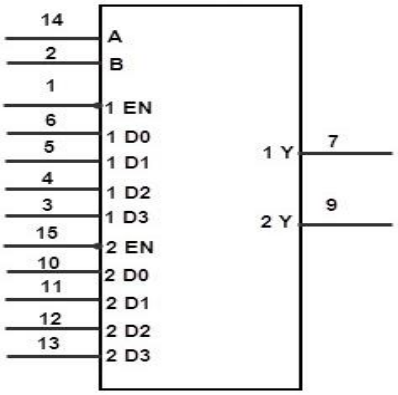
74138 : 3 to 8 active low output decoder



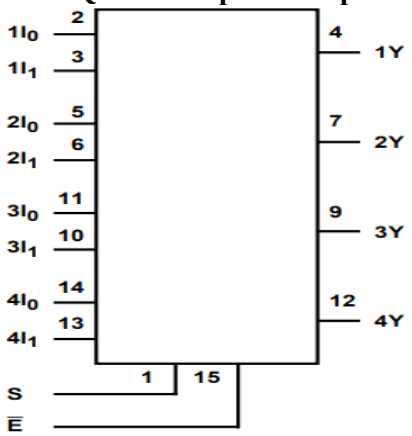
74151 : 8 input multiplexer



74153 : DUAL 4 input multiplexer

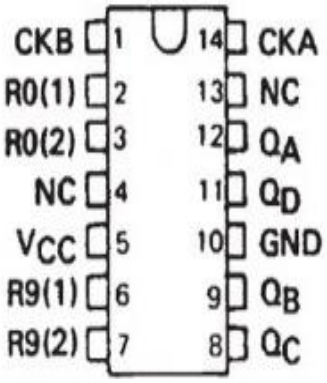


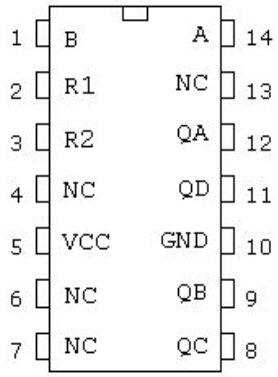
74157 : QUAD 2 input multiplexer



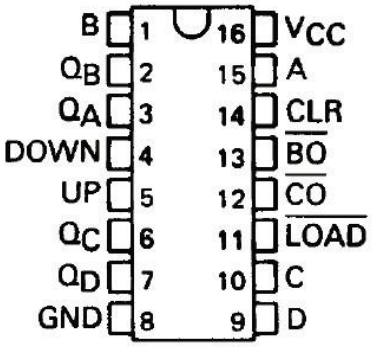
7490 : Asynchronous Decade counter

7493 : Asynchronous MOD 16 UP counter





74193 : Synchronous UP/DOWN counter



OBJECT ORIENTED PROGRAMMING

INTRODUCTION TO JAVA

Simple Program

```
/*
 This is a simple Java program.
  Call this file Example.java
. */
class Example
{    // A Java program begins with a call to main().
       public static void main(String args[])
  {
       System.out.println("Java drives the Web.");
  }
}
```

Data Types in Java:
There are two categories of data types available in Java:

Primitive Data Types:
byte (8 bits), short(16 bits), int (32 bits), long(64 bits), float(32 bits), double(64 bits), boolean, char (16 bits)
Reference/Object Data Types: Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. Examples: Employee, Puppy etc. Class objects, and various type of array variables come under reference data type. Default value of any reference variable is null. A reference variable can be used to refer to any object of the declared type or any compatible type.

Java Literals

Java Literals: A literal is a source code representation of a fixed value. They are represented directly in the code without any computation. Literals can be assigned to any primitive type variable. For example: **byte a = 68; char a = 'A';**
String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are: **"Hello World", "two\nlines", "\"This is in quotes\""**
Java language supports few special escape sequences for String and char literals as well. They are:

Notation	Character represented
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Form feed (0x0c) 62
\b	Backspace (0x08)
\s	Space (0x20)
\t	tab
\"	Double quote
\'	Single quote
\\	backslash
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

Arrays:

To declare a one-dimensional array, you can use this general form:
type array-name[ ] = new *type*[*size*];
int arr[ ]; // Declare an integer array
arr =new int[100]; // Allocate 100 elements of memory

int arr [ ] = new int [100];
//Declare and allocate an integer array in one statement.

The general form for initializing a one-dimensional array is shown here:
type array-name[ ] = { *val1, val2, val3, ... , valN* };

int arr [ ] = {1, 2, 3, 4}; // Initializing 1D array
int arr2D [] = new int[10][20]; //declare and allocate 2D array.

The general form of array initialization for a two-dimensional array is shown here:
type-specifier array_name[ ] [ ] =

```
{
        { val, val, val, ..., val },
        { val, val, val, ..., val },
        . . .
        { val, val, val, ..., val }
};
```

Java Operators:

Arithmetic Operators
+, -, *, / (addition, subtraction, multiplication, division)
%, ++, -- (modulus, increment, decrement)

Relational Operators
==, !=, >, < **(**equal, not equal, greater, lesser)
>=, <= (greater or equal, lesser or equal)

Logical Operators
&, |, !, ^, || , && (AND, OR, NOT, XOR, short -circuit OR, short -circuit AND)
Bitwise Operators
&, |, ~, ^ (AND, OR, NOT, XOR)

>>, >>>, << (shift right, shift right zero fill, shift left)

Conditional operator (?)
The **?** is called a *ternary operator* because it requires three operands. It takes the general form
Exp1 ? *Exp2* : *Exp3*;

absval = val < 0 ? -val : val; // get absolute value of val

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Branch Structures:
The syntax of **if** ,**if...else**, **if...else if...else**, Nested **if...else**, **switch- case, for, while, do-while break**, and **continue** statements are similar to that of C++.

Defining a class:
A class is created by using the keyword **class**. A simplified general form of a **class** definition is shown here:

```
class classname
{ // declare instance variables
      type var1;
      type var2;
       // ...
       type varN;
      // declare methods
      type method1(parameters) { // body of method }
      type method2(parameters) { // body of method }
      // ...
       type methodN(parameters) { // body of method }
 }
```

Java Access Modifiers:
Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:
Visible to the package: **the default**. No modifiers are needed.
Visible to the class only: **private**.
Visible to the world: **public.**
Visible to the package and all subclasses: **protected.**

INHERITANCE
Syntax:
access_modifier class *subclass-name* extends *existing-class-name*
{ . . // Changes and additions. . }
Example:
public class *Flying_Bird* extends *Bird*
{ . . // Changes and additions. . }

INTERFACE
Syntax:
access_modifier interface *interface-name*
{ . . // Abstract methods
. . // Interface Constants }

A class can both extend one other class and implement one or more interfaces.

PACKAGES
Syntax:
Following is the syntax for package creation:
package *package_name;*
Example:
package Mypack;

You can create a hierarchy of packages by separating them with a period.
package pkg1[.pkg2 [.pkg3]]
Example:
package com.course.in;

Importing packages:
Syntax:
Following is the syntax for importing packages:
import pkg1[.pkg2].(classname|*);
Example:
import java.util.Date;
import java.io.*;

ArrayList

Constructor	Description
ArrayList()	To build an empty array list.
ArrayList(Collection c)	To build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	To build an array list that has the specified initial capacity.

Method	Description
void add(int index, E element)	To insert the specified element at the specified position in a list.
boolean addAll(Collection c)	To append all of the elements in the specified collection to the end of this list.

void clear()	To remove all of the elements from this list
boolean add(Object e)	To append the specified element at the end of a list
boolean addAll(int index, Collection c)	To append all the elements in the specified collection, starting at the specified position of the list.
Object clone()	It is used to return a shallow copy of an ArrayList
int indexOf(Object o)	To return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element
int lastIndexOf(Object o)	To return the index in this list of the last occurrence of the specified element.

Strings

Method	**Description**
char charAt(int where)	Returns the character at the specified position
String replace(char original, char replacement)	Replaces the original character with the specified replacement and returns the string
boolean equals(Object str)	Compares two objects if they are same or not and returns true or false.
boolean equalsIgnoreCase(String str)	Compares two strings if they are same or not and returns true or false.
String concat(String str)	Concatenates str with the invoking String and returns it.
String trim( )	Returns the invoking String after removing leading and trailing spaces.
boolean startsWith(String str)	Checks if the invoking string starts with str and returns true if found otherwise returns false
boolean endsWith(String str):	Checks if the invoking string ends with str and returns true if found otherwise returns false
int compareTo(String *str)*	Compares two strings and based on the comparison returns zero or less than zero or greater than zero.
int indexOf(int ch)	Returns the index of the first occurrence of ch in the invoking String. Returns -1 if not found.
int lastIndexOf(int cha)	Returns the index of the last occurrence of ch in the invoking String. Returns -1 if not found.
int indexOf(String st)	Returns the index of the first occurrence of st in the invoking String. Returns -1 if not found.
int lastIndexOf(String st):	Returns the index of the last occurrence of st in the invoking String. Returns -1 if not found.
int indexOf(int ch, int fromIndex)	Here, fromIndex specifies the index at which point the search begins. For indexOf( ), the search runs from fromIndex to the end of the string. For lastIndexOf( ), the search runs from zero to tillIndex.
int lastIndexOf((int ch, int fromIndex)	
int indexOf(String st, int fromIndex)	
int lastIndexOf(String st, int tillIndex)	

String substring(int startIndex)	Here, startIndex specifies the beginning index, and endIndex specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index. If endIndex is not specified then the entire string from the startIndex is returned.
String substring(int startIndex, int endIndex)	

String Buffer

Constructor	**Description**
StringBuffer()	An object of StringBuffer created with default size of 16
StringBuffer(int size)	An object of StringBuffer created with specified size
StringBuffer(String s)	An object of StringBuffer created with specified string

Method	**Description**
int capacity()	Returns the number of characters it can hold
int length()	Returns the actual number of characters it is holding
char charAt(int pos)	Returns with the character found at the position in the given StringBuffer
void setCharAt(int i, char ch)	Sets the character in the given StringBuffer with ch at the specified position i
StringBuffer append(String s)	Concatenates string s to the invoking StringBuffer and returns the StringBuffer
StringBuffer insert(int i, String st)	Inserts string st to the invoking StringBuffer at the position i and returns the StringBuffer
String[] split(String regex)	The string is split as many times on the regex appearing in the invoked string and is returned as a string array.
String[] split(String regex, int limit)	The string is split as many times as specified by the limit on the regex appearing in the invoked string and is returned as a string array.
String regionMatches(int toffset, String other, int ooffset, int len):	Tests if the two Strings are equal. Using this method we can compare the substring of input String with the substring of specified String. Parameters: ignoreCase– if true, ignore case when comparing characters. toffset – the starting offset of the subregion in this string. other – the string argument. ooffset – the starting offset of the subregion in the string argument. len – the number of characters to compare.
String regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len):	

Exception Handling

Java Exception Keywords:

Keyword	Description
try	used to specify a block where we should place exception code.
catch	used to handle the exception.
finally	used to execute the important code of the program.
throw	used to raise an exception explicitly at runtime
throws	used to declare the exception that might raise during program execution

Multithreading

Method	Description
public void run():	Used to perform action for a thread.
public void start():	Starts the execution of the thread.JVM calls the run() method on the thread.
public void sleep(long miliseconds)	Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
public void join()	Waits for a thread to die.
public void join() throws InterruptedException	Waits for a thread to die for the specified miliseconds.
public int getPriority():	Returns the priority of the thread.
public int setPriority(int priority)	Changes the priority of the thread.
public String getName()	Returns the name of the thread.
public void setName(String name)	Changes the name of the thread.
public static Thread currentThread()	Returns the reference of currently executing thread.
public int getId():	Returns the id of the thread.
public Thread.State getState():	Returns the state of the thread.
public boolean isAlive():	Tests if the thread is alive.
public void yield():	Causes the currently executing thread object to temporarily pause and allow other threads to execute.
public void suspend():	Used to suspend the thread(depricated).
public void resume():	Used to resume the suspended thread(depricated).
public void stop():	Used to stop the thread(depricated).
public void interrupt():	Interrupts the thread.
public boolean isInterrupted():	Tests if the thread has been interrupted.
public static boolean interrupted():	Tests if the current thread has been interrupted.

public final void wait()throws InterruptedException	Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
public final void wait(long timeout)throws InterruptedException	
public final void notify()	Wakes up a single thread that is waiting on this object's monitor.
public final void notifyAll()	Wakes up all threads that are waiting on this object's monitor.

GENERICS

Generics are parameterized types enable us to create classes, interfaces, and methods in which the type of data on which they operate is specified as a parameter.
Generic methods: Generic methods are methods that introduce their own type parameters. The syntax for a generic method includes a list of type parameters, inside angle brackets, which appears before the method's return type. For static generic methods, the type parameter. For static generic methods, the type parameter section must appear before the method's return type.

Example:

```
public class GenericMethodTest {
  // generic method printArray
  public static < E > void printArray( E[] inputArray ) {
    // Display array elements
    for(E element : inputArray) {
      System.out.printf("%s ", element);
    }
    System.out.println();
  }
```

Generic class:

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.
Example:

```
public class Box<T> {
      private T t;
      public void add(T t) {
     this.t = t;
  }
  public T get() {
    return t;
  }
  public static void main(String[] args) {
    Box<Integer> integerBox = new Box<Integer>();
    Box<String> stringBox = new Box<String>();
     integerBox.add(new Integer(10));
           ….
}
```

IO Streams

Method	Description
String getName()	Returns the name of the file
String getParent()	Returns the name of the parent directory
String getPath()	Returns the relative path
String getAbsolutePath(	Returns the absolute path
boolean exists()	Returns true if the file exists, false if it does not
boolean canWrite()	Returns true if the file is writable
boolean canRead()	Returns true if the file is readable
boolean isDirectory()	Returns true if the file is a directory
boolean isFile()	Returns true if called on a file and false if called on a directory.
boolean isAbsolute()	Returns true if the file has an absolute path and false if its path is relative
long length()	Returns the size of the file
boolean renameTo(File *newName*)	Rename the file to newName
boolean mkdir( )	Create a directory for which path exists
boolean mkdirs( )	Create a directory for which no path exists. It creates a directory and all the parents of the directory
boolean delete()	Deletes the disk file represented by the path of the invoking File object.

Directories:

Method	Description
String [] list()	Returns the list of files in the directory
File[ ] listFiles( )	return the file list as an array of File objects
File listFiles(FilenameFilter *FFObj*)	Returns those files that satisfy the specified FilenameFilter

Methods defined by InputStream class:

Method	Description
int available( )	Returns the number of bytes of input currently available for reading.
void mark(int numBytes)	Places a mark at the current point in the input stream that will remain valid until numBytes bytes are read.
boolean markSupported( )	Returns true if mark( )/reset( ) are supported by the invoking stream.

int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte buffer[ ])	Attempts to read up to buffer.length bytes into buffer and returns the actual number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte buffer[ ], int offset, int numBytes)	Attempts to read up to numBytes bytes into buffer starting at buffer[offset], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long numBytes)	Ignores (that is, skips) numBytes bytes of input, returning the number of bytes actually ignored.

Methods defined by OutputStream class :

Method	**Description**
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers
void write(int b)	Writes a single byte to an output stream. Note that the parameter is an int, which allows you to call write( ) with expressions without having to cast them back to byte.
void write(byte buffer[ ])	Writes a complete array of bytes to an output stream
void write(byte buffer[ ], int offset, int numBytes	Writes a subrange of numBytes bytes from the array buffer, beginning at buffer[offset].

Methods defined by the Reader class :

Method	**Description**
void mark(int numChars)	Places a mark at the current point in the input stream that will remain valid until numChars characters are read.
boolean markSupported( )	Returns true if mark( )/reset( ) are supported on this stream
int read( )	Returns an integer representation of the next available character from the invoking input stream. –1 is returned when the end of the file is encountered.
int read(char buffer[ ])	Attempts to read up to buffer.length characters into buffer and returns the actual number of characters that were successfully read. –1 is returned when the end of the file is encountered.
abstract int read(char buffer[ ], int offset, int numChars	Attempts to read up to numChars characters into buffer starting at buffer[offset], returning the number of characters successfully read. –1 is returned when the end of the file is encountered.
boolean ready( )	Returns true if the next input request will not wait. Otherwise, it returns false.
void reset( )	Resets the input pointer to the previously set mark.

long skip(long numChars)	Skips over numChars characters of input, returning the number of characters actually skipped.

Methods defined by the Writer class :

Method	**Description**
Writer append(char ch)	Appends ch to the end of the invoking outputstream. Returns a reference to the invoking stream.
Writer append(CharSequence chars)	Appends chars to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence chars, int begin, int end)	Appends the subrange of chars specified by begin and end–1 to the end of the invoking ouput stream. Returns a reference to the invoking stream.
abstract void close( )	Closes the output stream. Further write attempts will generate an IOException.
abstract void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int ch)	Writes a single character to the invoking output stream. Note that the parameter is an int, which allows you to call write with expressions without having to cast them back to char.
void write(char buffer[ ])	Writes a complete array of characters to the invoking output stream.
abstract void write(char buffer[ ], int offset, int numChars)	Writes a subrange of numChars characters from the array buffer, beginning at buffer[offset] to the invoking output stream.
void write(String str)	Writes str to the invoking output stream.
void write(String str, int offset, int numChars)	Writes a subrange of numChars characters from the string str, beginning at the specified offset.

Commonly used methods defined by ObjectOutputStream class :

Method	**Description**
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(byte buffer[ ])	Writes an array of bytes to the invoking stream.
void write(byte buffer[ ], int offset, int numBytes)	Writes a subrange of numBytes bytes from the array buffer, beginning at buffer[offset].
void write(int b)	Writes a single byte to the invoking stream. The byte written is the low-order byte of b.
void writeBoolean(boolean b)	Writes a boolean to the invoking stream.
void writeByte(int b)	Writes a byte to the invoking stream. The byte written is the low-order byte of b.
void writeBytes(String str)	Writes the bytes representing str to the invoking stream.
void writeChar(int c)	Writes a char to the invoking stream.
void writeChars(String str)	Writes the characters in str to the invoking stream.
void writeDouble(double d)	Writes a double to the invoking stream.
void writeFloat(float f)	Writes a float to the invoking stream.

void writeInt(int i)	Writes an int to the invoking stream.
void writeLong(long l)	Writes a long to the invoking stream.
final void writeObject(Object obj)	Writes obj to the invoking stream.
void writeShort(int i)	Writes a short to the invoking stream.

Commonly used methods defined by ObjectInputStream class :

Method	**Description**
int available( )	Returns the number of bytes that are now available in the input buffer.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte buffer[ ], int offset, int numBytes)	Attempts to read up to numBytes bytes into buffer starting at buffer[offset], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
boolean readBoolean( )	Reads and returns a boolean from the invoking stream.
byte readByte( )	Reads and returns a byte from the invoking stream.
char readChar( )	Reads and returns a char from the invoking stream.
double readDouble( )	Reads and returns a double from the invoking stream.
float readFloat( )	Reads and returns a float from the invoking stream.
int readInt( )	Reads and returns an int from the invoking stream.
long readLong( )	Reads and returns a long from the invoking stream.
final Object readObject( )	Reads and returns an object from the invoking stream.
short readShort( )	Reads and returns a short from the invoking stream.

Some of the other methods of RandomAccessFile :

Method	**Description**
long length()	Returns the length of the file in bytes.
void setLength(long *len*) throws IOException	Sets the length of the invoking file to that specified by *len.*
int skipBytes(int n)	Add n to the file pointer. Returns actual number of bytes skipped. If n s negative, no bytes are skipped.
long getFilePointer() throws IOException	Returns the current offset, in bytes, from the beginning of the file to where the next read or write occurs.
void seek(long *newPos*) throws IOException	Sets the current position of the file pointer within the file. *newPos* specifies the new position (in bytes), of the file pointer from the beginning of the file

JFrame Methods:

Method	Description
void setSize(int *width*, int *height*)	Sets the dimensions of the JFrame window.
setVisible(boolean b)	Container is made visible or not visible by setting value b
setLayout(LayoutManager lm)	Sets the layout manager to the container.
add(component c)	Add the component c to the container.
setDefaultCloseOperation(int op);	Sets the operation the user clicks the close Box of a JFrame window. Operation op takes the following constants. JFrame.EXIT_ON_CLOSE JFrame.DO_NOTHING_ON_CLOSE

The methods of ActionEvent object.

Method	Description
String getActionCommand( )	Returns command name for the invoking ActionEvent object.
int getModifiers( )	Returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated.
long getWhen( )	Returns the time at which the event took place.

JavaFX

The JavaFX Packages

The JavaFX elements are contained in packages that begin with the javafx prefix. Most frequently use are: javafx.application, javafx.stage, javafx.scene, and javafx.scene.layout.

The Application class of the package javafx.application is the entry point of the application in JavaFX. To create a JavaFX application, you need to inherit this class and implement its abstract method start(). In this method, you need to write the entire code for the JavaFX graphics

In the main method, you have to launch the application using the launch() method. This method internally calls the start() method of the Application class as shown in the following program.

```
public class JavafxSample extends Application {
  @Override
  public void start(Stage primaryStage) throws Exception {
    /*
    Code for JavaFX application.
    (Stage, scene, scene graph)
    */
  }
  public static void main(String args[]){
    launch(args);
  }
}
```

JavaFX contains rich set of controls which are listed below:
JavaFX Button Control, *JavaFX CheckBox Control*, *JavaFX ChoiceBox Control*, *JavaFX ComboBox Control*, *JavaFX Hyperlink Control*, *JavaFX Label Control*,*JavaFX ListView Control*, *JavaFX MenuBar Control*, *JavaFX MenuButton Control*, *JavaFX PasswordField Control*, *JavaFX RadioButton Control*, *JavaFX ScrollBar Control*, *JavaFX ScrollPane Control*, *JavaFX Slider Control*, *JavaFX TableView Control*, *JavaFX TabPane Control*, *JavaFX TextArea Control*, *JavaFX TextField Control*, *JavaFX ToggleButton Control*, *JavaFX TreeView Control* *etc.*

CERT JAVA CODING STANDARD

The CERT Oracle Secure Coding Standard for Java provides rules for secure coding in the Java programming language. The goal of these rules is to eliminate insecure coding practices that can lead to exploitable vulnerabilities.

Rules and Recommendations:

Rules are the requirements that we should or must follow otherwise it leads to some vulnerabilities. Rules are the requirements for conformance with the standard.

Example for the Rule:

1.Do not ignore values returned by methods

```
public class Replace
 {  public static void main(String[] args) {
   String original = "insecure";    original.replace('i', '9');    System.out.println(original);  } }
```

This noncompliant code example ignores the return value of the String.replace() method, failing to update the original string. The following compliant solution correctly updates the String reference original with the return value from the String.replace() method.

```
public class Replace {  public static void main(String[] args) {
String original = "insecure";    original = original.replace('i', '9');    System.out.println(original);  } }
```

Recommendation describe good practices or useful advice. And do not establish conformance requirements.

Example for the Recommendations:

2. Do not declare more than one variable per declaration

```
int i, j = 1;
```

This noncompliant code might lead a programmer or reviewer to mistakenly believe that both i and j are initialized to 1. In the following compliant solution, it is readily apparent that both i and j are initialized to 1:

```
int i = 1;  // Purpose of i...
int j = 1;  // Purpose of j...
```

PRINCIPLES OF DATA COMMUNICATION

Thermal Noise : N = kTB
Where k = boltzmanns constant = 1.38 x10^{-23}
T = temperature in kelvin
N = Noise in watt

Channel Capacity

Nyquist Bandwidth: $C = 2B\log_2 M$

Here B is the bandwidth of the channel and M is the number of signal levels used to represent the data.

Shannon Capacity: $C = B \log_2 (1+SNR)$

Here B is the bandwidth of the channel in Hertz, SNR is the signal to noise ratio and C is the channel capacity in bits per second.
Signal to noise ratio is represented in decibels

$$SNR_{dB} = 10 \log_{10} \frac{\text{signal power}}{\text{noise power}}$$

Line Coding

B8ZS:
In **t**his line coding technique, Sequence of eight 0's replaced by **000 + - 0 - +,** if the previous pulse was positive. Sequence of eight 0's replaced by **000 - + 0 + -,** if the previous pulse was negative.

HDB3:
This line coding technique replaces a sequence of 4 zeros by a code as per the rule given in below table.

	Number of Bipolar Pulses (ones) since Last Substitution	
Polarity of Preceding Pulse	Odd	Even
−	000 −	+ 00 +
+	000 +	− 00 −

Modulation techniques

MFSK:
Transmitted signal is given by
$s_i(t) = A\cos(2\pi f_i t), \quad 1 \le i \le M$
where

$f_i = f_c + (2i - 1 - M) f_d$
f_c = *the carrier frequency*
f_d = *the difference frequency*
M = *number of different signal elements* $= 2^L$
L = *number of bits per signal element*

Period of signal element

$T_s = LT_b, \quad T_s : signal\ element\ period \quad T_b : bit\ period$

Minimum frequency separation

$1/T_s = 2f_d \quad \Rightarrow \quad 1/(LT_b) = 2f_d \quad \Rightarrow \quad 1/T_b = 2Lf_d \ (bit\ rate)$

MFSK signal bandwidth

$W_d = M(2f_d) = 2Mf_d$

Performance

The transmission bandwidth

ASK $B_T = (1 + r)R$

MFSK $B_T = \left(\frac{(1 + r)M}{\log_2 M}\right)R$

MPSK $B_T = \left(\frac{1 + r}{L}\right)R = \left(\frac{1 + r}{\log_2 M}\right)R$

Here r depends on modulation and filtering process where 0<= r <=1, R is bit rate, M is number of different signal element and L is number of bits.

Line of Sight transmission

Free space loss is

$P_t/P_r = (4\pi d)^2/\lambda = (4\pi fd)^2/c^2$

Where

P_t = signal power at the transmitting antenna

Pr= signal power at the receiving antenna

λ = carrier wavelength(in meters)

d=propagation distance between antennas (in meters)

c=speed of light (3×10^8 m/s)

This can be recast as

$L_{dB} = 10 \log (P_t/P_r) = 20 \log(4\pi d/\lambda) = -20 \log(\lambda) + 20 \log(d) + 21.98$ Db

$= 20 \log(4\pi fd/c) = 20 \log(f) + 20 \log(d) - 147.56$ dB

For other antennas, we must take into account the gain of the antenna, which yields the following free space loss equation:

$P_t/P_r = (4\pi)^2(d)^2/G_tG_r \lambda^2 = (\lambda d)^2/A_rA_t = (cd)^2/f^2A_tA_r$

Where

G_t = gain of the transmitting antenna

G_r = gain of the receiving antenna

A_t= effective area of transmitting antenna

A_r= effective area of receiving antenna

We can recast loss equation as

$L_{dB} = 20 \log(\lambda) + 20 \log(d) - 10 \log(A_tA_r)$

$= -20 \log(f) + 20 \log(d) - 10 \log(A_tA_r) + 169.54$ dB

The effective area of an ideal isotropic antenna is $\lambda^2/4\pi$, with a power gain of 1, the effective area of a parabolic antenna with a face area of A is 0.56A, with a power gain of $7A/\lambda^2$

$d = 3.57 \sqrt{h}$

where d is the distance between an antenna and the horizon in kilometres and h is the antenna height in meters. The effective, or radio, line of sight to the horizon is

$d=3.57\sqrt{Kh}$

Where K is an adjustment factor to account for the refraction. A good rule of thumb id K=4/3. Thus, the maximum distance between two antennas for LOS propagation is $3.57(\sqrt{Kh_1}+\sqrt{Kh_2})$, where h_1 and h_2 are the heights of the two antennas.

Error Detection and Correction

Single-bit error correction

To correct an error, the receiver reverses the value of the altered bit. To do so, it must know which bit is in error.

Number of redundancy bits needed

Let data bits = m

Redundancy bits = r

$\therefore$ Total message sent = $m+r$

The value of r must satisfy the following relation:

$$\mathbf{2^r \geq m+r+1}$$

Stop-and-Wait Flow Control

Maximum possible utilization of the link, U = 1/(1+2a), where a = $\frac{Propagation\ Time}{Transmission\ Time}$

Sliding Window Flow Control

Bit Length of a link,

$B = R \times (d/V)$,

where

B = length of the link in bits; this is the number of bits present on the link at an instance in time when a stream of bits fully occupies the link

R = data rate of the link, in bps

d = length, or distance, of the link in meters

V = velocity of propagation, in m/s

For a k-bit field the range of sequence numbers is 0 through 2^k-1 and frames are numbered modulo 2^k. Maximum window size = 2^k-1.

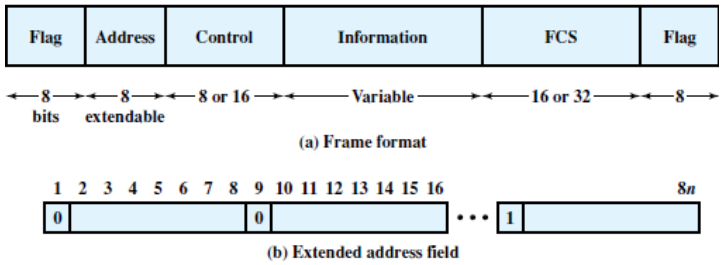
If the transmission time is normalized to one, the propagation delay can be expressed as **a= B/L**, where L is the number of bits in the frame (length of the frame in bits).

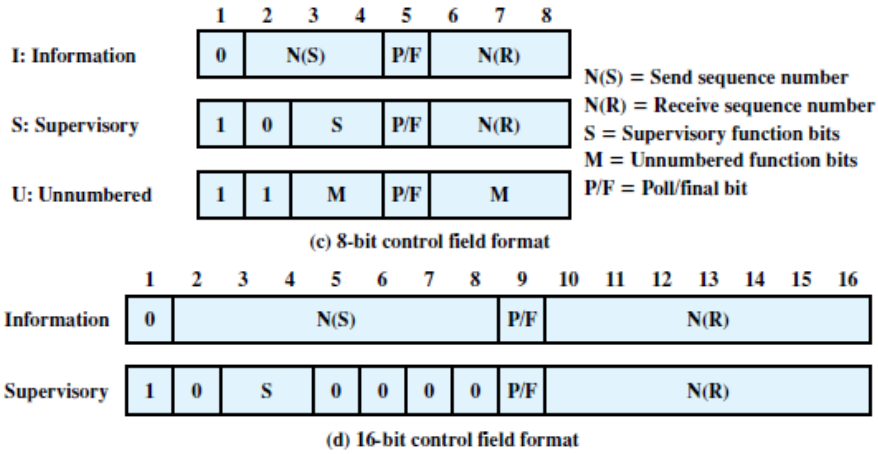
Throughput (in bps) depends on 'W' and 'a', where W is the window size.

Utilization, $U = \begin{cases} 1, & W \geq 2a+1 \\ \frac{W}{2a+1}, & W < 2a+1 \end{cases}$

HDLC

Frame format





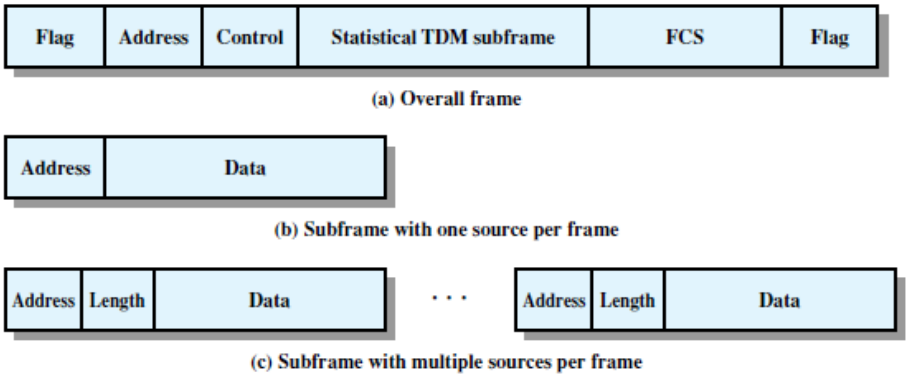
Commands and Responses

Name	Command/ Response	Description
Information (I)	C/R	Exchange user data
Supervisory (S)		
Receive ready (RR)	C/R	Positive acknowledgment; ready to receive I-frame
Receive not ready (RNR)	C/R	Positive acknowledgment; not ready to receive
Reject (REJ)	C/R	Negative acknowledgment; go back N
Selective reject (SREJ)	C/R	Negative acknowledgment; selective reject
Unnumbered (U)		
Set normal response/extended mode (SNRM/SNRME)	C	Set mode; extended = 7-bit sequence numbers
Set asynchronous response/extended mode (SARM/SARME)	C	Set mode; extended = 7-bit sequence numbers
Set asynchronous balanced/extended mode (SABM, SABME)	C	Set mode; extended = 7-bit sequence numbers
Set initialization mode (SIM)	C	Initialize link control functions in addressed station
Disconnect (DISC)	C	Terminate logical link connection
Unnumbered Acknowledgment (UA)	R	Acknowledge acceptance of one of the set-mode commands
Disconnected mode (DM)	R	Responder is in disconnected mode
Request disconnect (RD)	R	Request for DISC command
Request initialization mode (RIM)	R	Initialization needed; request for SIM command
Unnumbered information (UI)	C/R	Used to exchange control information
Unnumbered poll (UP)	C	Used to solicit control information
Reset (RSET)	C	Used for recovery; resets N(R), N(S)
Exchange identification (XID)	C/R	Used to request/report status
Test (TEST)	C/R	Exchange identical information fields for testing
Frame reject (FRMR)	R	Report receipt of unacceptable frame

S – Frame Codes	
00	RR
01	REJ
10	RNR
11	SREJ

Multiplexing

Statistical TDM Frame Formats



Random Access Protocols

Back-off time, T_B = (R x Frame Transmission Time) or (R x Propagation Time), where R is a random number between 0 and 2^K -1, and K is the number of retransmission attempts.
Maximum number of retransmission attempts, K_{max} = 15, usually.

Pure ALOHA:

Vulnerable Time = 2 x Frame Transmission Time
Throughput, $S = G \times e^{-2G}$, where G = the average number of frames generated by the system during one frame transmission time.
Maximum Throughput, S_{max} = 0.184, when G = 0.5

Slotted ALOHA:

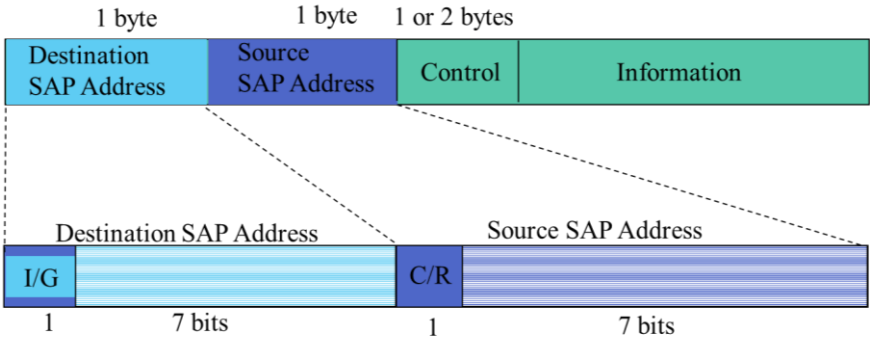
Vulnerable Time = Frame Transmission Time
Throughput, $S = G \times e^{-G}$
Maximum Throughput, S_{max} = 0.368, when G = 1

CSMA

Vulnerable Time = Propagation Time

IEEE LAN Standards

LLC PDU Structure

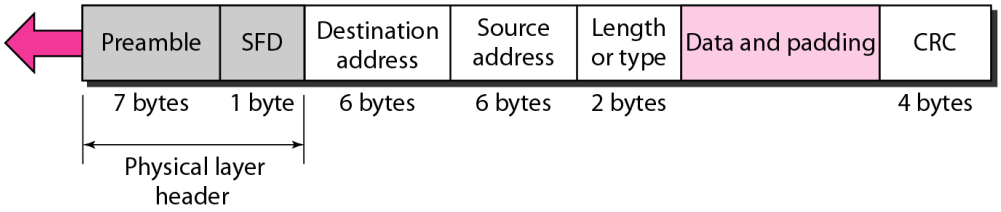


I/G = Individual or group address
C/R = Command or response frame

Standard Ethernet: 802.3 MAC Frame

Preamble: 56 bits of alternating 1s and 0s.

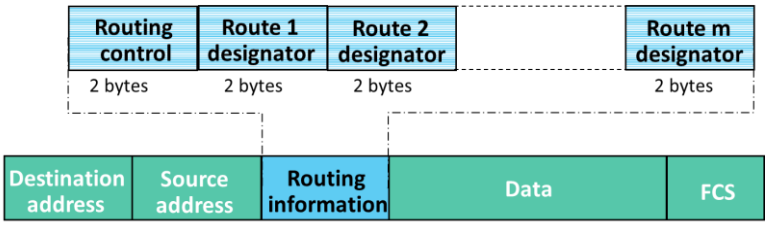
SFD: Start frame delimiter, flag (10101011)



Slot time = round-trip time + time required to send the jam sequence
Relationship between max length of network and slot time
Max Length = Propagation speed * slot time / 2 [Theoretical]

Bridges

Source Routing Bridges Frame Format



FDDI

Frame Format

Preamble	Starting delimeter	Frame control	Destination address	Source address	Info	FCS	Ending delimiter	Frame status
Bytes: 8	1	1	6	6	0-4478	4	1	1

COMPUTER NETWORK PROTOCOL

Switching

Design of Cross bar Switch

To connects n inputs to m outputs in a gridusing crossbar switch requires n × m crosspoints

Design of Multistage Swtich

For a three stage switch with N input and N output, total number of crosspoints is

$2kN+k(N/n)^2$

To design a three-stage switch, these steps should be followed:

1. Divide the N input lines into groups, each of n lines. For each group, use one crossbar of size n × k, where k is the number of crossbars in the middle stage. In other words, the first stage has N/n crossbars of n × k crosspoints.
2. Use k crossbars, each of size (N/n) × (N/n) in the middle stage.
3. Use N/n crossbars, each of size k × n at the third stage.

Design of Banyan Switch

For n inputs and n outputs, it has $\log_2 n$ stages with n/2 microswitches at each stage.

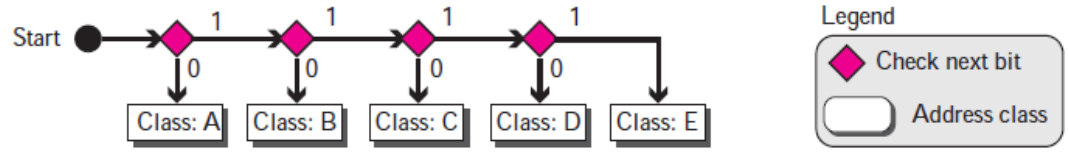
Clos Criteria

$n = (N)^{1/2}$ and $k \geq 2n-1$

Total number of crosspoints $\geq 4N [(2N)^{1/2} -1]$

IPv4 Addressing

Finding the address class using continuous checking



subnet mask : $n_{sub} = n + \log_2 s$ supernet mask : $n_{super} = n - \log_2 c$

Extracting Block Information in classless addressing

1. The number of addresses in the block, $N = 2^{32-n}$
2. First address = (any address) AND (network mask)
3. Last address = (any address) OR [NOT (network mask)]
4. Subnet mask, $n_{sub} = n + \log_2 (N/N_{sub})$

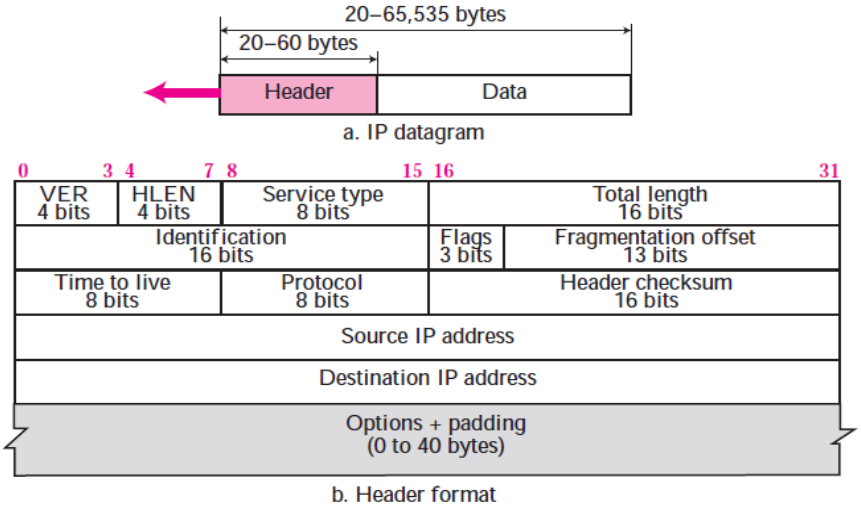
IPv6 Addressing

Prefixes for IPv6 Addresses

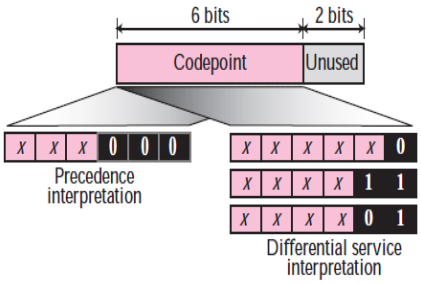
	Block Prefix	*CIDR*	*Block Assignment*	*Fraction*
1	0000 0000	0000::/8	Reserved (IPv4 compatible)	1/256
	0000 0001	0100::/8	Reserved	1/256
	0000 001	0200::/7	Reserved	1/128
	0000 01	0400::/6	Reserved	1/64
	0000 1	0800::/5	Reserved	1/32
	0001	1000::/4	Reserved	1/16
2	**001**	**2000::/3**	**Global unicast**	**1/8**
3	010	4000::/3	Reserved	1/8
4	011	6000::/3	Reserved	1/8
5	100	8000::/3	Reserved	1/8
6	101	A000::/3	Reserved	1/8
7	110	C000::/3	Reserved	1/8
8	1110	E000::/4	Reserved	1/16
	1111 0	F000::/5	Reserved	1/32
	1111 10	F800::/6	Reserved	1/64
	1111 110	FC00::/7	Unique local unicast	1/128
	1111 1110 0	FE00::/9	Reserved	1/512
	1111 1110 10	FE80::/10	Link local addresses	1/1024
	1111 1110 11	FEC0::/10	Reserved	1/1024
	1111 1111	FF00::/8	Multicast addresses	1/256

IPv4 Datagram

IP Datagram



Service Type



Category	*Codepoint*	*Assigning Authority*
1	XXXXX0	Internet
2	XXXX11	Local
3	XXXX01	Temporary or experimental

Protocols field value

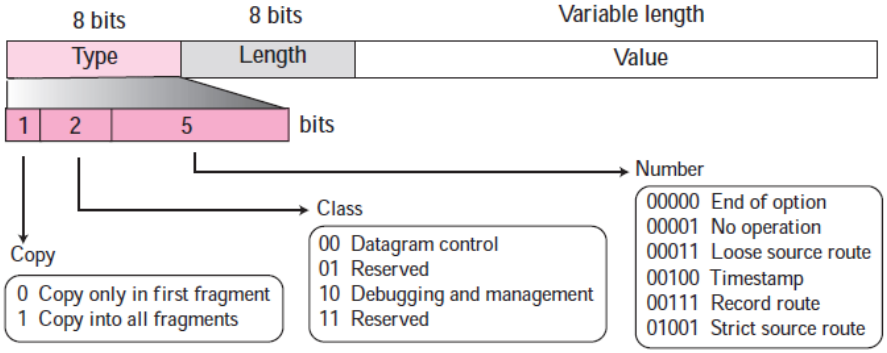
Value	*Protocol*	*Value*	*Protocol*
1	ICMP	17	UDP
2	IGMP	89	OSPF
6	TCP		

Flags field

D : Do not fragment if value is 1.

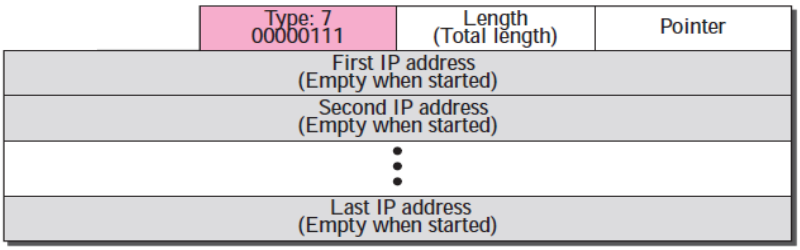
M : If value is 1 then the fragment is not the last fragment.

Options

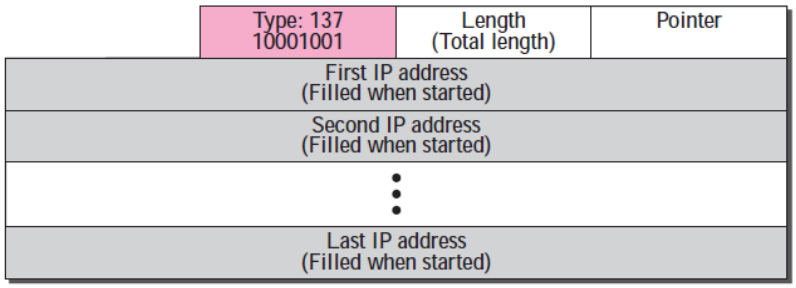


No operation option and End-of-option option

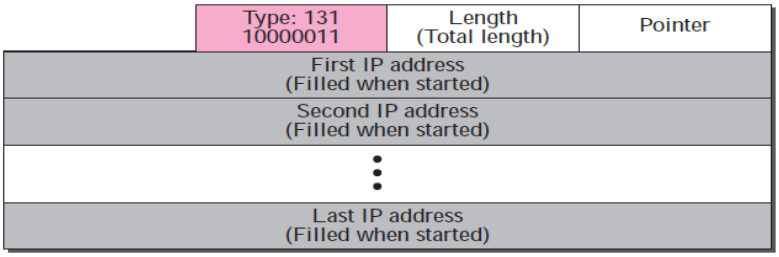
Record-route option



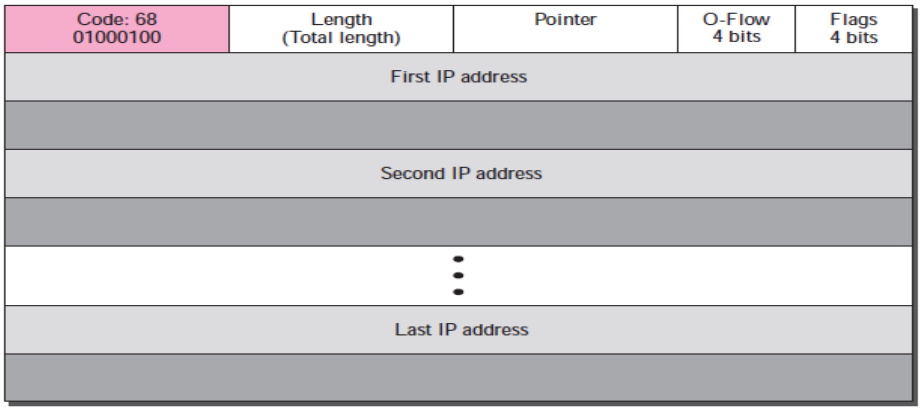
Strict-source-route option



Loose-source-route option



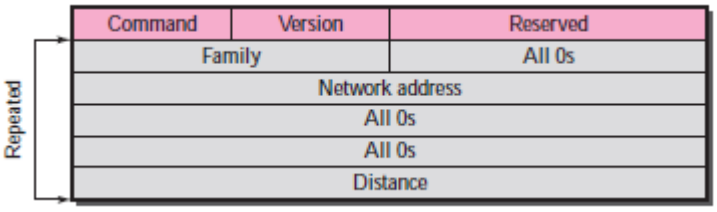
Timestamp option



Flag value is 0, each router adds only the timestamp in the provided field.
Flag value is 1, each router must add its outgoing IP address and the timestamp.
Flag value is 3, IP addresses given, enter timestamps

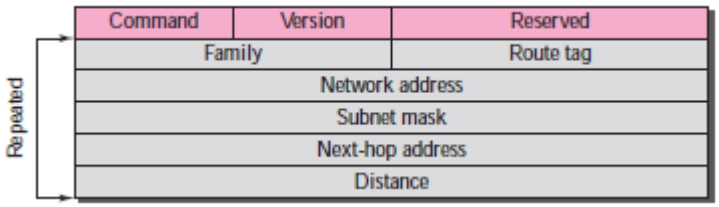
Unicast Routing Protocols

Routing Information Protocol (RIP) v1 Message Format:

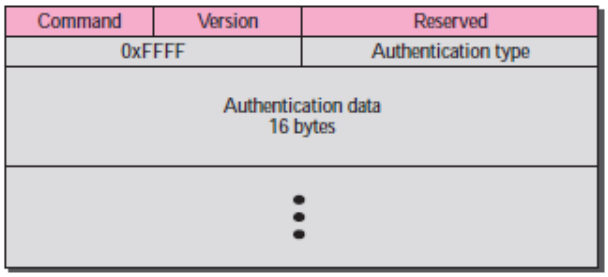


Command	Request: 1, Response: 2
Family	TCP/IP: 2

Routing Information Protocol (RIP) v2 Message Format:

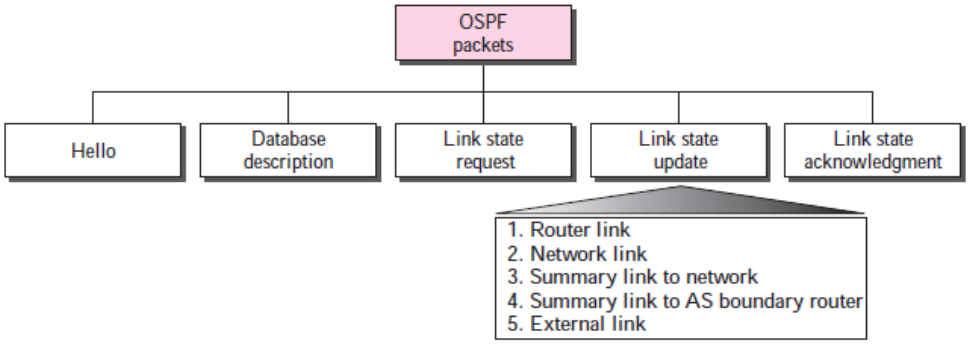


RIPv2 Authentication:

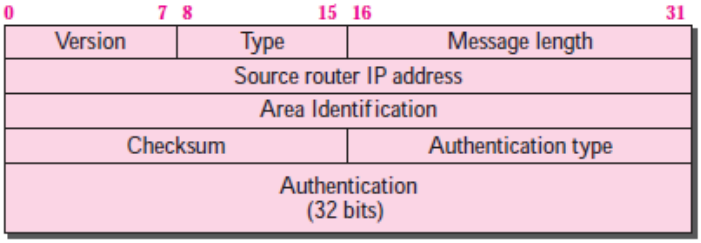


OSPF

Types of OSPF Packets

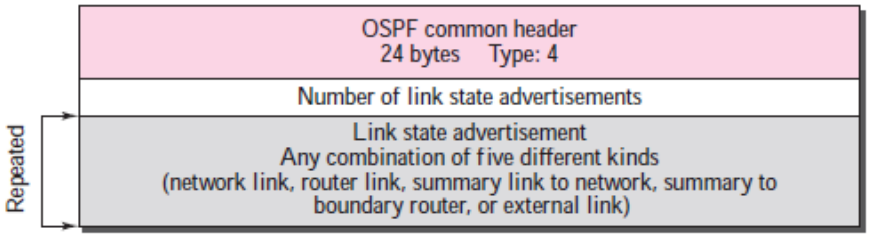


OSPF Common Header (Version 2)

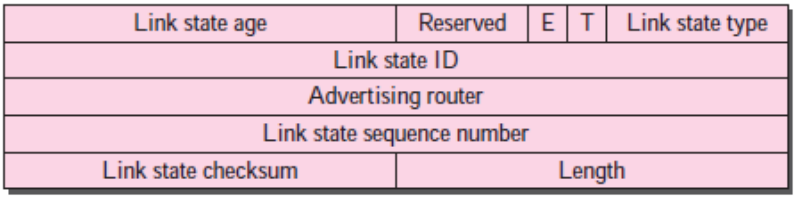


Authentication Type: 0 for None and 1 for Password.

Link State Update Packet

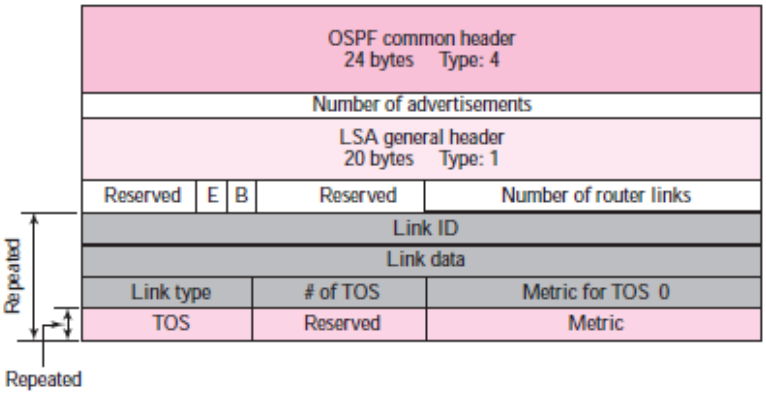


LSA General Header



E Flag (1 bit): 1 means Stub Area
T Flag (1 bit): 1 means router can handle multiple TOS
Link State Type: Router link (1), Network link (2), Summary link to network (3), Summary link to AS boundary router (4), and External link (5).

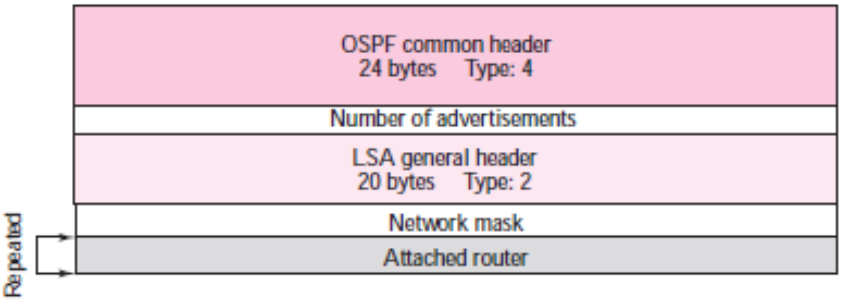
Router Link LSA



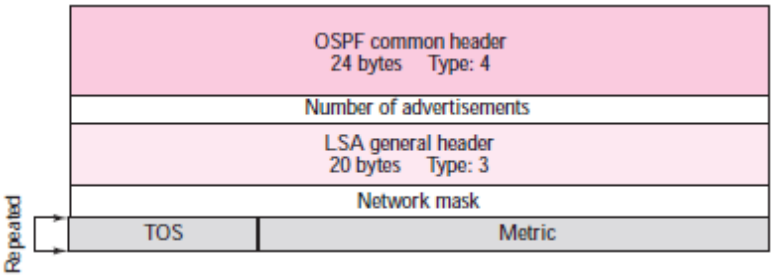
Link ID, Link Type and Link Data:

Link Type	*Link Identification*	*Link Data*
Type 1: Point-to-point	Address of neighbor router	Interface number
Type 2: Transient	Address of designated router	Router address
Type 3: Stub	Network address	Network mask
Type 4: Virtual	Address of neighbor router	Router address

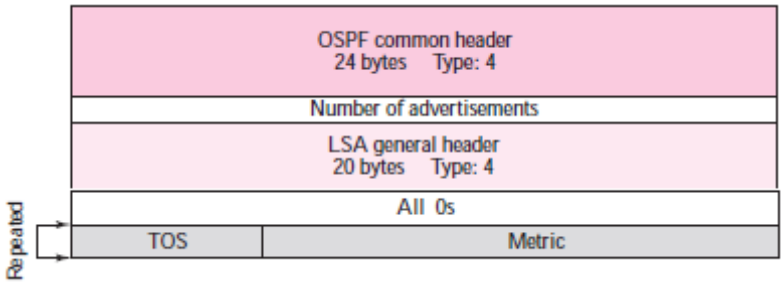
Network Link Advertisement Format



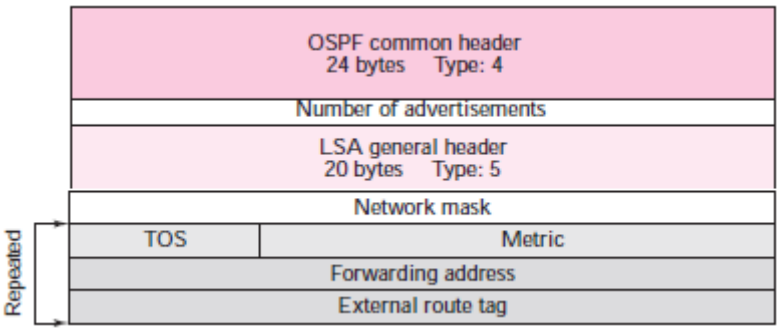
Summary Link to Network LSA



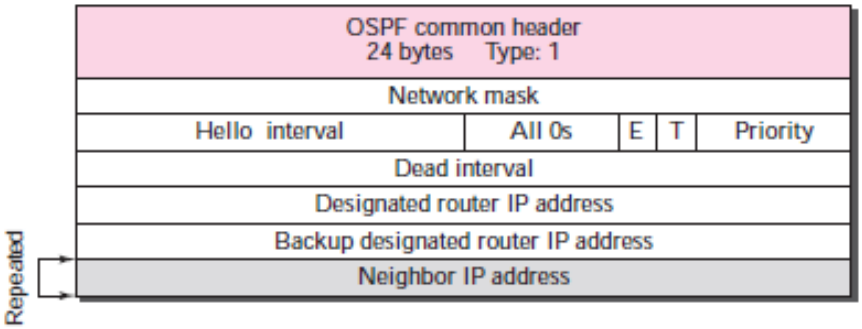
Summary Link to AS Boundary Router LSA



External Link LSA

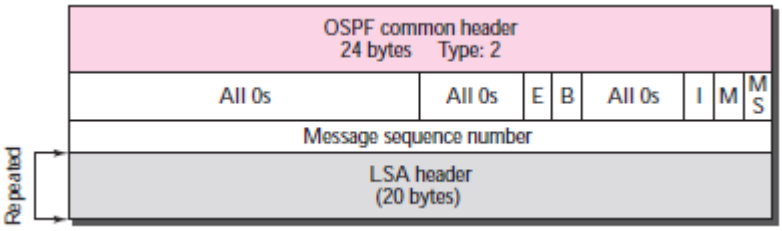


Hello Packet



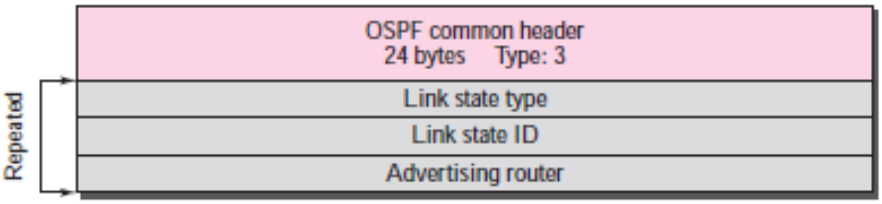
E Flag (1 bit): 1 means Stub Area
T Flag (1 bit): 1 means router can handle multiple TOS

Database Description Packet

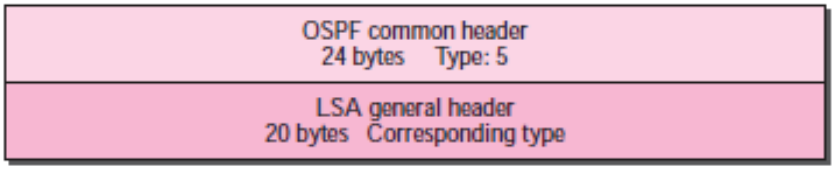


E Flag: 1 if the advertising router is an autonomous boundary router.
B Flag: 1 if the advertising router is an area border router.
I Flag: 1 if the message is the first message.
M Flag: 1 if this is not the last message.
M/S Flag: Master =1, Slave = 0

Link State Request Packet

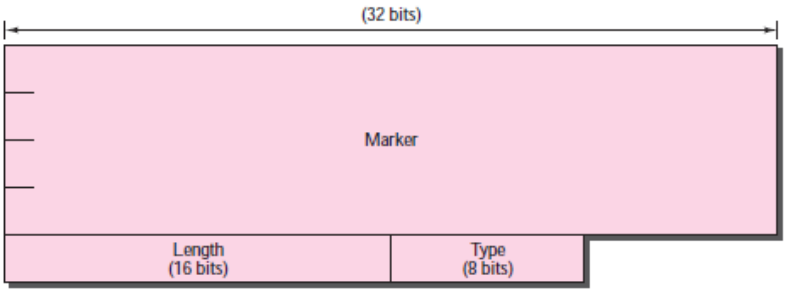


Link State Acknowledgement Packet

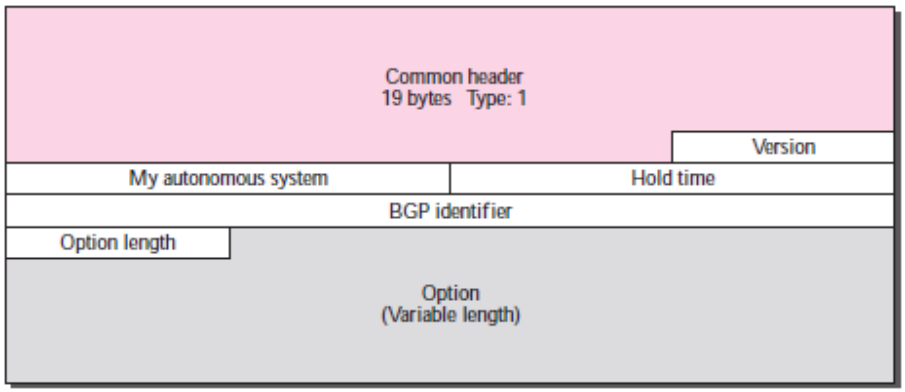


BGP

BGP Packet Header

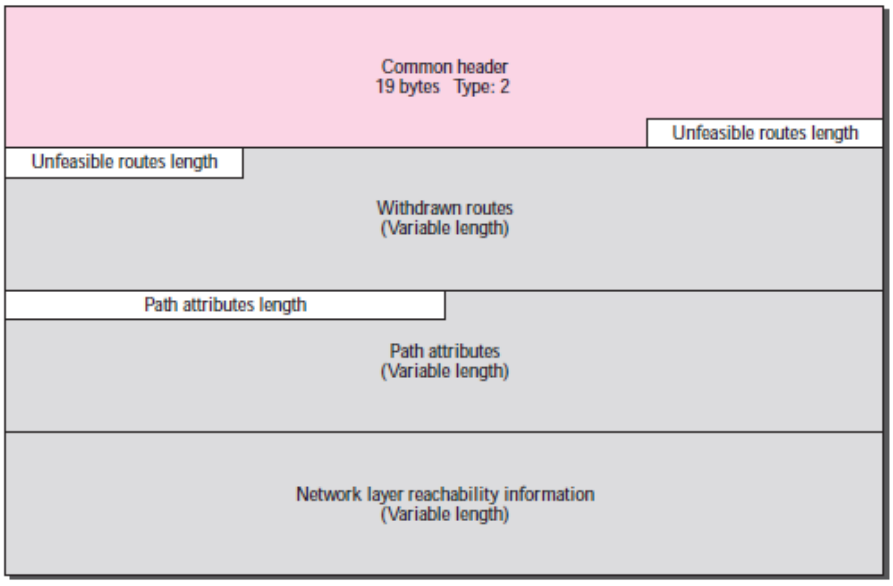


Open Message

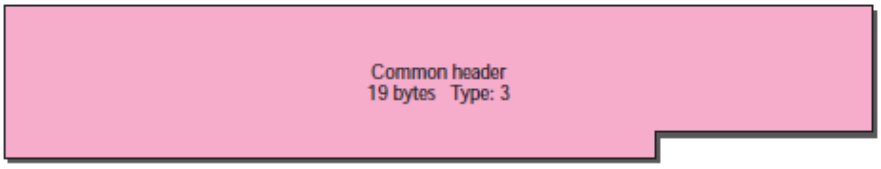


Version = 4

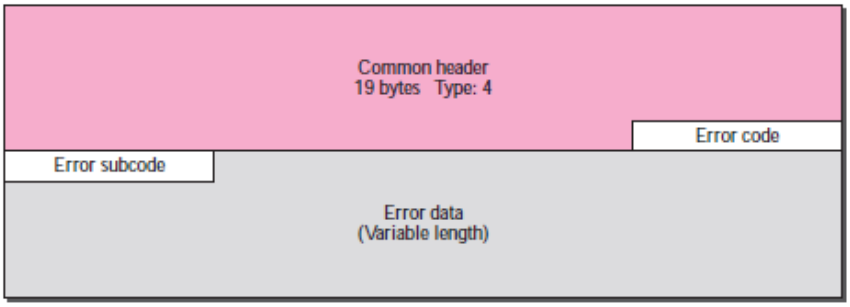
Update Message



Keepalive Message



Notification Message

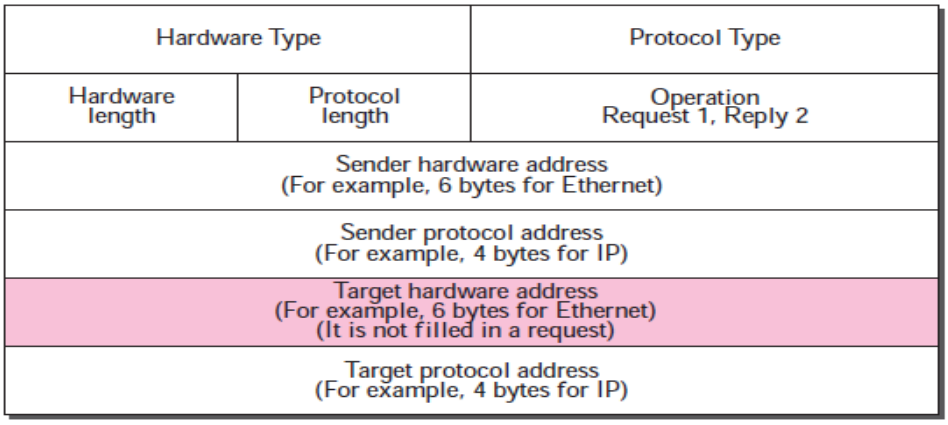


Error Codes

Error Code	*Error Code Description*	*Error Subcode Description*
1	Message header error	Three different subcodes are defined for this type of error: synchronization problem (1), bad message length (2), and bad message type (3).
2	Open message error	Six different subcodes are defined for this type of error: unsupported version number (1), bad peer AS (2), bad BGP identifier (3), unsupported optional parameter (4), authentication failure (5), and unacceptable hold time (6).
3	Update message error	Eleven different subcodes are defined for this type of error: malformed attribute list (1), unrecognized well-known attribute (2), missing well-known attribute (3), attribute flag error (4), attribute length error (5), invalid origin attribute (6), AS routing loop (7), invalid next hop attribute (8), optional attribute error (9), invalid network field (10), malformed AS_PATH (11).
4	Hold timer expired	No subcode defined.
5	Finite state machine error	This defines the procedural error. No subcode defined.
6	Cease	No subcode defined.

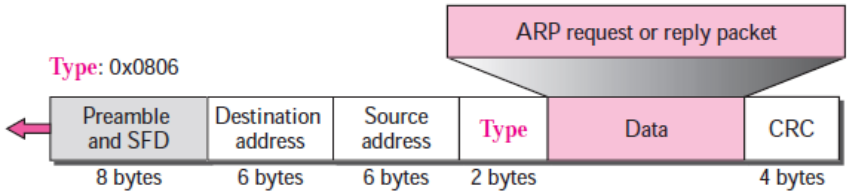
ARP

ARP packet



Hardware Type :16 bits ; Protocol Type :16 bits ; Hardware length :8 bits ; Protocol length : 8 bits; Operation : 16 bits

Encapsulation of ARP packet

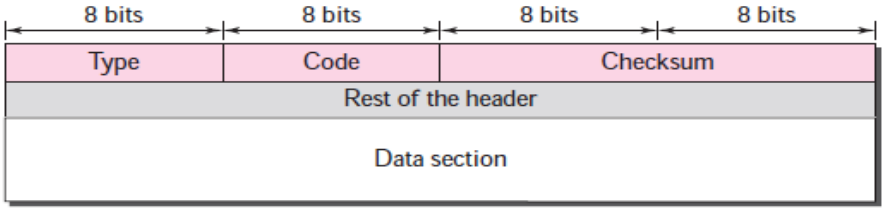


ICMP

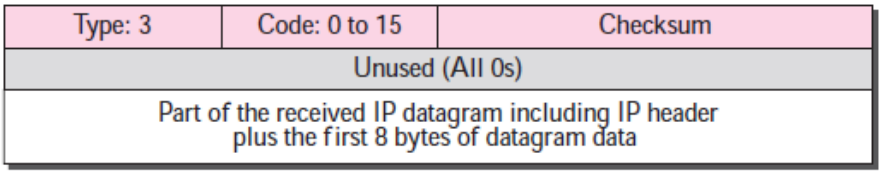
ICMP Messages

Category	*Type*	*Message*
Error-reporting messages	3	Destination unreachable
	4	Source quench
	11	Time exceeded
	12	Parameter problem
	5	Redirection
Query messages	8 or 0	Echo request or reply
	13 or 14	Timestamp request or reply

General Message format of ICMP message

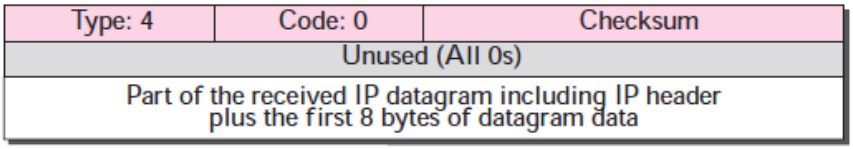


Destination Unreachable message format

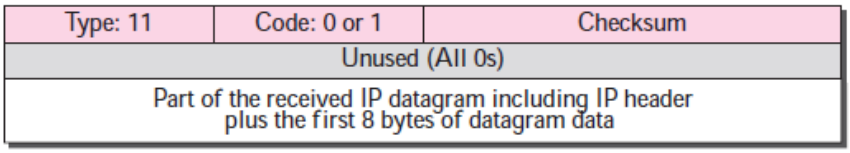


Code 0. The network is unreachable, possibly due to hardware failure.
Code 1. The host is unreachable. This can also be due to hardware failure.
Code 2. The protocol is unreachable.
Code 3. The port is unreachable.
Code 4. Fragmentation is required, but the DF (do not fragment) field of the datagram has been set.
Code 5. Source routing cannot be accomplished.
Code 6. The destination network is unknown.
Code 7. The destination host is unknown.
Code 8. The source host is isolated.
Code 9. Communication with the destination network is administratively prohibited.
Code 10. Communication with the destination host is administratively prohibited.
Code 11. The network is unreachable for the specified type of service.
Code 12. The host is unreachable for the specified type of service.
Code 13. The host is unreachable because the administrator has put a filter on it.
Code 14. The host is unreachable because the host precedence is violated.
Code 15. The host is unreachable because its precedence was cut off.

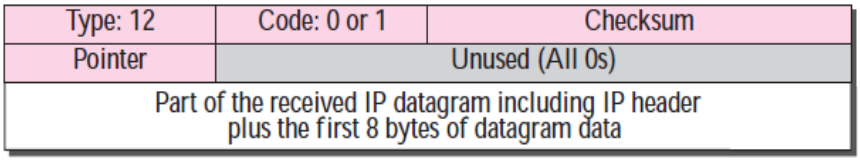
Source Quench format



Time Exceeded message format

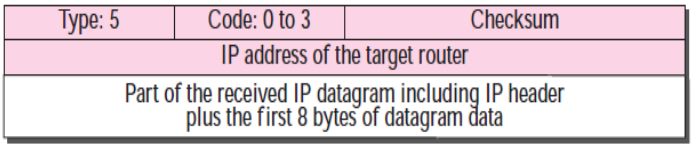


Parameter-problem message format



Code 0. There is an error or ambiguity in one of the header fields
Code 1. The required part of an option is missing.

Redirection message format

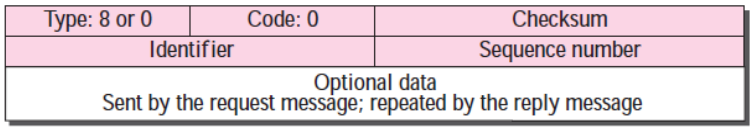


Code 0. Redirection for a network-specific route.
Code 1. Redirection for a host-specific route.
Code 2. Redirection for a network-specific route based on a specified type of service.
Code 3. Redirection for a host-specific route based on a specified type of service.

Query Messages

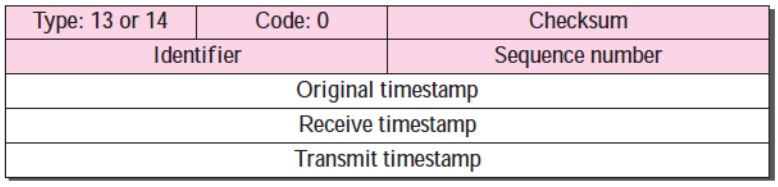
Echo request – reply messages

Type 8: Echo request
Type 0: Echo reply



Timestamp request reply messages

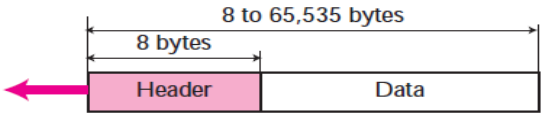
Type 13: request
Type 14: reply



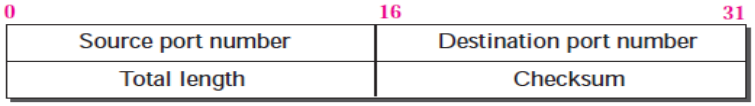
Calculations :
Sending time = receive timestamp – original timestamp
Receiving time = returned time – transmit timestamp
Round-trip time = sending time + receiving time
Time difference = receive timestamp – (original timestamp field + one way time duration)

UDP

User Datagram format

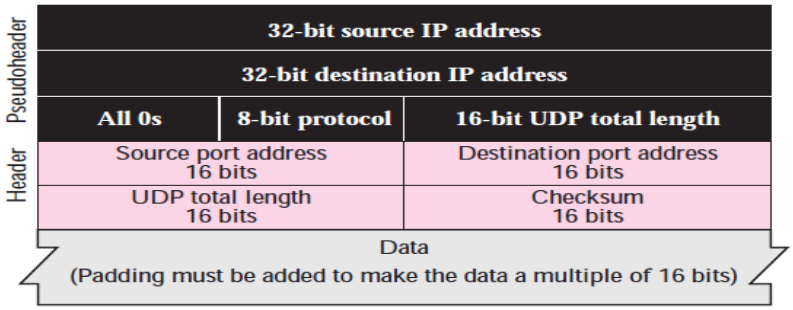


a. UDP user datagram



b. Header format

Pseudoheader for checksum calculation

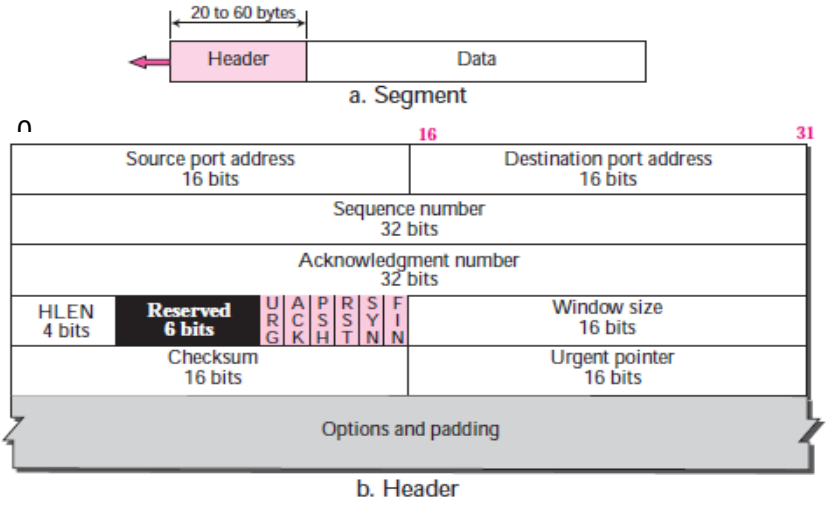


TCP

Well Known Ports

Port	*Protocol*	*Description*
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
20 and 21	FTP	File Transfer Protocol (Data and Control)
23	TELNET	Terminal Network
25	SMTP	Simple Mail Transfer Protocol
53	DNS	Domain Name Server
67	BOOTP	Bootstrap Protocol
79	Finger	Finger
80	HTTP	Hypertext Transfer Protocol

Segment Structure



RTT

Smoothed RTT, RTT_S:
After first measurement → $RTT_S = RTT_M$ where RTT_M means Measured RTT
After each measurement → $RTT_S = (1-\alpha)\ RTT_S + \alpha \times RTT_M$
Generally, $\alpha = 1/8$

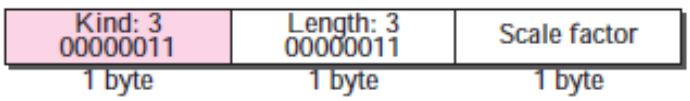
RTT Deviation, RTT_D:
After first measurement → $RTT_D = RTT_M/2$
After each measurement → $RTT_S = (1-\beta)\ RTT_D + \beta \times |RTT_S - RTT_M|$
Generally, $\beta = 1/4$

Retransmission Time-out (RTO):
After any measurement → $RTO = RTT_S + 4 \times RTT$

Options

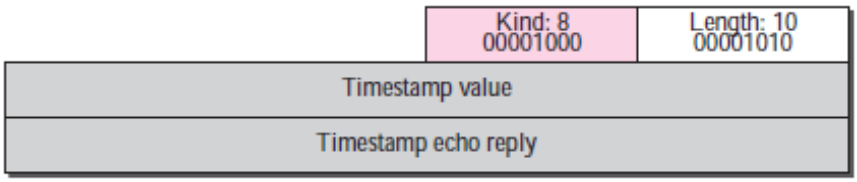
1. End-of-option: All zeros
2. No-operation option: Last bit is 1.
3. Maximum-segment-size option

4. Window-scale-factor option

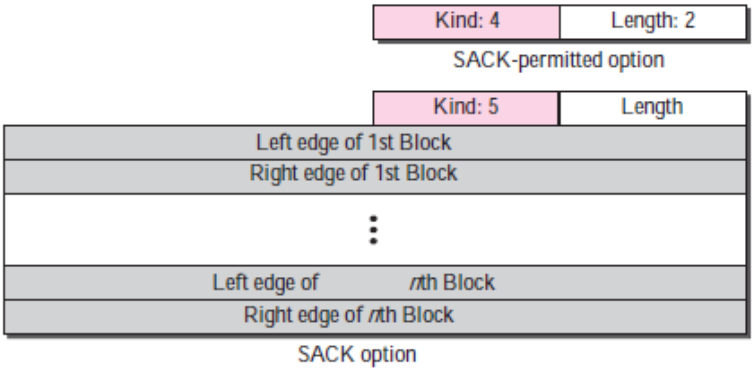


New window size = window size defined in header x $2^{\text{window scale factor}}$

5. Timestamp option

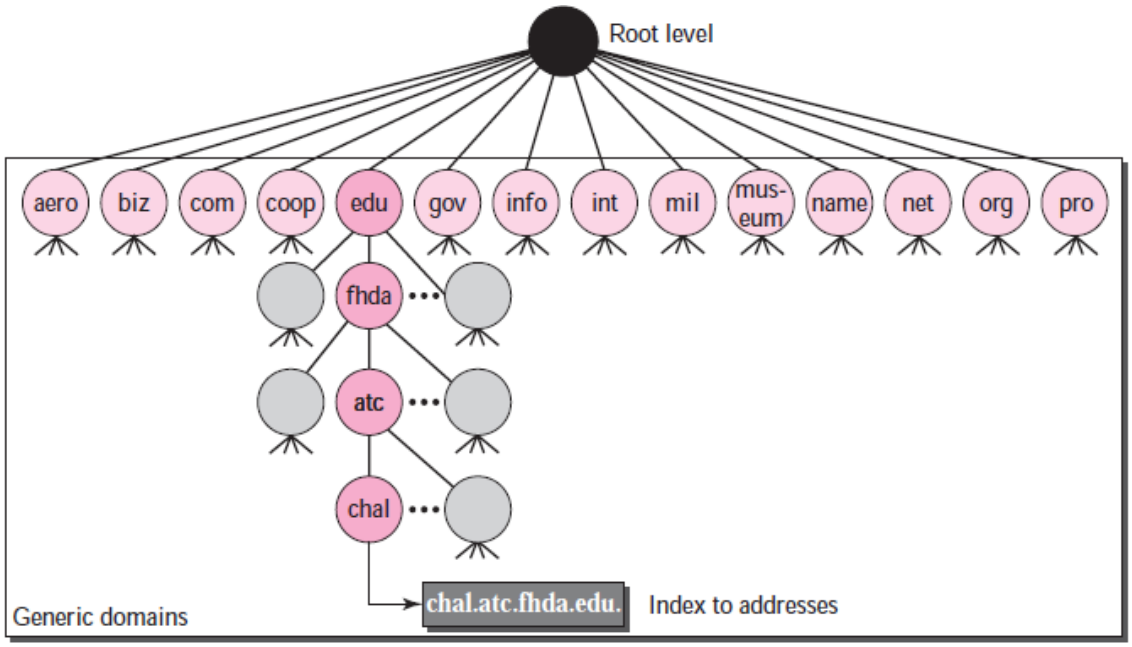


6.SACK



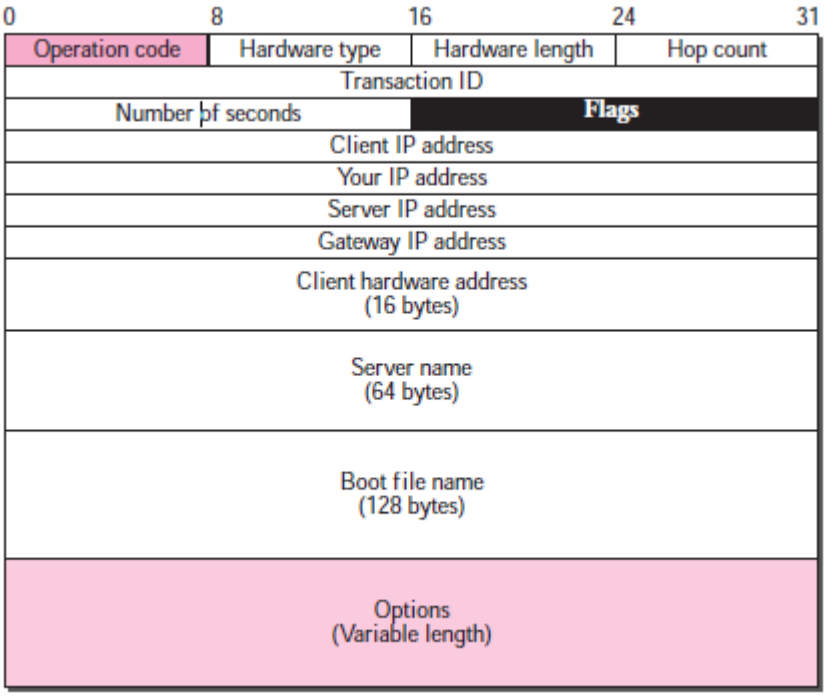
DNS

Generic domains

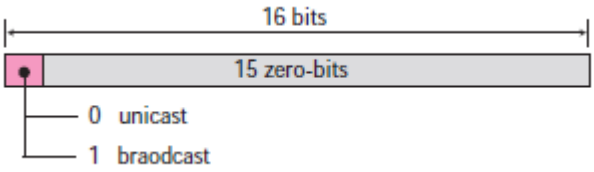


DHCP

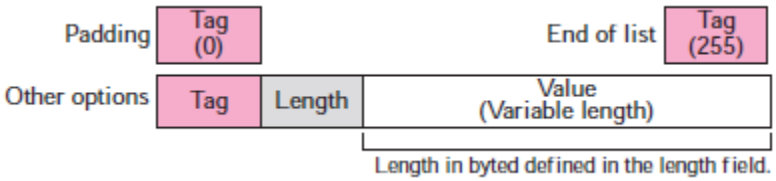
DHCP Packet Format



Flag Format



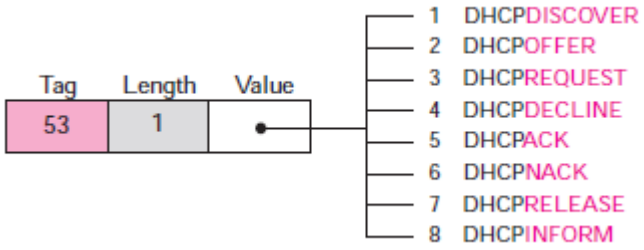
Option Format



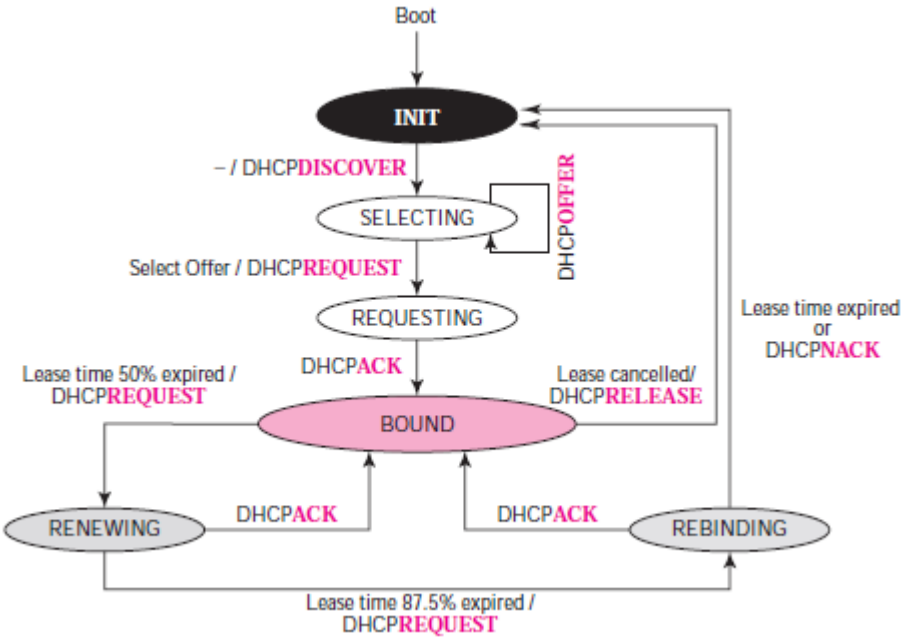
Tables for List of Options

Tag	*Length*	*Value*	*Description*
0			Padding
1	4	Subnet mask	Subnet mask
2	4	Time of the day	Time offset
3	Variable	IP addresses	Default router
4	Variable	IP addresses	Time server
5	Variable	IP addresses	IEN 16 server
6	Variable	IP addresses	DNS server
7	Variable	IP addresses	Log server
8	Variable	IP addresses	Quote server
9	Variable	IP addresses	Print server
10	Variable	IP addresses	Impress
11	Variable	IP addresses	RLP server
12	Variable	DNS name	Host name
13	2	Integer	Boot file size
53	1	Discussed later	Used for dynamic configuration
128–254	Variable	Specific information	Vendor specific
255			End of list

Options with tag 53



DHCP Client Transition Diagram



TELNET

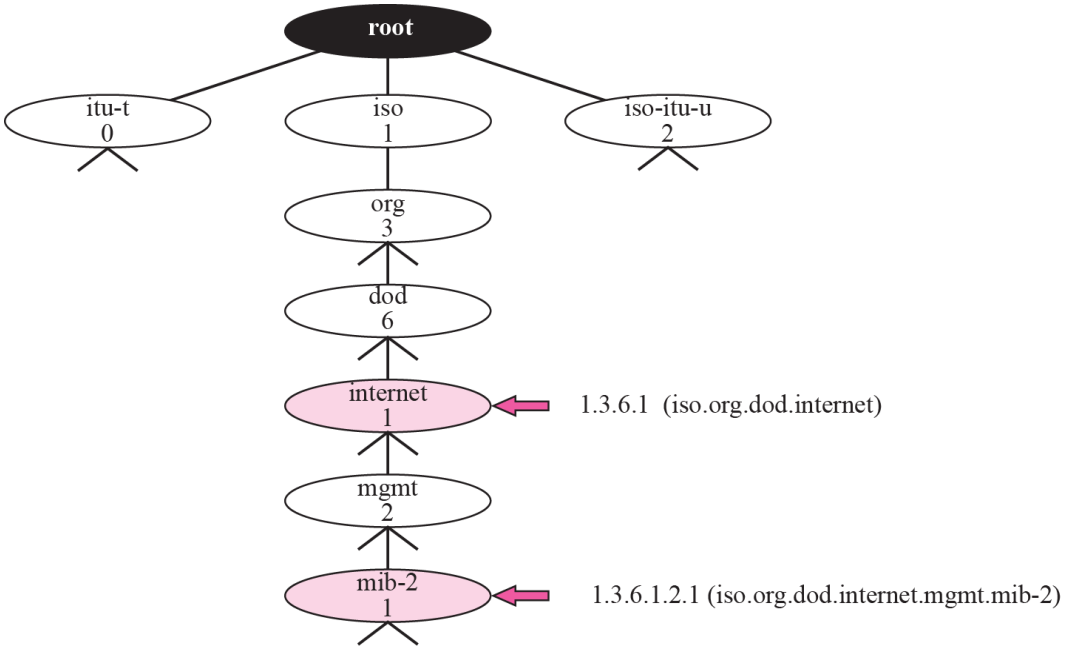
NVT control characters

Character	*Decimal*	*Binary*	*Meaning*
EOF	236	11101100	End of file
EOR	239	11101111	End of record
SE	240	11110000	Suboption end
NOP	241	11110001	No operation
DM	242	11110010	Data mark
BRK	243	11110011	Break
IP	244	11110100	Interrupt process
AO	245	11110101	Abort output
AYT	246	11110110	Are you there?
EC	247	11110111	Erase character
EL	248	11111000	Erase line
GA	249	11111001	Go ahead
SB	250	11111010	Suboption begin
WILL	251	11111011	Agreement to enable option
WONT	252	11111100	Refusal to enable option
DO	253	11111101	Approval to option request
DONT	254	11111110	Denial of option request
IAC	255	11111111	Interpret (the next character) as control

SNMP data type

Type	*Size*	*Description*
INTEGER	4 bytes	An integer with a value between -2^{31} and $2^{31}-1$
Integer32	4 bytes	Same as INTEGER
Unsigned32	4 bytes	Unsigned with a value between 0 and $2^{32}-1$
OCTET STRING	Variable	Byte-string up to 65,535 bytes long
OBJECT IDENTIFIER	Variable	An object identifier
IPAddress	4 bytes	An IP address made of four integers
Counter32	4 bytes	An integer whose value can be incremented from zero to 2^{32}; when it reaches its maximum value it wraps back to zero
Counter64	8 bytes	64-bit counter
Gauge32	4 bytes	Same as Counter32, but when it reaches its maximum value, it does not wrap; it remains there until it is reset
TimeTicks	4 bytes	A counting value that records time in 1/100ths of a second
BITS		A string of bits
Opaque	Variable	Uninterpreted string

Object identifier



SNMP PDU

Type	*Tag (Binary)*	*Tag (Hex)*
GetRequest	**10100000**	**A0**
GetNextRequest	**10100001**	**A1**
Response	**10100010**	**A2**
SetRequest	**10100011**	**A3**
GetBulkRequest	**10100101**	**A5**
InformRequest	**10100110**	**A6**
Trap (SNMPv2)	**10100111**	**A7**
Report	**10101000**	**A8**

DATABASE SYSTEM

ORACLE SQL COMMANDS

SELECT Statement
SELECT [DISNCT] {*, column [alias],...}
FROM table
[WHERE condition(s)]
[ORDER BY {column, exp, alias} [ASC|DESC]]

Cartesian Product
SELECT table1.*, table2.*,[...]
FROM table1,table2[,...]

Natural join
SELECT table1.*,table2.*
FROM table1 NATURAL JOIN table2

Inner join
SELECT table1.*,table2.*
FROM table1 INNER JOIN table2
ON table1.column = table2.column

Join Types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using (A_1, A_2, …)

Aggregation Selecting
SELECT [column,] group_function(column)
FROM table
[WHERE condition]
[GROUP BY group_by_expression]
[HAVING group_condition]
[ORDER BY column] ;

Group function
AVG([DISTINCT|ALL]n)
COUNT(*|[DISTINCT|ALL]expr)
MAX([DISTINCT|ALL]expr)
MIN([DISTINCT|ALL]expr)
SUM([DISTINCT|ALL]n)

Subquery
SELECT select_list
FROM table
WHERE expr operator(SELECT select_list FROM table);

single-row comparison operators
= > >= < <= <>
multiple-row comparison operators
IN ANY ALL

Multiple-column Subqueries
SELECT column, column, ...
FROM table
WHERE (column, column, ...) IN
(SELECT column, column, ...
FROM table WHERE condition) ;

View Definition
create view v as <query expression>;

Manipulating Data
INSERT Statement(one row)
INSERT INTO table [(column [,column...])]
VALUES (value [,value...]) ;

INSERT Statement with Subquery
INSERT INTO table [column(, column)]
subquery ;

UPDATE Statement
UPDATE table
SET column = value [, column = value,...]
[WHERE condition] ;

Updating with Multiple-column Subquery
UPDATE table
SET (column, column,...) =
(SELECT column, column,...
FROM table
WHERE condition)
WHERE condition ;

Deleting Rows with DELETE Statement
DELETE [FROM] table
[WHERE conditon] ;

Deleting Rows Based on Another Table
DELETE FROM table
WHERE column = (SELECT column
FROM table
WHERE condtion) ;

Transaction Control Statements
COMMIT ;
SAVEPOINT name ;
ROLLBACK [TO SAVEPOINT name] ;

CREATE TABLE Statement
CREATE TABLE [schema.]table
(column datatype [DEFAULT expr] [,...]) ;

Datatype
VARCHAR2(size) CHAR(size) NUMBER(p,s) DATE
LONG CLOB RAW LONG RAW
BLOB BFILE

ALTER TABLE Statement (Add columns)
ALTER TABLE table
ADD (column datatype [DEFAULT expr]
[, column datatype]...) ;

Changing a column's type, size and default of a Table
ALTER TABLE table
MODIFY (column datatype [DEFAULT expr]
[, column datatype]...) ;

Dropping a Table
DROP TABLE table ;

Changing the Name of an Object
RENAME old_name TO new_name ;

Truncating a Table
TRUNCATE TABLE table ;

Defining Constraints
CREATE TABLE [schema.]table
(column datatype [DEFAULT expr][NOT NULL]
[column_constraint],...
[table_constraint][,...]) ;

Column constraint level
column [CONSTRAINT constraint_name] constraint_type,

Constraint_type
PRIMARY KEY
REFERENCES table(column)
UNIQUE
CHECK (codition)

Table constraint level(except NOT NULL)
column,...,[CONSTRAINT constraint_name]
constraint_type (column,...),

NOT NULL Constraint (Only Column Level)
CONSTRAINT table[_column...]_nn NOT NULL ...

UNIQUE Key Constraint
CONSTRAINT table[_column..]_uk UNIQUE (column[,...])

PRIMARY Key Constraint
CONSTRAINT table[_column..]_pk PRIMARY (column[,...])

FOREIGN Key Constraint
CONSTRAINT table[_column..]_fk
FOREIGN KEY (column[,...])
REFERENCES table (column[,...])[ON DELETE CASCADE]

CHECK constraint
CONSTRAINT table[_column..]_ck CHECK (condition)

Adding a Constraint(except NOT NULL)
ALTER TABLE table
ADD [CONSTRAINT constraint_name] type (column) ;

Adding a NOT NULL constraint
ALTER TABLE table
MODIFY (column datatype [DEFAULT expr]
[CONSTRAINT constraint_name_nn] NOT NULL) ;

Dropping a Constraint
ALTER TABLE table
DROP CONSTRAINT constraint_name ;
ALTER TABLE table
DROP PRIMARY KEY | UNIQUE (column) |
CONSTRAINT constraint_name [CASCADE] ;

THE ENTITY-RELATIONSHIP MODEL

Entity-relationship (E-R) data model is a widely used data model for database design. It provides a convenient graphical representation to view data, relationships, and constraints.

The E-R model is intended primarily for the database-design process. It was developed to facilitate database design by allowing the specification of an **enterprise schema**. Such a schema represents the overall logical structure of the database. This overall structure can be expressed graphically by an **E-R diagram**.

An **entity** is an object that exists in the real world and is distinguishable from other objects. We express the distinction by associating with each entity a set of attributes that describes the bject.

A **relationship** is an association among several entities. A **relationship set** is a collection of relationships of the same type, and an **entity set** is a collection of entities of the same type.

The terms **superkey**, **candidate key**, and **primary key** apply to entity and relationship sets as they do for relation schemas. Identifying the primary key of a relationship set requires some care, since it is composed of attributes from one or more of the related entity sets.

Mapping cardinalities express the number of entities to which another entity can be associated via a relationship set.

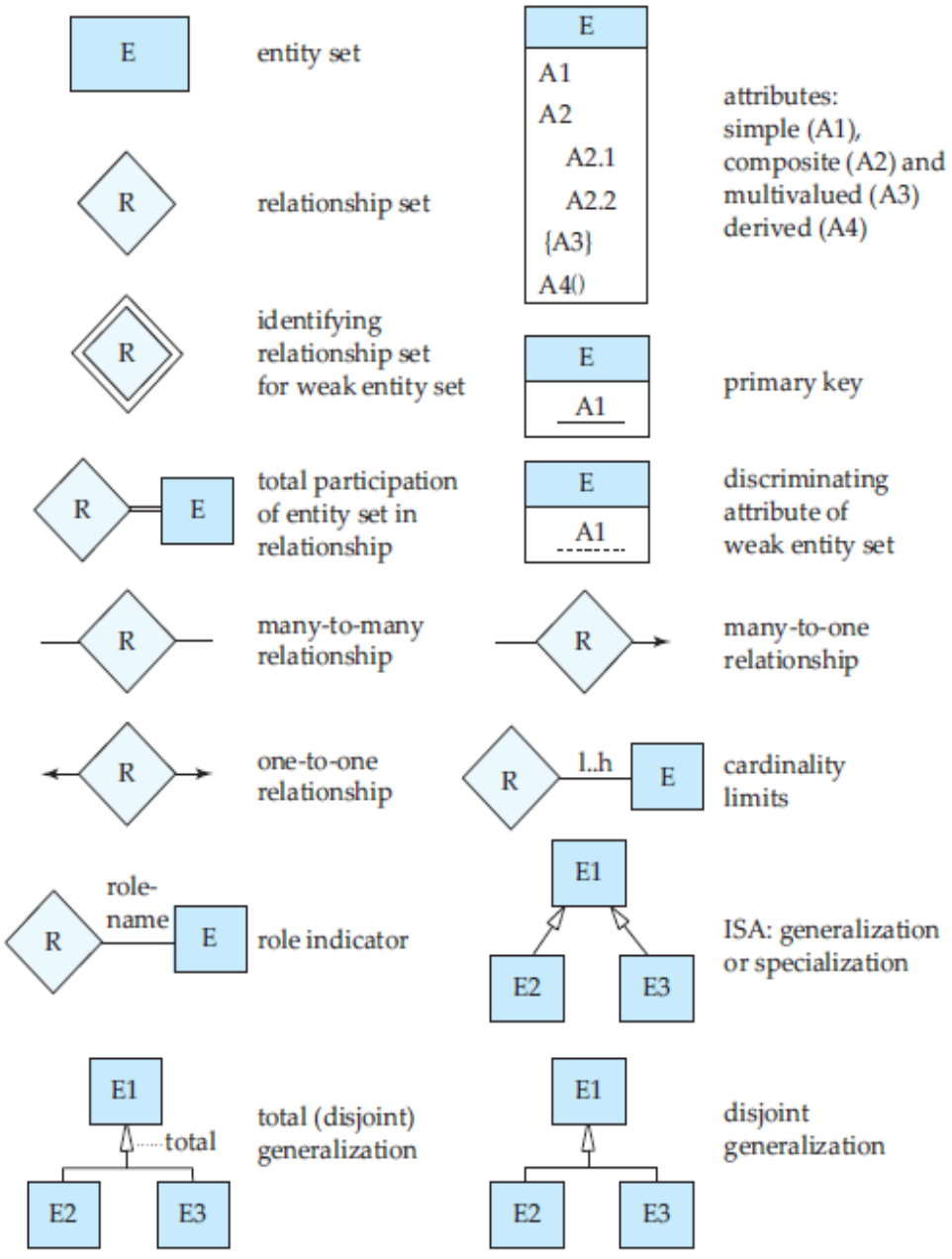
An entity set that does not have sufficient attributes to form a primary key is termed a **weak entity set**. An entity set that has a primary key is termed a **strong entity set**.

The various features of the E-R model offer the database designer numerous choices in how to best represent the enterprise being modeled. Concepts and objects may, in certain cases, be represented by entities, relationships, or attributes. Aspects of the overall structure of the enterprise may be best described by using weak entity sets, generalization, specialization, or aggregation. Often, the designer must weigh the merits of a simple, compact model versus those of a more precise, but more complex, one.

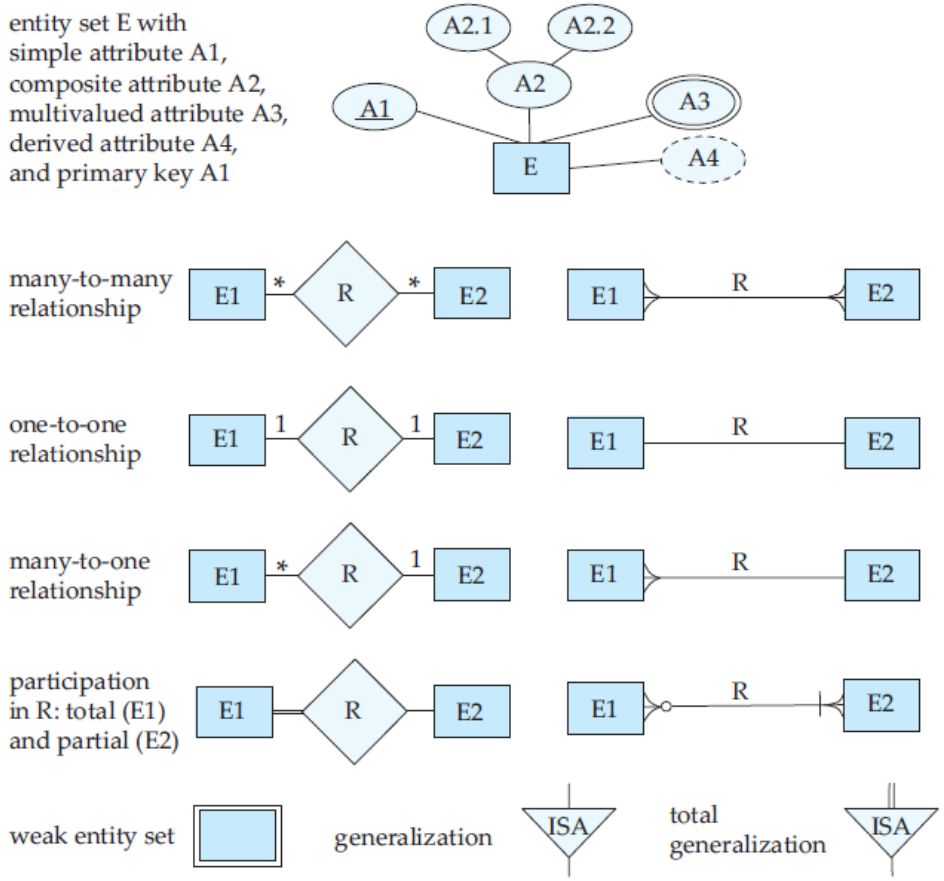
Specialization and **generalization** define a containment relationship between a higher-level entity set and one or more lower-level entity sets. Specialization is the result of taking a subset of a higher-level entity set to form a lower-level entity set. Generalization is the result of taking the union of two or more disjoint (lower-level) entity sets to produce a higher-level entity set.
The attributes of higher-level entity sets are inherited by lower-level entity sets.

Aggregation is an abstraction in which relationship sets (along with their associated entity sets) are treated as higher-level entity sets, and can participate in relationships.

Symbols used in E-R Notations :



Alternative E-R Notations:



RELATIONAL ALGEBRA

Fundamental Operations:

1. Select σ
   Notation: $\sigma_p(r)$ where p is the required condition/predicate on the relation r

2. Project Π
   Notation: $\Pi_{A_1, A_2, \ldots, A_k}(r)$ where A_1 and so on are attributes of relation r

3. Union ∪
   Notation: $r \cup s$ where r and s are the two relations

4. Set difference –
   Notation: $r - s$ where r and s are the two relations

5. Cartesian product x
   Notation: r x s where r and s are the two relations

6. Rename ρ
   Notation: $\rho_x(E)$ where the expression E is returned under the name X

Additional Operations:

1. Set Intersection ∩
   Notation: $r \cap s$ where r and s are two relations
2. Natural Join ⋈
   Notation : r ⋈ s where r and s are two relations
3. Assignment ←
4. Outer Join
   a. Left Outer Join ⟕
   b. Right Outer Join ⟖
   c. Full Outer Join ⟗
5. Division ÷
   Notation: r ÷ s where r and s are two relations

Extended Operations:

1. Generalized Projection $\prod_{F_1, F_2, ..., F_n}(E)$
   Where F_i and so on are arithmetic operations involving constants and attributes in the schema of E
2. Aggregate Functions $G_1, G_2, \ldots, G_n \quad F_1(A_1), F_2(A_2, \ldots, F_n(A_n)(E)$
   Where Gi and so on are the groups on which the aggregate functions Fi and so on are applied for a given attribute Ai of the relational expression E

NORMALIZATION

1NF
A relational schema R is in first normal form(1NF) if the domains of all attributes of R are atomic

Functional Dependency
Let *R* be a relation schema and let $\alpha \subseteq R$ *and* $\beta \subseteq R$
The functional dependency $\alpha \rightarrow \beta$ holds on *R* if and only if for any legal relations *r*(R), whenever any two tuples t_1 and t_2 of *r* agree on the attributes α, they also agree on the attributes β. That is,
$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$

Closure of functional Dependency, F^+
The set of all functional dependencies that can be inferred given the set *F*. F^+ contains all of the functional dependencies in *F*.

BCNF
A relation schema *R* is in BCNF with respect to a set *F* of functional dependencies if for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$ where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:
$\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
α is a super key for *R*

3NF
A relation schema *R* is in third normal form (3NF) if for all: $\alpha \rightarrow \beta$ in F^+ at least one of the following holds:
$\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
α is a super key for *R*
Each attribute *A* in $\beta - \alpha$ is contained in a candidate key for *R*.
(NOTE*:* each attribute may be in a different candidate key)

Armstrong's Axioms:

if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ (reflexivity)
if $\alpha \rightarrow \beta$, then $\gamma\ \alpha \rightarrow \gamma\ \beta$ (augmentation)
if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ (transitivity)

Additional rules:

If $\alpha \rightarrow \beta$ holds *a*nd $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds (union)
If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds (decomposition)
If $\alpha \rightarrow \beta$ holds *a*nd $\gamma\ \beta \rightarrow \delta$ holds, then $\alpha\ \gamma \rightarrow \delta$ holds (pseudo transitivity)

Extraneous Attribute

Consider a set *F* of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in *F*.

Attribute A is extraneous in α if $A \in \alpha$
and *F* logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
Attribute *A* is extraneous in β if $A \in \beta$ and the set of functional dependencies
$(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies *F*.

Canonical cover, F_c

Canonical Cover, F_c for *F* is a set of dependencies F_c such that
F logically implies all dependencies in F_c, and
F_c logically implies all dependencies in *F*, and
No functional dependency in F_c contains an extraneous attribute, and
Each left side of functional dependency in F_c is unique.

Lossless Join Decomposition

A decomposition of *R* into R_1 and R_2 is lossless join if at least one of the following dependencies is in F^+:

$R_1 \cap R_2 \rightarrow R_1$
$R_1 \cap R_2 \rightarrow R_2$

Restriction F to R_i

Let *F* be a set of functional dependencies on a schema *R*, and let *R*1, *R*2, ...*Rn* be a decomposition of *R*. The restriction of *F* to *Ri* is the set *Fi* of all functional dependencies in *F*+ that include *only* attributes of *Ri* .

Dependency Preserving

Let F be a set of functional dependencies on a schema R, and let R1, R2, …...Rn be a decomposition of R. The restriction of F to R1, R2, …...Rn is the set F_1, F_2 …. F_n . The decomposition is **dependency preserving**, if $(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$

Munltivalued Dependency(MVD)

Let *R* be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The multivalued dependency $\alpha \rightarrow\rightarrow \beta$ holds on *R* if in any legal relation *r(R)*, for all pairs for tuples t_1 and t_2 in *r* such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in *r* such that:

$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$
$t_3[\beta] = t_1[\beta]$
$t_3[R - \beta] = t_2[R - \beta]$
$t_4[\beta] = t_2[\beta]$
$t_4[R - \beta] = t_1[R - \beta]$

MVD Rules
For $\alpha, \beta \subseteq R$:

If $\alpha \rightarrow \beta$, then $\alpha \twoheadrightarrow \beta$

If $\alpha \twoheadrightarrow \beta$, then $\alpha \twoheadrightarrow R - \alpha - \beta$

Closure of Multivalued Dependency, D^+
The closure D^+ of *D* is the set of all functional and multivalued dependencies logically implied by *D*.

Fourth Normal Form, 4NF
A relation schema *R* is in **4NF** with respect to a set *D* of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \twoheadrightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:

$\alpha \twoheadrightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)

α is a superkey for schema *R*

Restriction of D to R_i
The restriction of D to R_i is the set D_i consisting of

All functional dependencies in D^+ that include only attributes of R_i

All multivalued dependencies of the form

$$\alpha \twoheadrightarrow (\beta \cap R_i)$$

where $\alpha \subseteq R_i$ and $\alpha \twoheadrightarrow \beta$ is in D^+

PROCEDURAL LANGUAGE

Procedures :

A procedure is a module performing one or more actions; it does not need to return any values. The syntax for creating a procedure is as follows:

```
CREATE OR [REPLACE] PROCEDURE name
        [(parameter[, parameter, ...])]
AS
        [local declarations]
BEGIN
        executable statements
[EXCEPTION
        exception handlers]
END [name];
```

A procedure may have 0 to many parameters.
Every procedure has two parts:

1. The header portion, which comes before **AS** (sometimes you will see IS—they are interchangeable), keyword (this contains the procedure name and the parameter list),
2. The body, which is everything after the AS keyword.

The word REPLACE is optional. When the word REPLACE is not used in the header of the procedure, in order to change the code in the procedure, it must be dropped first and then re-created.
There are three types of parameter modes: IN, OUT, and IN OUT. Modes specify whether the parameter passed is read in or a receptacle for what comes out. IN passes value into the procedure, OUT passes back from the procedure and INOUT does both.

In order to execute a procedure in SQLPlus use the following syntax:
EXECUTE Procedure_name (list of parameters);

Functions:

The syntax for creating a function is as follows:

```
CREATE [OR REPLACE] FUNCTION function_name (parameter list)
        RETURN datatype
IS BEGIN
        <body>
        RETURN (return_value);
END;
/
```

The significant difference between the Procedure and the function is that a function is a PL/SQL **block that *returns* a single value.** Using an anonymous block function is called.

PL/SQL anonymous block:

```
DECLARE
     [Local Variable  declaration section]
BEGIN
        Execution Section
[EXCEPTION
        Exception Section ]
END ;
```

PL/SQL Control Structures:

IF-THEN-ELSIF Statement

```
IF condition1 THEN
  sequence_of_statements1
ELSIF condition2 THEN
  sequence_of_statements2
ELSE
  sequence_of_statements3
END IF;
```

CASE Statement

```
CASE selector variable/Expression
  WHEN expression1 THEN sequence_of_statements1;
  WHEN expression2 THEN sequence_of_statements2;
  ...
  WHEN expressionN THEN sequence_of_statementsN;
 [ELSE sequence_of_statementsN+1;]
END CASE;
```

LOOP

```
LOOP
  sequence_of_statements
  EXIT WHEN boolean_expression;
        -- Instead of EXIT WHEN, use of EXIT statement forces a loop to complete
        unconditionally.
END LOOP;
```

WHILE-LOOP

```
WHILE condition LOOP
  sequence_of_statements
END LOOP;
FOR-LOOP
FOR counter IN [REVERSE] lower_bound..higher_bound LOOP
  sequence_of_statements
END LOOP;
```

Using the %TYPE Attribute:

The %TYPE attribute provides the datatype of a variable or database column.

```
DECLARE
  my_empno employees.employee_id%TYPE;
BEGIN
  my_empno := NULL;
END;
```

Using the %ROWTYPE Attribute:

The %ROWTYPE attribute provides a record type that represents a row in a table (or view).

```
DECLARE
-- %ROWTYPE can include all the columns in a table.
  EMP_REC employees%ROWTYPE;
BEGIN
-- EMP_REC can hold a row from the EMPLOYEES table.
  SELECT * INTO EMP_REC FROM employees WHERE Emp_Id = 10 ;
END;
```

User-defined Exceptions:

```
DECLARE
  my-exception EXCEPTION;
```

Example:

```
DECLARE
      ex_invalid_id  EXCEPTION;
BEGIN
  IF <cond> THEN
       RAISE ex_invalid_id;
```

```
  ELSE
    …..
  END IF;

EXCEPTION
  WHEN ex_invalid_id THEN
       dbms_output.put_line('ID must be greater than zero!');
END;
```

Triggers:

A database trigger is a stored PL/SQL program unit associated with a specific database table. ORACLE executes (fires) a database trigger automatically when a given SQL operation (like INSERT, UPDATE or DELETE) affects the table. Unlike a procedure, or a function, which must be invoked explicitly, database triggers are invoked implicitly.

A database trigger has three parts: A **triggering event**, an **optional trigger constraint**, and a **trigger action**. When an event occurs, a database trigger is fired, and a predefined PL/SQL block will perform the necessary action.

Syntax:

```
CREATE [OR REPLACE] TRIGGER   trigger_name
{BEFORE|AFTER} triggering_event ON table_name
[FOR EACH ROW]
DECLARE

   Declaration statements
        BEGIN
             Executable statements
        EXCEPTION
             Exception-handling statements
         END;
```

The trigger_name references the name of the trigger. BEFORE or AFTER specify when the trigger is fired (before or after the triggering event). The triggering_event references a DML statement issued against the table (e.g., INSERT,DELETE, UPDATE). The table_name is the name of the table associated with the trigger. A trigger may be a ROW or STATEMENT type. If the statement FOR EACH ROW is present in the CREATE TRIGGER clause of a trigger, the trigger is a row trigger. The clause, FOR EACH ROW, specifies a trigger is a row trigger and fires once for each modified row. A statement trigger, however, is fired only once for the triggering statement, regardless of the number of rows affected by the triggering statement

Enabling, Disabling, Dropping Triggers:

SQL>ALTER TRIGGER trigger_name DISABLE; SQL>ALTER TABLE table_name DISABLE ALL TRIGGERS;
SQL>ALTER TABLE table_name ENABLE trigger_name; SQL> ALTER TABLE table_name ENABLE ALL TRIGGERS;
SQL> DROP TRIGGER trigger_name

Cursor:

```
DECLARE
CURSOR <cursor_name> IS <SELECT statement> ;
        <cursor_variable declaration> ;
BEGIN
OPEN <cursor_name>;
FETCH <cursor_name> INTO <cursor_variable>;
    // Process the records
CLOSE <cursor_name>;
END;
```

Packages in PL/SQL:

Packages are schema objects that groups logically related PL/SQL types, variables, and subprograms.
A package will have two mandatory parts:

1. Package specification:
   Example

```
CREATE PACKAGE cust_sal AS
  PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;   /
```

2. Package body or definition:
   Example

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS
  PROCEDURE find_sal(c_id customers.id%TYPE) IS
  c_sal customers.salary%TYPE;
  BEGIN
    SELECT salary INTO c_sal
    FROM customers
    WHERE id = c_id;
    dbms_output.put_line('Salary: '|| c_sal);
  END find_sal;
END cust_sal;
/
```

Usage of Package Elements

```
package_name.element_name;
```

DESIGN AND ANALYSIS OF ALGORITHMS

Properties of Logarithms

1. $log_a 1 = 0$
2. $log_a a = 1$
3. $log_a x^y = y log_a x$
4. $log_a xy = log_a x + log_a y$
5. $log_a \frac{x}{y} = log_a x - log_a y$
6. $a^{log_b x} = x^{log_b a}$
7. $log_a x = \frac{log_b x}{log_b a} = log_a b \; log_b x$

Combinatorics

1. Number of permutations of an n-element set $P(n) = n!$
2. Number of k-combinations of an n-element set: $C(n,k) = \frac{n!}{k!(n-k)!}$
3. Number of subsets of an n-element set: 2^n

Important Summation Formula

1. $\sum_{i=l}^{u} 1 = 1 + 1 + 1 + \cdots + 1 = u - l + 1$ $(l, u \; are \; integer \; limits, l \leq u)$; $\sum_{i=1}^{n} 1 = n$
2. $\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots . + n = \frac{n(n+1)}{2} == \frac{1}{2} n^2$
3. $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots . + n^2 = \frac{n(n+1)(2n+1)}{6} == \frac{1}{3} n^3$
4. $\sum_{i=}^{n} i^k = 1^k + 2^k + \cdots . + n^k = \frac{n(n+1)(2n+1)}{6} == \frac{1}{k+1} n^{k+1}$
5. $\sum_{i=0}^{n} a^i = 1 + a + \cdots . + a^n = \frac{a^{n+1}-1}{a-1}$ $(a \; not \; equal \; to \; 1)$; $\sum_{i=0}^{n} 2^i = (2^{n+1} - 1)$
6. $\sum_{i=1}^{n} i 2^i = 1 * 2 + 2 * 2^2 + \cdots . + n 2^n = (n-1) 2^{n+1} + 2$
7. $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx$ ln n + c, where c ≈ 0.5772... (Euler's constant), Hamonic number
8. $\sum_{i=0}^{\infty} \frac{1}{2^i} = 1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^n} \approx 2$ // geometric series
9. $\sum_{i=1}^{n} \lg i \approx$ n lg n

Sum Manipulation Rules

1. $\sum_{i=l}^{u} C a_i$ =C$\sum_{i=l}^{u} a_i$
2. $\sum_{i=l}^{u} (a_i \pm b_i)$ =$\sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$
3. $\sum_{i=l}^{u} a_i$ =$\sum_{i=l}^{m} a_i + \sum_{i=m+1}^{u} a_i$ where l ≤m < u
4. $\sum_{i=l}^{u} (a_i - a_{i-1}) = a_u - a_{l-1}$

Approximation of a Sum by a Definite Integral

1. $\int_{l-1}^{u} f(x)dx \leq \sum_{i=l}^{u} f(i) \leq \int_{l}^{u+1} f(x)dx$ for a non-decreasing f(x)
2. $\int_{l}^{u+1} f(x)dx \leq \sum_{i=l}^{u} f(i) \leq \int_{l-1}^{u} f(x)dx$ for a non-increasing f(x)

Miscellaneous

1. n≈ $\sqrt{2\pi n} \left(\frac{n}{e}\right)^n$ as n → ∞ (Stirling's formula)

2. Modular arithmetic (n, m are integers, p is a positive integer)
   $(n + m) \bmod p = ((n \bmod p) + (m \bmod p)) \bmod p$
   $(n \times m) \bmod p = ((n \bmod p) \times (m \bmod p)) \bmod p$

Asymptotic Inequality
Constant $C < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots < 2^n < n! < n^n$

Arithmetic Progression
Sum of first n terms of a arithmetic series: a, a+d, a + 2d, ….. , a + (n-1)d is n/2 [2a + (n-1)d]

Geometric Progression
Sum of first n terms of a geometric series a, ar, ar^2, …, ar^{n-1} is
$a(r^n - 1) / (r - 1)$ if $(r > 1)$
$a(1 - r^n) / (1 - r)$ if $(r < 1)$

Strassen's Matrix Multiplication
C = A*B then
$D = A_1 (B_2 - B_4)$
$E = A_4 (B_3 - B_1)$
$F = (A_3 + A_4) B_1$
$G = (A_1 + A_2) B_4$
$H = (A_3 - A_1) (B_1 + B_2)$
$I = (A_2 - A_4) (B_3 + B_4)$
$J = (A_1 + A_4) (B_1 + B_4)$
$C_1 = E + I + J - G$
$C_2 = D + G$
$C_3 = E + F$
$C_4 = D + H + J - F$

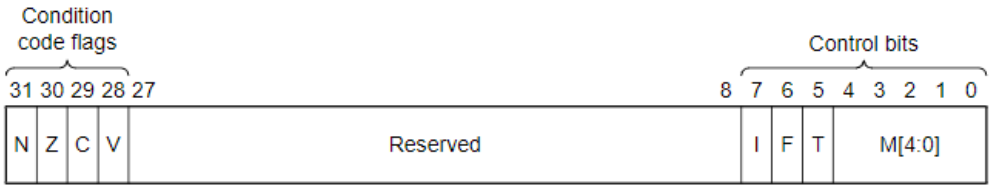
Master's Theorem

If $T(n) = a\,T(n/b) + g(n)$, for $n > 1$ and $T(1)$ for $n = 1$, where $n = b^k$, then
$T(n) = n^{\log_b a} [\, T(1) + f(n) \,]$, where $h(n) = g(n) / n^{\log_b a}$ and $f(n) = \sum_{j=1}^{k} h(b^j)$

Space Requirement

Type Name	32–bit Size	64–bit Size
char	1 byte	1 byte
short	2 bytes	2 bytes
int	4 bytes	4 bytes
long	4 bytes	8 bytes
long long	8 bytes	8 bytes
float	4 bytes	4 bytes
double	8 bytes	8 bytes
long double	16 bytes	16 bytes
near pointer	2 bytes	2 bytes
far pointer	4 bytes	4 bytes

CPSR Structure



M[4:0]	Mode bits	V	Overflow
T	Thumb state	C	Carry
F	FIQ (Fast Interrupt Request)	Z	Zero
I	Interrupt	N	Negetive

ARM Instruction format

[label] mnemonic [operands] [;comment]

Brackets indicate that the field is optional and not all lines have them. Brackets should not be typed in. Regarding the above format, the following points should be noted:

1. The label field allows the program to refer to a line of code by name. The label field cannot exceed a certain number of characters. Check your assembler for the rule.
2. The Assembly language mnemonic (instruction) and operand(s) fields together perform the real work of the program and accomplish the tasks for which the program was written. In Keil uVision, the mnemonic must be typed at least one tab space from the left margin.

Conditional Execution Mnemonics

Mnemonic Extension	Meaning
EQ	Equal
NE	Not equal
CS/HS	Carry Set/Higher or Same
CC/LO	Carry Clear/Lower
MI	Minus/Negative
PL	Plus
VS	V Set (Overflow)
VC	V Clear (No Overflow)
HI	Higher
HS	Lower or Same

GE	Greater than or Equal
LT	Less than
GT	Greater than
LE	Less than or Equal
AL	Always (unconditional)

SRAM Bit-Addressable Memory Region

The bit-band SRAM addresses 0x20000000 to 0x200FFFFF (1M bytes) is given alias addresses of 0x22000000 to 0x23FFFFFF.

GPIO extension connectors:

CNA

Pin CNA	PIN LPC1768	Description
1	81	P0.4/I2SRX_CLK/RD2/CAP2.0
2	80	P0.5/I2SRX_WS/TD2/CAP2.1
3	79	P0.6/I2SRX_SDA/SSEL1/MAT2.0
4	78	P0.7/I2STX_CLK/SCK1//MAT2.1
5	77	P0.8/I2STX_WS/MISO1/MAT2.2
6	76	P0.9/I2STX_SDA/MOSI1/MAT2.3
7	48	P0.10/TXD2/SDA2/MAT3.0
8	49	P0.11/RXD2/SCL2/MAT3.1
9	-	No connection
10	-	Ground

CNB

Pin CNB	Pin LPC1768	Description
1	37	P1.23/MCI1/PWM1.4/MISO0
2	38	P1.24/MCI2/PWM1.5/MOSI0
3	39	P1.25/MCOA1/MAT1.1
4	40	P1.26/MCOB1/PWM1.6/CAP0.0
5	53	P2.10/EINT0/NMI
6	52	P2.11/EINT1/I2STX_CLK
7	51	P2.12/EINT2/I2STX_WS
8	50	P2.13/EINT3/I2STX_SDA
9	-	No connection
10	-	Ground

Short JP4(1, 2) to use CNB pin 8.

CNC

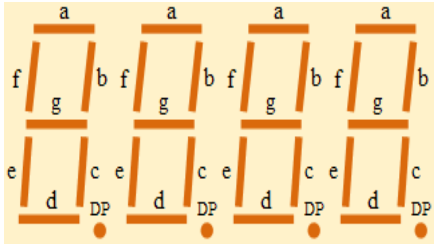
Pin CNC	Pin LPC1768	Description
1	62	P0.15/TXD1/SCK0/SCK
2	63	P0.16/RXD1/SSEL0/SSEL
3	61	P0.17/CTS1/MISO0/MISO
4	60	P0.18/DCD1/MOSI0/MOSI
5	59	P0.19/DSR1/SDA1
6	58	P0.20/DTR1/SCL1
7	57	P0.21/RI1/RD1
8	56	P0.22/RTS1/TD1
9	50	P2.13/I2STX_SDA
10	-	Ground

Short JP4(2, 3) to work CNC pin 9.

CND

Pin CND	Pin LPC1768	Description
1	9	P0.23/AD0.0/I2SRX_CLK/CAP3.0
2	8	P0.24/AD0.1/I2SRX_WS/CAP3.1
3	7	P0.25/AD0.2/I2SRX_SDA/TXD3
4	6	P0.26/AD0.3/AOUT/RXD3
5	25	P0.27/SDA0/USB/SDA
6	24	P0.28/SCL0/USB_SCL
7	75	P2.0/PWM1.1/TXD1
8	74	P2.1/PWM1.2/RXD1
9	-	No connection
10	-	Ground

Seven Segment Display Interface



Hex codes for displaying hex digits from 0 to F

0	0x3F	4	0x66	8	0x7F	C	0x39
1	0x06	5	0x6D	9	0x6F	d	0x5E
2	0x5B	6	0x7D	A	0x77	E	0x79
3	0x4F	7	0x07	b	0x7C	F	0x71

Control and Display commands for LCD

S. No.	Hex Code	Command to LCD Instruction Register
1	01	Clear Display
2	02	Return Home
3	04	Decrement Cursor (Shift cursor to left)
4	06	Increment Cursor (Shift Cursor to right)
5	05	Shift Display right
6	07	Shift Display left
7	08	Display Off, Cursor Off
8	0A	Display Off, Cursor On
9	0C	Display On, Cursor Off
10	0F	Display On, Cursor Blinking
11	10	Shift cursor position to left
12	14	Shift cursor position to right
13	18	Shift the entire display to the left
14	1C	Shift the entire display to the right
15	80	Force the cursor to the beginning of 1st line
16	C0	Force the cursor to the beginning of 2nd line
17	33 and 32	To initialize the lcd to 4-bit mode
18	28	Function set (Data length = 4-bit; Number of display lines=2; Display format=5*7 Dot matrix)

ADC

Va = (Vref/2^{12})* Dval
Where Va is the analog input, Vref is the reference voltage and Dval is the digital output.
Power control bit for ADC in PCONP register is 12. Its value on reset is zero.

DAC

V_{AOUT} = (VALUE * (V_{REFP}-V_{REFN})/1024) + V_{REFN}
If V_{REFN} = 0,
V_{AOUT} = VALUE * V_{REF}/1024
where VALUE is the 10-bit digital value which is to be converted into its analog counterpart and V_{REF} is the input reference voltage.

PWM and Timer

T_{PCLK} = 1/$PCLK_{Hz,}$ where PCLK is the peripheral clock.
T_{RES} = (PR+1)/$PCLK_{Hz}$, where T_{RES} is the timer resolution and PR is the Prescale Register value.
PR = ($PCLK_{Hz}$ * T_{RES}) – 1

Stepper motor

Steps per full rotation=Number of rotor teeth * Number of stators

UART

UART0/2/3 baud rate can be calculated as (n = 0/2/3):

$$UARTn_{baudrate} = \frac{PCLK}{16 \times (256 \times UnDLM + UnDLL) \times \left(1 + \frac{DivAddVal}{MulVal}\right)}$$

Where DivAddVal & MulVal are part of “Fractional Rate Divider” or “Baud-Prescaler” which is used in Baud-Rate generation.

Fractional Divider setting look-up table

FR	DivAddVal/ MulVal	FR	DivAddVal/ MulVal	FR	DivAddVal/ MulVal	FR	DivAddVal/ MulVal
1.000	0/1	1.250	1/4	1.500	1/2	1.750	3/4
1.067	1/15	1.267	4/15	1.533	8/15	1.769	10/13
1.071	1/14	1.273	3/11	1.538	7/13	1.778	7/9
1.077	1/13	1.286	2/7	1.545	6/11	1.786	11/14
1.083	1/12	1.300	3/10	1.556	5/9	1.800	4/5
1.091	1/11	1.308	4/13	1.571	4/7	1.818	9/11
1.100	1/10	1.333	1/3	1.583	7/12	1.833	5/6
1.111	1/9	1.357	5/14	1.600	3/5	1.846	11/13
1.125	1/8	1.364	4/11	1.615	8/13	1.857	6/7
1.133	2/15	1.375	3/8	1.625	5/8	1.867	13/15
1.143	1/7	1.385	5/13	1.636	7/11	1.875	7/8
1.154	2/13	1.400	2/5	1.643	9/14	1.889	8/9
1.167	1/6	1.417	5/12	1.667	2/3	1.900	9/10
1.182	2/11	1.429	3/7	1.692	9/13	1.909	10/11
1.200	1/5	1.444	4/9	1.700	7/10	1.917	11/12
1.214	3/14	1.455	5/11	1.714	5/7	1.923	12/13
1.222	2/9	1.462	6/13	1.727	8/11	1.929	13/14
1.231	3/13	1.467	7/15	1.733	11/15	1.933	14/15

FORMAL LANGUAGES AND AUTOMATA THEORY

Languages, Grammars and Automata

Alphabet: Finite nonempty set Σ of symbols, called the alphabet.
Examples: $\sum = \{a, b\}$, $\sum = \{0, 1\}$, and ASCII Characters
Strings: Finite sequence of symbols from the alphabet.
Example: If the alphabet is $\sum = \{a, b\}$, then aab, & bbaba are strings on $\sum$.
We use lowercase letters a, b, c,… for elements of Σ & u, v, w… for string names
String Operations:
Concatenation: The concatenation of two strings w and v is the string obtained by appending the symbols of v to the right end of w.
Example: If $w = a_1a_2\ldots a_n$ and $v = b_1b_2\ldots b_m$, then concatenation of w and v, given by

$$wv = a_1a_2\ldots a_nb_1b_2\ldots b_m$$

Reverse: The reverse of a string is obtained by writing the symbols in reverse order
Example: The reverse of w is given by

$$w^R = a_{n\ldots}a_2a_1$$

String Length: The length of a string w, denoted by |w|, is the number of symbols in the string
Example: The length string w is |w| = n
Empty string: A string with no symbols and it is denoted by λ. The length of empty string is $|\lambda| = 0$
Substring: Any string of consecutive symbols in some string w is said to be a substring of w.
Prefix and suffix of a String: If w = uv, then the substrings v and u are said to be a prefix and a suffix of w, respectively.
w^n operation: If w is a string, then w^n stands for the string obtained by repeating w n times. As a special case, we define $w^0 = \lambda$ and $w^1 = w$.
Star-closure (*) operation (zero or more): If ∑ is alphabet. Then ∑* denote the set of strings obtained by concatenating zero or more symbols from ∑.
Example: If $\sum = \{a, b\}$, then $\sum^* = \{\lambda, a, b, aa, ab, ba, aaa, aab, \ldots\}$
Positive-closure (+) operation (one or more): The set of all possible strings from alphabet ∑ except λ. It is given by $\sum^+ = \sum^* - \lambda$.

Languages

A formal language is a set of strings over a finite alphabet.
In other words a language is any subset of ∑*.
Example: If $\sum = \{a, b\}$, then $\sum^* = \{\lambda, a, b, aa, ab, ba, aaa, aab, \ldots\}$. The set {a, aa, aab} is a language on ∑. The set $L = \{a^nb^n : n \geq 0\}$ is also a language on ∑.
Sentence: A string in a language L will be called a sentence of L.
Since languages are sets, the union, intersection, and difference of two languages are simultaneously defined.
Complement: The complement of a language is defined with respect to ∑*; that is, the complement of L is

$$\overline{L} = \Sigma^* - L$$

Reverse: The reverse of a language is the set of all strings reversals, that is,

$$L^R = \{ w^R : w \in L\}$$

Concatenation: The concatenation of two languages L_1 and L_2 is the set of all strings obtained by concatenating any element of L_1 with any element of L_2. It is given by

$$L_1L_2 = \{xy: x \in L_1, y \in L_2 \}$$

L^n operation: We define L^n as L concatenated with itself n times, with special cases

$$L^0 = \{\lambda\} \text{ and } L^1 = L \text{ for every language L}$$

Star closure and Positive closure: We define the star-closure of a language as

$L^* = L^0 \cup L^1 \cup L^2 \ldots$

and the positive closure as

$L^+ = L^1 \cup L^2 \ldots$

Grammars

Definition: Mathematically, a grammar G is defined as a quadruple:

$G = (V, T, S, P)$

Where V is a finite set of objects called variables
T is a finite set of objects called terminal symbols
$S \in V$ is a special symbol called the Start symbol
P is a finite set of productions or "production rules"
Sets V and T are nonempty and disjoint
All the production rules have the form:

$x \rightarrow y$

where x is an element of $(V \cup T)^+$ and y is in $(V \cup T)^*$

Definition: Let G = (V, T, S, P) be a grammar. Then the set

$$L(G) = \left\{ w \in T^* : S \overset{*}{\Rightarrow} w \right\}$$

is the language generated by G.

Automata

An automaton is an abstract model of a digital computer
An automaton has (Refer Figure 1 below)

- Input file
- Control unit (with finite states)
- Temporary storage
- Output

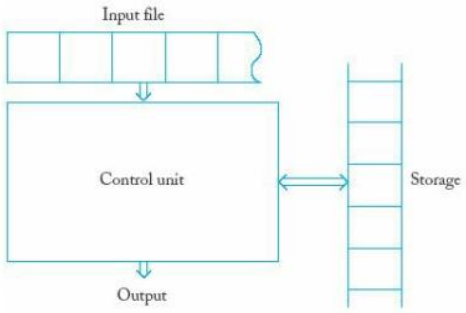


Figure 1

Finite Automata

Finite automata (FA) consists of a finite set of states and a set of transitions from one state to another state that occur on input symbols from an alphabet
Two types of Finite Automata:
Deterministic Finite automata (DFA): Has unique transition from one state to another state
Nondeterministic Finite Automata (NFA): May have several possible transitions from one state to another state and may have λ transition

Deterministic Finite automata (DFA):

A DFA is mathematically defined by a quintuple M = (Q, ∑, δ, q_0, F)

where, Q is a finite set of states

∑ is finite set of input symbols called the input alphabet

δ : Q x ∑ ⟶ Q is the transition function

$q_0 \in Q$ is the initial state or start state

$F \subseteq Q$ is a set of final states or accepting states

Definition: The language accepted by a DFA M = (Q, ∑, δ, q_0, F) is the set of all strings on Σ accepted by M. That is

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}$$

Definition: A language L is called *regular* if and only if there exists some DFA M such that L = L(M).

Nondeterministic Finite Automata (NFA):

Definition: A NFA is mathematically defined by the quintuple M = (Q, ∑, δ, q_0, F), where Q, ∑, q_0, and F are defined as before (See DFA definition), but the transition function is defined by

δ: Q x (∑ ∪{λ}) $\longrightarrow 2^Q$

Definition: The language L accepted by an NFA M = (Q, ∑, δ, q_0, F) is defined as

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \phi\}$$

Equivalence of NFA and DFA

Theorem: Let L be the language accepted by a NFA $M_N = (Q_N, \sum, \delta_N, q_0, F_N)$. Then there exists a DFA $M_D = (Q_D, \sum, \delta_D, q_0, F_D)$ such that $L = L(M_D)$

Regular languages and regular grammars

Regular Expressions: The regular language (RL) can be easily described by simple expressions called regular expressions (RE)

Each RE r denotes a RL L(r)

Note: We use + to denote denotes union,

We use . for concatenation and

We use * for star closure

Examples:

1. ∅ is a RE corresponding to the language L ={ ∅}.
2. λ is a RE corresponding to the language L= {λ}.
3. For each symbol a ∈ Σ, a is a regular expression.

Definition: Let Σ be a given alphabet. Then

1. Φ, λ, and a ∈ Σ are all REs. These are called primitive REs.
2. If r1 & r2 are REs, so are r1+ r2, r1.r2, r1*, and (r1).
3. A string is a RE if and only if it can be derived from primitive REs by a finite number of applications of the rules in (2).

Languages associated with regular expressions

Definition: If r1 and r2 are REs corresponding to the RLs L(r1) and L(r2), respectively, then

1. (r1 + r2) is a RE corresponds to the RL L(r1+r2) = L(r1)UL(r2)
2. (r1r2) is a RE corresponds to the RL L(r1.r2) = L(r1)L(r2)
3. (r1*) is RE corresponds to the RL L(r1*) = (L(r1))*

Theorem: Let r be a regular expression. Then there exits some NFA that accepts L(r). Consequently, L(r) is a RL

Generalized Transition Graph

A generalized transition graph (GTG) is a transition graph (TG) whose edges are labeled with regular expressions.

Any transition graph is reduced to two-state transition graph shown in Figure 2 and the regular expression for the same is given by: $r_1^* r_2(r_3 + r_4 r_1^* r_2)^*$

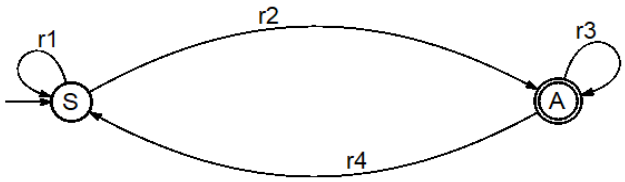


Figure 2

Theorem: Let L be a regular language. Then there exists a regular expression r such that L = L(r)

Regular Grammars

Regular grammar is another way of representing the regular language.

Definition: A grammar G = (V, T, S, P) is said to be linear grammar if all the productions contain at most one variable on the right side of the production, without restriction on the position of the variable.

Definition: A grammar G = (V, T, S, P) is said to be right-linear if all the productions are of the form

A→ **xB** or **A**→ **x**, where A, B ∈ V and x ∈ T*.

Definition: A grammar G = (V, T, S, P) is said to be left-linear if all the productions are of the form

A→ **Bx** or **A**→ **x**, where A, B ∈ V and x ∈ T*.

Definition: A regular grammar is one that is either right-linear or left-linear.

Theorem: Let G = (V, T, S, P) be a right-liner grammar. Then L(G) is a regular language.

Theorem: A language L is a regular if and only if there exists a right-linear grammar (or left-linear grammar) G = (V, T, S, P) such that L = L(G).

Theorem: A language is regular if and only if there exists a regular grammar G such that L = L(G)

Properties of regular languages

If L_1 and L_2 are regular languages, then so are

1. Union: $L_1 \cup L_2$
2. Intersection: $L_1 \cap L_2$
3. Concatenation: L_1L_2
4. Complement: $\overline{L_1}$
5. Star: : L_1^*
6. Difference: $L_1 - L_2$
7. Reversal: L_1^R

Homomorphism

Homomorphism is a substitution in which a single letter is replaced with a string.

Suppose Σ and Γ are alphabets, then a function h: $\Sigma \rightarrow \Gamma^*$ is called a homomorphism. If $w = a_1a_2\ldots a_n$ then $h(w) = h(a_1)h(a_2)\ldots h(a_n)$.
If L is a language on Σ, then its homomorphic image is defined as $h(L)= \{h(w) : w \in L\}$.

Theorem: The family of regular languages is closed under homomorphism.

Right Quotient
Let L_1 and L_2 be languages on the same alphabet, then the right quotient of L_1 and L_2 is defined as $L_1/L_2 = \{x: xy \in L_1 \text{ for some } y \in L_2 \}$

Theorem: The family of regular languages is closed under right quotient with a regular language.

Pumping Lemma: Given an infinite regular language L there exists some positive integer m such that for any string $w \in L$ with length $|w| \geq m$ can be decomposed as $w = xyz$ with $|xy| \leq m$ and $|y| \geq 1$ such that $w_i = xy^iz \in L$ for all $i = 0, 1, 2,\ldots$ This is known as the pumping lemma for regular languages.
Pumping lemma can be used to prove that certain languages are not regular.

Context free lanuages
Definition: A grammar G = (V, T, S, P) is said to be context-free if all productions in P have the form

$A \rightarrow x$, where $A \in V$ and $x \in (V \cup T)^*$.

A language L is said to be context-free if and only if there is a context-free grammar G such that L= L(G).

Sentential Form: A sentence that contains variables and terminals
Leftmost and Rightmost derivations

Definition: A derivation is said to be leftmost if in each step the leftmost variable in the sentential form is replaced. If in each step the rightmost variable is replaced, then the derivation is called rightmost derivation.

Derivation Trees
A derivation tree is an ordered tree in which the nodes are labeled with the left sides of the production and in which the children of a node represent its corresponding right side.
Simple grammar or s-grammar

Definition: A CFG G = (V, T, S, P) is said to be a simple grammar or s-grammar if all its productions are of the form

$A \rightarrow ax,$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Ambiguity in Grammars and Languages
Definition: A context free grammar G is said to be ambiguous if there exist some $w \epsilon L(G)$ that has two or more leftmost derivations or rightmost derivations.

Simplification of context-free grammars

Useless Productions, λ-Productions and Unit Productions

Definition: Let G = (V, T, S, P) be a CFG. A variable A ∈ V is said to be useful if and only if there is at least one w ∈ L(G) such that

$S \rightarrow xAy \rightarrow w$ with x, y in (VUT)*

In other words, a variable is useful if and only if it occurs in at least one derivation. A variable that is not useful is called useless. A production is useless if it involves any useless variable.

Definition: Any production of a CFG of the form

$A \rightarrow \lambda$

is called a λ-production.

Definition: Any production of a context-free grammar of the form

$A \rightarrow B$

where A, B ∈ V, is called a unit-production

Chomsky Normal Form

Definition: A context free grammar is in Chomsky normal form, if all productions are of the form

$A \rightarrow BC$ or $A \rightarrow a$

where A, B, C are in V(non-terminals) and a is in T(terminal)

Greibach Normal Form

Definition: A context free grammar is said to be in Greibach normal form if all productions have the form

$A \rightarrow ax$

where a ∈T and x ∈ V*.

Nondeterministic Pushdown Automata

A schematic representation of a pushdown automaton is given in Figure 3 below: It has input file, control unit and a temporary storage stack.

Each move of the control unit reads a symbol from the input file while at the same time changing the contents of the stack through the usual stack operations.

Each move of the control unit is determined by the current input symbol currently on top of the stack.

The result of the move is a new state of the control unit and a change in the top of the stack.

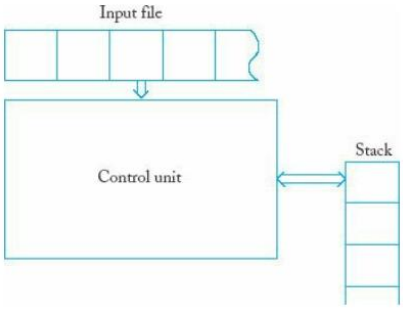


Figure 3

Definition: A nondeterministic pushdown acceptor (NPDA) is mathematically defined by the septuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F),$$

where
Q is a finite set of internal states of the control unit,
Σ is the input alphabet,
Γ is a finite set of symbols called the stack alphabet,
$\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow$ set of finite subsets of $Q \times \Gamma^*$ is the transition function,
$q_0 \in Q$ is the initial state of the control unit,
$z \in \Gamma$ is the stack start symbol,
$F \subseteq Q$ is the set of final states.

Definition: Let $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ be a nondeterministic pushdown automaton. The language accepted by M is the set

$$L(M) = \{w \in \Sigma^* : (q_0, w, z) \vdash^*_M (p, \lambda, u), p \in F, u \in \Gamma^*\}$$

In words, the language accepted by M is the set of all strings that can put M into a final state at the end of a string. The final stack content u is irrelevant to this definition of acceptance.

Theorem: For any context-free language L, there exits an NPDA M, such that L = L(M).

Deterministic Pushdown Automata (dpda) and Deterministic Context-free Languages

Definition: A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ is said to be deterministic if for every $q \in Q$, $a \in \Sigma \cup \{\lambda\}$ and $b \in \Gamma$,
1. $\delta(q, a, b)$ contains at most one element,
2. if $\delta(q, \lambda, b)$ is not empty, then $\delta(q, c, b)$ must be empty for every $c \in \Sigma$.
The first of these conditions simply requires that for any given input symbol and any stack top, at most one move can be made. The second condition is that when a λ-move is possible for some configuration, no input-consuming alternative is available.

Definition: A language is said to be a deterministic CFL if and only if there exists a DPDA M such that L = L(M)

Properties of Context-free Languages

Pumping Lemma: Let L be an infinite context-free language. Then there exists some positive integer m such that any $w \in L$ with $|w| \geq m$ can be decomposed as $w = uvxyz$, with $|vxy| \leq m$, and $|vy| \geq 1$, such that $uv^i xy^i z \in L$, for all $i = 0, 1, 2, \ldots$ This is known as the pumping lemma for context-free languages. The pumping lemma can be used to prove that certain languages are not context-free.

Closure Properties and Decision Algorithms for Context-free Languages

Theorem: The family of context-free languages is closed under union, concatenation, and star-closure.

Theorem: The family of context-free languages is not closed under intersection and complementation.

Theorem: Let L_1 be a context-free language and L_2 be a regular language. Then $L_1 \cap L_2$ is context-free.

TURING MACHINES

The Standard Turing Machine

A Turing machine is an automaton whose temporary storage is a tape. This tape is divided into cells, each of which is capable of holding one symbol. Associated with the tape is a read-write head that can travel right or left on the tape and that can read and write a single symbol on each move.

Turing machine's storage is actually quite simple. It can be visualized as a single, one-dimensional array of cells, each of which can hold a single symbol. This array extends indefinitely in both directions and is therefore capable of holding an unlimited amount of information. The information can be read and changed in any order.
A diagram giving institutive visualization of a Turing machine is shown in Figure 4

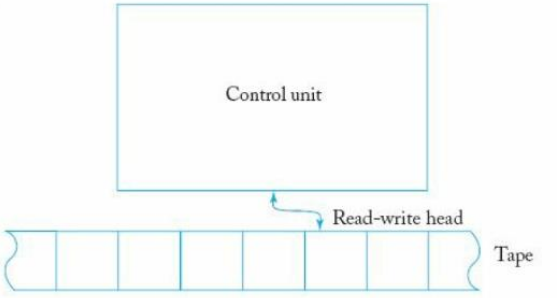


Figure 4

Definition: A Turing machine *M* is mathematically defined by

$M = (Q, \Sigma, \Gamma, \delta, q0, \square, F)$**,**

Where

Q is the set of internal states,
Σ is the input alphabet
Γ is the finite set of symbols called the tape alphabet,
δ is the transition function,
$\square \in \Gamma$ is a special symbol called the blank,
$q_0 \in Q$ is the initial state,
$F \subseteq Q$ is the set of final states.

The transition function δ is defined as

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}.$$

Figure 5 below shows the situation before and after the move

$$\delta\,(q_0, a) = (q_1, d, R).$$

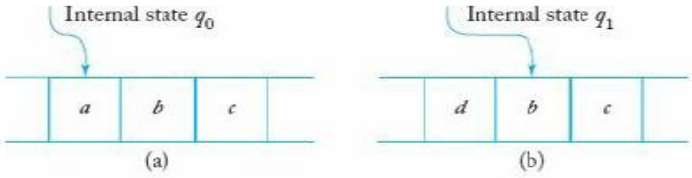


Figure 5: The situation (a) before the move and (b) after the move.

Definition: Let M= (Q, Σ, Γ, δ; q0,□, F) be a Turing machine. Then the language accepted by M is

$$L(M) = \left\{ w \in \Sigma^+ : q_0 w \overset{*}{\vdash} x_1 q_f x_2 \text{ for some } q_f \in F, x_1, x_2 \in \Gamma^* \right\}$$

Definition: **A** function f with domain D is said to be Turing-computable or just computable if there exists some Turing machine M = (Q, Σ, Γ, δ, q0,□, F) such that

$$q_0 w \overset{*}{\vdash}_M q_f f(w), \qquad q_f \in F,$$

for all w ∈ D .

Nondeterministic Turing Machines (ntm)

Definition: A nondeterministic Turing machine is a Turing machine where transition function δ is defined by

$$\delta : Q \times \Gamma \longrightarrow 2^{Q \times \Gamma \times \{L, R\}}.$$

Linear Bounded Automata (lba)

Definition: Definition: A LBA is a nondeterministic TM M = (Q, Σ, Γ, δ, q0, □, F) subject to the restriction that Σ must contain two special symbols, left-end marker [and the right-end marker], such that

$\delta(q_i, [ ) = (q_j, [, R)$ and $\delta(q_i,]) = (q_j,], L)$

Recursive and recursively enumerable languages

Definition: A language L is said to be recursively enumerable if there exists a Turing machine that accepts it.

Definition: A language is recursive if some Turing machine accepts it and halts on any input string.

Theorem: There exists a recursively enumerable language whose complement is not recursively enumerable.

Theorem: There exists a recursively enumerable language that is not recursive.

Unrestricted Grammars

Definition: A grammar G = (V, T, S, P) is called unrestricted if all the productions are of the form

$$\mathbf{u} \rightarrow \mathbf{v},$$

where u is in (V ∪ T)+ and v is in (V ∪ T)*.

Theorem: Any language generated by an unrestricted grammar is recursively enumerable.

Theorem: For every recursively enumerable language L, there exists an unrestricted grammar G, such that L =L (G).

Context-sensitive Grammars and languages

Definition: A grammar G = (V, T, S, P) is said to be context-sensitive if all productions are of the form

$$x \rightarrow y,$$

where $x, y \in (V \cup T)^+$ and $|x| \leq |y|$.

Context-sensitive Languages and Linear Bounded Automata

Definition: A language L is said to be context-sensitive if there exists a context-sensitive grammar G, such that L = L (G) or L = L (G) ∪{λ}.

Theorem: For every context-sensitive language L not including λ, there exists some linear bounded automaton M such that L = L (M).

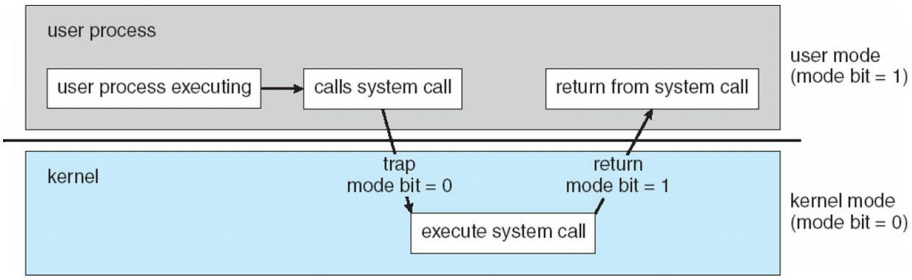
Theorem: If a language *L* is accepted by some linear bounded automaton *M*, then there exists a context-sensitive grammar that generates *L*.

Relation between Recursive and Context-sensitive Languages

Theorem: Every context-sensitive language *L* is recursive.

Theorem: There exists a recursive language that is not context-sensitive.

Operating system dual mode operation:

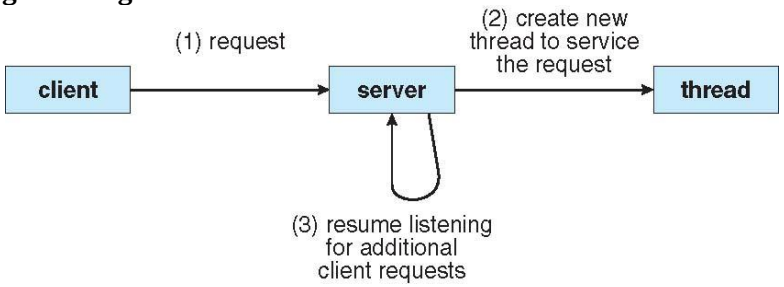


Examples of Windows and Unix System Calls

	WINDOWS	**UNIX**
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

PROCESS MANAGEMENT

Multithreaded programming



Multithreaded server architecture

Process Scheduling

Turnaround time = Completion Time - Arrival Time
Waiting Time = Turnaround Time - Burst Time
Determining Length of Next CPU Burst
$\tau_{n+1} = \alpha\, t_n + (1- \alpha)\, \tau_n$
t_n= actual length of nth CPU burst
τ_{n+1}= Predicted value of next CPU burst
$\alpha, 0 \leq \alpha \leq 1$

Synchronization

Semaphore

Counting semaphore – integer value can range over an unrestricted domain
Binary semaphore – integer value can range only between 0 and 1
Semaphore ***S*** – integer variable
Two standard operations modify ***S***: **wait()** and **signal()**
Originally called **P()** and **V()**
Syntax:

```
wait (S)
 {
   while (S <= 0)
     ; // busy wait
   S--;
}
signal (S) {
   S++;}
```

Monitors

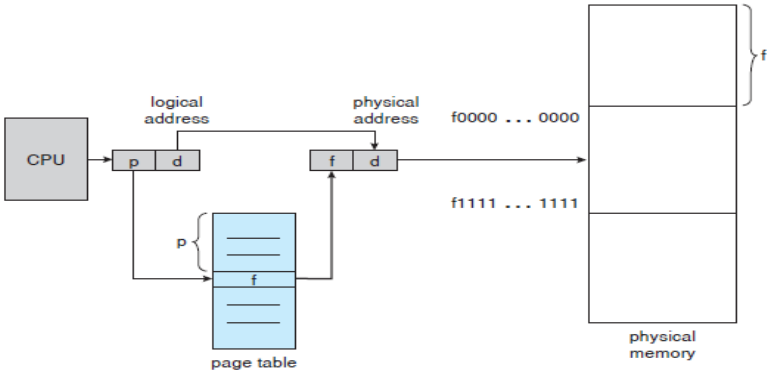
Syntax of a monitor

```
monitor monitor-name
{
        // shared variable declarations
        procedure P1 (…) { …. }
        procedure Pn (…) {……}
  Initialization code (…) { … }
        }
}
```

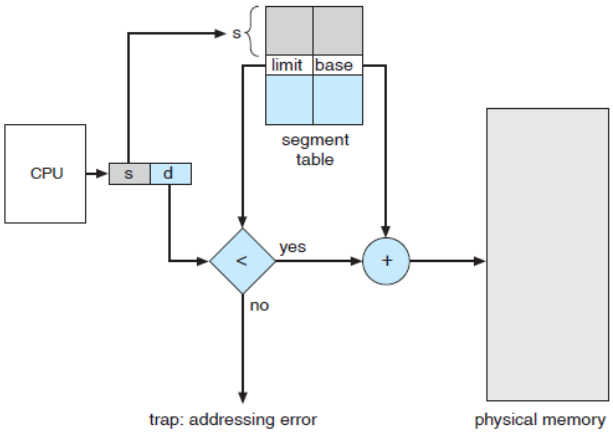
Deadlocks

Data Structures for the Banker's Algorithm
Let n = number of processes, and m = number of resources types.
Available: Vector of length m. If available [j] = k, there are k instances of resource type R_j available
Max: n x m matrix. If Max [i,j] = k, then process P_i may request at most k instances of resource type R_j
Allocation: n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j
Need: n x m matrix. If $Need$[i,j] = k, then P_i may need k more instances of R_j to complete its task
$Need$ [i,j] = Max[i,j] – $Allocation$ [i,j]

Memory Management



Paging hardware



SEGMENTATION HARDWARE

Virtual Memory

Performance of Demand Paging:
Let p be the probability of a page fault ($0 \le p \le 1$).
The **effective access time** is then effective access time = $(1 - p)$ × memory access time + p × page fault time.

Frame Allocation Algorithms

a. Equal allocation

b. Proportional allocation

Let the size of the virtual memory for process *pi* be *si* , and define $S = \sum si$ .Then, if the total number of available frames is *m*, we allocate *ai* frames to process *pi*, where *ai* is approximately

$ai = si/S \times m$.

Working-set model

The working-set size is computed as, *WSSi* , for each process in the system, $D = \sum WSSi$,where *D* is the total demand for frames.

Storage management

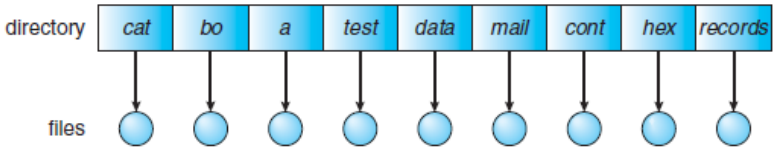
Common file types

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

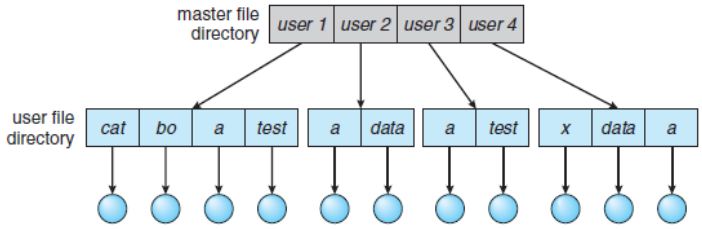
Simulation of sequential access on a direct-access file

sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;
write_next	write cp; cp = cp + 1;

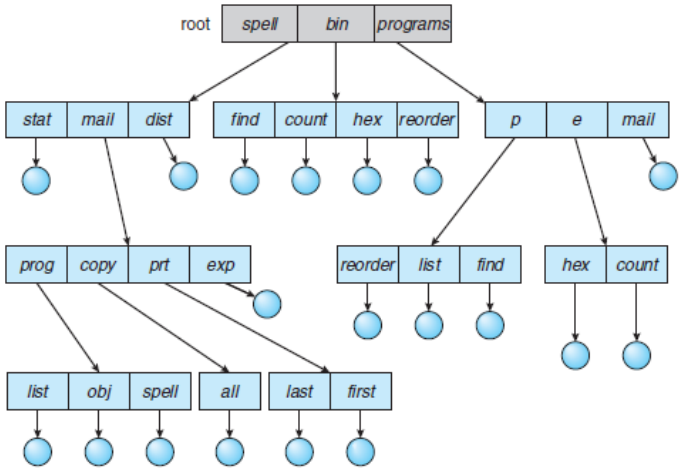
Single level directory structure



Two-level directory structure



Tree-structured directory structure



Real Time Operating System

Real time operating system helps the real time applications/tasks to meet their deadlines by using a task scheduler.

Real-time Task Scheduling: Real-time task scheduling essentially refers to determining the order in which the various tasks are to be taken up for execution by the operating system.

Types of real time tasks: Periodic, Sporadic and Aperiodic

Notation

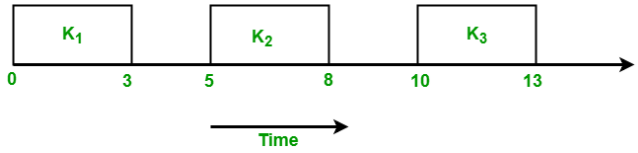
The 4-tuple $T_i = (\phi_i, p_i, e_i, D_i)$ refers to a periodic task T_i with phase ϕ_i, period p_i, execution time e_i, and relative deadline D_i

- Default phase of T_i is $\phi_i = 0$, default relative deadline is the period $D_i = p_i$
- Omit elements of the tuple that have default values

Examples:

i) $T_1 = (1, 10, 3, 6) \rightarrow \phi_1 = 1, p_1 = 10, e_1 = 3, \quad D_1 = 6$

ii) Consider the task T_i with period = 5 and execution time = 3. Phase is not given so, assume the release time of the first job as zero. So the job of this task is first released at t = 0 then it executes for 3s and then next job is released at t = 5 which executes for 3s and then next job is released at t = 10. So jobs are released at t = 5k where k = 0, 1, . . ., n



Real time task scheduling algorithms have been divided into: **Clock driven, Event driven, Hybrid**

Theorem 1: The major cycle of a set of tasks ST = $\{T_1, T_2, \dots, T_n\}$ is LCM ($\{p_1, p_2, \dots, p_n\}$) even when the tasks have arbitrary phasing, where $p_1, p_2, \dots, p_n$ are the periods of $T_1, T_2, \dots, T_n$

Theorem 2: The minimum separation of the task arrival from the corresponding frame start time (min(Δt)), considering all instances of a task T_i, is equal to gcd(F, p_i).

Rate-monotonic scheduling

Necessary condition:

$$\sum_{i=1}^{n} e_i/p_i = \sum_{i=1}^{n} U_i \leq 1$$

where e_i is the worst case execution time and p_i is the period of the task T_i, n is the number of tasks to be scheduled, and U_i is the CPU utilization due to the task T_i

Sufficient Condition: The sufficient condition as given by Liu and Layland for a set of n real-time periodic tasks that are schedulable under RMA, if

$$\sum_{i=1}^{n} U_i \leq n\left(2^{1/n} - 1\right)$$

where U_i is the utilization due to task T_i. If a set of tasks satisfies the sufficient condition, then it is guaranteed that the set of tasks would be RMA schedulable.

Earliest Deadline First

Priority-driven Approach - the earlier the deadline, the higher the priority

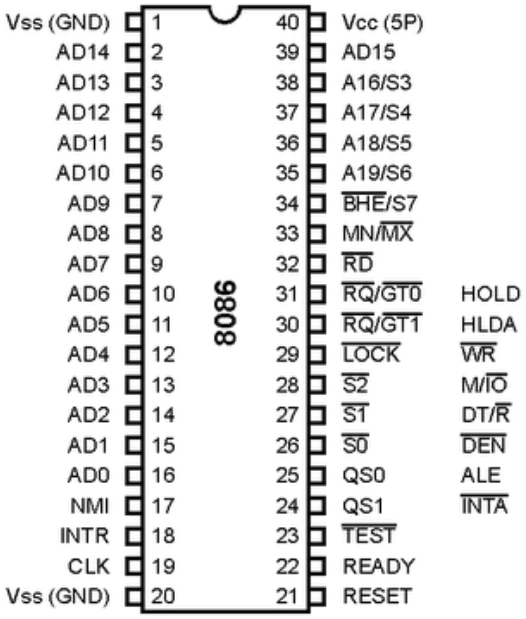
Theorem [EDF bound]: A set of n periodic tasks, each of whose relative deadline equals its period, can be feasibly scheduled by EDF if and only if

$$\sum_{i=1}^{n} e_i/p_i \leq 1$$

8086 MICROPROCESSOR SYSTEMS

8086 Microprocessor Pin Diagram

8086 is a 16-bit microprocessor available in 40 pin Dual In Package (DIP).



8086 INSTRUCTION SET

Data Transfer Instructions

MOV Destination, Source
The MOV instruction copies a word or byte of data from a specified source to a specified destination.

XCHG Destination, Source
This instruction exchanges the content of a register with the content of another register or with the content of memory location(s).

LEA Register, Source
This instruction determines the offset of the variable or memory location named as the source and puts this offset in the 16-bit register.

LDS Register, Memory address of the first word
This instruction loads new values into the specified register and into the DS register from four successive memory locations.

LES Register, Memory address of the first word
This instruction loads new values into the specified register and into the ES register from four successive memory locations.

Arithmetic Instructions

ADD Destination, Source
This instruction adds a number from some *source* to a number in some *destination* and puts the result in the specified destination.

ADC Destination, Source
This instruction adds a number from some *source* to a number in some *destination* along with carry flag and puts the result in the specified destination.

SUB Destination, Source
This instruction subtracts the number in some *source* from the number in some *destination* and puts the result in the destination.

SBB Destination, Source
This instruction subtracts the number in some *source* and the carry flag from the number in some *destination* and puts the result in the destination.

MUL Source
This instruction multiplies an *unsigned* byte in some *source* with an *unsigned* byte in AL register or an unsigned word in some *source* with an unsigned word in AX register. When a byte is multiplied by the content of AL, the product is put in AX. When a word is multiplied by the content of AX, the result is put in DX and AX registers.

IMUL Source
This instruction multiplies a *signed* byte from *source* with a *signed* byte in AL or a *signed* word from some *source* with a *signed* word in AX. When a byte from source is multiplied with content of AL, the signed product will be put in AX. When a word from source is multiplied by AX, the signed product is put in DX and AX.

DIV Source
This instruction is used to divide an *unsigned* word by a byte or to divide an *unsigned* double word by a word. When a word is divided by a byte, the word must be in the AX register. After the division, AL will contain the 8-bit quotient, and AH will contain the 8-bit remainder. When a double word is divided by a word, the most significant word of the double word must be in DX, and the least significant word of the double word must be in AX. After the division, AX will contain the 16-bit quotient, and DX will contain the 16-bit remainder.

IDIV Source
This instruction is used to divide a *signed* word by a *signed* byte, or to divide a *signed* double word by a *signed* word. When dividing a signed word by a signed byte, the word must be in the AX register. The divisor can be in an 8-bit register or a memory location. After the division, AL will contain the signed quotient, and AH will contain the signed remainder. When a double word is divided by a word, the most significant word of the double word must be in DX, and the least significant word of the double word must be in AX. After the division, AX will contain the 16-bit signed quotient, and DX will contain the 16-bit signed remainder.

INC Destination
The INC instruction adds 1 to the destination word or bye.

DEC Destination
This instruction subtracts 1 from the destination word or byte.

DAA
This instruction is used to adjust the sum of two packet BCD bytes available in AL register in to packed BCD.

DAS
This instruction is used to adjust the difference of two packet BCD bytes available in AL register in to packed BCD.

CBW
This instruction converts a byte in AL into word in AX by copying the sign bit of AL to all the bits in AH.

CWD
This instruction converts a word in AX into double word in DX: AX by copying the sign bit of AX to all the bits in DX.

AAA
This instruction allows us to add the ASCII codes for two decimal digits without masking the higher order nibble 3. The unpacked BCD sum is available in AL and the carry flag is set in case of any adjustment.

AAS
This instruction allows us to subtract the ASCII codes for two decimal digits without masking the higher order nibble 3. The unpacked BCD difference is available in AL and the carry flag is set in case of any adjustment.

AAM
This instruction is used to adjust the product in AX in to unpacked BCD in AX.

AAD
AAD converts two unpacked BCD digits in AH and AL in to the equivalent binary number in AX.

Logical Instructions

AND Destination, Source
This instruction performs the bitwise logical AND operation of source and destination operands and stores the result in the destination operand.

OR Destination, Source
This instruction performs the bitwise logical OR operation of source and destination operands and stores the result in the destination operand.

XOR Destination, Source
This instruction performs the bitwise logical XOR operation of source and destination operands and stores the result in the destination operand.

NOT Destination
This instruction performs the 1's complement of a byte or word in the specified destination.

NEG Destination
This instruction performs the 2's complement of a byte or word in the specified destination.

CMP Destination, Source
This instruction compares the destination with the source operand by subtracting the source from the destination. Result is nowhere stored. Flags are affected accordingly.

TEST Destination, Source
This instruction performs the bitwise logical AND operation of the source operand with the destination operand. Result is nowhere stored. Flags are affected accordingly.

Rotate and Shift Instructions

ROL Destination, Count
This instruction rotates destination operand **Count** number of bit positions to the left.

ROR Destination, Count
This instruction rotates destination operand **Count** number of bit positions to the right.

RCL Destination, Count
This instruction rotates destination operand through the carry flag, **Count** number of bit positions to the left.

RCR Destination, Count
This instruction rotates destination operand through the carry flag, **Count** number of bit positions to the right.

SAL Destination, Count
SHL Destination, Count
This instruction logically shifts the destination operand, **Count** number of bit positions to the left.

SHR Destination, Count
This instruction logically shifts the destination operand, **Count** number of bit positions to the right.

SAR Destination, Count
This instruction arithmetically shifts the destination operand, **Count** number of bit positions to the right.

Program Execution Transfer Instructions

JMP label
This instruction will fetch and execute the instruction from the address of the label rather than from the next address after the JMP instruction.

JC label
This instruction will fetch and execute the instruction from the address of the label if Carry Flag is equal to 1.

JNC label
This instruction will fetch and execute the instruction from the address of the label if Carry Flag is equal to 0.

JZ label
JE label
This instruction will fetch and execute the instruction from the address of the label if Zero Flag is equal to 1.

JNZ label
JNE label
This instruction will fetch and execute the instruction from the address of the label if Zero Flag is equal to 0.

JO label
This instruction will fetch and execute the instruction from the address of the label if Overflow Flag is equal to 1.

JNO label
This instruction will fetch and execute the instruction from the address of the label if Overflow Flag is equal to 0.

JP label
JPE label
This instruction will fetch and execute the instruction from the address of the label if Parity Flag is equal to 1.

JNP label
JPO label
This instruction will fetch and execute the instruction from the address of the label if Parity Flag is equal to 0.

JS label
This instruction will fetch and execute the instruction from the address of the label if Sign Flag is equal to 1.

JNS label
This instruction will fetch and execute the instruction from the address of the label if Sign Flag is equal to 0.

JA label
JNBE label
This instruction will fetch and execute the instruction from the address of the label if Carry Flag is equal to 0 and Zero Flag is equal to 0.

JAE label
JNB label
This instruction will fetch and execute the instruction from the address of the label if Carry Flag is equal to 0 or Zero Flag is equal to 1.

JNA label
JBE label
This instruction will fetch and execute the instruction from the address of the label if Carry Flag is equal to 1 or Zero Flag is equal to 1.

JNAE label
JB label
This instruction will fetch and execute the instruction from the address of the label if Carry Flag is equal to 1 and Zero Flag is equal to 0.

JG label
JNLE label
This instruction will fetch and execute the instruction from the address of the label if Sign Flag is equal to Overflow Flag, and Zero Flag is equal to 0.

JGE label
JNL label
This instruction will fetch and execute the instruction from the address of the label if Sign Flag is equal to Overflow Flag, or Zero Flag is equal to 1.

JNG label
JLE label
This instruction will fetch and execute the instruction from the address of the label if Sign Flag is not equal to Overflow Flag, or Zero Flag is equal to 1.

JNGE label
JL label
This instruction will fetch and execute the instruction from the address of the label if Sign Flag is not equal to Overflow Flag, and Zero Flag is equal to 0.

JCXZ label
This instruction will fetch and execute the instruction from the address of the label, if the CX register contains all 0's.

LOOP label
This instruction will automatically decrement CX by 1. If CX is not 0, execution will jump to a destination specified by a label in the instruction. If CX = 0 after the auto decrement, execution will simply go on to the next instruction after LOOP.

LOOPE label
LOOPZ label
This instruction will automatically decrement CX by 1. If CX is not 0 and Zero Flag is equal to 1, execution will jump to a destination specified by a label in the instruction.

LOOPNE label
LOOPNZ label
This instruction will automatically decrement CX by 1. If CX is not 0 and Zero Flag is equal to 0, execution will jump to a destination specified by a label in the instruction.

CALL procedure_name
The CALL instruction is used to transfer execution to a subprogram or a procedure.

RET
The RET instruction will return execution from a procedure to the next instruction after the CALL instruction which was used to call the procedure.

String Manipulation Instructions

MOVSB

This instruction copies a byte from location in the data segment pointed by SI register to a location in the extra segment pointed by DI register. After the byte is moved, SI and DI are automatically incremented or decremented by 1 to point to the next source element and the next destination element based on the settings of DF.

MOVSW

This instruction copies a word from location in the data segment pointed by SI register to a location in the extra segment pointed by DI register. After the word is moved, SI and DI are automatically incremented or decremented by 2 to point to the next source element and the next destination element based on the settings of DF.

LODSB

This instruction copies a byte from the data segment pointed to by SI to AL. After the byte is moved, SI is automatically incremented or decremented by 1 based on the settings of DF.

LODSW

This instruction copies a word from the data segment pointed to by SI to AX. After the word is moved, SI is automatically incremented or decremented by 2 based on the settings of DF.

STOSB

This instruction copies a byte from the AL to a location in the extra segment pointed to by DI. After the byte is moved, DI is automatically incremented or decremented by 1 based on the settings of DF.

STOSW

This instruction copies a word from the AX to a location in the extra segment pointed to by DI. After the word is moved, DI is automatically incremented or decremented by 2 based on the settings of DF.

CMPSB

This instruction compares a byte in the data segment pointed by SI register with a byte in the extra segment pointed by DI register. After the comparison, SI and DI are automatically incremented or decremented by 1 to point to the next source element and the next destination element based on the settings of DF.

COMPSW

This instruction compares a word in the data segment pointed by SI register with a word in the extra segment pointed by DI register. After the comparison, SI and DI are automatically incremented or decremented by 2 to point to the next source element and the next destination element based on the settings of DF.

SCASB

This instruction compares AL register with a byte in the extra segment pointed by DI register. After the comparison, DI is automatically incremented or decremented by 1 based on the settings of DF.

SCASW

This instruction compares AX register with a word in the extra segment pointed by DI register. After the comparison, DI is automatically incremented or decremented by 2 based on the settings of DF.

REP

REP is a prefix, which is written before the string instruction. It will cause the CX register to be decremented and the string instruction to be repeated until CX = 0.

REPE/REPZ
REPE and REPZ are two mnemonics for the same prefix. They will cause the string instruction to be repeated as long as the compared bytes or words are equal (ZF = 1) and CX is not yet counted down to zero.

REPNE/REPNZ
REPNE and REPNZ are two mnemonics for the same prefix. They will cause the string instruction to be repeated as long as the compared bytes or words are not equal (ZF = 0) and CX is not yet counted down to zero.

Flag Manipulation Instructions

STC
This instruction sets the carry flag to 1.

CLC
This instruction resets the carry flag to 0.

CMC
This instruction complements the carry flag.

STD
This instruction sets the direction flag to 1.

CLD
This instruction resets the direction flag to 0.

STI
This instruction sets the Interrupt flag to 1.

CLI
This instruction resets the Interrupt flag to 0.

LAHF
The LAHF instruction copies the lower byte of the 8086 flag register to AH register.

SAHF
The SAHF instruction replaces the lower byte of the 8086 flag register with a byte from the AH register.

Stack Related Instructions

PUSH Source
The PUSH instruction decrements the stack pointer by 2 and copies a word from a specified source to the location in the stack segment to which the stack pointer points.

POP Destination
The POP instruction copies a word from the stack location pointed to by the stack pointer to a destination specified in the instruction and increments the stack pointer by 2.

PUSHF
The PUSH instruction decrements the stack pointer by 2 and copies flag register to the location in the stack segment to which the stack pointer points.

POPF
The POP instruction copies a word from the stack location pointed to by the stack pointer to the flag register and increments the stack pointer by 2.

Input-Output Instructions
IN AL, Port_8
IN AX, Port_8
IN AL, DX
IN AX, DX
The IN instruction copies data from a port to the AL or AX register. Port_8 is the 8 bit fixed port address of the port. For variable port addressing, DX holds the 16-bit address of the port.

OUT Port_8, AL
OUT Port_8, AX
OUT DX, AL
OUT DX, AX
The OUT instruction copies a byte from AL or a word from AX to the specified port. Port_8 is the 8 bit fixed port address of the port. For variable port addressing, DX holds the 16-bit address of the port.

Miscellaneous Instructions

HLT
The HLT instruction causes the 8086 to stop fetching and executing instructions.

NOP
This instruction simply uses up three clock cycles and increments the instruction pointer to point to the next instruction.

ESC
This instruction is used to pass instructions to a coprocessor, such as the 8087 Math coprocessor, which shares the address and data bus with 8086.

INT type
This instruction causes the execution of interrupt handler specified by the type.

INTO
If the overflow flag is set, this instruction causes the 8086 to do an indirect far call to a procedure you write to handle the overflow condition.

IRET
The IRET instruction is used at the end of the interrupt service procedure to return execution to the interrupted program.

LOCK
The LOCK prefix allows a microprocessor to make sure that another processor does not take control of the system bus while it is in the middle of a critical instruction, which uses the system bus.

WAIT
When this instruction is executed, the 8086 enters an idle condition in which it is doing no processing.

XLAT
XLATB
The instruction copies the byte from the address pointed to by (BX + AL) in the data segment into AL.

8086 DOS INTERRUPTS – TYPE 21H

Function 01- Character input with echo

Action: Reads a character from the standard input device and echoes it to the standard output device. If no character is ready it waits until one is available.
On entry: AH = 01h
Returns: AL = 8-bit data input

Function 02 - Character output
Action: Outputs a character to the standard output device.
On entry: AH = 02h
DL = 8 bit data (usually ASCII character)

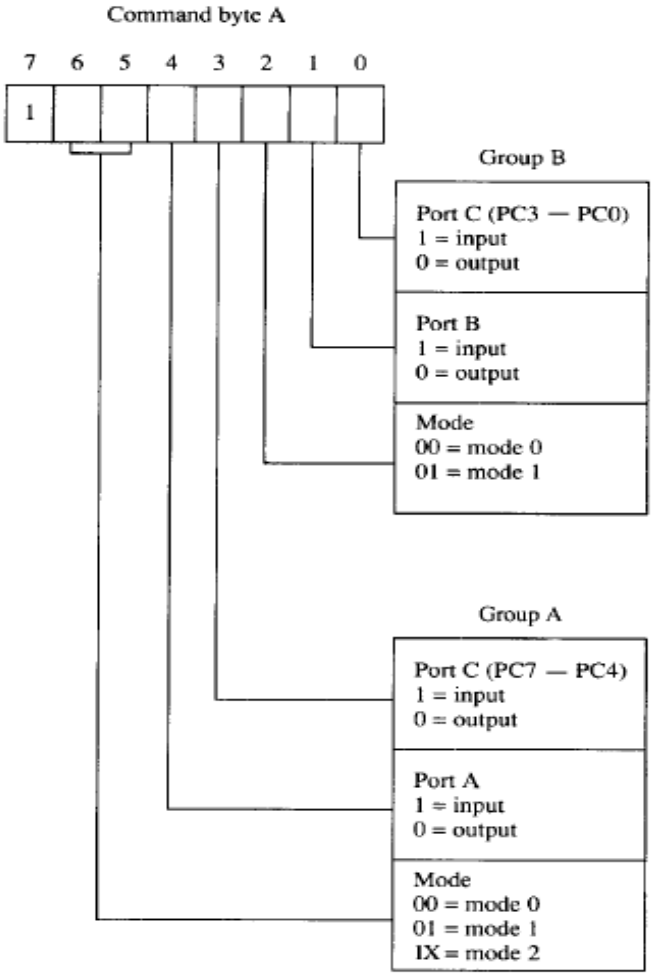
Function 06- Direct console I/O
Action: Reads a character from the standard input device or returns zero if no character available. Also can write a character to the current standard output device.
On entry: AH = 06h
DL = function requested: 0h to FEh = output
(DL = character to be output)
FFh = Input request
Returns: If output - nothing
If input - data ready: zero flag clear, AL = 8-bit data
If data not ready: zero flag set

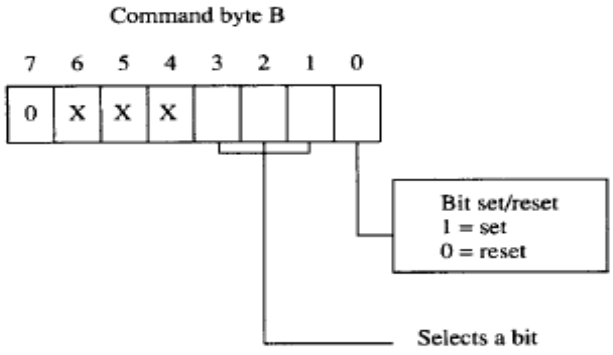
Function 08- Character input with no echo
Action: Reads a character from the standard input device without copying it to the display. If no character is ready it waits until one is available.
On entry: AH = 08h
Returns: AL = 8 bit data input

Function 09- Output character string
Action: Writes a string terminated with $ character to the display.
On entry: AH = 09h
DS:DX = segment: offset of string

Function 0Ah - Buffered input
Action: Reads a string from the current input device up to and including an ASCII carriage return(0Dh), placing the received data in a user-defined buffer
On entry: AH = 0Ah
DS:DX = segment:offset of string buffer . The first byte of the buffer specifies the maximum number of characters it can hold. The second byte of the buffer is set by DOS to the number of characters actually read, excluding the terminating RETURN. The input string is stored from the third byte onwards.

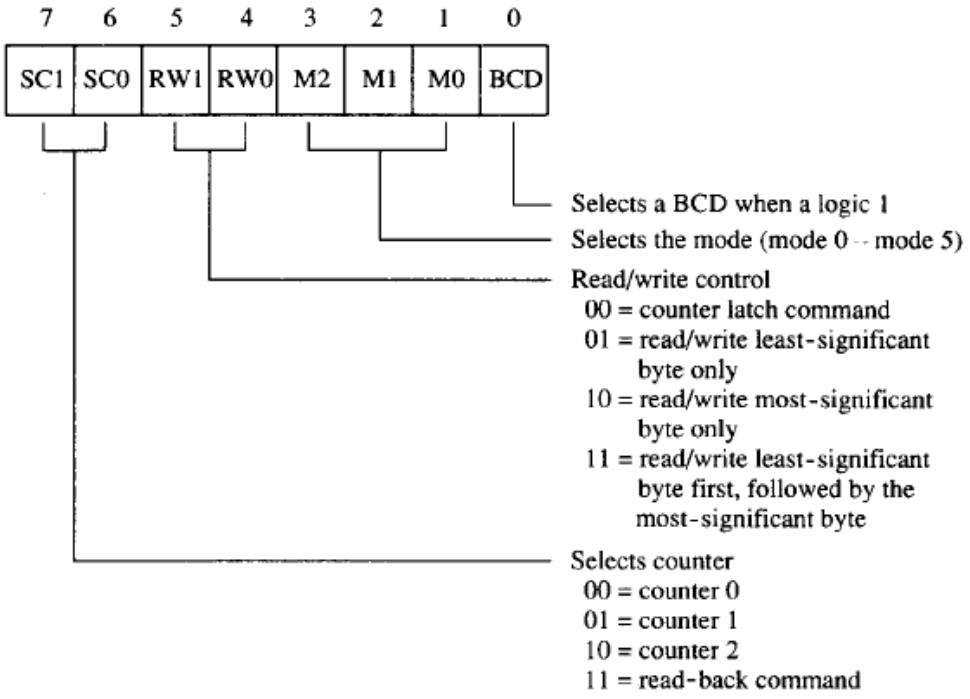
8255 Programmable Peripheral Interface



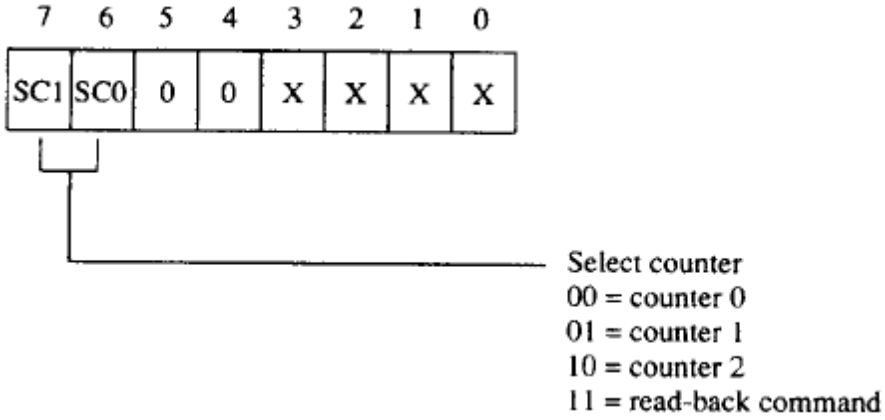


8254 Programmable Interval Timer

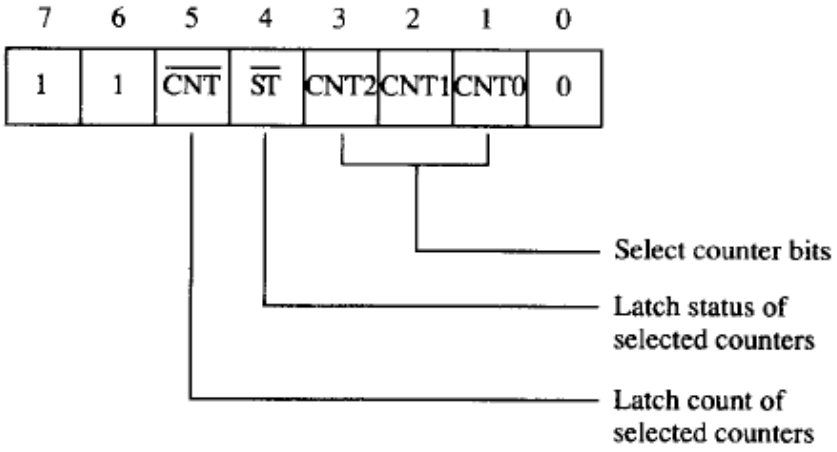
Control Word Register



Counter Latch Command

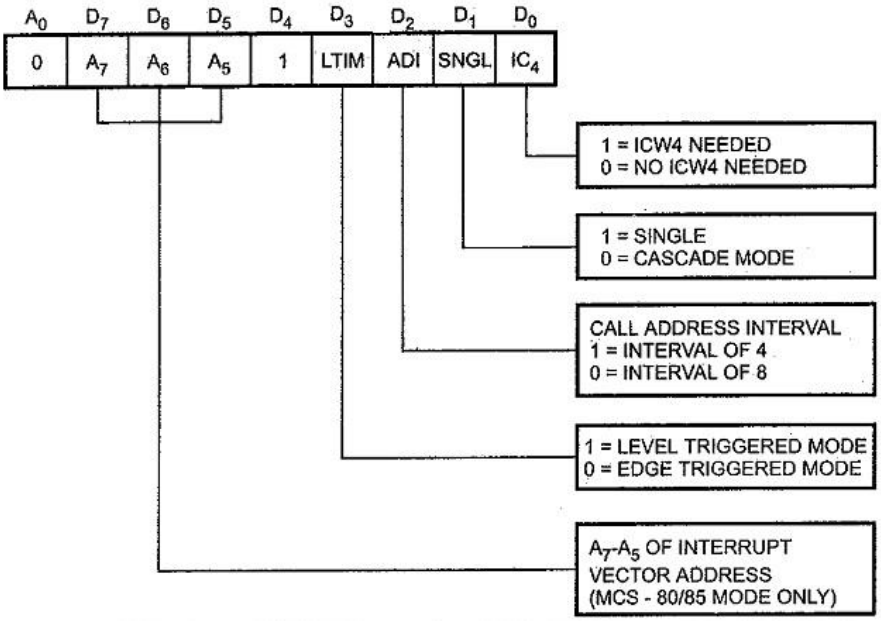


Read-back Command

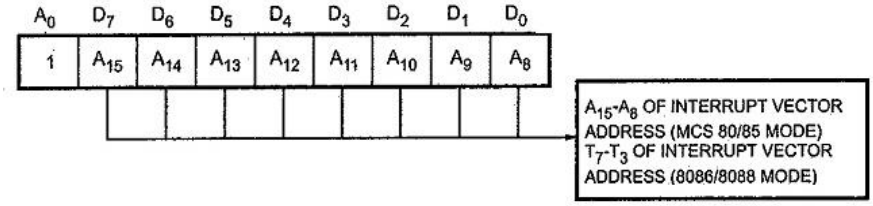


8259 Programmable Interrupt Controller

ICW1



ICW2



ICW3

ICW3 (MASTER DEVICE)

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0

1 = IR INPUT HAS A SLAVE
0 = IR INPUT DOES NOT HAVE A SLAVE

ICW3 (SLAVE DEVICE)

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	0	0	0	0	ID_2	ID_1	ID_0

SLAVE ID

0	1	2	3	4	5	6	7
0	1	0	1	0	1	0	1
0	0	1	1	0	0	1	1
0	0	0	0	1	1	1	1

ICW4

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	0	0	SFNM	BUF	M/S	AEOI	µPM

1 = 8086/8088 MODE
0 = MCS - 80/85 MODE

1 = AUTO EOI
0 = NORMAL EOI

0	X
1	0
1	1

NON BUFFERED MODE
BUFFERED MODE/SLAVE
BUFFERED MODE/MASTER

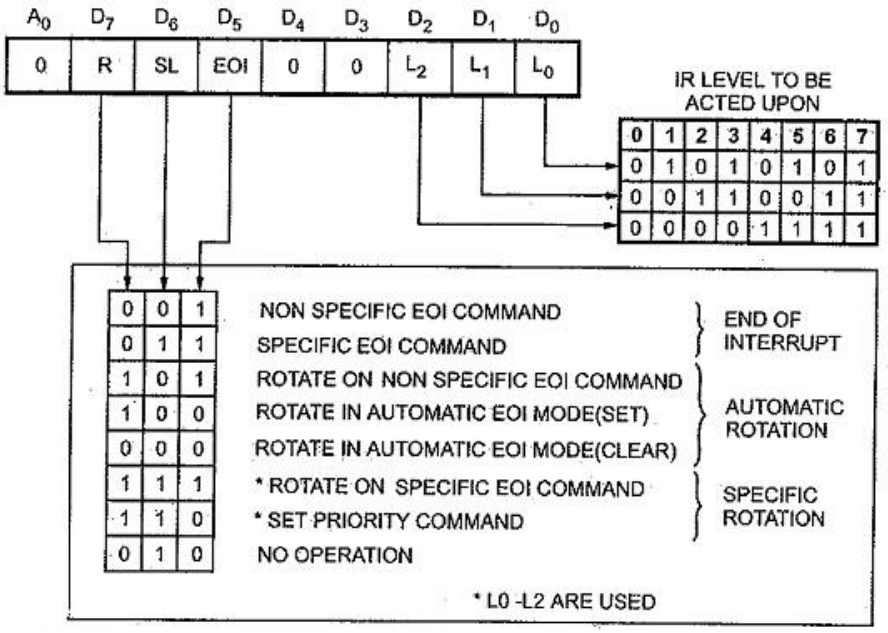
1 = SPECIAL FULLY NESTED MODE
0 = NOT SPECIAL FULLY NESTED MODE

OCW1

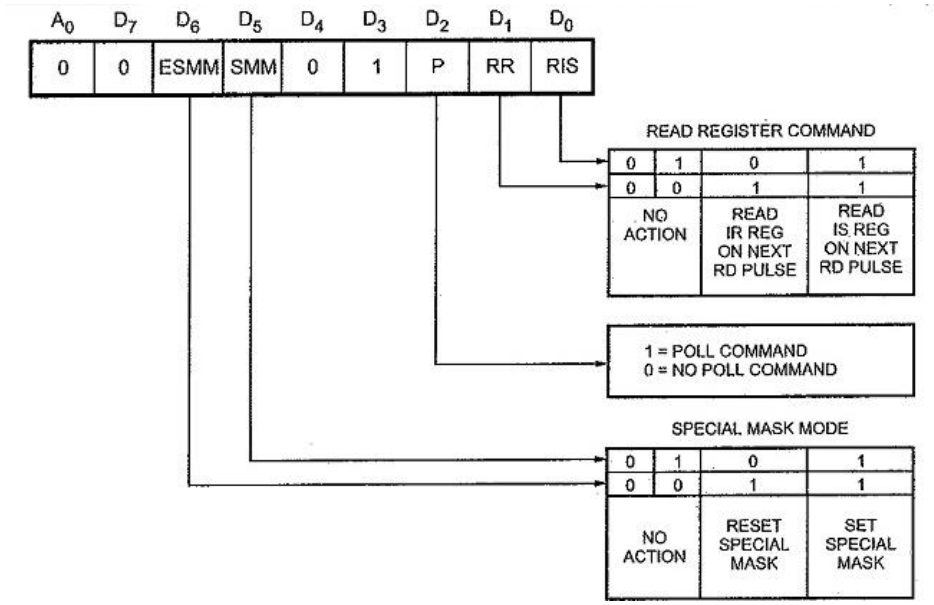
A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	M_7	M_6	M_5	M_4	M_3	M_2	M_1	M_0

INTERRUPT MASK
1 = MASK SET
0 = MASK RESET

OCW2


A0 D7 D6 D5 D4 D3 D2 D1 D0
0 R SL EOI 0 0 L2 L1 L0
IR LEVEL TO BE ACTED UPON
0 1 2 3 4 5 6 7
0 1 0 1 0 1 0 1
0 0 1 1 0 0 1 1
0 0 0 0 1 1 1 1
0 0 1 NON SPECIFIC EOI COMMAND
0 1 1 SPECIFIC EOI COMMAND
1 0 1 ROTATE ON NON SPECIFIC EOI COMMAND
1 0 0 ROTATE IN AUTOMATIC EOI MODE(SET)
0 0 0 ROTATE IN AUTOMATIC EOI MODE(CLEAR)
1 1 1 * ROTATE ON SPECIFIC EOI COMMAND
1 1 0 * SET PRIORITY COMMAND
0 1 0 NO OPERATION
END OF INTERRUPT
AUTOMATIC ROTATION
SPECIFIC ROTATION
* L0 -L2 ARE USED

OCW3


A0 D7 D6 D5 D4 D3 D2 D1 D0
0 0 ESMM SMM 0 1 P RR RIS
READ REGISTER COMMAND
0 1 0 1
0 0 1 1
NO ACTION
READ IR REG ON NEXT RD PULSE
READ IS REG ON NEXT RD PULSE
1 = POLL COMMAND
0 = NO POLL COMMAND
SPECIAL MASK MODE
0 1 0 1
0 0 1 1
NO ACTION
RESET SPECIAL MASK
SET SPECIAL MASK