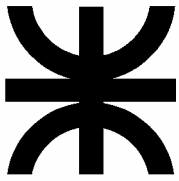


UNIVERSIDAD TECNOLOGICA NACIONAL
FACULTAD REGIONAL BS. AS.

Sistemas de Procesamiento de Datos

Autor: Ing. Fidel SANTOS

Contenidos

1 SISTEMAS DE NUMERACIÓN

1.1 Introducción

En este capítulo expondremos brevemente (a modo de repaso) conceptos básicos sobre los sistemas de numeración.

No por sencillo el tema deja de ser importante pues nos permite comenzar a acostumbrarnos a los sistemas de numeración utilizados en computación, especialmente el binario y el hexadecimal, tarea no trivial si tenemos en cuenta el "lastre" que significan años y años de práctica con el sistema decimal exclusivamente.

Podemos entender un sistema de numeración como un conjunto de símbolos y un conjunto de reglas de combinación de dichos símbolos que permiten representar los números enteros y/o fraccionarios.

Dentro de los sistemas de numeración posibles un conjunto importante, destacado, es el constituido por los sistemas de numeración posicionales

1.2 Sistemas posicionales

En estos sistemas la representación de un número se realiza mediante los símbolos y su posición relativa dentro de la expresión.

Como ejemplo de un sistema posicional podemos citar al Romano, en el cual es claro que la posición relativa de los símbolos influye en la representación. Ej.: VI corresponde al 6 y IV al 4.

Dentro de los sistemas posicionales están incluidos los que serán objeto de nuestro estudio: los sistemas con base.

1.3 Sistemas con base

En los sistemas con base N (un número cualquiera), se representa mediante un polinomio de la forma:

$$N = a_n b^n + ... + a_0 b^0 + a_{-1} b^{-1} + ...$$

donde ai es un símbolo del sistema, al que llamamos dígito, y b es la base.

La base es igual a la cantidad de símbolos del sistema. Notando que los dígitos son la representación en el sistema de los números enteros menores que la base, tenemos que se cumple la condición $b > a_i \geq 0$

La base b la representamos siempre en el sistema decimal (por supuesto si la representáramos en el sistema del cual es base su representación sería 10).

Habitualmente la representación omite las potencias de la base y coloca un punto (o coma) para separar la parte de potencias positivas de la parte con potencial negativas, quedando:

$$N \Rightarrow a_n a_{n-1} ..... a_0 . a_{-1} . a_{-2} . a_{-p}$$

- *Sistema decimal*: ***El sistema de numeración utilizado en la vida cotidiana es el decimal, cuya base es diez, utilizando los conocidos diez símbolos 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9.***
- *Sistema binario*: ***Es el sistema de base 2 en el cual los dos símbolos utilizados son el 0 y el 1, los que reciben el nombre de bit (binary digit).***
- *Sistema Octal*: ***Es el sistema de base 8 en el cual se usan los símbolos 0, 1, 2, 3, 4, 5, 6, 7.***
- *Sistema Hexadecimal*: ***Es el sistema de base 16 en el cual se usan los símbolos 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.***

La base del sistema en el que está representado un número se suele indicar con un subíndice al final del número y en los casos particulares de base 2 (binario), base 8 (octal), base 16 (hexadecimal) con las letras b, o (ó q) y h respectivamente. Si no se indica nada se asume base 10.

Ejemplos:

$$1101_{\ 2} = 1101_{\ b} = 1.2^3 + 1.2^2 + 0.2^1 + 1.2^0 = 13\,(decimal)$$

$$1101,11_{\ 2} = 1101,11_{\ b} = 1.2^3 + 1.2^2 + 0.2^1 + 1.2^0 + 1.2^{-1} + 1.2^{-2} = 11,75\,(decimal)$$

$$A2F_{16} = A2F_h = 10.16^2 + 2.16^1 + 15.16^0 = 2560 + 32 + 15 = 2607\ (decimal)$$

Estudiaremos a continuación los paralelismos para que, dada la representación de un número en una cierta base podamos hallar la correspondiente representación en otra base dada.

1.4 Conversión de base de enteros.

Nuestro deseo es dado un número N entero en una base B representado por

$$N = A_n B^n + ... + A_0 B^0$$

se desea hallar su expresión en una base b
En definitiva lo que buscamos es hallar los valores de a_m , a_{m-1} , . . . , a_0

- *Caso* **A:**

Conversión de una base B a una base b usando la aritmética de la base b (muy útil para pasar de cualquier base a la base 10).
La conversión se hace a través del polinomio característico, expresando los símbolos $A_n \ldots A_0$ y la base B en la base b y evaluando el polinomio, realizando las operaciones en la base b.
Es el caso de los ejemplos ya vistos:

$$A2F_h = 10.16^2 + 2.16^1 + 15.16^0 = 2607$$

- *Caso* **B:**

Conversión de una base B a una base b usando la aritmética de la base B (muy útil para pasar de base 10 a cualquier base)
Previamente notemos que:

$$N = b.N_1 + a_0$$
$$N_1 = b.N_2 + a_1$$
.
.
.
$$N_{n-1} = b.N_n + a_n$$

Por lo que los valores $a_0 \ldots a_n$ son los restos de las divisiones de N entre b realizadas en la aritmética de la base B.
Ejemplo: Convertir 653 a binario.

653	1
326	0
163	1
81	1
40	0
20	0
10	0
5	1
2	0
1	1

653=1010001101

Ejemplo: Convertir 653 a base 5.

653	3
130	0
26	1
5	0
1	1

$653 = 10103_5$
Los ejemplos vistos son siempre de decimal a otra base; si quisiéramos pasar desde una base b_1 (b_1 <> 10) a la base b_2 existe la posibilidad de hacer las operaciones con la base b_1 o cambiar primero a base 10 y luego de esta a la base b_2.

1.5 Conversión de números con parte fraccionaria.

Sea un número $N = N_e + N_f = a_n \cdot b_n + ... + a_1 \cdot b + a_0 + a_{-1} \cdot b_{-1} + ...$ donde N_e y N_f son la parte entera y la parte fraccionaria respectivamente. La parte fraccionaria sigue siempre a la parte entera en cualquier base. Por lo tanto N_e puede convertirse igual que antes y N_f se convierte por separado
Estudiaremos entonces como convertir partes fraccionarias. Sean

$N_f = A_{-1} \cdot B^{-1} + A_{-2} \cdot B^{-2} + ...$ *en base B*

$N_f = a_{-1} \cdot b^{-1} + a_{-2} \cdot b^{-2} + ...$ *en base b*

- *Caso **A***:

Conversión de base B a una base b usando la aritmética de la base b (muy útil para pasar de cualquier base a base 10)

Sea $N_f = A_{-1}.B^{-1} + A_{-2}.B^{-2} + ... + A_{-m}.B^{-m}$

lo que hago es desarrollar el polinomio equivalente.

Sea $P(x) = A_{-1}.x^1 + A_{-2}.x^2 + ... + A_{-m}.x^m$

Si se calcula el valor numérico de $P(x)$ para $x = B^{-1}$ usando aritmética b obtendremos el valor buscado.

Ejemplo : pasar $0{,}213_8$ a base decimal

$N = 2 . 8^{-1} + 1 . 8^{-2} + 3 . 8^{-3} \Rightarrow P(x) = 3 . x^3 + x^2 + 2 . x$

El valor numérico para $x = 1/8$ será:

$P(x) = 3 \cdot (0.125)^3 + (0.125)^2 + 2 \cdot (0.125) = 0.27148...$

- *Caso B*:

Conversión de una base B a una base b operando con la aritmética de la B (lo que la hace muy útil para pasar de base 10 a cualquier base)

Para determinar los coeficientes a_{-1}, a_{-2}, etc. para la base b se observa que cada uno de tales coeficientes es, en si mismo un entero.
Primero se multiplica por b (con aritmética B):

$b \cdot N_f = a_{-1} + a_{-2} \cdot b_{-1} + ... + a_{-3} \cdot b_{-2}$

En donde, la parte entera de b.Nf es a-1. A continuación se resta a-1 y se multiplica de nuevo por b:

$b\ (b \cdot N_f - a_{-1}) = a_{-2} + a_{-3} \cdot b_{-1} + ...$ determinando así a_{-2}. Se sigue este proceso hasta que se obtengan tantos coeficientes como se deseen. En el siguiente procedimiento puede ocurrir que el proceso no termine.

Ejemplo: Convertir 653.61 a base 2

$2 \cdot (0.61) = 1.22 \Rightarrow a_{-1} = 1$

$2 \cdot (0.22) = 0.44 \Rightarrow a_{-2} = 0$

$2 \cdot (0.44) = 0.88 \Rightarrow a_{-3} = 0$

$2 \cdot (0.88) = 1.76 \Rightarrow a_{-4} = 1$

$2 \cdot (0.76) = 1.52 \Rightarrow a_{-5} = 1$

$653 = 1010001101b \Rightarrow 653.61 = 1010001101.10011...b$

1.6 Caso Particular bases 8 y 16.

La base 8 (octal) y la base 16 (hexadecimal) tienen una íntima relación con la base 2. Puesto que $8 = 2^3$ cada dígito octal corresponde a 3 dígitos binarios. El procedimiento entonces para convertir un número binario en número octal es dividir en grupos de 3 bits a partir del punto binario y asignando el dígito octal correspondiente a cada grupo.

Ejemplo: convertir 11001010011.111110011_2 a base 8

11	001	010	011	.	111	110	011
3	1	2	3	.	7	6	3

$= 3123.7630_8$

La conversión de base 8 a base 2 se hace a la inversa, convirtiendo en binario cada dígito octal, así:

732_8 *es*

$7_8 = 111_b$

$3_8 = 011_b \Rightarrow 732_8 = 111011010_b$

$2_8 = 010_b$

El equivalente hexadecimal de un número binario se obtiene simplemente, dividiendo al primero en grupos de 4 bits.
Ejemplo :

0011	1010	.	0110	1000	1011	1101
3	A	.	6	8	B	D

=3A.68BDh

Análogamente se realiza para pasar de hexadecimal a binario. La tabla 1 presenta la combinación binaria equivalente a cada uno de los símbolos del sistema hexadecimal.

0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Tabla 1 - Conversión binario hexadecimal.

2 REPRESENTACION INTERNA DE DATOS

2.1 Introducción.

En este capítulo estudiaremos la forma en que se representan los distintos tipos de objetos con que trabajamos en computación. En los lenguajes de alto nivel manejamos distintos tipos de datos: caracteres, strings, números enteros, números reales. Para trabajar con ellos, en general sólo nos interesa saber que son, como se opera con cada uno y esto se estudia en los cursos de programación. En esta materia nos ocuparemos de otro aspecto: como están implementados, a nivel interno, los distintos tipos de datos.
La unidad elemental de información que se usa en computación es un objeto que toma solo 2 valores posibles (0, 1): el BIT (dígito binario)
Los distintos tipos de datos se construyen en base a estructuras de bits, los cuales serán en general arrays de n elementos que reciben el nombre de palabra de largo n. En el caso particular de n = 8 el array se denomina byte.
Estudiaremos entonces, la representación interna de los datos como la expresión de los distintos tipos en función de estructuras de bits.

2.2 Códigos.

Por lo anterior resulta que los distintos tipos de datos se representan a través de códigos binarios. Es decir existe un proceso de codificación de los objetos de información en función de otros (las estructuras de bits) con los que se trabajará en realidad.
Por esta razón es interesante estudiar, aunque sea brevemente, un problema fundamental en la manipulación de códigos binarios: la detección y corrección de errores.
No nos referiremos aquí a errores en la codificación de los objetos sino a los que aparecen cuando se manipula con ellos. Usualmente los objetos de información se almacenan y/o se transmiten. Estas dos operaciones comunes se realizan en definitiva utilizando medios físicos (memorias, discos, canales de comunicación, etc.) los cuales no están libres de errores, por lo tanto es de particular interés el estudio de la posibilidad de detectar o corregir errores en códigos binarios.
De esta manera nos aseguraríamos que un dato recuperado de una unidad de almacenamiento es correcto (coincide con el almacenado) o que un dato recibido por un canal de comunicaciones lo es (coincide con el enviado por el emisor).

2.3 Detección y corrección de errores.

Todos los sistemas de codificación que permiten detección y corrección de errores se basan en una misma idea: redundancia.
El fundamento es sencillo: para poder distinguir si un valor es correcto o no, o sea representa un objeto o no, debo agregar información adicional al código. Entonces en todo sistema de codificación con capacidad de detectar errores el número de objetos representados es siempre menor que el número de valores posibles del código binario utilizado.
Esta afirmación quedaría más clara cuando veamos las distintas estrategias utilizadas, de todas formas un ejemplo puede ser de utilidad: supongamos que tenemos 16 objetos a representar; en

principio, con un código binario de 4 bits alcanzaría (2_4 = 16), pero con 4 bits no estaríamos en condiciones de detectar errores puesto que todos los valores posibles del código binario utilizado corresponderían a objetos válidos. De esta manera si por ejemplo hubiera un error en un bit del código que representa a un objeto A se transformaría en el código que representa a un objeto B por lo que no habría posibilidad de detectarlo.
Para disponer, en este caso de la posibilidad de distinguir un código binario correcto de uno incorrecto, deberíamos utilizar un código binario de, por ejemplo, 6 bits. Con 6 bits tendríamos 64 valores posibles de los cuales sólo 16 representarían objetos reales. Si elegimos convenientemente los códigos asociados a cada objeto estamos en condiciones de detectar errores producidos por el cambio de un bit en el código que representa un objeto.

2.4 Paridad.

Uno de los mecanismos para generar sistemas de codificación con capacidad de detectar errores es el de la paridad, o codificación con bit de paridad.
La idea es agregar a cada código binario de n bits que representan objetos válidos, un nuevo bit calculado en función de los restantes. La forma en que se calcula el bit de redundancia (de paridad) es tal que la cantidad de unos en el código completo (original + bit de paridad) será par (en cuyo caso hablamos de paridad par) o impar (en cuyo caso se trata de paridad impar).
Cuando se genera el código (se almacena o se transmite) se calcula este bit mediante uno de los dos criterios expuestos a partir de los n bits originales. Cuando se recupera el código (se lee o se recibe) se recalcula el bit y se chequea con el almacenado o transmitido. En caso de no coincidir estamos ante un error, si el chequeo cierra podemos decir que para nuestro sistema no hubo errores.
Notemos que no podemos afirmar en forma absoluta la ausencia de error aunque el bit de paridad recalculado coincida con el recuperado. Esto es una regla general para todo sistema de codificación y en este caso se ve claramente: si cambian a la vez 2 bits del código o, en general, un número par de ellos, el sistema de detección de errores por paridad no funciona, en el sentido que no reporta el error.
Los sistemas de codificación con capacidad de detección (o detección y corrección) de errores funcionan correctamente dentro de ciertos límites, si se cumplen ciertas hipótesis de trabajo.
Un conjunto de supuestos que de cumplirse aseguran el buen funcionamiento de los sistemas que estamos analizando es:

(a) La probabilidad de que falle un bit es baja.
(b) Las fallas de bits son sucesos independientes (la posibilidad que un bit falle no tiene relación alguna con la falla de otro bit del código).

En esta hipótesis la probabilidad de que fallen 2 bits a la vez es igual al cuadrado de la probabilidad que falle uno (por la segunda hipótesis) que ya era un valor pequeño (por la primera) por lo que da un valor muy bajo. Por lo tanto en estas hipótesis el sistema de detección por paridad funciona bien: cuando no detecta error es altamente probable que el código sea efectivamente el correcto.
Es importante señalar que las hipótesis efectuadas se ajustan al caso de las memorias de las computadoras modernas. No ocurre lo mismo con los dispositivos de almacenamiento que graban la información en forma serial (un bit a continuación del otro) ni con los sistemas de transmisión de datos seriales ya que en estos casos el hecho que falle un bit está vinculado, en forma no despreciable, a la falla de otro (en particular del vecino "anterior"), por lo cual la hipótesis b) no se cumple para estos sistemas, de donde el control por paridad no sería demasiado efectivo en estos casos.
Volviendo al mecanismo de bit de paridad y recordando la definición de la operación lógica XOR y su propiedad asociativa es fácil ver que el bit de paridad se puede expresar como

$$P = b_0 \oplus b_1 \oplus \ldots \oplus b_n \text{ (paridad par)}$$

o

$$P = (b_0 \oplus b_1 \oplus \ldots \oplus b_n)' \text{ (paridad impar)}$$

El chequeo de la corrección de un código recuperado se puede realizar evaluando la expresión (para paridad par)

$$P \oplus b_0 \oplus b_1 \oplus \ldots \oplus b_n$$

con los valores de P, $b_0 \ldots b_n$ recuperados. Si el resultado es 0 "no hubo error", si es 1 entonces se detectó un error.
Una variante de este sistema es el mecanismo de paridad horizontal/vertical. Este método se puede aplicar cuando se almacenan o transmiten varias palabras de código. Cada palabra de n bits de código tiene su bit de paridad (que se denomina paridad horizontal) y cada n palabras almacenadas/transmitidas se agrega una palabra de paridad generada como el XOR (bit a bit,

incluyendo el de paridad horizontal) que recibe el nombre de paridad vertical. Por ejemplo para n=8 sería algo así:

A_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_h
B_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_h
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
V_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_h

$a_h = a_0 \oplus a_1 \oplus . . . \oplus a_7$ (idem para b, c, d, e, f, g, h)
$v_i = a_i \oplus b_i \oplus . . . \oplus h_i$ (i= 0, 1, 2, 3, 4, 5, 6, 7, h)

2.5 Distancia.

La distancia entre dos representaciones binarias se define como el número de bits distintos entre los dos códigos.
Es decir si tenemos dos códigos binarios a y b

$(a_0, a_1, . . ., a_n)$
$(b_0, b_1, . . ., b_n)$

la distancia entre ellos esta dada por la cantidad de unos en el código formado por:

$(a_0 \oplus b_0, a_1 \oplus b_1, . . . a_n \oplus b_n)$

Por ejemplo, los códigos:

01101
10110

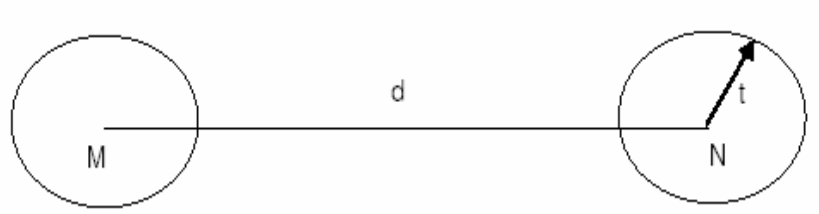
tienen una distancia 4 (cuatro)
La "distancia" tal cual la hemos definido tiene las siguientes propiedades (que no vamos a demostrar):

1) $d(a,b) = d(b,a)$
2) $d(a,b) = 0$ *si y sólo si* $a = b$
3) $d(a,b) + d(b,c) \geq d(a,c)$

Se pueden generar sistemas de codificación en binario que tengan una determinada distancia. En el caso de un sistema de códigos se llama "distancia" del código a la mínima distancia que exista entre dos valores válidos del código.
Por ejemplo el código "*2 de 5*" tiene la forma:

11000
10100
10010
10001
01100
01010
01001
00110
00101
00011

Como vemos este código tiene distancia 2 ya que todos los elementos del código difieren por lo menos en 2 bits (ej: *11000* y *10100*). De los 32 posibles objetos que permiten representar 5 bits, este código permite representar tan solo 10 objetos.
La posibilidad de detectar y corregir errores está fuertemente vinculada, como es de suponer, a la distancia del código (sistema de codificación) utilizado.
Consideremos un código de distancia "d" y dos objetos válidos del código M y N.
Consideremos el conjunto de valores posibles del código (no necesariamente válidos) que se obtienen de modificar hasta "t" bits del valor "M". Ídem para los del valor "N". Estos conjuntos se pueden visualizar como "esferas" en el espacio de "*n*" dimensiones (siendo n la cantidad de bits del código) rodeando a cada uno de los valores.



Como se puede ver si estos conjuntos no tienen puntos en común, podemos afirmar que, no solo podemos detectar un error, sino que además podemos afirmar, con alta probabilidad, a que código correcto corresponde un valor "degenerado" dado, permitiendo por tanto la "corrección" del error. Si los conjuntos tienen puntos en común no podremos corregir, pero si detectar errores, siempre y cuando dentro del conjunto de puntos asociado al valor M no incluya al valor N, puesto que si no, la alteración de un código válido puede conducir a otro válido, impidiendo así detectar el error. Es decir que la condición para que se puedan detectar errores es que:

$$t < d$$

la condición para que se puedan corregir es:

$$t < d/2$$

Por ejemplo para poder corregir errores de hasta un bit, hay que utilizar un código de distancia 3 por lo menos.

2.6 Bits de Redundancia.

Se puede demostrar que para generar códigos de distancia 3 para objetos representables en "k" bits, se necesitan utilizar "p" bits adicionales, llamados bits de redundancia, tal que se cumpla:

$$2_p \geq p + k + 1$$

Por ejemplo, supongamos que queremos diseñar un sistema de memoria de una computadora de 16 bits, de manera que sea capaz de corregir errores de hasta 1 bit. De acuerdo a la expresión necesitamos agregar 5 bits de redundancia para que esto sea posible.

$$2_5 = 32 \geq 5 + 16 + 1 = 22$$

2.7 Códigos de Hamming.

Los códigos de Hamming son una forma práctica de generar códigos de distancia 3. Los veremos a través de un ejemplo en concreto. Supongamos que tenemos 16 objetos representados, en principio, en binario:

$$a_4\, a_3\, a_2\, a_1$$

de acuerdo a lo visto antes precisamos 3 bits de redundancia (ya que $2_3 = 8 \geq 3 + 4 + 1 = 8$)

$$p_3\, p_2\, p_1$$

Si armamos el código (que tendrá entonces 7 bits) de la siguiente manera:

$$a_3\, a_2\, p_3\, a_1\, p_2\, p_1$$

y convenimos que las representaciones válidas de los 16 objetos a representar son aquellas en los cuales los bits de redundancia se calculan de la siguiente manera:

$$p_1 = a_4 \oplus a_2 \oplus a_1$$
$$p_2 = a_4 \oplus a_3 \oplus a_1$$
$$p_3 = a_4 \oplus a_3 \oplus a_2$$

podemos al recuperar el código calcular los bits "s" que están vinculados, al igual que los "p" a la posición de los distintos bits en el código:

	a_4	a_3	a_2	p_3	a_1	p_2	p_1
s_0	x		x		x		x
s_1	x	x			x	x	
s_2	x	x	x	x			

$$s_0 = p_1 \oplus a_4 \oplus a_2 \oplus a_1$$
$$s_1 = p_2 \oplus a_4 \oplus a_3 \oplus a_1$$
$$s_2 = p_3 \oplus a_4 \oplus a_3 \oplus a_2$$

estos bits "s" representan, en binario, un número "S"

$$S = S_2\, S_1\, S_0$$

Si al calcular "S" se da que es cero no hay errores (por lo menos para nuestro sistema de codificación). Si "S" da distinto de 0, el número que de me indica el bit que esta errado (siendo el 1 el

de más a la derecha, o sea p_1). De esta manera es posible reconstruir el valor correcto del código, cambiando el bit que identificamos como corrupto (si esta en 0 lo pasamos a 1 y viceversa).

2.8 Tipos de datos.

I. Tipo Carácter.

Comenzaremos viendo cómo se representan los caracteres mediante códigos binarios.
Entendemos por caracteres los símbolos que se utilizan en el lenguaje natural escrito: letras, números, símbolos de puntuación, símbolos especiales, etc.
En la actualidad, la forma más difundida de representar estos símbolos es la establecida por el denominado código ASCII (American Standard Code for Information Interchange). El ASCII es un código de 7 bits que especifica la representación de las letras y símbolos especiales usados en el idioma Inglés (más exactamente Inglés norteamericano) además de los números, y es muy similar al alfabeto número 5 del CCITT (Comité Consultivo Internacional sobre Telefonía y Telecomunicaciones), organismo que, entre otras funciones, establece propuestas de estandarización en materia de comunicaciones.
De los 128 valores posibles del código (7 bits) 10 se utilizan para los dígitos decimales (del 30h al 39h), 26 para las letras mayúsculas (del 41h al 5Ah), 26 para letras minúsculas (del 61h al 7Ah), 34 para símbolos especiales (espacio, !, #, $, %, /, &, +, -, *, etc.) y los 32 primeros se denominan genéricamente "caracteres de control" y se utilizan básicamente en la comunicación de datos y con fines de dar formato a los textos en impresoras y pantallas de video.
Algunos de estos caracteres de control son:
00h ⇨ NUL (Null) Es la ausencia de información, se utiliza como carácter de relleno.
02h ⇨ STX (Start of Text) Muchos protocolos de comunicación lo utilizan para indicar el comienzo de un texto.
03h ⇨ ETX (End of Text) Idem para fin de texto.
06h ⇨ ACK (Acknowledge) Se utiliza en comunicaciones para contestar afirmativamente la recepción correcta de un mensaje.
15h ⇨ NAK (Negative acknowledge) Idem para recepción incorrecta.
0Ah ⇨ LF (Line Feed) Indica pasar a la siguiente línea (en una impresora o pantalla).
0Ch ⇨ FF (Form Feed) Indica pasar a página siguiente.
0Dh ⇨ CR (Carriage Return) Indica volver a la primera posición dentro de la línea.
11h ⇨ DC1 (Data Control 1) Indica dispositivo libre (disponible).
12h ⇨ DC2 (Data Control 2) Indica dispositivo ocupado (no disponible).

Supongamos que enviamos caracteres a una impresora y ésta no está en condiciones de recibir más (ha llenado su "buffer") entonces la impresora envía el carácter DC2 al computador (para indicarle que no envíe más caracteres), cuando queda en condiciones de recibir nuevos caracteres, envía el carácter DC1. Cuando se utilizan caracteres con este fin, se dice que se hace uso de un protocolo XON/XOFF (Transmit on/Transmit off).

1Bh ⇨ ESC (Escape)

Indica el comienzo de una "secuencia de escape". Los caracteres que vienen a continuación tienen un significado especial. Estas secuencias se usan típicamente para enviar comandos a las impresoras y/o terminales de visualización. Por ejemplo, para posicionar el cursor en la pantalla, cambiar el tipo de letra en una impresora.
La forma más habitual de representar el código ASCII, y en general todos los sistemas de codificación de caracteres, es a través de una matriz cuyas columnas están asociadas a los 3 bits más significativos del código, y sus filas a los 4 bits menos significativos tal como se muestra en la siguiente tabla:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	A	Q
2	STX	DC2	"	2	B	R	B	R
3	ETX	DC	#	3	C	S	C	S
4	EOT	DC4	$	4	D	T	D	T
5	ENQ	NAK	%	5	E	U	E	U
6	ACK	SYN	&	6	F	V	F	V
7	BEL	ETB	'	7	G	W	G	W
8	BS	CAN	(	8	H	X	h	X
9	HT	EM	)	9	I	Y	i	Y
A	LF	SUB	*	:	J	Z	j	Z
B	VT	ESC	+	;	K	[	k	{

C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Tabla 2 - Código ASCII

Como ya hemos establecido, el ASCII codifica los caracteres utilizados para escribir textos en idioma Inglés y los signos de puntuación y símbolos especiales propios de dicha lengua. ¿Qué pasa entonces con otras lenguas como el español, en particular?. Cómo se puede observar, la Ñ, por ejemplo, no está codificada, tampoco las letras acentuadas, así como signos de puntuación característicos de nuestro idioma como "?" y "!". La situación en este campo es todavía confusa y no existen propuestas de codificación estándar aceptadas universalmente. Existen dos mecanismos básicos de atacar el problema: uno es modificar el ASCII de 7 bits adaptándolo a cada lengua; el otro es utilizar un código de 8 bits, que en sus primeros 128 valores coincida con el ASCII y los restantes utilizarlos para representar los caracteres propios de un conjunto relativamente grande de idiomas.

Un ejemplo del mecanismo de modificación del ASCII es el utilizado por EPSON en sus impresoras que se ha convertido en un "estándar de facto" en el mundo de las computadoras personales al ser aceptado y soportado por la casi totalidad de los otros fabricantes de impresoras. El mecanismo consiste en sustituir caracteres (en general símbolos especiales) por los caracteres que le "faltan" al ASCII para adaptarse a cada lengua en particular. Así tenemos un "ASCII español", un "ASCII francés", un "ASCII inglés (de Inglaterra)", etc.

Un ejemplo de código de 8 bits es el utilizado por IBM en sus computadoras personales, el cual puede considerarse también un "estándar de facto". Utiliza los 128 valores más altos para letras y símbolos de otros idiomas, para caracteres gráficos (que permiten dibujar recuadros y cosas similares). En la tabla 3 se muestran los caracteres "agregados" al ASCII (incompleta). Finalmente mencionamos un código de 8 bits que, al igual que el anterior, la "parte baja" es prácticamente igual al ASCII y en la "parte alta" representa letras y símbolos especiales así como 32 caracteres de control adicionales. El código que recibe el nombre de Multinacional ISO 8859/1, está representado en la tabla 4.

8	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	Ç	É	á	...		α	≡
1	Ü	Æ	í			β	±
2	É	Æ	ó			Γ	≥
3	Â	Ö	ú			Π	
4	Ä	Ò	ñ			Σ	⌠
5	À	Û	Ñ			σ	⌡
6	Å	Û	ª			µ	÷
7	Ç	Ù	º			τ	≈
8	Ê	Ÿ	¿			Φ	°
9	ë	Ö				Θ	•
A	È	Ü	¬			Ω	′
B	Ï	¢	½			δ	
C	Î	£	¼			∞	η
D	ì	¥	¡			∅	
E	Ä	×	«			∈	
F	Â	ƒ	»		**...**	∩	

Tabla 3 - Código ASCII de 8 bits.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
80			‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ			
90		‘	’	“	”	•	–	—	˜	™	š	›	œ			Ÿ
A0		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B0	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Tabla 4 - Multinacional ISO 8859/1.

II. Tipo String.

El tipo STRING es una sucesión de caracteres. Existen varias formas de representación interna. Lo principal a saber es dónde termina la sucesión.

- Una primera manera es emplear largo fijo. Esta representación es demasiado rígida.
- Una segunda manera es reservar un código especial para fin de string. Este código especial no podrá formar parte del string. Por ejemplo, el lenguaje "C" utiliza el NULL para fin de string.
- Una tercera manera consiste en convertir los strings en un registro donde el primer campo tiene el largo y el segundo tiene el contenido. El único inconveniente es que la estructura es más compleja. En BASIC se pueden encontrar ejemplos de esta representación.

III. Tipo Entero sin signo.

Los enteros sin signo (siempre positivos) poseen la representación más simple: como un entero en base 2. Se utilizan para contadores, direcciones, punteros y para derivar otros tipos.
Las operaciones elementales de este tipo son las cuatro usuales para los números enteros (+ - * /).
En la suma de 2 enteros sin signo, se aplica el algoritmo usual para los números binarios.
Ejemplo:

	0110000 ⇨ carry
25	11001
+74	+1001000
99	1100001

Los bits de carry de la operación anterior (0110000) se presentan en la primera línea de la operación.
El problema principal en esta representación es que estamos restringidos en el tamaño. Los tamaños usuales para representar los enteros sin signo son: el byte (0 a 255), la palabra de 2 bytes (16 bits, 0 a (2^{16})-1), y la palabra de 4 bytes (32 bits, 0 a (2^{32})-1).

IV. Tipo entero con signo.

2.8.4.1 Valor absoluto y signo.

Si tenemos *n* bits para representar el número, tomamos uno para el signo y el resto representa el valor absoluto del número en binario.

Msb				lsb
b_{n-1}	b_{n-2}	...	b_1	b_0

b_{n-1} = *Signo.*

$b_{n-2} \ldots b_1 b_0$ = *Valor absoluto*

- Si b_{n-1} = 1 entonces el número es negativo.
- Si b_{n-1} = 0 entonces es positivo.

Ejemplo: (en 4 bits)

0110 ⇨ 6

1110 ⇨ 6

Para *n* bits el rango del número representado es:

$-(2^{n-1}-1) \leq N \leq 2^{n-1}-1$

En esta representación tenemos dos formas de representar el cero, 1000 y 0000 (para n=4).
Esto puede verse como un inconveniente. Además las operaciones no trabajan directamente con la representación sino que deben interpretarse en base a los signos relativos.

2.8.4.2 Complemento a uno.

Los números positivos se representan en binario, y los números negativos se representan como el valor absoluto complementado bit a bit.

Para *n* bits el rango del nro. representado es:

$-(2^{n-1}-1) \ldots\ldots\ldots 2^{n-1}-1$

n = 8

		msb	***lsb***
5	⇨	*0000*	*0101*
-5	⇨	*1111*	*1010*

Ejemplo: (*n = 4*)

-7	*1000*
-6	*1001*
-5	*1010*
-4	*1011*
-3	*1100*
-2	*1101*
-1	*1110*
0	*0000*
0	*1111*
1	*0001*
2	*0010*
3	*0011*
4	*0100*
5	*0101*
6	*0110*
7	*0111*

El orden en binario, no corresponde al orden de los números que representa. Otra desventaja, es que hay 2 representaciones distintas para el cero (*0000 y 1111*).

2.8.4.3 Desplazamiento.

La representación por desplazamiento supone un corrimiento de los valores a representar según un valor *d* (llamado desplazamiento), y posteriormente se la aplica el módulo para que se pueda almacenar en el tamaño de la representación deseada. Para el desplazamiento, se supone que el valor codificado (resultado de la operación $N + d$), es un número que para n bits es un valor entre *0* y 2^n-1, por lo que permite representar valores desde $-d$ a 2^n-d-1.En general para representar 2^n números diferentes, se asigna a *d* el valor 2^{n-1}, o $2^{n-1}-1$ aplicando un módulo de 2^n.

N⇨(N + d) representado con n bits

Ejemplo: *n=4, d= 8*

-8 ⇨ 0000

-7 ⇨ 0001

....

....

-1 ⇨ 0111

0 ⇨ 1000
1 ⇨ 1001
....
....
7 ⇨ 1111

El rango para *n* bits es: $-(2)^{n-1} \ldots 2^{n-1}-1$.
La propiedad más importante de esta representación es que los códigos conservan el orden de los números, con o sin signo. En particular, toda representación de un número negativo es menor que cualquiera de un número positivo. Otra ventaja, es que hay una sola representación para el cero.
El principal inconveniente de esta representación es que los algoritmos de las operaciones usuales son más complejos.

2.8.4.4 Complemento a dos.

Los números positivos se representan directamente en binario y para conseguir el código de los negativos, se complementa el valor absoluto y se los incrementa en uno.

$N_{(10} \Rightarrow N_2$ *si* $N_{(10} \geq 0$
$N_{(10} \Rightarrow not\ (N_2) + 1$ *si* $N_{(10} < 0$

Por ejemplo, para números de 8 bits se tiene:
Si *70 = 01000110*, entonces para lograr *-70* hago el complemento a uno (negación bit a bit) de esta configuración y luego le sumo uno, luego:

01000110	(70)
10111001	(complemento a 1)
+1	
10111010	(-70)

Las propiedades más importantes de esta representación son:

- Mantiene la suma (la suma con o sin signo es la misma operación, es decir que el algoritmo es el mismo).
- Es coherente la representación del cero:

00000000	⇨	0
11111111	⇨	Not 0
+1		
00000000		

repr(0) = rep(-0)

- Se pierde la relación de orden. El algoritmo de comparación de A < B depende de los signos de A y de B.
- Veamos el algoritmo de la resta.
  A - B = A + (-B) = A + not B + 1

Sucede que si nos interesa sumar *70 + 70 = 140* siguiendo este procedimiento, veremos que la suma no es representable en 8 bits. Aparece entonces un bit que llamamos de *overflow*. Se puede detectar el "error" utilizando un bit adicional, llamado bit de *carry*. Para saber si hubo *overflow* al final de la operación, verifico si hubo acarreo en uno, (y solamente en uno) de los dos bits más significativos, en caso contrario (si existió carry en ninguno o en ambos), el resultado es correcto.
En la multiplicación debemos escribir el algoritmo. Lo que se hace es multiplicar números positivos y luego colocarles el signo según las reglas algebraicas. El algoritmo que se usa es el mismo que en la escuela, girando uno de los factores y luego sumar.

V. El tipo decimal.

Se usan para sumar cantidades de muchos dígitos donde no se puede perder precisión. En aplicaciones comerciales se tiene un caso típico. Pueden ser del largo que se quiera y se representan internamente en forma similar un STRING.

```
type DECIMAL is array (1..LARGO) of DIGITO;
type DIGITO is (0,1,2,3,4,5,6,7,8,9);
```

Los algoritmos son similares a los empleados en las operaciones decimales hechas a mano.
Observamos que en la representación en 8 bits para dígitos, se usan 10 de los 256 códigos. La codificación típica es:

0 ⇨ 00000000
1 ⇨ 00000001
.....
9 ⇨ 00001001

En esta representación se emplean 8 bits para cada lugar, cuando solamente necesita 4 bits. Por esta razón se definen los decimales empaquetados, en los cuales se codifica con 4 bits. También se les llama enteros BCD (Binary Coded Digit).

Vemos un ejemplo de codificación:

$$12545 \Rightarrow \begin{array}{ccccc} 0001 & 0010 & 0101 & 0100 & 0101 \\ \hline 1 & 2 & 5 & 4 & 5 \end{array}$$

Veamos un ejemplo de operación suma en el caso de dos dígitos. La suma binaria es:

$$\begin{array}{rcl} 7 & \Rightarrow & 00000111 \ \text{(bcd)} \\ +5 & \Rightarrow & 00000101 \ \text{(bcd)} \\ \hline 12 & \Rightarrow & 00011000 \end{array}$$

El resultado puede ser o no BCD. Si no es, debe ser corregido. El algoritmo de corrección consiste en restar 10 al dígito menos significativo y sumar 1 al dígito siguiente. Pero sumar 1 al dígito más significativo es sumar 16 a todo, por tanto le sumo 6 y obtengo los dígitos correctos. Es decir, por ser los dígitos de 4 bits, sumar 1 al de la izquierda, es como sumar 16 al nro. de 8 bits formado por ambos.

$$\begin{array}{rcll} 7 & \Rightarrow & 00000111 \ \text{(bcd)} & \\ +5 & \Rightarrow & 00000101 \ \text{(bcd)} & \\ \hline 12 & \Rightarrow & 00011000 & \text{Como no es válido le sumo 6} \\ +6 & \Rightarrow & 0110 & \\ \hline 12 & \Rightarrow & 00010010 & \end{array}$$

El algoritmo de suma para decimales empaquetados es: sumar binario, dígito a dígito del menos al más significativo, si se encuentra un dígito no valido o hay un carry, sumar 6 y obtener el dígito correcto y el carry.

VI. Representación en punto flotante

Para expresar números reales utilizamos la siguiente notación:

$$n = (-1)^s . b^e . F$$

donde: s es el bit de signo
e es el exponente
F es la mantisa
b es la base de representación (se utiliza 2).

Dado que esta representación es ambigua (existen varias representaciones para un mismo numero) se utiliza una versión más restringida que se llama normalizada. Los números normalizados son aquellos en que el bit más significativo de la mantisa es distinto de cero o, que es equivalente, que la mantisa sea máxima.

2.9 Estándar IEEE 754 de punto flotante

Para que los números representados en punto flotante fueran posibles intercambiar con distintas arquitecturas se establece el estándar IEEE 754 que define el formato y las operaciones con estos.

El estándar define tres formatos :

	s(bits)	e(bits)	F(bits)	Total(bytes)
Simple precisión	1	8	23	4
Doble precisión	1	11	52	8
Precisión extendida	1	15	64	10

Los números se almacenan de la siguiente forma:

s	e	F

Donde el exponente se representa con desplazamiento. Los números normalizados son de la forma: 1, F. donde el bit más significativo de la mantisa es un 1. Como todos los números normalizados tienen un uno en el bit más significativo el estándar define una representación diferente que omite este bit.
Este consiste en : un 1 implícito, una coma implícita y luego la mantisa.
Por lo tanto la representación a utilizar es de la forma:

$n=(-1)^s.2^{e+127}.(1,F)$ para números de simple precisión
$n=(-1)^s.2^{e+1023}.(1,F)$ para números de doble precisión

Además de operar con números normalizados el estándar opera con números desnormalizados :

Normalizados	0<Exp<Max	Cualquier Combinación
Desnormalizados	0	<>0
Cero	0	0
Infinito	1111……1	0
No válido	1111……1	<>0

Los números desnormalizados sirven para operar con números menores que el menor número normalizado representable. Estos números asumen un 0 implícito en vez del 1 implícito de los números normalizados. Por lo tanto cuando tenemos un número en notación punto flotante desnormalizado estamos representando el número:

$(-1)^s.2^{e+127}.(0,F)$ donde e es siempre cero.

3 ALGEBRA DE BOOLE

3.1 Introducción.

El álgebra de Boole es una herramienta de fundamental importancia en el mundo de la computación. Las propiedades que se verifican en ella sirven de base al diseño y la construcción de las computadoras que trabajan con objetos cuyos valores son discretos, es decir las computadoras digitales, en particular las binarias (en las cuales los objetos básicos tienen solo 2 valores posibles) las que son, en definitiva, la casi totalidad de las computadoras de uso corriente.
Desde ya adelantemos que no se verán aquí detalles formales de la construcción algebraica, ni todas las propiedades que se verifican, así como tampoco todos los métodos de síntesis de funciones booleanas que habitualmente se incluyen en este tema en cursos de lógica y/o diseño lógico.
Como toda álgebra, la de Boole parte de un cuerpo axiomático, el cual puede adquirir diversas formas, variando la cantidad y calidad de los axiomas. Aquí en particular tomaremos uno: el propuesto por Huntington en 1904 que tiene la ventaja de ser consistente e independiente.

3.2 Axiomas.

1. Existe un conjunto G de objetos, sujetos a una relación de equivalencia, denotada por "=" que satisface el principio de sustitución. Esto significa que si $a = b$, b puede sustituir a a en cualquier expresión que la contenga, sin alterar la validez de la expresión.
2.
    a. Se define una regla de combinación "+" en tal forma que $a + b$ está en G siempre que tanto a como b lo estén.
    b. Se define una regla de combinación "·" en tal forma que $a \cdot b$ está en G siempre que tanto a como b lo estén.
3. Neutros.
    a. Existe un elemento 0 en G tal que para cada a de G: $a + 0 = a$
    b. Existe un elemento 1 en G tal que para cada a de G: $a \cdot 1 = a$
4. Conmutativos. Para todo par de elementos a y b pertenecientes a G se cumple:
    a. $a + b = b + a$
    b. $a . b = b . a$
5. Distributivos. Para toda terna de elementos a, b, c pertenecientes a G se cumple:

a. $a + (b \cdot c) = (a + b) \cdot (a + c)$
b. $a \cdot (b + c) = a \cdot b + a \cdot c$

6. Complemento. Para cada elemento a de G existe un elemento $\overline{a}$ tal que:

$$a \cdot \overline{a} = 0$$

$$a + \overline{a} = 1$$

7. Existen por lo menos dos elementos x, y en G tal que x <> y

La similitud de muchos de estos postulados con los del álgebra común. Sin embargo, la primera de las reglas distributivas (sobre la suma) y la existencia del complemento diferencian en forma fundamental esta álgebra de la común.

3.3 Modelo aritmético.

El ejemplo más simple del álgebra de Boole se compone de un conjunto G de 2 elementos:
"0" y "1". Como es natural estos dos elementos deben coincidir con los neutros de las reglas de combinación para satisfacer el axioma 3. Las reglas de combinación debemos definirlas de manera de satisfacer los axiomas.
Así de acuerdo al axioma 3 :

$0 + 0 = 0 \quad 0 \cdot 1 = 0$

$1 + 0 = 1 \quad 1 \cdot 1 = 1$

De acuerdo al axioma 4

$0 + 1 = 1 \quad 0 \cdot 1 = 0$

y teniendo presente el axioma 5 :

$1 + (1 \cdot 0) = (1 + 1) \cdot (1 + 0)$ (5a con a = 1, b = 1, c = 0)

$1 + 0 = (1 + 1) \cdot 1$

$1 = 1 + 1$ (por axioma 3)

$0 \cdot (0 + 1) = (0 \cdot 0) + (0 \cdot 1)$ (5a con a = 0, b = 0, c = 1)

$0 \cdot 1 = (0 \cdot 0) + 0$

$0 = 0 + 0$ (por axioma 3)

Por lo tanto las reglas *completas* **son:**
Nosotros usaremos esta versión "binaria" del álgebra de Boole.

3.4 Propiedades.

- Dualidad

Si analizamos los postulados veremos que los mismos se presentan de a pares y en tal forma que uno de la pareja se obtiene de otro cambiando "0" por "1" junto con "+" por "·" (y viceversa). Esto asegura que cada propiedad que se demuestre en esta Álgebra tiene una "dual" que también es cierta (para demostrar la dual bastaría con repetir la demostración realizada sustituyendo cada postulado o propiedad utilizada por su dual).

- Asociativa

a) $a + (b + c) = (a + b) + c$

b) $a \cdot (b \cdot c) = (a \cdot b) \cdot c$

Si bien las leyes asociativas son muchas veces incluidas dentro del cuerpo axiomático, de hecho son demostrables a partir de los axiomas aquí presentados, (demostración que no haremos) por lo cual las presentamos como propiedades.

- Idempotencia

Para todo elemento en G se cumple:

$a + a = a$

$a.a = a$

Demostración:

$a + a = (a + a).1$ (3b)

$a + a = (a + a).(a + \overline{a})$ (6)

$a + a = a + (a.\overline{a})$ (5a)

$a + a = a + 0$ (6)

$\Rightarrow a + a = a$

$\Rightarrow a.a = a$ (Dualidad)

- Neutros Cruzados

Para todo elemento en G se cumple

$a + 1 = 1$

$a.0 = 0$

Demostración:

$a. + 1 = a + (a + \overline{\overline{a}})$ (6)

$a. + 1 = (a + a) + \overline{\overline{a}}$ (asociativa)

$a. + 1 = a + \overline{\overline{a}}$ (idempotencia)

$\Rightarrow a. + 1 = 1$

$\Rightarrow a..0 = 0$ (dualidad)

Entonces los axiomas 1, 2, 3, 4, 5 y 7 se satisfacen por definición y es fácil verificar que el G (complemento) también es cierto.
Construimos por lo tanto un modelo "aritmético" de álgebra de Boole que podemos denominar "binario" y es en definitiva con la que trabajaremos.

Muchas veces las reglas de combinación se presentan como tablas (como las funciones booleanas más generales que veremos más tarde)

a	b	a+b
0	0	0
0	1	1
1	0	1
1	1	1

a	b	a.b
0	0	0
0	1	0
1	0	0
1	1	1

En general notaremos a·b como ab, además la operación · tendrá mayor precedencia que la operación +.

- Complemento de complemento

Para cada elemento de G se cumple : $a = \overline{\overline{a}}$

Para todo par de elementos de G se cumple :

$a + ab = a$

$$a.(a + b) = a$$

Para todo par de elementos de G se cumple :

$$a + \bar{a}b = a + b$$

$$a.(\bar{a} + b) = ab$$

- **Ley de *De Morgan***

Para todo par de elementos de G se cumple :

$$a + ab = a$$

$$a.(a + b) = a$$

Estas reglas de De Morgan pueden generarse para cualquier número de variables.

3.5 Modelo lógico.

Los valores que pueden asignarse a un juicio, desde el punto de vista lógico, son dos: verdadero (V) o falso (F).
Un juicio al cual se le aplica el operador lógico no (negación) forma un nuevo juicio.
Dos juicios pueden combinarse para formar un tercero mediante los operadores lógicos "o" e "y".
Si vinculamos los valores booleanos 0 y 1 con los valores lógicos F y V respectivamente, encontramos que las operaciones del álgebra de Boole "binaria" asigna correctamente los valores lógicos del juicio combinación.
Esto se comprueba observando que:

verdadero o verdadero es verdadero
verdadero o falso es verdadero
falso o verdadero es verdadero
falso o falso es falso

verdadero y verdadero es verdadero
verdadero y falso es falso
falso y verdadero es falso
falso y falso es falso

Por lo cual se puede concluir que el modelo "lógico" es isomorfo con el "aritmético" (binario) realizando la correspondencia.

$$F \Leftrightarrow 0$$
$$V \Leftrightarrow 1$$
$$\vee \Leftrightarrow +$$
$$\wedge \Leftrightarrow .$$
$$\neg \Leftrightarrow \bar{}$$

Es posiblemente consecuencia de este isomorfismo que las reglas de combinación "+" y "." del álgebra de Boole reciban los nombres de OR ("o" en inglés) y AND ("y" en inglés) respectivamente.

3.6 Modelo circuital.

Otro modelo posible es el que surge de considerar llaves eléctricas y asociar el valor A a la llave abierta (no pasa corriente, circuito abierto) y el valor C con la llave cerrada (pasa corriente, circuito cerrado).
Es fácil comprobar que la combinación de llaves en paralelo o en serie cumplen las mismas reglas definidas en el modelo aritmético para "+" y "." respectivamente.

LL1	LL2	Circuito PQ"+"
A	A	A
A	C	C
C	A	C
C	C	C

LL1	LL2	Circuito PQ"."
A	A	A
A	C	A
C	A	A
C	C	C

Entonces existe también un isomorfismo entre el modelo "circuital" y el "aritmético" si hacemos la asociación

$A \Leftrightarrow 0$
$C \Leftrightarrow 1$
$paralelo \Leftrightarrow +$
$serie \Leftrightarrow .$

Este isomorfismo es de fundamental importancia para la construcción práctica de las computadoras binarias.

3.7 Expresiones booleanas.

Llamamos constante a todo elemento del conjunto G que define al álgebra. Las variables podrán tomar como valor cualquier elemento de G (0 o 1 en el caso en que trabajamos).
Una expresión la podemos definir recursivamente como

1. las constantes y las variables
2. el complemento de una expresión booleana
3. el OR (+) o el AND (·) de dos expresiones booleanas.

3.8 Funciones booleanas.

Una función "F" de n variables $x_1 \ldots x_n$ booleanas es una aplicación del espacio G_n sobre el espacio G de tal forma que para cada valor posible de la n-upla $x_1 \ldots x_n$, donde cada variable puede tomar cualquier valor del conjunto (en nuestro caso {0, 1}), se asocia un valor del recorrido G.
Una de las formas de expresar F es a través de las denominadas tablas de verdad que indican el resultado de F para cada valor posible de la n-upla; por ejemplo :

a	b	c	d
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Otras formas de representar F incluyen el método de indicar sólo los puntos en los cuales F vale 1 o sólo los puntos en los cuales vale 0 (representaciones Σ y Π respectivamente). Para indicar los puntos en que la función vale 1 puede usarse la notación σ en lugar de Σ.
Por ejemplo, la función anterior se puede expresar :

$$f(a,b,c) = \sum (1,4,5,7) \qquad f(a,b,c) = \Pi (0,2,3,6)$$

Otra forma de expresar las funciones es a través de expresiones; ejemplo, la función anterior sería:

$$f(a,b,c) = ac + a\overline{b} + \overline{a}\,\overline{b}c$$

3.9 Conectivas binarias.

Un caso interesante de estudiar es el de las funciones booleanas de 2 variables. Por ser dos variables las combinaciones posibles son 4, es decir "F" tiene 4 duplas (4 puntos) por tanto existen 16 funciones booleanas de dos variables posibles. Algunas de ellas no son de interes, veamos las tablas de verdad de las más útiles.

a	b	or	and	xor	nor	nand	Equiv.	Idemp	Tautol.
0	0	0	0	0	1	1	1	0	1
0	1	1	0	1	0	1	0	0	1
1	0	1	0	1	0	1	0	0	1

1	1	1	1	0	0	0	1	0	1

Nota :
la NOR es el complemento de la OR
la NAND es el complemento de AND
la XOR ("O" exclusivo) puede definirse como: $b\ a\ b\ a\ b\ a + = \oplus$

I. OR exclusivo.

El XOR es una función muy importante (es la suma aritmética binaria módulo 2) y cumple las propiedades :

1) Asociativa
2) Conmutativa
3) Distributiva: $a(b \oplus c) = ab \oplus ac$
4) $a \oplus 0 = a$
5) $a \oplus 1 = \overline{a}$
6) $a \oplus a = 0$
7) $a \oplus b \oplus ab = a \oplus b$
8) Cancelativa: $a \oplus b = a \oplus c \rightarrow b = c$

II. Suma de productos canónicos.

Desarrollaremos a continuación un método sistemático para encontrar una expresión algebraica para una función cualquiera dada.
Definamos producto canónico de n variables x_1 ... x_n al producto de todas ellas en el cual cada variable aparece una y sólo una vez, en forma simple o complementada.
Una suma de productos canónicos es una expresión formada sumando productos canónicos.
Existe un teorema (que no demostraremos) que afirma que toda función f de n variables puede expresarse como:

$$\begin{aligned} f(x_1, x_2, ..., x_n) = {} & x_1.x_2.x_3 ..... x_n.f(1,1,1,....., 1) + \\ & \overline{x_1}.x_2.x_3 ..... x_n.f(0,1,1,....., 1) + \\ & .x_1\overline{x_2}.x_3 ..... x_n.f(1,0,1,....., 1) + \\ & ..... \quad ... + \\ & \overline{x_1}.\overline{x_2}.\overline{x_3} ..... \overline{x_n}.f(0,0,,....., 0) + \end{aligned}$$

Es decir toda función de n variables puede expresarse como la suma de todos sus productos canónicos afectado cada producto canónico por un coeficiente. Este coeficiente es el valor de la función evaluado en el punto tal que las variables que en el producto canónico asociado aparecen simples tengan el valor 1 y las que aparecen complementadas el valor 0.
Este teorema, de fundamental importancia nos permite enunciar un método de construcción de una expresión que represente una función dada. El método es el siguiente: se tienen en cuenta sólo los puntos en los que la función vale 1 (por el teorema los productos asociados con los puntos en los que la función vale 0 desaparecen por estar afectados por un coeficiente nulo: el propio valor de la función); en esos puntos se busca el producto canónico asociado que es aquel donde la variable aparece simple si en el punto vale 1 o complementada si vale 0. De esta manera la función puede expresarse como suma de los productos canónicos así elegidos.

Por ejemplo sea la función de tres variables

a	b	c	f	
0	0	0	0	
0	0	1	1	⇨ $\overline{a}\overline{b}c$
0	1	0	0	
0	1	1	1	⇨ $\overline{a}bc$
1	0	0	1	⇨ $a\overline{b}\overline{c}$
1	0	1	0	
1	1	0	0	
1	1	1	0	

Entonces f puede expresarse como $f(a,b,c) = \overline{a}.\overline{b}.c + \overline{a}.b.c + a.\overline{b}.\overline{c}$

III. Productos de sumas canónicas.

Como todo en el álgebra de Boole, existe un método dual del anterior: el producto de sumas canónicas. En este caso deben considerarse los puntos en los que la función vale 0 y buscar las sumas canónicas asociadas que son aquellas en las que la variable aparece simple si tiene valor 0 y complementada si tiene valor 1.

3.10 Operadores lógicamente completos.

Un conjunto de operadores se llama lógicamente completo si cualquier función booleana puede expresarse mediante los mismos.
Del teorema de los productos canónicos, ya enunciado, se extrae una conclusión fundamental tanto del punto de vista lógico como del circuital: el conjunto de operadores +, . y ' es lógicamente completo.
Otra consecuencia es que para probar que un cierto conjunto de operadores es lógicamente completo, alcanza con probar que con ellos se pueden implementar el OR, el AND y el NOT (complemento).
Es fácil probar que el conjunto OR, NOT es lógicamente completo notando que el AND se puede construir como:

$$\overline{(ab)} = \overline{a} + \overline{b}(porDeMorgan) \Rightarrow a.b = \overline{(\overline{a} + \overline{b})}$$

Más interesante aún es mostrar que un solo operador, como el NAND, es lógicamente completo.

Debemos ver como implementar el NOT y el AND y el OR (representaremos con # el NAND):

$a\#a = \overline{a.a} = \overline{a}.$ (complemento)

$(a\#b)\#(a\#b) = \overline{(a\#b)} = \overline{\overline{(a.b)}} = a.b$ (AND)

$(a\#a)\#(b\#b) = \overline{(\overline{a}.\overline{b})} = \overline{\overline{(a+b)}} = a+b$ (OR) por De Morgan

Por lo cual el NAND es lógicamente completo.

3.11 Simplificación.

Hasta ahora hemos visto un método sistemático de expresar las funciones booleanas como expresión de sus variables. Pero este método no asegura que la expresión lograda sea la más simple posible. El hecho que la expresión de una función sea lo más simple posible no es algo trivial o caprichoso, es de fundamental importancia en la construcción práctica de circuitos lógicos, por eso analizaremos algunos métodos para simplificar expresiones booleanas, de manera de aplicarlos a las expresiones obtenidas como sumas de productos canónicos.

I. Método algebraico.

El método consiste en la aplicación, más o menos ingeniosa, de transformaciones algebraicas de manera de lograr expresiones más sencillas. Por supuesto que este no es un método sistemático, pero es la base, al fin, de los métodos sistemáticos.
Resumamos aquí algunas propiedades vistas del álgebra que serán de utilidad en la tarea de simplificar:

1) $f.\overline{f} = 0$

2) $f + \overline{f} = 1$

3) $g.f + \overline{g}.f = f$

4) $g.f + f = f$

5) $f + \overline{f}.g = f + g$

Veamos un par de ejemplos de como se aplican estas propiedades para reducir expresiones:

a) Sea $f_1 = \bar{a}bc + \bar{a}b\bar{c} + ab\bar{c}$

Por la aplicación de la propiedad 3 a los primeros dos términos y a los dos últimos queda

$$f_1 = \bar{a}b + b\bar{c}$$

b) Sea $f_2 = \bar{a}\bar{b}c + a\bar{b}c + a\bar{b}\bar{c} + \bar{a}bc$

Aplicando la propiedad 3 a los dos primeros términos queda

$$f_2 = \bar{b}c + a\bar{b}\bar{c} + \bar{a}bc$$

$$f_2 = \bar{b}(c + a\bar{c}) + \bar{a}bc \quad \text{(por la propiedad distributiva)}$$

$$f_2 = \bar{b}(c + a) + \bar{a}bc \quad \text{(por la propiedad 5 al paréntesis)}$$

$$f_2 = c(\bar{b} + \bar{a}b) + a\bar{b} \quad \text{(por distributiva aplicada dos veces)}$$

$$f_2 = c(\bar{a} + \bar{b}) + a\bar{b} \quad \text{(por propiedad 5 al paréntesis)}$$

Entonces la expresión de f_2a la que llegamos es:

$$f_2 = a\bar{b} + \bar{a}c + \bar{b}c$$

Esta sin embargo no es la expresión más reducida de f_2. Vemos como hubiera quedado aplicando la propiedad 3 al primer y cuarto miembro y al segundo y tercero

$$f_2 = \bar{a}c + a\bar{b}$$

siendo esta sí, la expresión más reducida.

Como vemos entonces el procedimiento descrito no asegura reducir la expresión a un mínimo ya que depende de como se elijan las propiedades a aplicar y los términos sobre los que se aplican.

II. Métodos sistemáticos.

Los modelos sistemáticos se basan en la propiedad 3 $g.f + \bar{g}.f = f$ y son básicamente uno "gráfico" (Diagrama de Karnaugh) y otro "algorítmico" o "numérico" (Método de Quine- McKlusky)

A continuación veremos una introducción al método gráfico

3.11.2.1 Diagrama de Karnaugh.

Este método consiste en representar en forma gráfica una función como suma de productos canónicos y hacerlo de tal forma que sea sencillo establecer procedimientos sistemáticos para hallar las agrupaciones de términos más convenientes para simplificar la expresión. Esto se logra utilizando una cuadrícula en la cual a cada cuadrado elemental corresponde un producto canónico posible y tal que al pasar de uno a otro cualquiera de sus cuatro adyacentes solo cambie el valor de una de las variables en juego. Por ejemplo para tres variables la cuadrícula es:

c/ab	00	01	11	10
0				
1				

Para cuatro variables:

cd/ab	00	01	11	10
00				
01				
11				
10				

En esta cuadrícula se marcan con "1" los lugares para los cuales la combinación de valores de las variables hace que la función valga 1 y el método consiste en buscar agrupar los "unos" formando los rectángulos más grandes posibles (que tengan todos 1 en su interior), repitiendo este proceso hasta que todos los puntos donde la función vale "1" estén comprendidos en algún rectángulo (siendo la cantidad total de rectángulos utilizados la menor posible). Es necesario aclarar que la cantidad de elementos agrupados debe ser una potencia de 2.

Nota: el diagrama es circular (los de cada borde son adyacentes con los del borde simétrico)

Ejemplos:

cd/ab	00	01	11	10
00	1	1		
01	1	1	1	
11				
10				

cd/ab	00	01	11	10
00	1		1	1
01	1	1		
11				
10			1	1

Una vez realizado el proceso anterior la mínima expresión de la función se obtiene sumando los términos asociados a cada rectángulo los cuales son el producto de las variables (simples o complementadas) cuyo valor no cambia en él.
Por ejemplo en los diagramas anteriores sería

1) $f_3 = \bar{a}\bar{c} + b\bar{c}d$

2) $f_4 = a\bar{d} + \bar{a}\bar{b}\bar{c} + \bar{a}\bar{c}d$

Para finalizar veamos como se aplica el método al caso ya visto de la función:

$$f_2 = \bar{a}\bar{b}c + a\bar{b}c + a\bar{b}\bar{c} + \bar{a}bc$$

El diagrama de Karnaugh correspondiente es

c/ab	00	01	11	10
0				1
1	1	1		1

Entonces la función queda $f_2 = \bar{a}c + a\bar{b}$ la cual coincide, como era de esperar, con la expresión más reducida hallada por el método "algebraico".
La primer expresión de f_2 hallada con dicho método tenía además el término $c\ b$ que corresponde al rectángulo formado por los dos extremos inferiores del diagrama que aquí queda evidente que no era necesario puesto que con los otros dos términos cubrimos todos los puntos donde la función vale "1".