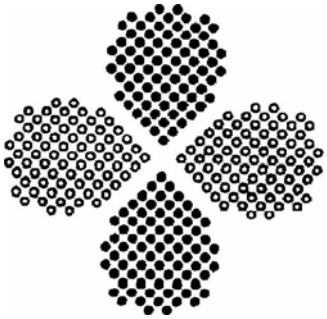
Representación de Datos y Aritmética Básica en Sistemas Digitales



Departamento de Sistemas e Informática

Escuela de Electrónica

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Universidad Nacional de Rosario

Ing. Rosa Corti

2005

Contenido del apunte

1. Introducción

Los sistemas digitales están presentes en muchas actividades cotidianas. Basta mencionar algunos ejemplos como cajeros automáticos, reproductores de CD, aplicaciones de telefonía, terminales de pago, sistemas de cómputo o juegos electrónicos para ver hasta que punto forman parte de nuestra vida diaria.

Como ya se ha visto en asignaturas previas, estos sistemas manipulan en forma natural datos binarios ya que en su arquitectura se utilizan componentes con dos estados diferenciados. Sin embargo, por la diversidad de las aplicaciones en que se los utiliza, los sistemas digitales necesitan también procesar otros tipos de datos como números para cálculos aritméticos, letras del alfabeto y símbolos que se usan con fines específicos. Por lo tanto, un sistema digital deberá en general ser capaz de realizar las siguientes tareas:

- ✓ Traducir la información del mundo real a un "lenguaje binario" que el sistema sea capaz de comprender y manipular.
- ✓ Procesar la información recibida en forma de ceros y unos obteniendo los resultados para los cuales el sistema ha sido diseñado.
- ✓ Devolver la/s respuesta/s de una forma comprensible para los usuarios.

Por lo tanto, un diseñador debe establecer una correspondencia entre los dígitos binarios que los sistemas digitales son capaces de comprender y procesar, y los eventos, cantidades y situaciones de la vida real. Este primer capítulo de la asignatura se ocupa entonces de analizar la forma de representar y manipular los distintos tipos de datos en sistemas digitales.

2. Sistemas de numeración

Los diferentes sistemas numéricos tienen una base o raíz r distinta. Esta base expresa la cantidad de símbolos del alfabeto con que el sistema representará a los distintos números. Cualquier número se construye como una serie de símbolos de dígito. Son sistemas posicionales, o sea, para determinar la cantidad que representa el número, se multiplicará cada dígito por la potencia entera de r que corresponda a su posición y se los sumará. Si tomamos como ejemplo el sistema decimal, cuyo alfabeto consta de 10 símbolos de dígito, el número 851,7 representa a la cantidad:

$$8 \times 10^2 + 5 \times 10^1 + 1 \times 10^0 + 7 \times 10^{-1}$$

2.1 Sistema de numeración binario.

Este sistema utiliza un alfabeto constituido por dos símbolos {0,1} y por lo tanto su base o raíz es dos. Por ejemplo la serie de símbolos de dígito 101101,11 representa a la cantidad:

$$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 45{,}75.$$

O sea $(101101{,}11)_2 = (45{,}75)_{10}$.

Queda claro entonces que la determinación de la cantidad representada brinda una forma sencilla de convertir números desde el sistema binario al decimal. Este procedimiento es general, y sirve para convertir datos desde un sistema de numeración de base r al sistema decimal. Por lo tanto, el valor decimal de un número en base r queda

representado por la suma de una serie de coeficientes multiplicados por las correspondientes potencias de r, como se muestra:

$$N = a_{n-1} \times r^{n-1} + a_{n-2} \times r^{n-2} + \ldots + a_2 \times r^2 + a_1 \times r^1 + a_{-1} \times r^{-1} + a_{-2} \times r^{-2} + \ldots + a_{-m} \times r^{-m}$$

O en notación más compacta: $$N = \sum_{i=-m}^{n-1} a_i \times r^i$$

Dónde r es la base con que se ha representado el número y el mismo consta de n dígitos a la izquierda del punto r-ario y de m dígitos a la derecha.

2.2 Conversión desde el sistema decimal

En general, la conversión de un número decimal a otro sistema de numeración en base r, se realiza separándolo en sus partes entera y fraccional. La parte entera se convierte mediante sucesivas divisiones por la base r hasta que el cociente entero resulta cero, acumulando los restos enteros. La conversión de la parte fraccional se logra a través de sucesivas multiplicaciones por r hasta que la parte fraccional resulta 0 , acumulando los dígitos enteros que se obtienen. Este procedimiento se fundamenta en la definición realizada de los sistemas de numeración, y del análisis de las cantidades que cada tira de símbolos representa en cada uno. Como ejemplo mostraremos en la Figura 1 la conversión del número $(41{,}6875)_{10}$ al sistema binario:

Parte entera = 41 Parte fraccionaria = 0.6875

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

0,6875
x 2
1,3750
x 2
0,7500
x 2
1,5000
x 2
1,0000

$(41)_{10} = (101001)_2$ $(0{,}6875)_{10} = (0{,}1011)_2$

$(41{,}6875)_{10} = (101001{,}1011)_2$

Figura 1: Conversión desde el sistema decimal al binario

Este procedimiento, se ha realizado desde el sistema decimal al binario, pero es general como se planteo al principio, para toda base r. Queda para el alumno realizar la ejercitación correspondiente a la conversión desde el sistema decimal hacia los otros sistemas de numeración estudiados.

2.3 Sistemas de numeración octal y hexadecimal.

En los sistemas digitales, y en particular cuando hablamos de computadoras digitales, son importantes para la representación de datos otros sistemas de numeración.

Esto se debe a que los registros y locaciones donde se almacena información tienen muchos bits, y por lo tanto, especificar su contenido mediante valores binarios implica tipear una larga tira de ellos. Por este motivo los diseñadores idearon otras formas de representar la información basadas en el uso de los sistemas de numeración octal (base 8) y hexadecimal (base 16). Al consultar bibliografía sobre el tema, se verá el amplio uso de estos sistemas de numeración que permiten una importante reducción de la notación. Además el lenguaje ensamblador que se estudia en esta asignatura, también hace un uso extenso de la notación hexadecimal, ya que el listado de un programa en este lenguaje presenta usando este sistema de numeración, todas las direcciones, instrucciones en código de código de máquina y el contenido de las constantes de datos.

El alfabeto del código octal es {0,1,2,3,4,5,6,7} y el del sistema hexadecimal { 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}. En el último, las letras de la A a la F representan a las cantidades desde 10 a 15 respectivamente.

Considerando el intenso uso de los sistemas de numeración binario, octal y hexadecimal en el área digital, resulta habitual la necesidad de realizar conversiones entre ellos. Estas conversiones son sencillas si tomamos en cuenta la relación existente entre las raíces o bases de estos tres sistemas de numeración: $2^3 = 8$ y $2^4 = 16$. Esta relación permite afirmar que tres dígitos binarios corresponden a un dígito octal y que cuatro dígitos binarios corresponden a un dígito hexadecimal. Como ejemplo de cómo esto permite realizar conversiones sencillas entre los sistemas, analizaremos el caso de un registro de 16 bits cargado con el número $(44899)_{10}$ expresado en binario. En la Figura 2 puede verse que comenzando por el bit menos significativo se forman grupos de 3 bits. A cada grupo se le asigna el dígito correspondiente en el sistema octal y la serie de dígitos octales obtenida es el equivalente en ese sistema de numeración del número $(44899)_{10}$. El procedimiento para la conversión a hexadecimal es análogo, sólo que los grupos a definir son de cuatro bits.

1	2			7			5			4			3			Sistema octal
1	0	1	0	1	1	1	1	0	1	1	0	0	0	1	1	Sistema binario
A				F			6			3						Sistema hexadecimal

Figura 2: Conversión entre sistemas binario, octal y hexadecimal.

Si el número binario posee bits a derecha e izquierda del punto binario, las conversiones se realizan hacia ambas direcciones desde el punto binario, formando siempre grupos de tres dígitos para el sistema octal y de 4 para el hexadecimal. En caso de ser necesario se completan los grupos de dígitos con ceros. En la figura 3 se presenta un ejemplo.

$$(10 \,.\, 1011001011)_2 = 010 \,.\, 101\ 100\ 101\ 100 = (2.5454)_8$$
$$= 0010 \,.\, 1011\ 0010\ 1100 = (2.B2C)_{16}$$

Figura 3: Conversión entre sistemas binario, octal y hexadecimal.

Estos ejemplos también son útiles para introducir la idea de "codificación binaria" de cualquiera de ambos sistemas de numeración. Desde este punto de vista se puede pensar que los grupos de tres bits (para el sistema octal) y los de cuatro (para el sistema hexadecimal) codifican en binario los dígitos correspondientes de ambos sistemas. En el punto siguiente volveremos sobre esta idea.

3. Representación decimal.

El sistema de numeración binario es el que utilizan los sistemas digitales, pero en la vida cotidiana todos estamos acostumbrados a utilizar el sistema decimal. Es por este motivo que las interfaces externas de muchos sistemas digitales pueden leer o presentar datos decimales. Para hacer esto es usual convertir a binario los números decimales para su procesamiento, volviendo a convertirlos al sistema decimal para presentar los resultados al usuario. Otra posibilidad es que el sistema opere directamente con números decimales, y de hecho, algunos sistemas lo hacen. Si bien la circuitería para ejecutar aritmética decimal es más compleja que la requerida para aritmética binaria, existen algunas ventajas al utilizarla. Las mismas se basan en que los usuarios utilizan el sistema decimal, y al recurrir a la aritmética decimal se evitan las conversiones necesarias para procesamiento binario. Por esta razón algunas computadoras y calculadoras operan directamente en decimal. Sin embargo, para poder abordar la aritmética decimal, es necesario codificar los números decimales de una manera que resulte comprensible para el sistema.

3.1 Números decimales codificados en binario (BCD).

En este caso necesitamos codificar los 10 símbolos utilizados por el sistema decimal en binario, lo que indica que necesitaremos tiras de cuatro dígitos binarios para diferenciarlos. O sea $2^3 = 8$ no nos alcanza por lo que necesitaremos $2^4 = 16$ aunque nos sobren 6 combinaciones (que es lo que ocurre siempre que las combinaciones que necesitamos codificar en base r no resultan ser una potencia de la base). Si bien pueden definirse muchos códigos distintos, el más utilizado es el que se muestra en la Tabla 1, que es conocido como código BCD.

Es importante destacar la diferencia entre la conversión de un número decimal al sistema binario, y la codificación binaria de un número decimal, que es lo que ahora proponemos. Si por ejemplo realizamos la conversión del número $(99)_{10}$ el resultado es: $(1100011)_2$. En cambio si lo codificamos utilizando la Tabla 1 será $(99)_{10} = (1001\ 1001)_{BCD}$. En el último caso la base del sistema sigue siendo 10 (el número es decimal) pero está codificado en binario.

Dígito decimal	Dígito decimal codificado en binario
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Tabla 1: Código para un dígito decimal.

4. Representación de enteros negativos

Para representar números negativos podríamos optar por dos formas distintas:

- ✓ Magnitud y signo
- ✓ Complementos

La primera opción es la que nos resulta más familiar, pero en los sistemas digitales no resulta precisamente la más sencilla de manipular, sobretodo cuando se piensa en implementar la operatoria correspondiente. Por este motivo, se recurre a la utilización de los llamados "complementos" que fueron ideados para simplificar la implementación de la operatoria necesaria. La simplicidad que introducen los complementos en los sistemas digitales se comprenderá con claridad cuando se analice la aritmética correspondiente en la seccion 5.

Existen dos tipos de complementos para cada sistema de numeración de base r: el complemento a r (complemento a la base) y el complemento a r-1 (complemento a la base menos 1 o complemento disminuido). Sin embargo es importante destacar, que la representación de magnitud y signo se utilizará cuando se trate con números reales. En ese momento se volverá entonces sobre este tipo de representación.

4.1 Complemento (r –1).

Dado un número N en base r con n dígitos, el complemento (r – 1) de N se define como $(r^n – 1) – N$.

4.1.1 Complemento a 1.

En el sistema binario, el complemento a (r – 1) se conoce como complemento a 1. Aquí el complemento a 1 de un número de n bits será $(2^n – 1) – N$. Si consideramos que por ejemplo n = 5, entonces $2^5 = (100000)_2$ y $2^5 – 1 = (11111)_2$. Por lo tanto, obtener el complemento a 1 de un número binario cualquiera de 5 bits significará restar cada uno de ellos de 1, lo que en este sistema ocasiona un cambio de 1 a 0 y viceversa. Esto muestra que obtener el complemento a 1 de un número binario resulta sumamente sencillo a nivel hardware, lo que justifica su uso.

Ejemplo:

$N = (01101)_2$

$-N = (10010)_2$

Este complemento tiene un inconveniente muy importante que es la doble representación del cero. El cero queda representado tanto por una tira de dígitos 0 (+0), como por $r^n – 1$ (-0). Esta doble representación obliga a la detección de la situación al realizar cualquier operación, lo que complica la implementación hardware de la unidad aritmética. Por este motivo, el complemento a la base menos uno constituye un paso intermedio para obtener el complemento a la base, que es el realmente utilizado para la operatoria.

4.1.2 Complemento a 9.

Para los números decimales el complemento a (r – 1) corresponde al complemento a 9. Si consideramos un número decimal N de n dígitos, su complemento a 9 resultará $(10^n – 1) – N$. Ahora, 10^n representa un uno seguido por n ceros, y $10^n – 1$ representa un

número formado por n nueves. Por lo tanto, el complemento a 9 de un número decimal se obtiene restando cada dígito de 9.

Ejemplo:

$N = (31.479)_{10}$

$-N = (68.520)_{10}$

4.2 Complemento r.

Dado un número N en base r con n dígitos, el complemento a r de N se define como $r^n - N$.

4.2.1 Complemento a 2.

En el sistema binario, el complemento r se conoce como complemento a 2. Aquí el complemento a 2 de un número de n bits será $2^n - N$. Por lo tanto el complemento a 2 de un número N binario puede obtenerse sumando 1 al complemento a 1 que resultaba tan sencillo de obtener. Otra forma de verlo es tener en cuenta que 2^n consiste en un uno seguido de n ceros. Por lo tanto si analizamos el número desde el bit menos significativo, para obtener el complemento a 2 se conservarán los ceros hasta encontrar el primer uno que no se alterará, y a partir de allí, los bits más significativos se invertirán (0 a 1 y viceversa).

Ejemplo:

$N = (01101)_2$

$-N = (10011)_2$

4.2.2 Complemento a 10.

Para los números decimales el complemento r corresponde al complemento a 10. Si consideramos un número decimal N de n dígitos, su complemento a 10 resultará $10^n - N$. Ahora, 10^n representa un uno seguido por n ceros, entonces el complemento a 10 de un número decimal se obtiene restando el dígito menos significativo de 10 y los restantes dígitos de 9. Esto es lo mismo que pensar que puede obtenerse este complemento sumando uno al complemento a 9.

Ejemplo:

$N = (31.479)_{10}$

$-N = (68.521)_{10}$

Nota 1: *En las definiciones anteriores no se tuvo en cuenta la coma o punto r-ario (pues estamos tratando números enteros). Si el número N posee coma r-aria, la misma debe quitarse para realizar la obtención del complemento y luego restituirse en la misma posición.*

Nota 2: *El complemento del complemento en cualquiera de sus formas nos devuelve siempre el número original N.*

4.3 Representación del signo.

Los números positivos, incluyendo el cero pueden representarse sin signo. Pero si deseamos representar números negativos necesitamos definir una notación que resulte conveniente. Estamos acostumbrados a preceder un número negativo con un signo menos y uno positivo con un signo más, pero esto no resulta práctico cuando trabajamos con sistemas digitales.

4.3.1 Representación del signo para el sistema binario.

La convención adoptada es representar el signo con un bit colocado a la izquierda del bit más significativo del número. Dicho *bit de signo* será 1 si el número es negativo y 0 si resulta positivo.

Ejemplo:

$N = (14)_{10} = (0\ 00001110)_2$

$-N = (-14)_{10} = (1\ 11110010)_2$ Utilizando complemento a 2.

$-N = (-14)_{10} = (1\ 11110001)_2$ Utilizando complemento a 1.

4.3.2 Representación del signo para el sistema decimal.

La representación adoptada para los números decimales es similar a la utilizada para el sistema binario. Sin embargo, la representación del signo en el sistema decimal se realiza con cuatro dígitos binarios para que resulte congruente con la representación de cada dígito decimal. También se acostumbra designar el signo más con cuatro ceros y el signo menos con el equivalente BCD del 9 (1001).

Ejemplo:

$N = (+\ 258)_{10} = (\ 0000\ \ 0010\ \ 0101\ \ 1000\ )_{BCD}$

$-N = (-\ 258)_{10} = (\ 1001\ \ 0111\ \ 0100\ \ 0010\ )_{BCD}$, en complemento a 10.

5. Operaciones aritméticas con enteros.

Presentaremos a continuación operaciones aritméticas con números enteros utilizando la representación con complementos ya descripta.

5.1 Suma aritmética.

Como los números a sumar pueden ser positivos o negativos, los representaremos en el complemento elegido y los sumaremos con sus correspondientes bits de signo. O sea, el bit de signo se incorpora al número para realizar las operaciones, y es tratado como un bit más. Esto explica la mayor sencillez en la resolución aportada por la utilización de los complementos.

5.1.1 Suma en complemento a la base.

Sean dos números A y B en base r. Deseamos obtener S = A + B. Para ello, analizaremos los distintos casos posibles con el fin de obtener conclusiones generales.

- *Caso 1:*

  $A > 0$ y $B > 0$

  $S = A + B$ Suma correcta. El resultado es positivo ya que ambos números lo son.

- *Caso 2:*

  $A < 0$ y $B < 0$

  $S = (r^n – A) + (r^n – B) = r^n + r^n – (|A| + |B|)$ Suma correcta si se ignora r^n. El resultado es negativo (ya que ambos números lo son) y queda expresado en el complemento correspondiente. El acarreo descartado es 1 pues se está sumando el resultado de la operación.

- *Caso 3:*

  $A < 0$ y $B > 0$ y $|A| < |B|$

  $S = (r^n – A) + B = r^n + (|B| - |A|)$ Suma correcta si se ignora r^n. El resultado es positivo por la relación propuesta para los valores absolutos. El acarreo desde r^n es igual a uno pues se suma una cantidad > 0.

- *Caso 4:*

  $A > 0$ y $B < 0$ y $|A| < |B|$

  $S = A + (r^n – B) = r^n - (|B| - |A|)$ Suma correcta. El resultado es negativo y está expresado en el complemento correspondiente.

Cuando sumamos números positivos y negativos utilizando complemento a r, se obtendrá el resultado correcto siempre, si se ignora r^n.

Esto confirma que la suma implementada con complemento a al base, es muy simple. Esta característica de sencillez y eficiencia ha resultado fundamental para su aceptación y uso generalizado.

Ejemplo: Se realizará considerando r = 2.

- *Caso 1:*

	Sistema decimal	Acarreo - Sistema binario
A	+ 6	0 0000110
B	+13	0 0001101
S = A + B	+19	0 0 0010011

- *Caso 2:*

	Sistema decimal	Acarreo - Sistema binario
A	- 6	1 1111010
B	-13	1 1110011
S = A + B	-19	1 1 1101101 (*)

- *Caso 3:*

	Sistema decimal	Acarreo - Sistema binario
A	- 6	1 1111010
B	+13	0 0001101
S = A + B	+ 7	1 0 0000111 (*)

- *Caso 4:*

	Sistema decimal	Acarreo - Sistema binario
A	+ 6	0 0000110
B	-13	1 1110011
S = A + B	- 7	0 1 1111001

(*) En estos casos de acuerdo al análisis previo, se ignora el bit 2^n.

5.1.2 Implementación de la suma en complemento a 2.

En un sistema digital esta operación deberá implementarse en hardware para poder incorporarla de forma que interactúe con el resto del circuito. Con esta finalidad, existen bloques MSI (Mediana Esccla de Integración) para implementar la operación de suma binaria[1]. El esquema lógico con el que se los representa es el siguiente:

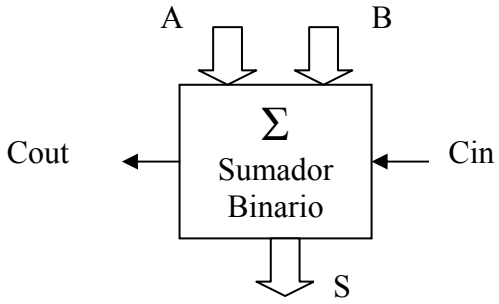


Figura 4: Sumador Binario

Estos bloques están disponibles con un número estándar de bits, en general múltiplos de 2, tanto para las entradas como para las salidas. Las entradas y salidas de acarreo se utilizan para la conexión en cascada de estos bloques, lo que permite obtener sumadores con mayor número de bits. Al implementar la suma en complemento a 2 el acarreo final no se tiene en cuenta de acuerdo con el análisis ya realizado. El resultado entregado por el sumador resultará correcto a no ser que se presente una situación de overflow. Si se trabaja con números negativos, se ingresarán al bloque ya complementados.

La implementación detallada de estos bloques en base a componentes combinacionales (compuertas de distinto tipo), así como la justificación de la misma, escapa al alcance de este apunte y puede ser consultada por los alumnos en la bibliografía de la materia.

5.1.3 Implementación de la suma en BCD.

En el código BCD los dígitos decimales están codificados en binario, y por lo tanto es lógico pensar que la suma puede implementarse utilizando un sumador binario. Por otro lado, la suma de dos dígitos BCD, considerando un posible acarreo desde un dígito menos significativo, no puede ser mayor a 9 + 9 +1 = 19, puesto que ninguno de los dígitos BCD puede superar el valor 9. Los resultados correctos BCD para la suma de dos dígitos y los valores binarios obtenidos a partir del uso de un sumador binario se muestran en la tabla 2.

A partir del análisis de dicha tabla surge que se debe realizar una corrección al resultado binario si supera el valor 9. Esta corrección, como puede deducirse a partir de los valores mostrados, consiste en la adición de 6 para todos los resultados mayores a 9 (sombreados en gris en la tabla). La suma de 6 al resultado, (en binario 0110), logra la representación BCD correcta y produce el acarreo de salida necesario. Desde ya, los módulos MSI que abordan esta operación realizan la corrección necesaria en forma interna, detectando los resultados binarios superiores a 9 y sumando 6 si corresponde. Si los números a sumar son negativos, se introducen al sumador ya complementados, generalmente en complemento a la base (complemento a 10), que es el más usual.

[1] Podemos mencionar el sumador binario de 4 bits 74x283.

Suma Binaria K Z_8 Z_4 Z_2 Z_1	Suma BCD C S_8 S_4 S_2 S_1	Valor decimal
0 0 0 0 0	0 0 0 0 0	0
0 0 0 0 1	0 0 0 0 1	1
0 0 0 1 0	0 0 0 1 0	2
0 0 0 1 1	0 0 0 1 1	3
0 0 1 0 0	0 0 1 0 0	4
0 0 1 0 1	0 0 1 0 1	5
0 0 1 1 0	0 0 1 1 0	6
0 0 1 1 1	0 0 1 1 1	7
0 1 0 0 0	0 1 0 0 0	8
0 1 0 0 1	0 1 0 0 1	9
0 1 0 1 0	1 0 0 0 0	10
0 1 0 1 1	1 0 0 0 1	11
0 1 1 0 0	1 0 0 1 0	12
0 1 1 0 1	1 0 0 1 1	13
0 1 1 1 0	1 0 1 0 0	14
0 1 1 1 1	1 0 1 0 1	15
1 0 0 0 0	1 0 1 1 0	16
1 0 0 0 1	1 0 1 1 1	17
1 0 0 1 0	1 1 0 0 0	18
1 0 0 1 1	1 1 0 0 1	19

Tabla 2: Suma BCD de dos dígitos

Veamos entonces un par de ejemplos. En los mismos se trabajará con complemento a 10 y se representará el signo como un dígito BCD: 0 para el signo positivo y 9 para el signo negativo:

- *Ejemplo 1:*

S = A + B, dónde A = (+ 184) y B = (+ 576).

	Signo	Centena	Decena	Unidad	Suma Decimal
Acarreo BCD		1	1		
	0000	0001	1000	0100	184
	0000	0101	0111	0110	+ 576
Suma Binaria	0000	0111	1 0000	1010	
Corrección			0110	0110	
Suma BCD	0000	0111	0110	0000	760

- *Ejemplo 2:*

S = A + B, dónde A = (- 184) y B = (- 576). En este caso se deben obtener los correspondientes complementos a 10 de los números:

A = 0 184, por lo tanto $C_{10}(A) = 9\ 816$

B = 0 576, por lo tanto $C_{10}(B) = 9\ 424$

	Signo	Centena	Decena	Unidad	Suma Decimal
Acarreo BCD	1		1		
	1001	1000	0001	0110	- 184
	1001	0100	0010	0100	- 576
Suma Binaria	1 1011	1100	0100	1 1010	
Corrección	0110	0110		0110	
Suma BCD	1 1001	1 0010	0100	0000	- 760

5.1.4 Suma en complemento a la base menos 1.

Sean dos números A y B en base r. Deseamos obtener S = A + B. Para ello, analizaremos los distintos casos posibles con el fin de obtener conclusiones generales.

- *Caso 1:*

  $A > 0$ y $B > 0$

  $S = A + B$ Suma correcta. El resultado es positivo ya que ambos números son positivos.

- *Caso 2:*

  $A < 0$ y $B < 0$

  $S = (r^n - 1 - A) + (r^n - 1 - B) = r^n - 1 + r^n - 1 - (|A| + |B|)$ Suma correcta si se suma r^n (que es 1) a los dígitos menos significativos para compensar el 1 que está sobrando. El resultado es negativo y queda expresado en el correspondiente complemento.

- *Caso 3:*

  $A < 0$ y $B > 0$ y $|A| < |B|$

  $S = (r^n - 1 - A) + B = r^n - 1 + (|B| - |A|)$ Suma correcta si se suma r^n (que es 1) a los dígitos menos significativos para compensar el 1 que está sobrando. De esta manera el resultado es positivo, por ser mayor el valor absoluto de B.

- *Caso 4:*

  $A > 0$ y $B < 0$ y $|A| < |B|$

  $S = A + (r^n - 1 - B) = r^n - 1 - (|B| - |A|)$ Suma correcta. El número es negativo y queda expresado en el complemento correspondiente. No hay acarreo desde el bit r^n, pues se le resta 1 y la diferencia positiva de los valores absolutos.

Cuando sumamos números positivos y negativos utilizando complemento a r – 1, se obtendrá el resultado correcto siempre, si se suma r^n a los dígitos menos significativos.

Para este caso queda para el alumno la realización de los ejemplos correspondientes a los casos analizados.

5.1.5 Implementación de la suma en complemento a 1

El complemento a la base menos uno es mucho menos usual que el complemento a la base. Sin embargo, si se desea implementar la suma en complemento a 1 se pueden utilizar los bloques MSI de sumadores binarios ya descriptos, tomando la precaución de realizar la correspondiente realimentación de los acarreos. Esto se muestra en la figura 5.

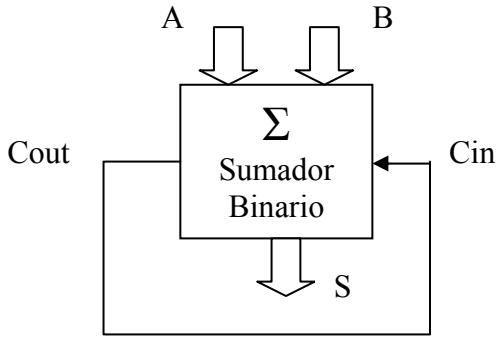


Figura 5: Implementación de un sumador que trabaja con complemento a 1

5.2 Resta aritmética.

Siempre podemos pensar una resta de dos números en base r de la siguiente forma:

R = A – B = A + (-B)

O sea, al minuendo le sumamos el complemento del sustraendo. Desde este punto de vista, podemos utilizar el análisis anterior y extender las conclusiones obtenidas para ambos complementos.

Respecto a la implementación hardware de esta operación, se realiza en base a la arquitectura de los bloques de suma binaria. Existen bloques MSI sumadores restadores, que disponen de entradas de selección de la operación a realizar.

5.3 Sobreflujo (Overflow).

El sobreflujo ocurre cuando al sumar dos números de n bits, el resultado ocupa n + 1bits.

En los sistemas digitales el sobreflujo es un problema porque el tamaño de los registros es finito. Por este motivo es usual que se realice una detección del mismo y la situación se informe al usuario.

Al trabajar con números positivos y negativos podemos afirmar que el sobreflujo puede presentarse sólo si los números que sumamos son ambos positivos o ambos negativos. O sea, si los signos de los sumandos son iguales y el signo del resultado es diferente estamos frente a una situación de sobreflujo. Sin embargo, analizando los siguientes ejemplos, puede encontrarse una detección aún más sencilla, cuando trabajamos con el complemento binario más usual (complemento a 2).

Ejemplo 1: Ambos números son positivos

Acarreos		0 1
A	+ 70	0 1000110
B	+ 80	0 1010000
S = A + B	+ 150	1 0010110

Ejemplo 2: Ambos números son negativos

Acarreos		1 0
A	- 70	1 0111010
B	- 80	1 0110000
S = A + B	- 150	0 1101010

En ambos casos el resultado es incorrecto por haberse presentado una situación de sobreflujo. Lo interesante es notar que también en ambos casos, resultan distintos los acarreos resultantes hacia el bit de signo y desde el bit de signo. Por lo tanto podemos concluir, que cuando esto ocurre se presenta sobreflujo, y detectarlo mediante una simple compuerta OR exclusiva.

5.4 Multiplicación, división y ALUs

La multiplicación de números binarios se realiza del mismo modo que todos conocemos en decimal con lápiz y papel. El multiplicando se multiplica por cada bit del multiplicador, comenzando por el bit menos significativo. Cada uno de estos productos llamado producto parcial se desplaza sucesivamente a la izquierda (por el peso posicional de cada bit) y finalmente se suman. Por lo tanto, puede inferirse que un módulo multiplicador combinacional puede ser implementado en hardware utilizando compuertas y sumadores binarios. Si se trata de números positivos y negativos, se puede realizar la multiplicación de los valores absolutos con estos módulos combinacionales y luego obtener el signo del resultado a partir de los signos de los multiplicandos (multiplicación en magnitud y signo). En general, si multiplicamos dos números de m y n bits respectivamente, el resultado ocupará m + n bits (si estamos hablando de implementación hardware, usualmente m = n). La arquitectura de estos bloques multiplicadores puede abordarse con distintos enfoques, pero escapa al alcance de este apunte y puede consultarse en la bibliografía de la asignatura.

Respecto de la división binaria, el algoritmo más simple para resolverla se basa también en el método de desplazamiento y resta que todos conocemos. Para números con signo, el signo del resultado puede obtenerse en forma análoga al signo del producto.

Existe además lo que se conoce como Unidad Aritmética y Lógica (ALU), que es un bloque MSI que puede realizar las operaciones aritméticas y lógicas básicas sobre un par de operandos de n bits[2]. Posee un conjunto de entradas para los operandos y también para seleccionar el tipo de operación (aritmética ó lógica) a realizar.

[2] Se puede mencionar la ALU de 4 bits 74x181

6. Representación de números reales.

En este caso es necesaria la representación de una coma o punto "r-ario"(será decimal, binaria etc. de acuerdo al sistema de numeración con que se trabaje). Existen dos fomas de resolverlo, teniendo en cuenta las características de los sistemas con los que trabajaremos:

- ✓ Se considera la coma o punto fijo en cierta posición.
- ✓ Se almacena la posición que ocupa la coma o punto en el número.

6.1 Representación de punto fijo.

Este método considera que el punto o coma está siempre en una misma posición. En los sistemas digitales es común utilizar registros para almacenar los datos y entonces se adoptan dos posiciones posibles:

- ✓ En el extremo izquierdo con lo cual se convierte al número en una fracción.
- ✓ En el extremo derecho con lo cual se convierte al número en un entero.

En ninguno de los dos casos el punto o coma existe en realidad, pero su presencia se supone porque el número almacenado se trata como una fracción o un entero. Desde este punto de vista las representaciones analizadas hasta ahora para enteros se consideran de punto fijo.

6.2 Representación de punto flotante.

La aproximación anterior tiene limitaciones. Los números muy grandes o las fracciones muy pequeñas no pueden representarse. Además en la división de dos números grandes puede perderse la parte fraccionaria del cociente. Es sabido que para números decimales esta limitación puede superarse con la notación científica, que permite representar números muy grandes y muy pequeños utilizando pocos dígitos. Esta técnica puede aplicarse también con números representados en otras bases y es así que surge la representación de punto flotante. Esta representación de un número consta de dos partes. La primera parte representa un número de punto fijo con signo llamado mantisa. La segunda parte representa la posición del punto o coma r-aria y se llama exponente. De esta forma el punto flotante se interpreta como la representación de un número de base r de la siguiente manera:

$$N = m \times r^e$$

En la fórmula anterior, m es la mantisa y e el exponente, que son los que se representan en forma física en registros (incluyendo sus signos). Los sistemas que utilizan este tipo de representación asumen siempre la base del número y la posición en el registro del punto fijo de la mantisa.

Ejemplo 1: Sistema decimal.

Número decimal	Mantisa	Exponente
+ 5123,321	+ 0,5123321	+ 04

Ejemplo 2: Sistema binario

Número binario	Mantisa	Exponente
+ 101,011	0 1010110	0 00011

En el segundo ejemplo, un número binario se representa con una mantisa fraccionaria de 8 bits (incluyendo el signo), y un exponente con signo de 6 bits. El punto o coma fijo de la fracción se encuentra inmediatamente después del bit de signo, pero no está representada en el registro. Por lo tanto:

$$N = m \times r^e = +(,1010110) \times 2^{+3}$$

6.2.1 Normalización.

Un número con punto flotante está normalizado si el dígito más significativo de la mantisa es distinto de cero.

Ejemplo:

Número binario	Número sin normalizar		Número normalizado	
	Mantisa	Exponente	Mantisa	Exponente
+ 0011,011	0 0011011	0 00100	0 1101100	0 0010

En este ejemplo, se asume que el número normalizado tiene una mantisa fraccionaria de 7 bits y que el punto o coma binaria se encuentra a la derecha del bit de signo. Al realizar la normalización se deberá tener en cuenta la corrección del exponente.

Los números normalizados proporcionan la máxima precisión posible para los números con punto flotante. Esto quiere decir que permiten la máxima representación de dígitos significativos para la cantidad de bits estipulada. Por lo tanto si bien las operaciones aritméticas con punto flotante son más complicadas que las de punto fijo, esta representación resulta imprescindible por los problemas relacionados con las escalas cuando se requiere una precisión aceptable en los cálculos.

6.3 Códigos de detección de error en sistemas digitales.

Un error en un sistema digital es la alteración de los datos almacendados. Este tipo de errores se deben a fallas físicas que pueden ser temporales o permanentes. Además, la información binaria que se transmite mediante algún medio de comunicación, está sujeta a ruido externo que puede cambiar los bits de 1 a 0 y viceversa corrompiendo los datos que se transmiten. Los efectos de estos errores sobre los datos se predicen utilizando lo que se conoce como “modelos de error”. El modelo de error más sencillo es el modelo de error

independiente que supone que sólo un bit de la tira es afectado. La detección de este tipo de error es por supuesto más sencillo y barato que si pensamos en un modelo de errores múltiples. Por supuesto, los errores múltiples se presentan en los sistemas digitales, pero su apareción se considera menos probable que la de los errores simples.

Un código de detección de error es un código capaz de detectar errores digitales.

Un sistema que utiliza un código de detección de errores, genera transmite y almacena solamente palabras del código. Por lo tanto los errores pueden ser detectados de una forma simple: Si la tira de bits corresponde a una palabra del código es correcta, en caso contrario ha sido afectada por un error.

Los errores se detectan pero no se corrigen. En general se suele analizar la frecuencia de los errores. Si ocurren aleatoriamente, la información errónea se transmite de nuevo, si en cambio ocurren con demasiada frecuencia, se verifica el funcionamiento del sistema porque esto indica la posible presencia de una falla.

En códigos de detección de errores simples se necesitan n + 1 bits para construir las 2^n palabras del código. Los primeros n bits son los bits de información y el bit restante se utiliza para implementar el código. Existen códigos más complejos (y costosos) que sirven para detectar y corregir errores múltiples en los datos, pero en este apunte nos limitaremos a analizar dos códigos de detección de errores simples.

6.3.1 Bit de paridad.

Los dos códigos de detección de errores simple más usuales son los que utilizan un bit adicional de paridad. Consisten en agregar un bit al mensaje, de forma de lograr que la cantidad total de unos resulte par o impar de acuerdo al código utilizado. En la Tabla 3 se muestra un mensaje de tres bits con los dos bits de paridad posible.

Mensaje	P (impar)	P (par)
000	1	0
001	0	1
010	0	1
011	1	0
100	0	1
101	1	0
110	1	0
111	0	1

Tabla 3: Generación de un bit de paridad.

El bit de paridad se agrega al mensaje y la cantidad de unos que se contabiliza luego de la recepción incluye a todos los bits del mensaje y al bit de paridad. El bit de paridad par como puede verse en la tabla es el complemento del de paridad impar.

En el punto donde se envía la información, se utiliza un generador de paridad, que obtiene el bit de paridad correspondiente. Cuando se recepciona el mensaje, todos los bits recibidos se aplican al comprobador de paridad que verifica la paridad para determinar si se ajusta a lo convenido. El mayor problema de estos códigos es que sólo son capaces de detectar un número impar de errores, ya que los cambios pares no pueden ser observados.

Las redes generadoras y detectoras de paridad, se pueden implementar utilizando puertar OR exclusivas. Queda para el alumno plantear y justificar algún circuito posible.

7. Otros códigos binarios

Además de los códigos para representar datos y los códigos de detección de error que ya hemos mencionado, es habitual en el diseño de sistemas digitales utilizar una tira de bits para controlar una acción , indicar una condición o representar un estado del sistema. Si existen n diferentes acciones, condiciones o estados, podemos representarlos utilizando tiras de b bits, dónde b es el entero más pequeño que cumple con $2^b >= n$. Estas tiras de bits constituyen un código que representa los elementos que necesitamos procesar, de una manera comprensible para el sistema.

Existen muchos códigos que se han desarrollado con el objetivo de optimizar algún parámetro de diseño en particular, y que se utilizan con diversos fines. En este apunte mencionaremos el código de 7 segmentos muy usado cuando se necesita presentar resultados en un display.

7.1 Código 7 segmentos.

Es habitual hacer visible la lectura de un instrumento o dispositivo mediante una representación visual en un sistema numérico y en una base determinada. Existens varias maneras de hacerlo, y una de ellas es mediante un indicador numérico de 7 segmentos, el cual consiste en una matriz de diodos LED con la siguiente configuración:

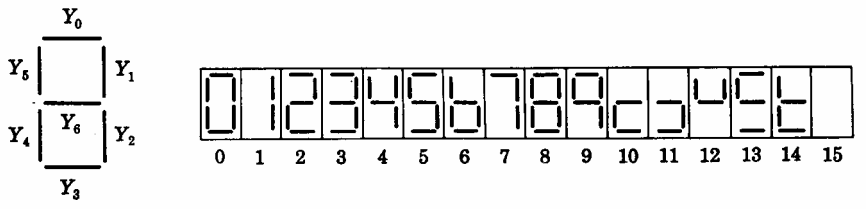


Figura 6: Configuración de un dígito de 7 segmentos e imágenes presentadas.

En la Figura 6 se muestran también las imágenes presentadas para un dígito decimal. Las primeras 10 son válidas y las imágenes del 10 al 15 son símbolos usados unicamente para identificar condiciones no válidas en el código BCD.

Según el conexionado de los diodos se diferencian estas matrices en ánodo común y cátodo común. Además en el mercado se consiguen displays de distinta cantidad de dígitos (1,1 ½, 2, 2 ½ y 3 dígitos), con el punto decimal a la derecha o a la izquierda , con signos , en distintos colores y tamaños.

De acuerdo al circuito que controle el display se pueden tener las diferentes representaciones como por ejemplo BCD (0 a 9), hexadecimal (0 a F), binaria (0, 1). Existen en el mercado una variedad de circuitos integrados que para las diferentes familias lógicas permiten realizar las distintas configuraciones. A continuación se presenta una hoja de datos a modo de ejemplo en la Figura 7.

Numeric LED Display Outlines (cont'd)

Fig. P37

	ECG3074 - Red	ECG3075 - Red
1.	E – Cathode - 1	E – Anode - 1
2.	D – Cathode - 1	D – Anode - 1
3.	C – Cathode - 1	C – Anode - 1
4.	Dec Pt Cathode - 1	Dec Pt Anode - 1
5.	E – Cathode - 2	E – Anode - 2
6.	D – Cathode - 2	D – Anode - 2
7.	G – Cathode - 2	G – Anode - 2
8.	C – Cathode - 2	C – Anode - 2
9.	Dec Pt Cathode - 2	Dec Pt Anode - 2
10.	B – Cathode - 2	B – Anode - 2
11.	A – Cathode - 2	A – Anode - 2
12.	F – Cathode - 2	F – Anode - 2
13.	Common Anode - 2	Common Cathode - 2
14.	Common Anode - 1	Common Cathode - 1
15.	B – Cathode - 1	B – Anode - 1
16.	A – Cathode - 1	A – Anode - 1
17.	G – Cathode - 1	G – Anode - 1
18.	F – Cathode - 1	F – Anode - 1

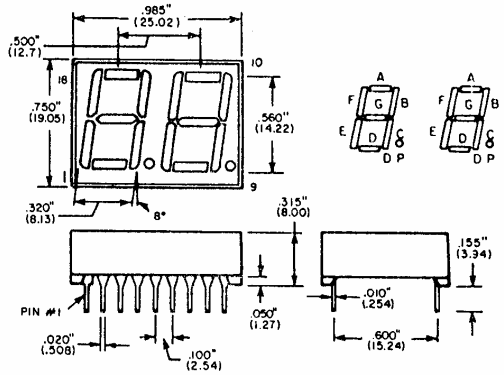


Fig. P38

	ECG3076 - Red	ECG3077 - Red
1.	C – Cathode - 1	C – Anode - 1
2.	D – Cathode - 1	D – Anode - 1
3.	B – Cathode - 1	B – Anode - 1
4.	Dec Pt Cathode - 1	Dec Pt Anode - 1
5.	E – Cathode - 2	E – Anode - 2
6.	D – Cathode - 2	D – Anode - 2
7.	G – Cathode - 2	G – Anode - 2
8.	C – Cathode - 2	C – Anode - 2
9.	Dec Pt Cathode - 2	Dec Pt Anode - 2
10.	B – Cathode - 2	B – Anode -2
11.	A – Cathode - 2	A – Anode - 2
12.	F – Cathode - 2	F – Anode - 2
13.	Common Anode - 2	Common Cathode - 2
14.	Common Anode - 1	Common Cathode - 1
15.	A – Cathode - 1	A – Anode - 1
16.	No Connection	No Connection
17.	No Connection	No Connection
18.	No Connection	No Connection

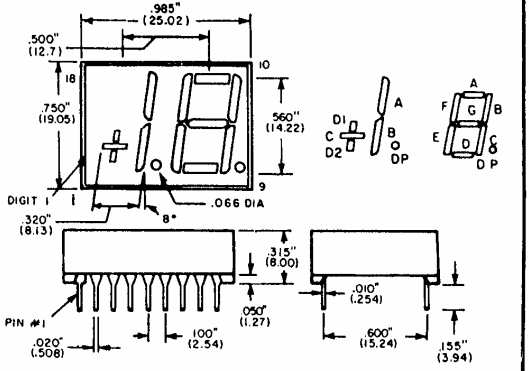


Fig. P39

	ECG3078 - Red	ECG3079 - Red
1.	E – Cathode	E – Anode
2.	D – Cathode	D – Anode
3.	Common Anode	Common Cathode
4.	C – Cathode	C – Anode
5.	Dec Pt Cathode	Dec Pt Anode
6.	B – Cathode	B – Anode
7.	A – Cathode	A – Anode
8.	Common Anode	Common Cathode
9.	F – Cathode	F – Anode
10.	G – Cathode	G – Anode

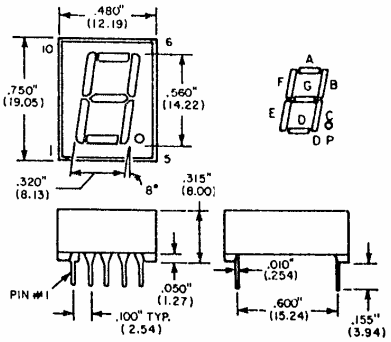


Fig. P40

ECG3080 - Red

1.	No Pin	10.	Decimal Point
2.	Segment A	11.	Segment D
3.	Segment F	12.	Common Anode
4.	Common Anode	13.	Segment C
5.	Segment E	14.	Segment G
6.	Common Anode	15.	Segment B
7.	No Connection	16.	No Pin
8.	No Pin	17.	Common Anode
9.	No Pin	18.	No Pin

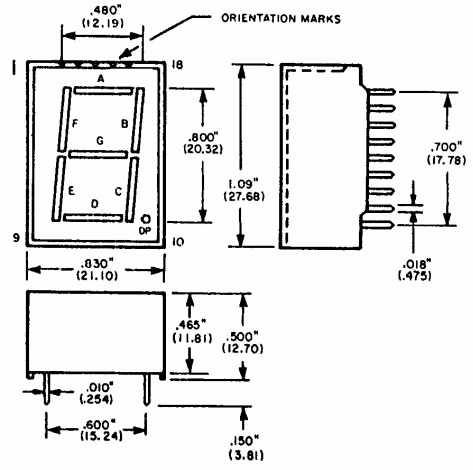


Figura 7: Hoja de datos.

Finalmente en la Figura 8 se presenta la tabla de conversión de dígitos decimales a código 7 segmentos. De la misma puede obtenerse utilizando álgebra de Boole y técnicas de simplificación la expresión del comportamiento deseado para cada uno de los siete segmentos del dígito. Los segmentos están identificados como se indica en la Figura 6 y al asumir el estado "0" se encuentran activos.

Entradas en código binario decimal				Información decodificada	Salidas en código para el indicador de siete segmentos						
$X_3 = D$	$X_2 = C$	$X_1 = B$	$X_0 = A$	W_n	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	W_0	1	0	0	0	0	0	0
0	0	0	1	W_1	1	1	1	1	0	0	1
0	0	1	0	W_2	0	1	0	0	1	0	0
0	0	1	1	W_3	0	1	1	0	0	0	0
0	1	0	0	W_4	0	0	1	1	0	0	1
0	1	0	1	W_5	0	0	1	0	0	1	0
0	1	1	0	W_6	0	0	0	0	0	1	1
0	1	1	1	W_7	1	1	1	1	0	0	0
1	0	0	0	W_8	0	0	0	0	0	0	0
1	0	0	1	W_9	0	0	1	1	0	0	0
1	0	1	0	W_{10}	0	1	0	0	1	1	1
1	0	1	1	W_{11}	0	1	1	0	0	1	1
1	1	0	0	W_{12}	0	0	1	1	1	0	1
1	1	0	1	W_{13}	0	0	1	0	1	1	0
1	1	1	0	W_{14}	0	0	0	0	1	1	1
1	1	1	1	W_{15}	1	1	1	1	1	1	1

Figura 8: Conversión de un dígito decimal a código 7 segmentos.

8. Representación computacional de los datos

Hasta aquí, hemos tratado los datos numéricos desde el punto de vista matemático, y hemos analizado algunos bloques MSI capaces de implementar la operatoria propuesta. Ahora nos abocaremos a estudiar cómo se representan computacionalmente estos datos. En ese ámbito son usuales los siguientes tipos numéricos:

- ✓ Enteros o en punto fijo.
- ✓ En punto flotante.
- ✓ Decimales.
- ✓ Tipo carácter (código ASCII).

Las operaciones internas de una computadora son en esencia binarias, sin embargo, es normal la utilización de otros tipos de datos en la vida diaria. Por lo tanto se realizan conversiones entre los distintos tipos, ya sea para obtener un resultado, como para presentarlo al usuario. Además, cuando trabajamos con computadoras, la información debe ser representada internamente de forma de poder ser procesada y está limitada tanto en su *magnitud* como en su *precisión*. Analizaremos entonces estas cuestiones, para las diferentes representaciones disponibles en una computadora.

8.1 Representación computacional de enteros.

Podemos mencionar tres formas posibles de representación:

- ✓ ***Byt:*** Los datos de tipo Byte son enteros con y sin signo. La diferencia radica en que si se trata de un entero con signo el bit más significativo de la tira es el bit de signo. Por lo tanto en el caso de enteros sin signo el rango representable va desde 0 a 255, mientras que en el caso de enteros con signo de –128 a +127. En el caso de trabajar con signo se representa los enteros negativos utilizando el complemento a dos.
- ✓ ***Word:*** Una palabra (16 bits) se forma con dos bytes. En los microprocesadores de la familia Intel el byte menos significativo se almacena en la localidad de memoria más baja, y el más significativo en la más alta. Este formato se conoce como little endian. Las palabras también pueden ser con y sin signo. Nuevamente la diferencia radica en el bit más significativo. En el caso de trabajarse con signo, los enteros negativos se representan en complemento a dos.
- ✓ ***Double Word:*** La información de palabra doble requiere 4 bytes de memoria pues ocupa 32 bits. También se almacena en memoria usando el formato little endian (siempre hablando de la familia Intel). También puede ser con o sin signo y los enteros negativos se representan usando complemento a dos.

También es posible almacenar en memoria enteros de cualquier longitud. Las tres representaciones anteriores son estándar, pero esto no quiere decir que si necesitamos trabajar con enteros más grandes no podamos hacerlo.

8.1.1 Conversión entre longitudes diferentes.

Es a veces deseable tomar un entero de n bits y almacenarlo en m bits, siendo m > n. Cuando trabajamos con enteros en complemento a 2 para no alterar el número original es necesario trasladar el bit de signo a la posición más a la izquierda y completar los bits intermedios del registro copiando el bit de signo. O sea si el número es positivo se completa con ceros y si es negativo con unos. El lenguaje de programación assembler proporciona instrucciones para realizar la conversión en forma apropiada (siempre desde una longitud menor a una mayor).

Ejemplo:

```
+ 18 =              00010010  (complemento a dos, 8 bits).
+ 18 = 0000000000010010  (complemento a dos, 16 bits).
- 18 =              11101110  (complemento a dos, 8 bits).
- 18 = 1111111111101110  (complemento a dos, 16 bits).
```

8.2 Representación computacional de decimales.

Ya se ha mencionado que las computadoras operan en binario. Las aplicaciones que requieren manipular datos en BCD son en general las terminales de puntos de venta y aquellos sistemas que realizan un mínimo de cálculos sencillos. Si se requiere una operatoria más compleja es muy raro que se recurra a aritmética BCD, que es más complicada para implementar y menos eficiente.

Si el programa define sus propios datos binarios, el problema está resuelto, pero en ciertas aplicaciones los datos se introducen por teclado como caracteres ASCII en base 10. Del mismo modo, al presentar resultados por pantalla, los mismos están en formato ASCII.

Por lo tanto, las computadoras deben ser capaces de manipular este tipo de datos y realizar las conversiones necesarias. Es así que el lenguaje assembler proporciona un conjunto de instrucciones capaces de abordar estas tareas. Es importante destacar que el procesador realiza aritmética en valores ASCII y BCD de un dígito a la vez. Es responsabilidad del programador utilizar las instrucciones apropiadas para realizar las conversiones necesarias y considerar los acarreos desde los dígitos BCD menos significativos a los más significativos. Esto se analizará cuando se estudie el lenguaje.

Se pueden mencionar tres formas de representación de este tipo de datos:

- ✓ ***BCD desempaquetado:*** Dígito BCD representado en un byte. El dígito ocupa los cuatro bits menos significativos, mientras que los cuatro más significativos son cero. Las operaciones BCD de multiplicación y división requieren que los operandos sean convertidos a este formato.
- ✓ ***BCD empaquetado:*** Representación de 2 dígitos BCD en un byte. Este formato es utilizado por algunas instrucciones del lenguaje assembler que se utilizan en suma y resta BCD.
- ✓ ***Modo carácter:*** Representación de 2 dígitos BCD en dos bytes utilizando el código ASCII. Estos datos provienen del teclado.

Nº	BCD Empaquetado	BCD Desempaquetado	ASCII
12	0001 0010	0000 0001 0000 0010	0011 0001 0011 0010
623	0000 0110 0010 0011	0000 0110 0000 0010 0000 0011	0011 0110 0011 0010 0011 0011
910	0000 1001 0001 0000	0000 1001 0000 0001 0000 0000	0011 1001 0011 0001 0011 0000

Tabla 4: Datos decimales en distintos formatos

Algunas aplicaciones necesitan manipular números negativos, al introducir los datos se puede incluir el signo menos de forma que el programa verifique el signo al realizar las conversiones. Al finalizar los cálculos, si es necesario se puede insertar un signo menos en un campo ASCII para presentar los resultados.

8.3 Representación computacional de números en punto flotante.

Para poder utilizar los números en punto flotante en sistemas de cómputo, sus valores característicos de exponente y mantisa deben ser almacenados en registros. Por razones de eficiencia del sistema se trabaja con las bases 2, 4, 8 o 16.

Las principales características de la representación son:

- ✓ La mantisa normalizada, se representa en la convención de magnitud y signo, asumiendo que la misma es una fracción (el punto o coma está implícito en la representación).
- ✓ Al exponente se le asigna un tipo de representación sesgada (biased) o polarizada. En la misma, se le suma un valor fijo o sesgo para obtener el valor final a representar, que resulta siempre positivo.
- ✓ La base del sistema con que se trabaja se asume, o sea no está representada.

Para ejemplificar, asumamos que disponemos de 24 bits para representar un número binario, distribuidos de la siguiente forma:

1 bit de signo	7 bits para el exponente	16 bits para la mantisa

Con 7 bits para el exponente podemos representar 128 valores distintos [0 , 127]. Si elegimos un sesgo de 64, con esos valores representamos en realidad [-64, 63].

La ventaja de utilizar exponentes de este tipo, es que al ser siempre positivos, resulta más sencillo comparar sus magnitudes durante las operaciones de alineamiento de las mantisas. En la Figura 9 pueden verse algunos ejemplos en base 2 y 16.

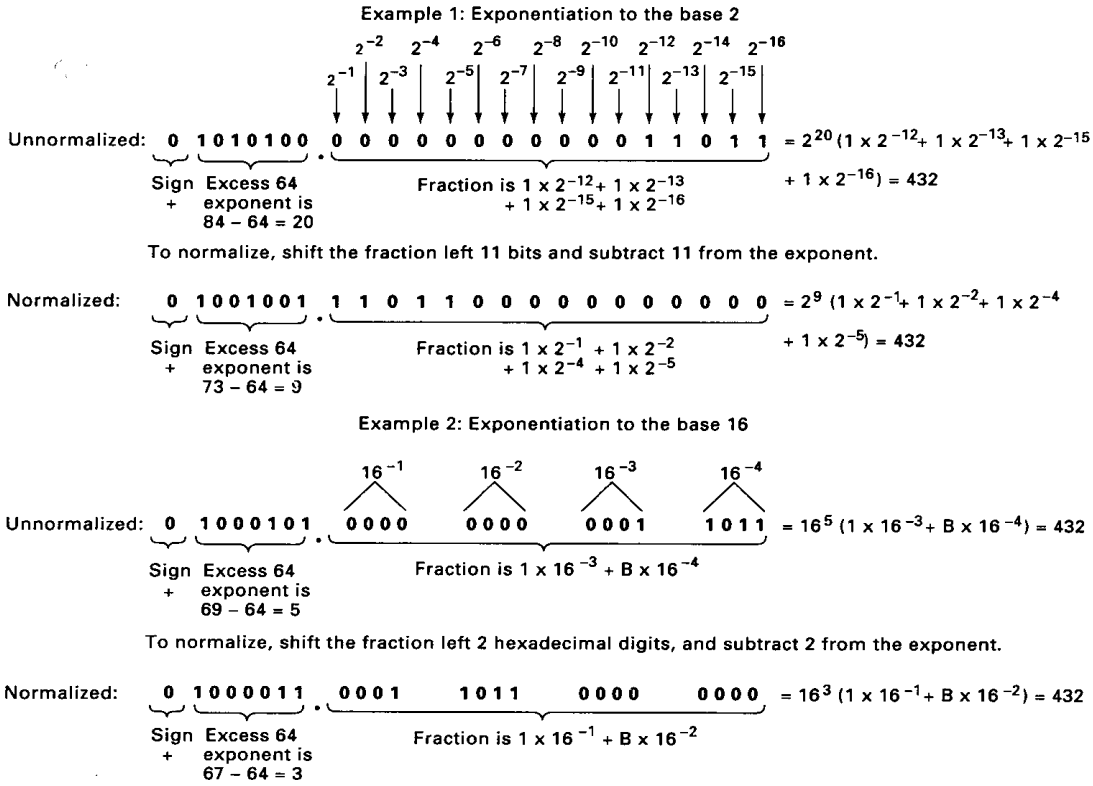


Figura 9: Ejemplos de números en punto flotante.

Si retomamos la representación de un número en punto flotante de 24 bits (3 bytes), podemos destacar algunos aspectos importantes:

- ✓ Se pueden representar 2^{24} números distintos.
- ✓ Se pueden representar los siguientes rangos:
  - ✓ Números negativos entre $-(1-2^{-16})\,2^{63}$ y $-0{,}5\,2^{-64}$.
  - ✓ Números positivos entre $0{,}5\,2^{-64}$ y $(1-2^{-16})\,2^{63}$.

En la recta numérica real hay entonces 5 regiones excluidas de los rangos representables, como puede apreciarse en la Figura 10:

- ✓ Los números negativos menores que – (1- 2^{-16}) 2^{63}, región que se denomina *desbordamiento negativo.*
- ✓ Los números negativos mayores que – 0,5 2^{-64}, denominada *desbordamiento a cero negativo.*
- ✓ El cero.
- ✓ Los números positivos menores que 0,5 2^{-64} , denominada *desbordamiento a cero positivo*.Los números positivos mayores que (1- 2^{-16}) 2^{63} , denominada *desbordamiento positivo*.

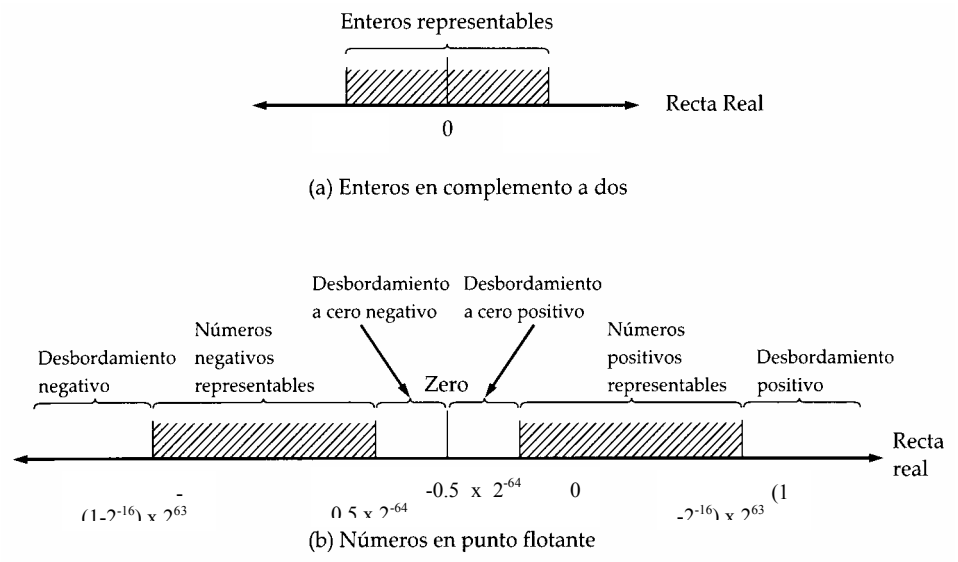


Figura 10: Números representables en formatos típicos de 24 bits.

El desbordamiento a cero ocurre cuando una magnitud es demasiado pequeña como para ser representada. No se trata de un problema serio, porque el resultado puede generalmente aproximarse satisfactoriamente a cero.

Un desbordamieto u "overflow" ocurre cuando una operación aritmética da un número cuyo exponente es mayor que 63, por lo cual no puede ser representado.

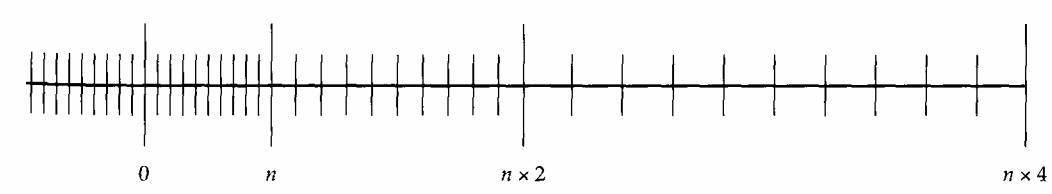


Figura 11: Densidad de los números en punto flotante.

Es importante destacar que si utilizamos 24 bits (3 bytes) para representar un número, la cantidad de valores distintos que pueden representarse son 2^{24}. Lo que se hace con la notación en punto flotante es distribuirlos en dos intervalos uno positivo y otro negativo. Además, al contrario de lo que ocurre con los números en punto fijo cuyos valores están equiespaciados, los números representados en punto flotante no se distribuyen por igual en la recta numérica. En la Figura 11 puede verse que los valores están más

próximos cerca del origen y más separados a medida que nos alejamos de él. Por lo tanto existe un compromiso entre rango y precisión. Si se incrementa el número de bits del exponente crece el rango de números representable, pero como la cantidad de números representables sigue siendo 2^{24}, se espacían entre sí y se pierde precisión. La única forma de incrementar tanto el rango como la precisión es utilizar más bits.

8.3.1 Standard IEEE para números en punto flotante.

Hasta los 80 cada fabricante de computadoras tenía su propio formato de punto flotante, no sólo distintos entre sí sino en algunos casos con errores desde el punto de vista matemático. Para corregir esta situación la IEEE organizó un comité con el fin de lograr un standard para la representación. Esto permitiría portabilidad y a la vez proveería a los fabricantes de un modelo correcto. El resultado fue el standard 754 (IEEE 1985), que define tres formatos:

- ✓ Precisión simple.
- ✓ Precisión doble.
- ✓ Precisión extendida.

Los formatos pueden verse en la Figura 12 y sus características en la Tabla 5.

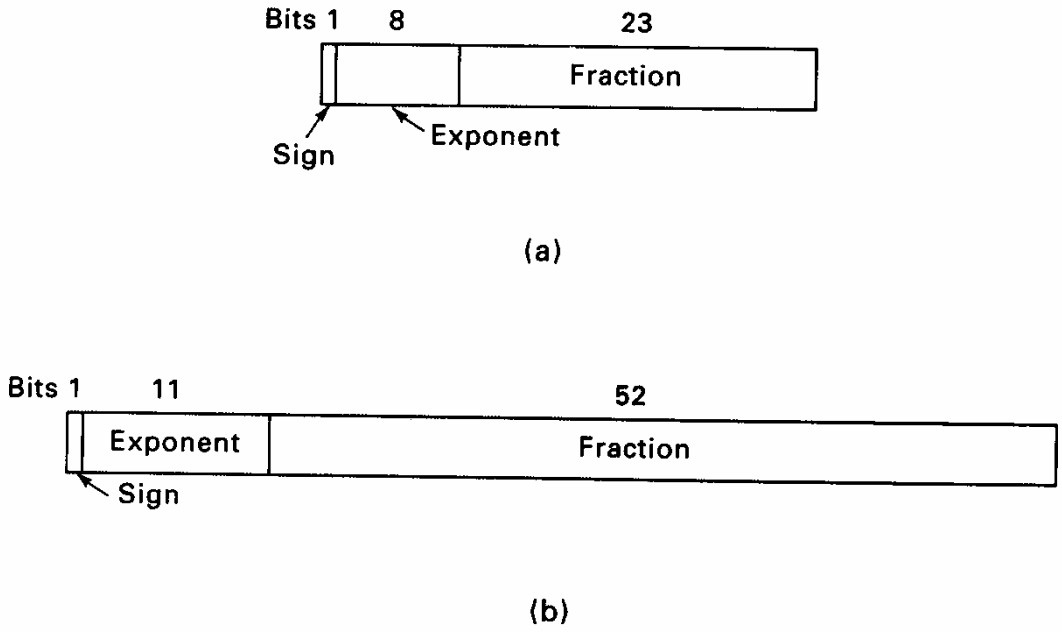


Figura 12: Formatos del standard 754 para precisión simple (a) y doble (b).

Ambos formatos comienzan con el bit de signo. Luego se representa el exponente utilizando un sesgo de 127 para precisión simple y de 1023 para precisión doble. Los valores mínimo (0) y máximos (255 y 2047) no se utilizan para números normalizados, sino que tienen usos especiales que se analizarán luego. Finalmente se representa la mantisa fraccionariade 23 y 52 bits respectivamente.

Item	Single precision	Double precision
Bits in sign	1	1
Bits in exponent	8	11
Bits in fraction	23	52
Bits, total	32	64
Exponent system	Excess 127	Excess 1023
Exponent range	−126 to +127	−1022 to +1023
Smallest, normalized	2^{-126}	2^{-1022}
Largest, normalized	approx. 2^{+128}	approx. 2^{+1024}
Decimal range	approx. 10^{-38} to 10^{+38}	approx. 10^{-308} to 10^{+308}
Smallest, denormalized	approx. 10^{-45}	approx. 10^{-324}

Tabla 5: Caracterísctcias de los números en punto flotante del standard 754.

Una fracción normalizada binaria, comienza siempre con un punto binario, seguido por un bit igual a 1 y luego el resto de los bits de la representación. Entonces, ese bit siempre igual a 1 no necesita ser almacenado ya que puede asumirse su presencia. Por lo tanto, el standard define esta fracción de una forma distinta a la usual, y para evitar confusiones la llama *significante*. Este *significante* consta de un 1 implícito seguido del punto o coma binaria y luego por 23 o 52 bits arbitrarios. Si todos los bits son cero, el *significante* será 1.0, si todos ellos son unos, resultará ligeramente inferior a 2.

El IEEE 754 va más allá de la simple definición de formatos, detallando cuestiones prácticas específicas y procedimientos para que la aritmética en punto flotante produzca resultados uniformes y predecibles, independientemente de la plataforma utilizada. Con este fin, el standard 754 agrega a los números normalizados descriptos hasta el momento cuatro tipos numéricos que se describen en la Figura 13.

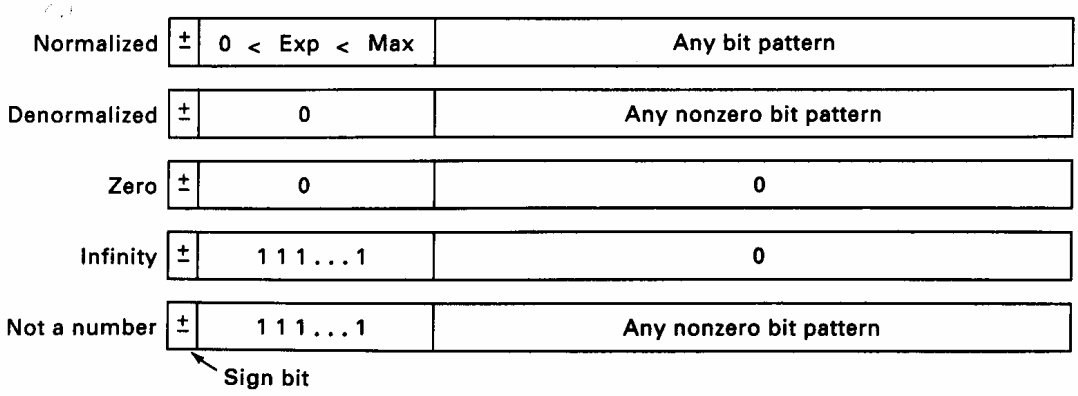


Figura 13: Tipos numéricos del standard IEEE 754.

Números denormalizados:

El menor número normalizado en simple precisión es 1.0×2^{-126}. Antes de la definición del standard si se presentaba un problema de underflow se debía aproximar directamente a cero. Los números denormalizados del estándar proveen una mejor aproximación al problema. Este tipo de números tienen un exponente cero (no permitido para los números normalizados, que corresponde una vez restado el sesgo a –127 o –1023 según el formato) y una fracción de 23 o 52 bits (según el formato). El uno implícito del significante es cero para ellos, por lo que el exponente resultante es de -126 ó -1022 de acuerdo con el formato. Por lo tanto, permiten representar números mucho más pequeños que el tipo normalizado y logran una aproximación a cero más "elegante", conocida como *desbordamiento gradual*.

Infinito:

El overflow es complicado de abordar. Sin embargo el standard 754 provee una representación para $\pm \infty$. Estos números pueden ser utilizados como operandos y cumplen con las reglas matemáticas que tratan las operaciones aritméticas con infinito como casos límite de la aritmética con números reales. Al utilizar los tipos que representan infinito las operaciones producen los resultados conocidos:

- ✓ $N + \infty = + \infty$
- ✓ $N - \infty = - \infty$
- ✓ $+ \infty + \infty = + \infty$
- ✓ $- \infty - \infty = - \infty$

Formato N a N:

Son entidades simbólicas codificadas en punto flotante. Su formato incluye un exponente con todos unos y una parte fraccionaria distinta de cero. El patrón de bits real de dicha parte fraccionaria depende de cada implementación concreta, ya que sus valores pueden servir para distinguir entre condiciones de excepción particulares. Sirven para representar valores de variables no inicializadas y tratamientos de tipo aritmético que no están contemplados en el standard. Ejemplos son: 0 / 0 ; ∞ / ∞; 0 x ∞; etc. Cuando estas situaciones anormales se producen originan una excepción, cuyo tratamiento es altamente dependiente del lenguaje y del sistema operativo utilizado. Por este motivo este formato es considerado por varios autores como la característica más notable del estándar, pues permite que el cálculo continúe aún en condiciones excepcionales, tal como dividir por cero o calcular la raíz cuadrada de un número negativo. En estos casos, el resultado que se obtiene es un N a N (Not a Number). Cuando el argumento de una operación es un N a N el resultado es otro N a N y así la situación se propaga sin necesidad de testeos intermedios ni codificación especial, permitiendo al diseñador del sistema tomar una decisión cuando lo estime conveniente.

8.4 Representación alfanumérica (Código ASCII).

En muchas aplicaciones se hace necesario manipular además de números, letras y caracteres especiales. Para representar información textual es necesario utilizar un código

binario para las letras del alfabeto. Un *conjunto de caracteres alfanuméricos* es un conjunto que incluye los 10 dígitos decimales, las 26 letras del alfabeto y caracteres especiales como $, * , +. Este conjunto contiene entre 32 y 64 elementos (sólo incluye las mayúsculas) o entre 64 y 128 (incluye mayúsculas y minúsculas). En el primer caso se necesitarán 6 bits para realizar la codificación binaria y en el segundo 7.

$b_4\ b_3\ b_2\ b_1$	$b_7\ b_6\ b_5$							
	000	001	010	011	100	101	110	111
0000	NUL	DEL	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	“	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Caracteres de control			
NUL	Nulo	DEL	Escape de enlace de datos
SOH	Comienzo de encabezado	DC1	Control de dispositivo 1
STX	Comienzo de texto	DC2	Control de dispositivo 2
ETX	Fin de texto	DC3	Control de dispositivo 3
EOT	Fin de transmisión	DC4	Control de dispositivo 4
ENQ	Consulta	NAK	Reconocimiento negativo
ACK	Reconocimiento	SYN	Inactivo sincrónico
BEL	Campana	ETB	Fin de bloque de transmisión
BS	Retroceso	CAN	Cancelar
HT	Tabulador horizontal	EM	Fin de medio
LF	Alimentación de línea	SUB	Sustituto
VT	Tabulador vertical	ESC	Escape
FF	Alimentación de forma	FS	Separador de archivo
CR	Retorno de carro	GS	Separador de grupo
SO	Tecla de mayúscula oprimida	RS	Separador de registro
SI	Tecla de mayúsculas sin oprimir	US	Separador de unidad
SP	Espacio	DEL	Borrar

Tabla 6: Código ASCII

Si bien podrían definirse muchos códigos distintos el código alfanumérico estándar es el ASCII (American Standard Code for Information Interchange), que utiliza 7 bits para

codificar 128 caracteres. El mismo se muestra en la Tabla 5. Los 34 caracteres de control que incluye el código se presentan abreviados y con sus denominaciones completas. Se pueden dividir en tres tipos: *de formato*, *separadores de información* y *de control de información*. Los de formato controlan la distribución de la impresión. Incluyen caracteres como retroceso, retorno de carro y tabulación horizontal. Los separadores de información permiten dividir la información en párrafos y páginas e incluyen controles como separador de archivo y separador de registro. Los de control de información son útiles durante la transmisión de datos entre terminales remotas. Algunos ejemplos son inicio de texto y fin de texto. El código ASCII es un código de 7 bits, pero en las computadoras se manipula como unidad el byte que incluye 8 bits. Por lo tanto, lo más común es almacenar un carácter ASCII por byte. El bit extra se utiliza a veces para otros fines dependiendo de la aplicación, por ejemplo, en algunos sistemas su usa para mantener la paridad de los datos.

Muchas aplicaciones Windows utilizan el sistema *Unicode* para almacenar información alfanumérica. *Unicode* almacena cada carácter en 16 bits, donde los códigos de 0000H a 00FFH corresponden al código ASCII estándar, y las combinaciones restantes corresponden a los conjuntos de caracteres especiales de los distintos alfabetos del mundo. Esto permite que el software sea utilizado en cualquier país.

9. Aritmética de computadoras

Las computadoras digitales incluyen en su set de instrucciones algunas que se encargan de resolver las operaciones aritméticas básicas, para producir los resultados necesarios en la solución de problemas. Estas instrucciones ejecutan cálculos aritméticos y son las responsables de la mayor parte de la actividad para el procesamiento de datos. A partir de las cuatro operaciones básicas es posible formular otras y resolver problemas de distinta naturaleza. Como los datos pueden ser de distintas características y pueden representarse de distinto modo en una computadora, existen instrucciones diferentes para trabajar con ellos.

9.1 Operaciones aritméticas binarias de punto fijo.

Estas operaciones con números enteros pueden utilizar, como se explicó en su momento, dos representaciones distintas: Magnitud-signo y complementos.

Cuando se opera con números en punto fijo, es común utilizar los complementos dada su simplicidad. Sin embargo, al tratar la representación en punto flotante para números con parte entera y fraccionaria, vimos que el significante se representa en magnitud y signo, por lo tanto, el sistema debe ser capaz de abordar esta operatoria.

El lenguaje assembler incluye instrucciones para realizar las cuatro operaciones aritméticas considerando números binarios con y sin signo. Además estos datos pueden ser de distinto tamaño. Estas instrucciones y las consideraciones a realizar al momento de utilizarlas se analizarán al estudiar el lenguaje.

9.2 Operaciones aritméticas ASCII y BCD.

La operatoria decimal de punto fijo se basa en las instrucciones disponibles para trabajar con datos binarios. Sin embargo, teniendo en cuenta las diversas formas de representar los datos decimales en computadoras digitales, el programador debe utilizar una serie de instrucciones de ajuste propias de cada representación para lograr un resultado correcto. No realizaremos aquí un análisis pormenorizado de las mismas, ya que se volverá sobre el tema al momento de estudiar el lenguaje.

9.3 Suma y resta en punto flotante.

En aritmética de punto flotante las operaciones de suma y resta resultan más complejas que las de multiplicación y división. Esto ocurre pues al sumar o restar es necesario realizar un ajuste de los significantes respectivos. Además, luego se debe normalizar el resultado para que quede correctamente representado. Cabe acotar que en la asignatura no trabajaremos con este tipo de datos, por lo tanto sólo se comentarán algunas cuestiones a tener en cuenta cuando se implementan estas operaciones. En general las operaciones de suma y resta implican la realización de los siguientes pasos:

- Comprobar si alguno de los números es cero.
- Ajustar los significantes.
- Sumar o restar los significantes.
- Normalizar el resultado.

El algoritmo que se utiliza para abordar los pasos indicados y resolver las operaciones resulta bastante largo y complicado. Como su tratamiento detallado escapa al alcance de este apunte, nos limitaremos a explicar algunos de los pasos anteriores y a dar ejemplos.

Dado que la suma y la resta son idénticas excepto por el cambio de signo, el proceso comienza cambiando el bit de signo del sustraendo si se trata de una resta. A continuación si alguno de los valores es cero se da como resultado el otro.

La siguiente etapa del algoritmo manipula los números para que los exponentes resulten iguales. Para comprender la necesidad de este paso analicemos la siguiente suma decimal:

$$S = 123\ 10^{0} + 456\ 10^{-2}$$

Puede verse que no podemos sumar los significantes en forma directa. Los dígitos deben ponerse primero en posiciones equivalentes, es decir, el 4 del segundo número debe alinearse con el 3 del primero. Bajo estas condiciones los exponentes serían iguales, que es la condición matemática para que los números se puedan sumar. Así:

$$S = 123\ 10^{0} + 456\ 10^{-2} = 123\ 10^{0} + 4{,}56\ 10^{0} = 127{,}56\ 10^{0}$$

La alineación o ajuste se consigue bien desplazando a la derecha el número más pequeño (incrementando su exponente) o desplazando a la izquierda el más grande. Ya que cualquiera de estas operaciones puede hacer que se pierdan dígitos significativos, es el más pequeño el que se desplaza, con lo que los dígitos que se pierden tienen una importancia relativa menor. El ajuste se consigue repitiendo desplazamientos de un dígito a la derecha del significante e incrementando el exponente hasta que ambos exponentes resulten iguales. Si en el proceso se obtiene un significante cero, se da como resultado el otro número. Por lo tanto cuando dos números tienen exponentes muy diferentes se pierde el menor.

A continuación se suman los dos significantes teniendo en cuenta sus signos. Como los signos pueden ser distintos, el resultado puede ser cero. Existe también la posibilidad de desbordamiento de la suma en un dígito. Si es así, se debe desplazar a la derecha el significante de la suma en un dígito incrementando su exponente. Como resultado puede ocurrir un desbordamiento del exponente, esto se debe avisar y la operación se detendrá.

La siguiente fase normaliza el resultado. La normalización consiste en desplazar a la izquierda los dígitos del significante hasta que el más significativo sea distinto de cero. Cada desplazamiento causa el decremento del exponente, lo que puede ocasionar un

desbordamiento a cero del mismo. Finalmente, el resultado debe redondearse a un número representable en el estándar. El tema del redondeo se analizará más adelante.

9.4 Multiplicación y división en punto flotante.

La multiplicación y la división en punto flotante son procesos mucho más sencillos que la suma y la resta. Esto se debe a que la operatoria se reduce a sumar exponentes (multiplicación), o restarlos (división) y a realizar la multiplicación o división de los significantes. Por supuesto, la tarea incluye verificaciónes de desbordamiento del significante o exponente que llevará a las correcciones necesarias (o aviso de situaciones de error), y a la normalización y redondeo correspondiente para presentar el resultado conforme a la norma.

9.5 Consideraciones sobre precisión: Bits de guarda.

Previo a la realización de una operación en punto flotante, se cargan el exponente y el significante en registros. En el caso del significante, el tamaño del registro es casi siempre mayor que su longitud más el bit implícito. El registro contiene bits adicionales, llamados bits de guarda o de respaldo, que se añaden a la derecha del significante en forma de ceros.

La razón de utilizar estos bits de guarda se ilustra en los ejemplos. En los mismos se trabaja con números en el formato IEEE con un significante de 24 bits, restituyendo el bit implícito a la izquierda del punto binario. Dos números de valor muy próximo son:

$X = 1{,}000....00\ 2^1$

$Y = 1{,}11....111\ 2^0$

Para restar el menor del mayor, debe desplazarse aquél para igualar los exponentes, como se muestra en la primer parte del ejemplo. En este proceso Y pierde un bit significativo, el resultado que se obtiene es 2^{-22}. En la segunda parte del ejemplo se repite la misma operación pero con bits de guarda. Ahora el bit menos significativo no se pierde al ajustar el significante, y el resultado es 2^{-23}, diferente en un factor 2 del resultado anterior. Esto permite implementar un sistema de redondeo que minimice en la medida de lo posible la pérdida de precisión.

Ejemplo 1: Sin bits de guarda.

$$X = 1{,}000.....00 \times 2^1$$

$$-Y = 0{,}111.....11 \times 2^1$$

$$Z = 0{,}000.....01 \times 2^1$$

$$Z = 1{,}000.....00 \times 2^{-22}$$

Ejemplo 2: Con bits de guarda.

$$X = 1{,}000.....00 \quad 0000 \times 2^{1}$$

$$-Y = 0{,}111.....11 \quad 1000 \times 2^{1}$$

$$Z = 0{,}000.....00 \quad 1000 \times 2^{1}$$

$$Z = 1{,}000.....00 \quad 0000 \times 2^{-23}$$

9.5.1 Redondeo.

Otro detalle que afecta a la precisión del resultado es la política de redondeo aplicada. El resultado de cualquier operación sobre los significantes generalmente se almacena en un registro más grande (con bits de guarda). Cuando el resultado se pone de nuevo en el formato de punto flotante, hay que eliminar los bits extra.

Se han explorado diversas técnicas para realizar el redondeo. De hecho el standard IEEE 754 enumera cuatro aproximaciones alternativas:

- *Redondeo al más próximo.*
- *Redondeo hacia $+\infty$.*
- *Redondeo hacia $-\infty$.*
- *Redondeo hacia cero.*

La más sencilla de las cuatro es la que se conoce como *redondeo hacia cero*, que consiste simplemente en truncar los bits extra. De este modo, el resultado es siempre menor o igual que el valor original, introduciéndose un sesgo en sentido decreciente en la operación. Este sesgo puede llegar a ser importante, pues es siempre en el mismo sentido y afecta a todas las operaciones para las cuales haya algún bit extra (de guarda) distinto de cero.

Por este motivo el modo implícito asumido por el standard es el *redondeo al más próximo* que se define como sigue: Debe tomarse el valor representable más próximo al resultado exacto; y si hay dos valores representables igualmente próximos se adopta aquél cuyo bit menos significativo es cero (o sea se redondea los casos intermedios al valor par). Por ejemplo, si suponemos que el sistema utiliza 5 bits de guarda y estos bits son 10010, la cantidad indicada supera la mitad del valor correspondiente a la última posición representable. En tal caso, la respuesta correcta es sumar 1 al último bit representable, redondeando por exceso al número siguiente. Si los bits extra son en cambio 01111, corresponden a una cantidad menor que la mitad de la última posición representable. Entonces los bits extra se ignoran, lo que equivale a redondear al número representable anterior. El caso especial en que la distancia es la misma (o sea que los bits extra sean 10000), la norma establece que se fuerza un resultado par. Como puede verse, esta política de redondeo evita un sesgo siempre del mismo sentido, lo que reduce el error introducido por las limitaciones en cantidad de bits de la representación al realizar operaciones aritméticas.

Finalmente, las dos opciones restantes, *redondeo a más o menos infinito*, son útiles en la implementación de una técnica conocida como aritmética de intervalos. El origen de esta técnica es el siguiente: Al final de una secuencia de operaciones en punto flotante no podemos saber el resultado exacto debido a las limitaciones del hardware, que unas veces redondeará por exceso y otras por defecto. Si realizamos todos los cálculos de la secuencia dos veces, una vez redondeando por defecto y otra por exceso, obtendremos los límites inferior y superior del resultado correcto. Si el rango entre dichos límites es suficientemente estrecho, el resultado obtenido es bastante preciso. Si no, al menos lo sabremos y podremos realizar análisis adicionales. Por supuesto, esta técnica se aplica cuando por las características de la aplicación se justifique.

10.

Guía de Problemas sugeridos.

1. Convierta los siguientes números binarios a decimales :

    1.1 101110

    1.2 1110101

    1.3 110110100

2. Convierta a decimales los siguientes números :

    2.1 $(12121)_3$

    2.2 $(4310)_5$

    2.3 $(1A9)_{12}$

    2.4 $(2056)_7$

3. Convierta los siguientes números decimales a los sistemas indicados :

    3.1 7562 a octal.

    3.2 1938 a hexadecimal.

    3.3 245 a binario.

4. Convierta el número hexadecimal F3A7C2 a binario y octal.

5. Muestre el contenido de un registro de 12 bits donde se ha almacenado el número decimal 215 en :

    5.1 binario.

    5.2 octal codificado en binario.

    5.3 hexadecimal codificado en binario.

    5.4 decimal codificado en binario.

6. Obtenga los complementos a 1 y a 2 de los siguientes números binarios :

    6.1 10101110

    6.2 10000001

    6.3 10000000

    6.4 00000001

    6.5 00000000

7. Obtenga el complemento a 9 y 10 de los siguientes números decimales :

7.1 12349876

7.2 00980100

7.3 90000051

7.4 10000000

7.5 00000000

8. Convierta al formato standard de punto flotante IEEE 754 de simple precisión, los siguientes números. Presente el resultado como 8 dígitos hexadecimales.

8.1 9 8.2 -5 / 32 8.3

8.4 5 / 32 8.5 6.125 8.6

9. Convierta los siguientes números en formato IEEE 754 en simple precisión de hexadecimal a decimal:

9.1 42E48000H

9.2 00800000H

9.3 3F880000H

9.4 C7F00000H

10. Presente los circuitos para un generador de paridad de 3 bits y para un comprobador de paridad de 4 bits utilizando un bit de paridad par.

11. Realice las operaciones aritméticas indicadas en binario utilizando la representación de complemento a 2 y de complemento a 1.

11.1 (+107) + (-89) =

11.2 (-107) – (- 89) =

12. Realice las operaciones indicadas en decimal, utilizando la representación de complemento a la base con signo.

12.1 (- 638) + (- 785) =

12.2 (+ 638) – (- 785) =

13. Realice las operaciones aritméticas siguientes con números binarios en complemento a dos. Utilice 7 bits para alojar cada número con signo. En cada caso, determine si hay un sobreflujo al comprobar los acarreos hacia adentro y hacia fuera de la posición de bit de signo.

13.1 (+ 35) + (+ 40) =

13.2 (- 35) + (- 40) =

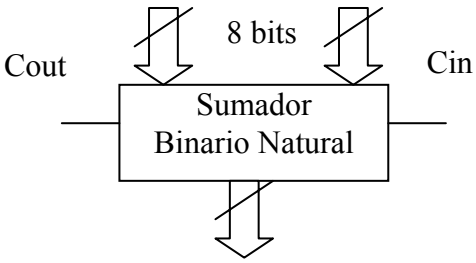
13.3 (- 35) – (+ 40) =

14. Resuelva (- 9) + (- 6) con números binarios en complemento a 1 con signo, utilizando 5 dígitos (incluyendo el signo).

    14.1 Funciona en este caso el método propuesto para detección de sobreflujo de los dos acarreos distintos?

    14.2 Proponga un método alternativo para detectar el sobreflujo en este complemento.

15. Utilizando el circuito indicado, opere en binario usando C2 y C1 e indique en cada caso si el resultado es ***correcto***. Si no lo es, justifique con claridad el/los motivos que lo causan y explique, detallando en qué consiste, alguna modificación al circuito que permita obtener el resultado correcto.



    15.1 S = $-(24)_{16} - (43)_5$

    15.2 S = $-(167)_8 - (31)_4$