IPF

José María Uriburu 820 - Formosa

TECNICATURA SUPERIOR EN DESARROLLO DE SOFTWARE MULTIPLATAFORMA

ASIGNATURA: Arquitectura y Sistemas Operativos

Año 2022

Instituto Politécnico Formosa

UNIDAD III - Parte 2

Representación de la información

Índice

3.5. BCD exceso tres

Este código se deriva del BCD y se obtiene sumando 3 al mencionado código. Este código es particularmente útil en la ejecución de operaciones aritméticas usando complementos. Al igual que el código BCD ponderado, este código sirve para representar números decimales a binarios, por grupos de 4 bits por cada dígito decimal.

Como se dijo al principio, se obtiene a partir del BCD puro, sumando un 3 binario a cada cifra decimal. Así se logra un código simétrico o autocomplementario, que facilita la operación de hallar el complemento de un número (complemento restringido decimal, complemento lógico o negación) sólo con invertir sus dígitos.

En la figura, se muestran las equivalencias entre decimal y BCD exceso tres. Debemos observar que 0111 (4) es el complemento restringido o a uno de 1000 (5), o que 0011 (0) es el complemento restringido de 1100 (9). Completamos el cuadro siguiente y confirmamos lo anterior.

Decimal	BCD	BCD exceso tres
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	
6		1001
7		
8		
9		

"Código autocomplementario"

Como su nombre lo indica, cada carácter codificado en exceso en tres es tres unidades mayores que en BCD. Así, seis o 0110 se escriben 1001, que es nueve en BCD. Ahora bien, 1001 solamente es nueve en BCD, en el código de exceso en tres, 1001 es seis.

EJEMPLO:

Decimal		Exceso en tres
2	----------→	0101
25	----------→	0101 1000
629	----------→	1001 0101 1100
3271	----------→	0110 0101 1010 0100
4785	----------→	
9746	----------→	

El código de exceso en tres facilita la operación aritmética, es decir:

3	0110
+ 9	+ 1100
12	0001 0010
	+ 0011 0011
	0100 0101

Se lee 0100 0101 o 12 (exceso en tres).

Existen algunas reglas especiales aplicables a la suma (como la adición de 3 a cada uno de los números del ejemplo anterior), pero estos pasos se realizan fácilmente, y de modo automático, en la computadora, haciendo del código de exceso en tres muy conveniente para las operaciones aritméticas. En el código de exceso en tres, el reconocimiento de la representación de las cifras no es directo, ya que al leer cada dígito debe restarle mentalmente en tres, si bien ello resulta más fácil que la conversión de números grandes representados en el sistema binario puro.

El BCD es un código ponderado; sin embargo, el de Exceso en Tres no lo es. Un bit de la segunda posición (2) de BCD representa un 2. En el código de exceso en tres, un bit situado en una cierta posición no indica la adición de un valor numérico al número. Por ejemplo, en BCD, 0100 es 4 y al sumarle el bit 2 se añade un 2, resultando el número 0110, o sea, dos unidades mayor. En el código de Exceso en tres, 0111 representa la cifra 4 y la cifra 6 es 1001, no existiendo un cambio numérico sistemático.

3.6. BCD AIKEN o 2421

Este código adjudica a cada dígito “1" binario el peso que resulta de la combinación 2421, en lugar del BCD puro que impone un peso igual a 8421. Por ejemplo, el número 7en BCD 2421 es “1101" y surge de la suma de los dígitos “1” multiplicados por sus pesos, o sea, 2+4+1 = 7.

También se usa en gran medida por su simetría, que permite hallar con facilidad el complemento de un número, o sea que cuando una operación BCD involucra el complemento restringido de un dígito, éste se obtiene invirtiendo los bits de la combinación de la misma forma que se hace en sistema binario.

En la figura 3.6 se observan las equivalencias entre decimal y BCD 2421.

Decimal	BCD 2421
0	0000
1	0001
2	0010
3	0011
4	0100
5	1011
6	1100
7	1101
8	1110
9	1111

“Código autocomplementario”

Continuando con los ejemplos presentados, se pueden observar las diferencias de representación de un número en estos últimos dos sistemas.

Decimal	BCD exceso tres	BCD 2421
$15_{(10)}$	0001 1000	0001 1011
$256_{(10)}$	0101 1000 1111	0010 1011
$1_{(10)}$	0100	0001
$284_{(10)}$	0101 1011 0111	0010 1110 0100
$679_{(10)}$		
$127_{(10)}$		
$307_{(10)}$		
$687_{(10)}$		
$67_{(10)}$		
$47_{(10)}$		

3.7. Códigos de representación numérica no decimal

Estos códigos permiten la **operación aritmética de datos** en sistema binario o en otras bases no decimales. Los operandos ingresan primero en código alfanumérico para luego convertirse a alguno de estos convenios. Si bien la conversión es más compleja que cuando se opera en sistema decimal, los datos que intervendrán en cálculos reducen su tamaño en grado considerable, con lo que se operan con mayor rapidez. A modo de ejemplo, se puede recordar cuántos dígitos ocupa el número $15_{(10)}$ en un convenio decimal, en contraposición al binario:

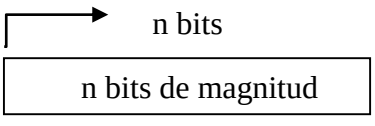
$$0001\ \ 0101_{(BCD)} \qquad 1111_{(2)}$$

3.7.1. Coma o punto fijo sin signo (Enteros Positivos)

Es el formato más simple para representar números enteros positivos, expresados como una secuencia de dígitos binarios:

$$X_{n-1};\ X_{n-2};....x_2;\ X_1;\ X\sigma$$

Como este formato sólo permite números sin signo, para la representación de un número se utiliza la totalidad de bits del formato.

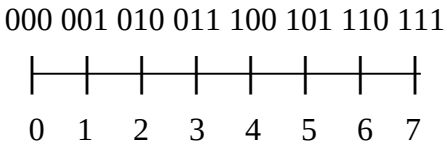


El rango de variabilidad para un número X en este formato es:

$$0 \leq X \leq 2^{n} - 1$$

donde "n" es la cantidad de dígitos que componen el número; también se la conoce como tipo de dato "ordinal sin signo".

Por ejemplo, para un formato de n = 3 bits, el rango de variabilidad estará comprendido entre 0 y 7. Expresado de otra manera, permite representar los números comprendidos en ese rango, como se observa en la recta de representación siguiente:



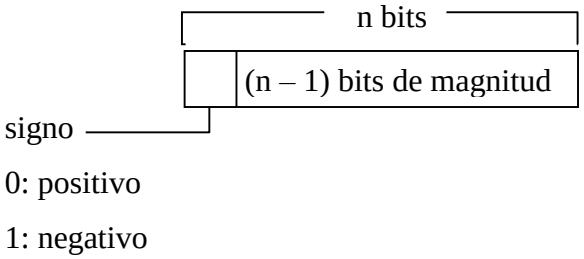
Por lo tanto, si se suman dos números cuyo resultado supere el rango de variabilidad estipulado, este resultado será erróneo, ya que perderá el dígito más significativo. El ejemplo siguiente muestra esta condición para n = 3:

101	5
+ 111	+ 7
$1100_{(2)}$	$12_{(10)}$

Las unidades de cálculo en una CPU actualizan "señalizadores", llamados *flags* o banderas, que indican, entre otras, esta circunstancia de desborde u *overflow* que, para este ejemplo en particular, implica la pérdida del bit de orden superior.

3.7.2. Coma o punto fijo con signo (enteros)

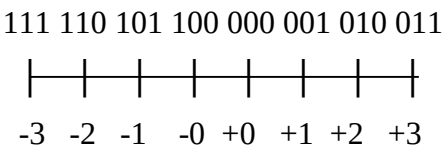
Es igual al formato anterior, pero reserva el bit de extrema izquierda para el signo, de manera que si ese bit es igual a *0* indicará que el número es positivo; asimismo, si, por el contrario, es igual a *1* indicará que el número es negativo. A este tipo de representación también se la llama "magnitud con signo".



El rango de variabilidad para los binarios puros con signo es:

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

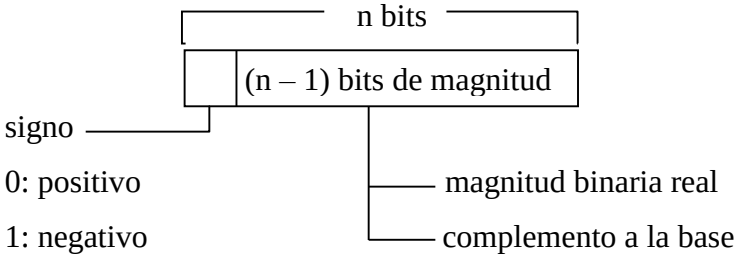
Para el ejemplo de n = 3 bits, el rango de variabilidad estará comprendido entre -3 y +3, según la recta de representación siguiente:



Nótese la presencia de un cero positivo y otro negativo. Este último surge del cambio de signo, 0 por 1, para esa misma magnitud.

3.7.3. Coma o punto fijo con signo con negativos complementados a "2" (enteros)

Es igual al formato anterior, reservando el bit de extrema izquierda para el signo. La diferencia radica en que los n -1 bits restantes se representan con su magnitud binaria real para el caso de los positivos y en el complemento a la base de la magnitud para el caso de los negativos.



El rango de variabilidad para los binarios con signo en complemento a la base es:

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

3.7.4. Coma o punto fijo con signo con negativos complementado a "1" (enteros)

Es igual al formato anterior, reservando el bit de extrema izquierda para el signo. No obstante, los n -1 bits restantes se representan con su magnitud binaria real para el caso de los positivos y en complemento **restringido** de la magnitud para el caso de los negativos.

El rango de variabilidad para los binarios puros con signo en complemento restringido o complemento a 1, es:

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

En esta codificación también hay dos "ceros", uno positivo y otro negativo.

3.7.5. Reales en coma o punto flotante (números muy grandes y números reales)

Los números analizados hasta ahora se pueden clasificar como enteros sin signo y enteros signados, en general limitados por un formato de 8, 16, 32 o 64 bits; sin embargo, a veces es necesario operar datos mayores (por ejemplo, para programas que resuelvan cálculos científicos).

La primera solución para tratar números muy grandes es ampliar el formato a tres, cuatro u ocho octetos y obtener así valores decimales que oscilan entre alrededor de los 3 millones y los 9 trillones (considerando enteros signados). De todas maneras, en los cálculos no siempre intervienen valores enteros y es entonces donde aparece el inconveniente de cómo representar el punto para manejar cantidades fraccionarias. Por un lado, el punto debería expresarse con un dígito “0", o bien, con un dígito "1", dentro de la cadena de bits que representa al número binario; por otro lado, el punto no asume una posición fija sino variable dentro de la cadena, o sea que si el punto se representa con un bit “1" no se podrá diferenciar en una secuencia "1111" el binario 1111 del binario 111,1, o bien del binario 1,111. La solución para este problema es eliminar el punto de la secuencia e indicar de alguna manera el lugar que ocupa dentro de ella. De esta forma, los números se representan considerando dos partes. La primera indica los bits que representan al número y se llama **mantisa**, la otra indica la posición del punto dentro del número y se llama **exponente**. Esta manera de tratar números fraccionarios tiene cierta semejanza con lo que se conoce en matemática como notación científica o exponencial. Veamos cómo se puede representar un número decimal:

$$3{,}5_{(10)} = \quad 35 \cdot 10^{-1} \quad \text{o bien, es igual a}$$
$$0{,}35 \cdot 10^{+1}$$

En el primer caso, *35* es un número entero que, multiplicado por la base *10* y elevado a la *-1*, expresa el mismo número inicial sin representar la coma dentro de él. Así, "*35*" es la mantisa y “-1" , el exponente, que se puede leer como "el punto está **un** lugar a la izquierda del dígito menos significativo de la mantisa”.

En el segundo caso, *0,35* es un número fraccionario puro (se indica entonces que es una fracción **normalizada**, dado que se supone que el punto está a la izquierda del dígito más significativo y distinto de cero de la mantisa). Luego, *0,35* multiplicado por la base *10* y elevado a la *+1* expresa el mismo número inicial, sin representar la coma dentro de él. En este caso, "+1" indica que el punto está un lugar a la derecha del dígito más significativo de la mantisa.

Cuando se trata de números binarios con punto fraccionario se utilizan, entonces, dos entidades numéricas, una para la mantisa (por lo general una fracción normalizada y con signo) y

otra para el exponente. La base del sistema es siempre conocida, por lo tanto, no aparece representada. Este nuevo formato de números se conoce como binarios de punto (coma) flotante, ya que el punto varía su posición ("flota") según el contenido binario de la entidad exponente.

La fórmula $\boxed{\pm M \cdot B^{\pm p}}$

representa el criterio utilizado para todo sistema posicional, donde "M" es la mantisa, que podrá tratarse como **fracción pura** o como **entero**, según se **suponga** la coma a **izquierda** o a **derecha** de "M". "B" es la base del sistema; "P" es el exponente que indica la posición del punto en "M" y los signos "+" y "-" que la anteceden indican su desplazamiento a derecha o izquierda respecto del **origen**. En la memoria sólo se almacenarán la mantisa "M" y la potencia "P" en binario. La base "B" y el **origen** del punto siempre son determinados con anterioridad por el convenio y conocidos por los programas que utilizan esta representación.

La representación de datos en punto flotante incrementa el rango de números más allá de los límites físicos de los registros. Por ejemplo, se sabe que en un registro de 16 bits el positivo mayor definido como entero signado que se puede representar es el +32767. Si se considera el registro dividido en partes iguales, 8 bits para la mantisa entera y 8 bits para el exponente, el entero mayor se calcula según la fórmula siguiente:

$$+(2^7 - 1) \cdot 2^{+(2_7 - 1)} = +127 \cdot 2^{+127}$$

que implica que al binario representativo del entero 127 se le agregan 127 ceros a derecha; de modo que enteros muy grandes encuentran su representación en esta modalidad.

Si ahora se representa el valor $3{,}5_{(10)}$ en binario 11,1 resulta:

$+3{,}5_{(10)} = +11{,}1_{(2)}$ expresado en notación científica es igual a

$+111{,}0_{(2)} \cdot 10_{(2)}^{-1_{(2)}} = +7_{(10)} \cdot 2_{(10)}^{-1_{(10)}} = +7/2 = +3{,}5_{(10)}$ mantisa entera

$+0{,}111_{(2)} \cdot 10_{(2)}^{+10_{(2)}} = +0{,}875_{(10)} \cdot 2_{(10)}^{+2_{(10)}} = +3{,}5_{(10)}$ mantisa fraccionaria

Dado que los registros que operen estos datos tendrán una longitud fija, se asume un formato de "m" bits para la mantisa y "p" bits para el exponente, entonces se puede definir el rango de representación de números reales, que es siempre finito. O sea que determinados reales son "representables" en el formato y otros quedan "afuera" de él.

Cuando el resultado de una operación supera los límites mayores definidos por el rango, entonces es incorrecto. El error se conoce como ***overflow*** de resultado y actualiza un bit (flag de overflow) en un registro asociado a la ALU, conocido como registro de estado o *status register*.

Cuando el resultado de una operación cae en el hueco definido por los límites inferiores del rango (del que se excluye el cero), esto es:

$0 > x > 0{,}1$ o $0 < x < 0{,}1$

es incorrecto. El error se conoce como ***underflow*** de resultado.

Ejemplo 1:

Si tenemos 1,234E7; es un número muy grande. Esto indica un producto, para éste caso en sistema decimal.

$1{,}234 \times 10^7 = 12340000$ => aquí el exponente nos indica donde va la coma. Es decir, para el ejemplo son siete dígitos después de la coma.

1,234E7; LA BASE SE CONSIDERA IMPLICITA, en el ejemplo es base 10.

MANTISA EXPONENTE

Punto flotante en binario:

Si tenemos un sistema de mantisa entera de binario con signo de 4 bits y un exponente de binario sin signo de 3 bits. Queremos saber qué número está representado en base 10, por la siguiente cadena:

Tenemos: * Mantisa Entera BCS 4 bits.

* Exponente BSS 3 bits.

$0110010_2 = ?_{10}$

Armamos la mantisa y su exponente

\+ mantisa exponente

\- 1 1 0 , 0 1 0 => queda: 0 1 1 0 , 0 1 0

$d_2\ d_1\ d_0$

bits menos significativo, indica el lugar de la coma.

-Luego interpretamos la mantisa de acuerdo al sistema planteado, quedando:

-Utilizamos los pesos de los bits.

$+ (0.2^0 + 1.2^1 + 1.2^2) * (0.2^0 + 1.2^1 + 0.2^2)$

6 * $2^{Exp=2}$ → Exponente

Base (implícita)

Queda: $6 * 2^2 = 24_{10}$

Ejemplo 2:

$1011111_2 = ?_{10} = -384$

$- 011, 111 =$

$- (1*2^0 + 1*2^1 + 0*2^2) * (1*2^0 + 1*2^1 + 1*2^2) =$

$-3*2^7 = - 3 * 128 = -384$

a) Interpretamos la mantisa y el exponente:

Ejemplo 3: Mantisa fraccionaria en BCS de 4 bits, exponente BSS de 3 bits.

$1010011_2 = ?_{10} = -16$

- 0, Mantisa 0,0101…

$-010\ 011_2 = ?_{10}$

$-(0*2^{-1} + 1 * 2^{-2} + 0 *2^{-3}) * (1*2^0 + 1*2^1 + 0*2^2) =$

$- (2^{-2} * 2^3) = - 2$

EJERCICIOS: Representar las cadenas en punto flotante cuya mantisa es fraccionaria, con 6 bits y exponente BSS entero con 4 bits.

CADENA	M en BCS	M en BSS
0101110110		
0000010000		
0000111001		
1111111111		

0000000000		
0000001111		
1111110000		
1000000000		
0000011111		

3.8. Representaciones redundantes

3.8.1. Códigos de detección y/o corrección de errores. Introducción

El cambio de un bit en el almacenamiento o la manipulación de datos origina resultados erróneos. Para detectar e incluso corregir estas alteraciones se usan códigos que agregan "bits redundantes". Por ejemplo, los bits de paridad se agregan al dato en el momento de su envío y son corroborados en el momento de su recepción por algún componente de la computadora, y lo mismo sucede al revés. De todas formas, la cantidad de errores que puedan producirse son medibles probabilísticamente. El resultado de "medir" la cantidad de "ruido" que puede afectar la información se representa con un coeficiente conocido como **tasa de error**. De acuerdo con el valor de la tasa de error, se selecciona un código, entre los distintos desarrollados, para descubrir e incluso corregir los errores. Estos códigos reciben el nombre de códigos de paridad.

3.8.2. Paridad vertical simple o a nivel carácter

Al final de cada carácter se incluye un bit, de manera que la suma de "unos" del carácter completo sea par (paridad par) o impar (paridad impar). Este tipo de codificación se denomina paridad vertical. Suele acompañar la transmisión de octetos en los periféricos y la memoria.

Los ejemplos muestran el bit de paridad que se agrega al octeto, en el caso de usar paridad par o impar:

00110010 **|** 1 se agrega un uno que permita obtener paridad par

01001011 **|** 0 se agrega un cero para lograr paridad par

Este código de detección de errores sólo se utiliza si la tasa de error en la cadena de bits que se han de transmitir no es superior a uno; esto es, si hay dos bits erróneos no permite detectarlos.

3.8.3. Paridad horizontal a nivel de bloque

Por cada bloque de caracteres se crea un byte con bits de paridad. El bit 0 será el bit de paridad de los bits 0 del bloque; el bit 1, el de paridad de los bits 1 del bloque, y así sucesivamente.

3.8.4. Código de Hamming

Este código permite detectar y corregir los errores producidos en una transmisión con sólo agregar *p* bits de paridad, de forma que se cumpla la relación siguiente:

$$2^P \geq i + p + 1$$

donde i es la cantidad de dígitos binarios que se han de transmitir y p es la cantidad de bits de paridad.

Supongamos que se desean transmitir 4 dígitos binarios. Según el método de Hamming a éstos deberá agregarse la cantidad de bits de paridad necesarios para satisfacer la relación enunciada antes:

i = 4 para que la relación $2^P \geq i + p + 1$ se cumpla debe ser

p = ? $p = 3$, luego $2^3 \geq 4 + 3 + 1$

De esta relación se deduce que la transmisión será de 4 bits de información más 3 de paridad par. Se debe tener en cuenta que con 2 bits de paridad se puede corroborar la transmisión de 1 dígito de información, con 3 de paridad se corroboran hasta 8 bits de información, con 4 de paridad se corroboran hasta 11 de información, y así sucesivamente.