Arquitectura y Organización de Computadoras

Representación digital

Hasta el momento hemos considerado únicamente la representación y la aritmética del conjunto de los naturales y el cero, es decir, de **números enteros no negativos**.

Sin embargo, en nuestro dominio, la computación, tarde o temprano nos enfrentaremos con algunas dificultades:

- Existen otros conjuntos numéricos, como los **enteros (positivos, cero y negativos)**, que aparecen frecuentemente en los problemas de la vida real.
- Otros conjuntos muy importantes que aún no podemos manejar son los **racionales**, los **reales** y los **complejos**.
- Además, hemos visto anteriormente que las computadoras disponen de una **cantidad limitada de dígitos** binarios para la representación numérica.

Estudiaremos entonces cómo podemos resolver el problema de representar diferentes tipos de datos numéricos con una **cantidad finita** de dígitos binarios.

Datos y tipos de datos

Al procesar los datos, las computadoras los mantienen en unidades de almacenamiento de un tamaño determinado. Los tamaños habituales para estas unidades son, por ejemplo, **8, 16, 32 o 64 bits**. Por otro lado, los lenguajes de programación proveen una cantidad limitada de **tipos de datos**, con los que pueden operar los programas escritos en esos lenguajes, y que se guardan en unidades de almacenamiento de un tamaño conveniente.

Además del tamaño de las unidades de almacenamiento, diferentes tipos de datos tienen determinadas características. Por ejemplo, los tipos de datos **enteros** pueden ser **con signo o sin signo**, o bien los tipos de datos **reales** pueden tener **una determinada precisión**. Cuando un programador necesita manejar un cierto dato dentro de un programa, elige cuidadosamente un tipo de dato provisto por el lenguaje y que le ofrezca las características que necesita para su problema.

Aclaremos que cuando hablamos de **enteros con o sin signo**, nos referimos a los tipos de datos, y no a los datos mismos. Es decir, un número puede ser positivo (y no llevar un signo “menos”) pero estar representado con un tipo de dato con signo. Como la información de si un número es o no negativo corresponde a una **pregunta binaria**, un tipo de dato entero **con signo** necesita destinar un bit para almacenar esta información.

Rango de representación

Al representar números enteros, con o sin signo, con una cantidad finita **n** de dígitos, en un sistema de base **b**, tenemos una primera limitación. No todos los números pueden ser representados. ¿Cuántos números podemos representar en el sistema decimal, con **n** dígitos?

- El menor número **sin signo** que podemos representar es el 0; pero el mayor número **sin**

signo representable en un sistema de base **b** resulta ser $\mathbf{b^n-1}$. El intervalo **[0, $\mathbf{b^n-1}$] es el llamado rango de representación** del sistema. En el sistema binario los números **sin signo** abarcan el rango **de 0 a $\mathbf{2^n-1}$**, siendo **n** la cantidad de dígitos.

- Cuando consideramos números **con signo**, este rango de representación cambia según la técnica de representación que usemos.

Representación de enteros con signo

Veremos dos métodos de representación[1]. El primero se llama **signo-magnitud**, y resulta el más intuitivo para los humanos; sin embargo, tiene algunos inconvenientes.

Signo-magnitud

Un número en notación **signo-magnitud** representa el **signo** con su bit de orden más alto, o bit más significativo (el bit más a la izquierda), y el **valor absoluto** del número con los restantes bits. Por ejemplo, si la unidad de almacenamiento del dato mide 8 bits, **-1** se representará como **10000001**, y +1 como **00000001** (igual que si se tratara de un entero sin signo). El método es intuitivo y directo, y fácil de hacer a mano sabiendo representar los enteros sin signo.

Notemos que al usar un bit para signo, automáticamente perdemos ese bit para la representación de la magnitud o valor absoluto, disminuyendo así el rango de representación, el cual se divide por 2. Con la misma unidad de almacenamiento de 8 bits, el entero **sin signo** más grande representable era $\mathbf{2^8-1 = 255}$ (que se escribe **11111111**). Ahora, con signo-magnitud, el entero con signo más grande representable será $\mathbf{2^7-1 = 127}$ (que se escribe **01111111**).

El rango de representación en **signo-magnitud** con **n dígitos** es $\mathbf{[-2^{n-1} + 1, 2^{n-1} - 1]}$.

Para los humanos, el método de signo-magnitud resulta sumamente cómodo, pero presenta algunos inconvenientes para su uso dentro de la computadora. En primer lugar, el 0 tiene dos representaciones, lo que complica las cosas. En segundo lugar, la aritmética en signo-magnitud es bastante dificultosa, tanto para los humanos como para la computadora.

Para ver la dificultad de operar en este sistema, basta con tratar de hacer a mano algunas operaciones sencillas de suma o resta en aritmética binaria en signo-magnitud. Primero hay que decidir si los signos son iguales; en caso contrario, identificar el de mayor magnitud para determinar el signo del resultado; luego hacer la operación con los valores absolutos, restituir el signo, etc. La operación se vuelve bastante compleja. Otros métodos de representación (los sistemas de complemento) resuelven el problema con mayor limpieza.

Complemento a 2

El segundo método es el de **complemento a 2**, que resuelve el problema de tener dos representaciones para el 0 y simplifica la aritmética binaria, tanto para los humanos como para las computadoras. El método de complemento a 2 también emplea el bit de orden más alto de acuerdo al signo, pero la forma de representación es diferente:

- Los positivos se representan igual que si se tratara de enteros sin signo (como en el caso de signo-magnitud).
- Para los negativos, se usa el procedimiento de **complementar a 2,** que puede resumirse en tres pasos:

1 http://es.wikipedia.org/wiki/Representaci%C3%B3n_de_n%C3%BAmeros_con_signo

- expresamos en binario el **valor absoluto** del número,
- **invertimos los bits**
- y **sumamos 1**.

Ejemplo

Expresar los números **-23, -9 y 8** en sistema binario en complemento a 2 con 8 bits.

- El valor absoluto de -23 es **23**. Expresado en binario en 8 bits es **00010111**. Invertido bit a bit es **11101000**. Sumando 1, queda **11101001**, luego $\mathbf{-23_{(10} = 11101001_{(2}}$.
- El valor absoluto de -9 es **9**. Expresado en binario en 8 bits, es **00001001**. Invertido bit a bit es **11110110**. Sumando 1, queda **11110111**, luego $\mathbf{-9_{(10} = 11110111_{(2}}$.
- El número **8** es positivo. Su expresión en el sistema de complemento a 2 es la misma que si fuera un número sin signo. Queda $\mathbf{8_{(10} = 00001000_{(2}}$.

El rango de representación para el sistema de **complemento a 2 con n dígitos** es $\mathbf{[-2^{n-1}, 2^{n-1} - 1]}$.

Es decir, contiene un valor más que **signo-magnitud**.

Otra propiedad interesante del método es que, si sabemos que un número binario es **negativo** y está escrito en complemento a 2, podemos recuperar su valor en decimal usando casi exactamente el mismo procedimiento: **invertimos los bits** y **sumamos 1**. Convertimos a decimal como si fuera un entero sin signo y luego agregamos el signo habitual de los decimales.

Ejemplo

Convertir a decimal los números, **en complemento a 2**, 11101001, 11110111 y 00000111.

- **11101001** es negativo; invertido bit a bit es 00010110; sumando 1 queda 00010111 que es 23. Agregando el signo decimal queda $\mathbf{11101001_{(2} = -23_{(10}}$.
- **11110111** es negativo; invertido bit a bit es 00001000; sumando 1 queda 00001001 que es 9. Agregando el signo decimal queda $\mathbf{11110111_{(2} = -9_{(10}}$.
- **00000111** es positivo, porque su bit de orden más alto es 0. Lo interpretamos igual que si fuera un entero sin signo y entonces $\mathbf{00000111_{(2} = 7_{(10}}$.

Ejercicios

- Utilizando sólo dos bits, representar todos los números permitidos por el rango de representación en complemento a 2 e indicar su valor decimal.
- Idem con tres bits.
- ¿Cuál es el rango de representación en complemento a 2 para **n = 8**?

Nota

Es interesante ver cómo los diferentes sistemas de representación asignan diferentes valores a cada combinación de símbolos. Veámoslo para unos pocos bits.

Decimal	Sin signo	Signo-magnitud	Complemento a 2
-8			1000
-7		1111	1001
-6		1110	1010
-5		1101	1011
-4		1100	1100
-3		1011	1101
-2		1010	1110
-1		1001	1111
0	0000	0000 y 1000	0000
1	0001	0001	0001
2	0010	0010	0010
3	0011	0011	0011
4	0100	0100	0100
5	0101	0101	0101
6	0110	0110	0110
7	0111	0111	0111
8	1000		
9	1001		
10	1010		
11	1011		
12	1100		
13	1101		
14	1110		
15	1111		

La tabla muestra que la representación en complemento a 2 es más conveniente para las operaciones aritméticas:

- La operación de complementar a 2 da un número negativo que resuelve las restas. Las computadoras **no restan: suman complementos**.
- El cero tiene una única representación.
- Se ha recuperado la representación del -8, que estaba desperdiciada en signo-magnitud.
- El rango de representación tiene un valor más.
- Los números en complemento a 2 se disponen como en una rueda. Si tomamos cada número en la columna de complemento a 2 (salvo el último) y le sumamos 1, obtenemos en forma

directa el número aritméticamente correcto. Similarmente si restamos 1 (salvo al primero). Al sumar 1 al **7**, obtenemos el menor número negativo posible. Al restar 1 a **-8**, obtenemos el mayor positivo ("entramos a la tabla por el extremo opuesto"). Esta propiedad, que no se cumple para signo-magnitud, hace más fácil la implementación de las operaciones aritméticas en los circuitos de la computadora.

- Una suma en complemento a 2 se ejecuta sencillamente bit a bit, sin las consideraciones necesarias para signo-magnitud.

La representación en complemento a 2 es más natural y consistente. Por estos motivos se elige normalmente para las computadoras, en lugar de otros métodos de representación.

Suma y resta en complemento a 2

La suma se realiza bit a bit desde la posición menos significativa, con acarreo (nos "llevamos un bit" cuando la suma de dos bits da $10_{(2}$). Cuando uno de los sumandos es negativo, lo complementamos y hacemos la suma. Una resta **A – B** se interpreta como la suma de **A** y el complemento de **B**.

Ejemplo

Restar **$2_{(10}$ de $23_{(10}$**. Complementamos 2 para convertirlo en negativo: **$2_{(10}$ = 11111110 $_{(2}$,** y lo sumamos a **$23_{(10}$**. La cuenta es **$23_{(10}$ + (-$2_{(10}$)= 00010111 $_{(2}$ + 11111110 $_{(2}$ = 00010101** que es positivo y vale **21**.

Sumar **$23_{(10}$ y -$9_{(10}$** con 8 bits. Se tiene que **$23_{(10}$ = 00010111 $_{(2}$** y **-$9_{(10}$** = **11110111$_{(2}$**. La suma da **00001110$_{(2}$**, que es positivo y equivale a **$14_{(10}$**.

Overflow

Notemos que en el caso de sumar 23 y -9, el acarreo (o "**carry**") puede llegar hasta el bit de signo (**carry-in**), y también pasar más allá, siendo descartado (**carry-out**). El acarreo a veces denuncia un problema con las magnitudes permitidas por el rango de representación del sistema, como se ve en este ejemplo.

Ejemplo

Sumar **$123_{(10}$ y $9_{(10}$** con 8 bits. Se tiene que **$123_{(10}$ = 01111011 $_{(2}$** y **$9_{(10}$** = **00001001$_{(2}$**. Notemos que la suma provoca el acarreo de un **1** que llega al bit de signo ("**carry-in**"). El resultado da **10000100$_{(2}$**. Claramente tenemos un problema, porque la suma de 123 + 9 debería dar positiva, pero el resultado que obtuvimos tiene el bit de orden más alto en **1**; por lo cual es negativo. Este es un caso patológico de la suma que se llama **overflow**.

Un riesgo de disponer de finitos dígitos binarios es que las operaciones aritméticas pueden exigir más bits de los que tenemos disponibles. Los números de nuestro ejemplo, **$123_{(10}$** y **$9_{(10}$**, pueden expresarse individualmente como enteros con signo en 8 bits, pero suman **$132_{(10}$**, número positivo que excede el rango de representación. El resultado **no se puede** escribir en 8 bits con signo en complemento a 2.

Cuando pasa esto se dice que ha ocurrido una condición de **overflow** o **desbordamiento aritmético**[2]. Ocurre **overflow** cuando se suman dos números positivos y se obtiene un resultado en bits que sería negativo en el rango de representación del sistema, o cuando se suman dos negativos y el resultado es positivo. En complemento a 2, nunca puede ocurrir **overflow** cuando se suman un

2 http://es.wikipedia.org/wiki/Desbordamiento_aritm%C3%A9tico

positivo y un negativo.

Las computadoras detectan la condición de overflow porque el bit de acarreo al llegar a la posición del signo (**carry-in**) y el que se descarta (**carry-out**) son de diferente valor. Al sumar 23 más -9, el carry-in y el carry-out eran iguales, por lo cual la suma no sufrió overflow. Al sumar 123 más 9, el carry-in es 1 pero no existe carry-out, lo que nos advierte que ha ocurrido overflow.

	1	**1**	1	1	1	1	1	
23	0	0	0	1	0	1	1	1
+								
-9	1	1	1	1	0	1	1	1
	0	0	0	0	1	1	1	0

	1	1	1	1	1	1	1	
123	0	1	1	1	1	0	1	1
+								
9	0	0	0	0	1	0	0	1
	1	0	0	0	0	1	0	0

Representación de números racionales

Habiendo visto la representación de enteros, con una simple convención podemos representar racionales.

Números reales