Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 Arquitectura de Computadores

Representación de números

1. Motivación

Todo tipo de información puede ser representada mediante números. Una palabra, por ejemplo, puede ser representada mediante números reemplazando cada letra por un número específico (como el código ASCII). Una imagen puede representarse como un conjunto de posiciones (números) y el valor de color de la imagen en esa posición (también números). En definitiva, podemos reducir el problema de representar información al problema de representar números, por lo que es fundamental conocer las representaciones numéricas existentes y como se utilizan.

2. Representaciones numéricas

La representación numérica más simple corresponde a usar un símbolo por cada incremento en uno de un número. Por ejemplo, si ocupamos el símbolo «*» para representar el número cuatro, dibujamos cuatro símbolos: «****». El problema de esta representación es que no escala: para representar el número un millón, necesitamos dibujar un millón de veces el símbolo «*» lo que sería un costo tanto en espacio (por ejemplo papel si estuviésemos escribiendo el número en un cuaderno) y de tiempo (escribir un millón de veces el símbolo tomará al menos un millón de segundos, aproximadamente 11 días sin parar).

Para solucionar los problemas que presenta esta representación, se inventaron representaciones más avanzadas, que permiten de alguna forma representar números grandes con pocos símbolos. Una de estas representaciones fueron los números romanos, que ocupaban símbolos específicos para acumulaciones de números: V para cinco, X para diez, L para cincuenta, etc. Esta representación, a pesar de ser una mejora sobre la anterior, tiene serias limitaciones prácticas, ya que no contempla un símbolo para el cero ni permite hacer operaciones algebraicas.

Otra representación ideada fueron los números indoarábigos, que es la que ocupamos hoy en día, basada en diez símbolos: 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. Esta representación se caracteriza por pertenecer a un grupo de representaciones denominadas «representaciones posicionales».

2.1. Representaciones posicionales

Las representaciones posicionales, como la indoarábiga, se basan en dos elementos para determinar el valor de un número: la posición de los símbolos en la secuencia de estos y la cantidad de símbolos posible, lo que se denomina la base. En el caso de los números indoarábigos la base es diez (i.e. hay diez símbolos posibles), y por esto esta representación numérica también se denomina «representación decimal».

Para entender como afecta la posición en el valor numérico, es necesario realizar un ejemplo. Como estamos tan acostumbrados a asociar un número a su representación en el sistema decimal, para este ejemplo,

vamos a reemplazar los símbolos asociados al número uno y dos por «?» y «&» de manera de poder abstraernos del número y fijarnos en la representación.

Supongamos se tiene la secuencia de numérica «??&». La expresión para poder interpretar este número en el sistema decimal es la siguiente:

$$? \times diez^{dos} + ? \times diez^{uno} + \& \times diez^{cero} \tag{1}$$

Al evaluar esta expresión obtendremos el valor de la secuencia numérica, pero debemos primero elegir la representación en la que queremos hacer el cálculo. Si utilizamos la representación decimal, y reemplazamos los símbolos «?» y «&» por sus valores 1 y 2 obtenemos:

$$1 \times 10^2 + 1 \times 10^1 + 2 \times 10^0 = 112 \tag{2}$$

A partir de esto podemos obtener la regla general para obtener el valor de una secuencia de símbolos en representación decimal:

$$\sum_{k=0}^{n-1} s_k \times 10^k \tag{3}$$

donde s_k corresponde al símbolo en la posición k, n corresponde al número de símbolos en la secuencia y k toma valores desde 0 hasta $n-1$, es decir, las posiciones parten desde 0, tomando como inicio el símbolo de más a la derecha.

De la expresión anterior se puede inferir que podemos crear representaciones posicionales con otras bases, para lo cual bastaría reemplazar el número diez por la base deseada. De esta manera, la expresión general para obtener el valor numérico de una determinada representación posicional es:

$$\sum_{k=0}^{n-1} s_k \times b^k \tag{4}$$

donde b es la base de la representación elegida, la cual podría ser cualquier valor numérico. De hecho, el uso de la base diez que consideramos habitual se debe solamente a que los seres humanos tenemos diez dedos, y por tanto atribuimos a ese número una cierta característica especial. Si tuviéramos ocho dedos, seguramente ocuparíamos una sistema posicional con base ocho. En definitiva, no existe un sistema que de por si sea mejor que otro, todo depende de las circunstancias y de su uso.

2.2. Representaciones binaria, octal y hexadecimal

Existen tres representaciones habitualmente usadas en el contexto de la computación: binaria, octal y hexadecimal, siendo la primera la de mayor importancia. Tal como sus nombres lo señalan, la representación binaria tiene base dos, la octal base ocho y la hexadecimal base dieciséis. Vamos a comenzar analizando la base octal, para luego pasar a las otras dos.

Dado que un número octal tiene base ocho, necesitamos ocho símbolos distintos para su representación. Por comodidad utilizaremos los primeros ocho números indoarábigos: 0, 1, 2, 3, 4, 5, 6 y 7. Es importante destacar que aunque podríamos ocupar cualquier otro grupo de ocho símbolos, sólo ocuparemos estos por conveniencia, dado que estamos acostumbrados a trabajar matemáticamente con éstos.

Tomemos como ejemplo de número octal el 112. El primer elemento importante a destacar es que esta secuencia de símbolos no representa al número «ciento doce» que habitualmente asociaríamos a esos símbolos si estuviesen en representación decimal. Es decir, la secuencia de símbolos es la misma, pero la representación distinta. Para evitar confusiones, para todas las bases no decimales se usa especificar la base junto a la secuencia de símbolos. Entonces, en nuestro caso, el número sería $(112)_8$, es decir la secuencia 112 en representación octal.

Si utilizamos la expresión (4) podemos obtener el valor numérico de esta secuencia octal. Es importante resaltar que, al ocupar esta expresión, debemos decidir en que representación queremos el resultado. Lo habitual es que éste quede en representación decimal, pero podría quedar también en otra si lo quisiéramos. Más adelante veremos ejemplos de este tipo, pero por ahora haremos el cálculo de manera de obtener el valor en representación decimal:

$$1 \times 8^2 + 1 \times 8^1 + 2 \times 8^0 = 74 \tag{5}$$

Es decir, la secuencia 112 en octal, representa el número decimal 74.

Revisemos ahora la representación binaria, la cual tiene como base el número dos, y por tanto requiere sólo dos símbolos. Ocuparemos, nuevamente y por comodidad, los dos primeros números indoarábigos, 0 y 1. Tomemos como ejemplo el número $(1011)_2$. Nuevamente es importante recordar que esta secuencia no tiene ninguna relación con el número «mil once» en representación decimal, solamente tiene la misma secuencia de símbolos. Podemos nuevamente aplicar la expresión (4) para obtener el valor numérico en representación decimal:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 1^0 = 11 \tag{6}$$

La ventaja de esta representación es que nos basta con dos símbolos para representar todos los posibles números. La desventaja es que necesitamos más posiciones para representar números. Esta desventaja la hace poco práctica para ser usada por humanos, pero la ventaja de tener sólo dos estados la hace ideal para realizar cálculos automáticos, por ejemplo en un computador.

Podemos también definir una representación cuya base sea mayor que diez. Una representación importante que cumple con esto es la representación hexadecimal, la cual tiene como base el dieciséis. Más adelante veremos que la utilidad de esta representación está en su relación con la representación binaria, y lo fácil que es convertir de una a otra. Para esta representación necesitamos dieciséis símbolos, por lo que a diferencia de las otras representaciones estudiadas no nos basta con los números indoarábigos, necesitamos seis símbolos más. Para no tener que inventar nuevos símbolos específicos de esta representación, habitualmente se ocupan los diez símbolos indoarábigos para representar los diez primeros dígitos y las primeras seis letras del abecedario (A=diez, B=once, C=doce, D=trece, E=catorce, F=quince) para representar los seis restantes.

Por ejemplo, la secuencia $(A1F)_{16}$ puede ser interpretada como un número hexadecimal. Para obtener el valor numérico en representación decimal, nuevamente ocupamos la expresión (4):

$$A \times 16^2 + 1 \times 16^1 + F \times 16^0 = ? \tag{7}$$

En este caso tenemos un problema al evaluar directamente la expresión: aparecen símbolos (A y F) que no son válidos en la representación decimal. Para solucionar esto, podemos convertir estos símbolos directamente a representación decimal: A=10 y F=15, y con estos valores reescribir la expresión para obtener el resultado:

$$10 \times 16^2 + 1 \times 16^1 + 15 \times 16^0 = 2560 + 16 + 15 = 2591 \tag{8}$$

Este ejemplo demuestra una regla importante: si ocupamos la expresión (4) para convertir un número de una representación A a otra representación B, en el caso de que la base de A sea mayor que B (por ejemplo dieciséis es mayor que diez) necesariamente necesitamos un paso previo antes de realizar el cálculo: convertir todos los símbolos no válidos de la representación A a la representación B (en el ejemplo $A = 10$ y $F = 15$).

A modo de ejemplo de la regla anterior, veamos el caso de convertir un número decimal a representación binaria, por ejemplo el número 123. Si aplicamos directamente la ecuación (4) tenemos:

$$1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 = 1 \times 100 + 2 \times 10 + 3 \times 1 \tag{9}$$

El primer problema que se nos presenta es el explicado en la regla anterior: los símbolos 2 y 3 no existen en la representación binaria, por lo cual debemos en primer lugar reemplazarlos por su representación: $2 = (10)_2$ y $3 = (11)_2$:

$$(1)_2 \times 100 + (10)_2 \times 10 + (11)_2 \times 1 \tag{10}$$

Ahora tenemos un segundo problema: los números 100 y 10 están representados en decimal, por tanto debemos también reemplazarlos por su valor en esta representación: $100 = (1100100)_2$ y $10 = (1010)_2$.

$$(1)_2 \times (1100100)_2 + (10)_2 \times (1010)_2 + (11)_2 \times (1)_2 = (1100100)_2 + (10100)_2 + (11)_2 = (1111011)_2 \tag{11}$$

Este ejemplo nos muestra una clara desventaja de ocupar la expresión (4) para transformar desde representación decimal a una de menor base: necesitamos saber a priori la representación de la base y todas las potencias de la base (en este caso 100 y 10) en la representación no decimal para realizar el cálculo. Más adelante estudiaremos otros mecanismos que permiten realizar esta conversión sin necesidad de este conocimiento.

Nota: En general, cuando todos los números que se están usando sean de la misma representación, se puede obviar la notación $(num)_{base}$. Para el caso de números binarios y hexadecimales, existen otras notaciones habituales usadas cuando se quiere diferenciar la representación:

- Un número binario $(num)_2$ se suele escribir también como $numb$, por ejemplo $(1011)_2$ se puede escribir como $1011b$
- Un número hexadecimal $(num)_{16}$ se suele escribir también como $numh$, por ejemplo $(A1)_{16}$ se puede escribir como $A1h$
- Un número hexadecimal $(num)_{16}$ se suele escribir también como $0xnum$, por ejemplo $(A1)_{16}$ se puede escribir como $0xA1$

2.3. Aritmética en distintas representaciones

La gran ventaja de las representaciones posicionales es que los procedimientos aritméticos como suma y multiplicación son equivalentes para toda representación. Estudiaremos primero la aritmética de la representación decimal, a la cual estamos habituados y a partir de ésta generalizaremos las reglas de la suma que son válidas para cualquiera de estas representaciones.

Como ejemplo realizaremos paso a paso la suma entre los números 112 y 93. El algoritmo tradicional para realizar la suma es el siguiente:

1. Escribir uno de los números debajo del otro, alineados por la derecha:

$$\begin{array}{r} 132 \\ 93 \\ \hline \end{array}$$

2. Sumar los dos dígitos de más a la derecha. Si la suma es menor que diez (5 en este caso), continuar con el siguiente dígito hacia la izquierda:

$$\begin{array}{r} 132 \\ 93 \\ \hline 5 \end{array}$$

3. Sumar los siguientes dos dígitos. Si la suma es menor que diez, continuar con el siguiente dígito hacia la izquierda. Si la suma es mayor o igual que diez (12 en este caso), restarle 10 a la suma y colocar como resultado la resta (2 en este caso). Convertir los 10 en 1 y agregarlo como sumando a los siguientes dígitos (este valor se denomina **acarreo** o **carry** en inglés) :

$$\begin{array}{r} {\scriptstyle 1} \\ 132 \\ 93 \\ \hline 25 \end{array}$$

4. Sumar los siguientes dos dígitos. En caso de haber acarreo sumarlo también. Revisar los mismos casos que antes :

$$\begin{array}{r} {\scriptstyle 1} \\ 132 \\ 93 \\ \hline 225 \end{array}$$

El algoritmo general para la suma en representación decimal de dos números $num1$ y $num2$ sería:

1. Comenzar desde la derecha en la posición 0 de ambos números.
2. Mientras no lleguemos al fin de ambos números por la izquierda ni queden acarreos sin sumar:
   a) Sumar los dígitos de la posición actual más algún posible acarreo previo.
   b) Si la suma es menor que 10, colocar en la posición actual del resultado el valor de la suma.
   c) Si la suma es mayor o igual que 10, restarle 10 a la suma, colocar lo que resulta de la resta en la posición actual del resultado y agregar un acarreo para la siguiente posición.

En el algoritmo anterior usamos solamente dos veces el número 10: al comparar que la suma de los dígitos sea mayor que 10 y luego en caso de que se cumpla esto, al restarle 10 al valor obtenido. En base a esto podemos generalizar este algoritmo para sumar cualquier par de números en representación posicional de base b reemplazando el número 10 por la variable b:

1. Comenzar desde la derecha en la posición 0 de ambos números.
2. Mientras no lleguemos al fin de ambos números por la izquierda ni queden acarreos sin sumar:
   a) Sumar los dígitos de la posición actual más algún posible acarreo previo.
   b) Si la suma es menor que b, colocar en la posición actual del resultado el valor de la suma.
   c) Si la suma es mayor o igual que b, restarle b a la suma, colocar lo que resulta de la resta en la posición actual del resultado y agregar un acarreo para la siguiente posición.

Usemos este algoritmo con un ejemplo de dos números binarios $(110)_2$ y $(11)_2$, en el cual la base $b = 2$:

1. Comenzar desde la derecha en la posición 0 de ambos números:

$$\begin{array}{r} 110 \\ 11 \\ \hline \end{array}$$

2. Mientras no lleguemos al fin de ambos números por la izquierda ni queden acarreos sin sumar

   a) Sumar los dígitos de la posición actual más algún posible acarreo previo. Si la suma **(en este caso 1)** es menor que 2, colocar en la posición actual del resultado el valor de la suma.

$$\begin{array}{r} 110 \\ 11 \\ \hline \mathbf{1} \end{array}$$

   b) Sumar los dígitos de la posición actual más algún posible acarreo previo. Si la suma **(en este caso 2)** es mayor o igual que 2, restarle 2 a la suma, colocar lo que resulta de la resta en la posición actual del resultado **(en este caso 0)** y agregar un acarreo para la siguiente posición.

$$\begin{array}{r} \mathbf{1} \\ 110 \\ 11 \\ \hline \mathbf{0}1 \end{array}$$

   c) Sumar los dígitos de la posición actual más algún posible acarreo previo. Si la suma **(en este caso 2)** es mayor o igual que 2, restarle 2 a la suma, colocar lo que resulta de la resta en la posición actual del resultado **(en este caso 0)** y agregar un acarreo para la siguiente posición.

$$\begin{array}{r} \mathbf{1}1 \\ 110 \\ 11 \\ \hline \mathbf{0}01 \end{array}$$

   d) Sumar los dígitos de la posición actual más algún posible acarreo previo. Si la suma **(en este caso 1)** es menor que 2, colocar en la posición actual del resultado el valor de la suma.

$$\begin{array}{r} 11 \\ 110 \\ 11 \\ \hline \mathbf{1}001 \end{array}$$

Ejercicio: Multiplique los números binarios $(101)_2$ y $(10)_2$. Para hacerlo, primero analice el algoritmo de multiplicación de números decimales y luego aplíquelo a la multiplicación de números binarios.

2.4. Algoritmos de conversión entre representaciones

Como se analizó previamente la ecuación (4) representa un método general de conversión entre bases. Esta ecuación funciona bien para transformar un número en representación no decimal a la decimal (en la cual podemos realizar operaciones aritméticas rápidamente), sin embargo, como también se vio, para el caso inverso no era tan útil, y por tanto es necesario explorar otros algoritmos. Adicionalmente, para el caso de los números binarios existen otros algoritmos o heurísticas que pueden resultar más simples.

En esta sección se revisarán primero algoritmos de conversión binario-decimal y decimal-binario, y luego algoritmos de conversión entre representaciones hexadecimal, octal y binaria.

2.4.1. Algoritmos de conversión binario-decimal

La ecuación (4) representa un método útil para convertir entre números binarios y decimales, sin embargo, requiere que seamos capaces de recordar rápidamente las potencias del número dos, lo cual puede ser fácil para potencias bajas, pero para potencias altas puede volverse complejo y por tanto puede ser necesario tener que calcular estas potencias previo a la conversión. A continuación se presentan dos algoritmos que permiten convertir un número binario a decimal sin conocer las potencias de dos.

Algoritmo de acarreo inverso

La idea de este algoritmo es comenzar desde la izquierda e ir invirtiendo el proceso de acarreo, es decir por cada vez que vemos un número 1 lo devolvemos a la derecha como un número 2. Veamos el algoritmo a través de un ejemplo: vamos a convertir el número $(101101)_2$ a representación decimal.

1. Tomamos el primer 1, **1**01101, lo convertimos en 2 y lo sumamos al dígito de la derecha: 0**2**1101
2. Repetimos el paso para el siguiente dígito. Esta vez tenemos un 2 lo que equivale a dos 1 y por tanto debemos sumar dos veces 2 = 4 al siguiente dígito de la derecha: 00**5**101
3. Repetimos el paso para el siguiente dígito. Esta vez tenemos un 5 lo que equivale a cinco 1 y por tanto debemos sumar cinco veces 2 = 10 al siguiente dígito de la derecha: 000**11**01
4. Repetimos y ahora sumamos 22 al siguiente dígito de la derecha: 0000**22**1
5. Repetimos y ahora sumamos 44 al siguiente dígito de la derecha: 00000**45**. Llegamos al fin del número, y por tanto tenemos que el resultado de la conversión es 45.

Algoritmo de multiplicar e incrementar

Este algoritmo es equivalente al anterior, pero presenta reglas más simples y no requiere pensar el proceso de acarreo inverso. Las reglas de este algoritmo son:

- Tomar el 1 de más a la izquierda del número binario, e ir avanzando de izquierda a derecha. Comenzar con el resultado en 1.
- Por cada número 0 que se encuentra, multiplicar el resultado actual por 2.
- Por cada número 1 que se encuentra, multiplicar el resultado actual por 2 y sumar 1.

Veamos el algoritmo a través de un ejemplo: vamos a convertir el número $(101101)_2$ a representación decimal.

1. Tomamos el 1 de más a la izquierda, **1**01101. Comenzamos con el resultado = 1.
2. Tomamos el 0 que sigue, 1**0**1101: multiplicamos por 2 el resultado: resultado = 2
3. Tomamos el 1 que sigue, 10**1**101: multiplicamos por 2 el resultado y sumamos 1: resultado = 5
4. Tomamos el 1 que sigue, 101**1**01: multiplicamos por 2 el resultado y sumamos 1: resultado = 11
5. Tomamos el 0 que sigue, 1011**0**1: multiplicamos por 2 el resultado: resultado = 22
6. Tomamos el 1 que sigue, 10110**1**: multiplicamos por 2 el resultado y sumamos 1: resultado = 45

2.4.2. Algoritmos de conversión decimal-binario

Como mencionamos anteriormente, la ecuación (4) no es un método práctico para realizar esta conversión. A continuación presentaremos dos algoritmos para realizar la conversión:

Algoritmo de acarreos sucesivos

La idea de este algoritmo es comenzar con todos los números acumulados a la derecha (se puede pensar como si se tuvieran fichas acumuladas como una torre) e ir acarreando grupos de a dos hacia la izquierda, convirtiéndolos en 1 a medida que se acarrean.

Veamos el algoritmo a través de un ejemplo: vamos a convertir el número 45 a representación binaria.

1. Comenzamos con 45 «fichas» a la derecha.
2. Acarreamos todos los pares de fichas que haya a la izquierda, en este caso 22 y por cada par, sumamos un 1 a la izquierda. En la primera posición nos sobra una ficha, lo que representa un número 1. En este momento llevamos el número 22 **1**
3. Acarreamos todos los pares de fichas que haya en la próxima posición, en este caso 11 y por cada par, sumamos un 1 a la izquierda. Esta vez no nos sobra una ficha, lo que representa un número 0. En este momento llevamos el número 11 **0**1
4. Acarreamos todos los pares de fichas que haya en la próxima posición, en este caso 5 y por cada par, sumamos un 1 a la izquierda. Esta vez sobra una ficha, lo que representa un número 1. En este momento llevamos el número 5 **1**01
5. Acarreamos todos los pares de fichas que haya en la próxima posición, en este caso 2 y por cada par, sumamos un 1 a la izquierda. Esta vez sobra una ficha, lo que representa un número 1. En este momento llevamos el número 2 **1**101
6. Acarreamos todos los pares de fichas que haya en la próxima posición, en este caso 1 y por cada par, sumamos un 1 a la izquierda. Esta vez no nos sobra una ficha, lo que representa un número 0. En este momento llevamos el número 1 **0**1101.
7. Como el número de más a la izquierda es un 1 no tenemos nada más que acarrear y el resultado es 101101

Algoritmo de divisiones sucesivas

Este algoritmo es equivalente al anterior, pero permite mecanizar de mejor forma la conversión, sin tener que pensar en el acarreo. Consiste en los siguientes pasos:

- Dividir el número actual por 2: si la división es exacta (es decir, no hay resto), agregar un 0 a la izquierda del resultado. Actualizar el número actual como el resultado de la división.
- Dividir el número actual por 2: si la división no es exacta (es decir, hay resto), agregar un 1 a la izquierda del resultado. Actualizar el número actual como el resultado de la división.
- Detenerse si es que el número actual es 0.

Veamos el algoritmo a través de un ejemplo: vamos a convertir el número 45 a representación binaria.

a) Dividimos el número actual 45 en dos = 22, resto = 1. Como hay resto, agregamos un 1 a la izquierda del resultado=**1**.

b) Dividimos el número actual 22 en dos = 11, resto = 0. Como no hay resto, agregamos un 0 a la izquierda del resultado=**0**1.

c) Dividimos el número actual 11 en dos = 5, resto = 1. Como hay resto, agregamos un 1 a la izquierda del resultado=**1**01.

d) Dividimos el número actual 5 en dos = 2, resto = 1. Como hay resto, agregamos un 1 a la izquierda del resultado=**1**101.

e) Dividimos el número actual 2 en dos = 1, resto = 0. Como no hay resto, agregamos un 0 a la izquierda del resultado=**0**1101.

f) Dividimos el número actual 1 en dos = 0, resto = 1. Como hay resto, agregamos un 1 a la izquierda del resultado=**1**01101.

g) Nos detenemos porque el resultado actual es 0.

2.4.3. Algoritmos de conversión entre representaciones hexadecimal, octal y binaria

La conversión entre representaciones binaria-octal y binaria-hexadecimal es mucho más simple que la conversión entre binarios y decimales. Esto se debe a la relación de sus bases: ocho y dieciséis son potencias de dos. De esta forma, para convertir un número binario a octal, basta ir agrupando de a **tres** ($8 = 2^{\mathbf{3}}$) dígitos binarios e ir reemplazando su valor por el número octal correspondiente. A modo de ejemplo, realicemos la conversión del número $(10110)_2$ a octal:

binario:	010 110
octal:	2 6

Se observa además que el método funciona en ambas direcciones: para convertir el número octal $(26)_8$ en binario basta tomar cada uno de sus dígitos, representarlo como número binario y luego ubicarlos sucesivamente: $(2)_8 = (010)_2$ y $(6)_8 = (110)_2$ por lo tanto $(26)_8 = (10110)_2$

De manera equivalente, la conversión binaria-hexadecimal consiste en ir agrupando de a **cuatro** ($16 = 2^{\mathbf{4}}$) dígitos binarios e ir reemplazando su valor por el número hexadecimal correspondiente. A modo de ejemplo, realicemos la conversión del número $(10110)_2$ a hexadecimal:

binario:	0001 0110
hexadecimal:	1 6

Al igual que con los octales, la conversión es bidireccional: para convertir el número hexadecimal $(16)_{16}$ en binario basta tomar cada uno de sus dígitos, representarlo como número binario y luego ubicarlos sucesivamente: $(1)_{16} = (0001)_2$ y $(6)_{16} = (0110)_2$ por lo tanto $(16)_{16} = (10110)_2$.

Debido a esta fácil conversión entre representaciones, es habitual que se ocupen tanto la representación octal como la hexadecimal para referirse a números binarios, ya que requieren menos símbolos.

2.5. Representación de números negativos

Hasta ahora hemos visto representaciones sólo de números enteros y positivos. La primera pregunta que surge es ¿cómo representamos números negativos? Nuestra experiencia nos dice que la respuesta es simple: agregamos un «-» a la izquierda del número, por ejemplo -12 sería un número decimal negativo; $(-1100)_2$ sería un número binario negativo.

El problema de esto es que necesitamos agregar un nuevo símbolo a nuestro sistema numérico para poder representar números negativos. En el caso de los números binarios, en vez de tener sólo dos símbolos, necesitaríamos tener uno adicional sólo para indicar que un número es negativo. Nos gustaría buscar un mecanismo que aproveche los símbolos que ya tenemos en nuestra representación, y así evitar tener que agregar un símbolo nuevo.

A continuación se revisarán distintos métodos para representar números negativos binarios ocupando sólo los símbolos 0 y 1.

Dígito de signo

Un primer método que se puede ocupar es agregar un dígito extra a la izquierda del número que en caso de ser 0 indica que el número es positivo, y en caso de ser 1, que el número es negativo. Por ejemplo si tenemos el número 10011 y queremos representar el número -10011, reemplazamos el «-» por un 1, obteniendo: **1**10011.

Importante: al ocupar este tipo de representaciones ya no basta sólo con ver la secuencia de símbolos para saber su valor, debemos saber además que **tipo** de información se está guardando, por ejemplo en este caso un número entero, tal que el primer dígito indica el signo.

El problema de esta representación es que no nos sirve para realizar operaciones aritméticas. Para que una representación de un número negativo sirva debe cumplir la ecuación: $A + (negativo(A)) = 0$ es decir, debe ser un inverso aditivo válido.

En nuestro caso, si al número positivo **0**10011 le sumamos el número negativo **1**10011 obtenemos 1000110. Debemos entonces buscar otras representaciones para poder realizar operaciones aritméticas con números negativos.

Complemento a 1

Una posible mejora corresponde la representación denominada «complemento a 1». Esta consiste en agregar un dígito de signo, al igual que en el caso anterior, y luego, para obtener el negativo de un número, reemplazar todos los 0s por 1s y los 1s por 0s. Siguiendo nuestro ejemplo con el número 10011, primero se le agrega un cero a la izquierda: **0**10011, lo que se interpreta como «agregar el bit de signo al número positivo». Luego se hace la inversión para todos los dígitos **1**01100, lo que nos deja un número «negativo» dado que su bit de signo es 1.

Ahora si realizamos la suma entre **0**10011 y **1**01100 obtenemos **1**11111. Si hacemos lo mismo con el número 101 por ejemplo, obtendremos que el número negativo sería **1**010 y si los sumamos, obtenemos **1**111.

Podemos observar que aparece un patrón común al ocupar complemento a 1: la suma entre un número y su inverso aditivo resultará en una secuencia de números uno. Esto nos indica que estamos por buen camino, y nos falta sólo un detalle para llegar a una representación más adecuada.

Complemento a 2

La representación que soluciona el problema del complemento a 1 se denomina «complemento a 2». Consiste en ejecutar los mismos pasos que en complemento a 1, y adicionalmente sumarle 1 al número obtenido. Veamos paso a paso con un ejemplo, el número 10011:

- El primer paso consiste en agregar el cero a la izquierda: **0**10011
- Luego se realiza el reemplazo de 0s y 1s: **1**01100
- Por último, le sumamos un 1 al número: **1**01101

Ahora probaremos si efectivamente esta representación nos entrega un inverso aditivo válido. Si sumamos **0**10011 con **1**01101 obtenemos: **1**000000, lo que podemos interpretar como «0», ya que el 1 de más a la izquierda lo podemos interpretar como el signo, y sabemos que $0 = -0$ por lo tanto se cumple que la representación en complemento a 2 si funciona como inverso aditivo.

De ahora en adelante, cuando hablemos de un número binario negativo, asumiremos que está representado en complemento a 2, a menos que se diga lo contrario.

3. Representación de números reales

Los números enteros representan sólo un porcentaje menor de todos los posibles números que se pueden representar, por lo que es necesario estudiar como representar números fraccionales y reales en un computador. Sin embargo, la representación de números fraccionales presenta una serie de complicaciones y limitaciones que debemos entender para poder ocuparla correctamente y evitar errores.

3.1. Fracciones decimales y binarias

Un número entero en representación decimal puede ser representado en su forma posicional como la suma de sus dígitos ponderado por potencias de la base 10. Por ejemplo el número 112 puede representarse como:

$$1 \times 10^2 + 1 \times 10^1 + 2 \times 10^0 = 112 \tag{12}$$

De manera similar, un número en representación binaria puede ser también representado en su forma posicional como la suma de sus dígitos ponderado por potencias de la base, en este caso 2. Por ejemplo el número $(1100)_2$ puede representarse como:

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 8 + 4 + 0 + 0 = 12 \tag{13}$$

La representación posicional puede ser extendida para números fraccionarios, ocupando potencias negativas de la base al ponderar. Por ejemplo el número en representación decimal 112,234 puede representarse como:

$$1 \times 10^2 + 1 \times 10^1 + 2 \times 10^0 + 2 \times 10^{-1} + 3 \times 10^{-2} + 4 \times 10^{-3} = 100 + 10 + 2 + 0{,}2 + 0{,}03 + 0{,}004 = 112{,}234 \tag{14}$$

De manera equivalente, podemos extender la representación posicional para números fraccionarios en representación binaria. Por ejemplo el número $(1100{,}011)_2$ puede representarse como:

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 8 + 4 + 0 + 0 + 0 + 0{,}25 + 0{,}125 = 12{,}375 \tag{15}$$

Al igual que en los números enteros, la ecuación (4) puede ser interpretada como un algoritmo de conversión binario-decimal. Nos gustaría encontrar también un algoritmo de conversión decimal-binario para números fraccionales, de manera de por ejemplo obtener la representación del número 0,1 en binario. Un algoritmo simple es el siguiente:

- Reescribir el número decimal en su forma fraccional: $0{,}1 = \frac{1}{10}$
- Transformar el numerador y denominador a binario: $\frac{1}{10} = \frac{(1)_2}{(1010)_2}$
- Realizar la división: $1 : 1010 = ?$

Para poder completar este algoritmo, debemos primero revisar como se realiza la división de números binarios.

División decimal y binaria

Tal como en el caso de la multiplicación y la suma, la división de números binarios ocupa exactamente el mismo procedimiento que su contraparte decimal. Revisaremos primero un ejemplo de una división decimal: $60 : 25$:

- El primer paso corresponde a ver cuántas veces cabe completamente el divisor (número de la derecha) en el dividendo (número de la izquierda). En este caso cabe 2 veces, ya que $2 \times 25 = 50$, por lo cual lo escribimos como primer dígito del resultado el número 2 y debajo del dividendo el valor efectivo de la multiplicación entre el resultado y el divisor, en este caso 50:

$$\begin{array}{l} 60 : 25 = 2 \\ 50 \end{array}$$

- El siguiente paso consiste en restarle al dividendo el resultado de la multiplicación resultado-divisor. En este caso $60 - 50 = 10$:

$$\begin{array}{rl} & 60 : 25 = 2 \\ - & 50 \\ \hline & 10 \end{array}$$

- Ahora se repite el primer paso, pero esta vez ocupando como dividendo el resultado de la resta (10). En caso de que el dividendo sea menor que el divisor, agregamos 0s a la derecha del dividendo hasta que este sea mayor o igual que el divisor. Por cada 0 que se agrega en el dividendo, se compensa avanzando en un dígito fraccional del resultado a la derecha. En este caso basta agregar un 0 y como tal quedamos posicionados en el primer dígito fraccional del resultado

$$\begin{array}{rl} & 60 : 25 = 2. \\ - & 50 \\ \hline & 10\mathbf{0} \end{array}$$

- Finalmente dividimos ahora si el dividendo actual por el divisor, en este caso nos da como resultado 4. Como el divisor cabe exactamente en el dividendo, nos detenemos y el resultado final es 2,4.

$$\begin{array}{rl} & 60 : 25 = 2{,}4 \\ - & 50 \\ \hline & 100 \\ - & 100 \\ & 0 \end{array}$$

La división binaria es equivalente. La única diferencia es que las operaciones aritméticas intermedias deben ser realizadas ocupando aritmética binaria. Veremos el proceso con un ejemplo: $3 : 4 = (11)_2 : (100)_2$

- El primer paso corresponde a ver cuántas veces cabe completamente el divisor en el dividendo. En este caso no cabe, y por tanto debemos aplicar la técnica de agregar 0s al dividendo y desplazarnos en los dígitos fraccionales

$$11\mathbf{0} : 100 = 0.$$

- Al agregar un 0 el divisor (4) cabe una vez en el dividendo (6), por tanto agregamos un 1 al resultado, multiplicamos el resultado por el divisor y se lo restamos al dividendo, obteniendo en este caso como resto $(10)_2 = 2$.

$$\begin{array}{rl} & 110 : 100 = 0{,}1 \\ - & 100 \\ \hline & 010 \end{array}$$

- Repetimos el paso de agregar un 0 en el resto, que es ahora nuestro nuevo dividendo, quedando este con el valor $(100)_2 = 4$. Vemos que el divisor cabe exactamente 1 vez en el dividendo, y por tanto agregamos un 1 al resultado, y obtenemos resto 0 lo que nos indica que terminamos la división.

$$\begin{array}{rl} & 110 : 100 = 0{,}11 \\ - & 100 \\ \hline & 0100 \\ - & 0100 \\ \hline & 0 \end{array}$$

Podemos comprobar que el resultado es correcto convirtiéndolo a decimal:

$$0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 0 + 0{,}5 + 0{,}25 = 0{,}75 = \frac{3}{4} \tag{16}$$

Ahora que sabemos como dividir en binario, podemos completar el último paso de nuestro algoritmo de conversión, que era dividir $(1)_2 : (1010)_2$:

- Primero agregamos ceros hasta que el divisor quepa. Una vez conseguido esto multiplicamos por 1 el divisor y se lo restamos al dividendo.

$$\begin{array}{rl} & 1\mathbf{0000} : 1010 = 0{,}0001 \\ - & 01010 \\ \hline & 00110 \end{array}$$

- Repetimos el proceso con el nuevo divisor, agregando un 1 al resultado, restando la multiplicación resultado-divisor en el dividendo, y actualizando el dividendo como el resto.

$$\begin{array}{rl} & 10000 : 1010 = 0{,}00011 \\ - & 01010 \\ \hline & 00110\mathbf{0} \\ - & 001010 \\ \hline & 000010 \end{array}$$

- Agregamos los 0s necesarios nuevamente para continuar el proceso

$$\begin{array}{rl} & 10000 : 1010 = 0{,}00011001 \\ - & 01010 \\ \hline & 00110\mathbf{0} \\ - & 001010 \\ \hline & 000010\mathbf{000} \\ - & 000001010 \\ \hline & 000000110 \end{array}$$

Podemos notar que a esta altura estamos repitiendo divisiones que ya hicimos exactamente igual, y por tanto sabemos que nunca terminaremos esta división. De hecho si continuamos dividiendo observaremos que el resultado es de la forma 0,00011001100110011... lo que corresponde a un número infinito semi-periódico: $0{,}0\overline{0011}$.

Este resultado es completamente contraintuitivo e inesperado, porque básicamente nos dice que **el número** 0,1 **tiene una representación infinita en binario**. Podemos ir incluso más allá: esto demuestra que dado un número fraccional cualquiera, el hecho de que su representación sea finita o infinita depende

exclusivamente de la base utilizada en la representación. Por ejemplo la fracción $\frac{1}{3}$ que en base decimal tiene la representación infinita periódica $0.\overline{3}$, en base ternaria (3) tendrá la representación finita 0,1.

La relevancia de esto es que dado que los computadores tienen un espacio finito para almacenar información, si ocupamos números binarios para representar números que en base decimal son finitos (como el 0,1), pero son infinitos en base binaria, obligatoriamente podemos almacenar sólo una aproximación de éste, lo que afectará en los resultados de operaciones que realicemos con este tipo de números.

3.2. Representaciones en un computador

Debido a que los computadores tienen espacio de almacenamiento limitado, los números que se almacenen en estos se guardan en porciones limitadas también. En general un número se almacenará mediante una cantidad fija de dígitos binarios (conocidos como **bits** de su nombre en inglés **bi**nary digi**ts**). Esto aplica tanto para números enteros como fraccionales, pero como se vio en la sección anterior es de particular importancia para los números fraccionales, que en muchos casos tendrá representación infinita, y como tal, al tener una cantidad fija de bits se deberán almacenar aproximaciones de los números.

La forma específica en que se guardan los números fraccionales en los bits del almacenamiento de un computador ha variado en el tiempo, pero son dos las principales representaciones usadas: punto fijo y punto flotante. La mayoría de los computadores actuales ocupa la segunda, pero ambas tienen posibles ventajas y desventajas que se deben considerar.

3.2.1. Representación de punto fijo (fixed point)

La representación de punto fijo consiste en que dado un espacio de n bits para almacenar un número, se reservan t bits para almacenar la parte entera del número y f bits para almacenar la parte fraccional, donde $n = t + f + 1$ (el bit extra se utiliza para almacenar el signo). De esta forma el punto (o coma) de la representación fraccional queda «fijo» en la t-ésima posición de la secuencia de bits.

Como ejemplo supongamos el número binario fraccional 10,111. Si tenemos $n = 8$ bits para almacenar todo el número, $t = 4$ bits para almacenar la parte entera y $f = 4$ bits para almacenar la parte fraccional, la representación almacenada del número sería:

0	010	1110
signo	t	f

3.2.2. Representación de punto flotante (floating point)

El problema que tiene la representación de punto fijo es que limita el **rango** posible de números. Para el ejemplo anterior ($n = 8$, $t = 3$ y $f = 4$) el máximo número positivo que podemos representar es el 111,1111 y el mínimo es 000,0001. Si pudiésemos mover o «flotar» el punto (o coma) libremente entre los 7 bits podríamos representar el número 1111111 y también el 0,000001, lo que nos daría un mayor rango, para así permitir trabajar tanto con números muy grandes como con números muy chicos. La representación usada para lograr esto se denomina representación de «punto flotante».

Para lograr que el punto «flote» se debe codificar de alguna forma para cada número la posición actual del punto. Una representación decimal que permite esto es la representación de notación científica, la cual codifica un número como una multiplicación entre un **significante** con una base (10) elevada a un **exponente**. En esta representación, el significante representa el valor del número y, dado que multiplicar por una potencia de 10 en representación decimal es equivalente a mover el punto, el valor del exponente está indicando la posición del punto.

Por ejemplo, el número 1023,456 se puede codificar en notación científica como: $1{,}023456 \times 10^3$. En este caso el significante sería 1,023456, la base 10 y el exponente 3, que se puede interpretar como «mover el punto 3 posiciones a la derecha». El número también podría codificarse como $10{,}23456 \times 10^2$ o $102345{,}6 \times 10^{-2}$, pero

en general se prefiere la que se usó inicialmente, con sólo un dígito a la izquierda de la coma del significante. Cuando un número en notación científica cumple con esta condición, se denomina **normalizado**.

La representación de punto flotante usada en el computador aplica la misma codificación de la notación científica, pero ahora con números binarios. De esta forma, ahora el significante y el exponente son representados como números binarios. Para mantener el hecho de que el exponente codifique la posición del punto, se utiliza como base el número 2 en vez de la base 10, ya que en representación binaria multiplicar por una potencia de 2 es equivalente a mover el punto.

De esta forma, por ejemplo, el número 10,111, lo podemos representar como $(1{,}0111)_2 \times 2^{(01)_2}$. El exponente $(01)_2 = 1$ al igual que en notación científica, lo interpretamos como «mover el punto 1 posición a la derecha».

Si queremos almacenar este número en $n = 8$ bits y definimos nuestra representación de manera de tener $s = 3$ bits de significante (normalizado), 1 bit de signo para el significante, $e = 3$ bits de exponente y 1 bit de signo para el exponente, podríamos almacenar el número de la siguiente forma:

0	101	0	001
signo s	s	signo e	e

Esto nos muestra de inmediato una clara desventaja de esta representación respecto a la de punto fijo: existe una pérdida de **precisión** es decir, de la cantidad de bits disponibles para almacenar un determinado valor. La precisión de un número de punto flotante está dada por la cantidad de bits de su significante, en este caso 3. La precisión de un número de punto fijo en cambio está dada por la cantidad de bits totales usadas por el número, en nuestro caso 7.

La ventaja es que aumentamos el **rango**: el máximo valor positivo representable en este caso es $(1{,}11)_2 \times 2^{(111)_2} = 11100000$ y el mínimo es $(0{,}01)_2 \times 2^{-(111)_2} = 0{,}00000001$. Este es un trade-off inevitable para una cantidad limitada de bits: para aumentar el rango, debemos reducir la precisión y viceversa. La representación de punto flotante se prefiere porque la pérdida de precisión se puede compensar, en parte, aumentando la cantidad de bits.

El estándar IEEE754

La representación de punto flotante antes descrita es una de muchas que se podría utilizar. Los parámetros relevantes para una representación son: el número total de bits, el número de bits asignados al significante, la normalización o no del significante y el número de bits asignados al exponente. En 1985 se definió el estándar IEEE754 que especifica como representan los computadores un número de punto flotante. El estándar define varias representaciones siendo dos las principales: «**single** precision floating point» y «**double** precision floating point».

La representación «single» (conocida en los lenguajes de programación Java y C# como **float**) define un tamaño de 32 bits para los números, de los cuales se ocupa 1 bit para el signo del significante, 23 bits para el valor del significante y 8 bits para el exponente:

1 bit	8 bits	23 bits
signo significante	exponente	significante

Esta representación tiene ciertas características especiales:

- El significante se almacena normalizado, pero sin el 1 que va a la izquierda de la coma. Por ejemplo el significante 1,101, se almacena como 10100000000000000000000, es decir se asume que todo significante comienza en 1. La ventaja de tener este dígito implícito es que aunque se almacenan 23 bits, la precisión del número es de 24 bits.

- El exponente se almacena desfasado en 127, es decir en vez de almacenar un bit de signo aparte, o representar el número en complemento a 2, se desfasa el número de manera de tener sólo valores positivos. La ventaja de esto está en hacer más simple la aritmética.
- Dado que al significante se le agrega un 1 implícito a la izquierda del punto, es imposible representar directamente el número cero. Para representarlo, se reservó el exponente 00000000 y se definió que la representación del número 0 es la secuencia con ese exponente y con 0s en el significante. Dada esta definición existen dos posibles 0s: $+0 = 00000000000000000000000000000000$ y $-0 = 10000000000000000000000000000000$
- El exponente 11111111 también se reservó, para poder representar ciertos números especiales:
  - +Infinito : 01111111100000000000000000000000
  - -Infinito : 11111111100000000000000000000000
  - NaN: not a number : $01111111xxxxxxxxxxxxxxxxxxxxxx$ donde alguno de los x debe cumplir con ser distinto de 0.

La representación «double» define un tamaño de 64 bits para los números, de los cuales se ocupa 1 bit para el signo del significante, 52 bits para el valor del significante y 11 bits para el exponente:

1 bit	11 bits	52 bits
signo significante	exponente	significante

Las características especiales de la representación «single» también aplican a esta, diferenciándose en que: la precisión total, contando el bit implícito es de 53 bits; el exponente está desfasado en 1023.

Aritmética de punto flotante

A diferencia de los números enteros y de la representación de punto fijo, un número representado como punto flotante requiere un manejo aritmético distinto. Veremos que es esta aritmética especial, en particular el caso de la suma y resta, una de las causas principales de los problemas de esta representación.

Multiplicación y división

Vamos a comenzar con la multiplicación y división que en punto flotante son operaciones más simples. Revisemos primero estas operaciones en notación científica, veremos que son transferibles los algoritmos a punto flotante.

Tomemos como ejemplo los números $1{,}2 \times 10^2$ y 2×10^{-1}. El algoritmo de multiplicación es el siguiente:

- El significante del resultado se obtiene como la multiplicación de los significantes de los multiplicandos: $1{,}2 \times 2 = 2{,}4$.
- El exponente del resultado se obtiene como la suma de los exponentes de los multiplicandos: $2 + (-1) = 1$.
- Resultado final: $2{,}4 \times 10^1$

La explicación del algoritmo es simple:

- Si realizamos la multiplicación directamente obtenemos: $1{,}2 \times 10^2 \times 2 \times 10{-}1$.
- Luego, si agrupamos los significantes y las potencias obtenemos: $1{,}2 \times 2 \times 10^2 \times 10^{-1}$

- El primer paso entonces era multiplicar los significantes: $2{,}4 \times 10^2 \times 10^{-1}$
- Ahora multiplicamos las potencias, y sabemos que la regla para multiplicar potencias con base igual es sumar los exponentes: $2{,}4 \times 10^1$

El algoritmo para los números de punto flotante es equivalente. Supongamos nuestra representación previa ($n = 8$, $s = 3$, $e = 3$) y los números $(1{,}1)_2 \times 2^{-(1)_2} = (0\mathbf{1101001})_{float}$ y $(1{,}0)_2 \times 2^{(1)_2} = (0\mathbf{1000001})_{float}$:

- El significante del resultado se obtiene como la multiplicación de los significantes de los multiplicandos: $(1{,}1)_2 \times (1)_2 = (1{,}1)_2$.
- El exponente del resultado se obtiene como la suma de los exponentes de los multiplicandos: $(1)_2 + (-1)_2 = 0$.
- Resultado final: $(1{,}1)_2 \times 2^0 = (0\mathbf{1100000})_{float}$

Suma y resta

Para sumar dos fracciones binarias representadas como punto fijo, basta ir sumando bit a bit de derecha a izquierda, acarreando cuando corresponda, lo que corresponde al mismo algoritmo que la suma de enteros. En el caso de punto flotante es distinto, ya que para poder sumar dos números deben tener el mismo exponente, lo que implica que en caso de que esto no se cumpla, debemos modificar los números para que si tengan el mismo exponente y puedan ser sumados o restados.

Veamos un ejemplo, primero con notación científica que involucra los mismos elementos aritméticos que el punto flotante: Tenemos dos números para sumar: $1{,}23 \times 10^2$ y $5{,}12 \times 10^{-1}$. Los pasos para completar la suma son los siguientes:

- Equilibrar los exponentes: debemos ajustar el menor de los números para que quede con el mismo exponente que el primero. Si restamos los exponentes tenemos una diferencia de 3, que es el número de veces que hay que mover la coma a la izquierda en el significante del menor número, resultando en: $0{,}00512 \times 10^2$"
- Una vez equilibrados los exponentes, se procede a sumar directamente los significantes: $1{,}23512 \times 10^2$

El algoritmo para los números de punto flotante es equivalente. Supongamos nuestra representación previa ($n = 8$, $s = 3$, $e = 3$) y los números $(1{,}1)_2 \times 2^{-(1)_2} = (0\mathbf{1101001})_{float}$ y $(1{,}0)_2 \times 2^{(1)_2} = (0\mathbf{1000001})_{float}$:

- Equilibrar los exponentes: debemos ajustar el menor de los números para que quede con el mismo exponente que el primero. Si restamos los exponentes tenemos una diferencia de 2, que es el número de veces que hay que mover la coma a la izquierda en el significante del menor número, resultando en: $(0{,}011)_2 \times 2^{(1)_2}$"
- Una vez equilibrados los exponentes, se procede a sumar directamente los significantes: $(1{,}011)_2 \times 2^{(1)_2}$

Tenemos un problema: el significante tiene precisión = 4 bits, y nuestro formato soporta hasta 3 bits. Inevitablemente tendremos que perder **exactitud**, por ejemplo podríamos truncar el último bit y obtener el número $(1{,}01)_2 \times 2^{(1)_2} = (0\mathbf{1010001})_{float}$. Este es uno de los problemas principales de la suma en punto flotante: a diferencia de la multiplicación (y la división), con la suma (y la resta) es muy fácil que perdamos exactitud al realizar una operación, por lo que es importante tener esto en cuenta al momento de realizar operaciones aritméticas con números de punto flotante.

Redondeo

En el ejemplo anterior, cuando obtuvimos como resultado el número $(1{,}011)_2 \times 2^{(1)_2} = 1{,}375$ notamos que dada la precisión de la representación era imposible almacenar toda la información, ya que debíamos eliminar un bit. Sin embargo, hay distintas formas en que podemos redondear el número de 4 a 3 bits, siendo algunas opciones mejores que otras.

El método más simple es el **redondeo hacia cero** que básicamente corresponde a truncar los bits que no caben en la representación. En el ejemplo anterior, aplicar este método resulta en $(1{,}01)_2 \times 2^{(1)_2} = 1{,}25$. El problema es que esta es la peor forma de redondeo, ya que introduce el mayor error y un sesgo hacia el cero.

Un mejor método se denomina **redondeo de la mitad a cero** que es básicamente el método que tradicionalmente ocupamos para redondear números decimales. Por ejemplo, si el número decimal 3,95 se debe representar en dos dígitos, se redondea a 4,0 dado que el 5 está a mitad de camino del valor de la base (10). En el caso de los números binarios, se aplica el mismo criterio: si el dígito está a mitad de camino o más de la base, se aumenta en 1 el dígito siguiente. En nuestro ejemplo, como el último dígito es 1 que es la mitad de la base (2), debemos aumentar en 1 el siguiente dígito, lo que resulta finalmente en el número: $(1{,}10)_2 \times 2^{(1)_2} = 1{,}5$. En este caso particular el error es el mismo que con el método anterior, pero en términos generales conviene realizar este redondeo, ya que se evita el sesgo de truncar hacia a cero siempre, lo que a la larga compensa en parte los errores.

3.2.3. Alternativas a la representación de punto flotante

Debido a los problemas de exactitud que pueden ocurrir con la representación de punto flotante, en muchas circunstancias se deben ocupar otro tipo de representaciones que permitan manejar de mejor forma números fraccionales.

Enteros

Una posible alternativa para manejar números fraccionales es tratarlos como enteros en otra unidad. La frase que resume esto es: «hacer cálculos monetarios directamente en centavos y no en dólares», es decir, si es posible, convertir los valores numéricos en la unidad más pequeña, de manera de siempre trabajar con enteros.

Esta alternativa tiene la ventaja de ser simple y no requerir tipos especiales, pero tiene el problema de que los números enteros no entregan tanto rango como los números de punto flotante: se puede pensar que un número entero es un número de punto fijo con el punto más allá del bit menos significativo. Dado esto los números enteros presentan las mismas limitaciones que los números de punto fijo y por tanto sólo conviene usarlos si no hay mejor alternativa.

Punto flotante con base decimal

Una alternativa mejor que los números enteros es usar representación de punto flotante, pero con base 10 en vez de base 2. Esta representación tiene la ventaja de que se eliminan los casos poco intuitivos en que una representación decimal finita (como el 0,1) tiene representacion infinita en binario. Al trabajar directamente en base 10, podemos representar 0,1 simplemente como: $(1{,}0)_2 \times 10^{(-1)_2}$, es decir el significante y exponente se almacenan en binario, pero al calcular el número completo se ocupa base decimal.

El estándar IEEE754 en su versión del 2008 especificó tres representaciones de punto flotante decimales: decimal32 (32 bits), decimal64 (64 bits), decimal128 (128 bits), las cuales son implementadas en muchos de los computadores modernos. En particular, el lenguaje C# provee el tipo de datos `decimal` el cual corresponde a la especificación decimal128.

La principal desventaja de esta representación es que hace mucho más lentos los cálculos, y es por eso que no es la representación principalmente usada. En esta representación multiplicar el significante por una potencia de la base **no** se traduce en sólo mover el punto, lo que complica la aritmética. Además, como el resto de los números manejados en el computador si tienen base 2, es necesario estar realizando conversiones para operar entre estos números y los de punto flotante decimal. De todas maneras, si estas representaciones están disponibles **siempre es recomendable utilizar este tipo de datos en aplicaciones que trabajen con números usados por seres humanos (como aplicaciones financieras)** para evitar problemas de exactitud.

Punto flotante con base decimal y precisión arbitraria

La representación de punto flotante decimal, aunque elimina los errores de representación de fracciones decimales como el 0,1 no elimina la limitación de espacio presente en todas las representaciones. Una mejor representación es la denominada de punto flotante decimal con precisión arbitraria, que va dinámicamente aumentando el espacio disponible para aumentar el significante, es decir, tiene precisión sólo limitada por el tamaño total de almacenamiento disponible en el computador. Aunque no existe soporte de hardware directo para este tipo de representaciones, algunos lenguajes de programación proveen clases que permiten trabajar con estos tipos. Por ejemplo en Java está la clase `BigDecimal` que permite trabajar con puntos flotantes decimales de precisión arbitraria.

La desventaja, al igual, que en el caso anterior, está en que las operaciones son más lentas. En este caso además de las conversiones decimal-binaria, se requiere ir aumentando dinámicamente el tamaño del significante lo que agrega un overhead adicional. Este tipo de representaciones conviene usarlo sólo en casos en que se requieran precisiones altísimas, como puede ser en cálculos científicos muy sofisticados.

4. Ejercicios

- Escriba un programa en Java que convierta un String de 1s y 0s, que representa un número binario, a un String que represente: un número decimal, un número hexadecimal.
- Describa el algoritmo para la división de números de notación científica y aplíquelo para restar números de punto flotante.
- Describa el algoritmo para la resta de números de notación científica y aplíquelo para restar números de punto flotante.

5. Referencias

- Morris Mano, M.; Computer System Architecture, 3 Ed., Prentice Hall, 1992. Capítulo 3: Representación de datos.
- The Floating Point Guide, http://floating-point-gui.de/
- Goldberg, D.; What Every Computer Scientist Should Know About Floating-Point Arithmetic, 1991, http://docs.sun.com/source/806-3568/ncg_goldberg.html
- Hyde, R. The Art of Assembly Language, 2003. Chapter 14: Floating Point Arithmetic http://webster.cs.ucr.edu/AoA/DOS/pdf/ch14.pdf

Apéndice: algoritmo de comparación de números de punto flotante

```
    public static boolean nearlyEqual(float a, float b, float epsilon)
    {
        final float absA = Math.abs(a);
        final float absB = Math.abs(b);
        final float diff = Math.abs(a - b);

        if (a * b == 0) { // a or b or both are zero
            // relative error is not meaningful here
            return diff < (epsilon * epsilon);
        } else { // use relative error
            return diff / (absA + absB) < epsilon;
        }
    }
```