DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN
ESCUELA DE INGENIERÍA
PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

IIC2343 - Arquitectura de Computadores

Representación de números racionales

1. Representación de números racionales

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 racionales en un computador. Sin embargo, la representación de este conjunto numérico presenta una serie de complicaciones y limitaciones que debemos entender para poder ocuparlas correctamente y evitar errores.

1.1. 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:

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

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:

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

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:

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

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

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

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 racionales, 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 racional: $0{,}1 = \frac{1}{10}$
- 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 primero paso corresponde a ver cuantas 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 divisón 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 primero paso corresponde a ver cuantas 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:

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

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.

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

- 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.

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

- Agregamos los 0s necesarios nuevamente para continuar el proceso

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

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, por que 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 racional 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.

1.2. 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 racionales, pero como se vio en la sección anterior es de particular importancia para los números racionales, que en muchos caso 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 racionales 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.

1.2.1. 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 racional, donde $n = t + f + 1$ (el bit extra se utiliza para almacenar el signo). De esta forma el punto (o coma) de la representación racional queda «fijo» en la t-ésima posición de la secuencia de bits.

Como ejemplo supongamos el número binario racional 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

1.2.2. 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 uso inicialmente, con un sólo 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 al rango, debemos reducir la precisión y vice-versa. La representación de punto flotante se prefiere por que 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 cuáles 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 dél 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 desafasa el número de manera de tener sólo valores positivos. La ventaja de esto está en hacer más simple la aritmética.
- Dado que se tiene 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 : 01111111$xxxxxxxxxxxxxxxxxxxxxxx$ 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 cuáles 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 queden 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 queden 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 podíamos 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 este 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 en este caso 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.

1.2.3. 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 racionales.

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 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 intui-

tivos 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 10 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 estás 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 eliminar 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 usarlos sólo en casos en que se requieran precisiones altísimas, como puede ser en cálculos científicos muy sofisticados.

2. Ejercicios

- Describa el algoritmo para la divisón 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.

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;
    }
}
```