Biblioteca Aritmética de operaciones en Tiempo Real para Números en Coma Flotante

Rafael Espí Botella
Higinio Mora Mora
Jerónimo Mora Pascual

Technical Report: I2RC-SPAL-2007-002

marzo 2007

Biblioteca Aritmética de operaciones en Tiempo Real para Números en Coma Flotante

by

Rafael Espí Botella
Higinio Mora Mora
Jerónimo Mora Pascual

Technical Report: I2RC-SPAL-2007-005

marzo 2007

Specialized Processors Architecture − Laboratory
Industrial Computer Science and Computer Networks
Department of Computer Science Technology and Computation
University of Alicante
Alicante, Spain

Abstract

En este informe se describen los algoritmos empleados para realizar operaciones aritméticas en tiempo real entre números que siguen el estándar IEEE754.

Index Terms: biblioteca de funciones aritméticas, IEEE754 de doble precisión, tiempo real

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF CODES

1. INTRODUCCIÓN

El estándar IEEE754 [1], [2], cuyo nombre completo es *IEEE Standard for Binary Floating-Point Arithmetic* (*ANSI/IEEE Std 754-1985*) también conocido por *IEC 60559:1989 Binary floating-point arithmetic for microprocessor systems*, es el formato para la representación de números en coma flotante de IEEE, ampliamente utilizado en representación numérica de números reales, y es seguida en muchas de las implementaciones de CPU. Este estándar determina una serie de formatos para la representación de números en coma flotante –dentro de esta representación se definen cinco casos especiales: 0.0, -0.0, Infinito, -Infinito y *NaN* (*not a number*), cuándo se producen y cómo han de ser operados— y valores desnormalizados junto con un conjunto de operaciones en punto flotante que opera sobre estos valores así como cuatro sistemas de redondeo (redondeo al par, hacia +infinito, hacia –infinito, hacia cero). Además, se definen cinco casos de excepción para las operaciones –*overflow, underflow*, división por cero, operación no válida, resultado no exacto— cuándo ocurren y cómo se tratan.

IEEE754 establece cuatro formatos para la representación de valores en coma flotante [1], [2]: precisión simple (32-bits), precisión doble (64-bits), precisión simple extendida (no muy difundida) y precisión doble extendida generalmente implementada con 80-bits). IEEE754 requiere únicamente valores de 32-bits, los otros son opcionales. Muchos lenguajes especifican que formatos y aritmética de la IEEE implementan, por ejemplo, en los lenguajes C/C++ y Java, el tipo *float* representa números en simple precisión y el tipo *double* representa números en doble precisión) y definen los operadores aritméticos básicos (suma, resta, producto y división) para operar con estos números. Sin embargo, estos lenguajes de alto nivel no permiten operar a nivel de bit con los números en punto flotante, por lo tanto, no se permite obtener el valor de un bit concreto ni aplicar operadores como el desplazamiento de bits.

El estándar IEEE754 se aprobó en marzo de 1985 [1] y se reafirmó en diciembre de 1990. Actualmente, el estándar está en revisión.

2. ESTÁNDAR IEEE754

2.1. Introducción

La obtención del valor numérico de un número normalizados según el estándar IEEE754 siguen la siguiente fórmula [1], [3]:

$$N = (-1)^s M B^E \quad (1)$$

donde *N* es el número a representar, *S* es el signo, *M* es la mantisa y *E* es el exponente.

El número de bits n que se va a utilizar para la representación de los números en punto flotante se obtiene a partir de la siguiente fórmula:

$$n = ns + ne + nm \quad (2)$$

donde *n* el número de bits totales para el número, *ns* el número de bits para el signo, *ne* el número de bits para el exponente y *nm* el número de bits para la mantisa.

Para el estándar IEEE754 de simple precisión, se dispone de n = 32 bits, donde:

$$ns = 1,\ ne = 8 \text{ y } nm = 23 \quad (3)$$

Para el estándar IEEE754 de doble precisión, se dispone de n = 64 bits, donde:

$$ns = 1,\ ne = 11 \text{ y } nm = 52 \quad (4)$$

Debido a que estamos trabajando con números normalizados, la mantisa se representa con la parte entera implícita en la representación. El exponente se representa de forma sesgada y el signo, valdrá 0 si el número es positivo, ó 1, si el número es negativo.

El estándar de IEEE 754 incluye además de los números positivos y negativos, los infinitos positivos y negativos, y el valor especial del *Not-a-Number* (*NaN*).

Estos casos especiales se representan en la siguiente tabla:

TABLA 1: CASOS ESPECIALES DE REPRESENTACIÓN

Número	**Signo**	**Exponente**	**Mantisa**
Infinito	0	111…1	0
Menos Infinito	1	111…1	0
Cero	0	0	0
Menos Cero	1	0	0
NaN	Cualquiera	111…1	<>0

Los valores de cero positivo y negativo se comparan igual, de modo que el resultado de la expresión 0.0==-0.0 es cierto y el resultado de 0.0>-0.0 es falso. Hay operaciones que pueden distinguir cero positivo y cero negativo, por ejemplo, el valor 1.0/0.0 es infinito positivo, mientras que el valor de 1.0/-0.0 es infinito negativo. Otro ejemplo es SQRT(0.0) = 0.0 y SQRT(-0.0)=-0.0.

Las excepciones como resultado de las operaciones se muestran en la siguiente tabla [10], [11], [12], [5]:

TABLA 2: EXCEPCIONES EN OPERACIONES

Excepción	**Causada por**	**Resultado**
Overflow	El resultado de la operación es un número que excede el rango de representación en su extremo superior	∞ ó $\pm N_{max}$
Underflow	El resultado de la operación es un número que excede el rango de representación en su extremo inferior	0 o $\pm N_{min}$
División por cero	$x/\pm 0.0$	$\pm\infty$
No válida	Operaciones no definidas	*NaN*
Inexacto	Resultados no exactos	ROUND(x)

Se utiliza el valor *NaN* para representar el resultado de ciertas operaciones inválidas tales como la raíz cuadrada de un número negativo. Cualquier operación en la que alguno de sus operandos es *NaN* se considera como una operación no válida. Cuando el resultado de aplicar una operación no válida es un valor en punto flotante, dicho resultado será siempre *NaN*.

Los valores representables en punto flotante se pueden ordenar, excepto el *NaN*, que es desordenado, por lo tanto, en el caso de aplicar un operador relacional (<, <=, >, ==) para una operación no válida, el valor de esa comparación siempre será *false*, por ese motivo, *NaN==NaN → false y NaN!=NaN → true.*

Cuando se produce un resultado inexacto, es necesario redondear dicho resultado. IEEE754 define cuatro modos de redondeo [1], [10], [11], [12], [5]:

- Redondeo hacia infinito positivo: El resultado se redondea por exceso hacia más infinito. Muchos lenguajes de programación proporcionan este tipo de redondeo mediante la función *ceil (x)*.

- Redondeo hacia infinito negativo: el resultado se redondea por defecto hacia menos infinito. Muchos lenguajes de programación proporcionan este tipo de redondeo mediante la función *floor (x).*
- Redondeo hacia cero: El resultado es un truncamiento del número, es decir, se desechan los bits de la mantisa. Muchos lenguajes de programación proporcionan este tipo de redondeo mediante la función *round (x).*
- Redondeo al más próximo: el resultado es el valor representable más próximo al valor que se quiere representar. Si se produce un empate, esta situación se resuelve mediante redondeo al par, es decir, sólo se le suma 1 a la mantisa si el valor del bit de menor peso o bit_0 es 1.

Una vez se ha redondeado la mantisa, se debe comprobar si se produce desbordamiento, en este caso, se vuelve a normalizar el número.

2.2. IEEE754 en JAVA

A. Tipos, formatos, y valores en Punto Flotante

Los tipos punto flotante son *float* y *double*, que se asocian con los valores IEEE754 de simple precisión de 32 bits y doble precisión de 64 bits y las operaciones especificadas en el estándar de IEEE para la aritmética en punto flotante binaria [4].

Las constantes de *NaN* de los tipos *float* y *double* se predefinen como *Float.NaN* y *Double.NaN.*

Cada elemento del sistema de representación de doble precisión es necesariamente también un elemento dentro del sistema de representación *double-extended-exponent*. El modo de representación *double-extended-exponent* tiene una mayor gama de valores para el exponente que el sistema de representación estándar correspondiente, pero no tiene más precisión.

Los elementos dentro del conjunto de valores *float* son exactamente los valores que se pueden representar usando el formato de punto flotante de simple precisión definido en el estándar de IEEE 754. Los elementos dentro del conjunto de valores *double* son exactamente los valores que se pueden representar usando el formato de punto flotante de doble precisión definido en el estándar de IEEE 754. Sin embargo, los elementos de los tipos *float-extended-exponent* y *double-extended-exponent* definidos en Java no corresponden con los valores que se pueden representar utilizando los formatos extendidos para simple y doble precisión definidos en IEEE 754.

TABLA 3: TIPOS DE PUNTO FLOTANTE EN JAVA

Parámetro	float	float-extended-exponent	double	double-extended-exponent
nm	24	24	53	53
ne	8	≥11	11	≥15
E_{max}	127	≥1023	1023	≥16383
E_{min}	-126	≤-1022	-1022	≤-16382

Mientras que cada arquitectura de hardware utiliza una configuración de bits particular para *NaN* cuando se genera un *NaN* nuevo, un programador puede también crear *NaNs* con diversas configuraciones de bits para codificar, por ejemplo, la información de diagnóstico retrospectiva. Generalmente, la plataforma Java trata los valores *NaN* mediante un único valor canónico (y por lo tanto esta especificación se refiere normalmente a un *NaN* arbitrario como si a un valor canónico). A partir de la versión 1.3, en la plataforma Java, se incluyen métodos que permiten al programador distinguir entre los valores *NaN*: los métodos *Float.floatToRawIntBits* y *Double.doubleToRawLongBits.*

B. Operaciones en punto flotante

El lenguaje de programación de Java proporciona los siguientes operadores:

- Los operadores de comparación, que dan lugar a un valor del tipo booleano: <, <=, >, >=, == y ¡=
- Los operadores numéricos, que dan lugar a un valor del tipo *float* o *double*:
- La suma y la resta unarios + y –
- Los operadores multiplicativos *, /, y %
- Los operadores aditivos + y –
- El operador de incremento ++, prefijo y posfijo
- El operador del decremento --, prefijo y posfijo
- El operador condicional ?:
- El operador de *cast* puede convertir de un valor en punto flotante a cualquier valor de cualquier tipo numérico.
- El operador de concatenación de cadenas +, que, dados un operando de tipo *String* y un operando en punto flotante, convertirá el operando en punto flotante a una cadena que representa su valor en forma decimal (sin la pérdida de precisión), y entonces genera una cadena nueva concatenando las dos cadenas.
- Existen otros constructores, métodos, y constantes útiles que se predefinen en las clases *Float, Double, y Math.*

Si por lo menos uno de los operandos de un operador binario es un número en punto flotante de doble precisión, se realiza la operación usando aritmética en punto flotante de 64 bits, y el resultado del operador numérico es un valor del tipo *double*. Si el otro operando no es un *double*, se produce una promoción de tipo hacia *double* para ese operando. Si no, se realiza la operación usando aritmética *floating-point* de 32 bits, si alguno de los operandos es número en punto flotante de simple precisión y el resultado del operador numérico es un valor del tipo *float*. Si el otro operando no es un *float*, se produce una promoción de tipo a *float* para ese operando. Los operadores en números en punto flotante se comportan según lo especificado por IEEE 754 (a excepción del operador *Remainder*.

Particularmente, el lenguaje de programación Java requiere el soporte de números en punto flotante IEEE 754 desnormalizados y de desbordamiento por *underflow* gradual, que hacen más fácil probar las características deseables de algoritmos numéricos particulares. Las operaciones en punto flotante "redirigen a cero" (*flush to zero*) si el resultado calculado es un número desnormalizado.

El lenguaje de programación de Java requiere que la aritmética en punto flotante se comporte como si cada operador de punto flotante redondeara su resultado en punto flotante a la precisión del resultado.

Los resultados inexactos se deben redondear al valor representable más cercano al resultado exacto, si los dos valores representables más cercanos están a la misma distancia, se escoge el par más cercano. Éste es el modo de redondeo del defecto del estándar de IEEE 754 conocido como redondo al más próximo.

El lenguaje Java utiliza el redondeo hacia cero para convertir un valor en punto flotante a un número entero, que actúa, en este caso, como si el número fuera truncado, desechando los dígitos de la mantisa. El redondeo hacia cero elige en su resultado el valor del formato más cercano y no mayor en a la magnitud que el resultado exacto.

Los operadores en punto flotante pueden lanzar una excepción *NullPointerException* si se aplica una conversión de tipo *unboxing* a una referencia nula. Los operadores en punto flotante de incremento ++ y de decremento -- pueden lanzar una excepción *OutOfMemoryError* si se requiere una conversión de tipo *boxing* y no hay suficiente memoria disponible para realizar la conversión.

Una operación donde se produce desbordamiento por *overflow* genera un infinito con signo, una operación en la que se produce desbordamiento por *underflow* genera un valor desnormalizado o un cero con signo, y una operación que no tenga ningún resultado matemáticamente definido produce *NaN*. Todas las operaciones numéricas con *NaN* como operando producen *NaN* consecuentemente. Como se describió anteriormente, *NaN* es desordenado, así que una operación numérica de comparación que implica uno o dos *NaN*s devuelve *false* y cualesquiera comparación ¡= que implica *NaN* devuelve *true*, incluyendo x!=x cuando x es *NaN*.

No obstante, a pesar de que *NaN* es no ordenable, al utilizar el método *compareTo* de las clases *Double* o *Float*, dicho método determina que *NaN* es mayor (1) que cualquier número en punto flotante distinto a NaN, y si los dos operandos son *NaN*, entonces *compareTo* determina que esos dos números son iguales (0).

3. TRANSFORMACIÓN DE FORMATOS

En el desarrollo de este apartado se utilizarán las siguientes siglas: *M* para la mantisa *S* para el signo, *Exp* para el exponente en representación sesgada y *exp* para el exponente sin el sesgo [1],[2],[3], [5].

Para transformar de decimal a IEEE754, se propone el siguiente algoritmo:

- Se fija el bit de signo S = 0, si el número es positivo, y en el otro caso, S = 1.
- Si el número es negativo, se convierte a positivo.
- Se calcula el exponente
  - Si el número es mayor o igual que 1.0: el exponente se calcula obteniendo el entero inferior más próximo de calcular la expresión exp = log2 N.
  - Si el número es menor que 1.0: El exponente se calcula obteniendo el entero inferior más próximo de aplicar la fórmula exp = – log2 (N).
- La mantisa la se calcula mediante: $M = N / 2^{exp}$
- Se obtiene el exponente en representación sesgada: E = exp + S

Nota: En este artículo, la base de los logaritmos siempre vale 2.

Se van a mostrar dos ejemplos:

Ejemplo 1: 5.75

Signo = 0
exp = log 5.75 = 2,321928094887362347870319429489 4 → exp = 2
M = 5.75 / 2^2 = 1,4375, sin el 1 implícito 0.4375
Exp = exp + Sesgo = 1025

Se comprueba el resultado:

(-1)^0 * 1.475 * 2^2 = 5.75

Se obtiene la siguiente representación:

0	10000000001	0111000000000000000000000000000000000000000000000000

Ejemplo 2: −0.875

Signo = 1
exp = – log 0.875 = –0,30102999566398119521373889472449 → –1
M = 0.875 / 2^(-1) = 1.75, quitándole el 1 implícito queda 0.75
Exp = exp + Sesgo = 1022

Se comprueba el resultado:

(-1)^1 * 1.75 * 2^(-1) = -0.875

Se obtiene la siguiente representación:

1	01111110	11000000000000000000000000000000000000000000000000000

4. OPERADORES

4.1. Suma

Para realizar la suma, se aplican los siguientes pasos [10], [11]:

- Se comprueba el exponente de los dos números a sumar.
- Si los exponentes son distintos, se igualan ambos exponentes, desplazando la mantisa del número que tenía el exponente menor a la derecha (añadiendo el 1 implícito) tantas posiciones como la resta del exponente mayor menos el menor.
- Si los números tienen distinto signo, para aplicar la suma, la mantisa del número negativo debe ser representada en complemento a 2. Además, el signo del resultado es el signo del número con mayor valor absoluto.
- A continuación, se suman las mantisas.
- Si los sumandos tienen distinto signo y el resultado es negativo, se vuelve a complementar la mantisa, desechando los bits que se generan a causa de realizar los complementarios.
- Se normaliza la mantisa, sumándole al exponente el número de desplazamientos a la derecha.

Se propone una mejora para el caso en el caso de que los números a sumar tienen distinto signo. La mejora consiste en que, en el momento de complementar la mantisa, se complementa la mantisa del número que tiene menor valor absoluto, de modo que se evita el caso en el que se obtiene un resultado cuya mantisa es negativa y debe ser complementada.

A continuación, se ilustran tres ejemplos significativos sobre la suma de dos números. Aunque la implementación se ha realizado sobre doble precisión, estos ejemplos se van a mostrar en simple precisión para mayor comodidad, pues los algoritmos se aplican del mismo modo.

Ejemplo 1

A= | 0 | 10000100 | 10111000000000000000000 |

B= | 1 | 01111111 | 01010000000000000000000 |

Como se puede observar, A es positivo, mientras que B es negativo. Lo primero es igualar los exponentes, al mayor de ellos y a continuación sumar las mantisas. Se halla cuantas posiciones hay que desplazar B, que es el operando que tiene el exponente menor, restando los exponentes. La mantisa de B se desplaza cinco posiciones a la derecha, y su valor es 00.00001010100000000000000

A continuación se complementa la mantisa de B en complemento a 2: 11.11110101100000000000000

Se calcula la suma de las mantisas como se muestra en el siguiente esquema:

```
  0 1.|1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
+ 1 1.|1 1 1 1 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
------|-----------------------------------------------
  0 1.|1 0 1 0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figura 1: Suma de mantisas del ejemplo 1

La mantisa resultante es 1.10101101100000000000000. Como el valor absoluto de A es mayor que el valor absoluto de B el signo es positivo → 0.

Finalmente la representación del resultado es la siguiente:

0	10000100	10101101100000000000000

Ejemplo 2

A= | 0 | 10000100 | 10111000000000000000000 |

B= | 0 | 01111111 | 01010000000000000000000 |

Como se puede observar, A y B son positivos. Lo primero es igualar los exponentes, al mayor de ellos, antes de sumar las mantisas. Se halla cuantas posiciones hay que desplazar B, que tiene el exponente menor, restando los exponentes. Se desplaza cinco posiciones a la derecha la mantisa de B: 00.00001010100000000000000

Se suman las mantisas tal como se muestra en la siguiente figura:

	0	1.	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+	0	0.	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	1.	1	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 2: Suma de mantisas del ejemplo 2

La mantisa resultado es 1.11000010100000000000000. Como se trata de una suma de dos números positivos, el signo es positivo→0.

Finalmente la representación es la siguiente:

0	10000100	11000010100000000000000

Ejemplo 3

Este caso va a ser analizado desde el punto de vista de la versión original del algoritmo y desde el punto de vista de la versión mejorada

A= | 1 | 10000100 | 10111000000000000000000 |

B= | 0 | 01111111 | 01010000000000000000000 |

Como se puede observar, A es negativo, mientras que B es positivo. Lo primero es igualar los exponentes, al mayor de ellos, para sumar las mantisas. Se halla cuantas posiciones hay que desplazar B, ya que es el operando con el exponente menor, restando los exponentes:

La mantisa de B se desplaza cinco posiciones a la derecha: 00.00001010100000000000000. Se complementa la mantisa de A, ya que es un número negativo: 11.01001000000000000000000.

Se suman las mantisas se muestra en la siguiente figura:

	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+	0	0.	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 3: Suma de mantisas del ejemplo 3

Como el resultado de la mantisa es un número negativo, es necesario complementar el resultado. La mantisa resultante de aplicar el complemento a 2 es 1.10101101100000000000000.

Debido a que los operandos tienen distinto signo y el sumando que tiene mayor valor absoluto es negativo, el signo del resultado será negativo → 1.

Finalmente:

1	10000100	10101101100000000000000

A continuación, se va a estudiar el ejemplo desde el punto de vista de la versión mejorada.

Se complementa la mantisa de B en C2: 11.11110101100000000000000

Se calcula la suma de las mantisas:

	0	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+	1	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	1	1	0	1	0	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 4: Suma de dos mantisas en la versión mejorada

La mantisa resultado es 1.10101101100000000000000. El valor absoluto de B es menor que el valor absoluto de A, por lo tanto el signo es negativo → 1. Finalmente el número resultado es el siguiente:

1	10000100	10101101100000000000000

Se observa que la segunda forma de llevar a cabo la suma es menos costosa que la primera. Por lo tanto, en lugar de una versión en la que se consideran tres posibles casos (suma de dos números con el mismo signo, suma de dos números con distinto signo que produce número positivo, suma de dos números de distinto signo que produce número negativo) se utiliza una versión más sencilla en la que se distinguen 2 casos (suma de dos números con el mismo signo y suma de dos números con distinto signo) realizando la comprobación de qué numero es menor en valor absoluto y aplicando el complemento a ese número.

Además, se tendrán en cuenta las siguientes excepciones:

TABLA 4: EXCEPCIONES PARA LA SUMA

Operador 1	Operador 2	Resultado
Infinito	Infinito	Infinito
Infinito	Menos Infinito	*NaN*
NaN	Cualquiera	*NaN*
Infinito	Caso no especial	Infinito
Menos Infinito	Caso no especial	Menos infinito
Cero	Cualquiera	Operador2

Otro aspecto a tener en cuenta es que si se suman dos números positivos o dos números negativos, se deben considerar los desbordamientos, en ese caso, se devuelve el máximo numero representable si el resultado es positivo, o el mínimo representable si es negativo.

4.2. Resta

En la expresión A – B = C, se transforma la resta en una suma aplicando un cambio de signo a B, es decir A + (–B) = C.

4.3. Multiplicación

Inicialmente, el valor del exponente del resultado es la suma del valor de los exponentes de los dos números:

$$\begin{aligned} E1 &= exp1 + S \\ E2 &= exp2 + S \\ Er &= exp1 + exp2 + S \end{aligned} \qquad (5)$$

Se multiplican las mantisas, añadiéndole el 1.0 implícito, y a continuación se normaliza la mantisa del número resultante, sumándole al exponente el número de desplazamientos a la derecha que se realizan en esta normalización.

Si los dos operandos tienen el mismo signo, el signo del resultado será positivo, en caso contrario, será negativo. Un ejemplo de multiplicación es el siguiente:

A= | 0 | 10000100 | 10111000000000000000000 |

B= | 1 | 01111111 | 01010000000000000000000 |

Primero, se calcula el exponente:

Exp = ExpA + ExpB – Sesgo = 10000100 + 01111111 – 01111111 = 10000100

A continuación, se multiplican las mantisas como se muestra en la siguiente figura:

```
                      1. 1 0 1 1 1
X                   - 1. 0 1 0 1
-------------------------------------
                      1  1 0 1 1 1 1
                   0  0  0 0 0 0
                1  1  0  1 1 1
             0  0  0  0  0 0
          1  1  0  1  1  1
-------------------------------------
- 1 0. 0 1 0 0  0 0 0 1 1
```

Figura 5: Suma de dos mantisas en la versión mejorada

El resultado obtenido tiene como mantisa el valor -10.010000011 y como exponente 10000100, por lo que dicho resultado debe ser normalizado: -1.0010000011 x 2e10000100.

Una vez calculada la multiplicación y normalizado el resultado, se debe procesar el signo, como A es positivo y B es negativo, el resultado del producto es un número negativo:

| 1 | 10000101 | 00100000110000000000000 |

Se consideran los casos especiales que muestra la siguiente tabla:

TABLA 5: EXCEPCIONES PARA LA SUMA

Operador 1	Operador 2	Resultado
Infinito	Infinito	Infinito
Infinito	Menos Infinito	Menos Infinito
Infinito	Cero	*NaN*
Menos Infinito	Cero	*NaN*
NaN	Cualquiera	*NaN*
Infinito	Caso no especial Positivo	Infinito
Infinito	Caso no especial Negativo	Menos infinito
Menos Infinito	Caso no especial Positivo	Menos infinito
Menos Infinito	Caso no especial Negativo	Infinito
Cero	Caso no especial	Cero

Como se trata de un producto, aplicando la propiedad conmutativa se obtiene el mismo resultado.

Otro aspecto a tener en cuenta es que al multiplicar dos números con valor absoluto mayor que 1.0, se deben tener en cuenta los desbordamientos, en ese caso, se devuelve el máximo numero representable si el resultado es positivo, o el mínimo representable si es negativo.

El producto de las mantisas se puede realizar a partir del algoritmo de las sumas sucesivas, en el que se dispone de un registro *multiplicando* para la mantisa del multiplicando y un registro *multiplicador* para la mantisa del multiplicador, y un registro *producto* con el doble de tamaño de la mantisa, donde *productoH* se refiere a la mitad izquierda y *productoL* se refiere a la mitad derecha:

```
multiplicando = mantisa del multiplicando
multiplicador = mantisa del multiplicador
n = número de bits de la mantisa + 1
producto = 0
Repetir n veces
        Si el bit 0 del multiplicador = 1 entonces
                productoH = productoH + multiplicando
        Fin si
       Desplazar un bit a la derecha el registro producto
       Desplazar un bit a la derecha el registro multiplicador.
Fin repetir
```

Algoritmo 1: Algoritmo para la multiplicación de mantisas (versión inicial).

Esta versión inicial puede ser mejorada como se muestra en el siguiente fragmento de código:

```
n = número de bits de la mantisa + 1
productoL = multiplicando
productoH = 0
Repetir n veces
        Si el bit 0 del multiplicador = 1 entonces
                productoH = productoH + multiplicando
        Fin si
       Desplazar un bit a la derecha el registro producto
Fin repetir
```

Algoritmo 2: Algoritmo para la multiplicación de mantisas (versión mejorada)

Una versión alternativa a este algoritmo es el algoritmo de *Booth* [6], que se basa en localizar las transiciones entre unos y ceros en el multiplicador, y aplicando sumas y restas según las transiciones, como indica la siguiente tabla:

TABLA 6: TRANSICIONES EN EL ALGORITMO DE BOOTH

Bit actual	Bit de la izquierda	Sustitución
0	0	0 (no hay transición)
0	1	-1 (transición hacia negativo)
1	0	+1 (transición hacia positivo)
1	1	0 (no hay transición)

Finalmente, se expone el pseudocódigo para el algoritmo de *Booth*:

```
n = número de bits de la mantisa + 1
qm1 = 0
Repetir n veces:
        Si q0 = 0 y qm1 = 0 entonces
                ProductoH = productoH + multiplicando
        Fin si
        Si q0 = 0 y qm1 = 1 entonces
                productoH = productoH – multiplicando
        Fin si
        Desplazamiento aritmético a la derecha de Producto y qm1
Fin repetir
```

Algoritmo 3: Algoritmo de Booth para el producto de mantisas

Nota: el término qm1 se refiere a q_{-1}, que se corresponde con el bit que hay a la derecha de q_0.

4.4. División

Inicialmente, el valor del exponente del resultado es la resta del valor del exponente del dividendo menos el exponente del divisor:

$$\begin{aligned} E1 &= exp1 + S \\ E2 &= exp2 + S \\ Er &= exp1 - exp2 + S \end{aligned} \qquad (6)$$

Se dividen las mantisas, añadiéndole el 1.0 implícito, y a continuación se normaliza la mantisa del número resultante, sumándole al exponente el número de desplazamientos a la derecha si el número es mayor que 1, o restándole al exponente los desplazamientos a la izquierda si el número es menor que 1, que se realizan en esta normalización. Si el signo de los dos números es igual, el resultado tiene signo positivo, en caso contrario, el signo es negativo.

Una forma alternativa de aplicar la división es realizar el producto del dividendo por el inverso del divisor.

A continuación, se muestra un ejemplo de división:

Primero debemos calcular el exponente

$$Exp = ExpA - ExpB + Sesgo = 10000100 - 01111111 + 01111111 = 10000100$$

A continuación, se dividen las mantisas, como se muestra en la siguiente figura:

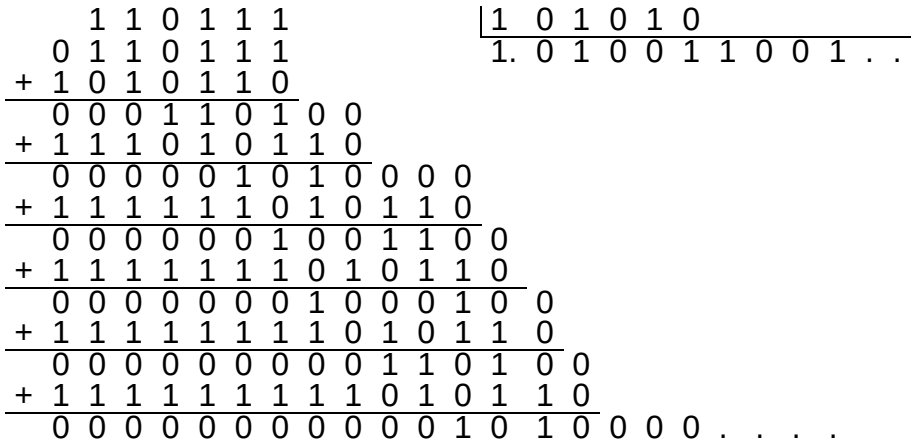

Figura 6: División de mantisas

El resultado que se obtiene de la división es un número periódico: 1.010011, por lo tanto, se debe ajustar la mantisa a 23 dígitos a partir de un modo de redondeo, en este ejemplo se utiliza el redondeo al más próximo:

01001100110011001100110 | 01100110 → 01001100110011001100110

Como se puede observar, los bits de guarda 011 no alcanzan el punto medio, por lo tanto, no hay que redondear. Se calcula el signo: como A es positivo y B es negativo, el signo del resultado es negativo → 1

El resultado final es:

1	10000100	01001100110011001100110

Los casos especiales se muestran en la siguiente tabla:

TABLA 7: EXCEPCIONES PARA LA DIVISIÓN

Operador 1	Operador 2	Resultado
Infinito / Menos Infinito	Infinito / Menos Infinito	*NaN*
NaN	Cualquiera	*NaN*
Infinito	Caso no especial Positivo o Cero	Infinito
Infinito	Caso no especial Negativo o Menos Cero	Menos infinito
Menos Infinito	Caso no especial Positivo o Cero	Menos infinito
Menos Infinito	Caso no especial Negativo o Menos Cero	Infinito
Cero	Cualquiera excepto *NaN* y Cero	Cero
Caso no especial	Cero/Menos Cero	Infinito/ Menos Infinito
Cero	Cero	*NaN*

Para realizar la división de las mantisas se proponen dos técnicas: la división por convergencia y la multiplicación por el recíproco (inverso).

La división por convergencia sigue la siguiente fórmula:

$$Q = \frac{N}{D} = \frac{N \cdot R_0 \cdot R_1 \cdots R_0}{D \cdot R_0 \cdot R_1 \cdots R_0} \rightarrow \frac{Q}{1} \qquad R_{i+1} = 2 - D_{i+1} \tag{7}$$

En la ecuación anterior, sólo se calcula el cociente, si se necesitara el resto, se calculará aparte.

Un paso fundamental para este método es escoger los factores correctos para asegurar la convergencia del denominador a 1. Debido que los números que forman parte de la división son las mantisas normalizadas, el resultado de la división está acotado en el rango]0.5, 2.0[.

La división por recíproco se calcula multiplicando la mantisa del dividendo por el inverso de la mantisa del denominador, como se muestra en el siguiente apartado. Se puede corregir el error generado al aplicar la aproximación de Newton-Raphson calculando el resto.

$$Q' = N \cdot \frac{1}{D} \tag{8}$$

$$R = N - Q \cdot D' \tag{9}$$

$$Q = Q' + \frac{R}{D} \tag{10}$$

4.5. Recíproco

Para calcular el inverso del número, se aplica la aproximación de *Newton-Raphson*, la cual se basa en la fórmula (11) que se aplica de forma iterativa [7], [8].

$$x = x * (2 - b * x) \tag{11}$$

En la fórmula anterior, *b* representa el valor de la mantisa del número, y en *x* se almacena el resultado parcial de calcular el inverso en cada iteración. El valor inicial de *x* será una semilla, la cual, para producir convergencia, deberá situarse a la izquierda del resultado esperado. Dicho valor inicial de la semilla será siempre 0.5 ya que el inverso de un número que pertenece al rango]1.0, 2.0[se encuentra acotado en el rango]0.5, 1[.

Se proponen los siguientes pasos para calcular el inverso:

- Si el valor de *Mr* es 1.0, significa que el número es potencia de 2, por lo tanto, para calcular el inverso, no es necesario iterar, será suficiente con modificar el número de entrada asignándole el opuesto de su exponente, es decir:

  $$\begin{aligned} &Mr = 1.0 \\ &Er = -exp + Sesgo \\ &Signor = signo \end{aligned} \tag{12}$$

- Si el valor de *Mr* es distinto de 1.0, entonces, se aplica la fórmula sobre la mantisa, y se le resta al exponente de x el exponente del número original. Se obtiene:

  $$\begin{aligned} &Mr = mantisa_x \\ &Er = exp_x - exp + Sesgo \\ &Signor = Signo \end{aligned} \tag{13}$$

A continuación, se ilustran dos ejemplos:

Ejemplo 1: Inverso de 5.0:

$5.0_{10} = 1.01_2$ * 2^2, es decir 1.25_{10} * 2^2, donde exp=2, Exp=2+Sesgo M=1.01 S=0
S_r =0
$exp\_r = exp_x - exp = -3$
x inicial = 0.1_2, es decir 0.5_{10}

Se itera como se ilustra en la siguiente tabla, y el valor de *x* se queda en

TABLA 8: TRAZA PARA EL INVERSO POR NEWTON-RAPHSON

Iteración	X
0	0,5000000000000000
1	0,6875000000000000
2	0,7841796875000000
3	0,7996871471405020
4	0,7999998776538600
5	0,7999999999999810
6	0,8000000000000000
7	0,8000000000000000
8	0,8000000000000000
9	0,8000000000000000
10	0,8000000000000000
11	0,8000000000000000

X = 0.8, donde $S_x = 0$, $exp_x = -1$ y $M_x = 1.6$

$S_r = 0$

$exp_r = -1 -2 = -3$

$M_r = 1.6$

Se comprueba el resultado: 0.2 = (-1)^0*2^(-3)* 1.6

Ejemplo 2: Inverso de 8

1.0 * 2^3 → 1.0 * 2^(-3) = 0.001 o 0.125.

Casos especiales:

TABLA 9: EXCEPCIONES PARA EL OPERADOR INVERSO

Operando	Resultado
Cero	Infinito
Menos Cero	Menos Infinito
NaN	*NaN*
Infinito	Cero
Menos Infinito	Menos Cero

4.6. Raíz cuadrada

Una opción para calcular la raíz cuadrada de un número es la aproximación por *Newton-Raphson*, que se basa en el siguiente algoritmo [9]:

```
Guess := N / 2
Loop
       Next := N / Guess
       If |Guess – Next| < Epsilon Then Exit
       Guess := (Guess + Next) / 2
EndLoop
```

Se aplica el algoritmo sobre el número desnormalizado, y una vez obtenido el resultado, se vuelve a normalizar.

Ejemplo: N = 9 → S = 0 E = 130 M = 1.125
N = 1001 → S = 0, E = 10000010 M= 1.001
En este caso, exp = 3, luego desplazamos la mantisa tres posiciones a la izquierda
M = 1.001 → 1001

El resultado de iterar se muestra en la siguiente tabla:

TABLA 10: TRAZA PARA LA RAÍZ CUADRADA POR NEWTON-RAPHSON

iteración	Next	Guess
0		4,5
1	2	3,25
2	2,76923077	3,00961538
3	2,99041534	3,00001536
4	2,99998464	3
5	3	3
6	3	3
7	3	3
8	3	3
9	3	3

Los casos especiales se muestran a continuación:

TABLA 11: EXCEPCIONES PARA LA RAÍZ CUADRADA POR NEWTON-RAPHSON

Operando	**Resultado**
Infinito	Infinito
NaN	*NaN*
Negativos	*NaN*
Menos Cero	Menos Cero

4.7. Valor absoluto

El valor absoluto de un número es su valor numérico sin tener en cuenta su signo. Por lo tanto, para hallar el valor absoluto de un número representado según el estándar IEEE754, se le asigna 0 al signo (positivo).

4.8. Opuesto

El opuesto de un número es ese número que tiene el mismo valor numérico y signo contrario. Para un número N representado según el estándar IEEE754, se obtiene el signo del número opuesto O, S_O a partir del complemento a 1 del valor del signo de N, S_N.

$$S_O = \text{Not } (S_N) \quad (14)$$

Otra forma de aplicar este cálculo es mediante los operadores módulo y AND:

$$S_O = (S_N + 1) \text{ MOD } 2 = (S_N + 1) \text{ AND } 1 \quad (15)$$

4.9. Exponencial

Para el cálculo de la función exp(x) se propone la siguiente aproximación:

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \rightarrow \frac{x^n}{n!} = \left(\frac{x^{n-1}}{(n-1)!} \right) \cdot \frac{x}{n} \tag{16}$$

A partir de la ecuación anterior, se extrae el siguiente algoritmo:

```
n := 0
oldsum := 0
newsum := 1.0
term := 1.0

while (newsum <> oldsum) do
   oldsum := newsum
   n := n + 1;
   term := term * x / n
   newsum := newsum + term
end
```

Algoritmo 5: Algoritmo para la función $exp(x)$ con números positivos

Si x es un número negativo, se produce la cancelación de algunos términos de la sucesión, (al sumar términos positivos y negativos), dando un resultado no correcto. Para solucionar este problema, modificaremos el algoritmo a partir de la siguiente propiedad:

$$\exp(x) = \frac{1}{\exp(-x)} \tag{17}$$

El algoritmo, una vez aplicada la corrección, es el siguiente:

```
x' = abs (x)
n := 0
oldsum := 0
newsum := 1.0
term := 1.0

while (newsum <> oldsum) do
   oldsum := newsum
   n := n + 1;
   term := term * x' / n
   newsum := newsum + term
end

if x < 0 then
   newsum = 1.0 / newsum
endif
```

Algoritmo 6: Algoritmo para la función $exp(x)$ con números positivos

4.10. Truncate

Este operador devuelve la parte natural de un número real, es decir, produce un truncamiento de la mantisa.

Para realizar el redondeo a entero por truncamiento se le aplica a la mantisa una máscara donde los bits que están a la derecha del bit de menor peso de la parte natural a 0 y el resto a 1. El número de bits de la mantisa que corresponde a la parte natural se obtiene a partir del exponente sin el sesgo.

Si el exponente sin el sesgo era negativo, no se produce el truncamiento de la mantisa, y se devuelve 0.0 directamente.

Los números a los que no se les aplica el redondeo son los casos especiales 0.0, -0.0, Infinito, -Infinito y NaN, por lo que si el operando a truncar es uno de estos números, se devolverá como resultado dicho operando.

4.11. Ceil

Este operador calcula el valor entero más pequeño para un número que no es menor que ese número.

Para realizar el redondeo a entero hacia infinito se le aplica a la mantisa el truncamiento si el exponente sin el sesgo es positivo como se ha explicado en el redondeo por truncamiento. Si al aplicar el truncamiento la mantisa ésta cambia su valor (es decir, existen bits que se han reseteado a 0), y el número es positivo, se le suma 1.0 a la mantisa.

En el caso en el que el exponente sin el sesgo es negativo, se examina el signo, y si el signo es positivo, se devuelve directamente 1.0, si es negativo se devuelve 0.0.

Los números a los que no se les aplica el redondeo son los casos especiales 0.0, -0.0, Infinito, -Infinito y *NaN*, por lo que si el operando a truncar es uno de estos números, se devolverá como resultado dicho operando.

4.11. Floor

Este operador calcula el valor entero más grande para un número que no es mayor que ese número.

Para realizar el redondeo a entero hacia menos infinito se le aplica a la mantisa el truncamiento si el exponente sin el sesgo es positivo como se ha explicado en el redondeo por truncamiento. Si al aplicar el truncamiento la mantisa ésta cambia su valor (es decir, existen bits que se han reseteado a 0), y el número es positivo, se le suma 1.0 a la mantisa.

En el caso en el que el exponente sin el sesgo es negativo, se examina el signo, y si el signo es negativo, se devuelve directamente 1.0, si es positivo se devuelve 0.0.

Los números a los que no se les aplica el redondeo son los casos especiales 0.0, -0.0, Infinito, -Infinito y *NaN*, por lo que si el operando a truncar es uno de estos números, se devolverá como resultado dicho operando.

4.12. Round

Este operador calcula el valor entero más próximo para un número real.

Para realizar el redondeo al entero más próximo se le aplica a la mantisa el truncamiento si el exponente sin el sesgo es positivo como se ha explicado en el redondeo por truncamiento. Si al aplicar el truncamiento la mantisa ésta cambia su valor (es decir, existen bits que se han reseteado a 0), y el número es positivo, se le suma 1.0 a la mantisa.

En el caso en el que el exponente sin el sesgo es negativo, si éste vale -1 significa que el número en valor absoluto es mayor o igual que 0.5, y se devuelve directamente ± 1.0. Si el exponente sin signo es negativo menor que -1, se devuelve directamente 0.0.

Los números a los que no se les aplica el redondeo son los casos especiales 0.0, -0.0, Infinito, -Infinito y *NaN*, por lo que si el operando a truncar es uno de estos números, se devolverá como resultado dicho operando.

4.13. Escalado

El operador de escalado por potencia positiva de 2 multiplica el número que se le pasa como operando por una potencia de 2 con exponente entero positivo. Para ello, se le suma el exponente de esa potencia de 2 al exponente del número.

El operador de escalado por potencia negativa de 2 multiplica el número que se le pasa como operando por una potencia de 2 con exponente entero negativo. Para ello, se le resta el exponente de esa potencia de 2 al exponente del número.

Se debe tener en cuenta los desbordamientos por overflow, underflow, y el tratamiento de los números renormalizados.

Al aplicar el escalado positivo se puede producir un desbordamiento por overflow, por lo tanto, se comprueba el exponente del valor de salida, y si es mayor de 0x7ff entonces se devuelve infinito con signo.

Al aplicar el escalado positivo por potencia de 2 con exponente *exp* a un número renormalizado, primero se procesa la mantisa desplazándola a la izquierda *dsp* posiciones, hasta que la mantisa alcance el valor 1.0 o (*exp – dsp)* valga 0, y a continuación se le suma al exponente del valor resultante la cantidad *(exp – dsp)*.

Al aplicar escalado negativo se puede generar un número renormalizado o producirse *underflow*, en ese caso, se debe poner el exponente a 0, y desplazar la mantisa a la derecha, obteniéndose 0.0 cuando se produce *underflow*.

5. SIMULACIÓN DE TIEMPO REAL

Se considera conveniente introducir el concepto de Tiempo Real, para ello se ha escogido las siguientes tres definiciones [7]:

- **Oxford Dictionary of Computing:** "Cualquier sistema en el que el tiempo de procesamiento es significativo, porque usualmente la entrada corresponde con algún cambio del mundo real y la salida debe estar relacionada con ese cambio. El tiempo de operación debe ser lo suficientemente pequeño para cumplir de forma aceptable con la restricción temporal"
- **Burns and Wellings (2001):** "Cualquier actividad de procesamiento de información o sistema que tiene que responder a un estímulo generado externamente en un tiempo finito y determinado"
- **Laplante (1993):** "Sistema que tiene que satisfacer explícitamente las restricciones temporales establecidas o arriesgarse a sufrir consecuencias que incluyen fallos"

El común denominador en un sistema de tiempo real es la *predecibilidad*. Los sistemas de tiempo real deben ser deterministas, esto es, con ciertas asunciones sobre la carga del sistema y sus anomalías, debe ser posible comprobar durante el *diseño del sistema q*ue todas las restricciones temporales de la aplicación se cumplen.

El determinismo es fácil de asegurar si se usa una arquitectura muy sencilla.

Determinar el tiempo máximo de ejecución de una tarea se reduce a contar el número de ciclos que tardan las instrucciones en ejecutar la ruta más larga y tener en cuenta posibles retrasos por el servicio de interrupciones.

Una forma de aplicar restricciones de tiempo real a un sistema consiste en aplicar diferentes tipos de precisión a los datos de entrada, a los resultados parciales y a los finales. Dependiendo de las restricciones temporales del sistema, los cálculos se realizan aplicando una serie de etapas, en las que se le va añadiendo precisión a los resultados obtenidos. Para cada precisión concreta se conocen las cotas de tiempo y error cometidas. Dependiendo de las restricciones impuestas, la calidad del resultado es crítica, por lo que es necesario tener en cuenta cómo afecta las pérdidas de precisión de los resultados parciales a los resultados totales. Por lo tanto, es indispensable tener también acotado el error para operaciones concatenadas.

Para simular una arquitectura que permita aplicar operadores para tiempo real utilizando números de doble precisión según el estándar IEEE754, se propone aplicar 6 etapas con las siguientes precisiones: 12, 20, 28, 36, 44, 52 bits, a las que se les aplica las siguientes máscaras a la mantisa, dependiendo de la cantidad de etapas que el sistema determine que deben aplicarse: 0xFFFFFF0000000000, 0xFFFFFFFF00000000, 0xFFFFFFFFFF000000, 0xFFFFFFFFFFFF0000, 0xFFFFFFFFFFFFFF00, 0xFFFFFFFFFFFFFFFF}; respectivamente.

Como se ha indicado anteriormente, el tiempo se expresará en función del número de puertas lógicas empleado.

Para ello, indicaremos, inicialmente, cuáles son los costes en puertas lógicas para cada operador dependiendo del número de etapas:

TABLA 12: NÚMERO DE PUERTAS LÓGICAS POR OPERACIÓN Y POR ETAPAS

Operación/Etapas	***1***	***2***	***3***	***4***	***5***	***6***
Suma	3,5	4,5	5,5	6,5	7,5	8,5
Multiplicación	6,5	9,5	12,5	15,5	18,5	21,5
División	24	56	88	120	152	184
Raíz	25,5	74,5	124	180	225	270

A continuación, analizaremos como ejemplo el cálculo del coeficiente de variación.

Coeficiente de variación. Esta medida se define como:

$$d = \frac{s}{\bar{x}} \tag{18}$$

La desviación estándar se define como:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2} \tag{19}$$

La varianza se define como:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2 \tag{20}$$

El promedio se define como:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x \tag{21}$$

Una vez se han ilustrado las fórmulas utilizadas, se procede siguiendo una estrategia divide y vencerás para deducir el número de operaciones. Para ello, inicialmente se resuelven los cálculos más sencillos, y a continuación se va ascendiendo en complejidad.

En el caso del promedio: n sumas + 1 división

En el caso de la varianza, aplicando loop-invariant code motion, calculamos el promedio solo una vez al ser su cálculo independiente del bucle donde se realiza el cuadrado de la resta, entonces, Operaciones_promedio * 1 + n restas + n productos + n sumas + 1 división = (n sumas + 1 división) + n restas + n productos + n sumas+ 1 división = 3n (sumas / restas) + 2 división + n productos

Para la desviación típica: operaciones_varianza + 1 raiz_cuadrada = 3n (sumas / restas) + 2 división + n productos + 1 raiz_cuadrada

Para el coeficiente de variación: operaciones_DesvTipica + operaciones_promedio + 1 división = (3n sumas / restas + 2 divisiones + n productos + 1 raiz_cuadrada) + (n sumas + 1 división) + 1 división = 4n sumas / restas + 4 divisiones + n productos + 1 raíz cuadrada

Una vez expuestos cuáles son los costes en puertas para los operadores según el número de etapas, se indica cuáles son los costes del algoritmo en función del número de etapas.

Como se ha indicado anteriormente, se calculan los costes en puertas lógicas siguiendo esta fórmula: 4n sumas / restas + 4 divisiones + n productos + 1 raíz cuadrada

Para una muestra de 10000 elementos, se realizan las siguientes operaciones:

TABLA 13: NÚMERO DE OPERACIONES PARA EL C.V

Operando	Resultado
Suma	40000
Multiplicación	10000
División	4
Raíz	1

A partir de la fórmula anterior, se obtiene la siguiente tabla de costes:

TABLA 14: NÚMERO DE PUERTAS LÓGICAS PARA EL C.V

Etapas	***1***	***2***	***3***	***4***	***5***	***6***
Puertas	205121.5	275298.5	345476	415660	485833	556006

6. IMPLEMENTACIÓN

Se ha creado la clase *DoublePrecNumber* dentro del paquete *ieee754_Operator*, donde se implementan la codificación numérica, la transformación de formatos y los operadores.

```
/**
<P><FONT COLOR="blue">
 *<B>DoublePrecNumber:</B> Esta clase simula la aritmetica para el estandar IEEE754.<BR>
 *<B>Descripcion</B><BR>
 *<FONT COLOR="blue">
 *DoublePrecNumber es una clase que representa a los numeros de doble precision
siguiendo
 *el estandar IEEE754. Dicho formato consta de Signo (1bit) Exponente (11bits)
Mantisa(52 bits).
 *El exponente se representa en notacion sesgada, siendo este sesgo de 1023 = 2**(11 -
1) - 1.
 *Se supondra la mantisa normalizada, esto es, el 1. va implïcito.
 *En dicha clase se incorporan los operadores aritmeticos tradicionales, como la suma,
 *la multiplicacion o la raiz cuadrada.
 *El concepto de Tiempo Real influira en las operaciones en cuanto a su precision.
*/

package ieee754_Operator;

public class DoublePrecNumber implements Comparable, Cloneable
{
…….
}
```

Código 1. Clase DoublePrecNumber

6.1. Codificación de los números

El signo y el entero se representan mediante un entero (*int*) de 32 bits, mientras que la mantisa se representan con un entero largo (*long*) de 64 bits.

```
/**Mantisa (52 bits) para el numero IEE754 de doble precision*/
protected long mantisa;

/**Exponente (11) bits para el numero IEE754 de doble precision*/
```

```
protected int exponente;

/**Signo (1 bit) para el numero IEE754 de doble precision*/
protected int signo;

/**SESGO para doble precision en IEEE754 = 2^10-1 */
public static final int kSESGO = 0x3FF;

/**Representa el valor limite para la mantisa en IEEE754 con doble precision*/
protected final static long kMANTISA = (1l << 52);

/**Representa los 52 bits que tiene la mantisa*/
protected final static long kBITSMANTISA = 0xFFFFFFFFFFFFFl;
```

Código 2: Representación del número

Además, se han agregado constantes para el sesgo, el valor de mantisa para el 1.0 y 1.9 y para los casos especiales:

```
/**Representa el valor NaN en el estandar IEEE754 con doble precision*/
public final static DoublePrecNumber NaN = new DoublePrecNumber(0, 0x7ff, (long)1 <<
52);
/**Representa el valor Infinito en el estandar IEEE754 con doble precision*/
public final static DoublePrecNumber Infinito = new DoublePrecNumber (0, 0x7ff, 0);
/**Representa el valor -Infinito en el estandar IEEE754 con doble precision*/
public final static DoublePrecNumber MenosInfinito = new DoublePrecNumber (1, 0x7ff, 0);

/**Representa el valor 0.0 en el estandar IEEE754 con doble precision*/
public final static DoublePrecNumber Cero = new DoublePrecNumber (0, 0, 0);

/**Representa el valor -0.0 en el estandar IEEE754 con doble precision*/
public final static DoublePrecNumber MenosCero = new DoublePrecNumber (1, 0, 0);

/**Representa el valor maximo representable en el estandar IEEE754 con doble precision*/
public final static DoublePrecNumber MaxDoublePrecNumber = new DoublePrecNumber
(0,0x7fe, DoublePrecNumber.kBITSMANTISA);

/**Representa el valor minimo representable en el estandar IEEE754 con doble precision*/
public final static DoublePrecNumber MinDoublePrecNumber = new DoublePrecNumber (0, 0,
2);
```

Código 3: Casos especiales de representación numérica

6.2. Implementación de conversión de formatos

Al implementar el algoritmo citado anteriormente, se deben analizar primero los casos especiales, y si se trata de un número "normal", entonces, se aplica el algoritmo

```
/**Constructor por defecto.
 * Crea un DoublePrecNumber a partir de un numero de tipo double.
 *@param num Numero double
 */
public DoublePrecNumber (double num)
{

	if (Double.isNaN(num))
	{
		this.mantisa = DoublePrecNumber.NaN.mantisa;
		this.signo = DoublePrecNumber.NaN.signo;
		this.exponente = DoublePrecNumber.NaN.exponente;
	}
	else if (num == Double.NEGATIVE_INFINITY)
	{
		this.mantisa = DoublePrecNumber.MenosInfinito.mantisa;
		this.signo = DoublePrecNumber.MenosInfinito.signo;
		this.exponente = DoublePrecNumber.MenosInfinito.exponente;
	}
```

```
        else if (num == Double.POSITIVE_INFINITY)
        {
                this.mantisa = DoublePrecNumber.Infinito.mantisa;
                this.signo = DoublePrecNumber.Infinito.signo;
                this.exponente = DoublePrecNumber.Infinito.exponente;
        }
        else if (num == 0)
        {
                this.exponente = 0;
                this.mantisa = 0;
                if (1.0/num == Double.POSITIVE_INFINITY) //DEBEMOS HACERLO ASI PORQUE
0.0==-0.0 ES TRUE
                        this.signo = 0;
                else
                        this.signo = 1;

        }

        //TRATAMOS LOS NUMEROS "NORMALES"
        else
        {
                //TRATAMOS EL SIGNO. SI ES NEGATIVO, PONEMOS EL SIGNO A NEGATIVO
                // Y EL NUMERO A POSITIVO
                if (num >= 0)
                        signo = 0;
                else
                {
                        signo = 1;
                        num = -num;
                }

                if (num > 1)
                {

                        exponente = (int) (Math.log (num) / Math.log (2));
                        double  base_exp = Math.pow(2, exponente);
                        double mantisa_double = (num) / base_exp;
                        //PUEDE HABER ERRORES DE REDONDEO QUE HAGAN LA MANTISA NEGATIVA

                        if (mantisa_double < 0)
                                mantisa_double= 0;
                        mantisa =  (long) (mantisa_double * DoublePrecNumber.kMANTISA);
                        exponente = exponente + DoublePrecNumber.kSESGO;
                }
                else //VALORES MENORES QUE 1.0
                {

                        exponente = -(int) Math.ceil (-Math.log (num) / Math.log (2));
                        if (exponente >= -1022) //VALORES NORMALIZADOS
                        {
                                mantisa = (long) (((num * Math.pow(2, -exponente)) *
kMANTISA));
                                exponente = exponente + DoublePrecNumber.kSESGO;
                        }
                        else // VALORES DESNORMALIZADOS
                        {
                                mantisa = (long) (num /Math.pow(2.0, -1023) *
DoublePrecNumber.kMANTISA);
                                exponente = 0;
                        }
                }
        }
}
```

Código 4: Conversión de formato

La expresión *M = N / 2^exp* no se debe implementar mediante un desplazamiento a la izquierda con la idea de que dará mayor rendimiento ya que esta operación producirá desbordamientos cuando el desplazamiento sea mayor de 64.

Además, se dispone de un método para generar el número en su representación decimal y binaria para posteriormente mostrarlo por pantalla.

Este método básicamente, lo que hace es utilizar el método de clase de la clase *Long toBinaryString*, que devuelve la representación de ese *Long* en una cadena de 0s y 1s. Será

necesario completar esa cadena con 0s a la izquierda para obtener el número de bits que nos dicta el estándar.

Para obtener el número en su representación decimal, básicamente, se sigue el procedimiento inverso a la transformación en *Signo-Exponente-Mantisa*.

En el caso de los números especiales, se genera la cadena de salida directamente.

```
/**Imprime en un String el nmero en este formato: Double: Signo Exponente Mantisa
 *@return String siguiendo el formato Double: Signo Exponente Mantisa*/

public String toString ()
{
	//TRATAMOS LOS CASOS ESPECIALES
	if (this.compareTo(DoublePrecNumber.Infinito) == 0)
		return "Infinito" + ": " + signo + " " +
Integer.toBinaryString(exponente) + " " +
"00000000000000000000000000000000000000000000000000000000000000000";
	else if (this.compareTo(DoublePrecNumber.MenosInfinito) == 0)
		return "Menos Infinito" + ": " + signo + " " +
Integer.toBinaryString(exponente) + " " +
"00000000000000000000000000000000000000000000000000000000000000000";
	else if (this.compareTo(DoublePrecNumber.NaN) == 0)
		return "NaN" + ": " + signo + " " + Integer.toBinaryString(exponente) + "
" + Long.toBinaryString(mantisa);
	else if (this.exponente == 0 &&  this.mantisa == 0 && this.signo == 0)
		return "0.0" + ": " + signo + " " + "00000000000" + " " +
"00000000000000000000000000000000000000000000000000000000000000000";
	else if (this.exponente == 0 &&  this.mantisa == 0 && this.signo == 1)
		return "-0.0" + ": " + signo +  " " + "00000000000" + " " +
"00000000000000000000000000000000000000000000000000000000000000000";

	String temp;
	temp = Integer.toBinaryString (this.exponente);

	//GENERAMOS CADENA DE BITS PARA EL EXPONENTE
	//RELLENANDO CON CEROS A LA IZQUIERDA HASTA TENER 11 BITS
	String exponente = "";
	for (int i = temp.length (); i < 11; i++)
		exponente = exponente.concat ("0");
	exponente = exponente.concat (temp);

	//GENERAMOS CADENA DE BITS PARA LA MANTISA,
	//RELLENAMOS CON CEROS A LA IZQUIERDA, HASTA TENER 52 BITS
	temp = Long.toBinaryString (this.mantisa);

	String mantisa = "";

	for (int i = temp.length (); i < 52; i++)
		mantisa = mantisa.concat ("0");
	mantisa = mantisa.concat (temp);
	if (this.mantisa >= DoublePrecNumber.kMANTISA)
		mantisa = mantisa.charAt(0) + "." + mantisa.substring(1);
	else
		mantisa = "0." + mantisa;

	return this.NumeroReal() + ": " + signo + " " + exponente + " " + mantisa;

}
```

Código 5: Conversión a cadena de texto

6.3. Implementación del operador Suma

Inicialmente, se comprueban los casos especiales, y si es así, se devuelve el valor directamente. También se devuelve directamente el valor cuando alguno de los operandos es cero. Si se trata de una suma de números normales, se aplica el algoritmo de la suma citado anteriormente. Finalmente, una vez hallado el resultado, se comprueba si se ha producido desbordamiento, en ese caso, se devuelve el máximo o el mínimo representable, según sea positivo o negativo.

En este caso, se operan las mantisas en tipo *long*, ya que permite aplicar los desplazamientos, que consumen menos tiempo que otros operadores. En este caso, aunque las mantisas están multiplicadas por una constante, se cumple que:

```
mantisa1 * kMANTISA + mantisa2 * kMANTISA = (mantisa1 + mantisa2) * kMANTISA
```

Para transformar la mantisa de entero largo (*long*) a número real (*double*) es suficiente con dividir la mantisa por esa constante.

```
/**Funcion de SUMA.
 * Obtiene el resultado de sumar dos numeros de doble
 * precision, aplicando un operador suma.
 * @param n2 segundo sumando, numero de doble precision en formato IEEE754
 * @return Resultado de la resta en formato de numero de doble precision IEEE754
 */

public DoublePrecNumber Suma (DoublePrecNumber n2)
{

	//TRATAMOS LOS CASOS ESPECIALES
	//UNO DE LOS SUMANDOS VALE 0
	if (this.exponente == 0 && this.mantisa == 0)
		return new DoublePrecNumber(n2);
	else  if (n2.exponente == 0 && n2.mantisa == 0)
		return new DoublePrecNumber(this);
	//SE SUMAN DOS NUMEROS OPUESTOS, RESULTADO VALE 0
	else if (this.exponente == n2.exponente && this.mantisa == n2.mantisa &&
this.signo != n2.signo && this.exponente < 0x3ff)
		return new DoublePrecNumber(DoublePrecNumber.Cero);
	//ALGUNO DE LOS OPERANDOS ES NaN
	else if (this.exponente == 0x7ff && this.mantisa != 0 || n2.exponente == 0x7ff &&
n2.mantisa != 0)
		return new DoublePrecNumber(DoublePrecNumber.NaN);
	else if (this.exponente == 0x7ff && this.mantisa == 0 && this.signo == 0)
	{
		if (n2.exponente == 0x7ff && n2.mantisa == 0 && n2.signo == 1)
			return new DoublePrecNumber(DoublePrecNumber.NaN);
		else
			return new DoublePrecNumber(DoublePrecNumber.Infinito);
	}
	else if (this.exponente == 0x7ff && this.mantisa == 0 && this.signo == 1)
	{
		if (n2.exponente == 0x7ff && n2.mantisa == 0 && n2.signo == 0)
			return new DoublePrecNumber(DoublePrecNumber.NaN);
		else
			return new DoublePrecNumber(DoublePrecNumber.MenosInfinito);
	}
	else if (n2.exponente == 0x7ff && n2.mantisa == 0 && n2.signo == 0)
	{
		if (this.exponente == 0x7ff && this.mantisa == 0 && this.signo == 1)
			return new DoublePrecNumber(DoublePrecNumber.NaN);
		else
			return new DoublePrecNumber(DoublePrecNumber.Infinito);
	}
	else if (n2.exponente == 0x7ff && n2.mantisa == 0 && n2.signo == 1)
	{
		if (this.exponente == 0x7ff && this.mantisa == 0 && this.signo == 0)
			return new DoublePrecNumber(DoublePrecNumber.NaN);
		else
			return new DoublePrecNumber(DoublePrecNumber.MenosInfinito);
	}


	int exp1 = this.exponente;
	long mantisa1 = (this.mantisa) << 8;
	int signo1 = this.signo;

	int exp2 = n2.exponente;
	long mantisa2 = (n2.mantisa) << 8;
	int signo2 = n2.signo;


	//AJUSTAMOS LOS EXPONENTES Y LAS MANTISAS
	//ANYADIMOS A LAS MANTISAS EL 1.0 QUE IEEE754 TIENE IMPLICITO
	//SI EL PRIMER OPERANDO ES MAYOR QUE EL SEGUNDO, DESPLAZAMOS LA MANTISA DEL
SEGUNDO A LA DCHA
	//SI EL SEGUNDO ES MAYOR QUE EL PRIMERO, DESPLAZAMOS LA MANTISA DEL PRIMERO A LA
DERECHA
```

```
	if (exp1 > exp2)
	{
		int dsp = exp1-exp2;
		if (dsp < 60)
		{
			mantisa2 = (mantisa2) >> dsp;
		}
		else
		{
			mantisa2 = 1;
		}

		exp2 = exp1;
	}
	else if (exp1 < exp2)
	{
		int dsp = exp2 - exp1;
		if (dsp < 60)
		{
			mantisa1 = (mantisa1) >> dsp;
		}

		else
		{
			mantisa1 = 1;//SETS STICKY BIT TO 1
		}

		exp1 = exp2;
	}

	if (signo1 != signo2) //SI LOS OPERANDOS TIENEN DISTINTO SIGNO
	{
		if (mantisa2>mantisa1)
			mantisa1 = -mantisa1;
		else
			mantisa2 = -mantisa2;
	}


	mantisa1 = (mantisa1 + mantisa2);


	//TRATAMOS LOS ACARREOS DE LA MANTISA
	if (signo1 != signo2) //SI LOS OPERANDOS TIENEN DISTINTO SIGNO
	{
		if (this.exponente < n2.exponente || (this.exponente == n2.exponente &&
this.mantisa < n2.mantisa ))
			signo1 = signo2;
		int dsp = 0;
		while ( (mantisa1<<dsp) < DoublePrecNumber.kMANTISA << 8)dsp++;
		mantisa1 = (mantisa1 << dsp);

		long GRS = ((mantisa1) & 0xFF);
		mantisa1>>=8;
		if (GRS > 128) mantisa1++;
		if (GRS == 128 && (mantisa1 & 0x1) == 1) mantisa1++;

		exp1 -= dsp;

	}

	else
	{

		int dsp = (int) (mantisa1 >> 61) & 0x1;
		exp1 += dsp;
		long GRS = ((mantisa1>>(dsp)) & 0xFF);
		mantisa1 = (mantisa1 >> (8+dsp));
		if (GRS > 128)mantisa1++;
		if (GRS == 128 && (mantisa1 & 0x1) == 1) mantisa1++;

	}

	//OVERFLOW
	if (exp1 >= 0x7ff)
	{
		if (signo1 == 0)
```

```
                        return new DoublePrecNumber (DoublePrecNumber.Infinito);
                else
                        return new DoublePrecNumber (DoublePrecNumber.MenosInfinito);
        }
        //DENORMALIZADO
        if (exp1 < 0)
        {
                return new DoublePrecNumber (signo1, 0, mantisa1 >> -exp1);
        }

        return new DoublePrecNumber (signo1, exp1, mantisa1);
}
```

Código 6: Suma

Inicialmente, el desplazamiento de la mantisa del número con menor valor absoluto se implementó mediante el operador >>, sin embargo, cuando el desplazamiento es muy grande (uno de los operandos tiene un valor absoluto muy grande, mientras que el otro tiene un valor absoluto muy pequeño), el desplazamiento a la derecha es muy grande, y se produce un desbordamiento, por lo tanto, se decidió implementar como se observa en el código.

6.4. Implementación del operador Resta

Se plantean dos alternativas para implementar la resta. La primera consiste en realizar la implementación a partir de la suma, pasándole como operandos el minuendo y el operando del sustrayendo. La segunda consiste en adaptar el código de la suma para la resta. La segunda opción es más conveniente que la primera porque la creación de objetos y las llamadas a funciones consumen mucho tiempo, por eso, se ha replicado el código para la resta, la única modificación realizada es que se pone el signo opuesto para el segundo operando para convertir la suma en una resta:

```
int signo2 = (n2.signo + 1) % 2;
```

Código 7: Cambio de signo del operando 2 de la resta para convertirla en una suma

Nota: Las excepciones también deben ser adaptadas.

6.5. Implementación del operador Multiplicación

Se ha realizado una implementación inicial del operador producto, en la que se realiza el producto de las mantisas utilizando directamente el operador * de Java, por lo tanto, se debe convertir la mantisa de *long* a *double*, pues si se multiplican los *long* directamente, además de producir un desbordamiento, se debe dividir el resultado por *KMANTISA*^2. Para realizar esta conversión, simplemente se divide la mantisa por la constante *kMANTISA*:

```
mantisa_double =  mantisa_long / kMantisa
```

Código 8: Mantisa Multiplicación en formato *double*

Al realizar el producto de las mantisas, los operandos son dos números que se encuentran en el rango [1.0, 2[, cuyo producto está acotado en el rango [1.0, 4[. Por lo tanto, para realizar la normalización, se comprueba si el resultado del producto es mayor o igual a 2 comprobando el bit 53, si es así, se desplaza la mantisa hacia la derecha una posición y se incrementa en 1 el exponente; a continuación.

```
/*Se comprueba si el numero se encuentra en [2.0, 4.0[
   comprobando el valor del bit 53 */
int dsp = (int)(mantisa_ >> 53) & 0x1;
/*Desplazamos la mantisa y le quitamos 1 implicito*/
mantisa  = mantisa  >> dsp;
exponente_ = exponente_ + 1 + dsp;
```

Código 9: Normalización de la mantisa en el producto

El código queda de la siguiente manera:

```
/**Funcion de multiplicacion (directa).
 * Obtiene el resultado de multiplicar dos numeros de doble
 * precision.
 *
 * @param n2 Multiplicador, numero de doble precision en formato IEEE754
 *
 * @return Resultado de la multiplicacion en formato de numero de doble precision
IEEE754
 */

public DoublePrecNumber Producto (DoublePrecNumber n2)
{
	if (this.exponente == 0x7ff && this.mantisa != 0 || n2.exponente == 0x7ff &&
n2.mantisa != 0)
		return new DoublePrecNumber(DoublePrecNumber.NaN);
	else
		if (this.exponente == 0 && this.mantisa == 0 || n2.exponente == 0 &&
n2.mantisa == 0)
		{
			if (n2.exponente == 0x7ff || this.exponente == 0x7ff)
				return new DoublePrecNumber (DoublePrecNumber.NaN);
			else if (n2.signo == this.signo)
				return new DoublePrecNumber(DoublePrecNumber.Cero);
			else
				return new DoublePrecNumber(DoublePrecNumber.MenosCero);
		}

		else if (this.exponente == 0x7ff && this.mantisa == 0 || n2.exponente ==
0x7ff && n2.mantisa == 0)
		{
			if (this.signo == n2.signo)
				return new DoublePrecNumber(DoublePrecNumber.Infinito);
			else if (this.signo != n2.signo)
				return new
DoublePrecNumber(DoublePrecNumber.MenosInfinito);
		}

	//NO PODEMOS OPERAR DIRECTAMENTE LAS MANTISAS PORQUE HARIAN FALTA 104 BITS
	//Y CON LONG SOLO DISPONEMOS DE 64 BITS.
	double producto = ((double)(this.mantisa) / DoublePrecNumber.kMANTISA) *
((double)(n2.mantisa) / DoublePrecNumber.kMANTISA);

	int exponente_ = this.exponente + n2.exponente - DoublePrecNumber.kSESGO - 1;
//RESTAMOS 1 POR EL 1 IMPLICITO

	if (producto >= 2.0)
	{
		producto /= 2.0;
		exponente ++;
	}

	long mantisa_ = ((long)(producto * DoublePrecNumber.kMANTISA));


	exponente  = exponente  + 1;


	//AJUSTAMOS EL SIGNO:
	// + * + = +
	// + * - = -
	// - * + = -
	// - * - = +
	int     signo_ = (this.signo + n2.signo) & 0x1;

	if (exponente_ >= 0x7ff)
	{
		if (signo_ == 0)
			return new DoublePrecNumber (DoublePrecNumber.Infinito);
		else
			return new DoublePrecNumber (DoublePrecNumber.MenosInfinito);
	}
```

```
        //OVERFLOW
        if (exponente_ >= 0x7ff)
        {
                if (signo_ == 0)
                        return new DoublePrecNumber (DoublePrecNumber.Infinito);
                else
                        return new DoublePrecNumber (DoublePrecNumber.MenosInfinito);
        }
        //DENORMALIZADO
        if (exponente_ < 0)
        {
                return new DoublePrecNumber (signo_, 0, mantisa_ >> -exponente_);
        }

        return new DoublePrecNumber (signo_, exponente_, mantisa_);
}
```

Código 10. Producto (versión inicial)

Para disponer de una implementación más cercana a lo que ocurre en el hardware, se ha realizado la implementación de los dos algoritmos de producto de mantisas ilustrados anteriormente.

En ambas versiones, en la iterativa y en la de *Booth* se desplaza el registro producto, de doble tamaño al de mantisa a la derecha. En la siguiente implementación, ese desplazamiento del registro de doble tamaño se implementa con 3 variables de tipo *long* –*productoH* para la mitad izquierda, *productoL* para la mitad derecha y *tmp* para representar el desplazamiento aritmético de *productoH* a *productoL*-- ya que no se dispone de ningún tipo de datos con 106 bits, y ese desplazamiento se realiza en tres pasos:

1. Obtener en *tmp* el valor del bit 0 del producto y desplazarlo 52 bits a la izquierda
2. Desplazar el *productoH* a la derecha
3. Desplazar el *productoL* a la derecha y añadirle a su bit de mayor peso (bit 52) el bit que fue desplazado de *productoH* aplicándole a *productoL* una operación *OR* con *tmp*.

La versión iterativa del producto de las mantisas se ha implementado de la siguiente manera:

```
/**Funcion de multiplicacion a partir de suma iterativa.
 * Obtiene el resultado de multiplicar dos numeros de doble
 * precision.
 *
 * @param n2 Multiplicador, numero de doble precision en formato IEEE754
 *
 * @return Resultado de la multiplicacion en formato de numero de doble precision
IEEE754
 */


public DoublePrecNumber Producto_it (DoublePrecNumber n2)
{
        if (this.exponente == 0x7ff && this.mantisa != 0 || n2.exponente == 0x7ff &&
n2.mantisa != 0)
                return new DoublePrecNumber(DoublePrecNumber.NaN);
        else
                if (this.exponente == 0 && this.mantisa == 0 || n2.exponente == 0 &&
n2.mantisa == 0)
                {
                        if (n2.exponente == 0x7ff || this.exponente == 0x7ff)
                                return new DoublePrecNumber (DoublePrecNumber.NaN);
                        else if (n2.signo == this.signo)
                                return new DoublePrecNumber(DoublePrecNumber.Cero);
                        else
                                return new DoublePrecNumber(DoublePrecNumber.MenosCero);
                }

                else if (this.exponente == 0x7ff && this.mantisa == 0 || n2.exponente ==
0x7ff && n2.mantisa == 0)
                {
                        if (this.signo == n2.signo)
                                return new DoublePrecNumber(DoublePrecNumber.Infinito);
```

```
                        else if (this.signo != n2.signo)
                                return new
DoublePrecNumber(DoublePrecNumber.MenosInfinito);
                }

        long productoH = 0;
        long multiplicando = this.mantisa;
        long productoL = n2.mantisa;
        long tmp;
        int cont = 53;
        while (cont > 0)
        {
                if ((productoL & 0x1) == 1)
                        productoH +=  multiplicando;

                if (cont>1)
                {
                        tmp = (productoH  & 0x1) << 52;
                        productoH >>= 1;
                        productoL = (productoL>>1) | tmp;
                }
                cont--;
        }

        long mantisa_ = productoH;


        int exponente_ = this.exponente + n2.exponente - DoublePrecNumber.kSESGO;

        /*Comprobamos si el numero se encuentra en [2.0, 4.0[
           comprobando el valor del bit 53 */
        int dsp = (int)(mantisa_ >> 53) & 0x1;
        /*Desplazamos la mantisa*/
        productoL>>=dsp;
        tmp = (mantisa_& 0x1) << 52;
        productoL = productoL | (dsp*tmp);
        mantisa  = (mantisa  >> dsp);

        if (productoL > DoublePrecNumber.kMANTISA) mantisa_++;
        if (productoL == DoublePrecNumber.kMANTISA && (mantisa_ & 0x1) == 0x1)
mantisa_++;
        exponente  = exponente  + dsp;

        //AJUSTAMOS EL SIGNO:
        // + * + = +
        // + * - = -
        // - * + = -
        // - * - = +
        int     signo_ = (this.signo + n2.signo) & 0x1;


        //OVERFLOW
        if (exponente_ >= 0x7ff)
        {
                if (signo_ == 0)
                        return new DoublePrecNumber (DoublePrecNumber.Infinito);
                else
                        return new DoublePrecNumber (DoublePrecNumber.MenosInfinito);
        }
        //DENORMALIZADO
        if (exponente_ < 0)
        {
                return new DoublePrecNumber (signo_, 0, mantisa_ >> -exponente_);
        }

        return new DoublePrecNumber (signo_, exponente_, mantisa_);
}
```

Código 11. Producto por suma sucesiva

El producto aplicando el algoritmo de *Booth* tiene la siguiente implementación:

```
/**Funcion de multiplicacion mediante algoritmo de Booth.
 * Obtiene el resultado de multiplicar dos numeros de doble
 * precision.
 *
 * @param n2 Multiplicador, numero de doble precision en formato IEEE754
 *
 * @return Resultado de la multiplicacion en formato de numero de doble precision
IEEE754
 */

public DoublePrecNumber Producto_Booth (DoublePrecNumber n2)
{
	if (this.exponente == 0x7ff && this.mantisa != 0 || n2.exponente == 0x7ff &&
n2.mantisa != 0)
		return new DoublePrecNumber(DoublePrecNumber.NaN);
	else
		if (this.exponente == 0 && this.mantisa == 0 || n2.exponente == 0 &&
n2.mantisa == 0)
		{
			if (n2.exponente == 0x7ff || this.exponente == 0x7ff)
				return new DoublePrecNumber (DoublePrecNumber.NaN);
			else if (n2.signo == this.signo)
				return new DoublePrecNumber(DoublePrecNumber.Cero);
			else
				return new DoublePrecNumber(DoublePrecNumber.MenosCero);
		}

		else if (this.exponente == 0x7ff && this.mantisa == 0 || n2.exponente ==
0x7ff && n2.mantisa == 0)
		{
			if (this.signo == n2.signo)
				return new DoublePrecNumber(DoublePrecNumber.Infinito);
			else if (this.signo != n2.signo)
				return new
DoublePrecNumber(DoublePrecNumber.MenosInfinito);
		}

	long productoH = 0;
	long multiplicando = this.mantisa;
	long productoL = n2.mantisa;
	long tmp;

	int q0 = 0, qm1 = 0;



	int cont = 53;
	while (cont > 0)
	{
		q0 = (int)(productoL & 0x1);
//			productoH += (q0-qm1) *  multiplicando;
		if ( q0 == 1  && qm1 == 0)
		{
			productoH += multiplicando;
		}

		if ( q0 == 0  && qm1 == 1)
		{
			productoH -= multiplicando;
		}

		if (cont>1)
		{
			tmp = (productoH  & 0x1) << 52;
			productoH >>= 1;
			productoL = (productoL>>1) | tmp;
		}
		cont--;
	}


	long mantisa_ = productoH;

	int exponente_ = this.exponente + n2.exponente - DoublePrecNumber.kSESGO;

	/*Comprobamos si el numero se encuentra en [2.0, 4.0[
```

```
        comprobando el valor del bit 53 */
        int dsp = (int)(mantisa_ >> 53) & 0x1;

        /*Desplazamos la mantisa*/
        productoL>>=dsp;
        tmp = (mantisa_& 0x1) << 52;
        productoL = productoL | (dsp*tmp);
        mantisa  = (mantisa  >> dsp);

        if (productoL > DoublePrecNumber.kMANTISA) mantisa_++;
        if (productoL == DoublePrecNumber.kMANTISA && (mantisa_ & 0x1) == 0x1)
mantisa_++;

        exponente  = exponente  + dsp;

        //AJUSTAMOS EL SIGNO:
        // + * + = +
        // + * - = -
        // - * + = -
        // - * - = +
        int     signo_ = (this.signo + n2.signo) & 0x1;

        //OVERFLOW
        if (exponente_ >= 0x7ff)
        {
                if (signo_ == 0)
                        return new DoublePrecNumber (DoublePrecNumber.Infinito);
                else
                        return new DoublePrecNumber (DoublePrecNumber.MenosInfinito);
        }
        //DENORMALIZADO
        if (exponente_ < 0)
        {
                return new DoublePrecNumber (signo_, 0, mantisa_ >> -exponente_);
        }

        return new DoublePrecNumber (signo_, exponente_, mantisa_);
}
```

Código 12. Producto mediante algoritmo de *Booth*

6.6. Implementación del operador División

La implementación de la división directa es bastante similar a la del producto directo. En cuanto a la normalización de la mantisa, el rango en el que se encuentra el resultado de la división de mantisas es]0.5, 2[, por lo tanto, se comprueba si la mantisa del dividendo es menor que la mantisa del divisor, si es así, la mantisa resultado se desplaza un bit a la izquierda, restando 1 al exponente.

```
int dsp = (m1 >= m2) ? 0 : 1;
mantisa_ = (mantisa_ << dsp) - DoublePrecNumber.kMANTISA;
int exponente_ = this.exponente - n2.exponente + DoublePrecNumber.kSESGO - dsp;
```

Código 13: Normalización de la mantisa en la división

La implementación de la división por convergencia consiste en el siguiente código:

```
/**Funcion de division aplicando aproximacion por convergencia.
 * Obtiene el resultado de dividir dos numeros de doble
 * precision, aplicando la aproximacion por convergencia de modo que en el numerador
 * se obtiene el valor del cociente y en el denominador 1.
 *
 * @param n2 Divisor, numero de doble precision en formato IEEE754
 *
 * @return Resultado de la division en formato de numero de doble precision IEEE754
 */


public DoublePrecNumber DivisionConvergencia (DoublePrecNumber n2)
{
```

```
        //TRATAMOS LOS CASOS ESPECIALES
        //DIVISIONES POR CERO Y MENOS_CERO
        //ALGUNO DE LOS OPERANDOS ES NaN
        if (this.exponente == 0x7ff && this.mantisa != 0 || n2.exponente == 0x7ff &&
n2.mantisa != 0)
                return new DoublePrecNumber (DoublePrecNumber.NaN);

        else if (n2.exponente == 0  && n2.mantisa == 0 && n2.signo == 0)
        {
                if (this.exponente == 0 && this.mantisa == 0)
                        return new DoublePrecNumber (DoublePrecNumber.NaN);
                else if (this.signo == 0)
                        return new DoublePrecNumber(DoublePrecNumber.Infinito);
                else
                        return new DoublePrecNumber(DoublePrecNumber.MenosInfinito);
        }
        else if (n2.exponente == 0  && n2.mantisa == 0 && n2.signo == 1)
        {
                if (this.exponente == 0 && this.mantisa == 0)
                        return new DoublePrecNumber (DoublePrecNumber.NaN);
                else if (this.signo == 1)
                        return new DoublePrecNumber(DoublePrecNumber.Infinito);
                else
                        return new DoublePrecNumber (DoublePrecNumber.MenosInfinito);
        }


        //DIVIDENDO VALE CERO O MENOS_CERO
        else if (this.exponente == 0 && this.mantisa == 0 && this.signo == 0)
        {
                if (n2.exponente == 0 && n2.mantisa == 0)
                        return new DoublePrecNumber (DoublePrecNumber.NaN);
                else if (n2.signo == 0)
                        return new DoublePrecNumber (DoublePrecNumber.Cero);
                else
                        return new DoublePrecNumber (DoublePrecNumber.MenosCero);
        }
        else if (this.exponente == 0 && this.mantisa == 0 && this.signo == 1)
        {
                if (n2.exponente == 0 && n2.mantisa == 0)
                        return new DoublePrecNumber (DoublePrecNumber.NaN);

                else if (n2.signo == 1)
                        return new DoublePrecNumber (DoublePrecNumber.Cero);
                else
                        return new DoublePrecNumber (DoublePrecNumber.MenosCero);
        }

        //ALGUNO DE LOS OPERANDOS VALE INFINITO O MENOS INFINITO

        else if (this.exponente == 0x7ff && this.mantisa == 0)
        {
                if (n2.exponente == 0x7ff && n2.mantisa == 0)
                        return new DoublePrecNumber (DoublePrecNumber.NaN);
                else if (this.signo == n2.signo)
                        return new DoublePrecNumber (DoublePrecNumber.Infinito);
                else if (this.signo == n2.signo)
                        return new DoublePrecNumber (DoublePrecNumber.MenosInfinito);
        }
        else if (n2.exponente == 0x7ff && n2.mantisa == 0)
        {
                if (this.exponente == 0x7ff && this.mantisa == 0)
                        return new DoublePrecNumber (DoublePrecNumber.NaN);
                else if (this.signo == n2.signo)
                        return new DoublePrecNumber (DoublePrecNumber.Cero);
                else if (this.signo != n2.signo)
                        return new DoublePrecNumber (DoublePrecNumber.MenosCero);
        }


        DoublePrecNumber N = new DoublePrecNumber (0, DoublePrecNumber.kSESGO,
this.mantisa);
        DoublePrecNumber D = new DoublePrecNumber (0, DoublePrecNumber.kSESGO,
n2.mantisa);

        DoublePrecNumber Dos = new DoublePrecNumber (0, 0x400,
DoublePrecNumber.kMANTISA);//2.0
```

```
        for (int i = 0; i < 200; i++)
        {
                DoublePrecNumber tmp = Dos.Resta(D);
                N = N.Producto_it(tmp);
                D = D.Producto_it(tmp);
                if (D.exponente == DoublePrecNumber.kSESGO && D.mantisa == 0)
                        i = 1000;
        }

        int exponente_ = this.exponente - n2.exponente + N.exponente;
        long mantisa_ = N.mantisa;

        //AJUSTAMOS EL SIGNO:
        // + * + = +
        // + * - = -
        // - * + = -
        // - * - = +
        int     signo_ = (this.signo + n2.signo) & 0x1;



        //OVERFLOW
        if (exponente_ >= 0x7ff)
        {
                if (signo_ == 0)
                        return new DoublePrecNumber (DoublePrecNumber.Infinito);
                else
                        return new DoublePrecNumber (DoublePrecNumber.MenosInfinito);
        }
        //DENORMALIZADO
        if (exponente_ < 0)
        {
                return new DoublePrecNumber (signo_, 0, mantisa_ >> -exponente_);
        }

        return new DoublePrecNumber(signo_, exponente_, mantisa_);

}
```

Código 14: División por convergencia

La división por recíproco sigue el siguiente código:

```
/**Funcion de division aplicando aproximacion por Newton-Raphson.
 * Obtiene el resultado de dividir dos numeros de doble
 * precision, aplicando la multiplicacion del Dividendo por el inverso del divisor.
 *
 * @param arg Divisor, numero de doble precision en formato IEEE754
 *
 * @return Resultado de la division en formato de numero de doble precision IEEE754
 */

public DoublePrecNumber DivisionNewtonRaphson (DoublePrecNumber arg)
{
        //NO HACE FALTA TRATAR LOS CASOS ESPECIALES, PUES YA SE TRATAN EN INVERSO Y EN
PRODUCTO
        return this.Producto(new DoublePrecNumber(arg.InversoNewtonRaphson()));
}
```

Código 15: División por recíproco

```
/**Funcion de division aplicando aproximacion por Newton-Raphson, con correccion en
Remainder.
 * Obtiene el resultado de dividir dos numeros de doble
 * precision, aplicando la multiplicacion del Dividendo por el inverso del divisor.
 * Es mas precisa que DivisionNewtonRaphson, pero es mas lenta
 *
 * @param arg Divisor, numero de doble precision en formato IEEE754
 *
 * @return Resultado de la division en formato de numero de doble precision IEEE754
```

```
 */

public DoublePrecNumber DivisionNewtonRaphson2(DoublePrecNumber arg)
{
	//NO HACE FALTA TRATAR LOS CASOS ESPECIALES, PUES YA SE TRATAN EN INVERSO Y EN
PRODUCTO

	DoublePrecNumber Dinv = arg.InversoNewtonRaphson();
	DoublePrecNumber Q = this.Producto(new DoublePrecNumber(Dinv)); // Q = N * (1/D)
	DoublePrecNumber R = this.Resta(arg.Producto(Q)); // R = N - D * Q

	return Q.Suma(R.Producto(Dinv)); //Q' = Q + R * (1 / D)

}
```

Código 16: División por recíproco + residuo

6.7. Implementación del operador Recíproco

Inicialmente, se comprueba si se ha producido uno de los casos especiales, entonces, se devuelve el resultado directamente.

Para los casos "normales", se comprueba si el número es potencia de 2, en ese caso la mantisa vale 0, y si es el caso, se devuelve el número con el exponente opuesto, como se explica en el subapartado anterior.

Si el número no es potencia de dos, se aplica la ecuación de recurrencia vista anteriormente.

```
/**Calcula el inverso para un numero de doble precision en el estandar IEEE754 siguiendo
la aproximación de Newton-Raphson
 * @return Resultado segun el estandar IEEE754 de doble precision de aplicar la funcion
inverso a un numero de doble precision
 */

public DoublePrecNumber InversoNewtonRaphson ()
{
	//TRATAMOS LOS CASOS ESPECIALES
	//INVERSO DE CERO Y DE MENOS_CERO
	if (this.signo == 0 && this.mantisa == 0 && this.exponente == 0)
		return new DoublePrecNumber (DoublePrecNumber.Infinito);
	else if (this.signo == 1 && this.mantisa == 0 && this.exponente == 0)
		return new DoublePrecNumber (DoublePrecNumber.MenosInfinito);
	//INVERSO DE NaN
	else if (this.exponente == 0x7ff && this.mantisa != 0)
		return new DoublePrecNumber (DoublePrecNumber.NaN);
	//INVERSO DE INFINITO Y MENOS_INFINITO
	else if (this.exponente == 0x7ff && this.mantisa == 0 && this.signo == 0)
		return new DoublePrecNumber (DoublePrecNumber.Cero);
	else if (this.exponente == 0x7ff && this.mantisa == 0 && this.signo != 0)
		return new DoublePrecNumber (DoublePrecNumber.MenosCero);

	if (mantisa == DoublePrecNumber.kMANTISA) // EL NUMERO ES POTENCIA DE 2
	{
		int exp = this.exponente - DoublePrecNumber.kSESGO;
		return new DoublePrecNumber (signo, DoublePrecNumber.kSESGO - exp,
DoublePrecNumber.kMANTISA);
	}

	//INICIALIZAMOS LA SEMILLA

	DoublePrecNumber Dos = new DoublePrecNumber (0, 0x400,
DoublePrecNumber.kMANTISA);
	DoublePrecNumber b = new DoublePrecNumber (0, DoublePrecNumber.kSESGO,
this.mantisa);
	DoublePrecNumber x = new DoublePrecNumber (0, DoublePrecNumber.kSESGO - 1,
DoublePrecNumber.kMANTISA);
	//APLICAMOS LA FORMULA DE NEWTON RAPHSON PARA LA INVERSA

	for (int i = 0; i < 15; i++)
		x = x.Producto it(Dos.Resta(b.Producto it(x)));

	//REAJUSTAMOS EL EXPONENTE
	int exp = this.exponente - DoublePrecNumber.kSESGO;
```

```
        DoublePrecNumber result = new DoublePrecNumber(this.signo, x.exponente - exp,
x.mantisa);

        //OVERFLOW
        if (result.exponente >= 0x7ff)
        {
                if (result.signo == 0)
                        return new DoublePrecNumber (DoublePrecNumber.Infinito);
                else
                        return new DoublePrecNumber (DoublePrecNumber.MenosInfinito);
        }

        //DENORMALIZADO
        if (result.exponente < 0)
        {
                return new DoublePrecNumber (result.signo, 0, result.mantisa >> -
result.exponente);
        }
```

Código 17: Inverso *Newton-Raphson*

6.8. Implementación del operador raíz cuadrada por el método de Newton-Raphson

Inicialmente, se comprueban los casos especiales, y si es el caso, se devuelve el resultado directamente, así como para el cero, que se devuelve también cero directamente.

Si se trata de un caso normal, se procede a la desnormalización de la mantisa y se aplica el algoritmo citado en el subapartado anterior.

```
/**Funcion de raiz cuadrada empleando aproximacion por Newton Raphson.
 * Obtiene el resultado de aplicar la raiz cuadrada a un numero de doble precision
 *
 * @return Resultado de la raiz cuadrada en formato de numero de doble precision IEEE754
 */
public DoublePrecNumber RaizCuadradaNewtonRaphson ()
{

        //TRATAMOS LOS CASOS ESAPECIALES, NUMEROS NEGATIVOS
        if (this.signo == 1)
                return new DoublePrecNumber (DoublePrecNumber.NaN);
        //EL NUMERO VALE NaN
        else if (this.exponente == 0x7ff && this.mantisa != 0)
                return new DoublePrecNumber (DoublePrecNumber.NaN);
        //INFINITO
        else if (this.exponente == 0x7ff && this.mantisa == 0 && this.signo == 0)
                return new DoublePrecNumber (DoublePrecNumber.Infinito);
        //MENOS INFINITO
        else if (this.exponente == 0x7ff && this.mantisa == 0 && this.signo != 0)
                return new DoublePrecNumber (DoublePrecNumber.NaN);
        else if (this.signo == 0 && this.exponente == 0 && this.mantisa == 0)
                return new DoublePrecNumber (DoublePrecNumber.Cero);
        else if (this.signo == 1 && this.exponente == 0 && this.mantisa == 0)
                return new DoublePrecNumber (DoublePrecNumber.Cero);

        DoublePrecNumber gess = null;
        DoublePrecNumber next = null;
        try
        {
                //APLICAMOS LA FORMULA DE NEWTON RAPHSON PARA LA RAIZ CUADRADA

                gess = this.SHR(1); //gess=N/2
                next = new DoublePrecNumber(0, 0, 0);

                for (int i = 0; i < 15; i++)
                {
                        next = this.DivisionNewtonRaphson(gess); //next = num / gess;
                        gess = gess.Suma(next).SHR(1); //(gess + next) / 2;
                }

        }
        catch (Exception e)
        {
```

```
        }
        return next;
}
```

Código 18: Raíz cuadrada

6.9. Implementación del operador Valor Absoluto

```
/**Obtener el valor absoluto de un numero de doble precision.
 *
 * @return Valor absoluto del numero en doble precision
 */
public DoublePrecNumber Abs ()
{
        return new DoublePrecNumber(0, this.exponente, this.mantisa);
}
```

Código 19: Valor absoluto

6.10. Implementación del operador Opuesto

Debido a que el signo es de tipo entero, y no bit, el opuesto se implementa mediante el operador módulo en lugar de mediante operador complemento.

```
/**
 * Genera el opuesto del numero de doble precision (mismo valor absoluto, pero con
distinto signo)
 * @return El numero con el signo contrario.
 */
public DoublePrecNumber Opuesto()
{
        return new DoublePrecNumber ((this.signo + 1) & 0x1, this.exponente,
this.mantisa);
}
```

Código 20: Opuesto

6.11. Operadores de comparación

Se dispone de los seis operados de comparación (<, <=, >, >=, ==, ¡=) y del método *compareTo*, que sigue las especificaciones que dicta el estándar Java (-1 → menor; 0 → igual; 1 → mayor). A continuación, se ilustra el método *compareTo*:

```
/**Compara dos numeros de doble precision
 *
 * @param o Segundo operando de la comparacion
 * @return Resultado de la comparacion entre dos numeros de doble precision:
 * 1 si op1 > op2; 0 si op1==op2; -1 en otro caso.
 */
public int compareTo (Object o)
{
        DoublePrecNumber op2 = (DoublePrecNumber) o;

        if (this.mantisa == 0 && this.exponente == 0 && op2.mantisa == 0 && op2.exponente
== 0)
                return 0;
        else if (this.signo > op2.signo)
                return -1;
        else if (this.signo < op2.signo)
                return 1;
        else
        {

                if (this.exponente > op2.exponente && this.signo == 0)
                        return 1;
                else if (this.exponente > op2.exponente && this.signo == 1)
                        return -1;
```

```
                else if (this.exponente < op2.exponente && this.signo == 0)
                        return -1;
                else if (this.exponente < op2.exponente && this.signo == 1)
                        return 1;

                else
                {
                        if (this.signo == 0)
                                return new Long(this.mantisa).compareTo(new
Long(op2.mantisa));
                        else
                                return -new Long(this.mantisa).compareTo(new
Long(op2.mantisa));
                }
        }
}
```

Código 21: *compareTo*

Los operadores comparación se implementan de forma similar, se ilustra a continuación uno de ellos:

```
/**Funcion que compara dos numeros en punto flotante utilizando el operador Mayor O
Igual <strong>&ge;</strong>
 *
 * @param op2 Segundo operando para el operador de comparacion <strong>&ge;</strong>
 * @return <ul><li><em>false</em> si alguno de los operandos es
<strong>NaN</strong></li>
 * <li><em>true<em> si el primer operando es mayor o igual que el segundo</li>
 * <li><em>false<em> en otro caso</li></ul>.
 */

public boolean MayorOIgual(DoublePrecNumber op2)
{
        if (this.mantisa != 0 && this.exponente == 0x7ff || op2.exponente == 0x7ff &&
op2.mantisa != 0)
                return false;
        else if (this.mantisa == 0 && this.exponente == 0 && op2.mantisa == 0 &&
op2.exponente == 0)
                return true;
        else if (this.signo > op2.signo)
                return false;
        else if (this.signo < op2.signo)
                return true;
        else
        {
                if (this.exponente != op2.exponente && this.signo == 0)
                        return this.exponente>op2.exponente;
                        else if (this.exponente != op2.exponente && this.signo == 1)
                                return this.exponente < op2.exponente;
                        else
                        {
                                if (this.signo == 0)
                                        return this.mantisa >= op2.mantisa;
                                        else
                                                return this.mantisa <= op2.mantisa;
                        }
        }
}
```

Código 22: *Operador Mayor o igual*

6.12. Implementación de la función exponencial

La implementación para la función exponencial, a partir del algoritmo 6, es la siguiente:

```
/**
 * Funcion exp (x)
 * @return el resultado de aplicar la función exponencial a un número
 */
public DoublePrecNumber Exp ()
{
	DoublePrecNumber Uno = new DoublePrecNumber (0, DoublePrecNumber.kSESGO,
DoublePrecNumber.kMANTISA);
	DoublePrecNumber n = DoublePrecNumber.Cero;
	DoublePrecNumber oldsum = DoublePrecNumber.Cero;
	DoublePrecNumber newsum = Uno;
	DoublePrecNumber term = Uno;

	DoublePrecNumber x = new DoublePrecNumber (this);
	x.signo = 0;

	//while (newsum.Distinto(oldsum))
	while (newsum.mantisa != oldsum.mantisa || newsum.exponente != oldsum.exponente
|| newsum.signo != oldsum.signo)
	{
		oldsum = new DoublePrecNumber (newsum);
		n = n.Suma(Uno); // n++
		term = term.Producto it(x.Division(n));
		newsum = newsum.Suma(term);
	}
	return (this.signo == 0) ? newsum : newsum.InversoNewtonRaphson();
}
```

Código 23: Función $exp(x)$

Como se puede observar, en la condición del bucle, se ha optado por utilizar la comparación directamente en lugar de llamar a la función Distinto, con el objetivo de tener un mayor rendimiento.

6.13. Implementación de la función Truncate

A continuación se muestra la implementación del redondeo hacia cero.

```
/**Funcion de Truncamiento o Redondeo hacia Cero.
 * Obtiene el resultado de redondear un numero doble
 * precision, aplicando un truncamiento.
 * @return Resultado de redondear un numero real en formato
 * de numero de doble precision IEEE754 por truncamiento a
 * un numero entero en formato de doble precision IEEE754
 */

public DoublePrecNumber Truncate ()
{
	//TRATAMOS LOS CASOS ESPECIALES
	//UNO DE LOS SUMANDOS VALE 0
	if (this.exponente == 0 && this.mantisa == 0)
		return new DoublePrecNumber(DoublePrecNumber.Cero);
	//ALGUNO DE LOS OPERANDOS ES NaN
	else if (this.exponente == 0x7ff && this.mantisa != 0)
		return new DoublePrecNumber(DoublePrecNumber.NaN);
	else if (this.exponente == 0x7ff && this.mantisa == 0 && this.signo == 0)
	{
			return new DoublePrecNumber(DoublePrecNumber.Infinito);
	}
	else if (this.exponente == 0x7ff && this.mantisa == 0 && this.signo == 1)
	{
			return new DoublePrecNumber(DoublePrecNumber.MenosInfinito);
	}


	long mantisa1 = this.mantisa;
	if (this.exponente - DoublePrecNumber.kSESGO >= 0)
	{
```

```
                long desecho = this.mantisa & ( (1l<<(52 - this.exponente +
DoublePrecNumber.kSESGO))-1);

                mantisa1 &= ~desecho;
                return new DoublePrecNumber (this.signo, this.exponente, mantisa1);
        }
        else
                return new DoublePrecNumber (0, 0, 0);
}
```

Código 24: Función *Truncate(x)*

6.14. Implementación de la función Ceil

A continuación se muestra la implementación para el redondeo hacia más infinito de un número real a un número entero.

```
/**Funcion de Redondeo hacia Infinito.
 * Obtiene el resultado de redondear un numero doble
 * precision, aplicando redondeo hacia Infinito.
 * @return Resultado de redondear un numero real en formato
 * de numero de doble precision IEEE754 por redondeo a Infinito a
 * un numero entero en formato de doble precision IEEE754
 */

public DoublePrecNumber Ceil ()
{
        //TRATAMOS LOS CASOS ESPECIALES
        //CERO
        if (this.exponente == 0 && this.mantisa == 0)
                return new DoublePrecNumber(DoublePrecNumber.Cero);
        //NaN
        else if (this.exponente == 0x7ff && this.mantisa != 0)
                return new DoublePrecNumber(DoublePrecNumber.NaN);
        //INFINITO
        else if (this.exponente == 0x7ff && this.mantisa == 0 && this.signo == 0)
        {
                        return new DoublePrecNumber(DoublePrecNumber.Infinito);
        }
        //MENOS INFINITO
        else if (this.exponente == 0x7ff && this.mantisa == 0 && this.signo == 1)
        {
                        return new DoublePrecNumber(DoublePrecNumber.MenosInfinito);
        }

        long mantisa1 = this.mantisa;
        int exp1 = this.exponente;
        int signo1 = this.signo;

        if (this.exponente - DoublePrecNumber.kSESGO >= 0)
        {
                long desecho = this.mantisa & ( (1l<<(52 - this.exponente +
DoublePrecNumber.kSESGO))-1);

                mantisa1 &= ~desecho;
                if (desecho > 0 && signo == 0)
                {
                        mantisa1 <<= 8;
                        int exp2 = DoublePrecNumber.kSESGO;
                        long mantisa2 = (DoublePrecNumber.kMANTISA) << 8;


                        //AJUSTAMOS LOS EXPONENTES Y LAS MANTISAS
                        //ANYADIMOS A LAS MANTISAS EL 1.0 QUE IEEE754 TIENE IMPLICITO
                        //SI EL PRIMER OPERANDO ES MAYOR QUE EL SEGUNDO, DESPLAZAMOS LA
MANTISA DEL SEGUNDO A LA DCHA
                        //SI EL SEGUNDO ES MAYOR QUE EL PRIMERO, DESPLAZAMOS LA MANTISA
DEL PRIMERO A LA DERECHA
                        if (exp1 > exp2)
                        {
                                int dsp = exp1-exp2;
                                if (dsp < 60)
                                {
```

```
                                        mantisa2 = (mantisa2) >> dsp;
                                }
                                else
                                {
                                        mantisa2 = 1;
                                }

                                exp2 = exp1;
                        }

                        mantisa1 = (mantisa1 + mantisa2);



                        int dsp = (int) (mantisa1 >> 61) & 0x1;
                        exp1 += dsp;
                        long GRS = ((mantisa1>>(dsp)) & 0xFF);
                        mantisa1 = (mantisa1 >> (8+dsp));
                        if (GRS > 128)mantisa1++;
                        if (GRS == 128 && (mantisa1 & 0x1) == 1) mantisa1++;


                        //OVERFLOW
                        if (exp1 >= 0x7ff)
                        {
                                if (signo1 == 0)
                                        return new DoublePrecNumber
(DoublePrecNumber.Infinito);
                                else
                                        return new DoublePrecNumber
(DoublePrecNumber.MenosInfinito);
                        }
                        //DENORMALIZADO
                        if (exp1 < 0)
                        {
                                return new DoublePrecNumber (signo1, 0, mantisa1 >> -exp1);
                        }



                }

                return new DoublePrecNumber (signo1, exp1, mantisa1);
        }
        else
        {

                if (signo1 == 0)
                        return new DoublePrecNumber (0, DoublePrecNumber.kSESGO,
DoublePrecNumber.kMANTISA);
                else
                        return DoublePrecNumber.Cero;
        }
}
```

Código 25: Función *Ceil(x)*

6.15. Implementación de la función Floor

La implementación de *Floor* es bastante similar a la de *Ceil,* pues el caso en el que se le suma 1 a la mantisa truncada en *Floor* es el opuesto a cuando ocurre en *Ceil.*

6.16. Implementación de la función Round

A continuación se muestra la implementación para el redondeo al entero más próximo.

```
/**Funcion de Redondeo hacia el mas proximo.
 * Obtiene el resultado de sumar 1.0 a un numero doble
 * precision, aplicando un operador suma.
 * @return Resultado de la resta en formato de numero de doble precision IEEE754
 */

public DoublePrecNumber Round ()
{
	//TRATAMOS LOS CASOS ESPECIALES
	//CERO
	if (this.exponente == 0 && this.mantisa == 0)
		return new DoublePrecNumber(DoublePrecNumber.Cero);
	//NaN
	else if (this.exponente == 0x7ff && this.mantisa != 0)
		return new DoublePrecNumber(DoublePrecNumber.NaN);
	//INFINITO
	else if (this.exponente == 0x7ff && this.mantisa == 0 && this.signo == 0)
	{
			return new DoublePrecNumber(DoublePrecNumber.Infinito);
	}
	//MENOS INFINITO
	else if (this.exponente == 0x7ff && this.mantisa == 0 && this.signo == 1)
	{
			return new DoublePrecNumber(DoublePrecNumber.MenosInfinito);
	}

	long mantisa1 = this.mantisa;
	int exp1 = this.exponente;
	int signo1 = this.signo;

	int exp_sin_sesgo = this.exponente - DoublePrecNumber.kSESGO;
	if (exp_sin_sesgo >= 0)
	{
		long desecho = this.mantisa & ( (1l<<(52 - exp_sin_sesgo))-1);

		mantisa1 &= ~desecho;
		if (desecho >> (52 - exp_sin_sesgo -1) != 0)
		{
			mantisa1 <<= 8;
			int exp2 = DoublePrecNumber.kSESGO;
			long mantisa2 = (DoublePrecNumber.kMANTISA) << 8;


			//AJUSTAMOS LOS EXPONENTES Y LAS MANTISAS
			//ANYADIMOS A LAS MANTISAS EL 1.0 QUE IEEE754 TIENE IMPLICITO
			//SI EL PRIMER OPERANDO ES MAYOR QUE EL SEGUNDO, DESPLAZAMOS LA
MANTISA DEL SEGUNDO A LA DCHA
			//SI EL SEGUNDO ES MAYOR QUE EL PRIMERO, DESPLAZAMOS LA MANTISA
DEL PRIMERO A LA DERECHA
			if (exp1 > exp2)
			{
				int dsp = exp1-exp2;
				if (dsp < 60)
				{
					mantisa2 = (mantisa2) >> dsp;
				}
				else
				{
					mantisa2 = 1;
				}

				exp2 = exp1;
			}

			mantisa1 = (mantisa1 + mantisa2);



			int dsp = (int) (mantisa1 >> 61) & 0x1;
			exp1 += dsp;
```

```
                        long GRS = ((mantisa1>>(dsp)) & 0xFF);
                        mantisa1 = (mantisa1 >> (8+dsp));
                        if (GRS > 128)mantisa1++;
                        if (GRS == 128 && (mantisa1 & 0x1) == 1) mantisa1++;


                        //OVERFLOW
                        if (exp1 >= 0x7ff)
                        {
                                if (signo1 == 0)
                                        return new DoublePrecNumber
(DoublePrecNumber.Infinito);
                                else
                                        return new DoublePrecNumber
(DoublePrecNumber.MenosInfinito);
                        }
                        //DENORMALIZADO
                        if (exp1 < 0)
                        {
                                return new DoublePrecNumber (signo1, 0, mantisa1 >> -exp1);
                        }



                }

                return new DoublePrecNumber (signo1, exp1, mantisa1);
        }
        else
        {
                if (exp_sin_sesgo == -1)
                        return new DoublePrecNumber (signo1, DoublePrecNumber.kSESGO,
DoublePrecNumber.kMANTISA);
                else
                        return DoublePrecNumber.Cero;
        }
}
```

Código 26. Round(x)

6.17. Implementación de la función Escalado

Para facilitar la implementación, se ha separado el escalado de potencias de 2 positivas y negativas. En ambos casos, se debe pasar un valor entero positivo para el exponente de las potencias positivas y negativas de 2. Si el exponente que se recibe como argumento es negativo, el operador arroja una excepción, que debe ser capturada cuando se invoca a este operador.

El escalado positivo sigue la siguiente implementación:

```
/**Operador de Escalado con potencia positiva de 2.
 * @param dsp Exponente para una potencia positiva de 2
 * @return Resultado de aplicar el producto por una potencia positiva de 2 con exponente
<em>dsp</em>
 * @throws Exception si se intruduce un exponente negativo.
 */

public DoublePrecNumber SHL (int dsp) throws Exception
{
        if (dsp < 0) throw new Exception ("Error, el desplazamiento debe ser positivo");

        if ((mantisa == 0 && exponente == 0) || this.exponente == 0x7ff)
                return new DoublePrecNumber(this);

        DoublePrecNumber result = new DoublePrecNumber(this);

        //SI TENEMOS UN NUMERO DENORMALIZADO, PRIMERO DESPLAZAMOS LA MANTISA A LA
IZQUIERDA
        //Y LUEGO SE LE SUAM AL EXPONENTE LO QUE FALTA.
        while (result.mantisa < DoublePrecNumber.kMANTISA && dsp > 0)
        {
                result.mantisa <<= 1;
                dsp--;
        }
```

```
        result.exponente += dsp;

        //OVERFLOW
        if (result.exponente >= 0x7ff)
        {
                if (result.signo == 0)
                        return new DoublePrecNumber (DoublePrecNumber.Infinito);
                else
                        return new DoublePrecNumber (DoublePrecNumber.MenosInfinito);
        }

        return result;
}
```

Código 27. Escalado por potencia positiva de 2

El escalado negativo sigue la siguiente implementación:

```
/**Operador de Escalado con potencia negativa de 2.
 * @param dsp Exponente para una potencia negativa de 2
 * @return Resultado de aplicar el producto por una potencia negativa de 2 con exponente
<em>dsp</em>
 * @throws Exception si se intruduce un exponente negativo.
 */

public DoublePrecNumber SHR (int dsp) throws Exception
{
        //TRATAMOS LOS CASOS ESPECIALES
        //CERO, MENOSCERO, INFINITO Y MENOS INFINITO
        if (dsp < 0) throw new Exception ("Error, el desplazamiento debe ser positivo");
        if ((mantisa == 0 && exponente == 0) || this.exponente == 0x7ff)
                return new DoublePrecNumber(this);

        DoublePrecNumber result = new DoublePrecNumber(this);

        result.exponente -= dsp;

        //DENORMALIZADO
        if (result.exponente < 0)
        {
                return new DoublePrecNumber (result.signo, 0, result.mantisa >> -
result.exponente);
        }

        return result;

}
```

Código 28. Escalado por potencia negativa de 2

6.18. Operadores de tiempo real

Para representar los números que servirán como operandos para los operadores de tiempo real, se dispone de una clase que deriva de *DoublePrecNumber* a la que se le han añadido los métodos que implementan estos operadores de tiempo real a partir de los operadores estándar.

El procedimiento básico que se realiza para implementar estos operadores es aplicar el truncamiento de los números a partir del número de etapas y a continuación se aplica el operador estándar.

Se muestra como ejemplo el operador *Suma* para tiempo real.

```
/**Funcion de suma para tiempo real. Version no emulada, trunca los operandos a la
precision
 * y llama a la operacion suma normal.

 * @throws Exception si el parametro t recibe un valor fuera de rango
 *@param arg Segundo operando para la suma
 *@param t Numero de trozos sobre el que aplicar el operador Add
 *@return Numero resultado de aplicar la suma de dos numeros de doble
```

```
 *precision en formato IEEE754 sobre <em>t</em> trozos de mantisa*/

public DoublePrecNumberTR SumaTR (DoublePrecNumberTR arg, int t) throws Exception
{
        if (t < 1 || t > 6) throw new Exception ("parametro t fuera de rango");

        DoublePrecNumberTR op1 = new DoublePrecNumberTR(this);
        DoublePrecNumberTR op2= new DoublePrecNumberTR(arg);
        op1.TruncarNumero(t);
        op2.TruncarNumero(t);
        DoublePrecNumberTR result = new DoublePrecNumberTR (op1.Suma(op2));
        return result;
}
```

Código 29. Operador Suma para tiempo real

7. EJEMPLO DE USO

A continuación se introduce un sencillo ejemplo de uso de la librería. Se ha generado un paquete java con los archivos compilados *.class* que se llama *ieee754_Operator.jar.*

```
import ieee754_Operator.*;
/**
<P><FONT COLOR="blue">
 *<B>Prueba:</B> Esta clase tiene el rol de ilustrar
 *un sencillo ejemplo de uso de la libreria
 *<B>Descripcion</B><BR>
 *<FONT COLOR="blue">
 *La clase Prueba ilustra un ejempsencillo de las entradas y salidas
 *para algunos de los operadores de la libreria
*/

public class Prueba
{
        public Prueba()
        {
        }
        public static void main (String [] args)
        {
                DoublePrecNumber x = new DoublePrecNumber(29.0);
                DoublePrecNumber y = new DoublePrecNumber(13.0);
                System.out.println(x.Suma(y));
                System.out.println(x.Resta(y));
                System.out.println(y.Resta(x));
                System.out.println(x.Producto(y));
                System.out.println(x.Division(y));
                System.out.println(x.RaizCuadradaNewtonRaphson());
                System.out.println(x.InversoNewtonRaphson());
        }
}
```

Código 30. Clase de prueba

La salida para este pequeño programa de prueba sería la siguiente:

```
42.0: 0 10000000100 1.0101000000000000000000000000000000000000000000000000
16.0: 0 10000000011 1.0000000000000000000000000000000000000000000000000000
-16.0: 1 10000000011 1.0000000000000000000000000000000000000000000000000000
377.0: 0 10000000111 1.0111100100000000000000000000000000000000000000000000
2.230769230769231: 0 10000000000 1.0001110110001001110110001001110110001001110110001010
5.385164807134504: 0 10000000001 1.0101100010100110100010100100101010001101100111110011
0.034482758620689655: 0 01111111010 1.0001101001111011100101100001000110100111101110010110
```

Código 31. Salida para el programa prueba

9. EXPERIMENTACIÓN

A continuación se muestran los resultados obtenidos de aplicar para cada operador 1000 operaciones, para las 6 etapas, indicando el error máximo y el error promedio para estos operadores. Nótese que los valores obtenidos cuando se utilizan las 6 etapas se corresponden también con los operadores originales antes de incluir la adaptación para tiempo real. La experimentación para buscar el error máximo genera la siguiente tabla:

TABLA 15: ERROR MÁXIMO PARA 1000 OPERACIONES

Operacion/Precision	1	2	3	4	5	6
Suma	4,77E-04	1,87E-06	7,17E-09	2,72E-11	1,13E-13	0
Resta	4,74E-04	1,86E-06	7,23E-09	2,79E-11	1,11E-13	0
Producto	8,42E-04	3,25E-06	1,29E-08	5,01E-11	1,90E-13	0
Producto_it	8,42E-04	3,25E-06	1,29E-08	5,01E-11	1,90E-13	0
Producto_Booth	8,42E-04	3,25E-06	1,29E-08	5,01E-11	1,90E-13	0
Division NR	3,95E-04	1,30E-06	5,41E-09	2,15E-11	7,35E-14	4,44E-16
Division Convergencia	3,95E-04	1,30E-06	5,41E-09	2,15E-11	7,33E-14	6,66E-16
RaizCuadrada NR	1,21E-04	4,72E-07	1,83E-09	7,24E-12	2,75E-14	4,44E-16
Inverso	2,41E-04	9,36E-07	3,55E-09	1,44E-11	5,16E-14	1,11E-16
Inverso NR	2,41E-04	9,36E-07	3,55E-09	1,44E-11	5,16E-14	1,11E-16

La experimentación para buscar el error promedio genera la siguiente tabla:

TABLA 16: ERROR PROMEDIO PARA 1000 OPERACIONES

Operacion/Precision	1	2	3	4	5	6
Suma	1,55E-04	6,34E-07	2,39E-09	9,41E-12	3,65E-14	0
Resta	1,61E-04	6,53E-07	2,58E-09	1,05E-11	3,82E-14	0
Producto	3,52E-04	1,43E-06	5,63E-09	2,24E-11	8,39E-14	0
Producto_it	3,52E-04	1,43E-06	5,63E-09	2,24E-11	8,39E-14	0
Producto_Booth	3,52E-04	1,43E-06	5,63E-09	2,24E-11	8,39E-14	0
Division NR	6,35E-05	2,53E-07	9,40E-10	3,80E-12	1,47E-14	6,23E-17
Division Convergencia	6,35E-05	2,53E-07	9,40E-10	3,80E-12	1,47E-14	1,36E-16
RaizCuadrada NR	4,69E-05	1,97E-07	7,88E-10	3,15E-12	1,18E-14	7,97E-17
Inverso NR	5,69E-05	2,39E-07	9,58E-10	3,87E-12	1,43E-14	3,49E-17

8. CONCLUSIONES

IEEE754 es el estándar para aritmética en coma flotante de IEEE más utilizado en representación numérica de números reales. Este estándar determina una serie de formatos para la representación de números en coma flotante junto con un conjunto de operaciones en punto flotante que opera sobre estos valores así como modos de redondeo. Además, se definen casos de excepción para las operaciones, cuándo ocurren y cómo se tratan. El estándar, fue aprobado en 1985 y actualmente está en revisión.

Los formatos de representación más utilizados de IEEE754 son la simple precisión y la doble precisión, presentes en lenguajes de programación, en el caso de C/C++ y Java se utiliza en los formatos *float* y *double* respectivamente.

No obstante, dichos lenguajes de operación no permiten trabajar con los números en punto flotante a nivel de bit, motivo por el que se considera necesaria la creación de algún tipo de herramienta que permita realizar este tipo de operaciones.

Se ha desarrollado una librería en Java 5 con la que se pretende simular el funcionamiento de una unidad aritmético-lógica siguiendo el estándar IEEE754.

Para ello, se han implementado las operaciones básicas (suma, resta, producto, división, inverso y raíz cuadrada) siguiendo los pasos vistos en las asignaturas *Informática Básica, Estructuras de Computadores y Arquitecturas y Sistemas Operativos en Tiempo Real*, los cuales se implementarán del modo más determinista posible, para hacer lo más independientemente posible del tamaño de los operadores sobre los que se aplica el operador.

El objetivo principal de esta librería es que pueda servir de material de apoyo a la docencia, así como soporte para la experimentación.

Uno de los objetivos que se pretendía conseguir es una emulación software de lo que ocurriría en una *ALU* (Unidad aritmético-lógica) de modo que los tiempos de ejecución de las operaciones guarden una relación con lo que ocurriría realmente si se ejecutara en dicho HW.

Existen algunas asignaturas de arquitecturas de computadores en las que se proponen algoritmos que, posteriormente, deberían ser implementados en algún circuito integrado/arquitectura. Para un funcionamiento óptimo, se suele proponer una versión inicial del algoritmo, y se estudia cómo podrían mejorarse las operaciones de modo que el sistema funcione de la forma más óptima posible.

Sin embargo, la falta de medios obliga a implementar estos algoritmos en lenguajes de alto nivel, como por ejemplo C. Es cierto que tanto las librerías de C, como puede ser la matemática, como los operadores, están altamente optimizados, por lo que al sustituir ciertas operaciones pesadas por una combinación de operaciones ligeras, como puede ser productos por potencias de dos por sumas y desplazamientos, se presenta la paradoja en la que la latencia de la composición de operaciones ligeras es mayor que la de la operación pesada – ¡¡¡versión optimizada más lenta que la no optimizada!!!-- debido al número de operaciones, lo que se aleja a lo que ocurriría en un sistema HW.

Los esfuerzos por conseguir una emulación de Hardware, donde efectivamente ocurre que sumas y desplazamientos son operaciones más rápidas que productos y potencias, es poder observar que efectivamente se consigue ganancia al comparar la versión optimizada con la no optimizada.

9. TRABAJO FUTURO

La construcción de esta librería se deja abierta en el sentido de que se pretenden incorporar las operaciones lógicas, y algunas matemáticas mediante aproximaciones basadas en Taylor o CORDIC, así como estudios posteriores de la implementación para conseguir un mayor grado de optimización y determinismo.

En futuras revisiones se analizarán los errores de redondeo que se producen en operaciones con convergencia (los dos operadores de división, inverso y la raíz cuadrada).

El objetivo de realizar la implementación del modo más determinista posible es poder hacer una adaptación de la librería hacia la operatoria en tiempo real, de modo que se puedan observar las relaciones precisión/rendimiento.

REFERENCES

[1] IEEE Standard for Binary Floating-Point Arithmetic. http://kfe.fjfi.cvut.cz/~adamek/nm/ieee754.pdf

[2] Wikipedia. IEEE754. 2004. http://es.wikipedia.org/wiki/IEEE_754

[3] IEEE754 References. http://babbage.cs.qc.edu/courses/cs341/IEEE-754references.html

[4] M. D. Ercegovac, Digital arithmetic, Morgan Kaufmann,2003.

[5] M.L. Overton, Numerical computing with IEEE floating point arithmetic : including one theorem, one rule of thumb, and one hundred and one exercises, Society for Industrial & Applied, 2001.

[6] Apuntes de Estructuras de Computadores. Universidad de Alicante. 2006.

[7] Apuntes de Arquitecturas y Sistemas Operativos para Tiempo Real. Universidad de Alicante. 2006

[8] Jerónimo Mora, Unidades Aritméticas en Coma Flotante para Tiempo Real, Tesis Doctoral, 2001.

[9] CMPSCI 201, Lecture 15. 2004. http://www.cs.umass.edu/~verts/cmpsci201/spr_2004/Lecture_15_2004-03-03_A_Worked_Floating_Point_Problem.pdf

[10] Koren, Israel, Computer arithmetic algorithms. A K Peters Ltd, 2001.

[11] Flynn, M. J. Advanced computer arithmetic design, Wiley, 2001.

[12] Cavanagh, Joseph J.F., Digital computer arithmetic: design and implementation, McGraw-Hill College, 1984.