Apuntes de Clases

Operaciones Matemáticas con Números Binarios.

Realizado por Sergio Noriega

Introducción a los Sistemas Lógicos y Digitales
Departamento de Electrotécnia
Facultad de Ingeniería
Universidad Nacional de La Plata
2003

INDICE

1 - Introducción.

Los números binarios tienen la particularidad de que su representación con sólo dos símbolos, permiten poder introducirlos en un circuito digital donde se necesita establecer una correspondencia entre dichos símbolos y dos niveles de tensión ó corriente cualesquiera.

Como es natural, la intención de poder tratar con números es la de realizar operaciones que lleven al resultado requerido.

Dichas operaciones en el campo binario son dos:

Operaciones matemáticas: suma, resta, multiplicación, división, etc.

Operaciones lógicas: negación, intersección, unión, etc..

El primer grupo (operaciones matemáticas) tienen un tratamiento análogo al conocido en formato base diez (números decimales).

Como se vió anteriormente, existen distintos tipos de representaciones de números binarios:

Según su formato: punto fijo y punto flotante.

Según su signo en punto fijo: signo y módulo, complemento a 1, complemento a 2.

Cada uno de ellos tienen sus particularidades donde las reglas de juego son diferentes, las cuales vamos a tratar a continuación.

2 - Operaciones matemáticas en punto fijo.

En este tipo de formato de números binarios, un número es representado por diferentes campos, dependiendo si es con signo o sin signo:

En la representación sin signo, el número está compuesto por dos campos: parte entera y parte decimal (números naturales) ó solo por la parte entera
(números enteros).

En la representación de números binarios en punto fijo con signo, tenemos un campo mas, que es el del signo, tanto para números naturales, como para los enteros.

2.1 Operaciones matemáticas en punto fijo sin signo:

Suma:

Las reglas a tener en cuenta en operaciones de suma son las siguientes:

0 +0 = 0
0 +1 = 1
1+ 0 = 1
1 +1 = 0 $\rightarrow$ 1 = 10

Las mismas son para el caso de sumar dos números binarios de un bit cada uno.

Al igual que las reglas para sumar en base 10, tendremos que hay situaciones en las que el resultado ya no se puede expresar con sólo un símbolo (ej: 1 + 9 = 10). En tal caso existirá un **"carry"** (denominación en inglés) ó transporte a la posición izquierda inmediata.

En el caso binario el carry siempre se produce cuando se suma 1+1, dando un "0" en dicha posición y trasportándose un "1" en la posición del próximo símbolo mas significativo.

El resultado de 1+1 será10, que es igual a 2 en decimal.

Con esto en mente, se hace extensivo para cualquier cantidad de números que se deseen sumar.

Por ejemplo sumando 3 números de 1 bit tendremos:

0+0+0 = 0
0+0+1 = 1
0+1+0 = 1
0+1+1 = 10
1+0+0 = 1
1+0+1 = 10
1+1+0 = 10
1+1+1 = 11 = (1+1 = 0 → 10) + 1 = 11

Observar que en el último caso donde se suman los tres "unos" se hace primero la suma de dos de ellos que da como resultado 10 y luego éste se vuelve a sumar con el otro "uno" obteniendo 11 (3 en decimal).

Esto es importante cuando por ejemplo se deben sumar varios números de mas de un bit.

Por ejemplo: Sumar en binario 7+7+7.

Esto convertido a binario sería:

```
    111
  +
    111
  +
    111
  ------
    ???

 1   1
 1   1   1
     1   1   1
     1   1   1
     1   1   1
------------------
 1   0   1   0   1
```

En "**negritas**" se marcan los transportes que se pasan al otro símbolo al hacer las sumas.

Por ejemplo en la posición del primer bit menos significativo, se suman tres "unos", cuyo resultado es 11, donde el "1" mas significativo de ambos, se debe poner en la posición mas significativa siguiente y sumarlo con los otros 3 "unos" de esa posición.

Como resultado se deben sumar 4 "unos" cuyo resultado en esa posición es "0" y se generan dos "carry" ó transportes que deben sumarse en la posición siguiente (tercera columna menos significativa) y así siguiendo.

Consideraciones de rango:

Siempre que se sumen números de ***n*** bits de formato, el resultado de una suma tendrá ***m*** bits, donde $m \geq n$., dependiendo de la cantidad de números que se sumen:

Por ejemplo con 3 bits, 111+111 dará 1110 (un bit mas); 111+111+110 dará 10100 (dos bits mas), etc.

Este comentario se hace puesto que en general cuando uno trabaja con números binarios lo hace en un ambiente donde el **número de bits** que forman los números está previamente **definido** y son en general tomados de 4, 8, 16, 32 ó 64 bits de extensión, ya que en general esos son los formatos empleados para hacer arreglos de memorias y longitud de palabra de los microprocesadores.

Es necesario entonces, tener presente de que **no nos excedamos** de la cantidad de bits para **representar** no sólo a los números a sumar sino al **resultado**.

Sabemos que un número de ***n*** bits puede representar 2^n números diferentes, siendo el máximo número igual a 2^(n) - 1.

Ejemplo:

Con **4** bits puedo representar **16** números y el 15 es el mayor.
Con **8** bits puedo representar **256** números y el 255 es el mayor.
Con **10** bits puedo representar **1024** números y el 1023 es el mayor.
Con **12** bits puedo representar **4096** números y el 4095 es el mayor.
Con **16** bits puedo representar **65536** números y el 65535 es el mayor.
Con **32** bits puedo representar **4294967300** números y el 4294967299 es el mayor.
etc.

Resta:

Las reglas a tener en cuenta en operaciones de resta son las siguientes:

0 - 0 = 0
0 - 1 = 1 $\rightarrow$ 1
1 - 0 = 1
1 - 1 = 0

Al igual que en el caso de números decimales, cuando el minuendo es menor que el sustraendo, en la resta se genera un transporte hacia la posición de la izquierda, el cual se debe restar a los dígitos de esa posición.

En base 10 al hacer por ejemplo 12 - 9, tendremos que en la posición menos significativa debemos restar 9 al 2, donde el resultado es 3 y me llevo un 1 a la otra posición.

Ese 1 se debe restar al 1 que es el segundo dígito del 12, siendo el resultado 0. El número definitivo en este caso será entonces 03.

Para números binarios, el procedimiento es similar.

Como ejemplo tenemos que hacer la operación 11000 - 111 (24 - 7 en decimal).

La operación será:

	1	**1**	**1**	
1	1	0	0	0
		1	1	1
1	0	0	0	1

Como en el caso de la suma, ponemos en "**negritas**" a los transportes (ó **"borrow"** como se le denomina en inglés al transporte en una resta).

En la posición menos significativa tenemos que realizar 0 -1 que dará como resultado un 1 y un borrow (transporte) a la posición de la izquierda.

En la misma tendremos que hacer 0 -1 -**1**, lo cual dará 0 y un transporte hacia la siguiente posición.

En la tercera columna desde la derecha tendremos la misma operación anterior: 0 -1 -**1**, dando nuevamente 0 y un transporte.

En la cuarta columna menos significativa la operación de resta será: 1-**1**, siendo 0 el resultado y no habiendo transporte.

Por último en la quinta columna desde la derecha el resultado es 1, siendo el número final 10001 = 17 en decimal

NOTA: En el caso en que el minuendo sea de menor módulo que el sustraendo, la operación sólo puede realizarse restando al de mayor módulo el de menor módulo y luego colocando un bit mas para el signo que corresponda.

Multiplicación:

Esta operación se realiza de igual manera que con los números en base 10, es decir, dados dos números A de ***n*** dígitos y B de ***m*** dígitos, se procede a multiplicar cada dígito de B (empezando de derecha a izquierda) con los ***n*** dígitos de A, donde los resultados parciales de cada multiplicación se deben ir poniendo en fila pero corriendo la posición de los mismos en cada operación.

Posteriormente se debe realizar la suma de las ***m*** filas resultantes.

Como ejemplo tenemos en decimal que multiplicar 785 x 12:

		1	**1**	
	7	8	5	
		1	2	
	1	**1**		
1	5	7	0	
7	8	5		
9	4	2	0	

Primero se realiza la operación 785 x 2 y luego 785 x 1 donde este último resultado se coloca debajo del primero, desplazado un dígito hacia la izquierda.

Posteriormente se suman ambos y se obtiene el resultado definitivo.

En binario se procede en forma análoga. Por ejemplo realicemos la operación 11111 x 1101 (31 x 13 en decimal):

				1	1	1	1	1
					1	1	0	1
	1	**1**	**1**	**1**				
1	**1**	**1**	**1**	**1**	**1**			
				1	1	1	1	1
			0	0	0	0	0	
		1	1	1	1	1		
	1	1	1	1	1			
1	1	0	0	1	0	0	1	1

El 1 menos significativo de 1101 se multiplica por 11111 dando como resultado: 11111 (todo número multiplicado por 1 da ese mismo número).

Luego se multiplica el 0 de 1101 por 11111, dando 00000, el cual se coloca debajo del anterior resultado pero corrido un lugar hacia la izquierda.

Repitiendo estos pasos para los dos dígitos restantes del número 1101, se completa la tabla, en la cual luego resta sumar bit a bit como vimos.

El resultado final es 110010011 ó 403 en decimal.

División:

En este caso se deben realizar operaciones de multiplicación y de resta también como en el caso de números decimlaes.

Por ejemplo si deseamos hacer 14 dividido 2 en binario, tendremos:

1	1	1	0	1	0	
1	0			1	1	1
0	1	1				
	1	0				
	0	1	0			
		1	0			
			0			

El resultado es un número entero, 111 (ó 7 en decimal), el cual es correcto.

Si en cambio realizamos la operación 45 dividido 6, tendremos:

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

Tendremos aquí que el número resultante (7,5 en decimal) es exacto pero con parte decimal.

Puede suceder, que al dividir dos números, el resultado tenga un número grande de dígitos decimales pudiendo llegar a infinito (números periódicos).

Esto sucede cuando el resultado no es múltplo de potencia de 2^-n, como por ejemplo 0,5, 0,25, 0,125, etc.

En decimal tenemos por ejemplo que 10 dividido 3 nos dá: 3,333..... con infinitas cifras decimales. Igual puede pasar en formato binario.

En tal caso dependiendo de la exactitud que se requiera se deberán tomar los bits necesarios para que el error de representación esté acotado en un valor admisible para la operación en cuestión.

2.1 Operaciones matemáticas en punto fijo con signo:

Se ha visto que existen tres tipos diferentes de representación de números binarios con signo en punto fijo.

Estas son:

Signo y módulo.

Complemento a la base disminuída (complemento a 1: Ca1).

Complemento a la base (complemento a 2: Ca2).

Operaciones con el método de Signo y módulo

De los tres casos mencionados, realizar operaciones de suma y resta con signo y módulo requiere de un hardware lo suficientemente complejo para que pueda distinguir no sólo el tipo de operación a realizar (suma ó resta) sino el signo de los operandos.

Por ejemplo al sumar dos números de distinto signo, o al restar dos números de igual signo, en realidad se debe hacer la resta del número de mayor módulo al de menor módulo, debiéndose poner el signo del resultado correspondiente al número de mayor módulo.

Si la operación es la suma de dos números de igual signo ó la resta de dos números de distinto signo, la operación debe ser de suma donde el signo es el de cualquiera de los dos números (en el primer caso) ó el del primer operando (en el segundo caso).

Por el contrario, el método de signo y módulo se emplea para las operaciones de multiplicación y división, las cuales serían muy complejar de realizar en Ca1 y Ca2.

Operaciones con el método de Complemento a la base disminuída (Ca1):

Como se vió, el método de representación en Ca1, permite la representación de números binarios con signo.

Los números positivos se representan colocando un "0" como primer bit mas significativo al módulo del mismo, mientras que los números negativos se representan empleando la definición de complemento a 1, es decir, dado un número N de n bits, su complemento N*1, será 2^n - 1 - N. Donde en este caso N es el número inicial (positivo ó negativo) y N*1 su complemento (negativo ó positivo) que se quiere buscar.

NOTA: Es necesario recalcar que la definición de complemento significa que dado un número ya sea positivo ó negativo, al hallar su complemento obtenemos un número de signo opuesto pero de igual módulo.

Dada esta definición, entonces, cualquier operación puede resumirse como una suma.

Dados A y B, si hay que sumar, se suma como se vió en operaciones sin signo.

Dados A y B, si son de distinto signo, si hay que restar A - B, entonces se complementa B y se suma a A, es decir, A - B = A + (-B).

Siempre se suma, nunca se resta. Si hay un signo (-) adelante de un operando, entonces ese operando se complementa.

Ejemplos:

A = 15 decimal y B = -13 decimal.

Realizar las siguientes operaciones con 5 bits:

A + B, A - B, B - A y B + A.

A + B: En este caso se suman ambos números.

A como es positivo será = 01111.
B como es negativo y se debe sumar a A, se calcula en base al +13 y se complementa, es decir, +13 = 01101 y su complemento por definición será:

```
011111 = 2^5 - 1
001101 = +13
------------
010010 = B
```

Por lo tanto: A + B = 01111 + 10010 = 00001

```
01111 = A = +15
10010 = B =  -13
------------
00001 = A + B = +1
```

El resultado es incorrecto ya que debería haber dado +2 decimal ó 0010 en binario.

La razón de esto se puede ver en el disco de representación de numeros en Ca1, donde por simplicidad tomamos otro ejemplo de 3 bits, donde la operación es +3 -1, debiendo dar +2, pero dá +1.

```
011 = +3
110 =  -1
---------
001 = +1
```

.

Recordando que existe una doble representación del cero, hay casos como éste, donde cuando el resultado de una resta A - B es mayor que 0, al resultado se le debe sumar un "1" para que sea correcto.

En el disco, las operaciones de Ca1 se hacen siempre sumando, es decir, en el sentido de las agujas del reloj, por lo tanto si comenzamos parados en -1 = 110 y avanzando 3 números (+3), en sentido horario, tendremos que el resultado es +1.

Porqué? Porque hemos pasado por la doble representación del cero, primero por el -0 = 111 y luego por el +0 = 000.

Por lo tanto, en vez de llegar al +2 que era el resultado correcto, llegamos a un número menos (el +1).

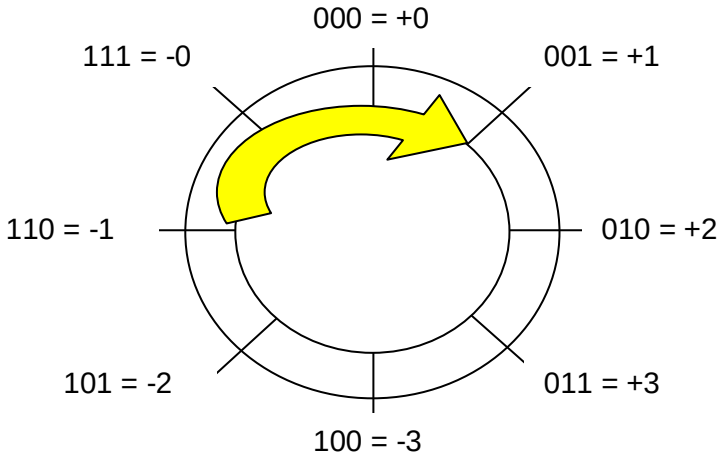

Figura 1

Esto es extensible al ejemplo que estabamos viendo (+15 - 13), donde el resultado era un número menos del esperado (+1 en vez de +2).

Si en cambio la resta de dos números da como resultado un número menor que 0, no existe este problema ya que nunca llegaremos a pasar por la doble representación del 0.
En este caso el resultado obtenido será el correcto.

Viendo esto en el ejemplo del disco de representación, si por ejemplo queremos hacer 2 - 3 = 2 + (-3), tendremos que partiendo de -3 = 100 y en sentido horario recorriendo dos posiciones (+2), llegamos a 110 que representa al -1, siendo el resultado correcto.

Como conclusión cuando en una resta de dos números el resultado es positivo (mayor que cero), se debe sumar al mismo un 1, que es el transporte de haber sumado dichos números, caso contrario el resultado será incorrecto.

Si el resultado de la resta es menor que cero, no se debe realizar ajuste alguno.

Cuando se suman dos números negativos también se debe sumar al resultado el carry si lo hubo para que el resultado sea correcto:

Por ejemplo si tenemos -2 -1 = -2 + 2*1 = 101 + 110 = 1←011 que es positivo.
Aquí no hay overflow (desborde) ya que el -3 que debería haber dado esta dentro de la capacidad de representación con 3 bits.
Para que el resultado sea correcto faltaría sumar el carry al 011:

```
   101 = -2
   110 = -1
  1←011
     1
   100
```

Matemáticamente podemos también explicar esto a través de las siguientes expresiones empleando la definición de Ca1:

Dados dos números A y B, si el resultado de A - B es positivo, tendremos:

$$A - B = A + B*1 = A + 2^n - 1 - B = A - B + 2^n - 1$$

es decir que A - B en realidad se compone de los términos de resta mas el término 2^n - 1.

para que quede perfectamente representado debemos sumar ese "1" ya que el término 2^n no tiene efecto debido a que escapa fuera del rango de representación con n bits.

Si en cambio tenemos que el resultado de A - B es negativo, esto se expresa como:

$$A - B = - (B - A) = (B - A)*1 = 2^n - 1 - (B - A)$$

donde queda perfectamente expresado por la definición de complemento y no debe sumarse ningún transporte al resultado.

A - B: En este caso como B es negativo se lo debe complementar a fin de realizar la suma.

A - B = A + (-B) = A + B*1

B*1 = 2^5 - 1 - B= 01111 - 10010 = 01101

A - B = 01111 + 01101 = 11100

$$\begin{array}{r} 01111 = A \\ \underline{01101 = - B = B*1} \\ 11110 = A + B*1 = - 3 \end{array}$$

El resultado deseado debería ser +15 + 13 = + 28, pero dió - 3.
Porqué?

La respuesta es porque nos excedimos de la capacidad de representación con los 5 bits que tomamos.

Recordando que en Ca1, el máximo número a representar con 5 bits era empleando la fórmula: + 2^(n-1) - 1 = + 2^4 - 1 = + 15, es razonable que el +28 no entre en este formato.

Se deben emplear al menos 6 bits para poder realizar esta operación
(número máximo positivo = +31 = 2^5 - 1 para n = 6).

Siempre es posible detectar este ***overflow*** (exceso de la capacidad de representación) de dos maneras diferentes:

1) Observando que en suma de dos números de igual signo ó resta de dos números de distinto signo, el resultado sea de signo contrario.

2) Que los estados de los carry (transportes) que hay entre la posición mas significativa y la siguiente sean diferentes.

En el primer caso se ve que al hacer +15 + 13 el resultado tiene un "1" en el bit mas significativo (es negativo).

En el segundo caso se observa que de la posición del cuarto dígito menos significativo hay carry hacia la última posición, pero desde esta última hacia la izquierda no la hay, por lo tanto los estados de carry son diferentes (hubo en uno y en el otro no), por lo tanto hubo **overflow**.

El caso 1 de detección es mas intuitivo para el ser humnao ya que puede rápidamente saber si hay o no overflow.

El caso 2 es empleado en los circuitos digitales tales como la UAL (unidad aritmético lógica) de los microprocesadores por ser mas fácil de implementar que el caso1.

NOTA: Este análisis es válido también para números representados en Ca2.

B - A: En este caso se debe complementar a A para realizar una operación de suma.

B - A = B + (-A) = B + A*1 = 10010 + 10000 .

```
   10010
   10000
  ------
 1←00010
```

El resultado es incorrecto pues el resultado es positivo en vez de ser negativo.
El 1 que se transporta a la izquierda es un carry pero no interviene en la representación.

Otra vez tenemos **overflow** ya que no podemos representar el -28 pues el máximo número negativo es el : - (2^(n - 1) - 1) que para 5 bits, es -15.
Habrá que emplear 6 bits para poder hacerlo (con 6 bits se puede representar hasta el -31).

B + A: Aquí por propiedad conmutativa tenemos el mismo razonamiento que con la operación A + B.

NOTA: En este último ejemplo se ha puesto de evidencia que la existencia de carry no implica desborde en Ca1 y tampoco en Ca2 como se verá.
En el único caso en que el carry representa desborde es en la representación de números sin signo, ya que si se tienen n bits y hay un "1" que se transporta a la posición n+1, eso indica que el resultado expresado con esos n bits es incorrecto ya que necesitaría un bit mas (ejemplo de sumar sin signo con 4 bits 1110 + 10, el resultado es 10000 pero como solo tengo 4 bits voy a leer 0000.

Operaciones con el método de Complemento a la base (Ca2):

A diferencia de Ca1 en Ca2 tenemos que existe una sola representación para el cero y esta es con todos los n bits de formato en 0.

La **primera ventaja** de esto es que disponemos de un número mas para representar los números negativos (el lugar de todos "unos" que antes en Ca1 representaba el - 0).

La **segunda ventaja** es operativa y muy importante, y es que no hay que tener en cuenta mas si el carry se debe sumar al resultado o no, para que éste sea correcto.

Esto simplifica la lógica que se debe implementar en los sumadores binarios.

Como se vió, la definición de complemento a la base de un número binario en punto fijo de ***n*** bits es: N*2 = 2^***n*** - N.

Al igual que en el caso de Ca1, las operaciones de suma y resta siempre serán de suma, debiéndose complementar al número que figure como negativo.

Ejemplo:

Dado A = -16 y B = - 26, realizar las operaciones en 6 bits:

A + B, A - B, B - A.

Para representar el -16 y el -26 en binario partimos del número +16 y del +26 y luego los complementamos.

16 = 010000

A = -16 = 16*2 = 2^6 - 16 = 100000 - 010000 = 110000

```
  1000000 = 2^6
   010000 = A
  ------
   110000
```

-16 = 110000

Por otro lado,

26 = 011010

B = -26 = 26*2 = 2^6 - 26 = 100000 - 011010 = 100110

```
  2^6 = 1000000
    B =  011010
  ---------------
       100110
```

-26 = 100110

A + B: -16 +(-26). Se suman directamente los dos números:

```
   A = 110000
   B = 100110
  ------------
  1←010110
```

En este caso nos encontramos que como en el ejemplo visto de Ca1, la suma de dos números negativos da como resultado un número positivo, es decir, hubo **overflow.**

Además hubo un **carry** desde el bit mas significativo, el cual ***no indica*** una condición de desborde u overflow.

Esto como se dijo, es importante de tener en cuenta.
El "1" que se transporta del sexto bit a la izquierda, en este caso, se escapa fuera de la representación con 6 bits, y no es necesario tenerlo en cuenta.

Por ejemplo si sumamos los siguientes números 11100 (- 4) y 01000 (+8)
(uno negativo y el otro positivo) nunca se puede dar un desborde, pero sin embargo
si hacemos la suma, tendremos un carry, aunque el resultado sea correcto (+4).

```
    11100
    01000
  ---------
  1←00100
```

A - B: En este caso se debe complementar B para sumarlo a A: A + B*2.

B*2 = - (- 26) = + 26.

B*2 = 011010

A - B = 110000 + 011010 = 001010 = + 10 en decimal.

```
   110000 = A
   011010 = B*2
  ------------
1←001010
```

Aquí hay otro ejemplo donde no hay desborde pero si carry.

B - A: En este caso hay que complementar a A antes de sumarlo a B:

B + A*2 = B + 2^6 - A

A*2 = 010000

```
B    = 100110
A*2 = 010000
  ------------
    110110
```

3 - Operaciones matemáticas en punto flotante.

Como se ha visto, los números en punto flotante se representan por tres campos: significando ó fracción del significando, signo del significando y el exponente.

Las operaciones de multiplicación y división son mas faciles de efectuar que las de suma y resta, como veremos a continuación.

Multiplicación y división:

Como se sabe si se tienen dos números expresados en punto flotante de la forma **m** 2^**n** y **o** 2^**p**, donde **m** y **o** son significandos y **n** y **p** los exponentes, en una multiplicación se deben multiplicar los significandos y sumar algebraicamente los exponentes:

m 2^n * o 2^p = (m * o) 2^(n + p)

En una división se deben dividir los significandos y restar los exponentes:

m 2^n / o 2^p = (m / o) 2^(n - p)

Aplicando esto a la representación en **punto flotante normalizado**, cuando se debe realizar por ejemplo una multiplicación, se deben tener en cuenta dos puntos importantes:

1) Los significandos se deben multiplicar, pero éstos están representados sólo con la parte decimal, donde tácitamente el entero es un "1" (simple y doble precisión).

Por lo tanto se debe desnormalizar los significandos poniéndolos en formato 1,x.....x y luego multiplicarlos.
Al resultado se lo vuelve a expresar con solo la parte decimal.

Hay que tener cuidado porque cuando se multiplican dos números que varían entre 1,0 y 1,1.....1 (casi 2), puede ser que el resultado tenga dos dígitos (10,x.....x ó 11,x.....x). En tal caso se debe volver a ponerlo en formato normalizado pero antes se debe correr de nuevo la coma, es decir, corregir el exponente sumandole un "1".

2) Los exponentes se deben sumar algebraicamente, pero como éstos están formateados con el bias, se deben antes desnormalizar y luego sumar.
El resultado de esa suma se debe luego formatear con el bias salvo que como se dijo en el punto 1), si de la multiplicación de los significandos hubo un resultado de la forma 10,x...x ó 11,.x...x, entonces se debe sumarle un "1" antes y luego formatear.

El caso de división es análogo, donde puede ser que al dividir los significandos el resultado de menor que 1, y se deba correr la coma hacia la izquierda, es decir, restar un "1" al resultado luego de restar los exponentes.

Suma y resta:

Sólo es posible realizar estas operaciones cuando los números en punto flotante tienen el mismo exponente.
En tal caso se deben sumar o restar, según la operación, los significandos previa desnormalización.

El resultado se debe volver a normalizar, teniendo en cuenta que debe ser de la forma 1,x.....x (precisión simple y doble), pudiendo ser necesario ajustar de nuevo los exponentes.

Si los números tienen diferente exponentes, entonces necesario expresar uno de los números con el mismo exponente del otro, lo cual implica que se perderá resolución, ya que se tiene que correr la coma a derecha o izquierda, según el caso.

Es posible que al realizar esta operación, si los exponentes son muy diferentes se cometa grandes errores.

Por ejemplo si un número tiene exponente +12 y otro -15 (la diferencia es de 27) y se deben sumar, se estaría fuera de la capacidad de representación en punto fijo simple precisión donde sólo se puede representar a la fracción del significando con 23 bits.
Queda en este ejemplo emplear una representación de mayor resolución como doble precisión ó precisión extendida.

4 - Operaciones con números BCD

Suma:

Esta operación se realiza con las mismas reglas de suma de números binarios.
Ejemplo: sumar los números 23 + 51 en BCD

```
  23 = 0010  0011
  51 = 0101  0001
 ----------------
  74 = 0111  0100
```

Pero hay casos donde hacer esto genera errores, por ejemplo sumando 2 + 8, tendremos:

```
  02 = 0010
  08 = 1000
 ----------
  10 = 1010
```

La suma es correcta ya que el resultado debe dar 10, pero el formato es BCD y no binario. Lo correcto es obtener 0001 0000.

El problema se debe a que al sumar dos números de 4 bits, recién hay un carry cuando se pasa de 15 a 16, en cambio como en BCD se sigue la regla en base diez, se debe generar el carry al pasar de 9 a 10.

Para obtener el resultado correcto en BCD pero trabajando con números binarios, se deb realizar lo que se denomina ***ajuste BCD.***
Cuando el resultado de la suma es mayor a 9 se debe sumar 6 al mismo a fin de producir el carry deseado.

En el ejemplo anterior, hacemos entonces:

```
02 =      0010
08 =    + 1000
        ------
          1010
        +  0110
        ------
       1  0000
```

5 - Bibliografía.

1 Circuitos Digitales y Microprocesadores. Herbet Taub.
Editorial Mc Graw Hill.
2 Sistemas Digitales: Principios y aplicaciones. Ronald Tocci.
Editorial Prentice - Hall.
3 Norma IEEE P754.
4 Electrónica Digital. James W. Bignell.
Editorial CECSA.