Práctica 1

1. Técnicas para interpretar y representar números binarios en distintos sistemas de representación

Para el **BSS**, la suma de potencias: cuando tenemos que representar un número, buscamos la potencia máxima de dos que sea igual o menor a ese número, luego se la restamos, y seguimos repitiendo el proceso hasta llegar a alguna potencia de dos, que debe anotarse.

Ejemplo: representar el 42 en 8 bits. Buscamos la potencia máxima de dos que sea igual o menor a 42. 64 no es, porque nos pasamos, así que es 32. Después restamos 42 - 32 = 10. Buscamos la siguiente potencia: 8. Restamos 10 - 8 = 2, y esa sería la última. Entonces 42 = 32 + 8 +2, que en 8 bits queda 00101010. Lo cual puede graficarse de la siguiente manera:

128	64	32	16	8	4	2	1	Resultado
0	0	1	0	1	0	1	0	42

El rango del BSS es $[0, 2^n - 1]$, que en 8 bits sería: [0, 255].

Para el **BCS** lo mismo, pero el primer bit es para el signo. O sea que, volviendo al mismo ejemplo, 42 = 00101010 y -42 sería 10101010. Por otro lado, esto implica que tenemos dos representaciones para el cero: +0 = 00000000, y -0 = 10000000, y que el rango es simétrico: $[-(2^{n-1} - 1), 2^{n-1} - 1]$. Que en ocho bits viene a ser [-127, 127].

Para el **Ca1**, los números positivos (aquellos cuyo primer bit es 0) son iguales a las representaciones anteriores, así que 42 quedaría igual. Pero los negativos no. Los negativos son, justamente, el complemento a uno del número positivo. O sea que si 42 = 00101010, -42 = 11010101.

¿Qué significa que sea el complemento a uno? Que si yo sumo dos números opuestos en este sistema, obtengo todos unos. Así:

```
  00101010    (42)
+ 11010101    (-42)
  11111111
```

Y para interpretar un número negativo en Ca1 (es decir, que empieza con 1), lo más sencillo es interpretar –en el resto de la cadena– los ceros como unos, y los unos como ceros. De esta

manera no hay que andar reescribiendo el número. Uno fácilmente puede notar que, por ejemplo, 11110011 es el -12, ya que el primer bit es 1, o sea que es negativo, y los que están en 0 son el 8 y el 4; y 8 + 4 = 12.

Aquí también tenemos dos ceros (+0 = 00000000, y -0 = 11111111), y rango simétrico (igual al de BCS).

En el **Ca2**, con los positivos sucede lo mismo que antes. Con los negativos no. Para representar un número negativo en Ca2, primero debemos obtener el Ca1 y después sumarle 1. Entonces, para llegar al -42, hacemos lo siguiente:

```
  11010101    (-42 en Ca1)
+ 00000001    (1 en BSS)
  11010110    (-41 en Ca1, -42 en Ca2)
```

En parte por eso, hay una sola representación del cero (= 00000000), y el rango es asimétrico = $[-(2^{n-1}), 2^{n-1} - 1]$. En 8 bits nos queda: [-128, 127].

Pero para hallar el número hay dos técnicas más eficientes, dependiendo el caso en el cual nos encontremos. Una es tomar el número en BSS, y yendo de derecha a izquierda dejar los bits tal como están, hasta el primer 1 (inclusive), y después invertir el resto. Por eso, tomando el 42 en BSS = 00101010, el -42 en Ca2 nos quedaría 11010110.

Otra forma es interpretar toda la cadena como una suma de potencias, solo que el bit más significativo está en negativo. De esta manera, comprendemos rápidamente que 10000000 = -128, o que 11111111 = -128 + 127 = -1. Sería como calcularlo con esta tabla:

-128	64	32	16	8	4	2	1	Resultado
1	1	0	1	0	1	1	0	-42

En el **Exceso con 2^{n-1}** todo es diferente. Los positivos empiezan con 1, los negativos con 0. ¿Por qué? Porque en la recta numérica, el cero está corrido. Aquí el cero, si tenemos –por ejemplo– 8 bits, no se escribe como ocho ceros, sino que se escribe como $2^{8-1} = 2^7 = 128$ en BSS, o sea, así: 10000000. O sea que para representar un número en este sistema, primero debemos sumarle el exceso y después escribir ese resultado como si estuviéramos en BSS. Por eso, para escribir el 0 hacemos 0 + 128 = 128, y escribimos lo de recién. Y para el 5 por ejemplo, hacemos 5 + 128 = 133, y escribimos 10000101.

Hay un par de técnicas para interpretar rápidamente en Ex2. Una es tomar el Ca2 del número en cuestión, e invertirle el primer bit. Revisemos el caso del -42. En Ca2 es 11010110, así que

en Ex2 será 01010110. Advertencia: esto solo funciona cuando tenemos la misma cantidad de bits en ambos sistemas (en este caso, 8 bits). De lo contrario, no funciona.

La otra manera es ver con qué número empieza. Si empieza con 1, lo tapamos con el dedo, y leemos el resto del número como si fuera en BSS. Si empieza con 0 leemos el número como si estuviera en BSS, pero después le restamos el exceso, o sea, 128 (si es que estamos en 8 bits). Así, el 10000101 lo leemos como 0000101 en BSS, que es 5, y al 01010110 en BSS y después le restamos el exceso, con lo cual nos queda 2 + 4 + 16 + 64 - 128 = 86 - 128 = -42.

En este sistema, el rango es igual al del Ca2.

2. Rango y resolución en sistemas de punto fijo

El rango está dado por los extremos de la recta numérica que pueden representarse, es decir, por el número mínimo y el número máximo que podemos representar. No hay grandes diferencias con los rangos de los propios sistemas numéricos vistos anteriormente.

La resolución, en los sistemas de punto fijo, es constante, no varía en ningún sector de la recta numérica. La resolución es la distancia que tenemos entre dos números representables, la cual está dada por el bit menos significativo. Si tenemos, por ejemplo, 8 bits de parte entera en BSS y ninguno de fraccionaria, la resolución es 1, porque la distancia entre un número representable (por ejemplo el 7) y el siguiente (o sea, el 8), es 1; la otra manera de entenderlo, es notando que el bit menos significativo está –por así decir– en la posición 2^0, que vale 1. Si, en cabio, tuviéramos 8 bits de parte entera y 3 bits para la parte fraccionaria, la resolución sería 2^{-3} o 0,125, ya que el siguiente de 7 sería 7,125 –y, obviamente, aquí el bit menos significativo vale justamente eso: 2^{-3}.

3. BCD y BCD empaquetado

El *Binary Coded Decimal* es una manera de correlacionar directamente los digitos decimales con los números binarios. Como los dígitos decimales son 10, y 10 no es potencia de 2, se desperdician algunas combinaciones de las 16 que podemos armar con 4 bits, pero aparentemente tiene algunas ventajas prácticas (que no estudiaremos ahora).

Si decimos BCD, nos referimos al BCD desempaquetado, el que ocupa 8 bits (= 1 byte) por cada dígito. Los cuatro primeros son unos, y el resto son el número en binario. (El subrayado es para que se note dónde está almacenado efectivamente el dígito).

Ejemplos: 0 = 1111<u>0000</u>, 12 = 1111<u>0001</u>1111<u>0010</u>, 100 = 1111<u>0001</u>1111<u>0000</u>1111<u>0000</u>.

En el BCD empaquetado, cada dígito ocupa la mitad, así que entran uno o dos por byte. Si hay solo uno, lo demás se rellena con ceros.

Ejemplos: 0 = 00000000, 1 = 00000001, 12 = 00010010, 100 = 0000000100000000

Normalmente, en esta materia, las operaciones matemáticas que se hacen en BCD, se hacen con el BCD empaquetado.

¿Cómo sumamos? Dígito a dígito, donde cada uno de ellos se trata como las sumas en binario de toda la vida, con la diferencia de que si alguno dió más de 9, tenemos que hacer algo especial, sino caemos en un fatal error. Pero es algo sencillo, es lo mismo que hacemos desde la primaria, con algún que otro truco.

Si tenemos que sumar

```
  19
+ 13
```

claramente notaremos que al sumar los dígitos de las unidades, o sea, el 9 y el 3, nos pasamos de 9; concretamente, nos da 12. Pero no anotamos el 12 abajo. Lo que hacemos automáticamente –sin pensar– es anotar el 2 en las unidades (el cual podemos pensarlo como 12 - 10), y "llevarnos" eso que restamos (el 10) en forma de 1 a las decenas; el cual, por estar en esa posición, valdrá 10. Y así es como sumamos desde siempre.

```
   1
  19
+ 13
  32
```

Lo mismo hay que hacer cuando operamos en BCD. Pongamos por caso

```
  22    0010 0010
+ 89    1000 1001
        1010 1011 →10 y 11
```

Pero ni 10 ni 11 son dígitos decimales, así que, en ambos casos, le restamos 10 y se lo pasamos al siguiente, en forma de 1. Lo haremos, naturalmente, de derecha a izquierda.

```
    1010  1011 →10 y 11
+ 0001 - 1010 → +1 y -10
    1011  0001 → 11 y 1

    1011  0001 → 11 y 1
+ 0001 - 1010 → +1 y -10
  0001  0001  0001 → 111
```

Así que ahora sí: 22 + 89 = 111.

Ahora bien, hay una manera de simplificar este proceso aprovechando la relación entre el "desbordamiento en decimal" y la numeración binaria de 4 bits. ¿Cómo sería eso? Al usar 4 bits, tenemos 16 dígitos, de los cuales el sistema decimal solo usa 10, y quedan 6 de más. Si

en algún momento durante una suma, un dígito tiene más de 9, hay que sumarle esos 6 para que el dígito "desborde" ese valor con un carry, y que lo tome el siguiente. A fin de cuentas, es exactamente lo mismo que hicimos recién, pero simplifica más la escritura. Sería algo así:

```
   22     0010 0010
+  89     1000 1001
         1010 1011 →10 y 11
```

```
       1  11  1 11    → Los 1 en negrita son de carry
          1010 1011 →10 y 11
+         0110 0110 →6 y 6
     0001 0001 0001 → 1, 1 y 1 → 111
```

Por último, falta darle el toque final: que el resultado sea múltiplo de 8 bits. Para ello, rellenamos lo que falta con ceros, y nos queda: 0000000100010001.

También existe el BCD con signo: la $C_{(16)} = 1100_{(2)}$ es el + y la $D_{(16)} = 1101_{(2)}$ el -.

En el BCD desempaquetado con signo, este se almacena en el relleno del último dígito. Entonces, por ejemplo, el 834 = 11111000 11110011 11000100 (lo que está subrayado es el signo); y el -834 = 11111000 11110011 11010100.

En el BCD empaquetado con signo, primero van los dígitos y por último el signo, entonces los números anteriores nos quedan así: 834 = 10000011 01001100 y -834 = 10000011 01001101. Si los bits de los dígitos y del signo no ocuparan exactamente un múltiplo de 8 bits (como es el caso del -34 = 0011 01001101), rellenamos con ceros a la izquierda, como los que marcamos en negrita a continuación: **0000**0011 01001101 = -34.

4. De binario a hexadecimal y viceversa

Hay una correspondencia directa (*i.e.*, uno a uno) entre los números que se pueden formar con 4 bits y los 16 dígitos del sistema hexadecimal. Por eso, podemos convertir números directamente de un sistema a otro sin pasar por el sistema decimal.

Para ir de binario a hexadecimal, juntamos los bits en grupos de cuatro yendo de derecha a izquierda, y si en el último grupo de la izquierda nos quedan menos de 4 bits, rellenamos con ceros (a la izquierda, para no alterar el número). Hecho esto, cambiamos cada uno de esos cuatro bits por sus correspondientes dígitos en hexadecimal, y listo. Veamos un ejemplo: $1010000010000_{(2)} \rightarrow 1\ 0100\ 0001\ 0000 \rightarrow 0001\ 0100\ 0001\ 0000 \rightarrow 1410_{(16)}$.

Para ir de hexadecimal al sistema hexadecimal codificado en binario (o sea, a BCH), sólo hay que hacer el camino inverso, *poniendo siempre 4 bits por cada número hexadecimal*, como en el siguiente ejemplo: $2A_{(16)} \rightarrow 00101010_{(2)}$.

5. Sumas, restas y banderitas

Empecemos por las sumas. 1 + 0 = 0 + 1 = 1, 0 + 0 = 0, pero $1 + 1 = 10_{(2)}$ y $1 + 1 + 1 = 11_{(2)}$. Cuando el resultado es un número de dos dígitos, bajamos el de la derecha y el de la izquierda "nos lo llevamos" para el siguiente bit –o sea, al de la izquierda. Solamente cuando eso sucede al final, es que se llama carry. O sea, cuando ese 1 que nos llevamos no entra en los bits que tenemos. Veamos algunos ejemplos de sumas:

1	11	111	1111	11111	11111
00011101 →	00011101 →	00011101 →	00011101 →	00011101 →	00011101
<u>00011011</u>	<u>00011011</u>	<u>00011011</u>	<u>00011011</u>	<u>00011011</u>	<u>00011011</u>
0	00	000	1000	11000	00111000

En este caso, los flags quedan:

C = 0, porque no hubo carry
Z = 0, porque el resultado no son todos ceros
N = 0, porque el resultado no es negativo (o sea, no empieza con 1)
V = 0, porque de la suma de dos positivos, obtuvimos un resultado positivo

Los primeros dos flags pueden interpretarse independientemente del sistema, pero los últimos dos tienen sentido y están pensados para cuando estamos en Ca2. Así que lo mejor es interpretar todos los flags en Ca2 y listo. Pasemos a otro ejemplo:

1111
10011101
<u>01110010</u>
00001111

En este caso, los flags quedan: C = 1, Z = 0, N = 0, V = 0.

¿Cuándo se activa el overflow? Cuando, interpretando en Ca2, pasa lo que no tiene que pasar. En la suma, se activa cuando de dos positivos sale un negativo, o de dos negativos sale un positivo. En la resta, cuando a un positivo le restamos un negativo, y da negativo; y cuando a un negativo le restamos un positivo, y da positivo.

Los flags pueden dar mucha información, pero lo que importa acá es esto: si se activó el carry, la cuenta, interpretada en BSS, salió mal; si no, salió bien. Y si se activó el overflow, la cuenta, interpretada en Ca2, salió mal; si no, salió bien.

Pasemos a las restas. 1 - 0 = 1, 1 - 1 = 0 = 0 - 0, pero 0 - 1 no se puede realizar sin antes pedir prestado al de la izquierda, y $10_{(2)} - 1 = 1$. También se puede pedirle prestado (*borrow*) a un bit imaginario que está a la izquierda de todo. Cuando eso sucede, se activa el carry. Primero un ejemplo sencillo (donde la x indica que se trata de una operación no permitida):

```
                    0           0
01110110 →  011101+[1]0 → 011101+[1]0
01110001    0111000 1   0111000 1
      x           1     00000101   →  C =0, Z = 0, N = 0, V = 0
```

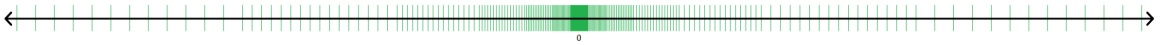
Otra un poco más rara sería:

```
00000000 → +[1]00000000
10000000    10000000
x0000000    10000000   →   C =1, Z = 0, N = 1, V = 1.
```

Práctica 2

6. Mantisa, base y exponente: los ingredientes del punto flotante

El punto flotante permite ampliar el rango sin gastar tantos bits, y, para lograrlo, deja que la resolución varíe a lo largo de la recta numérica. Lo cual es bastante conveniente, ya que normalmente solo necesitamos alta resolución con números cercanos al cero, y poca o nada cuando hablamos de miles o millones. Gráficamente, la distribución de los números en la recta numérica sería algo así:



O sea: mucha densidad de números cerca del 0, y poca en los extremos.

En pocas palabras, el punto flotante consiste en implementar la notación científica con números binarios. Así que empecemos por repasar dicha notación.

Pensemos en un número grande, como el 96000000000. Si yo multiplico a este número por uno, no lo modifico, así que hagamos eso, pero en vez de poner uno, pongamos 10^0. Con lo cual tenemos $96000000000 = 96000000000 \cdot 10^0$. Pero a los ceros –cuando están escritos así, todos seguidos–, en el sistema decimal podemos "representarlos" como potencias de 10, que es la base del sistema. (En realidad lo que estamos haciendo es dividir a la mantisa por la base *n* cantidad de veces, y multiplicar a la base por ella misma (*n* cantidad de veces); en este caso 9). Entonces podemos reescribir el número en cuestión como $96000000000 \cdot \frac{10^9}{10^9} = \frac{96000000000}{10^9} \cdot 10^9 = 96 \cdot 10^9$. En este ejemplo, el 96 es la mantisa, el 10 es la base, y el 9 el exponente. También podríamos, desde ya, escribirlo como $9,6 \cdot 10^{10}$.

¿Qué tiene esto de flotante? Que uno puede "hacer flotar" el punto (o la coma) hacia la izquierda o a la derecha, siempre y cuando uno haga los cambios correspondientes en el exponente. En el ejemplo anterior, la coma estaba implícita al final (ya que es un número muy grande, sin decimales), y pasó a estar 9 posiciones a la izquierda. Por cada una de esas posiciones, el exponente incrementó en uno el número en el cual se encontraba. Lo mismo vale para números muy pequeños, donde la coma debe correrse a la derecha (con lo cual la mantisa aumenta su valor) y el exponente debe achicarse para mantener el resultado final.

En resumen, el equilibrio debe mantenerse: si la mantisa se achica, se agranda el exponente, y si la mantisa se agranda, el exponente disminuye.

Este esquema general de $M \cdot B^E$, en binario nos queda $M \cdot 2^E$.

Algunas aclaraciones:

- Como la base siempre es la misma (siempre es 2), no se almacena.
- La mantisa y el exponente pueden estar expresados en diferentes sistemas de representación.
- La mantisa puede ser entera o fraccionaria; el exponente, obviamente, siempre son números enteros.
- La mantisa fraccionaria puede estar: 1) sin normalizar; 2) normalizada; y 3) normalizada y con bit implícito.

Veamos algunos ejemplos.

- *Mantisa entera de 5 bits en BSS [0, 31] y exponente en Ca2 de 5 bits [-16, 15].*

RANGO. Aquí no hay números negativos, así que el rango será fácil de calcular. El rango empieza desde el 0, y llega a la mantisa máxima con el exponente positivo máximo, o sea $31 \cdot 2^{15} = 1015808$.

RESOLUCIÓN. Antes, con el punto fijo, la resolución era constante a lo largo de la recta numérica. Ahora no: cerca del cero existe mejor resolución que lejos de él.

La resolución –se dijo en la p. 3– es la distancia entre dos números representables. ¿Y cómo sabemos qué número le sigue a otro en el punto flotante? ¿Lo obtenemos cambiando la mantisa o el exponente? Veamos. Tomemos un caso cualquiera que no esté ni al principio ni al final de esta recta: el 1024. Este puede escribirse como $01000\ 00111 = 8 \cdot 2^7 = 1024$. Si le sumo 1 a la mantisa obtenemos $01001\ 00111 = 9 \cdot 2^7 = 1152$. Si, en cambio, le sumo 1 al exponente tenemos $01000\ 01000 = 8 \cdot 2^8 = 2048$. La diferencia es notable.

Esto quiere decir que el número próximo se obtiene modificando la mantisa, no el exponente. Y se obtiene modificándola lo menos posible, es decir, lo que valga su bit menos significativo (en este caso, $2^0 = 1$, ya que estamos usando mantisa entera).

Ahora bien, cuando estamos en un sistema de punto flotante y nos piden la resolución, no interesa la resolución que anda por el medio, interesa la de los extremos: la que está más cerca del cero y la que está más lejos del cero. Eso significa "combinar" el bit menos significativo con el exponente más chico o el más grande, ya que los exponentes chicos nos dan mayor resolución y los exponentes grandes mayor alcance. Veamos qué nos da eso en este sistema en particular.

La resolución en el extremo inferior será el peso del bit menos significativo, elevado al exponente más pequeño. Esto nos da: $1 \cdot 2^{-16}$ o, simplemente, 2^{-16}.

La resolución en el extremo superior será el peso del bit menos significativo, elevado al exponente más grande. O sea: $1 \cdot 2^{15}$ o, simplemente, 2^{15}.

Tratemos ahora de representar algún número sencillo, como el 7,5. Para ello, primero lo escribimos en binario sin ponernos restricciones, de la forma más natural: con la coma en su lugar. De esa forma, nos queda: 111,1. Luego, como la mantisa es entera, debemos pasar todos los bits a la parte entera, así que nos queda: 1111. Y como agrandamos la mantisa, debemos achicar el exponente la cantidad de posiciones que movimos la coma, o sea, debemos restarle 1. Con todo esto, el número final nos queda $15 \cdot 2^{-1} = 7,5$, que en el sistema actual sería: 01111 11111.

- *Mantisa fraccionaria de 6 bits en BCS y exponente en Ca1 de 5 bits.*

Algo útil para no perderse, es calcular los rangos de la mantisa y del exponente. El del exponente es fácil: [-15, 15].

Calcular el rango de una mantisa fraccionaria podría hacerse sumando todos sus bits, al estilo de $2^{-1} + 2^{-2} + 2^{-3}$ etc. Pero hay un atajo: tomar el bit menos significativo de la mantisa, y se lo restamos a 1. Ese es el número más grande que podrá representar. En este caso, como el módulo de la mantisa son cinco bits, y el rango es simétrico entre los negativos y los positivos, nos queda: $[-(1 - 2^{-5}), (1 - 2^{-5})]$ o también [-0,96875, 0,96875].

Un ejemplo:

Mantisa fraccionaria en BCS							Exponente en Ca1					Resultado
Signo	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}							
1	1	1	1	1	1		0	1	1	1	1	$-(1-2^{-5})\cdot 2^{15}=$ $-(2^{15}-2^{10})$
$-(1-2^{-5})$							15					

Para el rango, debemos buscar la mantisa negativa máxima multiplicada por la base elevada al exponente positivo máximo y la mantisa positiva máxima multiplicada por la base elevada al exponente positivo máximo.

Rango $= [-(1-2^{-5})\cdot 2^{15}, (1-2^{-5})\cdot 2^{15}]$

$= [-(2^{15}-2^{10}), (2^{15}-2^{10})]$

$= [-31744, 31744]$

Para la resolución cerca del cero tomamos el valor del bit menos significativo de la mantisa y lo multiplicamos por la base elevada el exponente negativo máximo; para la resolución en los extremos, tomamos el valor del bit menos significativo de la mantisa y lo multiplicamos por la base elevada el exponente positivo máximo.

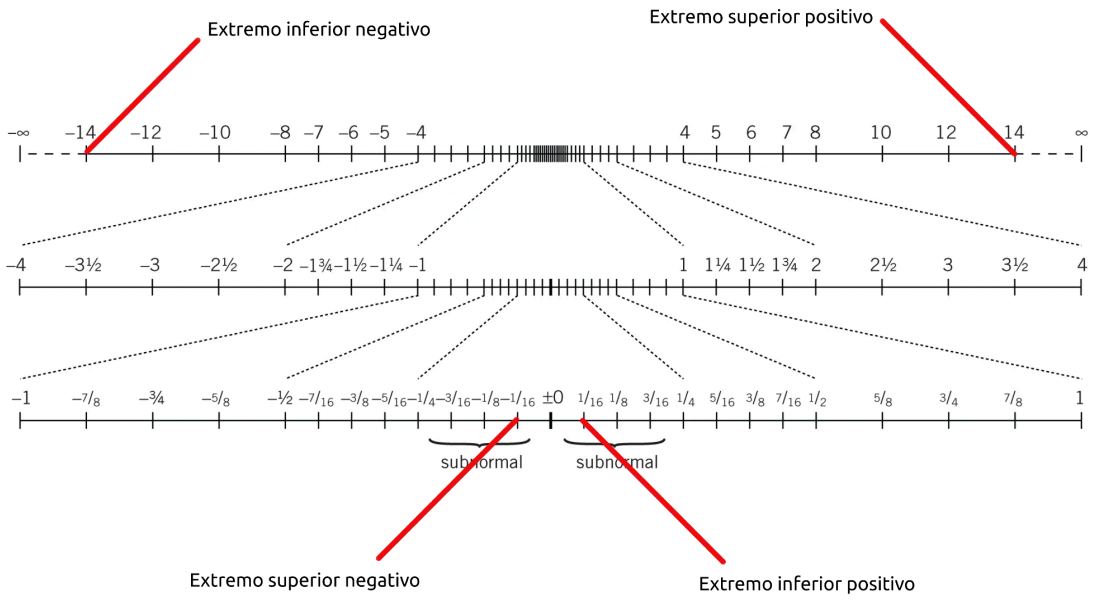
Resolución en el extremo inferior positivo y en el extremo superior negativo $= 2^{-5}\cdot 2^{-15}$

$= 2^{-20}$

Resolución en el extremo superior positivo y en el extremo inferior negativo $= 2^{-5}\cdot 2^{15}$

$= 2^{10} = 1024$

¿Por qué algunos extremos dan lo mismo? Porque, lógicamente, por lo dicho anteriormente, se calculan de la misma manera, ya que la recta es simétrica. Gráficamente (aunque con los valores de otro sistema), la recta numérica usando punto flotante se suele ver más o menos así:



- *Mantisa fraccionaria normalizada de 6 bits en BCS y exponente en Ca1 de 5 bits.*

Acá la cosa es parecida, pero no hay ninguna representación del cero, entonces para hacer bien el rango hay que partirlo en dos partes: primero el negativo y después el positivo.

$$\text{Rango} = [-\ (1 - 2^{-5}) \cdot 2^{15}\ , -\ (2^{-1}) \cdot 2^{-15}\ ; (2^{-1}) \cdot 2^{-15}\ , (1 - 2^{-5}) \cdot 2^{15}\ ]$$

$$= [-\ (2^{15} - 2^{-10}), -\ 2^{-16}\ ; 2^{-16}\ ,\ (2^{15} - 2^{10})\ ]$$

La resolución es la misma que antes.

- *Mantisa fraccionaria normalizada con bit implícito de 6 bits en BCS y exponente en Ca1 de 5 bits.*

Acá la cosa es parecida, pero “se agrega” un bit a la mantisa. En realidad, no se agrega nada, sino que se cambian los valores al momento de interpretar la cadena de bits. Pero para no confundirnos, podemos anotar el bit implícito, y hacer de cuenta como si estuviera ahí. En la siguiente tabla, es el bit que está en rojo. Es el único que siempre está en 1. Los demás bits pueden estar en 0 o en 1.

Mantisa fraccionaria en BCS								Exponente en Ca1				
Signo	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}						
x	1	x	x	x	x	x		x	x	x	x	x

Ahora se modifican tanto el rango como la resolución.

Rango = $[-(1-2^{-6})\cdot 2^{15}, -(2^{-1})\cdot 2^{-15}; (2^{-1})\cdot 2^{-15}, (1-2^{-6})\cdot 2^{15}]$

= $[-(2^{15}-2^{9}), -2^{-16}; 2^{-16}, (2^{15}-2^{9})]$

Resolución en el extremo inferior positivo y en el extremo superior negativo = $2^{-6}\cdot 2^{-15}$

$= 2^{-21}$

Resolución en el extremo superior positivo y en el extremo inferior negativo = $2^{-6}\cdot 2^{15}$

$= 2^{9} = 512$

Veamos ahora cómo quedarían algunos números en este último sistema.

Primero el 9,625. Empezamos por pasarlo a binario, y después acomodar la coma y el exponente:

9,625 = 1001,101 = $1001,101\cdot 2^{0} = ,1001101\cdot 2^{4}$

Tenemos 6 bits para la mantisa, pero tiene bit implícito, así que sería como tener 7 en total. 1 para el signo y 6 para el módulo = 0**1**00110. Pero el implícito no se guarda, así que queda = 000110. (Nótese que perdimos uno al final).

El exponente queda 00100.

Entonces M y E = 000110 00100 = $(2^{-1}+2^{-4}+2^{-5})\cdot 2^{4} = 2^{3}+2^{0}+2^{-1} = 8+1+0,5 = 9,5$

No llegamos al número exacto, porque tuvimos que recortar un bit. El error absoluto (EA) es $|x'-x| = |9,5-9,625| = 0,125$.

Y si elegimos la cadena 000111 00100 el número nos queda $(2^{-1} + 2^{-4} + 2^{-5} + 2^{-6}) \cdot 2^{4}$ = $2^{3} + 2^{0} + 2^{-1} + 2^{-2} = 8 + 1 + 0,5 + 0,25 = 9,75$. Con lo cual, el error es el mismo.

Esto es así porque el bit que recortamos era un solo 1. Si hubieran sido dos unos o más, hubiera convenido pasar el último bit a 1 en vez de dejarlo en 0. Pero en casos como este, da lo mismo.

El error relativo sirve para darnos una idea de cuán mal está la situación. La cosa es que nos da un número relativamente grande si no podemos representar bien los números chicos, y relativamente pequeño si le erramos por poco con números grandes. O sea, no es lo mismo equivocarse por 0,5 cuando estamos trabajando con números como el 3 o el 10, que cuando estamos con números como el 32 mil millones. En el primer caso el error es grave, en el segundo es insignificante.

En este caso, el error relativo (ER) sería: $\frac{EA(x)}{x} = \frac{0,125}{9,625} = 0,01298701299$

Pasemos al -0,4. En binario es periódico, el 0,4 sería así $0,\overline{0110}$, como si la secuencia fraccionaria se repitiera infinitamente, tipo $0,0110011001100110\ldots$

Entonces $0,4 = 0,0110011001 \cdot 2^{0} = ,110011001 \cdot 2^{-1}$

M = 1 para el signo, 1 bit implícito que no se anota, y **1**<u>10011</u>001 = 110011

E = 11110

O sea que nos queda: 110011 11110 = $-(2^{-1} + 2^{-2} + 2^{-5} + 2^{-6}) \cdot 2^{-1} = -(2^{-2} + 2^{-3} + 2^{-6} + 2^{-7}) = -0,3984375$

El error absoluto es de $|-0,3984375 - (-0,4)| = 0,0015625$.

El relativo es = $\frac{0,0015625}{0,4} = 0,00390625$

7. Cuentas flotantes

Para sumar dos números en punto flotante, sus exponentes tienen que ser los mismos. ¿Por qué? Porque en realidad la mejor manera de sumar estos números es sacando factor común, y sumar solamente las mantisas. Y para poder hacer eso, los exponentes tienen que ser iguales.

Veamos un ejemplo en decimal: $2 \cdot 10^2 + 3 \cdot 10^3 \neq 5 \cdot 10^5 = 500000$

$2 \cdot 10^2 + 3 \cdot 10^3 = 200 + 3000 = 3200$

Pero la mejor manera de resolver esto es sacando factor común, así:

$2 \cdot 10^2 + 3 \cdot 10^3 = 2 \cdot 10^2 + 3 \cdot 10 \cdot 10^2 = 10^2(2 + 3 \cdot 10) = 10^2(2 + 30)$

$= 10^2(32) = 3200$

La misma idea vale para el punto flotante con los números binarios. Solamente hay que tener cuidado de no perder bits al modificar los exponentes y no salirse de rango.

8. Pongámonos de acuerdo: IEEE 754

Para no andar cada loco con su punto flotante, está el IEEE 754, que estandariza un sistema de punto flotante en 32 bits y otro en 64.

El de 32 bits usa 1 para el signo, 8 para exponente y los 23 restantes para la mantisa.

- El bit del signo es 0 para los positivos, y 1 para los negativos.
- El exponente se representa en Exceso con $2^{n-1} - 1$, o sea que viene a ser 127.
- Y la mantisa es fraccionaria y está normalizada con bit implícito, pero el bit implícito no está en la posición 2^{-1}, sino que está en 2^0. O sea que la mantisa se interpreta tipo 1,M… A menos que se trate del caso especial en el cual debe interpretarse como sin normalizar y sin bit implícito.

El de 64 es lo mismo, pero con 1 bit de signo, 11 bits de exponente y 52 de mantisa.

Acá están los casos especiales (y los comunes también), de la versión en precisión simple.

	$M = 0$	$M \neq 0$
$E = 0$	$\pm\ 0$	$\pm\ 0, M \cdot 2^{-126}$
$0 < E < 255$	$1, M \cdot 2^{E-127}$	
$E = 255$	$\pm\ \infty$	NaN = not a number

Interpretando un IEEE 754.

0 01111111 00000000000000000000000 = $+\ 1,0 \cdot 2^{127-127} = 1 \cdot 2^{0} = 1 \cdot 1 = 1$

0 11111111 00000000000000000000000 = $+\ \infty$

Representando en IEEE 754.

Tomemos uno fácil, el 5,5.

Primero lo pasamos a binario: $5,5 = 101,1 = 101,1 \cdot 2^{0} = 1,011 \cdot 2^{2}$.

Ya sabemos que el signo será 0, y que la mantisa será 01100000000000000000000

Falta el exponente. Para eso, tomamos el número que debe ir, le sumamos el exceso, y lo escribimos en BSS. Así que sería 2 + 127 = 129 = 128 + 1 = 10000001.

Por último, unimos todo: 0 10000001 01100000000000000000000.