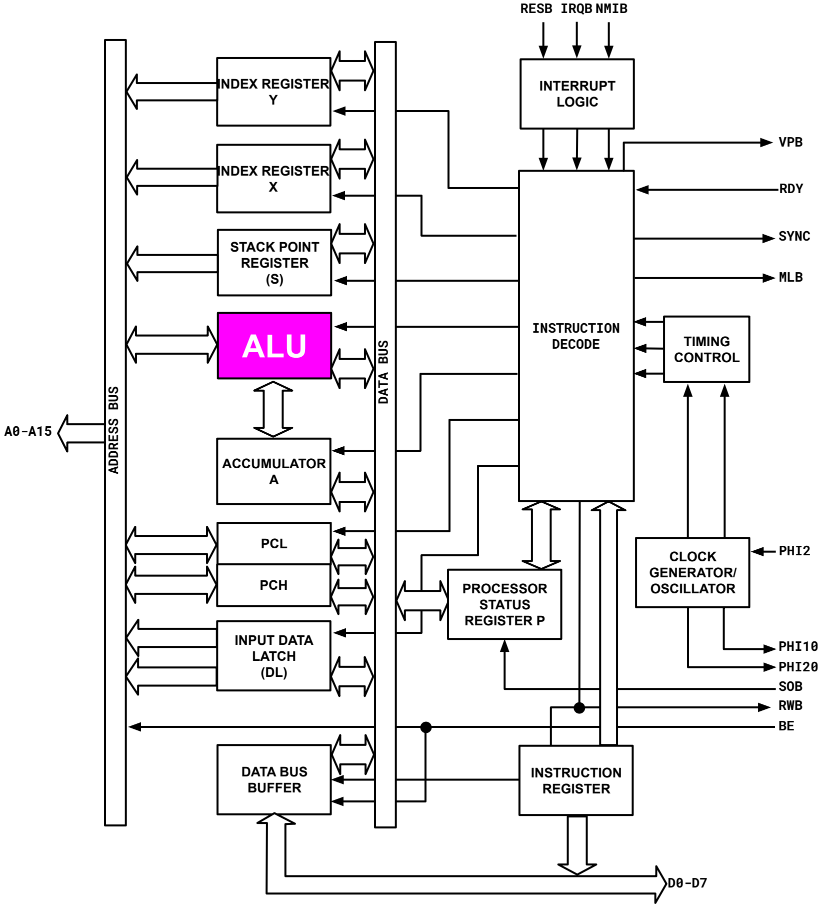
Circuitos Combinatorios

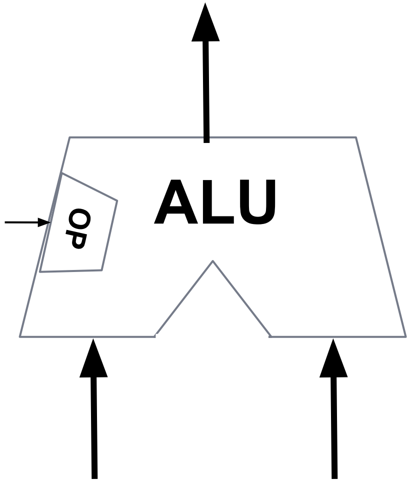
Unidad 2.1
Aritmética binaria

Versión 1.0.0

Carlos Maidana, Carlos Rodríguez, Edgardo Gho, Martín Ferreyra Biron, Jaír Hnatiuk

Una computadora...





Números sin signo (unsigned)

Un sistema de 4 bits

A	B	C	D	N
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Si usamos 4 bits para representar números, entonces existen 2^4 combinaciones posibles de estos bits.

Tenemos que darle un valor a cada combinación. Podríamos darle un valor arbitrario, por ejemplo el código ASCII, pero esto no nos sirve para hacer aritmética.

Vamos a buscar darle valores a las combinaciones que nos sirvan para hacer operaciones aritméticas.

Sumas

A	B	C	D	N
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**2**
0	0	1	1	**3**
0	1	0	0	**4**
0	1	0	1	**5**
0	1	1	0	**6**
0	1	1	1	**7**
1	0	0	0	**8**
1	0	0	1	**9**
1	0	1	0	**A**
1	0	1	1	**B**
1	1	0	0	**C**
1	1	0	1	**D**
1	1	1	0	**E**
1	1	1	1	**F**

Si utilizamos una representación binaria de enteros sin signo (unsigned int), solo representamos números positivos. Esta representación nos permite sumar y restar, pero si usamos 4 bits estamos limitados a resultados que se encuentren entre 0 y F (15). Vemos una suma que funciona…

	0	1	1		
	1	0	0	1	9
+	0	0	1	1	3
	1	1	0	0	C

Sumas

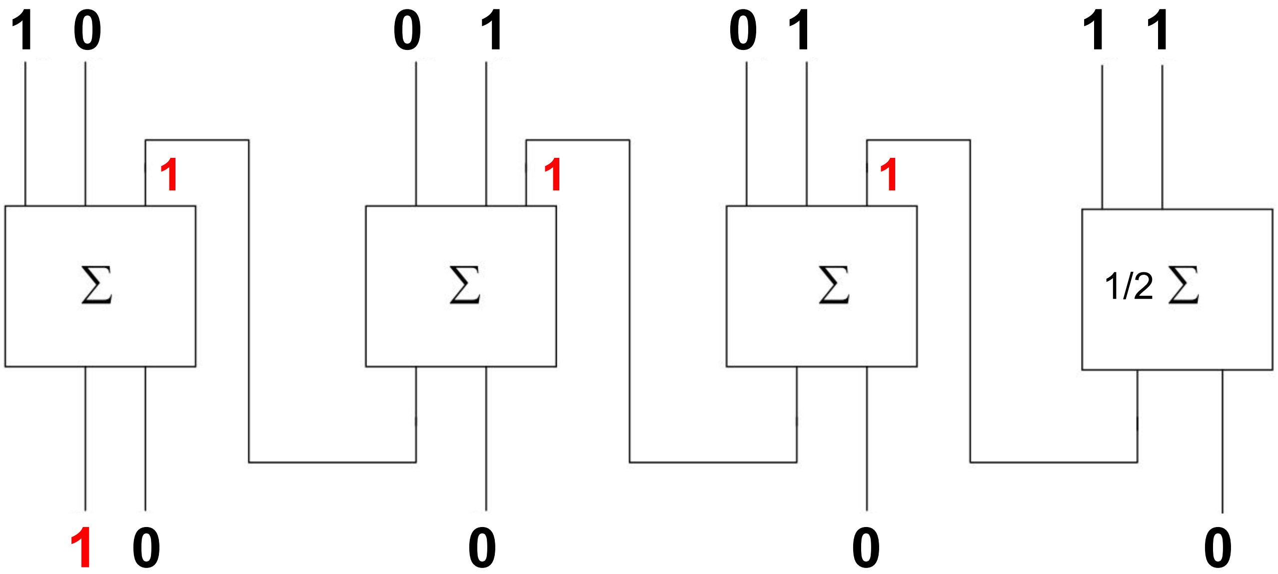
A	B	C	D	N
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**2**
0	0	1	1	**3**
0	1	0	0	**4**
0	1	0	1	**5**
0	1	1	0	**6**
0	1	1	1	**7**
1	0	0	0	**8**
1	0	0	1	**9**
1	0	1	0	**A**
1	0	1	1	**B**
1	1	0	0	**C**
1	1	0	1	**D**
1	1	1	0	**E**
1	1	1	1	**F**

Pero vemos que el sistema no funciona cuando el resultado queda afuera de los 4 bits. Si sumamos 9+7=16 … No existe el 16 en este sistema. El resultado en 4 bits es 0000 (0). Pero aparece un bit extra de acarreo (carry). Este bit indica que el resultado NO se puede representar en 4 bits.

	1	1	1	1		
		1	0	0	1	9
	+	0	1	1	1	7
Carry	1	0	0	0	0	0

Sumas

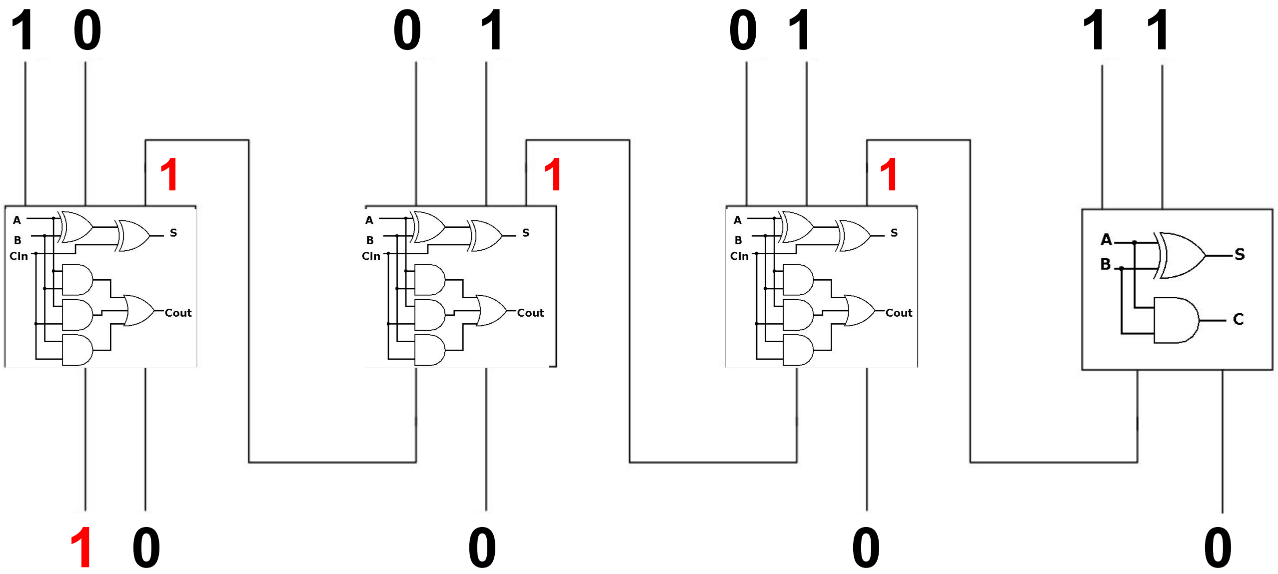
A	B	C	D	N
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**2**
0	0	1	1	**3**
0	1	0	0	**4**
0	1	0	1	**5**
0	1	1	0	**6**
0	1	1	1	**7**
1	0	0	0	**8**
1	0	0	1	**9**
1	0	1	0	**A**
1	0	1	1	**B**
1	1	0	0	**C**
1	1	0	1	**D**
1	1	1	0	**E**
1	1	1	1	**F**



	1	1	1	1		
		1	0	0	1	9
	+	0	1	1	1	7
Carry	1	0	0	0	0	0

Sumas

A	B	C	D	N
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**2**
0	0	1	1	**3**
0	1	0	0	**4**
0	1	0	1	**5**
0	1	1	0	**6**
0	1	1	1	**7**
1	0	0	0	**8**
1	0	0	1	**9**
1	0	1	0	**A**
1	0	1	1	**B**
1	1	0	0	**C**
1	1	0	1	**D**
1	1	1	0	**E**
1	1	1	1	**F**



1	1	1	1		
	1	0	0	1	9
+	0	1	1	1	7
Carry 1	0	0	0	0	0

A	B	C	D	N
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**2**
0	0	1	1	**3**
0	1	0	0	**4**
0	1	0	1	**5**
0	1	1	0	**6**
0	1	1	1	**7**
1	0	0	0	**8**
1	0	0	1	**9**
1	0	1	0	**A**
1	0	1	1	**B**
1	1	0	0	**C**
1	1	0	1	**D**
1	1	1	0	**E**
1	1	1	1	**F**

Restas

Si hacemos una resta cuyo resultado se puede representar entre 0 y 15, el sistema también funciona.
En una resta de A-B=C , A es el minuendo, B es el sustraendo y C es el resultado.
Veamos un ejemplo donde funciona…
Comenzamos con 1-1 = 0… pero luego vemos que tenemos 0-1 .. y en este caso el sustraendo es menor que el minuendo.. Necesitamos “pedir prestado” a la columna de mayor peso…

	1	0	0	1	9
-	0	0	1	1	3
				0	

Restas

A	B	C	D	N
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**2**
0	0	1	1	**3**
0	1	0	0	**4**
0	1	0	1	**5**
0	1	1	0	**6**
0	1	1	1	**7**
1	0	0	0	**8**
1	0	0	1	**9**
1	0	1	0	**A**
1	0	1	1	**B**
1	1	0	0	**C**
1	1	0	1	**D**
1	1	1	0	**E**
1	1	1	1	**F**

Este “prestamo” (borrow) viene del bit de la izquierda. Lo representamos con un 1 en la fila de borrow indicando que hay un préstamo hecho de esta columna a la anterior. Pero este préstamo tiene “más peso”, ya que viene de una columna con mayor peso. Por ende, cuando llega a la columna actual, vale 2 (la base). Por ende 0+2=2, en binario 10.... Y ahora podemos hacer la resta de $10_2 - 1_2 = 1_2$

		1			
	1	0	10	1	9
-	0	0	1	1	3
			1	0	

Restas

A	B	C	D	N
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**2**
0	0	1	1	**3**
0	1	0	0	**4**
0	1	0	1	**5**
0	1	1	0	**6**
0	1	1	1	**7**
1	0	0	0	**8**
1	0	0	1	**9**
1	0	1	0	**A**
1	0	1	1	**B**
1	1	0	0	**C**
1	1	0	1	**D**
1	1	1	0	**E**
1	1	1	1	**F**

Ahora en la tercer columna tenemos un préstamo dado (borrow), pero el minuendo es cero, así que le vamos a pedir prestado a la columna que sigue…

	1	1			
	1	0	10	1	9
-	0	0	1	1	3
			1	0	

Restas

A	B	C	D	N
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**2**
0	0	1	1	**3**
0	1	0	0	**4**
0	1	0	1	**5**
0	1	1	0	**6**
0	1	1	1	**7**
1	0	0	0	**8**
1	0	0	1	**9**
1	0	1	0	**A**
1	0	1	1	**B**
1	1	0	0	**C**
1	1	0	1	**D**
1	1	1	0	**E**
1	1	1	1	**F**

Ahora en la tercer columna tenemos un préstamo dado (borrow), pero el minuendo es cero, así que le vamos a pedir prestado a la columna que sigue… Pero ese borrow viene de una columna con más peso, por ende vale 2. Si a mi me prestan dos, pero yo le preste uno a la columna anterior, entonces me queda uno. Luego 1-0=1.

	1	1			
	1	1	10	1	9
-	0	0	1	1	3
		1	1	0	

Restas

A	B	C	D	N
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**2**
0	0	1	1	**3**
0	1	0	0	**4**
0	1	0	1	**5**
0	1	1	0	**6**
0	1	1	1	**7**
1	0	0	0	**8**
1	0	0	1	**9**
1	0	1	0	**A**
1	0	1	1	**B**
1	1	0	0	**C**
1	1	0	1	**D**
1	1	1	0	**E**
1	1	1	1	**F**

En la cuarta columna, tenemos un 1 en el minuendo, pero hicimos un prestamo.. Así que ese uno ahora va a pasar a ser un cero ya que fue prestado.

$$\begin{array}{cccccc|c}
 & 1 & 1 & & & & \\
 & \cancel{1} & 1 & 10 & 1 & & 9 \\
- & 0 & 0 & 1 & 1 & & 3 \\
\hline
 & & 1 & 1 & 0 & &
\end{array}$$

Restas

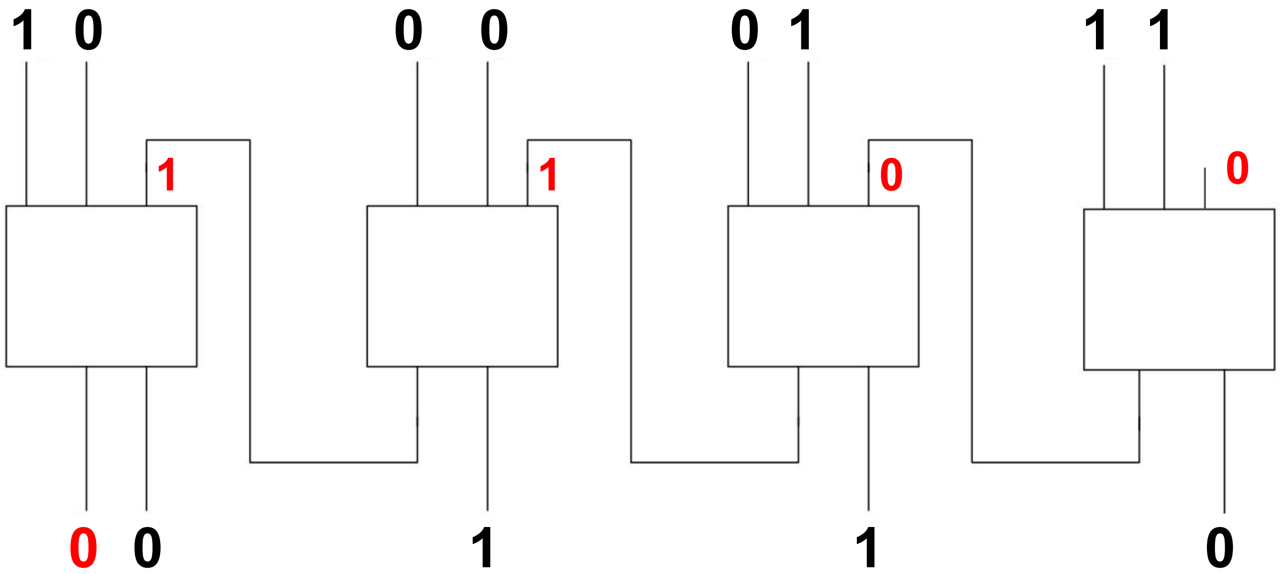
A	B	C	D	N
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**2**
0	0	1	1	**3**
0	1	0	0	**4**
0	1	0	1	**5**
0	1	1	0	**6**
0	1	1	1	**7**
1	0	0	0	**8**
1	0	0	1	**9**
1	0	1	0	**A**
1	0	1	1	**B**
1	1	0	0	**C**
1	1	0	1	**D**
1	1	1	0	**E**
1	1	1	1	**F**

En la cuarta columna, tenemos un 1 en el minuendo, pero hicimos un prestamo.. Así que ese uno ahora va a pasar a ser un cero ya que fue prestado.
Así que 0-0=0… y terminamos la resta… obtenemos como resultado 6, que es efectivamente 9-3.

	1	1			
	0	1	10	1	9
-	0	0	1	1	3
	0	1	1	0	6

entrada			salida	
A	B	B_{IN}	B_{OUT}	R
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Restas



	1	1			
	1	0	0	1	9
-	0	0	1	1	3
	0	1	1	0	6

Restas

A	B	C	D	N
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**2**
0	0	1	1	**3**
0	1	0	0	**4**
0	1	0	1	**5**
0	1	1	0	**6**
0	1	1	1	**7**
1	0	0	0	**8**
1	0	0	1	**9**
1	0	1	0	**A**
1	0	1	1	**B**
1	1	0	0	**C**
1	1	0	1	**D**
1	1	1	0	**E**
1	1	1	1	**F**

Al igual que en la suma.. Si el resultado queda fuera del sistema de representación, entonces el resultado no puede ser representado. Veamos 1-3 .. en la primer columna no tenemos problema, pero en la segunda tenemos 0-1.. Sabemos que tenemos que pedir un préstamo….

$$
\begin{array}{cccccc}
 & 0 & 0 & 0 & 1 & 1 \\
- & 0 & 0 & 1 & 1 & 3 \\
\hline
 & & & & 0 &
\end{array}
$$

Restas

A	B	C	D	N
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**2**
0	0	1	1	**3**
0	1	0	0	**4**
0	1	0	1	**5**
0	1	1	0	**6**
0	1	1	1	**7**
1	0	0	0	**8**
1	0	0	1	**9**
1	0	1	0	**A**
1	0	1	1	**B**
1	1	0	0	**C**
1	1	0	1	**D**
1	1	1	0	**E**
1	1	1	1	**F**

Por ende la tercer columna nos presta, y obtenemos 2-1=1.. Pero ahora en la tercer columna hicimos un préstamo pero no tenemos solvencia.. Así que tenemos que volver a pedir…

		1			
	0	0	10	1	1
-	0	0	1	1	3
			1	0	

Restas

A	B	C	D	N
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**2**
0	0	1	1	**3**
0	1	0	0	**4**
0	1	0	1	**5**
0	1	1	0	**6**
0	1	1	1	**7**
1	0	0	0	**8**
1	0	0	1	**9**
1	0	1	0	**A**
1	0	1	1	**B**
1	1	0	0	**C**
1	1	0	1	**D**
1	1	1	0	**E**
1	1	1	1	**F**

Ahora que pedimos, tenemos 2 (10) pero teníamos una deuda de uno, por ende nos queda solo uno. A este uno le restamos cero y nos queda uno. Pero ahora estamos en el mismo problema en la cuarta columna… PERO no tenemos más a quien pedir! Así que no queda otra que declarar default y marcar eso con un bit extra…

	1	1			
	0	1	10	1	1
-	0	0	1	1	3
		1	1	0	

Restas

A	B	C	D	N
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**2**
0	0	1	1	**3**
0	1	0	0	**4**
0	1	0	1	**5**
0	1	1	0	**6**
0	1	1	1	**7**
1	0	0	0	**8**
1	0	0	1	**9**
1	0	1	0	**A**
1	0	1	1	**B**
1	1	0	0	**C**
1	1	0	1	**D**
1	1	1	0	**E**
1	1	1	1	**F**

Aparece ahora un quinto bit en el resultado… lo llamamos bit de Borrow. Este bit en uno indica que el resultado no puede ser representado en 4 bits sin signo. Este bit es muy útil. Imaginemos que tenemos dos números A y B, y tenemos que saber si A<B.. entonces si hacemos A-B y el resultado tiene Borrow=1, entonces seguro A<B (solo en representación sin signo). Si Borrow=0 entonces A>=B.

	1	1	1			
		1	1	10	1	1
	-	0	0	1	1	3
Borrow	1	1	1	1	0	E

Números signados

Modulo y Signo :(

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-0
1	0	0	1	-1
1	0	1	0	-2
1	0	1	1	-3
1	1	0	0	-4
1	1	0	1	-5
1	1	1	0	-6
1	1	1	1	-7

Una representación <u>totalmente inutil</u> es la de módulo y signo. Utilizamos el bit más significativo para indicar el signo del número (0=positivo, 1=negativo) y el resto de los bits componen el módulo del número. Vemos que si hacemos la suma de 3 + (-1) obtenemos como resultado -4.. Cuando deberíamos tener como resultado 2 (0010).

Esta representación no permite hacer operaciones aritméticas en binario. Tenemos que utilizar la regla de signos... mucho trabajo!

$$
\begin{array}{rcccc|c}
 & & 1 & & 1 & \\
 & 0 & 0 & 1 & 1 & 3 \\
+ & 1 & 0 & 0 & 1 & -1 \\
\hline
 & 1 & 1 & 0 & 0 & -4
\end{array}
$$

Complemento a la base

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

La representación en complemento a la base permite hacer operaciones aritméticas. Siempre y cuando el resultado pueda ser representado en 4 bits signados con negativos en C.B, la aritmética funciona. Notamos que en 4 bits, 3+(-1)=2. Aparece el bit de carry nuevamente, pero en números signados lo ignoramos… el resultado es válido.

1	1	1	1		
	0	0	1	1	3
+	1	1	1	1	-1
1	0	0	1	0	2

Complemento a la base

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

También funciona la resta… si hacemos -5 - (-2)... Vemos que las primeras dos columnas no hay mayores problemas, pero en la tercera hacemos un borrow…

	1	0	1	1	-5
-	1	1	1	0	-2
			0	1	

Complemento a la base

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Ahora en la cuarta columna el minuendo tiene uno pero fue prestado, así que tiene que volver a pedir…

$$
\begin{array}{cccccc|c}
 & \color{red}{1} & & & & & \\
 & 1 & 10 & 1 & 1 & & -5 \\
- & 1 & 1 & 1 & 0 & & -2 \\
\hline
 & & 1 & 0 & 1 & & -3
\end{array}
$$

Complemento a la base

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Terminamos, y obtenemos -3, lo cual es correcto. Aparece el bit de borrow, pero nuevamente, en representación signada con negativos en C.B este bit lo ignoramos ya que el resultado es correcto.
Todo parece excelente con esta representación..
Pero.. Puede ocurrir que el resultado quede fuera del rango de representación.

1	1				
	10	10	1	1	-5
-	1	1	1	0	-2
1	1	1	0	1	-3

Suma fuera de rango

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Si sumamos 6+3 el resultado es 9, pero como vemos en la tabla, no existe el 9. Al hacer la suma obtenemos -7. Esto es inválido. Vemos que sumamos dos números positivos y obtenemos como resultado un número negativo! A esta condición la llamamos desborde (oVerflow).

	1	1			
	0	1	1	0	6
+	0	0	1	1	3
	1	0	0	1	-7

Suma fuera de rango

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Para detectar el oVerflow en la suma, vemos que los bits de signo de los operando son iguales (ambos 0 indicando positivos), pero el resultado es distinto (negativo). En la suma esta condición es oVerflow. Si se produce oVerflow el resultado de la suma no puede ser representado con este sistema (vamos a necesitar más bits).

$$\begin{array}{cccccc|c}
 & 1 & 1 & & & & \\
 & 0 & 1 & 1 & 0 & & 6 \\
+ & 0 & 0 & 1 & 1 & & 3 \\
\hline
 & 1 & 0 & 0 & 1 & & -7
\end{array}$$

Suma fuera de rango

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

También se puede producir overflow sumando números negativos.. Vemos que -7 + (-2) debería ser -9 pero no tenemos el -9 en este sistema. Eso se representa con el oVerflow en donde ambos operandos tienen el mismo signo (negativo) y el resultado el signo opuesto (positivo). El bit de carry nuevamente lo ignoramos en números signados.

$$
\begin{array}{cccccc|c}
 & 1 & & & & & \\
 & & 1 & 0 & 0 & 1 & -7 \\
 & + & 1 & 1 & 1 & 0 & -2 \\
\hline
 & 1 & 0 & 1 & 1 & 1 & 7
\end{array}
$$

Resta fuera de rango

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Podemos restar dos números y que el resultado también quede fuera del rango. Si restamos 6 - (-3) deberíamos obtener 9, pero el resultado desborda y obtenemos un número negativo (-7). En la **resta**, el oVerflow se produce cuando el bit de signo del minuendo es distinto del sustraendo, y el bit de signo del resultado es igual al sustraendo.

$$
\begin{array}{cccccc|c}
 & & 0 & 1 & 1 & 0 & 6 \\
- & & 1 & 1 & 0 & 1 & -3 \\
\hline
 & 1 & 1 & 0 & 0 & 1 & -7
\end{array}
$$

Resta fuera de rango

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

También podemos contar con el caso de restarle a un negativo un número positivo y desbordar. Si hacemos -7-3 deberíamos tener -10, pero eso no existe en este sistema, así que obtenemos el 6. En este caso el bit de signo del minuendo es distinto de sustraendo y el resultado tiene el mismo signo que el sustraendo.. Por ende oVerflow!

$$
\begin{array}{ccccc|c}
 & 1 & 0 & 0 & 1 & -7 \\
- & 0 & 0 & 1 & 1 & 3 \\
\hline
 & 0 & 1 & 1 & 0 & 6
\end{array}
$$

Comparando números signados

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

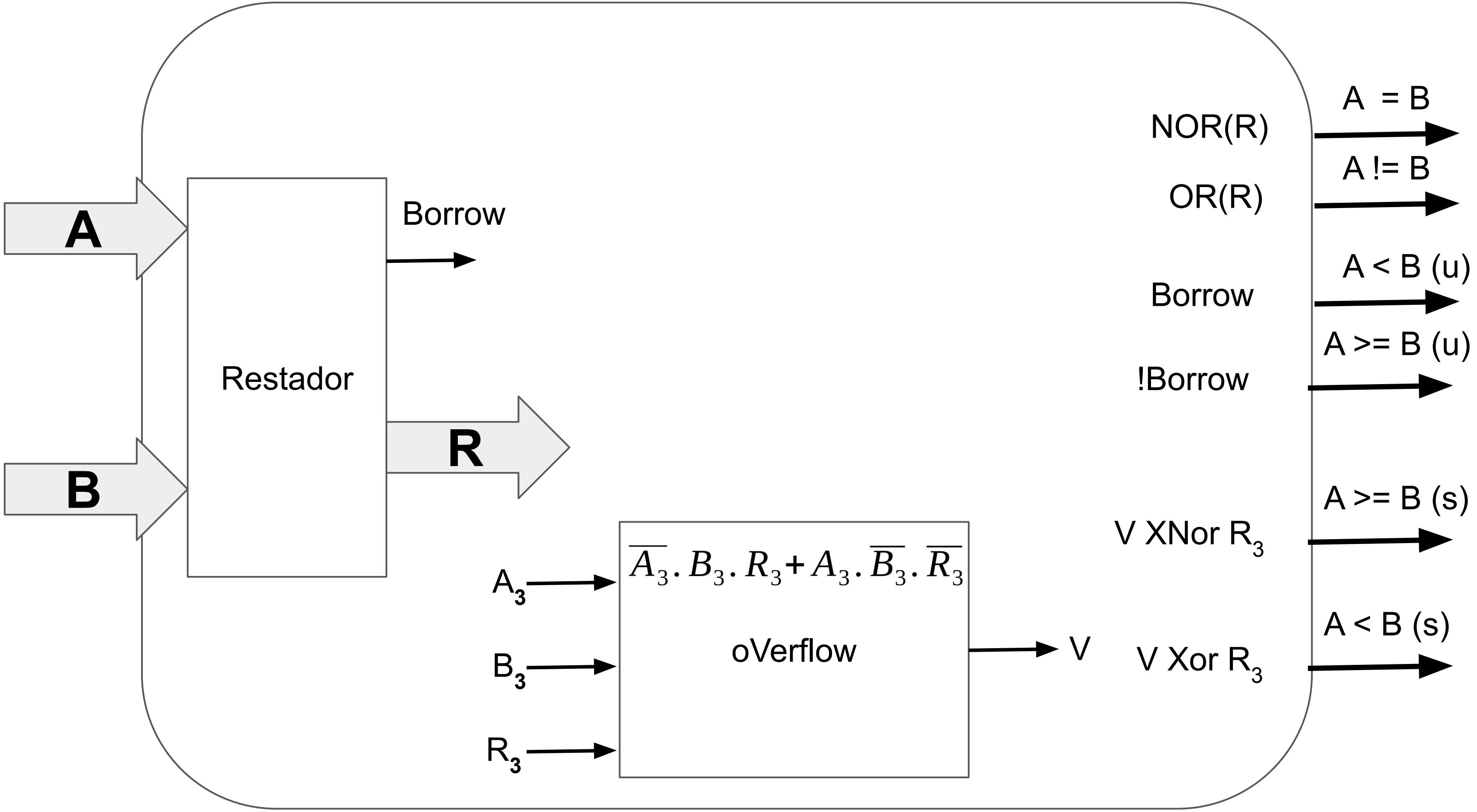
Podemos utilizar la resta como operación para comparar. Si tenemos un número A (A_3 A_2 A_1 A_0) y un número B (B_3 B_2 B_1 B_0) y **realizamos A-B** , vamos a obtener un resultado R (R_3 R_2 R_1 R_0). **Si R=0, entonces A=B** (siempre). Obviamente si **R!=0** entonces **A!=B**. Luego oVerflow se calcula como…

$$\overline{A_3}.B_3.R_3+A_3.\overline{B_3}.\overline{R_3}=V$$

Y teniendo V… si:

- **V == R_3 entonces A>=B**
- **V != R_3 entonces A < B**

Comparador de dos números signados (s) o no (u)

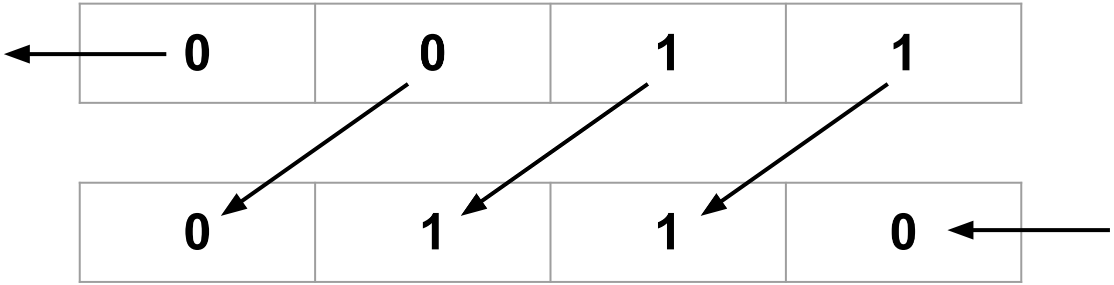


Desplazamientos

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Desplazamiento a izquierda

Si tenemos un número entero y lo “desplazamos” a la izquierda en un bit, agregando un cero en su bit más significativo, estamos efectivamente “corriendo” la coma a la derecha…multiplicando por la base (en este caso 2).

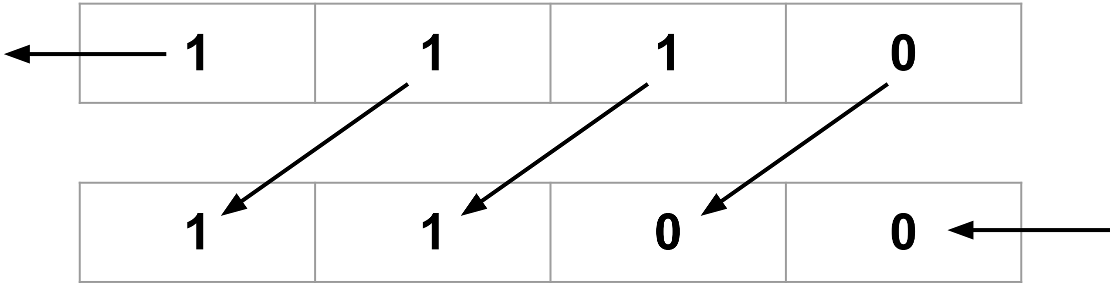


El número original (0011) 3 se desplaza un bit a la izquierda y queda (0110)6, que efectivamente es equivalente a multiplicar 3x2.

Desplazamiento a izquierda

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

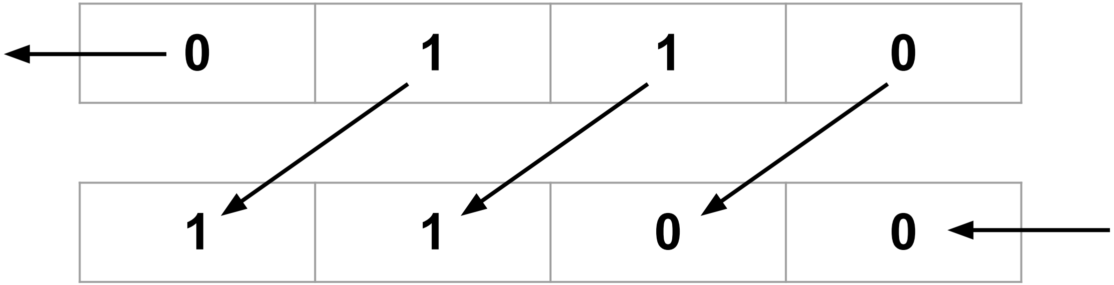
Si el número es negativo, también funciona… vemos que -2 (1110) corrido a la izquierda se convierte en el -4 (1100).



A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Desplazamiento a izquierda

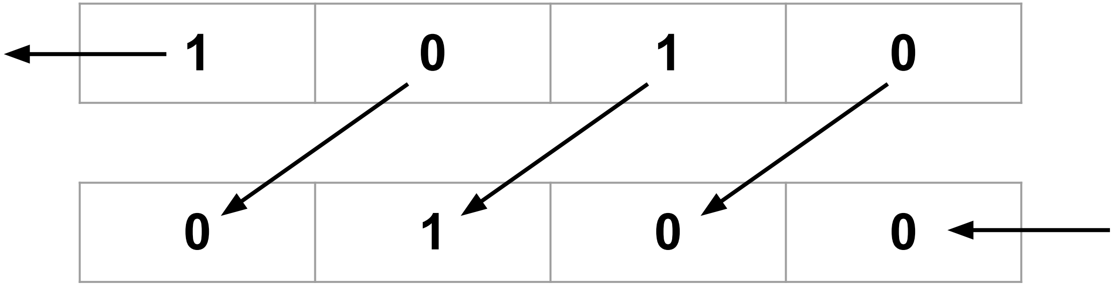
Si ocurre que el resultado se pasa del sistema de representación, entonces no sirve…vemos que 6x2 debería ser 12, pero obtenemos el -3 ya que el 12 no puede ser representado.



A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Desplazamiento a izquierda

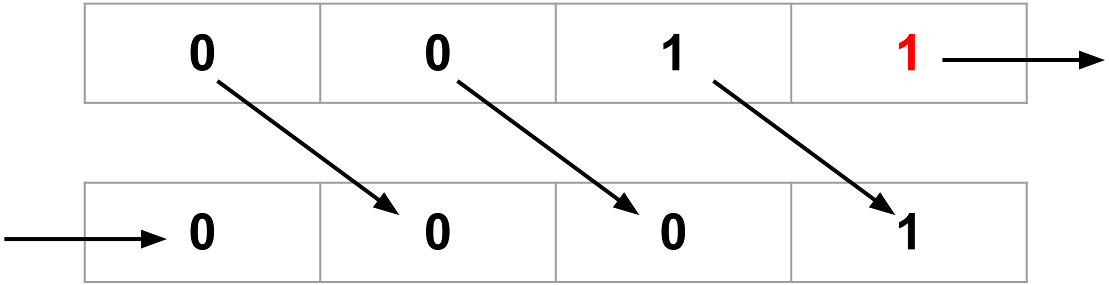
Lo mismo si el número es negativo… si desplazamos el -6 a la izquierda deberíamos obtener el -12, pero obtenemos 4.



Desplazamiento a derecha

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Si desplazamos a la derecha, es equivalente a correr la coma a la izquierda. Es decir, estamos dividiendo por la base, en esta caso 2. En este ejemplo tenemos el 3 (0011) y lo desplazamos uno a derecha, nos queda 1 (0001). Vemos que 3/2=1,5.. Parte entera 1.

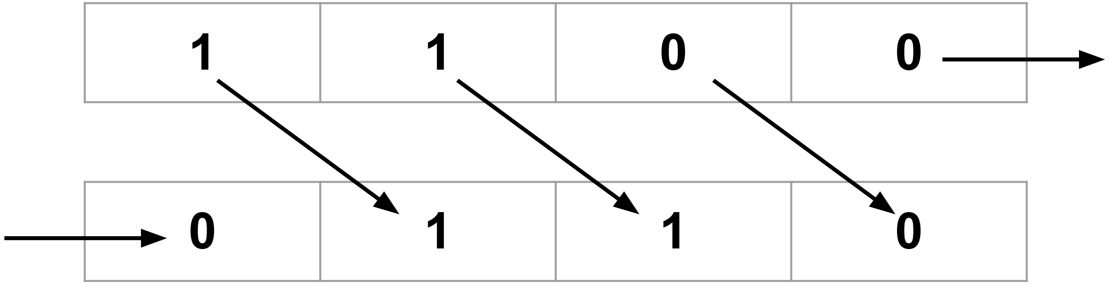


El bit menos significativo original (1) pasa a ser la parte fraccionaria. En binario queda como 0,1 y esa posición se encuentra por 2 a la menos 1, o sea un medio, o 0,5. Al trabajar en enteros se redondea hacia el cero.

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Desplazamiento a derecha (lógico)

En el caso de los negativos hay un problema.. Si tomamos -4 (1100) y lo desplazamos obtenemos el número 6 (0110). El resultado debería haber sido el número -2 (1110).

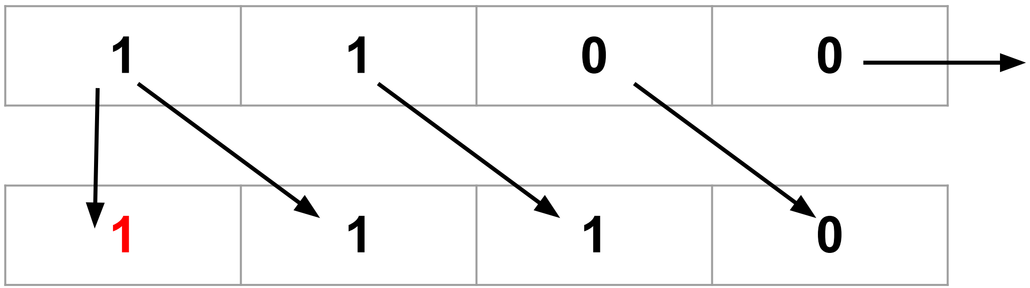


Esto se debe a haber completado con cero el bit más significativo. A este desplazamiento se lo conoce como desplazamiento a derecha **lógico.**

Desplazamiento a derecha (aritmético)

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Si repetimos el bit más significativo original en la posición del bit más significativo del nuevo número, entonces obtenemos el -2 (1110) esperado.

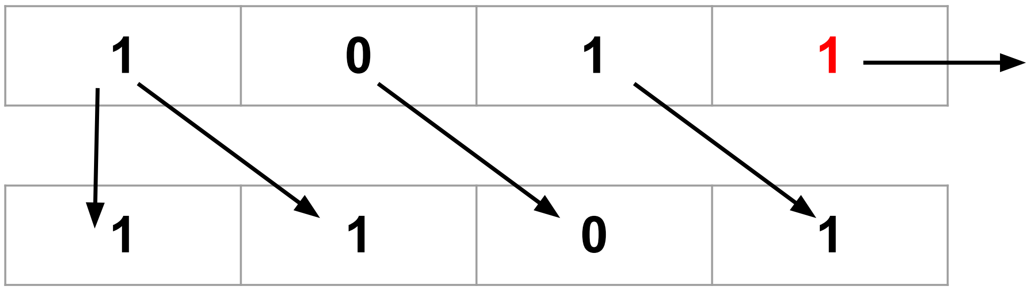


A este desplazamiento donde mantenemos el bit de signo original lo conocemos como desplazamiento aritmético.

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Desplazamiento a derecha (aritmético)

En el caso del redondeo con números negativos, el redondeo se hace hacia el menos infinito. Vemos que -5 sobre 2 deberia dar -2,5.. Siendo -2 parte entera pero el sistema redondea hacia la izquierda, entonces obtenemos -3.



En este caso podemos salvar el problema. Si verificamos que el bit menos significativo original era 1 y el signo es negativo, debemos sumar uno al resultado.. Siendo 1101 + 0001 = 1110 (-2).

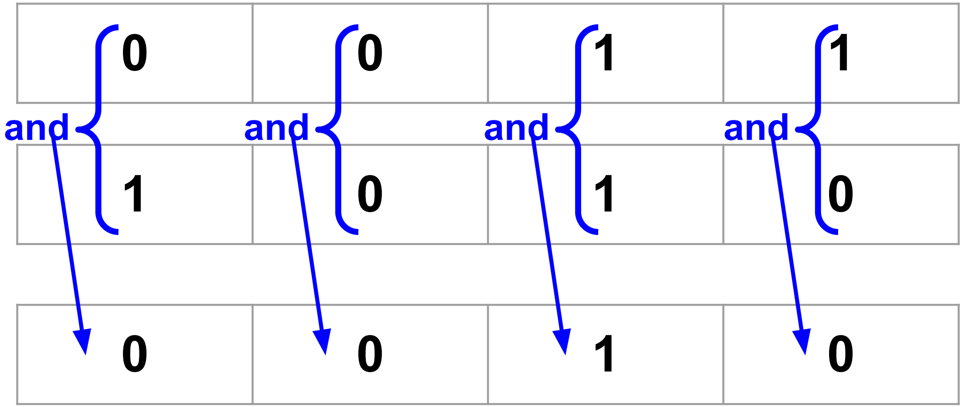
Circuitos lógicos

Operación AND bit a bit (bitwise)

A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Si tenemos dos números, A y B, cada uno de 4 bits, definimos la operación AND entre esos dos números como el resultado de aplicar la compuerta AND bit a bit entre cada bit que compone el número.

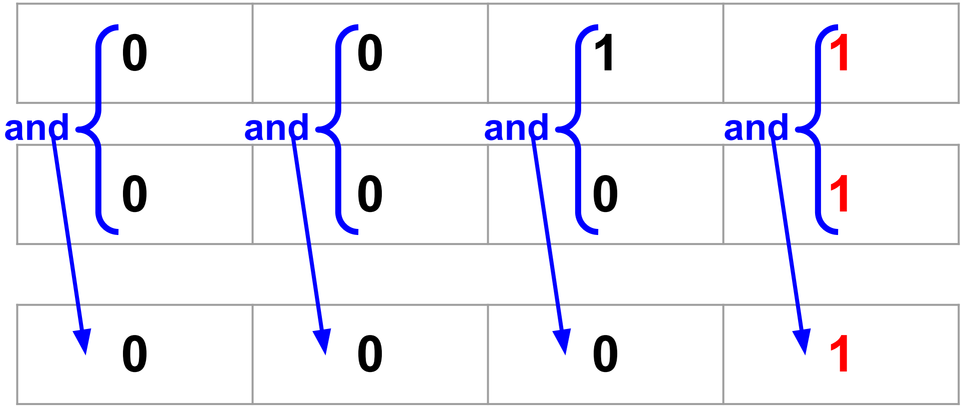
Ej: Si A=0011 y B=1010 , entonces hacemos



Operación AND bit a bit (bitwise)

A	B	C	D	N
0	0	0	0	0
0	**0**	**0**	**1**	**1**
0	0	1	0	2
0	**0**	**1**	**1**	**3**
0	1	0	0	4
0	**1**	**0**	**1**	**5**
0	1	1	0	6
0	**1**	**1**	**1**	**7**
1	0	0	0	-8
1	**0**	**0**	**1**	**-7**
1	0	1	0	-6
1	**0**	**1**	**1**	**-5**
1	1	0	0	-4
1	**1**	**0**	**1**	**-3**
1	1	1	0	-2
1	**1**	**1**	**1**	**-1**

Notemos que los números impares tienen un bit menos significativo igual a uno. Si hacemos cualquier número AND 0001, y el resultado es 1, el número original es impar. Veamos por ejemplo 0011AND0001 = 0001, por ende 0011 es impar.



A	B	C	D	N
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Operación OR bit a bit (bitwise)

Si tenemos dos números, A y B, cada uno de 4 bits, definimos la operación OR entre esos dos números como el resultado de aplicar la compuerta OR bit a bit entre cada bit que compone el número.

Ej: Si A=0011 y B=1010 , entonces hacemos

0	0	1	1
or	or	or	or
1	0	1	0
1	0	1	1