Representando información.

Queremos representar una **magnitud** a través de un **sistema de representación**:
Finito soporte fijo, cantidad de elementos acotados
Composicional diversas magnitudes podrían representarse con un conjunto de elementos atómicos que deben ser fáciles de implementar y componer
Posicional la posición de cada dígito determina unívocamente en qué proporción modifica su valor a la magnitud total del número
El soporte formal lo encontraremos en las **bases de representación numérica**.

Bases

En términos prácticos una base determina la **cantidad de símbolos distintos que podemos encontrar en un dígito dado** dentro de nuestra representación.
Una misma magnitud puede tener distintas representaciones en distintas bases. Por ejemplo la magnitud asociada al cuatro puede representarse como:

Base	Valor	Notación
2	100	$(100)_{(2)}$
3	11	$(11)_{(3)}$
10	4	$(4)_{(10)}$

- En base 2, usamos los símbolos 0 y 1 y escribimos los naturales: 0, 1, 10, 11, 100, 101, 110...
- En base 3, usamos los símbolos 0, 1 y 2 y escribimos los naturales: 0, 1, 2, 10, 11, 12, 20... ˆ ...y así...

Bases más comunes

Base	Símbolos usados
2 (binario)	0, 1
8 (octal)	0, 1, 2, 3, 4, 5, 6, 7
10 (decimal)	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
16 (hexadecimal)	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Cambios de bases

Recordemos que un **cambio de base** es una operación que transforma un n´umero n=[1,1,0,1](1101(2)) representado como lista de símbolos para una base dada, por ejemplo 2 (binario) y lo representa en otra base, por ejemplo 10 (decimal) [1, 3](13(10)).

$1101_{(2)} \rightarrow 13_{(10)}$

Podemos pensar que el cambio de base es una traducción entre dos formas de representar una misma magnitud.

Cambios de bases: Teorema de la división.

¿Para qué queremos el cambio de base?
Para convertir una magnitud de una representación que nos resulta natural (base 10) a la forma en que representan y almacenan los datos en la computadora (base 2).
La **división euclídea**, llamada también **teorema o algoritmo de la división** va a vincular una magnitud a y una base b diciendo que hay un único cociente y resto que permiten escribir la magnitud a en base al valor b.
Nos valemos de esto para aplicar operaciones sucesivas que descompongan a en base a valores b^n , b^n−1 , . . . , b^1 , sabiendo que estos b^i se relacionan con el símbolo ˆai que va en la posición i cuando queremos hacer el cambio de base: a → ˆa.

Teorema:
Sean a, b ∈ Z con b ≠ 0.
Existen q,r ∈ Z con 0 ≤ r < |b| tales que a = b × q + r
Además, q y r son únicos (de a pares).
¿Cómo lo usamos ?
a = b × q + r
a = (b × q1 + r1) × b + r
a = [(b × q2 + r2) × b + r1] × b + r
Podemos continuar con la expansión hasta que qN < b
a = {[(b × qN + rN) × b) + rN−1] × ...} × b + r × 1
Si distribuimos va a quedar: a = qN × b N+1 + rN × b N + ... + r1 × b + r × b 0
Representación posicional:
Podemos ver que el primer elemento de cada producto es el que va a aparecer en los dígitos de nuestra representación posicional, donde:
a = qN × b^N+1 + rN × b^N + ... + r1 × b + r × b 0
Se puede escribir como:

q_N	r_N	. . .	r_1	r

O de forma correcta, incluyendo la base:
(qNrN . . .r1r)(b)
Ejemplo:
27 = 2 · 10^1 + 7 · 10^0 = 1 · 2^4 + 1 · 2^3 + 0 · 2^2 + 1 · 2^1 + 1 · 2^0
27 = (27)(10) = (11011)(2)
Representación finita Cuando pensamos en los números y cuando los escribimos en la vida diaria, no contamos con una restricción evidente en la cantidad de dígitos con los que podemos operar. Pero en un soporte electrónico, como resulta ser el caso de la computadora, cada dato se representa con una cantidad finita de elementos. En el caso de los números podemos pensar que **lo que está acotado es la cantidad de dígitos que podemos emplear. Cada dígito va a poder tener tantos valores distintos como tenga la base.** En base 10 son 10 valores distintos, del 0 al 9, en base 2 son dos que van del 0 al 1.

Rango de representación El rango de representación viene asociado al tipo de dato, hasta ahora vimos números naturales (N_0), y a la cantidad de dígitos que podemos escribir. Por ahora la forma de computar el rango es aplicando el cálculo de combinaciones cruzadas, por ejemplo una tira de 4 dígitos:

$(a3a2a1a0)_{(b)}$

Donde cada dígito puede tener valores entre 0 y b – 1, tiene un rango igual a:

b × b × b × b = b^4

Por ejemplo, si representamos nuestros datos como naturales de ocho dígitos en base 2 tendremos: 2 × 2 × 2 × 2 × 2 × 2 × 2 × 2 = 28 = 256

Esto luego se conocería como el rango de un entero sin signo en 8 bits, y va del 0 al 255.

Overflow

Si una magnitud a representar cae fuera del rango de representación, tenemos una situación que se conoce como **overflow** (desborde), ya que no hay forma de representarla en formato actual. Por ejemplo, para los ocho dígitos de base dos del ejemplo, la magnitud 27 es representable:

$27 = (\mathbf{00011011})_{(2)}$

Noten que completamos con 0 los dígitos a izquierda para escribir los ocho elementos de nuestra representación finita. Ahora, la magnitud 770 no es representable por quedar fuera del rango **overflow**:

$770 = (\mathbf{1100000010})_{(2)}$

Precisamos 10 dígitos como mínimo para representar la magnitud en base 2.

Tipos de datos

Los datos tienen su **información asociada** y su **tipo de dato**. En el caso que se presentó **la información asociada son los valores de cada dígito** y **el tipo de dato serían los naturales acotados**.

El tipo nos indica cómo interpretar la información, en este caso **cómo vincular el dato con una magnitud** y qué operaciones podemos realizar con ella.

Tipos numéricos

Vamos a utilizar los siguientes tipos de datos para representar números naturales y enteros, todos a partir de la representación en base 2 (binaria), a saber:

- **Sin signo**
  representa únicamente números positivos
- **Signo + Magnitud**
  se usa el primer dígito (bit) para indicar el signo de la magnitud
- **Exceso** m
  Represento n como m + n. De esta manera, estamos desplazando la ubicación de la magnitud asociada al cero del comienzo del rango de representación a la posición m. Los valores a izquierda de m serán interpretados como negativos

Veamos ejemplos con rangos de representación para datos de 3 bits. El rango es:

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7

Los datos del rango son siempre iguales, lo que va a cambiar van a ser las magnitudes asociadas a cada elemento (cómo los interpretamos). Los datos son:

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
000	001	010	011	100	101	110	111

Las magnitudes asociadas al rango serán:

Posición	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
Dato	000	001	010	011	100	101	110	111
Sin signo	0	1	2	3	4	5	6	7
Signo+magnitud	0	1	2	3	−0	−1	−2	−3
Exceso m(m=2)	−2	−1	0	1	2	3	4	5

Noten que para signo+magnitud hay dos datos asociados al cero (el valor está desnormalizado).

Codificando números enteros en base binaria

Complemento a 2

Los positivos se representan igual.

Para los números negativos lo que se almacena es un complemento ˙n, a partir de: ˙n = 2^k (10) − n

Donde n es la magnitud interpretada, ˙n el dato almacenado y k la cantidad de dígitos (o bits) utilizados para representar al número. O sea lo que guardamos es la resta entre el primer número que se sale del rango y el valor absoluto de n.

$$\dot{n} = 2^{k}_{(10)} - n$$

Ejemplo

Quiero representar el número −2(10) con k = 3 dígitos binarios.

Entonces, 2^k (10) → 2^(3(10)) (10) = 8(10) = 1000(2)

Luego ˙2 = 1000 (2) − 2(10) donde ˙2 es el complemento a 2 del número.

Escribimos el 2 en la base correspondiente:

˙2 = 1000 (2) − 2(10) → ˙2 = 1000 (2) − 10(2).

Finalmente ˙2 = 110 (2)

Extendiendo la cantidad de bits de precisión:

Signo + Magnitud: Se extiende con 0's, pero el bit más significativo se mantiene indicando el signo.

Complemento a 2: Se extiende con el valor del bit más significativo.

Exceso a m: Se extiende siempre con 0's.

Sin Signo

Solo sirve para los positivos.

Numeral(dato) → número que representa

$1111 \to 15_{(10)}$	$0111 \to 7_{(10)}$
$1110 \to 14_{(10)}$	$0110 \to 6_{(10)}$
$1101 \to 13_{(10)}$	$0101 \to 5_{(10)}$
$1100 \to 12_{(10)}$	$0100 \to 4_{(10)}$
$1011 \to 11_{(10)}$	$0011 \to 3_{(10)}$
$1010 \to 10_{(10)}$	$0010 \to 2_{(10)}$
$1001 \to 9_{(10)}$	$0001 \to 1_{(10)}$
$1000 \to 8_{(10)}$	$0000 \to 0_{(10)}$

Para los numerales de 4 *bits*.

Signo+Magnitud

El primer bit es el signo, los demás son el *significado* (la magnitud del número en valor absoluto).

numeral → número que representa

$1111 \to -7_{(10)}$	$0111 \to 7_{(10)}$
$1110 \to -6_{(10)}$	$0110 \to 6_{(10)}$
$1101 \to -5_{(10)}$	$0101 \to 5_{(10)}$
$1100 \to -4_{(10)}$	$0100 \to 4_{(10)}$
$1011 \to -3_{(10)}$	$0011 \to 3_{(10)}$
$1010 \to -2_{(10)}$	$0010 \to 2_{(10)}$
$1001 \to -1_{(10)}$	$0001 \to 1_{(10)}$
$1000 \to -0_{(10)}$	$0000 \to 0_{(10)}$

Para los numerales de 4 *bits*.

Complemento a dos

Los numerales que representa positivos son iguales a los anteriores

Para los negativos, dado un n negativo se representan escribiendo $2^k - n$ en notación sin signo

cuentas → numeral → número que representa

$2^4 + (-1) = 15 \to$	$1111 \to -1_{(10)}$	$0111 \to 7_{(10)}$
$2^4 + (-2) = 14 \to$	$1110 \to -2_{(10)}$	$0110 \to 6_{(10)}$
$2^4 + (-3) = 13 \to$	$1101 \to -3_{(10)}$	$0101 \to 5_{(10)}$
$2^4 + (-4) = 12 \to$	$1100 \to -4_{(10)}$	$0100 \to 4_{(10)}$
$2^4 + (-5) = 11 \to$	$1011 \to -5_{(10)}$	$0011 \to 3_{(10)}$
$2^4 + (-6) = 10 \to$	$1010 \to -6_{(10)}$	$0010 \to 2_{(10)}$
$2^4 + (-7) = 9 \to$	$1001 \to -7_{(10)}$	$0001 \to 1_{(10)}$
$2^4 + (-8) = 8 \to$	$1000 \to -8_{(10)}$	$0000 \to 0_{(10)}$

Para los numerales de 4 *bits*.

Exceso a m

El número n se representa como $m + n$

cuentas $\rightarrow$ numeral $\rightarrow$ número que representa

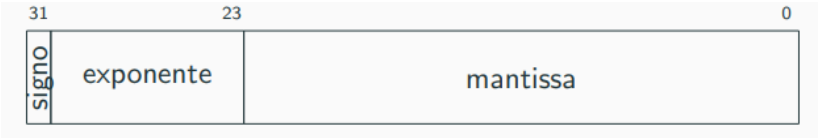
$5 + (10) = 15 \rightarrow 1111 \rightarrow 10_{(10)}$	$5 + (2) = 7 \rightarrow 0111 \rightarrow 2_{(10)}$
$5 + (9) = 14 \rightarrow 1110 \rightarrow 9_{(10)}$	$5 + (1) = 6 \rightarrow 0110 \rightarrow 1_{(10)}$
$5 + (8) = 13 \rightarrow 1101 \rightarrow 8_{(10)}$	$5 + (0) = 5 \rightarrow 0101 \rightarrow 0_{(10)}$
$5 + (7) = 12 \rightarrow 1100 \rightarrow 7_{(10)}$	$5 + (-1) = 4 \rightarrow 0100 \rightarrow -1_{(10)}$
$5 + (6) = 11 \rightarrow 1011 \rightarrow 6_{(10)}$	$5 + (-2) = 3 \rightarrow 0011 \rightarrow -2_{(10)}$
$5 + (5) = 10 \rightarrow 1010 \rightarrow 5_{(10)}$	$5 + (-3) = 2 \rightarrow 0010 \rightarrow -3_{(10)}$
$5 + (4) = 9 \rightarrow 1001 \rightarrow 4_{(10)}$	$5 + (-4) = 1 \rightarrow 0001 \rightarrow -4_{(10)}$
$5 + (3) = 8 \rightarrow 1000 \rightarrow 3_{(10)}$	$5 + (-5) = 0 \rightarrow 0000 \rightarrow -5_{(10)}$

Para los numerales de 4 *bits* en exceso 5.

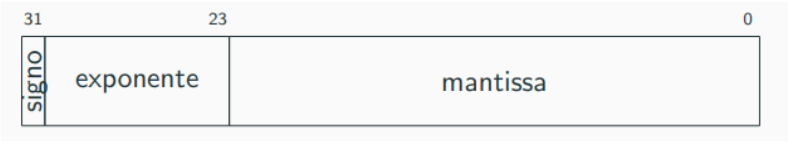
Para interpretar un valor, o sea una tira de valores binarios o bits, es necesario conocer su tipo. Tipos distintos para un mismo valor determinan (potencialmente) distintas magnitudes.

Representación de racionales, IEEE 754

Sólo para comparar un poco con las cosas que ya vimos, ya que en la materia no vamos a trabajar con representación de racionales en la práctica, veamos algunas similitudes entre los formatos. La así llamada representación de punto flotante propuesta en IEEE 754 una representación de punto flotante de precisión simple (32 bits) tiene este formato:



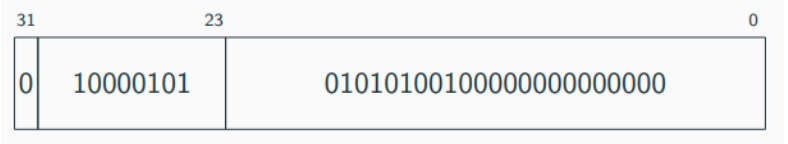
Donde el primer bit indica el signo (como en signo+magnitud o complemento a 2), el exponente indica a cuánto debe elevarse la mantisa en exceso 127 (como en exceso m). Para interpretar una magnitud a almacenada como:



Aplicamos:

$$a = -1^{signo} * (1 + mantissa * 2^{-23}) * 2^{(exponente-127)}$$

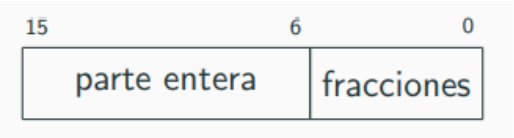
Por ejemplo, la magnitud 85.125 se representa como:



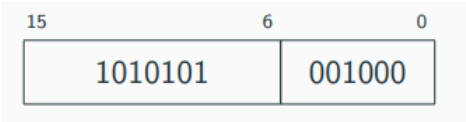
Ya que:
85,125(10) = 1010101,001(2) = 1,010101001(2)x2^(6(10)) (10)
El primer 1 de la mantissa no se almacena, o sea que está implícito, luego el signo es 0 por ser positivo y el exponente se guarda en exceso 127, o sea:

127(10) + 6(10) = 133(10) = 10000101(2)

Otra forma de representar racionales utiliza un formato más directo, donde una parte del dato está destinado a valores enteros y una parte a fracciones, por ejemplo podría tener un dato de 16 bits con seis bits para fracciones:



Donde los primeros seis dígitos a5 . . . a1a0 definen la magnitud de la siguiente forma:
$a_5 * 2^{-1} + \ldots + a_1 * 2^{-5} + a_0 * 2^{-6}$
Habiendo visto cómo interpretar la magnitud en punto fijo y recordando que nuestro ejemplo es:



Ya que: 85,125(10) = 1010101,001(2) = 1010101(2) + 1000x(2^−6(10)) (10)

Operaciones aritmético-lógicas

Vamos a presentar algunas operaciones aritméticas y lógicas en base 2, en particular **o lógico, y lógico, xor lógico, negación lógica, desplazamientos, suma, resta y multiplicación**. Todas las operaciones van a estar asociadas a un algoritmo de resolución, del mismo modo que utilizamos uno para realizar cuentas en base 10 en la vida cotidiana.

Propiedades de los datos y las operaciones

Es importante comprender que hay atributos que nos interesa observar tanto de **los datos como de las operaciones**.
Por ejemplo:
De los datos numéricos si son negativos o pares
De las operaciones si los resultados se mantienen dentro del rango de representación

Veamos cómo observar el dato nos permite determinar propiedades del valor representado. Por ejemplo, en complemento a 2 de un dato de 4 bits:

Posición	v_3	v_2	v_1	v_0
Interpretación	*signo*	*x*	*x*	*paridad*
Negativo	1	*x*	*x*	*x*
Par	*x*	*x*	*x*	0

Operaciones lógicas

Las operaciones lógicas que veremos pueden involucrar a uno o dos operandos, **se aplican sobre el dato almacenado, o sea los bits del valor representado**. Las presentamos rápidamente y una por una.

O lógico

El **o lógico** o disyunción se aplica bit a bit. La operación a ∨ b = c se puede describir atómicamente (por cada elemento indivisible) como ci = ai ∨ bi .
Veamos un ejemplo de 4 bits, noten que la aplicación no depende del tipo de dato, lo trata indistintamente:

Posición	v_3	v_2	v_1	v_0
a	1	0	1	0
b	0	0	1	1
$c = a \vee b$	1	0	1	1

Nota

Podemos notar que las operaciones lógicas se aplican **bit a bit**:

Posición	v_3	v_2	v_1	v_0
a	1	0	1	0
	+	+	+	+
b	0	0	1	1
	↓	↓	↓	↓
$c = a \vee b$	1	0	1	1

Es posible encontrar al signo + para representar la disyunción, porque equivale a una suma sin acarreo en bits. Lo mismo sucede entre la conjunción y el signo ∗.

Y lógico

El **y lógico** se aplica bit a bit. La operación a ∧ b = c se puede describir atómicamente (por cada elemento indivisible) como ci = ai ∧ bi . Veamos un ejemplo de 4 bits:

Posición	v_3	v_2	v_1	v_0
a	1	0	1	0
b	0	0	1	1
$c = a \wedge b$	0	0	1	0

Xor lógico

El **xor lógico** se aplica bit a bit. La operación a ⊻ b = c se puede describir atómicamente (por cada elemento indivisible) como ci = (ai ∧ ¬bi) ∨ (¬ai ∧ bi), exactamente uno de los bits vale 1. Veamos un ejemplo de 4 bits:

Posición	v_3	v_2	v_1	v_0
a	1	0	1	0
b	0	0	1	1
$c = a \veebar b$	1	0	0	1

Negación lógica

La negación lógica se aplica bit a bit. La operación ¬a = c se puede describir atómicamente (por cada elemento indivisible) como ci = ¬ai , dando vuelta los valores de cada elemento. Veamos un ejemplo de 4 bits:

Posición	v_3	v_2	v_1	v_0
a	1	0	1	0
$c = \neg a$	0	1	0	1

Desplazamiento a izquierda

El desplazamiento a izquierda se aplica desplazando los bits del dato tantas posiciones como se indiquen a izquierda. La operación a<< n = c para un dato de k bits se puede describir atómicamente como ci = a_{i-n} si i ≤ k − n − 1 y 0 en caso contrario. Veamos un ejemplo de 4 bits:

Posición	v_3	v_2	v_1	v_0
a	1	0	1	0
$c = a \ll 2$	1	0	0	0

Desplazamiento lógico a derecha

El desplazamiento lógico a derecha se aplica desplazando los bits del dato tantas posiciones como se indiquen a derecha. La operación a >>l n = c para un dato de k bits se puede describir atómicamente como ci = a_{i+n} si i ≥ n y 0 en caso contrario. Veamos un ejemplo de 4 bits:

Posición	v_3	v_2	v_1	v_0
a	1	0	1	0
$c = a \gg_l 2$	0	0	1	0

Desplazamiento aritmético a derecha

El desplazamiento aritmético a derecha se aplica desplazando los bits del dato tantas posiciones como se indiquen a derecha, pero copiando el valor del bit más significativo de origen en los valores vacantes del resultado. La operación a >>a n = c para un dato de k bits se puede describir atómicamente como ci = a_{i+n} si i ≥ n y a_{k-1} en caso contrario. Veamos un ejemplo de 4 bits:

Posición	v_3	v_2	v_1	v_0
a	1	0	1	0
$c = a \gg_a 2$	1	1	1	0

La aplicación de una operación lógica puede tener muchos motivos, pero en particular el desplazamiento a derecha o izquierda en n posiciones tiene el efecto de **multiplicar o dividir por 2^n el valor representado,** siempre que se trate de un entero sin signo o en complemento a dos.

Adición Binaria

Para realizar operaciones aritméticas como la suma, la resta o la multiplicación, podemos intentar utilizar dos conceptos importantes, por un lado el **razonamiento composicional** y por el otro un **análisis extensivo de los casos**. El primero tiene que ver con tratar de resolver la suma representada en base 2 a partir de cada dígito (descomposición) y el segundo con construir una tabla que calcule todos los resultados posibles para sumas de un dígito en base 2 (análisis extensivo).
Veamos qué puede suceder cuando sumamos dos dígitos cualesquiera en base dos:

a_i	b_i	$a_i + b_i$
0	0	0
0	1	1
1	0	1
1	1	10

Podemos notar que la última fila produce un resultado que no se puede representar con un sólo dígito. Para el caso atómico (de un único dígito) ese 1 va a conocerse como **carry** o acarreo porque va a afectar la suma del dígito inmediato a izquierda.
Redefinimos el resultado, dividiendo ai + bi entre el resto ri y su acarreo ci :

a_i	b_i	c_i	r_i
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Podemos extender la suma, para tener en cuenta el hecho de que quizás el dígito inmediato a derecha produjo acarreo en su suma, donde ci es el carry (acarreo) del dígito anterior:

a_i	b_i	c_{i-1}	c_i	r_i
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

Con esto ya podemos definir la adición binaria para dos números a y b sin signo de n dígitos de la siguiente forma:

$carry:$	c_{n-1}	c_{n-2}	$\dots$	c_0	
$a:$	$-$	a_{n-1}	$\dots$	a_1	a_0
$b:$	$-$	b_{n-1}	$\dots$	b_1	b_0
$a+b:$	c_{n-1}	r_{n-1}	$\dots$	r_1	r_0

Aquí cn es lo que consideramos el acarreo de la suma, y se puede interpretar como un indicador o **flag** de la propiedad de desborde de la operación, ya que el resultado no es representable en n bits.
Ejemplo de la suma

En decimal	En base 3
8 4 5	2 1 2
+ 3 4 2	+ 1 0 1
1 1 8 7	1 0 2 0

Resta Binaria

Podemos definir equivalentemente la resta binaria para dos números a y b sin signo de n dígitos de la siguiente forma:

$borrow:$	b_{n-1}	b_{n-2}	$\dots$	b_0	
$a:$	$-$	a_{n-1}	$\dots$	a_1	a_0
$b:$	$-$	b_{n-1}	$\dots$	b_1	b_0
$a-b:$	b_{n-1}	r_{n-1}	$\dots$	r_1	r_0

Aquí bn es lo que consideramos el préstamo o *borrow* de la resta. En este caso el *borrow* se produce cuando el sustraendo es mayor que el minuendo, por lo tanto, se debe "pedir" al dígito adyacente.

Sobre el Overflow y el Carry...

El truco para detectar un overflow es observar que si el bit de signo es igual en ambos operandos (ambos positivos o negativos) el resultado de la suma debería preservar el signo (suma de positivos produce un positivo, suma de negativos produce un negativo).

$$\text{overflow} \Longleftrightarrow ((a_{n-1} = b_{n-1}) \wedge (a_{n-1} \neq c_{n-1}))$$

Noten que el **overflow** y el **carry** son propiedades de una operación, a diferencia de la paridad o el signo, que son propiedades de un dato aislado. Para determinar si se cumple una propiedad de una operación, puede ser necesario observar tanto los operandos como el resultado (caso del overflow).