Representación de números en binario

Representación de números en binario

- Enteros.
- Overflow de enteros.
- Reales.
- Overflow y underflow de reales.

Enteros sin signo

- Equivalente a `unsigned int` de C/C++.
- Los enteros sin signo se representan en binario.
- Ejemplo:
- $17_{10} = 00010001_2$ con 8 bits.

Enteros con signo

- Equivalente a `int` de C/C++/Java.
- Los números no negativos se representan en binario.
- Ejemplo: $7_{10} = 00000111_2$ con 8 bits.
- Los números negativos se representan en complemento a 2.

Complemento a 2

- Convertir el valor absoluto del número a base 2.
- Intercambiar ceros por unos y viceversa.
- Sumar 1 al resultado.
- Ejemplo: -5_{10} con 4 bits.
- 5 = 0101.
- Invertir el número: 1010.
- Sumar 1: 1011
- Conclusión: $-5_{10} = 1011_2 = B_{16}$.

Complemento a 2

- **Ojo**: el complemento a 2 depende del número de bits que se usen para representar enteros (típicamente 32 o 64 bits).
- Ejemplos:
- $-5_{10} = 1011_2 = B_{16}$ en una CPU de 4 bits.
- $-5_{10} = 11111011_2 = FB_{16}$ en una CPU de 8 bits.
- $-5_{10} = 1111111111111011_2 = FFFB_{16}$ en una CPU de 16 bits.

Complemento a 2

- Usando complemento a 2 la resta se vuelve una suma.
- *A* – *B* se convierte en *A* + (-*B*).
- Ejemplo: 17 – 9.
- En binario: 10001 – 01001 se convierte en:

10001 (17)

+ 10111 (-9)

101000 (8 porque siempre se elimina el carry final)

Rangos

- Con n bits, los rangos son:
- Enteros sin signo: 0 a $2^n - 1$.
- Enteros con signo: -2^{n-1} a $2^{n-1} - 1$.
- Ejemplo:
- Con 4 bits:
- Enteros sin signo: 0 a $2^4 - 1 = 0$ a 15.
- Enteros con signo: -2^3 a $2^3 - 1 = -8$ a 7.

Rangos con 4 bits

- Enteros sin signo:

Base 10	Base 2	Base 10	Base 2
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

Rangos con 4 bits

- Enteros con signo:

Base 10	Base 2	Base 10	Base 2
0	0000	-8	1000
1	0001	-7	1001
2	0010	-6	1010
3	0011	-5	1011
4	0100	-4	1100
5	0101	-3	1101
6	0110	-2	1110
7	0111	-1	1111

Overflow con enteros

- El overflow ocurre cuando el resultado de una operación no se puede representar en el hardware.
- Con 4 bits, el rango de enteros con signo, usando complemento a dos para los negativos, es de -8 a +7.
- La suma 5 + 6 genera overflow.
- La resta -5 – 6 genera overflow

Overflow con enteros

- Sumando 5 + 6 con 4 bits:

```
   0101 (+5)
+  0110 (+6)
------------
   1011 (-5) ⇦ ¡error!
```

- Restando -5 – 6 con 4 bits:

```
   1011 (-5)
+  1010 (-6)
------------
   0101 (+5) ⇦ ¡error!
```

Detectando overflow

- El overflow ocurre en la suma cuando:
  - Al sumar dos positivos el resultado es negativo.
  - Al sumar dos negativos el resultado es positivo.
- El overflow ocurre en la resta cuando:
  - Al restar un negativo de un positivo el resultado es negativo.
  - Al restar un positivo de un negativo el resultado es positivo.

Detectando overflow

Operación	A	B	Resultado indicando overflow
A + B	> 0	> 0	< 0
A + B	< 0	< 0	> 0
A - B	> 0	< 0	< 0
A - B	< 0	> 0	> 0

Números reales

- Standard IEEE 754-2008.
- El standard define:
  - Formatos para representar números reales de punto flotante
  - Formatos de intercambio
  - Reglas de redondeo
  - Operaciones
  - Manejo de excepciones
- Usado por toda CPU diseñada desde 1980.

Conceptos

- Normalización de números reales.
- Un número real está normalizado si tiene un solo dígito (distinto de cero) en la parte entera.
- PI = 3.1415926 es un número real normalizado.
- 0.00007 es un número real sin normalizar.
- Para normalizarlo hay que recorrer el punto decimal a la derecha.
- 7×10^{-5} está normalizado.

Normalización

- 724.45 no está normalizado.
- Para normalizarlo hay que recorrer el punto decimal a la izquierda.
- 7.2445×10^2 está normalizado.
- En resumen:
- Si el punto decimal se recorre a la izquierda, se le suma uno al exponente por cada posición recorrida.
- Si el punto decimal se recorre a la derecha, se le resta uno al exponente por cada posición recorrida.

Normalización en binario

- Se aplican las mismas reglas.
- 10100.101 no está normalizado.
- Para normalizarlo hay que recorrer el punto decimal a la izquierda.
- 1.0100101×2^4 está normalizado.
- 0.011 no está normalizado.
- Para normalizarlo hay que recorrer el punto decimal a la derecha.
- 1.1×2^{-2} está normalizado.

Normalización en binario

- **Importante**: un número binario real normalizado siempre tiene un 1 en las unidades.

Precisión sencilla vs doble

- En Java:

Tipo	Precisión	Tamaño	Rango
float	Sencilla	4 bytes	1.4012984632481707e-45 a 3.4028234663852860e+38 (positivo o negativo)
double	Doble	8 bytes	4.94065645841246544e-324 a 1.79769313486231570e+308 (positivo o negativo)

Formatos IEEE 754-2008

Name	Common name	Base	Digits	Decimal digits	Exponent bits	Decimal E max	Exponent bias[6]	E min	E max	Notes
binary16	Half precision	2	11	3.31	5	4.51	$2^4-1 = 15$	−14	+15	not basic
binary32	Single precision	2	24	7.22	8	38.23	$2^7-1 = 127$	−126	+127	
binary64	Double precision	2	53	15.95	11	307.95	$2^{10}-1 = 1023$	−1022	+1023	
binary128	Quadruple precision	2	113	34.02	15	4931.77	$2^{14}-1 = 16383$	−16382	+16383	
binary256	Octuple precision	2	237	71.34	19	78913.2	$2^{18}-1 = 262143$	−262142	+262143	not basic
decimal32		10	7	7	7.58	96	101	−95	+96	not basic
decimal64		10	16	16	9.58	384	398	−383	+384	
decimal128		10	34	34	13.58	6144	6176	−6143	+6144	

Ver https://en.wikipedia.org/wiki/IEEE_floating_point

Precisión sencilla (float)

- Guarda el número real usando 4 bytes (32 bits).
- Los 32 bits se dividen en:
  - 1 bit de signo (1 negativo, 0 positivo).
  - 8 bits para el exponente.
  - 23 bits para la mantisa.

Precisión sencilla (float)

- Notas:

1. El exponente se guarda en exceso a 127.
2. En el campo de mantisa se guarda solo la parte fraccionaria, el 1 de la parte entera no se almacena.
3. La mantisa se guarda justificada a la izquierda rellenando de ceros a la derecha si es necesario.
4. El número se reconstruye así:

$$N = (-1)^s \times (1 + \text{mantisa}) \times 2^{E-127}$$

Ejemplo 1

- Representar 17.15 en el standard IEEE 754-2008 con precisión sencilla.
- El bit de signo es 0
- Se pasa 17.15 a binario 10001.00100110011001…
- Se normaliza $1.00010010011001…x\ 2^4$
- El exponente, 4, en exceso 127 es 131. En binario es 10000011
- La parte fraccionaria de la mantisa extendida a 23 bits es 00010010011001100110011

Ejemplo 1

- En conclusión, 17.15 se representa en binario como 01000001100010010011001100110011
- En base 16
- 0x41893333

Ejemplo 2

- Representar -118.625 en el standard IEEE 754-2008 con precisión sencilla.
- El bit de signo es 1
- 118.625 en binario es 1110110.101
- El binario normalizado es 1.110110101×2^6
- El exponente, 6, en exceso 127 es 133. En binario es 10000101
- La parte fraccionaria de la mantisa, rellenando con ceros a la derecha, es 11011010100000000000000

Ejemplo 2

- En conclusión -118.625 se representa en binario como 11000010111011010100000000000000
- En base 16
- 0xC2ED4000

Características principales

- Precisión sencilla:
- Dos ceros:
  - Cero positivo (+0): s = 0, e = 0, m = 0
  - Cero negativo (-0): s = 1, e = 0, m = 0
- Dos infinitos:
  - Infinito positivo: s = 0, e = 255, m = 0
  - Infinito negativo: s = 1, e = 255, m = 0
- Dos NaN (not a number):
  - NaN positivo: s = 0, e = 255, m > 0
  - NaN negativo: s = 1, e = 255, m > 0

Características principales

- Números más grandes:
  - Positivo: $2^{127} = 1.7014118 \times 10^{38}$
  - Negativo: $-2^{127} = -1.7014118 \times 10^{38}$
- Números más pequeños:
  - Normalizado positivo: $2^{-126} = 1.175494351 \times 10^{-38}$
  - Normalizado negativo: $-2^{-126} = -1.175494351 \times 10^{-38}$

Precisión doble (double)

- Guarda el número real usando 8 bytes (64 bits).
- Los 64 bits se dividen en:
  - 1 bit de signo (1 negativo, 0 positivo).
  - 11 bits para el exponente.
  - 52 bits para la mantisa.
- El exponente se guarda en exceso a 1023.

Resumen

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Overflow y underflow

- Se produce un overflow cuando un exponente positivo es tan grande que no cabe en el campo exponente.
- Se produce un underflow cuando un exponente negativo es tan grande que no cabe en el campo exponente.