UNIVERSITÀ
degli STUDI
di CATANIA

Variabili, tipi primitivi, operazioni aritmetiche in C

Corso di programmazione I

Corso di Laurea Triennale in Informatica

Prof. Giovanni Maria Farinella
Web: http://www.dmi.unict.it/farinella
Email: gfarinella@dmi.unict.it

Dipartimento di Matematica e Informatica

Indice

1. Variabili, costanti e commenti in C
2. Sistema di numerazione in base 2
3. Rappresentazione dei numeri al calcolatore: standard IEEE 754
4. Tipi in C
5. Operatori aritmetici, funzioni matematiche di base, conversioni

Variabili, costanti e commenti in C

Variabili in C

Cosa è una variabile?

Un contenitore di dati identificato da un **nome** all'interno del programma.

La variabile corrisponde ad un certo **indirizzo nella memoria del calcolatore** nel quale si esegue il programma.

Una variabile è associato un certo **tipo**, che deve essere **adeguato** a rappresentare l'informazione che si vuole memorizzare all'interno della variabile.

Variabili in C

Definizione di variabile

```
int numero_di_ordini = 10;
```

↯

Usare un **nome** che "descriva" il contenuto della variabile.

Inizializzare la variabile? Il C non lo rende obbligatorio ma è spesso necessario per evitare errori.

Il carattere "**;**" termina la singola istruzione che definisce la variabile.

Variabili in C

Definizione senza inizializzazione:

```
int j;
```

Definisce una variabile `int` e la inizializza con il valore 10:

```
int numero_di_ordini = 10;
```

Il contenuto della `totale_ordini_euro` viene inizializzato con il valore attuale di alcune variabili:

```
int totale_ordini_euro = prezzo * quantita;
```

Variabili in C

Errore in fase di compilazione!

```
int codice = "AA10";
```

Definisce una o più variabili contemporaneamente:

```
int v1, v2=3, v3;
```

NB: v1 e v3 nessun valore iniziale, v2 inizializzata!

Definisce una variabile di tipo carattere (e la inizializza):

```
char c='a';
```

Variabili in C

Valori letterali e variabili

Inizializziamo le variabili con espressioni che usano **letterali e/o altre variabili**.

Un valore **letterale** è un elemento del programma che rappresenta un valore.

`true`, `1.0`, `40` sono letterali di tipo, rispettivamente booleano, double, e intero.

`"acqua"` è un valore letterale di tipo stringa, mentre 'c' è un letterale di tipo char.

Variabili in C

Letterale	Tipo	Note
-6	int	int non ha parte frazionaria, può essere negativo
0.5	double	Viene rappresentato in memoria come un double
$0.5f$	float	Viene rappresentato in memoria come un float
1E6	double	Notazione esponenziale. Eqauivale a 1×10^6 oppure 1000000
10,456	#	**Errore** in fase di compilazione! Va usato il punto, non la vigola..
3 1/2	#	**Errore** in fase di compilazione! Va usata una espressione in forma decimale: 3.5

Variabili in C

Regole per i nomi delle variabili

- nomi debbono **iniziare** con una lettera, oppure underscore ("_");
- i **rimanenti** caratteri possono essere anche numeri, oppure ancora lettere o underscore.
- **NO spazi** nel nome delle variabili!
- ATTENZIONE: C **case-sensitive**!
  - `var` e `Var` non sono la stessa variabile.
- Le parole *riservate* (e.g. `double`) non si possono usare per i nomi di variabile..

Variabili in C

In che modo può cambiare il valore di una variabile nel tempo?

- Assegnamento: ES: `a=10;`, `b=z-20;`
- Incremento/decremento: (forma postfissa) `a++; b--;`, (forma prefissa) `++a; --b;`

Variabili in C

Usare la stessa variabile a destra e sinistra di un assegnamento.

Cosa succede nel seguente assegnamento?

$$\texttt{var} = \texttt{var} + 10;$$

1. **Leggi** il contenuto della variabile `var`
2. **Somma**, al valore letto in precedenza, il valore 10
3. **Copia** il valore ottenuto dalla precedente somma nella variabile `var`

Ordine di inizializzazione delle variabili

Il seguente codice è (semanticamente) corretto?

```
1 int a=10;
2 int b;
3 int volume = a * b * altezza;
4 b=15;
```

Attenzione alla inizializzatione (tardiva) di `b`.

Quale sarà il valore di `b` nella valutazione della espressione alla riga 3?

Variabili costanti

```
const int valore_banconota_A = 10;
```

Si vuole assegnare un nome ad uno o più valori **costanti**.

La parola riservata const per una variabile permette di indicare al compilatore che

- il valore di tale variabile const **non può cambiare** rispetto al suo valore iniziale.
- quella variabile const va **inizializzata in fase di creazione**.

Variabili costanti

Le costanti migliorano la **leggibilità**, quindi la comprensione del codice e permettono di ridurre gli errori in fase di codifica.

Esempio

(A)
```
int somma_iniziale = num_banconote * 10;
```

VS

(B)
```
int somma_iniziale = num_banconote * VALORE_BANCONOTA;
```

Variabili costanti

Esempio

```
int somma_iniziale = num_banconote * 10;
```

VS

(B)
```
int somma_iniziale = num_banconote * VALORE_BANCONOTA;
```

E se il programmatore avesse la necessità di cambiare il valore corrispondente a `valore_banconota` da 10 a 20?

Con (A) può farlo cambiando una singola riga di istruzione, mentre con (B)...

Variabili costanti

Il compilatore **ignora** il testo che rappresenta un commento

Commento singola linea

```
float alpha = 0.5; // deve essere < beta
```

Commento multi-linea

```
/*
Questo programma calcola il profitto medio:
- mensile
- annuo
*/
int main(){...}
```

Sistema di numerazione in base 2

Sistemi di numerazione posizionale

Nei sistemi di numerazione posizionale, i simboli assumono valori diversi in base alla **posizione** che occupano nella notazione.

Esempio

$$
\begin{aligned}
102_{10} &= 1 \times 10^2 + 0 \times 10^1 + 2 \times 10^0 \\
10101010_2 &= 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + \\
&\quad + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 170_{10}
\end{aligned}
$$

Il sistema binario è un sistema di numerazione posizionale in base **2**, mentre quello decimale è un sistema di numerazione posizionale in base **10**.

Sistema numerico binario

Conversione da base 2 a base 10. Numeri interi

$$\mathbf{10101010}_2 = 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + \\ + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 170_{10}$$

Da base 2 a base 10. Numeri con parte frazionaria

Conversione in base 10 del numero $\mathbf{101.0101}_2$

$$\mathbf{101}_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5_{10}$$
$$\mathbf{0101} = 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = \\ = \mathbf{0.3125}_{10}$$

Conversione da base 10 a base 2. Numeri interi

$136_{10} = \mathbf{10001000_2}$.

$136 : 2$	$=$	68	$r=\mathbf{0}$
$68 : 2$	$=$	34	$r=\mathbf{0}$
$34 : 2$	$=$	17	$r=\mathbf{0}$
$17 : 2$	$=$	8	$r=\mathbf{1}$
$8 : 2$	$=$	4	$r=\mathbf{0}$
$4 : 2$	$=$	2	$r=\mathbf{0}$
$2 : 2$	$=$	1	$r=\mathbf{0}$
$1 : 2$	$=$	0	$r=\mathbf{1}$

Conversione da base 10 a base 2. Numeri interi

$136_{10} = \mathbf{10001000_2}$.

$136 : 2$	$=$	68	r=**0**
$68 : 2$	$=$	34	r=**0**
$34 : 2$	$=$	17	r=**0**
$17 : 2$	$=$	8	r=**1**
$8 : 2$	$=$	4	r=**0**
$4 : 2$	$=$	2	r=**0**
$2 : 2$	$=$	1	r=**0**
$1 : 2$	$=$	0	r=**1**

I resti della divisione **in ordine inverso** costituirano la codifica binaria.

Conversione da base 10 a base 2. Numeri interi

$136_{10} = \mathbf{10001000_2}$.

$136 : 2$	$=$	68	r=**0**
$68 : 2$	$=$	34	r=**0**
$34 : 2$	$=$	17	r=**0**
$17 : 2$	$=$	8	r=**1**
$8 : 2$	$=$	4	r=**0**
$4 : 2$	$=$	2	r=**0**
$2 : 2$	$=$	1	r=**0**
$1 : 2$	$=$	0	r=**1**

Il resto ottenuto dalla **prima divisione** occuperà **l'ultima posizione** nella codifica (bit **meno significativo** o più a destra), e così via ...

Conversione da base 10 a base 2. Numeri interi

$136_{10} = \mathbf{10001000_2}$.

$136 : 2$	$=$	68	r=$\mathbf{0}$
$68 : 2$	$=$	34	r=$\mathbf{0}$
$34 : 2$	$=$	17	r=$\mathbf{0}$
$17 : 2$	$=$	8	r=$\mathbf{1}$
$8 : 2$	$=$	4	r=$\mathbf{0}$
$4 : 2$	$=$	2	r=$\mathbf{0}$
$2 : 2$	$=$	1	r=$\mathbf{0}$
$1 : 2$	$=$	0	r=$\mathbf{1}$

. . . Infine il resto ottenuto **dall'ultima divisione**, che occuperà la **prima posizione** nella codifica (bit **più significativo** o più a sinistra).

Da base 10 a base 2. Numeri con parte frazionaria

Il numero $28.125_{10} = 11100.001_2$

Parte intera (si converte nel solito modo): $\mathbf{28_{10} = 11100}$

Parte frazionaria:

$$\left|\begin{array}{lcll} 0.125 \times 2 & = & 0.250 & \text{riporto } \mathbf{0} \\ 0.250 \times 2 & = & 0.500 & \text{riporto } \mathbf{0} \\ 0.500 \times 2 & = & 1.000 & \text{riporto } \mathbf{1} \\ 0.000 \times 2 & = & 0.000 & \text{FINE} \end{array}\right|$$

$\mathbf{0.125_{10} = 0.001_2}$

Da base 10 a base 2. Numeri con parte frazionaria

Il numero $28.125_{10} = 11100.001_2$

Parte intera (si converte nel solito modo): $\mathbf{28_{10} = 11100}$

Parte frazionaria:

$$\left|\begin{array}{lcll} 0.125 \times 2 & = & 0.250 & \text{riporto } \mathbf{0} \\ 0.250 \times 2 & = & 0.500 & \text{riporto } \mathbf{0} \\ 0.500 \times 2 & = & 1.000 & \text{riporto } \mathbf{1} \\ 0.000 \times 2 & = & 0.000 & \text{FINE} \end{array}\right|$$

$\mathbf{0.125_{10} = 0.001_2}$

I **riporti costituiranno la codifica binaria** della parte frazionaria.

Da base 10 a base 2. Numeri con parte frazionaria

Il numero $28.125_{10} = 11100.001_2$

Parte intera (si converte nel solito modo): $\mathbf{28_{10} = 11100}$

Parte frazionaria:

$$\left|\begin{array}{lcll} 0.125 \times 2 & = & 0.250 & \text{riporto } \mathbf{0} \\ 0.250 \times 2 & = & 0.500 & \text{riporto } \mathbf{0} \\ 0.500 \times 2 & = & 1.000 & \text{riporto } \mathbf{1} \\ 0.000 \times 2 & = & 0.000 & \text{FINE} \end{array}\right|$$

$\mathbf{0.125_{10} = 0.001_2}$

Il riporto ottenuto dalla prima moltiplicazione occuperà la **prima posizione** nella codifica (bit **più significativo**), e così via ...

Da base 10 a base 2. Numeri con parte frazionaria

Il numero $28.125_{10} = 11100.001_2$

Parte intera (si converte nel solito modo): $\mathbf{28_{10} = 11100}$

Parte frazionaria:

$$\left|\begin{array}{lcll} 0.125 \times 2 & = & 0.250 & \text{riporto } \mathbf{0} \\ 0.250 \times 2 & = & 0.500 & \text{riporto } \mathbf{0} \\ 0.500 \times 2 & = & 1.000 & \text{riporto } \mathbf{1} \\ 0.000 \times 2 & = & 0.000 & \text{FINE} \end{array}\right|$$

$\mathbf{0.125_{10} = 0.001_2}$

. . . infine, il riporto ottenuto dall'ultima moltiplicazione occuperà **l'ultima posizione** nella codifica (bit **meno significativo**)

Da base 10 a base 2. Arrotondamento per troncamento

Il numero $\mathbf{17.55_{10}} = \mathbf{10001.10001100_2}$ (*)

Parte intera: $\mathbf{17_{10}} = \mathbf{10001}$

Parte frazionaria:

0.55×2	$=$	1.10	riporto $\mathbf{1}$
0.10×2	$=$	0.20	riporto $\mathbf{0}$
0.20×2	$=$	0.40	riporto $\mathbf{0}$
0.40×2	$=$	0.80	ripotto $\mathbf{0}$
0.80×2	$=$	1.60	riporto $\mathbf{1}$
0.60×2	$=$	1.20	riporto $\mathbf{1}$
0.20×2	$=$	0.40	riporto $\mathbf{0}$
0.40×2	$=$	0.80	riporto $\mathbf{0}$
$\ldots$	$=$	$\ldots$	$\ldots$

Parte frazionaria:

0.55×2	$=$	1.10	riporto $\mathbf{1}$
0.10×2	$=$	0.20	riporto $\mathbf{0}$
0.20×2	$=$	0.40	riporto $\mathbf{0}$
0.40×2	$=$	0.80	ripotto $\mathbf{0}$
0.80×2	$=$	1.60	riporto $\mathbf{1}$
0.60×2	$=$	1.20	riporto $\mathbf{1}$
0.20×2	$=$	0.40	riporto $\mathbf{0}$
0.40×2	$=$	0.80	riporto $\mathbf{0}$
$\ldots$	$=$	$\ldots$	$\ldots$

(*) Arrotondamento per **troncamento** alla ottava cifra:

$\mathbf{0.55_{10} = 0.10001100_2, \Rightarrow 17.55_{10} = 10001.10001100}$

Sistema numerico binario

Nel precedente esempio si è ottenuta una codifica binaria *parziale* del numero 0.55_{10}.

Bisognerebbe avere a disposizione più bit! Quanti? Infiniti!

Infatti alcuni numeri (con parte frazionaria) **non si possono rappresentare con un numero finito di cifre**!

Ciò accade sia nel sistema decimale, che nel sistema binario:

Ci vorrebbero infiniti bit!

Base 10: $\frac{1}{3} = 0.3333\ldots3\ldots = 0.\overline{3}$.

Base 2: $4.35_{10} = 100.0101100\ldots\ 1100\ldots = 100.010\overline{1100}$

Rappresentazione dei numeri al calcolatore: standard IEEE 754

Rappresentazione dei numeri nei calcolatori

Tipicamente, per la rappresentazione dei numeri nei calcolatori si impiegano **sequenze di bit di lunghezza variabile** (8, 16, 32, 64, ...).

Numeri interi

Per rappresentare gli **interi (con o senza segno)**, i bit si impiegano per rappresentare:

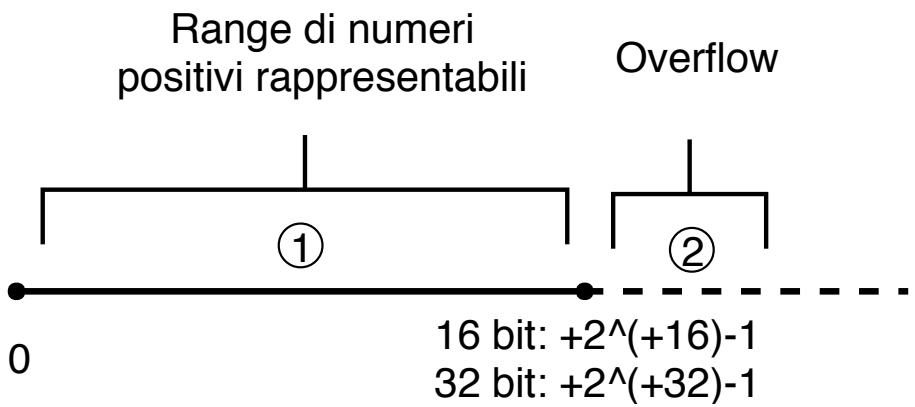
- il **valore assoluto** (o modulo) del numero stesso.
- Eventuale **segno**. In questo caso si "perde" un bit: il range di valori rappresentabili, in modulo, è dimezzato rispetto alla rappresentazione senza segno.

Numeri interi senza segno

Codifica di Numeri interi **senza segno** con 16 bit:
Intervallo numerico: $[0, 2^{16} - 1] = [0, 65.535]$

Codifica di numeri interi **senza segno** con 32 bit:
Intervallo numerico: $[0, 2^{32} - 1] = [0, 4294967295]$

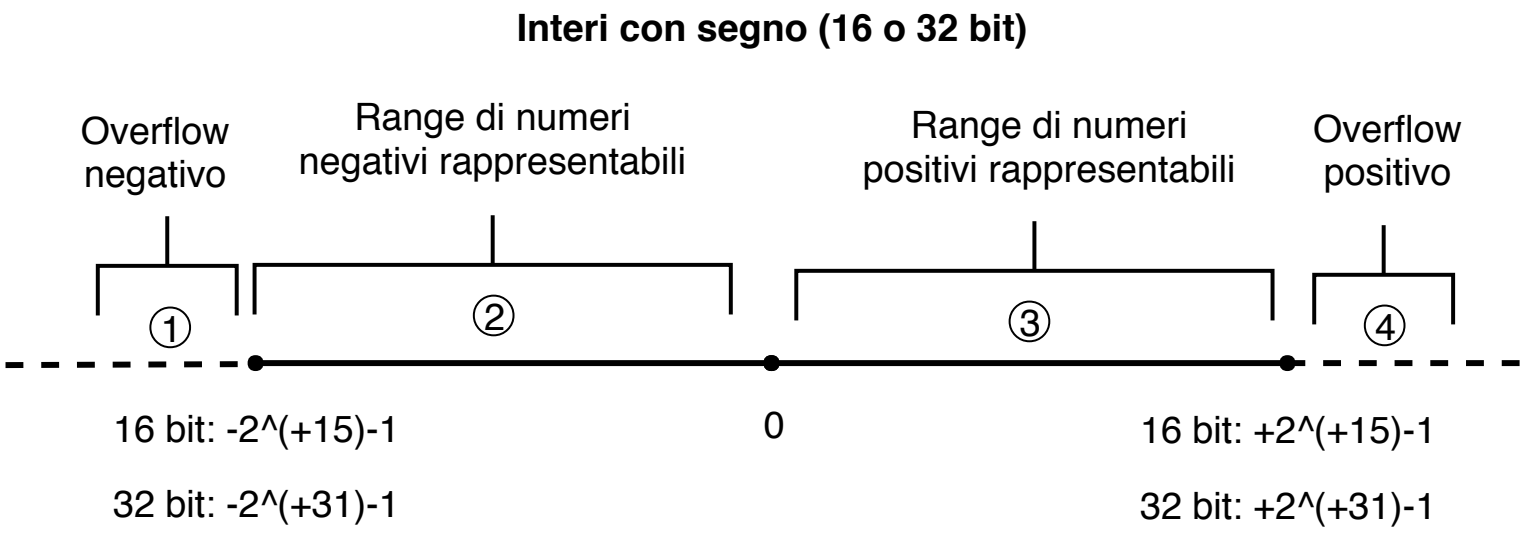
Interi senza segno (16/32 bit)



Numeri interi con segno

Codifica di numeri interi **con segno** a 16 bit:
Intervallo numerico $\pm(2^{15}-1) = 32767$ (1 bit per il segno)

Codifica di numeri interi **con segno** a 32 bit:
Intervallo numerico $\pm(2^{31}-1) = 2.147.483.647$ (1 bit per il segno)



Numeri reali

Nel calcolatore, i **numeri reali** sono rapprentati mediante un formato detto in **virgola mobile** (floating point).

Si tratta di una rappresentazione in forma compatta che deriva dalla rappresentazione scientifica.

Esempio in base 10

a) $96.103 = 0.96103 \times 10^{+2}$
b) $2.96 = 0.296 \times 10^{+1}$
c) $2.96 = 29.6 \times 10^{-1}$

Numeri reali

Esempio in base 10

a) $96.103 = 0.96103 \times 10^{+2}$

b) $2.96 = 0.296 \times 10^{+1}$

c) $2.96 = 29.6 \times 10^{-1}$

0.96103, 0.296 e 29.6 sono denominati **Mantissa** o **significando**.

10 è la **base**.

+2 e **+1** e **-1** sono denominati **esponente**.

Esempio in base 10

a) $96.103 = 0.96103 \times 10^{+2}$

b) $2.96 = 0.296 \times 10^{+1}$

c) $2.96 = 29.6 \times 10^{-1}$

Osservazione: i numeri b) e c) sono uguali, cambia solo la posizione della virgola, che si dice "flottante", ecco perchè il termine floating point.

Standard IEEE 754 per i calcoli in virgola mobile

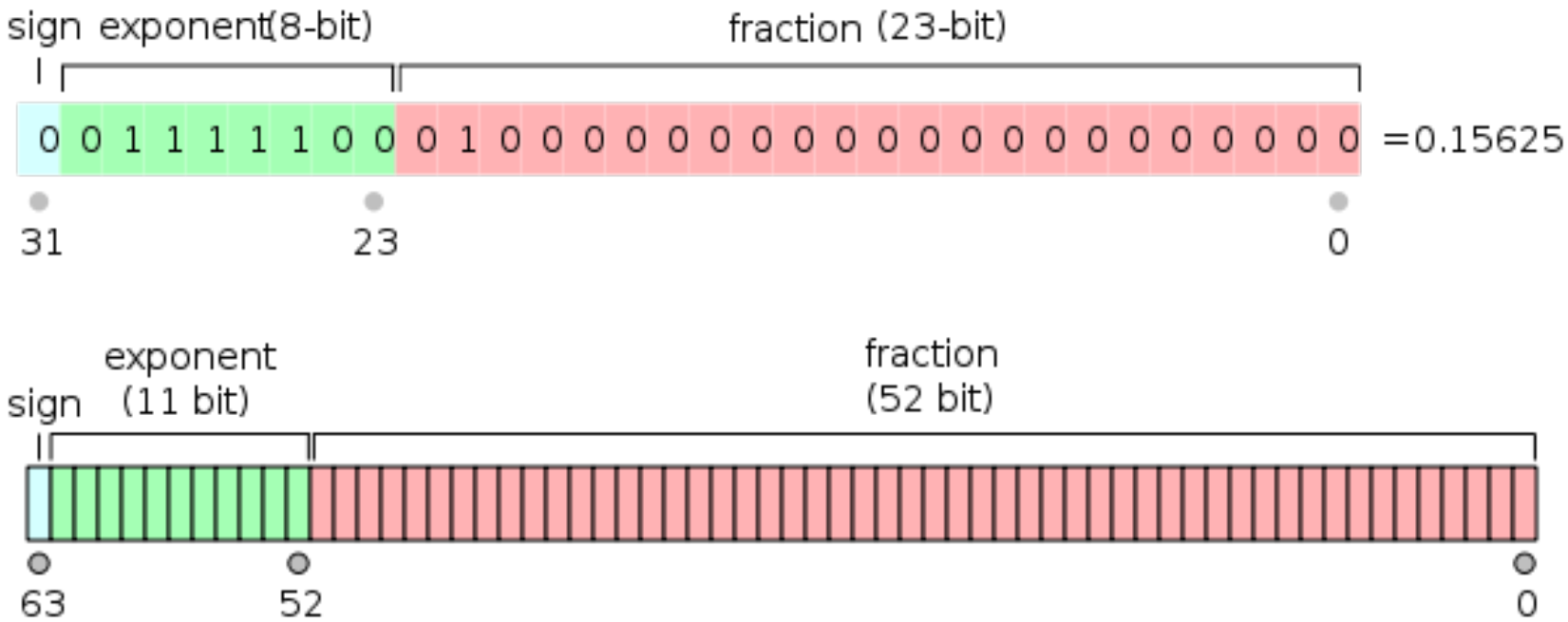
Lo standard IEEE 754 definisce il formato per la rappresentazione dei numeri in virgola mobile:

- 1 bit per la rappresentazione del **segno** (s);
- 8 o 11 bit per la rappresentazione dello **esponente** (E);
- 23 o 52 bit per la rappresentazione del **significando** o **mantissa** (M);

Tali che:

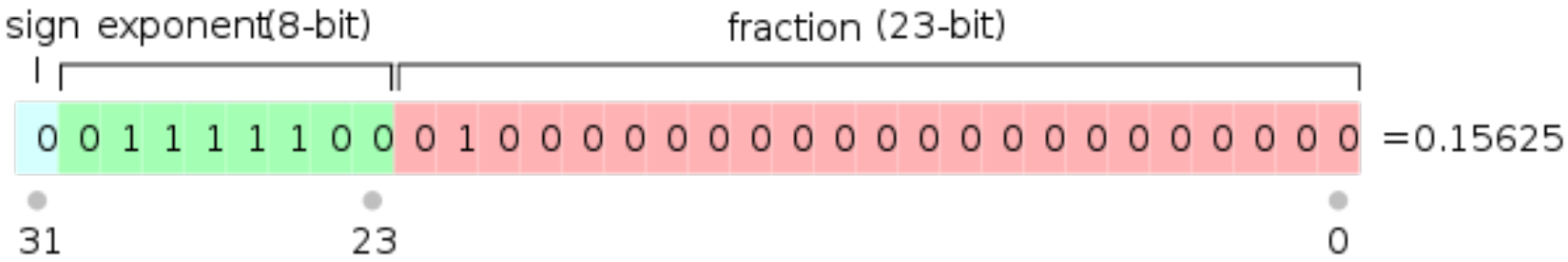
$$N = (-\mathbf{1})^{\mathbf{s}} \times \mathbf{2}^{\mathbf{E}} \times \mathbf{M}$$

IEEE 754



NB: “fraction” è la mantissa o significando.

Standard IEEE 754: Calcolo e rappresentazione della mantissa.



Mantissa o significando rappresentati in forma **normalizzata**:

- **si moltiplica o si divide** la codifica binaria della mantissa **per una certa potenza di 2**.
- In tal modo rappresentazione della mantissa rimarrà solo una cifra prima della virgola, cioè 1.
- Inoltre, dato che **la cifra prima della virgola** è sempre 1, questa **non viene rappresentata**, risparmiando cosi 1 bit.

Standard IEEE 754: Calcolo e rappresentazione della mantissa.

Esempio

Calcolo della mantissa o significando in formato **IEEE 754** a singola precisione (23 bit) per il numero $-\mathbf{113.25_{10}}$

1. Si calcola la **codifica** in base 2 del **valore assoluto** del numero: $113.25_{10} = 1110001.01_2$.
2. Per **normalizzare**, spostiamo la virgola di 6 posti : $1110001.01 = 1.11000101 \times 2^{+6}$.

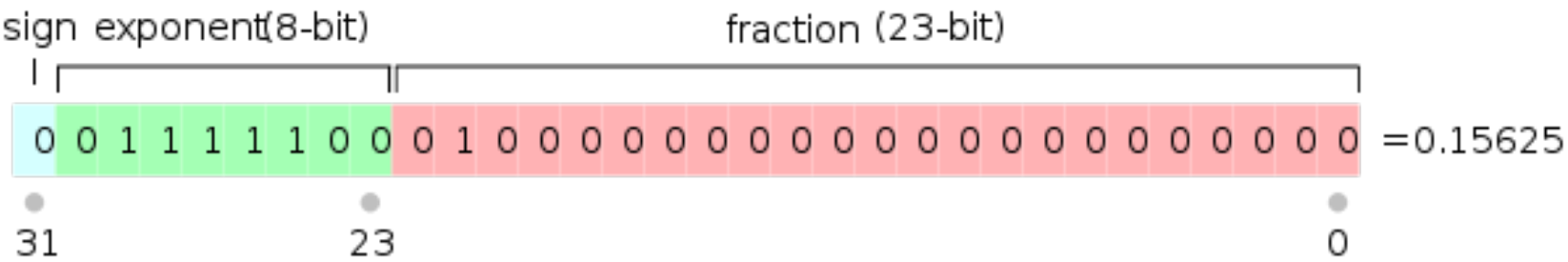
Standard IEEE 754: Calcolo e rappresentazione della mantissa.

Esempio

Calcolo della mantissa o significando in formato **IEEE 754** a singola precisione (23 bit) per il numero $-\mathbf{113.25_{10}}$

3. Ricordando che **la cifra alla sinistra delle virgola non si rappresenta**, la mantissa sarà costituita dai segg. 23 bit:
   - Bit **1-8** (dal piu significativo): 11000101
   - Bit **9-23**: (fino al bit meno significativo): 00000000000000.
4. Quindi sarà $M = 11000101000000000000000$

Standard IEEE 754: Calcolo e rappresentazione dello esponente

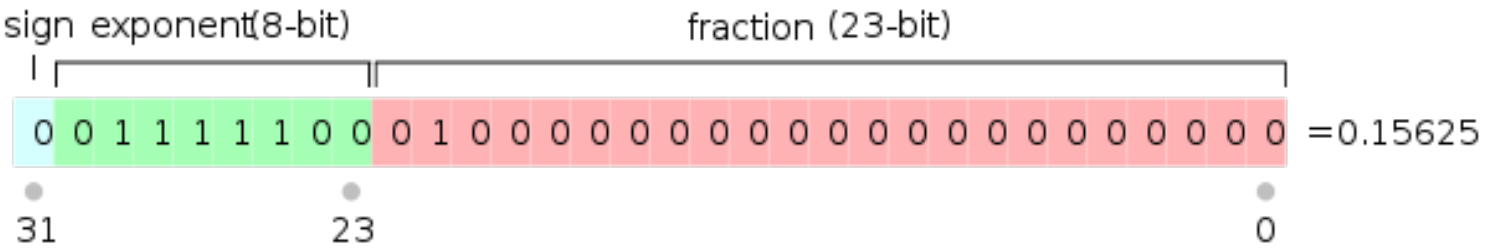


Con 8 bit, in teoria si avrebbero 256 combinazioni.

In pratica:

- I valori 0 e 255 sono **riservati per usi speciali** (se ne parlerà dopo).
- In pratica **si rappresentano 254 valori**, da -126 a $+127$.

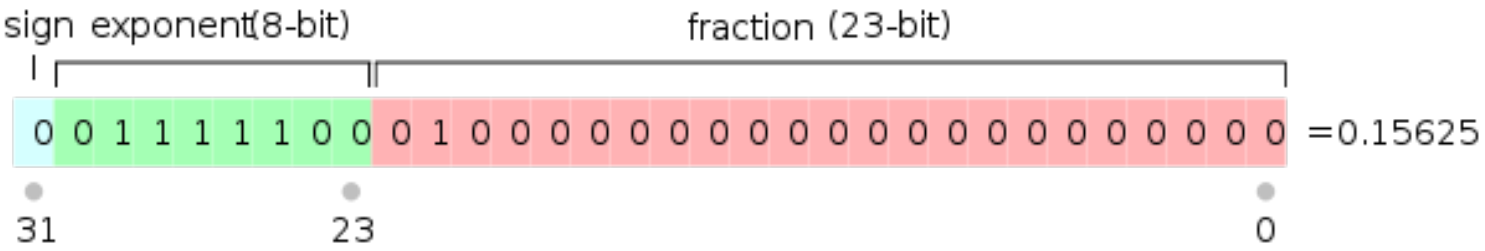
Standard IEEE 754: Calcolo e rappresentazione dello esponente



Infine, il valore dello esponente si rappresenta a meno di un valore k detto **bias**:

- $\mathbf{E} = \mathbf{e} + \mathbf{k}$.
- **E** è il valore **effettivamente rappresentato** nel campo esponente.
- **k** è il **bias**.
- **e** è il valore **esponente** ottenuto durante il **calcolo della mantissa**.

Standard IEEE 754: Calcolo e rappresentazione dello esponente



Nel caso dei floating point a 32 bit si ha $\mathbf{k} = +\mathbf{127}$,

Dunque se $-126 \leq e \leq +127$ ed $\mathbf{E} = \mathbf{e} + \mathbf{k}$ allora $\mathbf{1} \leq \mathbf{E} \leq \mathbf{254}$

In tal modo:

- (+) Le codifiche 0 e 255 si possono riservare per usi speciali.
- (+) Non si è costretti a rappresentare il segno per il campo esponente (in quanto E non ha segno). Rappresentare il segno darebbe problemi nel confronto tra numeri.

Standard IEEE 754: Calcolo e rappresentazione dello esponente

Nello ESEMPIO precedente era

$$113.25_{10} = 1110001.01_2 = \mathbf{1.11000101} \times \mathbf{2^6}.$$

Quindi $\mathbf{e = 6}$.

Allora $\mathbf{E = 6 + k = 6 + 127 = 133 = 10100001}$.

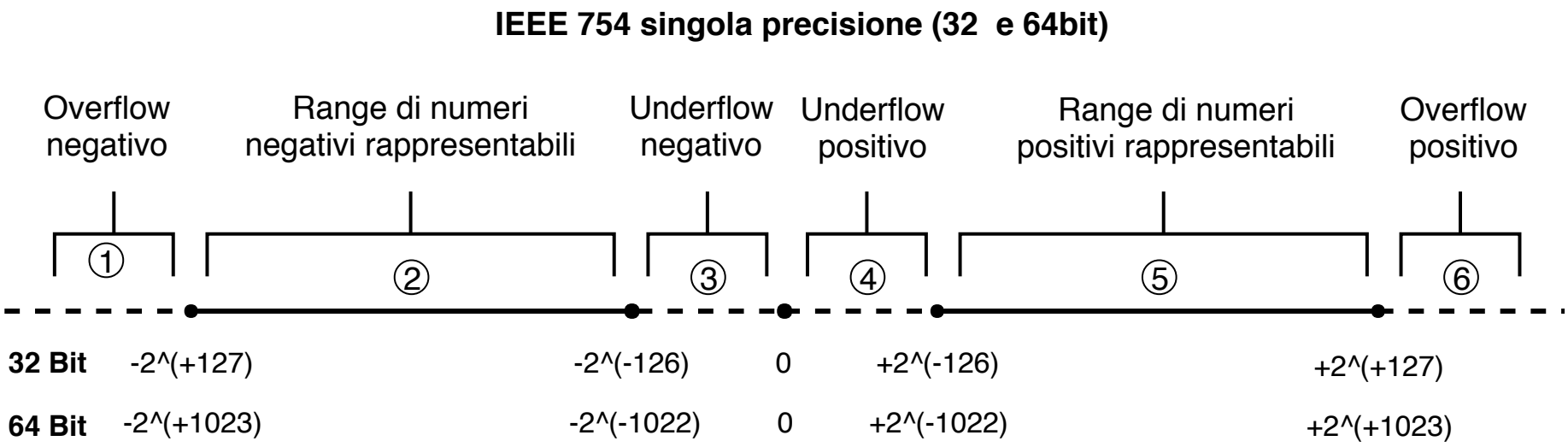
Infine, il **segno**: dato che il numero -113.25 è negativo, **s=1**.

Quindi la codifica a 32 bit floating point IEEE 754 del numero -113.25 è la seguente:

s	E	M
1	**10100001**	**11000101000000000000000**

Floating point IEEE 754: intervalli numerici

NB: Gli intervalli 2 e 5 sono costituiti da **un numero finito di elementi** che costituiscono un sottoinsieme di $\mathbb{R}$.

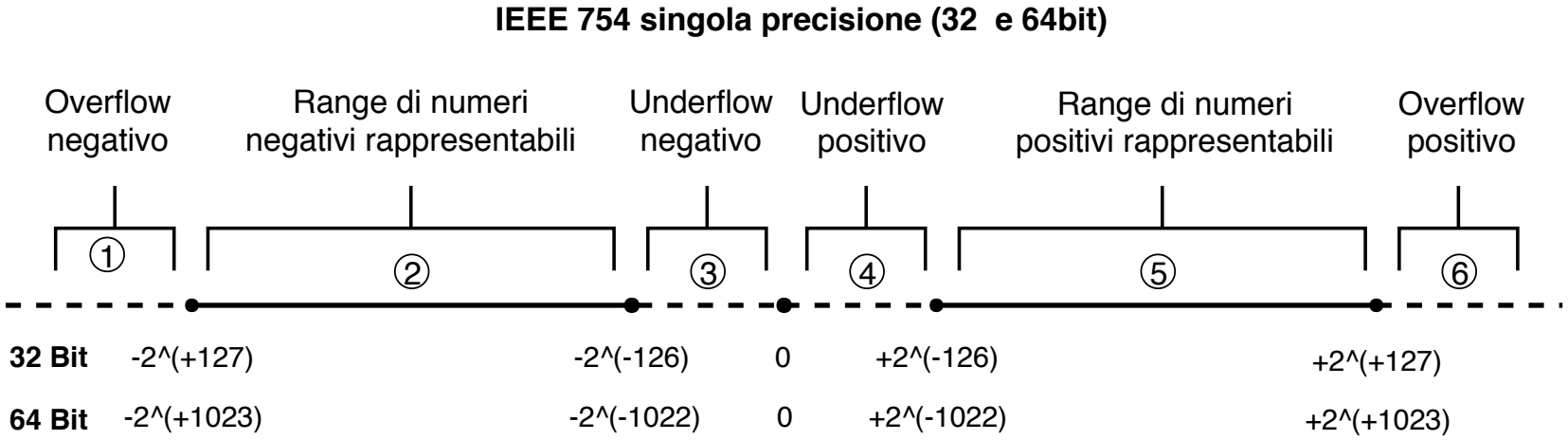


Appprossimazioni

1) **Valore non rappresentabile con un numero finito di cifre**, la sua rappresentazione sarà **troncata**.

ESEMPIO:$4.35_{10} = 100.010\overline{1100}$.

Il numero 4.35 ricade all'interno del range dell'intervallo 5, ma non ne fa parte (non è rappresentabile senza approssimazione).



Appprossimazioni

2) Valore con un **numero di cifre significative** maggiore del numero massimo rappresentabile nel campo mantissa (o significando).

Esempio

Si consideri il numero **9876543.25**.
Codifica floating point IEEE 754 a singola precisione (32 bit):

$$\begin{aligned} \mathbf{9876543.25_{10}} &= 100101101011010000111111.01 = \\ &= \mathbf{1.00101101011010000111111101} \times 2^{23}. \end{aligned}$$

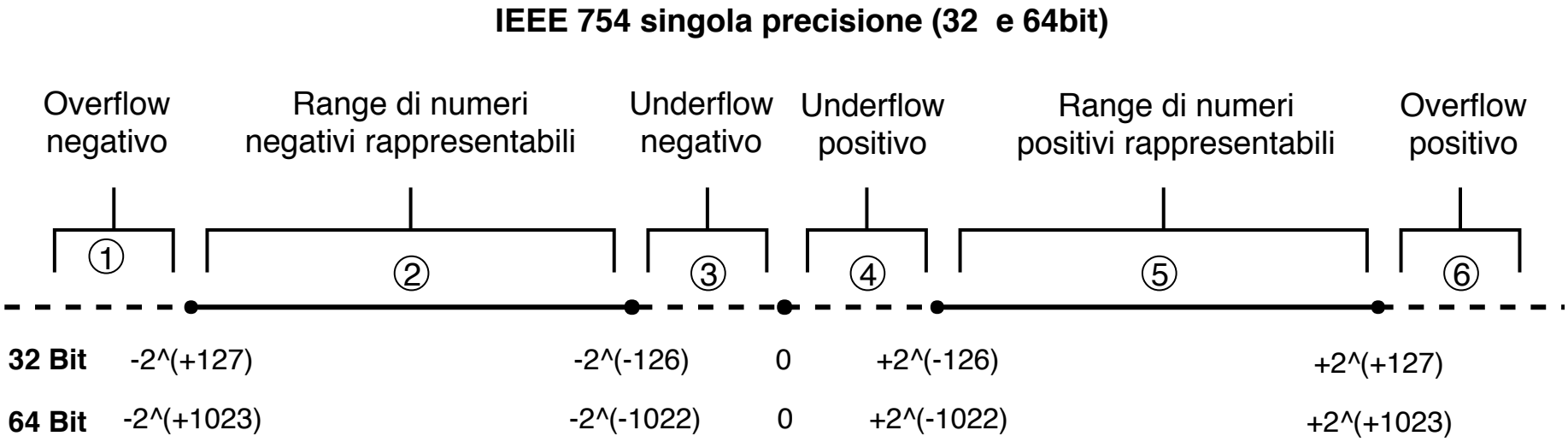
NB: Mantissa (o significando) di lunghezza **25**$>$ 23

Ma per **floating point 32 bit** lunghezza massima mantissa **23 bit!**.

$\Longrightarrow$ il valore sarà memorizzato in modo approssimato!

Appprossimazioni

Osservazione: Il numero **9876543.25** rientra, in valore assoluto, nel range dell'intervallo 5 di IEEE 754 a singola precisione.



Appprossimazioni

Precisione di una rappresentazione in virgola mobile

In generale una certa rappresentazione floating point è caratterizzata da **una precisione p**.

La precisione p è costituita dal numero di **cifre significative** che è possibile rappresentare in quel determinato formato.

Esempio

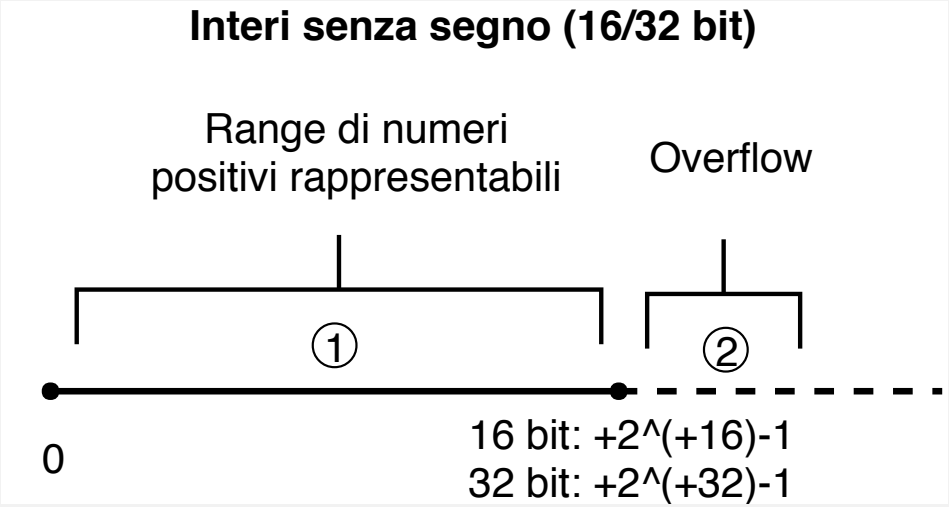
Si consideri il numero $\mathbf{1234.030405887000}_{10}$.

Le cifre significative sono costituite dalla sequenza **1234030405887**

Overflow

Overflow: Il risultato numerico di una certa espressione aritmetica è **maggiore**, in valore assoluto, del valore massimo rappresentabile.

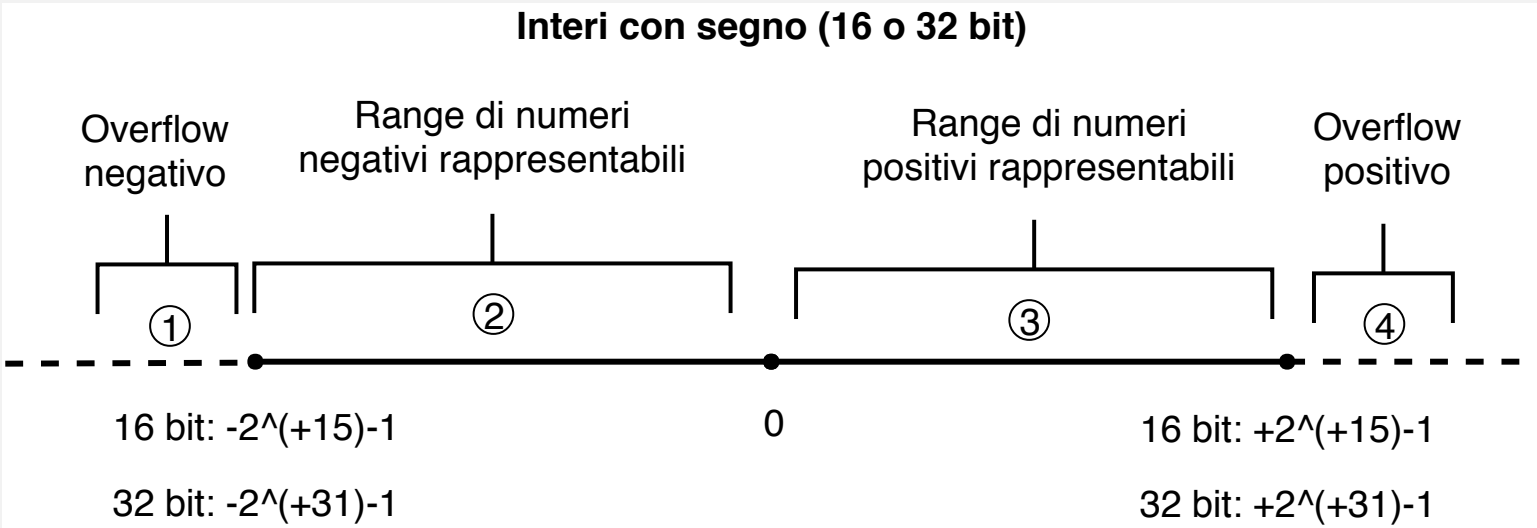
Overflow per interi senza segno



Overflow

Overflow: Il risultato numerico di una certa espressione aritmetica è **maggiore**, in valore assoluto, del valore massimo rappresentabile.

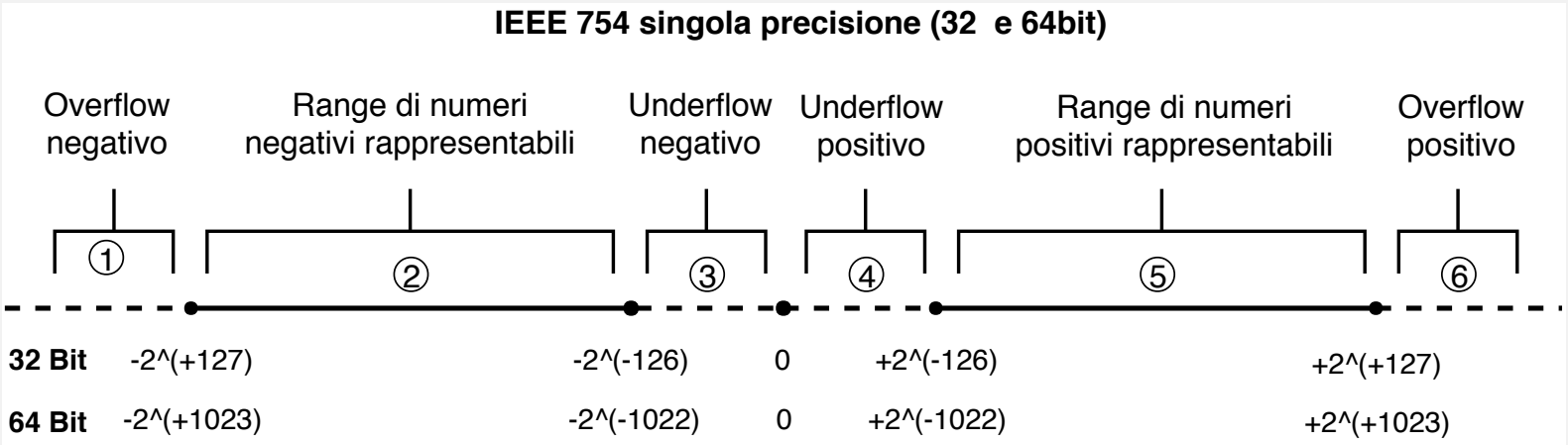
Overflow per interi con segno



Overflow

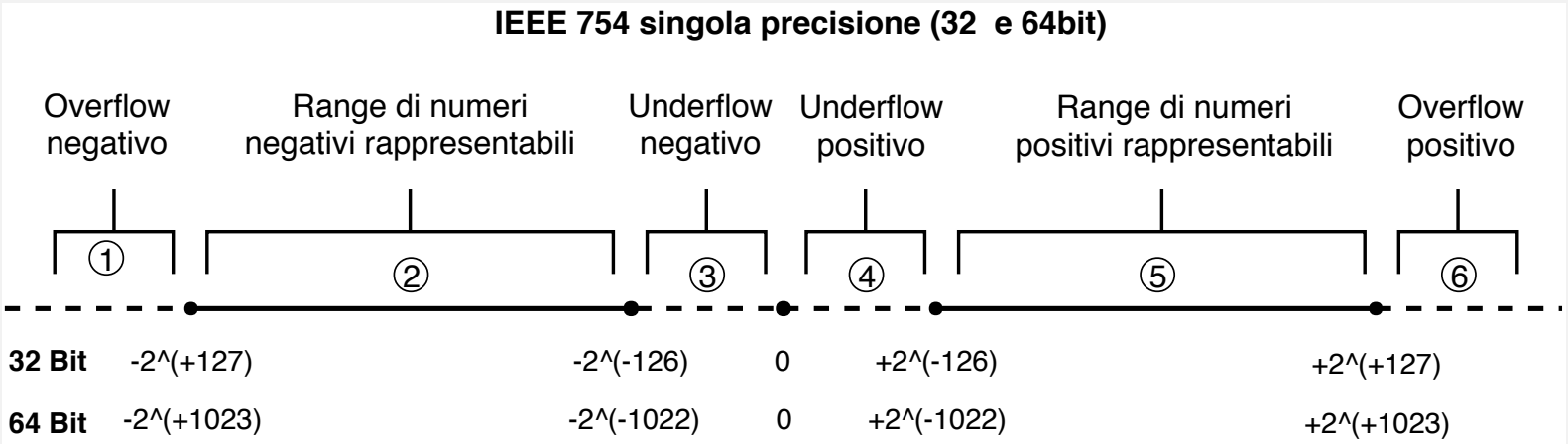
Overflow: Il risultato numerico di una certa espressione aritmetica è **maggiore**, in valore assoluto, del valore massimo rappresentabile.

Overflow per numeri floating point



Underflow

Underflow (Floating point): Il risultato di una operazione tra valori di un certo tipo è **minore**, in valore assoluto, del più piccolo valore rappresentabile.



IEEE 754: valori speciali

Lo standard IEEE 754 riserva alcune combinazioni di bit per la rappresentazione di alcuni **valori speciali**.

Valori Speciali IEEE 754

$\pm$**Inf**: Divisione di numeri in virgola mobile per zero oppure casi di **overflow (positivo o negativo)**: $\pm\frac{x}{0}$.

NaN Forme indeterminate (risultato indefinito) : $\frac{0}{0}, \frac{\pm Inf}{\pm Inf}$, oppure ancora $+Inf - Inf$, e cosi via.

$\pm$**0** I casi di underflow daranno come risultato **zero con segno** a seconda che il segno della espressione sia positivo o negativo. Una certa combinazione di bit permette di rappresentare entrambi gli zeri.

Riassumendo..

Rappresentazione dei numeri nei calcolatori

Per rappresentare i numeri nei **calcolatori** si usa la **rappresentazione base 2**.

Questa differisce a seconda che si debbano rappresentare **numeri interi** ($n \in \mathbb{Z}$) o numeri **decimali (reali)** ($n \in \mathbb{R}$).

Riassumendo..

Rappresentazione binaria di numeri interi

- Un bit (opzionale) per il **segno**
- **Tutti i numeri all'interno del range specificato** dalla rappresentazione **sono rappresentabili** (NO approssimazione, NO underflow).
- Possibile **OVERFLOW** (negativo o positivo)
- Al fine di **non sprecare spazio in memoria** e **non incorrere in overflow**, è importante scegliere una opportuna rappresentazione
  1. numero di bit (ES: 16 o 32);
  2. eventuale presenza del bit per il segno.

Riassumendo..

Numeri floating point (virgola mobile)

- Standard IEEE 754 per singola precisione (32 bit) o doppia precisione (64 bit).
- Codifica/rappresentazione *compatta* mediante **significando** (o mantissa), **segno** ed **esponente**.
- La **precisione** p di una codifica floating point è rappresentata dal **numero di cifre significative** che è possibile rappresentare nella mantissa o significando.
  - 6 cifre per la precisione singola
  - 15 cifre per la precisione doppia

Riassumendo..

Numeri floating point (virgola mobile)

- **Errori di approssimazione** quando
  - il numero non è rappresentabile con numero finito di cifre.
  - il numero di cifre significative del numero da rappresentare è maggiore della precisione p
- **Underflow** quando il numero da rappresentare, in valore assoluto, è troppo piccolo per essere rappresentato.
- **Overflow**: il valor assoluto del numero è maggiore, del numero piu grande (in valore assoluto) rappresentabile con quella codifica.