Programmazione e Architetture (Modulo B)

Lezione 2
Rappresentazione dei numeri

Luca Manzoni

Rappresentare l'informazione

Quando si hanno solo due simboli

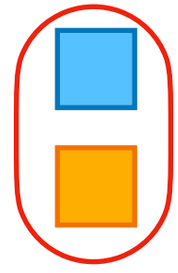
- Per elaborare l'informazione il computer utilizza solo due simboli, che indichiamo con 0 e 1

- Questo significa che **ogni** informazione deve essere codificata usando solo sequenze di questi simboli:
  - Numeri interi
  - Numeri reali
  - Caratteri
  - Istruzioni
  - …

Bits and Bytes

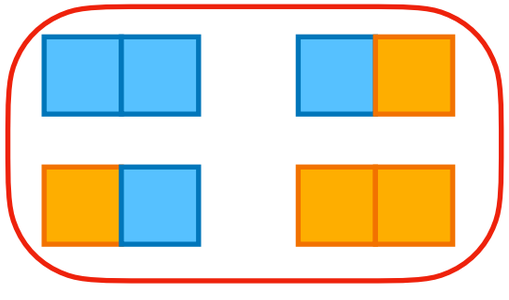
Dal singolo valore binario a 8 bit

2 sequenze di 1 bit

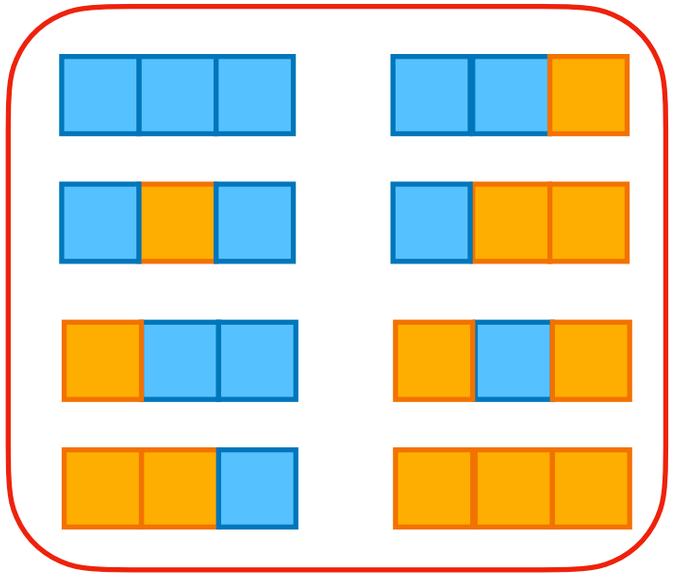


Il bit rappresenta una singola cifra binaria.
Ovvero può assumere solo due valori, 0 e 1

4 sequenze di 2 bit



8 sequenze di 3 bit



Una sequenza di 8 bit è detta **byte**

Quante diverse sequenze di 8 bit esistono?

Ogni posizione può assumere 2 valori, vi sono 8 posizioni,
quindi $\underbrace{2 \times 2 \times \ldots \times 2}_{\text{8 volte}} = 2^8 = 256$ possibili sequenze diverse

In generale abbiamo 2^n sequenze distinte di n bit

Kilo, Mega, Giga, Tera, …

Multipli in base 2 e in base 10

- Esiste una possibilità di fraintendimenti quando si parla di Kilobyte, Megabyte, etc
- Tradizionalmente, dato che $2^{10} = 1024$ è molto vicino a $10^3 = 1000$, il prefisso “Kilo” indicava 1024 byte, “Mega” $1024^2 = 2^{20}$ byte, etc
- In altri casi il prefisso “Kilo” indicava 1000 byte, “Mega” $1000^2 = 10^6$ byte, etc
- Per chiarire se si deve moltiplicare per 1000 (“base 10”) o 1024 (“base 2”) esistono nomi diversi…
- …ma non tutti li usano (e.g., la memoria di solito è in “base 2”, lo storage in “base 10”)

Kilo, Mega, Giga, Tera, …

Multipli in base 2 e in base 10

Nome	Sigla	Dimensione
Kilobyte	KB	1000 byte
Kibibyte	KiB	1024 byte
Megabyte	MB	1000 KB
Mebibyte	MiB	1024 KiB
Gigabyte	GB	1000 MB
Gibybyte	GiB	1024 MB
Terabyte	TB	1000 GB
Tebibyte	TiB	1024 GiB
Petabyte	PB	1000 TB
Pebibyte	PiB	1024 TiB

Numeri in base 2

Rappresentazione in base 10

Capire come scriviamo i numeri

Proviamo a vedere come interpretiamo un numero:

2508

$$2 \times 1000 + 5 \times 100 + 0 \times 10 + 8 \times 1$$

$$2 \times 10^3 + 5 \times 10^2 + 0 \times 10^1 + 8 \times 10^0$$

Dato che usiamo una notazione posizionale a ogni cifra corrisponde un valore che dipende dalla posizione in cui si trova

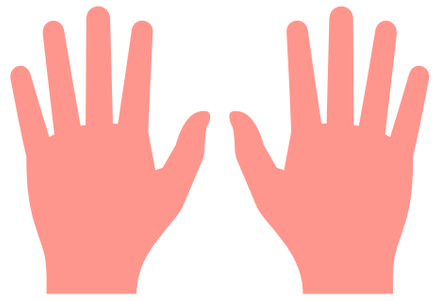
Notiamo una certa struttura nei valori che moltiplicano le singole cifre… sono tutte potenze di 10

In generale il valore denotato da una sequenza $\langle x_1 x_2 \ldots x_n \rangle_{10}$ di n interpretato in base 10 è dato da:

$$\langle x_1 x_2 \ldots x_n \rangle_{10} = \sum_{i=1}^{n} x_i 10^{n-i}$$

Rappresentazione in altre basi

Capire come scriviamo i numeri

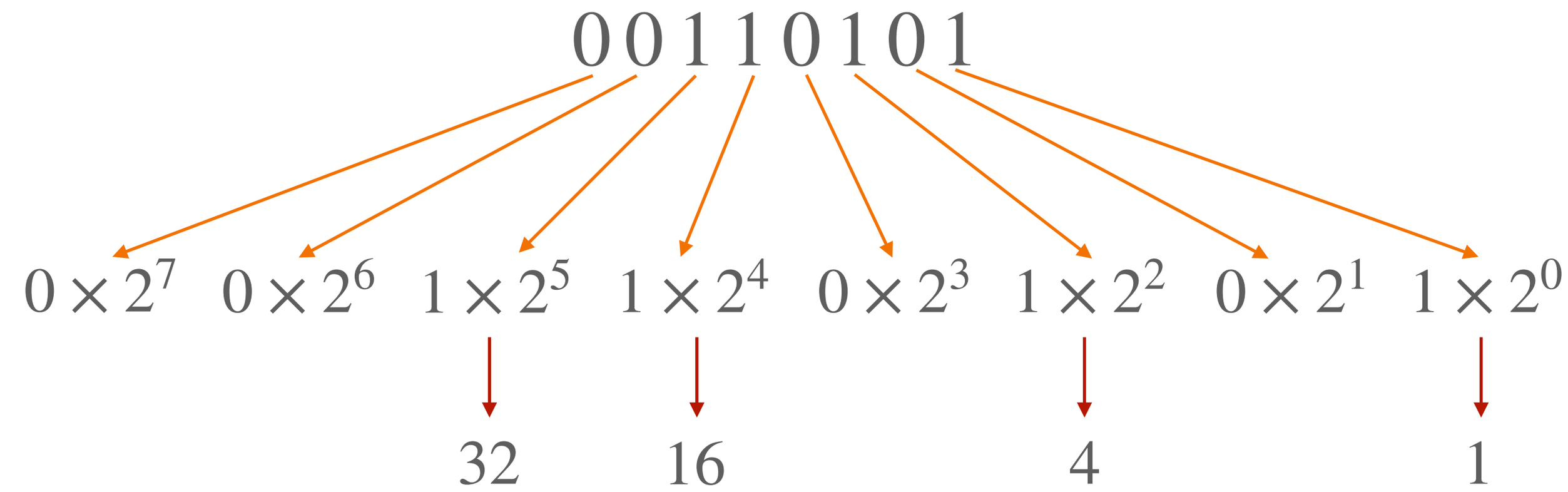


- Usiamo la base 10 perché abbiamo 10 cifre da 0 a 9
- Qualcosa ci limita ad usare esattamente 10 cifre?
- No, possiamo usare un qualunque numero di cifre:
  - Due: $0,1$
  - Otto: $0,1,2,3,4,5,6,7$
  - Sedici: $0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F$
- In generale in base b una sequenza di n cifre viene interpretata come

$$\langle x_1 x_2 \ldots x_n \rangle_b = \sum_{i=1}^{n} x_i b^{n-i}$$

Rappresentazione in base 2

Usare solo due cifre



$$32 + 16 + 4 + 1 = 53$$

Conversione binario-decimale

Alcuni esercizi

$0000\,0000$	0
$0000\,0100$	$2^2 = 4$
$0000\,0011$	$2^1 + 2^0 = 3$
$1111\,1111$	$2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 255$

Potenze di 2

$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$

Ottale e esadecimale

Usare 8 o 16 cifre

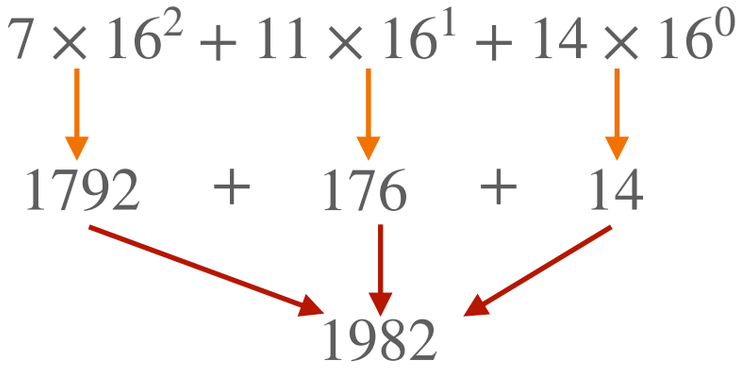
In aggiunta ad usare la base 2 altre basi che appaiono sono base 8 (o “ottale”) e base 16 (o “esadecimale”)

Per la base 8 usiamo le cifre $0,1,2,3,4,5,6,7$

Per la base 16 ci servono cifre oltre il 9 e, per convenzione, si usano le lettere dell’alfabeto: $0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F$

A sarà 10 in base 10, B sarà 11 in base 10, fino a F che sarà 15 in base 10

$7BE$ in esadecimale è



Somma di numeri in binario

Come in base 10, ma con due cifre

L'algoritmo per la somma di numeri in binario (o in qualsiasi base) rimane lo stesso che si usa per i numeri in base 10:

$$\begin{array}{r} {\scriptstyle 1\ 1\ \ \ \ \ \ \ \ } \\ 1101\,1001 + \\ \hline 0010\,0011 = \\ \hline 1111\,1100 \end{array}$$

$1_2 + 1_2 = 10_2$

Quindi abbiamo 0 col riporto di 1

Similmente è possibile definire la moltiplicazione di numeri in binario, sapendo che: $0 \times 0 = 0$, $0 \times 1 = 0$, $1 \times 0 = 0$, e $1 \times 1 = 1$

Overflow

Utilizzo di un numero finito di bit

- Solitamente in un calcolatore i numeri non sono rappresentati con un numero non limitato di bit ma con un numero *fissato* di bit
- Esempi: 16, 32, 64 bit
- Cosa succede quando una somma ci porterebbe oltre il massimo numero rappresentabile?
- Per esempio $1111\,1111 + 0000\,0001$

Overflow

Utilizzo di un numero finito di bit

$$\begin{array}{r} 1111\,1111+ \\ 0000\,0001\,= \\ \hline 1\,0000\,0000 \end{array}$$

I primi otto bit sono quelli che teniamo
e abbiamo ottenuto 0

Otteniamo quindi un **overflow** (“straripamento”), dato che il numero di bit che ci servirebbero è superiore a quello che abbiamo

In generale per i numeri senza segno di n bit possiamo rappresentare solo valori tra 0 e $2^n - 1$ e ogni operazione che porta a risultati fuori dal range avrà un risultato errato

Rappresentazione in complemento a due

Numeri negativi

E come rappresentarli

- Generalmente per rappresentare un numero negativo utilizziamo il segno “—”
- Questo però non è disponibile quando dobbiamo rappresentare tutto come sequenza 0 e 1
- Vedremo diversi approcci, ognuno con vantaggi e svantaggi:
  - Segno e modulo
  - Complemento a 1
  - Complemento a 2
  - Eccesso N

Segno e modulo

Un approccio naive

- Un primo approccio è quello di utilizzare il primo bit di un numero come bit di segno (0 indica un numero positivo e 1 un numero negativo)

- Per esempio $1000\,1011$ viene interpretato come:

  - 1 in prima posizione indica che il segno è meno

  - $000\,1011$ viene interpretato come
    $2^3 + 2^1 + 2^0 = 8 + 2 + 1 = 11$

  - Quindi $1000\,1011$ viene interpretato come -11

Segno e modulo

I problemi di questo approccio

- Con n bit possiamo rappresentare i valori tra $-2^{n-1} - 1$ e $2^{n-1} - 1$
- L'approccio con segno e modulo ha diversi problemi
  - Esistono **due** rappresentazioni del valore 0, una con segno meno e una con segno più:
    - $1000\,0000 = -0$ e $0000\,0000 = +0$
  - La somma tra numeri positivi e numeri negativi **non** funziona con la stessa procedura che usiamo per numeri senza segno

Complemento a uno

Un possibile miglioramento

- Un secondo approccio è quello del **complemento a uno**
- Dato un numero positivo (con il primo bit 0) che rappresenta il numero k, il numero $-k$ viene rappresentato invertendo ogni bit
- Per esempio $0000\,1101$ rappresenta il numero 13, per rappresentare -13 invertiamo tutti i bit: $1111\,0010$
- Il primo bit rappresenta comunque il segno
- Abbiamo ancora due rappresentazioni per 0: $0000\,0000$ e $1111\,1111$

Complemento a due

La rappresentazione più usata

- Nel complemento a due il bit più significativo in una sequenza di n bit viene interpretato non come moltiplicante 2^{n-1} ma come moltiplicante -2^{n-1}
- In 8 bit il valore più piccolo è quindi rappresentato da $1000\,0000$, ovvero $-2^7 = -128$
- Sempre in 8 bit il valore più grande è rappresentato da $0111\,1111$, ovvero $2^6 + 2^5 + \ldots + 2^0 = +127$
- In generale usando il complemento a 2 su n bit possiamo rappresentare tutti i numeri da -2^{n-1} a $2^{n-1} - 1$

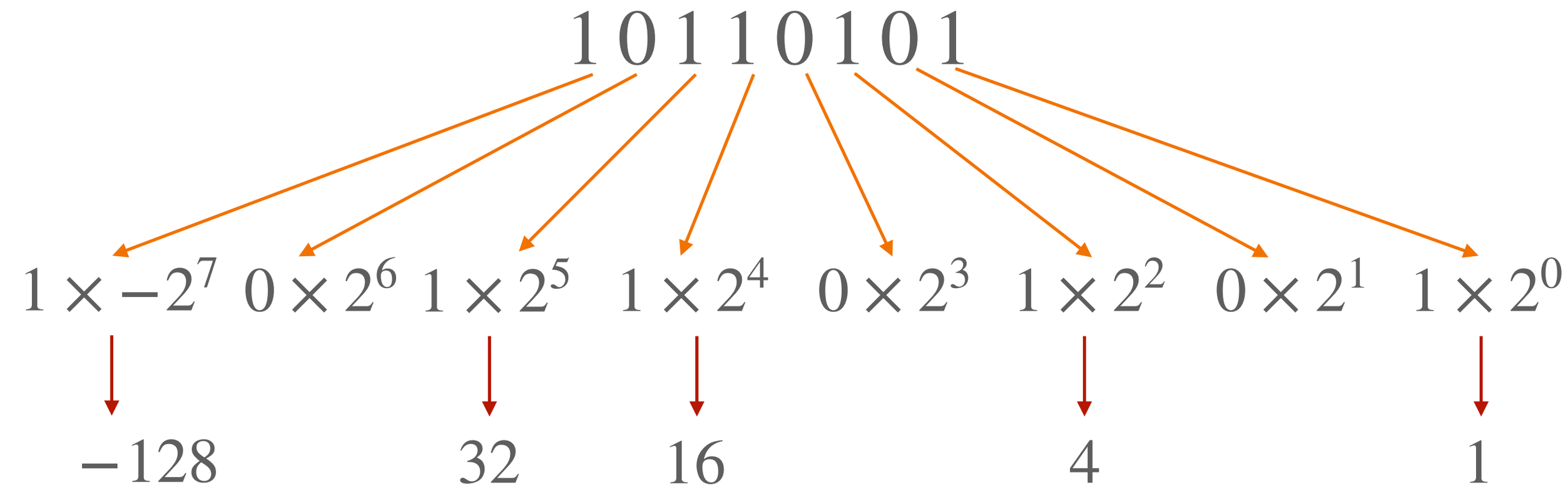
Complemento a due

La rappresentazione più usata

- Abbiamo che 0 ha una **unica** rappresentazione come $0000\,0000$
- Il primo bit ci indica comunque il segno
- Abbiamo altri vantaggi:
  - I numeri tra 0 e $2^{n-1} - 1$ non cambiano rappresentazione rispetto ai numeri senza segno
  - Possiamo usare gli stessi circuiti che usiamo per la somma di numeri senza segno

Complemento a 2

Esempio



$$-128 + 32 + 16 + 4 + 1 = -128 + 53 = -75$$

Eccesso N

Cambiare da dove si inizia a contare

- Nella notazione a eccesso N si sceglie un valore N e tutti i numeri sono interpretati come numeri binari positivi che rappresentano uno scostamento dal valore -N
- Per esempio se $N = 128$ avremmo:
  - $0000\,0000$ rappresenta $0 - 128 = -128$
  - $1000\,0000$ rappresenta $128 - 128 = 0$
  - $1111\,1111$ rappresenta $255 - 128 = 127$

Numeri floating point

Standard IEEE 754

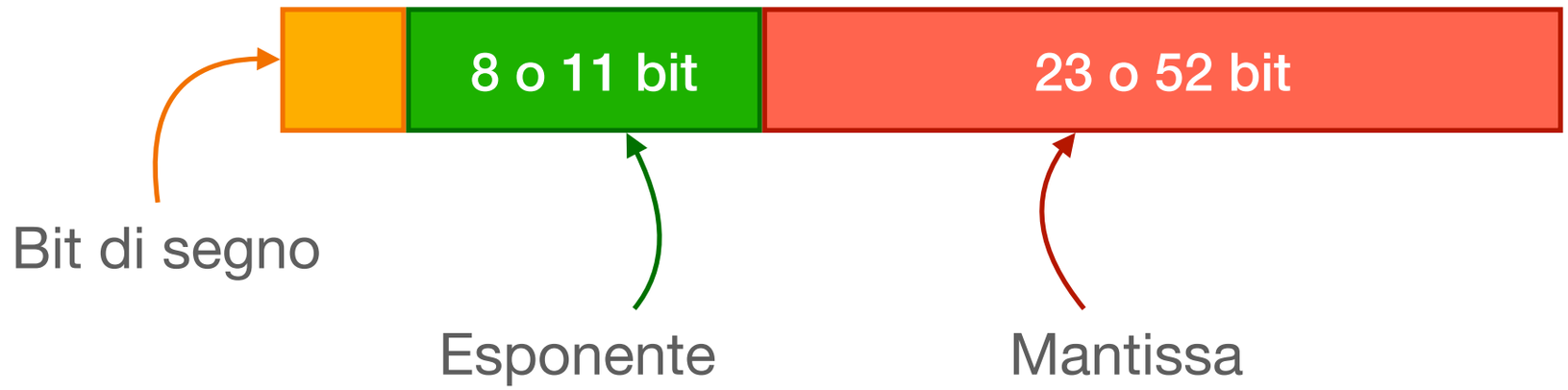
Rappresentazione dei numeri floating point

- In aggiunta ai numeri interi vogliamo poter rappresentare anche numeri reali
- Un computer, anche senza avere limiti di memoria **non** può rappresentare ogni numero reale
- La rappresentazione è necessariamente approssimata
- In generale vogliamo qualcosa di simile ad una notazione scientifica:
  - numero fissato di cifre significative (contenute nella *mantissa*)
  - moltiplicate per una base (10 in notazione scientifica, 2 nei numeri floating point)
  - elevata a un certo *esponente*

Standard IEEE 754

Struttura di un numero floating point

Lo standard IEEE754 prevede numeri a precisione **singola** (32 bit), **doppia** (64 bit) e, meno diffusi, a mezza precisione (16 bit) e quadrupla precisione (128 bit)



Il numero rappresentato è nella forma $\pm \text{mantissa} \times 2^{\text{esponente}}$

Quindi il numero di bit della mantissa ci dice quanta precisione abbiamo

Mentre il numero di bit dell'esponente ci dice quanto grandi (o piccoli) sono i numeri che possiamo rappresentare

Infiniti e NaN

Valori che non sono numeri reali

Lo standard IEEE754 prevede la possibilità che alcune operazioni ritornino dei valori che non rappresentano numeri validi, questo con il valore NaN (not a number)

In aggiunta a questo, lo standard prevede anche la possibilità di rappresentare un valore infinito positivo o negativo ($+\infty$ e $-\infty$)

Per rappresentare valori con una precisione ridotta vi è la possibilità di usare dei numeri **denormalizzati**.

Lo standard IEEE 754 per i numeri floating point è abbastanza complesso.
Vedremo facendo esercizi alcune delle sue peculiarità, ma non andremo troppo nei dettagli

Visualizzazione dei numeri floating point: https://bartaz.github.io/ieee754-visualization/

Problemi dei numeri floating point

Rappresentazioni inesatte

```
>>> 0.1 + 0.2
0.30000000000000004
```

Risultato inesatto dovuto all'approssimazione nella rappresentazione dei numeri floating point

Risultato inesatto dovuto alla limitata precisione dei numeri floating point

```
>>> 10**16 - 0.1 == 10**16
True
```

Non associatività delle operazioni

Come fare?

```
>>> 10**16 - (10**16 + 0.1) == 0
True
>>> (10**16 - 10**16) + 0.1 == 0
False
```

ASCII e Unicode

Rappresentazione di testo

Da ASCII a Unicode

- Per rappresentare il testo si usano comunque sequenze di bit
- Un formato tradizionale è ASCII, che richiede 7 bit per carattere (solitamente con “padding” a 8 bit per usare un intero byte)
- A ogni sequenza di 7 bit è associato un carattere:
  - Alfanumerici: 0,1,2,…, A,B,…,Z, a,b,…z (notare come maiuscole e minuscole siano entrambe presenti)
  - Punteggiatura: ,/,%,{,},…
  - Spazio: “ “
  - Caratteri di controllo

Rappresentazione di testo

Da ASCII a Unicode

- Con 7 bit si possono rappresentare solo 128 diversi caratteri (inclusi i caratteri di controllo)

- Sufficienti per l'inglese ma non per altre lingue (anche solo per i caratteri accentati in italiano)

- Una serie di estensioni per codifica di caratteri erano state proposte (si veda lo standard ISO/IEC 8859) con molte parti (a seconda della lingua che si voleva usare)

- Più recentemente lo standard unicode permette di codificare buona parte dei simboli usati per la scrittura. Ogni carattere è codificato in un numero variabile di byte da 1 a 4