

Architetture Degli Elaboratori

Lezione 1.b

Rappresentazione dei numeri e porte logiche

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
 - ...

Non una novità assoluta

D: quali altri sistemi di comunicazione usano due soli 'simboli'?

Non una novità assoluta

Steganografia:
cifrario di Bacone

A	..	J	..---	S	...	2	..---
B	---	K	--	T	-	3	...--
C	---.	L		U	..-	4	
D	---	M	--	V	...-	5	
E	.	N	--.	W	---	6	
F		O	---	X	---	7	
G	--.	P		Y	---	8	
H	...	Q	---.	Z	---	9	
I	..	R	---	1		0	

A B C D E F
Aaaaa aaaab. aaaba. aaabb. aabaa. aabab.
G H I K L M
aabba aabbb. abaaa. abaab. ababa. ababb.
N O P Q R S
abbaa. abbab. abbbba. abbbb. baaaa. baaab.
T V W X Y Z
baaba. baabb. babaa. babab. babba. babbb.

To encode a message each letter of the **plaintext** is replaced by a group of five of the letters 'A' or 'B'.

Bits and Bytes

Dal singolo valore binario a 8 bit

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



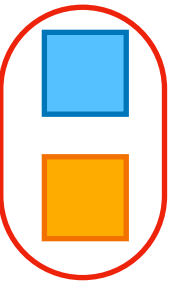
Una sequenza di 8 bit è detta **byte**

D: Quante diverse sequenze di 8 bit esistono?

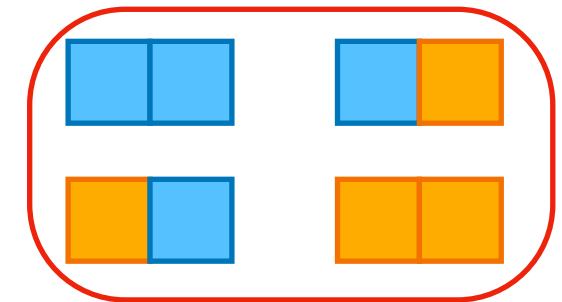
Ogni posizione può assumere 2 valori, vi sono 8 posizioni,
quindi $2 \times 2 \times \dots \times 2 = 2^8 = 256$ possibili sequenze diverse

8 volte

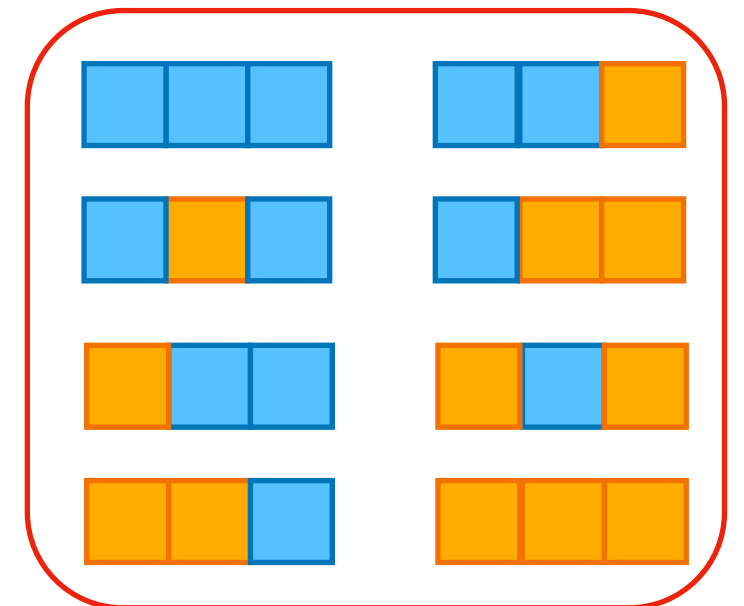
2 sequenze di 1 bit



4 sequenze di 2 bit



8 sequenze di 3 bit



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 $2^{10} = \mathbf{1024}$ è molto vicino a $10^3 = \mathbf{1000}$, 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
Gibibyte	GiB	1024 MB
Terabyte	TB	1000 GB
Tebibyte	TiB	1024 GiB
Petabyte	PB	1000 TB
Pebibyte	PiB	1024 TiB

Numeri senza segno

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

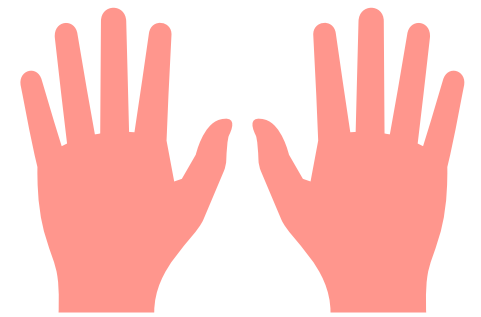
In generale il valore denotato da una sequenza

$\langle x_1 x_2 \dots x_n \rangle_{10}$ di n interpretato in base 10 è dato da: $\langle x_1 x_2 \dots x_n \rangle_{10} = \sum_{i=1}^n x_i 10^{n-i}$

Rappresentazione in altre basi

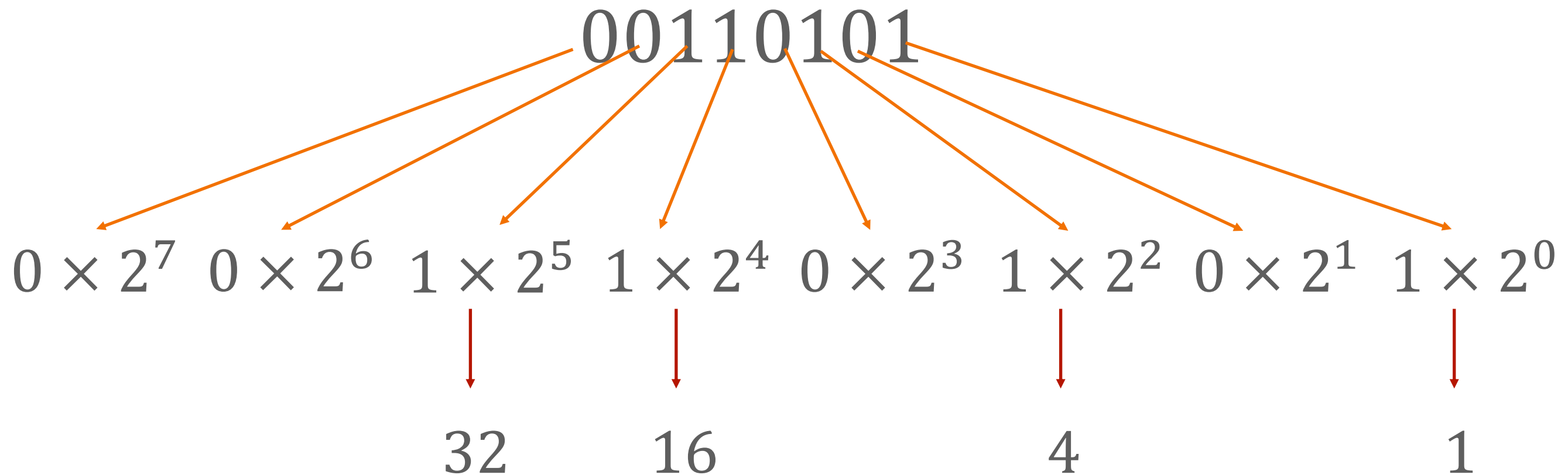
Capire come scriviamo i numeri

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



Rappresentazione in base 2

Usare solo due cifre



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

Conversione binario-decimale

Alcuni esercizi

D: che numeri sono in base 10?

Potenze di 2

$$2^0=1$$

$$2^1=2$$

$$2^2=4$$

$$2^3=8$$

$$2^4=16$$

$$2^5=32$$

$$2^6=64$$

$$2^7=128$$

0000 0000

0

0000 0100

$2^2=4$

0000 0011

$2^1+2^0=3$

1111 1111

$2^8-1=255$

D: che range di valori può contenere un byte che codifica interi positivi?

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

$$\begin{array}{rcccl} 7 \times 16^2 & + & 11 \times 16^1 & + & 14 \times 16^0 \\ \downarrow & & \downarrow & & \downarrow \\ 1792 & + & 176 & + & 14 \\ & & \downarrow & & \swarrow \\ & & 1982 & & \end{array}$$

D: perché base 16?

- $\langle x_1 x_2 x_3 x_4 \rangle_b = \langle y_1 \rangle_h$ facile
- 1 Byte: $\langle x_1 x_2 \rangle_h$

Rappresentazione in altre basi

Capire come scriviamo i numeri

D: i simboli ordinati qui sotto definiscono un sistema base 8?

è, ò, @, #, §, *, \$, -

0, 1, 2, 3, 4, 5, 6, 7

$$\text{òè} = 8^1 \times 1 + 8^0 \times 0 = 8$$

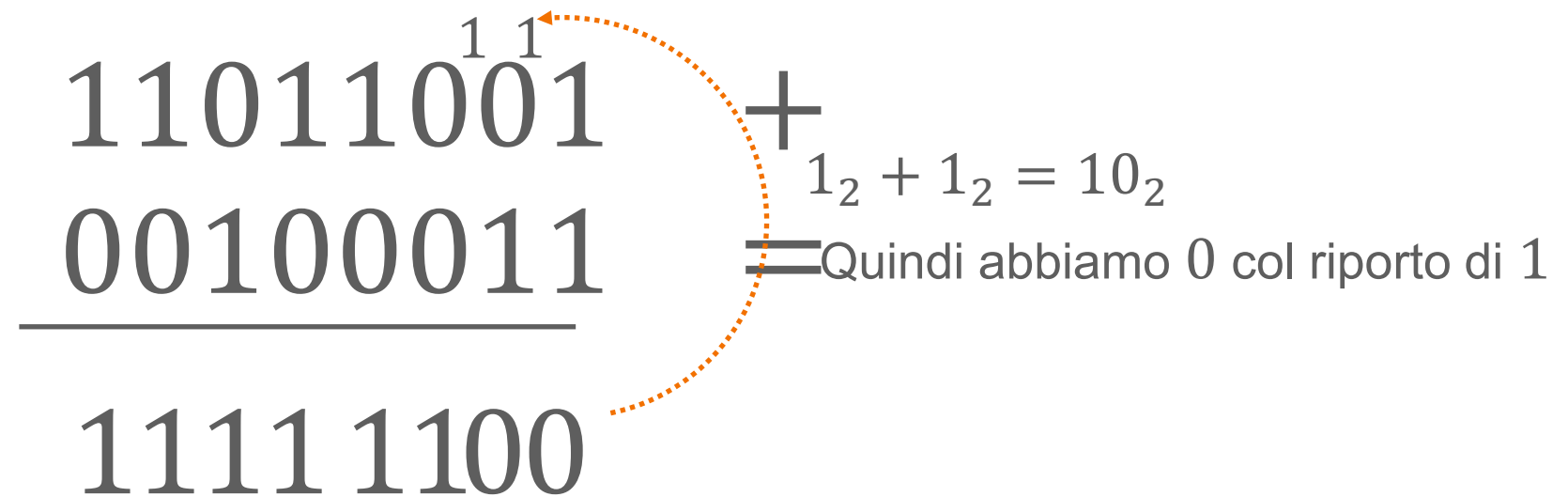
$$\text{@ò} = ?$$

$$\text{@ò} = 8^1 \times 2 + 8^0 \times 1 = 16 + 1 = 17$$

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:



The diagram illustrates the binary addition of 11011001 and 00100011. The numbers are aligned by their least significant bits. A horizontal line is drawn under the second number. The result 11111100 is shown below the line. A dotted orange arrow originates from the 5th bit position (the second column from the right) and points to the 6th bit position, indicating a carry of 1. To the right of the plus sign, the text $1_2 + 1_2 = 10_2$ is shown, followed by an equals sign and the text "Quindi abbiamo 0 col riporto di 1".

$$\begin{array}{r} 11011001 \\ 00100011 \\ \hline 11111100 \end{array}$$

+
 $1_2 + 1_2 = 10_2$
= 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 fissato di bit**
- Esempi: 16, 32, 64 bit
- Cosa succede quando una somma ci porterebbe **oltre il massimo** numero rappresentabile?
- Per esempio $11111111 + 00000001$

Overflow

Utilizzo di un numero finito di bit

$$\begin{array}{r} 11111111 \\ + 00000001 \\ \hline 100000000 \end{array}$$

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

Carry: numeri senza segno – dovremmo “portarci dietro” un 1

Overflow: numeri con segno – straripiamo nel bit del segno

Numeri con segno

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 naïve

- 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 10001011 viene **interpretato** come:
 - 1 in prima posizione indica che il segno è meno
 - 0001011 viene interpretato come $2^3 + 2^1 + 2^0 = 8 + 2 + 1 = 11$
 - Quindi 10001011 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 $2^{n-1} - 1 \rightarrow$ stiamo sprecando 1 bit. **D**: dove?
- Esistono **due** rappresentazioni del valore 0, una con segno meno e una con segno più:
 - $10000000 = -0$ e $00000000 = +0$
- La somma tra numeri con segno **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 00001101 rappresenta il numero 13, per rappresentare -13 invertiamo tutti i bit: 11110010
- Il primo bit rappresenta comunque il segno
- Unica procedura (circuito per somma)

ma aggiustamento per **end-around carry**

- Abbiamo ancora due rappresentazioni per 0:
00000000 e 11111111

Complemento a uno

End-around carry

Il carry va ri-aggiunto al bit meno significativo

Numeri a 4 bit

+3: 0011

-2: 1101

10000 – ri-aggiungo al bit meno significativo

0001 – corretto!

Complemento a due

Da base2 a c2

Codifica:

Se positivo, ok.

Se negativo [tranne -128]:

1. Invertire tutti i bit
2. Sommare 1

Esempio:

5 : 0000 0101

-5 : 1111 1011

4 : 0000 0100

-4 : 1111 1100

Somma :

5 : 0000 0101

-5 : 1111 1011

5-5 : 0000 0000

Somma :

5 : 0000 0101

-4 : 1111 1100

5-4 : 0000 0001

Complemento a due

Da c2 a base10

- 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}** ; il resto è uguale.
- In 8 bit il valore più piccolo è quindi rappresentato da 10000000, ovvero $-2^7 = -128$
- Sempre in 8 bit il valore più grande è rappresentato da 01111111, ovvero $2^6 + 2^5 + \dots + 2^0 = +127$
- In generale usando il complemento a 2 su n bit possiamo rappresentare tutti i numeri da **-2^{n-1} a $2^{n-1} - 1$**

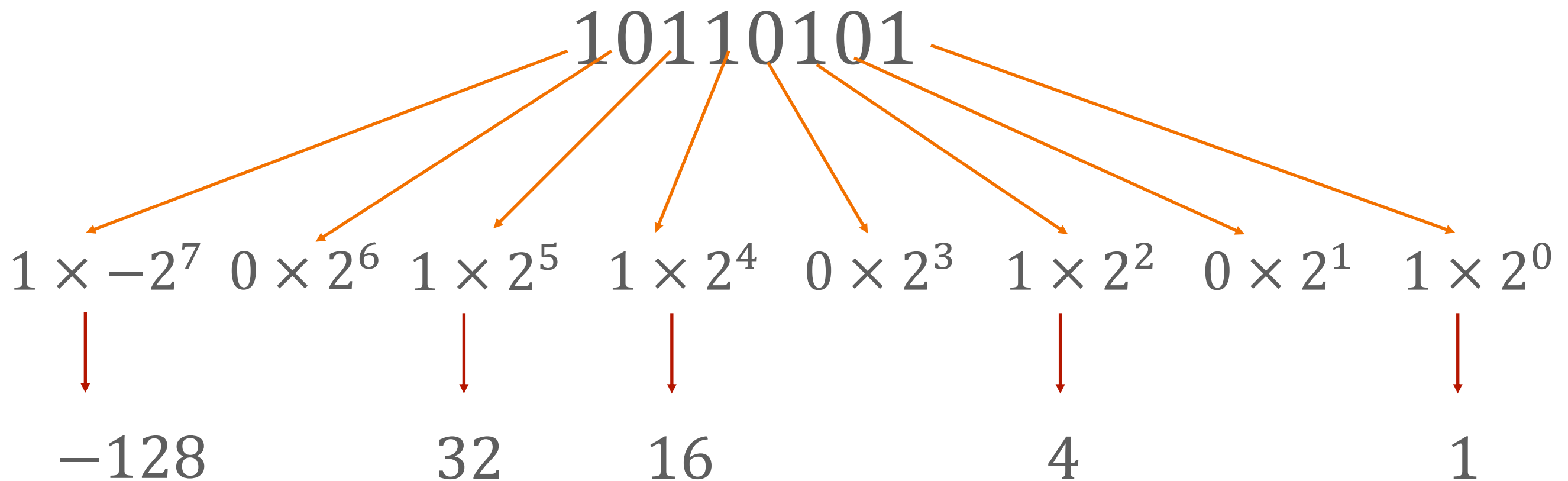
Complemento a due

La rappresentazione più usata

- Abbiamo che **0** ha una unica rappresentazione come 00000000
- 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:
 - 00000000 rappresenta $0 - 128 = -128$
 - 10000000 rappresenta $128 - 128 = 0$
 - 11111111 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
- **D**: un computer può rappresentare ogni numero reale nell'intervallo $(0,1)$?
- Un computer, anche senza avere limiti di memoria **non** può rappresentare ogni numero reale
- La **rappresentazione** è necessariamente **approssimata**

Standard IEEE 754

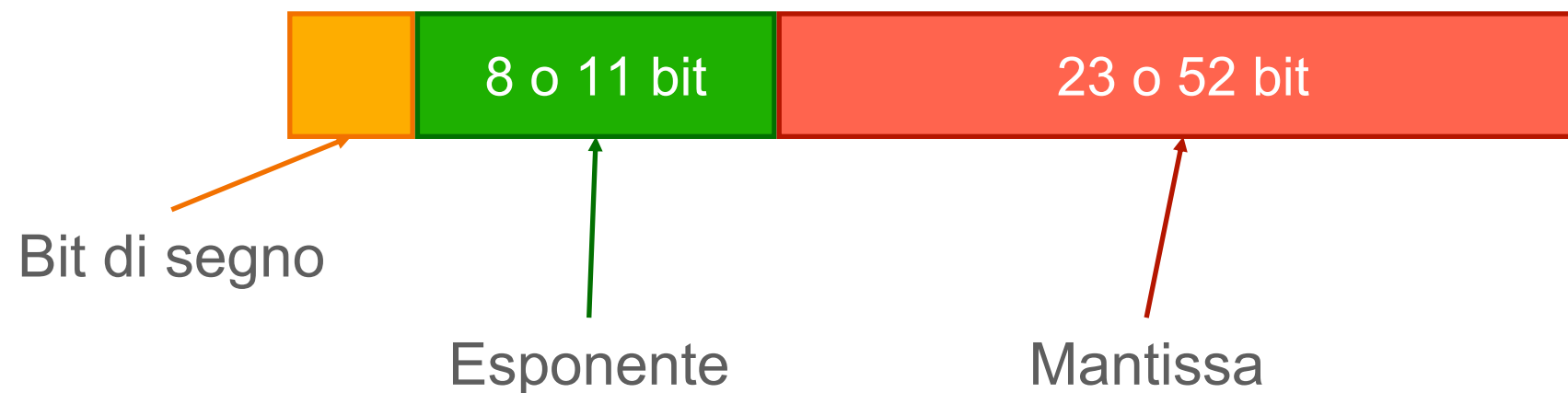
Rappresentazione dei numeri floating point

- Virgola fissa: fisso il numero di bit dopo la virgola
- **Virgola mobile**: simile ad una notazione scientifica:
 - *Esempio*: $328 = 3.28 \times 10^2$
 - 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 $\pm 1. \textit{mantissa} \times 2^{\textit{esponente}}$

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

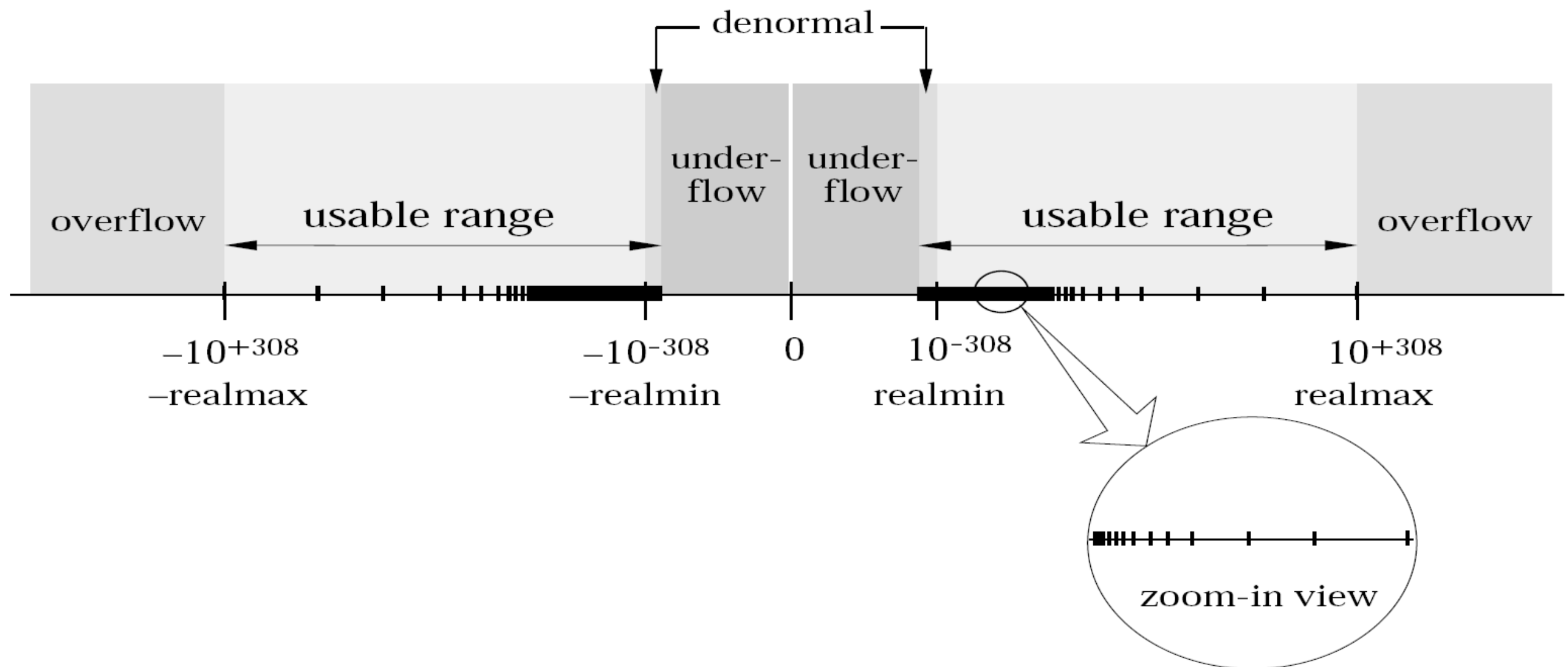
D: su cosa influisce il numero di bit dell'esponente?

Mentre il numero di bit dell'**esponente** ci dice **quanto grande (o piccolo)** può essere il **modulo** dei numeri che rappresentiamo

Float

Visualizzati sulla retta dei reali

Floating Point Number Line



Standard IEEE 754

Esempio

-6.75

Step 1: bit del segno

- Negativo → bit segno = 1

Step 2: convertire in binario

- 110.11 (4+2+0.5+0.25)
- 1.1011×2^2

Step 3: esponente

- eccesso 127
- esponente=129 → bit esponente = 10000001

Step 4: mantissa

- tengo solo parte decimale e aggiungo 0 in coda se necessario
- 1011 → bit mantissa = 101100000000000000000000

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 alla limitata
precisione dei numeri floating point
con modulo grande

Risultato inesatto dovuto
all'approssimazione
nella rappresentazione 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

D: cosa manca?

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

Esercizi

Per prendere la mano

- Convertire in esadecimale i numeri 10101010 e 11100110
- Rappresentare con 8 bit in complemento a 2 i numeri 18 e -5
- Fare la somma delle rappresentazione ottenute al punto precedente
- Convertire in decimale e verificare la correttezza della somma
- Trovare la rappresentazione float precisione singola di 13.1

Algebra di Boole e porte logiche

Algebra di Boole

Una brevissima introduzione

- Invece di avere come valori i numeri interi o i numeri reali abbiamo solo due valori: 0 e 1 (o vero e falso)
- Abbiamo tre principali operazioni
 - AND o prodotto logico, indicato come $\wedge$
 - OR o somma logica, indicato come $\vee$
 - NOT o negazione/complementazione, indicato come $\neg$
- Vediamo la semantica delle diverse operazioni introducendo le porte logiche

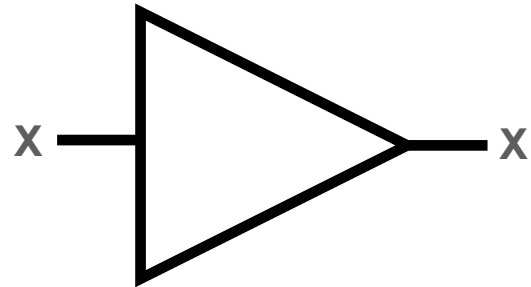
Cosa sono le porte logiche?

I mattoni con cui costruire un computer

- Una porta logica è un'astrazione che facciamo rispetto ai dispositivi fisici
- Possiamo implementare porte logiche usando transistor, relays, condutture, ingranaggi, etc.
- Una porta logica prende in input uno o più valori Booleani (0 o 1) e ritorna uno o più valori Booleani
- Le implementazioni fisiche delle porte logiche hanno molte più complicazioni (e.g., cosa succede se l'input o l'output non sono interpretabili come zero o uno? Quanto tempo ci mette una porta a produrre l'output, etc)

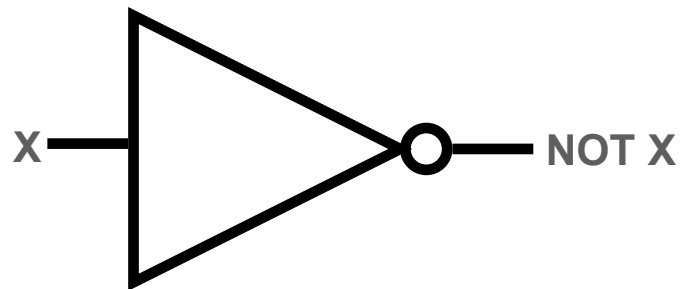
Porte logiche

Buffer e Not



Buffer

Rappresenta la funzione identità



NOT/Inverter

Inverte il valore ricevuto in input

Tabella di verità
per ognuna delle combinazioni di
input è indicato l'output



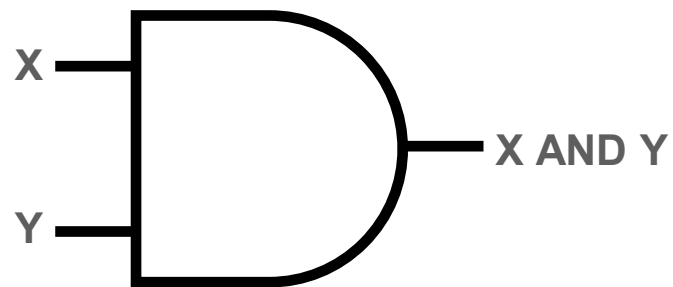
Input	Output
X	X
0	0
1	1

Input	Output
X	NOT X
0	1
1	0

D: quante righe ha una tabella di verità?

Porte logiche

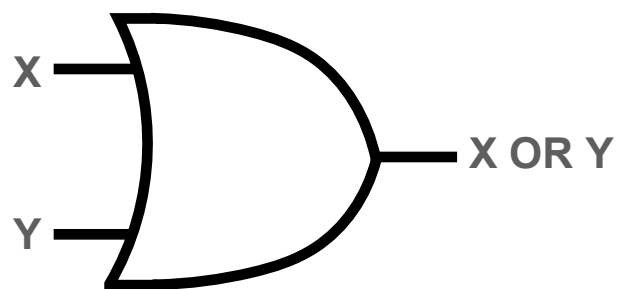
And e Or



AND

Restituisce 1 solo quando entrambi gli input sono 1

Input		Output
X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1



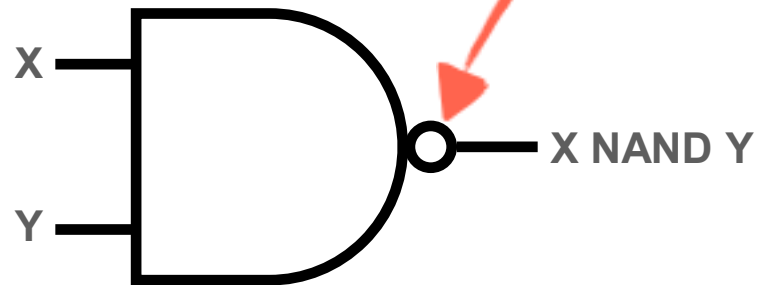
OR

Restituisce 1 quando almeno uno degli input ha valore 1

Input		Output
X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

Porte logiche

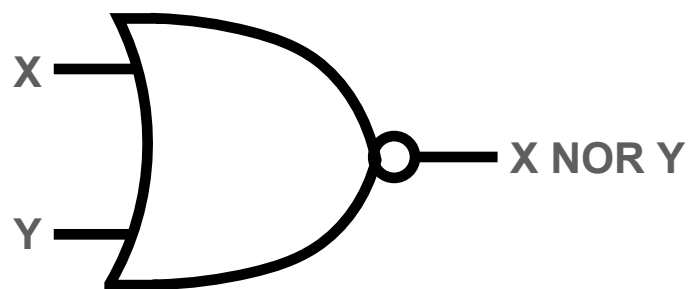
Nand e Nor



NAND

Inverte il valore ritornato da una porta AND.
Ha output 0 solo quando entrambi gli input hanno valore 1

Input		Output
X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0



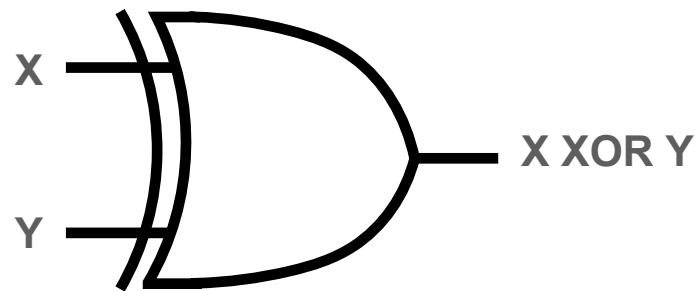
NOR

Inverte il valore ritornato da una porta OR.
Ha output 0 solo quando entrambi gli input hanno valore 1

Input		Output
X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0

Porte logiche

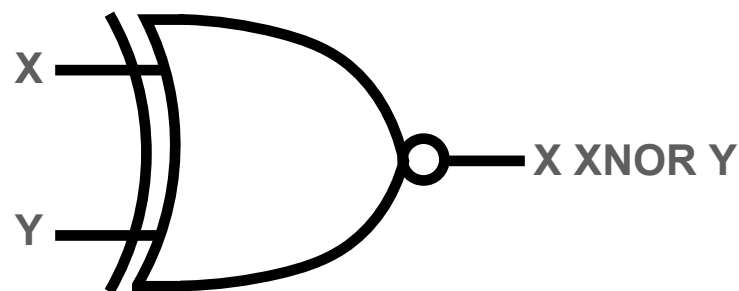
Xor e Xnor



XOR (OR Esclusivo)

L'output è 1 solamente quando *esattamente* uno degli input ha valore 1

Input		Output
X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0



XNOR

L'output è 0 solamente quando *esattamente* uno degli input ha valore 1 (l'inverso della porta XOR)

Input		Output
X	Y	X XNOR Y
0	0	1
0	1	0
1	0	0
1	1	1

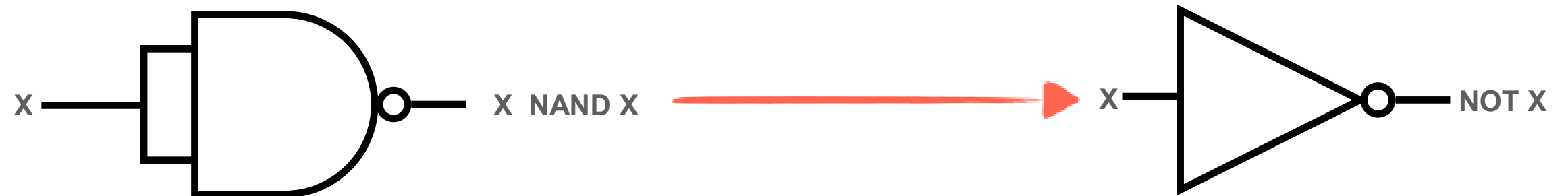
Porte logiche universali

Ci servono davvero tutte le porte logiche?

- Ci si potrebbe chiedere quali porte logiche siano essenziali e quali siano ricostruibili come combinazioni di altre porte logiche
- Sia NAND che NOR sono **universali**, ovvero basta poter implementare un tipo di porta tra NAND o NOR per ricostruire tutte le altre porte logiche
- Mostriamo come ricostruire tutte le porte logiche a partire da NAND

Tutte le porte logiche da NAND

NOT, Buffer e AND



Tutte le porte logiche da NAND

Or e leggi di De Morgan



Questa conversione ci porta a una delle leggi importanti per semplificare insiemi di porte logiche: le **Leggi di De Morgan**

In pratica queste leggi di De Morgan ci dicono che possiamo “portare dentro” la negazione invertendo l’operazione.
Ovvero, AND diventa OR e OR diventa AND:

$$\neg(x \wedge y) = \neg x \vee \neg y$$

$$\neg(x \vee y) = \neg x \wedge \neg y$$

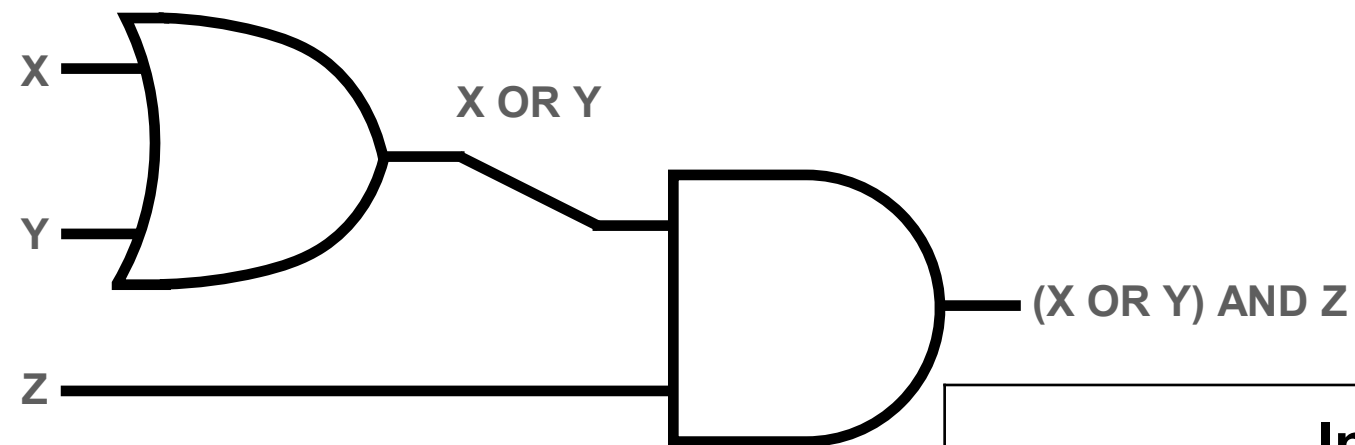
Esercizi

Per prendere la mano

- Abbiamo visto come possiamo usare la porta NAND per costruire ogni altra porta logica...
- ...provate con la porta NOR!
- Abbiamo visto porte con uno o due input. Possiamo combinarle per formare porte con tre input. Per esempio per calcolare
$$(x \vee y) \wedge z$$
 - Graficamente come rappresentiamo quella formula?
 - Quale è la sua tabella di verità?

Combinare porte logiche

Un esempio



Il numero di righe della tabella per n input
è 2^n (ogni nuovo input raddoppia le
configurazioni possibili)



Input			Output	
X	Y	Z	X OR Y	(X OR Y) AND Z
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	0
1	1	1	1	1

Materiale aggiuntivo

Algebra di Boole

proprietà delle algebre di Boole

- $x \wedge 0 = 0$ esistenza di un minimo
- $x \vee 1 = 1$ esistenza di un massimo
- $x \wedge y = y \wedge x$ commutatività dell'AND
- $x \vee y = y \vee x$ commutatività dell'OR
- $(x \wedge y) \wedge z = x \wedge (y \wedge z)$ associatività dell'AND
- $(x \vee y) \vee z = x \vee (y \vee z)$ associatività dell'OR

Algebra di Boole

proprietà delle algebre di Boole

- $x \wedge x = x$ idempotenza dell'AND
- $x \vee x = x$ idempotenza dell'OR
- $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ proprietà distributiva
- $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$ proprietà distributiva
- $x \wedge \neg x = 0$ e $x \vee \neg x = 1$ esistenza di un complemento
- $x \vee (x \wedge y) = x$ e $x \wedge (x \vee y) = x$ assorbimento